Analyzing Divergence in Bisimulation Semantics

Xinxin Liu Tingting Yu Wenhui Zhang

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
{xinxin,yutt,zwh}@ios.ac.cn

Abstract

Some bisimulation based abstract equivalence relations may equate divergent systems with non-divergent ones, examples including weak bisimulation equivalence and branching bisimulation equivalence. Thus extra efforts are needed to analyze divergence for the compared systems. In this paper we propose a new method for analyzing divergence in bisimulation semantics, which relies only on simple observations of individual transitions. This method can be applied for verifying several typical divergence preserving bisimulation equivalences including two well-known ones. As an application case study, we use the proposed method to verify the well-known HSY collision stack by establishing a kind of bisimulation relation between it and a simple specification using atomic block structures. This shows that our method is not over restrictive.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Concurrency, Bisimulation, Co-induction, Induction, Specification, Verification

1. Introduction

Divergence is a phenomenon in computing that can often make big differences in system behaviours. It refers to the existence of an infinite internal computation sequence. For example in Figure 1, if transitions labeled τ represent internal computations, then t_2 is a divergent state because from it there is an infinite internal computation sequence that goes through $t_2, t_3, t_4, t_4 \ldots$, while s_2 is not divergent. Divergence is an important issue in program verification. In verification of concurrent objects, various progress properties are important correctness criteria and closely related to divergent behaviours. As an example, let us look again at the transition graphs in Figure 1, if c labeled transitions represent a call action of a procedure and r_1, r_2 labeled actions represent two different return actions for c, then the typical progress property "after a procedure call, one of the return actions will eventually happen" is true for state s_1 but not for t_1 and u_1, because t_1 and u_1 may diverge after a procedure call and in which case none of r_1 or r_2 will ever happen. Thus s_1 and t_1 should not be considered equivalent in terms of progress property.

The notion of bisimulation, introduced by Park (2) and van Benthem (3) and advocated by Milner (4), is a corner stone in the theory of concurrency. It not only can be used to define equivalences at various abstraction levels but also can provide a powerful verification technique for these equivalences, known as the bisimulation technique: in order to show equality of two states one just needs to construct a binary relation satisfying certain simple conditions. However, divergence may become a problem for bisimulation in that the usual simple treatment of internal actions can only produce equivalences which do not preserve divergence. Weak bisimulation and branching bisimulation are two well-known examples. For readers who know weak bisimulation and branching bisimulation, it is easy to see that the states s_1, t_1 in Figure 1 are equivalent under weak bisimulation equivalence and branching bisimulation equivalence, but as we pointed out above they have different divergent behaviours and thus should not be considered as equal. The aim of this paper is to look for divergence preserving equivalences which are supported by a simple and direct verification method (like the bisimulation technique).

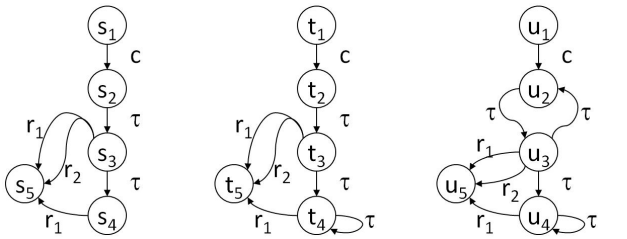

Figure 1. Examples of divergent states

Here we need to be more specific about what we want to achieve in this paper. Firstly, since divergence is a well-known topic and there are divergence preserving variations of weak bisimulation equivalences in the literature (5; 6), it is easy to raise the question: are we working on a solved problem? To answer that, we have to give a closer look at the bisimulation technique. The overall behaviour of a concurrent system, including finite and infinite action sequences and divergence etc., is a complicated matter which is the accumulated effects of some prime observations about individual transitions. The power of bisimulation technique enables us to analyze only the prime observations (individual transitions) while reaching conclusions about the overall behaviours (the sameness of concerned states in terms of finite and infinite action sequences etc.). In this respect, a good bisimulation technique for divergence preserving equivalences should also rely on analyzing only the simple observations of individual transitions to guarantee the property of divergence preservation, which is something that we have not

found in the literature and is what we want to offer through the work of this paper. In the literature, a common approach to obtain divergence preserving equivalences is to include divergence preservation as a specific requirement in the definition. In verification of such equivalences, an oracle is often needed to answer whether a state is divergent or not (which is often non-trivial for infinite-state systems). What we are looking for is a verification method without an oracle. A recent work by Yang *et al.* (19) proposed an original idea of using a divergence preserving version of bisimulation to prove correctness and progress of concurrent objects, and developed a set of methods supported by many significant case studies to illustrate the idea. This makes it more interesting to investigate effective methods for establishing such divergence preserving equivalence.

Thus we are not just proposing more divergence preserving bisimulation equivalences, more importantly we are looking for a verification method for such equivalences. To this end, we use the idea of induction to force divergence preservation in the notion of bisimulation and obtain a new verification method for different divergence preserving equivalence relations. We develop this idea in this paper and obtain the following results:

- We define a divergence preserving weak bisimulation equivalence, named weak bisimilarity with explicit divergence, and show that it can be verified by the new verification method mentioned above which we call inductive weak bisimulation in this case (section 2);
- We show that the method can also be used for two existing divergence preserving equivalences, i.e. branching bisimilarity with explicit divergence (10) and the usual divergence preserving weak bisimilarity (6), which in these cases are called inductive branching bisimulation and generalized inductive weak bisimulation respectively (section 3);
- In presenting the above results, we are using a powerful new notion of bisimulation for dealing with divergence, called complete weak (branching) bisimulation. It combines with inductive weak (branching) bisimulation, and weak (branching) bisimulation with explicit divergence to form a tool set for analyzing divergence in bisimulation from theory to practice.
- As a demonstration of the verification method, we show that the HSY collision stack implementation is correct by establishing an inductive weak bisimulation between the implementation and a stack specification described using atomic block structures (section 4).

We present the details of the results in the following sections, and discuss related works in section 5, and conclude in section 6.

2. Weak Bisimilarity with Explicit Divergence

The aim of this section is to introduce a divergence preserving version of weak bisimulation equivalence called weak bisimilarity with explicit divergence, and to find a verification method for it. We start with some preparation for the abstract setting and some terminologies and notations that we are going to use.

DEFINITION 2.1. *(*Labeled transition systems*). A* labeled transition system *(or LTS) is a triple $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ where:*

- *S is a set of* states*;*
- *A is a set of* actions*;*
- *$\longrightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is the* transition relation*. τ is the* silent action *which is assumed not in A. An element (s, α, t) of $\longrightarrow$, usually written as $s \xrightarrow{\alpha} t$, is called a* transition*;*
- *A finite run of $\mathcal{A}$ is a finite alternating sequence of states and actions: $\rho = s_0\alpha_0 s_1\alpha_1 \ldots s_{n-1}\alpha_{n-1}s_n$ which begins with a state and ends with a state, such that for $0 \leq i < n$, $s_i \xrightarrow{\alpha_i} s_{i+1}$, and ρ is called a finite run from s_0 to s_n;*
- *For $\rho = s_0\alpha_0 s_1\alpha_1 \ldots s_{n-1}\alpha_{n-1}s_n$, define $\mathrm{first}(\rho) = s_0$, $\mathrm{last}(\rho) = s_n$, $\mathrm{Act}(\rho) = \alpha_0\alpha_1 \ldots \alpha_{n-1}$, and $\mathrm{length}(\rho) = n$ which is the number of the transitions in σ;*
- *An infinite run of $\mathcal{A}$ is an infinite, alternating sequence of states and actions: $\rho = s_0\alpha_0 s_1\alpha_1 \ldots$ which begins with a state, such that for all $i \geq 0$, $s_i \xrightarrow{\alpha_i} s_{i+1}$.*
- *If $\rho = s_0\alpha_0 s_1\alpha_1 \ldots$ is an infinite run, then $\mathrm{first}(\rho) = s_0$, $\mathrm{Act}(\rho) = \alpha_0\alpha_1 \ldots$;*
- *A (finite or infinite) τ-run of $\mathcal{A}$ is a (finite or infinite) run of $\mathcal{A}$ in which all actions are τ's;*
- *A state s is called divergent, written $s \Uparrow$, if there is an infinite τ-run from s.* □

We will often use concatenation of runs, states, and actions to form new runs in the obvious way.

For a finite sequence of actions $l \in (A \cup \{\tau\})^*$, let $\widehat{l} \in A^*$ be the sequence obtained by deleting all τ's from l.

We use standard notations for multi-step τ transitions: write $s \Longrightarrow s'$ if there is a finite τ-run from s to s'; write $s \stackrel{\alpha}{\Longrightarrow} s'$ if there exist t, t' such that $s \Longrightarrow t, t \xrightarrow{\alpha} t', t' \Longrightarrow s'$. Note the important difference between $s \Longrightarrow s'$ and $s \stackrel{\tau}{\Longrightarrow} s'$: the former means that from s to s' there is a finite τ-run, while the latter means that from s to s' there is a finite τ-run with non-zero length. Thus $s \Longrightarrow s$ holds for all $s \in S$, while $s \stackrel{\tau}{\Longrightarrow} s$ holds only when s is on a τ-loop consists of one or more τ-transitions. Also for $l \in (A \cup \{\tau\})^*$ we will write $s \stackrel{\widehat{l}}{\Longrightarrow} s'$ if there is a finite run ρ from s to s' with $Act(\rho) = l$. Note that $s \stackrel{\epsilon}{\Longrightarrow} s'$ means exactly $s \Longrightarrow s'$, where ϵ is the empty string.

With these preparations, we have two subsections for the main contents of this section. In the first subsection we define the equivalence relation that we are going to study, and in the second subsection we describe a verification method for it.

2.1 Definition of Weak Bisimilarity with Explicit Divergence

Before introducing the new equivalence, we briefly review the theory of weak bisimulation.

DEFINITION 2.2. *(*Weak bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ be an LTS. A binary relation $R \subseteq S \times S$ is called a* weak bisimulation *if for all $(s, t) \in R$ the following conditions hold:*

1. *whenever $s \xrightarrow{\alpha} s'$, then there exists t' such that $t \stackrel{\widehat{\alpha}}{\Longrightarrow} t'$ and $(s', t') \in R$;*
2. *whenever $t \xrightarrow{\alpha} t'$, then there exists s' such that $s \stackrel{\widehat{\alpha}}{\Longrightarrow} s'$ and $(s', t') \in R$.*

We write $s \approx t$ if there is a weak bisimulation R such that $(s, t) \in R$. □

This is a typical co-inductive definition. With this definition, it is routine to prove the following results.

LEMMA 2.3. *If $\{R_i \mid i \in I\}$ is a set of weak bisimulations, then $\bigcup\{R_i \mid i \in I\}$ is a weak bisimulation.* □

For two binary relations R_1, R_2, we write $R_1 \cdot R_2$ for their composition, i.e.

$$R_1 \cdot R_2 = \{(s, t) \mid \exists u.(s, u) \in R_1, (u, t) \in R_2\}.$$

LEMMA 2.4. *If R_1, R_2 are weak bisimulations, then $R_1 \cdot R_2$ is also a weak bisimulation.* □

THEOREM 2.5. *$\approx$ is the largest weak bisimulation, and it is an equivalence relation.* □

All these are standard results. It follows from Lemma 2.3 that $\approx$ is the largest weak bisimulation (this is in fact an application of Knaster-Tarski fixed-point theorm), and follows from Lemma 2.4 that $\approx$ is an equivalence relation. With this theorem, $\approx$ is usually called weak bisimulation equivalence (or weak bisimilarity) in the literature.

Definition 2.2 and Theorem 2.5 literally form the most important part of the theory of weak bisimulation, in which the equivalence relation $\approx$ not only is defined but also comes with a verification technique such that in order to show $s \approx t$ for two states s, t, one just needs to find a binary relation R such that $(s,t) \in R$ and R is a weak bisimulation. For example, for states s_1, t_1 in Fig 1. $R = \{(s_1,t_1),(s_2,t_2),(s_3,t_3),(s_4,t_4),(s_5,t_5)\}$ is a weak bisimulation, thus $s_1 \approx t_1$, as we have mentioned in the introduction.

For an equivalence relation $\equiv$, we say that a state s is divergent with respect to $\equiv$, written $s \Uparrow_{\equiv}$, if from s there is an infinite τ-run which only goes through states that are $\equiv$-equivalent to s. A simple example is that $t_4 \Uparrow_{\approx}$ for t_4 in Fig 1.

Now it is not difficult to see that the cause for $\approx$ failing to preserve divergence is that there exists some state s such that s diverges in the same equivalence class, i.e. $s \Uparrow_{\approx}$. The reason is that, because of the means of abstraction applied in weak bisimulation (by ignoring τ-moves), the infinite τ-run for $s \Uparrow_{\approx}$ is completely ignored. Look at Fig 1. again, the weak bisimulation R above contains (s_4,t_4), thus $s_4 \approx t_4$ but t_4 diverges while s_4 not. To overcome this, a simple idea is to make this situation explicit. This is the idea used in the definition of branching bisimulation with explicit divergence (7). Thus we arrive at the following definition.

DEFINITION 2.6. *(*Weak bisimulation with explicit divergence*). Let $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ be an LTS, $\equiv$ be an equivalence on S. $\equiv$ is called a* weak bisimulation with explicit divergence *if $\equiv$ is a weak bisimulation and moreover whenever $s \equiv t$ then $s \Uparrow_{\equiv}$ if and only if $t \Uparrow_{\equiv}$.*

We write $s \approx^{\Delta} t$ if there is a weak bisimulation with explicit divergence $\equiv$ such that $s \equiv t$. □

The above looks like an innocent co-inductive definition for $\approx^{\Delta}$. However, at least for the moment, it only defines a binary relation $\approx^{\Delta}$. To justify that it is the wanted weak bisimilarity with explicit divergence, we need to prove that $\approx^{\Delta}$ is an equivalence, is a weak bisimulation, and is the largest weak bisimulation with explicit divergence.

It turns out that, unlike the case for $\approx$ (Theorem 2.5), proving the above results directly is far from an easy task. This is caused by a big difference between Definition 2.2 and Definition 2.6: the former has an underlying monotonic function, while the latter has not, which makes Knaster-Tarski fixed-point theorem not applicable anymore. Our approach to get around this difficulty is to construct another equivalence called complete weak bisimilarity, and prove that it is the largest weak bisimulation with explicit divergence. Next we first describe this complete weak bisimilarity.

DEFINITION 2.7. *(Divergence set). Let $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ be an LTS. A nonempty set $D \subseteq S$ is called a divergence set if for all $s \in D$ there exists $s' \in D$ such that $s \stackrel{\tau}{\Longrightarrow} s'$.*

Let $s \in S$, we write $s \Longrightarrow_{\omega} D$ if D is a divergence set and there exists $s' \in D$ such that $s \Longrightarrow s'$. In this case we also call D a divergence set of s. □

In words, $s \Longrightarrow_{\omega} D$ holds if and only if there is an infinite τ-run from s which passes through D infinitely often.

The purpose of introducing divergence set is to give an alternative description of divergence preservation in the context of weak bisimulation. From this definition, it is easy to see that a state has a divergent run if and only if it has a divergence set.

The following definition strengthens that of weak bisimulation by imposing conditions concerning divergence set.

DEFINITION 2.8. *(*Complete weak bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ be an LTS. A binary relation $R \subseteq S \times S$ is called a* complete weak bisimulation *if it is a weak bisimulation and moreover for all $(s,t) \in R$ the following hold:*

1. *whenever $s \Longrightarrow_{\omega} D$, then $t \Longrightarrow_{\omega} E$ for some divergence set E such that for all $t' \in E$ there exists $s' \in D$ with $(s',t') \in R$;*
2. *whenever $t \Longrightarrow_{\omega} E$, then $s \Longrightarrow_{\omega} D$ for some divergence set D such that for all $s' \in D$ there exists $t' \in E$ with $(s',t') \in R$.*

We write $s \approx_c t$ if there is a complete weak bisimulation R such that $(s,t) \in R$. □

The advantage of this definition is that, like Definition 2.2, it has an underlying monotonic function which guarantees good properties as stated in the following Lemma 2.9 and Theorem 2.11, something not enjoyed by Definition 2.6.

LEMMA 2.9. *If $\{R_i \mid i \in I\}$ is a set of complete weak bisimulations, then $\bigcup\{R_i \mid i \in I\}$ is a complete weak bisimulation.* □

LEMMA 2.10. *If R_1, R_2 are complete weak bisimulations, then $R_1 \cdot R_2$ is also a complete weak bisimulation.* □

THEOREM 2.11. *$\approx_c$ is the largest complete weak bisimulation, and it is an equivalence relation.* □

Like similar results for weak bisimulation, all these can be proved directly from the definitions. With this theorem, we call $\approx_c$ complete weak bisimulation equivalence, or complete weak bisimilarity.

The following theorem is straightforward.

THEOREM 2.12. *Every complete weak bisimulation is divergence preserving. That is to say, if R is a complete weak bisimulation and $(s,t) \in R$, then $s \Uparrow$ if and only if $t \Uparrow$.* □

LEMMA 2.13. *If $s \Longrightarrow t, t \Longrightarrow u$, and $s \approx_c u$, then $s \approx_c t$.*

Proof: Let $R = \{(s,t) \mid s \approx_c \cdot \Longrightarrow t, t \Longrightarrow \cdot \approx_c s\}$. By the fact that $\approx_c$ is a complete weak bisimulation, it is not difficult to prove that the following hold for all $(s,t) \in R$:

1. whenever $s \stackrel{\alpha}{\longrightarrow} s'$ then $t \stackrel{\widehat{\alpha}}{\Longrightarrow} t'$ for some t' such that $s' \approx_c t'$;
2. whenever $t \stackrel{\alpha}{\longrightarrow} t'$ then $s \stackrel{\widehat{\alpha}}{\Longrightarrow} s'$ for some s' such that $s' \approx_c t'$;
3. whenever $s \Longrightarrow_{\omega} D$ then $t \Longrightarrow_{\omega} E$ for some divergence set E such that for all $t' \in E$ there exists $s' \in D$ with $s' \approx_c t'$;
4. whenever $t \Longrightarrow_{\omega} E$ then $s \Longrightarrow_{\omega} D$ for some divergence set D such that for all $s' \in D$ there exists $t' \in D$ with $s' \approx_c t'$.

Note that $\approx_c \subseteq R$, thus according to 1), 2) above R is a weak bisimulation, and moreover according to 3), 4) above R is a complete weak bisimulation. □

This lemma, which is also called computation lemma, says that $\approx_c$ has the so-called stuttering property, a well-known property for branching bisimulation equivalence. Surprisingly perhaps, this property is also true for $\approx$, which we have not seen mentioned in the literature. Stuttering property of $\approx_c$ is very instrumental in establishing that $\approx_c$ is a weak bisimulation with explicit divergence.

THEOREM 2.14. *$\approx_c$ is a weak bisimulation with explicit divergence.*

Proof: By Theorem 2.11 $\approx_c$ is an equivalence relation and is a weak bisimulation. What left to show is that whenever $s \approx_c t$ and $s \Uparrow_{\approx_c}$ then $t \Uparrow_{\approx_c}$. Suppose $s \approx_c t$ and $s \Uparrow_{\approx_c}$, then there is an infinite τ-run $\rho = s\tau s_1 \tau \ldots$ such that $s_i \approx_c s$ for $i = 1, \ldots$. Let

$D = \{s_i \mid i = 1, 2, \ldots\}$, then clearly $s \Longrightarrow_\omega D$, and since $\approx_c$ is a complete weak bisimulation, then $t \Longrightarrow_\omega E$ for some divergence set E such that $t' \approx_c s$ for all $t' \in E$. Thus there is an infinite τ-run σ from t which passes through E infinitely often. Now by Lemma 2.13 it is easy to see that all states on σ are $\approx_c$-equivalent to t, thus $t \Uparrow \approx_c$. □

With this theorem, clearly $\approx_c \subseteq \approx^\Delta$. We will postpone the proof of $\approx^\Delta \subseteq \approx_c$ to the end of the next subsection where it would become easier to do. After that we have $\approx_c = \approx^\Delta$. Then by the fact that $\approx_c$ is an equivalence and is a weak bisimulation with explicit divergence, so is $\approx^\Delta$. Thus we would have completed the justification of Definition 2.6.

2.2 Verification via Inductive Weak Bisimulation

Remember that, besides defining $\approx^\Delta$, we have another task that is to find a verification method for $\approx^\Delta$. One may think that $\approx_c$ and $\approx^\Delta$ are the same relation after all, then complete weak bisimulation could be used as a verification technique for $\approx^\Delta$. However, Definition 2.8 requires checking all divergence sets of a state, which clearly is infeasible because there could well be infinitely many divergence sets of a state. To solve the problem, we introduce a new bisimulation definition.

DEFINITION 2.15. *(*Inductive weak bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow \rangle$ be an LTS. For a binary relation $R \subseteq S \times S$, define the* inductive τ-matching closure *of R, written $\mathcal{I}(R)$, as the smallest relation satisfying the following closure property: $\mathcal{I}(R)$ contains all $(s, t) \in S \times S$ such that*

1. *whenever $s \xrightarrow{\tau} s'$, then either $t \stackrel{\tau}{\Longrightarrow} t'$ for some $t' \in S$ with $(s', t') \in R$, or $(s', t) \in R \cap \mathcal{I}(R)$;*
2. *whenever $t \xrightarrow{\tau} t'$, then either $s \stackrel{\tau}{\Longrightarrow} s'$ for some $s' \in S$ with $(s', t') \in R$, or $(s, t') \in R \cap \mathcal{I}(R)$.*

A relation R is called an inductive weak bisimulation *if R is a weak bisimulation and moreover $R \subseteq \mathcal{I}(R)$. We write $s \approx_i t$ if (s, t) is contained in an inductive weak bisimulation.* □

Intuitively, the condition $R \subseteq \mathcal{I}(R)$ for an inductive weak bisimulation R would enforce that for all $(s, t) \in R$, if there is an infinite τ-run from s, then there must also be an infinite τ-run from t.

Before further development of the theory, let us pause to see an example of inductive weak bisimulation. If the transitions for

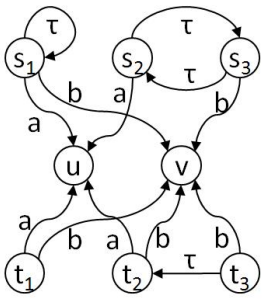

Figure 2. How to establish that $t_3 \approx_i t_1$?

$s_1, s_2, s_3, t_1, t_2, t_3$, and u, v are as shown in Fig 2., then $R = \{(t_3, t_1), (t_2, t_1), (u, u), (v, v)\}$ is an inductive weak bisimulation. To see that, first note that R is a weak bisimulation. Next we show $R \subseteq \mathcal{I}(R)$. Clearly $(u, u), (v, v), (t_2, t_1)$ are elements of $\mathcal{I}(R)$ directly. Only after that, we can show that (t_3, t_1) is also an element of $\mathcal{I}(R)$, because this will rely on the fact that (t_2, t_1) is already in $\mathcal{I}(R)$, since the transition $t_3 \xrightarrow{\tau} t_2$ cannot be matched by any $\stackrel{\tau}{\Longrightarrow}$ transition from t_1. On the other hand (t_1, s_1) can never be an element of an inductive weak bisimulation R. Because in order to match the transition $s_1 \xrightarrow{\tau} s_1$ for (t_1, s_1) to be included in any $\mathcal{I}(R)$, the only way is that (t_1, s_1) must be in $\mathcal{I}(R)$ first, which is impossible for inductively defined $\mathcal{I}(R)$.

LEMMA 2.16. *If $R_1 \subseteq R_2$ then $\mathcal{I}(R_1) \subseteq \mathcal{I}(R_2)$.*

Proof: Easy to prove by induction on the definition of $\mathcal{I}(R_1)$ in Definition 2.15. □

This lemma guarantees that, like Definition 2.2 and Definition 2.6, Definition 2.15 also has an underlying monotonic function manifested by the following good properties.

LEMMA 2.17. *If $\{R_i \mid i \in I\}$ is a set of inductive weak bisimulation, then $\bigcup\{R_i \mid i \in I\}$ is an inductive weak bisimulation.*

Proof: Follow from Lemma 2.16. □

THEOREM 2.18. *$\approx_i$ is the largest inductive weak bisimulation.* □

Now Definition 2.15 does not refer to any infinite sequences, thus it should be easier to check in application. Our remaining task in this section, is to show that $\approx_i$, $\approx_c$ and $\approx^\Delta$ are all the same relation, so that we can use Definition 2.15 for verification of $\approx^\Delta$. For that, we will establish that every weak bisimulation with explicit divergence is an inductive weak bisimulation, and that every inductive weak bisimulation is a complete weak bisimulation. These results combined with Theorem 2.14 will complete the task of showing that the three relations $\approx_i, \approx_c$, and $\approx^\Delta$ coincide.

LEMMA 2.19. *Let R be a weak bisimulation. Then for all $(s, t) \in \mathcal{I}(R)$ the following hold: if $(s, t) \in R$, and $s \Longrightarrow_\omega D$ for some divergence set D, then there exist s', t' such that $s' \in D$, $t \stackrel{\tau}{\Longrightarrow} t'$ and $(s', t') \in R$.*

Proof: Let $(s, t) \in \mathcal{I}(R)$, we prove the lemma by induction on the definition of $\mathcal{I}(R)$.

So suppose $(s, t) \in R$, and let $s^\dagger \in D$ with $s \stackrel{\tau}{\Longrightarrow} s^\dagger$, obviously such $s^\dagger$ must exist in D since $s \Longrightarrow_\omega D$. Moreover, let $s^\ddagger$ be a state such that $s \xrightarrow{\tau} s^\ddagger, s^\ddagger \Longrightarrow s^\dagger$, clearly such $s^\ddagger$ exists. Since $(s, t) \in \mathcal{I}(R)$, for the transition $s \xrightarrow{\tau} s^\ddagger$, there are two cases, i.e. there is $t^\ddagger$ such that $t \stackrel{\tau}{\Longrightarrow} t^\ddagger$ and $(s^\ddagger, t^\ddagger) \in R$, or $(s^\ddagger, t) \in R \cap \mathcal{I}(R)$.

In the first case, for $(s^\ddagger, t^\ddagger) \in R$ and $s^\ddagger \Longrightarrow s^\dagger$, since R is a weak bisimulation there exists t' such that $t^\ddagger \Longrightarrow t'$ and $(s^\dagger, t') \in R$. Let s' be $s^\ddagger$, then we found the required s', t'.

In the second case, since $(s^\ddagger, t) \in \mathcal{I}(R)$, $(s^\ddagger, t) \in R$, and $s^\ddagger \Longrightarrow_\omega D$, by the induction hypothesis there is s', t' such that $s' \in D, t \stackrel{\tau}{\Longrightarrow} t', (s', t') \in R$. □

THEOREM 2.20. *If R is an inductive weak bisimulation, then R is a complete weak bisimulation.*

Proof: Let R be a weak bisimulation and $R \subseteq \mathcal{I}(R)$. Suppose $(s, t) \in R$, and $s \Longrightarrow_\omega D$ for some divergence set D. Then by repeatedly applying Lemma 2.19, we can obtain an infinite state sequence $t_1 t_2 \ldots$ such that for all i there exists $s' \in D$ with $(s', t_i) \in R$. Clearly $E = \{t_i \mid i = 1, 2, \ldots\}$ is a divergence set such that $t \Longrightarrow_\omega E$ and for all $t' \in E$ there exists $s' \in D$ with $(s', t') \in R$. The other half can be proved in the same way.

Then it is clear that every inductive weak bisimulation is a complete weak bisimulation. □

THEOREM 2.21. *Let $\equiv$ be a weak bisimulation with explicit divergence. Then $\equiv \subseteq \mathcal{I}(\equiv)$.*

Particularly, every weak bisimulation with explicit divergence is an inductive weak bisimulation.

Proof: Let $\equiv$ be a weak bisimulation with explicit divergence. To prove $\equiv \subseteq \mathcal{I}(\equiv)$, we define a binary relation $\succ \subseteq S \times S$ such that $s \succ s'$ if and only if the following hold:

1. $s \Downarrow_{\equiv}$ (that is $s \Uparrow_{\equiv}$ does not hold), and
2. $s \equiv s'$, and
3. $s \xrightarrow{\tau} s'$.

Then clearly $\succ$ is well founded, i.e. there is no infinite descending chain $s \succ s_1 \ldots \succ s_i \ldots$. Otherwise $\sigma = s\tau s_1 \ldots \tau s_i \ldots$ is an infinite τ-run from s with $s_i \equiv s$ for all $i = 1, 2, \ldots$, a contradiction to $s \Downarrow_{\equiv}$.

Now suppose $s \equiv t$, we will show $(s,t) \in \mathcal{I}(\equiv)$ by well-founded induction on $\succ$. So let $s \xrightarrow{\tau} s'$, we will show that either A) there exists $t' \in S$ such that $t \stackrel{\tau}{\Longrightarrow} t'$ and $s' \equiv t'$, or B) $(s', t) \in \mathcal{I}(\equiv)$ and $s' \equiv t$ as follows. If $s' \not\equiv s$, then by the condition that $\equiv$ is a weak bisimulation, there is t' such that $t \Longrightarrow t'$ and $s' \equiv t'$, thus $t' \not\equiv t$ and $t \stackrel{\tau}{\Longrightarrow} t'$, A) holds. If $s' \equiv s$, we discuss by two subcases. The first is $s \Uparrow_{\equiv}$. In this case since $\equiv$ is a weak bisimulation with explicit divergence, $t \Uparrow_{\equiv}$, then obviously there is t' such that $t \xrightarrow{\tau} t'$ and $t' \equiv t$, thus $s' \equiv t'$ and A) holds. The second subcase is $s \Downarrow_{\equiv}$, in this case $s \succ s'$, then by the induction hypothesis $(s', t) \in \mathcal{I}(\equiv)$ and $s' \equiv t$, and B) holds. Similarly we can prove that if $t \xrightarrow{\tau} t'$ then either there is s' such that $s \stackrel{\tau}{\Longrightarrow} s'$ and $s' \equiv t'$ or $(s, t') \in \mathcal{I}(\equiv)$ and $s \equiv t'$. Then by the definition of $\mathcal{I}(\equiv)$ we have $(s,t) \in \mathcal{I}(\equiv)$. □

The following theorem summarizes the main results of this section.

THEOREM 2.22. *$\approx_c$ and $\approx_i$ and $\approx^{\Delta}$ all define the same divergence preserving weak bisimulation equivalence.*

Proof: By Theorem 2.14 $\approx_c \subseteq \approx^{\Delta}$. Now $\approx^{\Delta} \subseteq \approx_i$ follows from the fact that every weak bisimulation with explicit divergence is an inductive weak bisimulation (Theorem 2.21). $\approx_i \subseteq \approx_c$ follows from the fact that every inductive weak bisimulation is a complete weak bisimulation (Theorem 2.20).

Then by Theorem 2.12, $\approx_c$ is a divergence preserving weak bisimulation equivalence, so they are all divergence preserving weak bisimulation equivalences. □

Thus we have completed what we wanted to do in this section: we introduced an equivalence relation $\approx^{\Delta}$, which can be verified by using the definition of $\approx_i$, that is in order to establish $s \approx^{\Delta} t$, what we need to do is to find a binary relation R that is an inductive weak bisimulation such that $(s,t) \in R$. In fact the work of this section is more than showing equality of the three relations. It shows that the three notions of bisimulation can very well work together. Weak bisimulation with explicit divergence is intuitively appealing. Inductive weak bisimulation is practically useful. Complete weak bisimulation is theoretically powerful.

3. Other Divergence Preserving Equivalences

In this section, we will show that the idea of inductive weak bisimulation together with complete weak bisimulation can be adapted for providing verification method for other divergence preserving bisimulation equivalences.

3.1 Branching Bisimilarity with Explicit Divergence

The content for branching bisimilarity with explicit divergence is completely parallel to that for weak bisimilarity with explicit divergence. So the presentation in this subsection is very brief, with similar proofs omitted.

We also start with a brief review of branching bisimulation.

DEFINITION 3.1. *(*Branching bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow \rangle$ be an LTS. A binary relation $R \subseteq S \times S$ is called a* branching bisimulation *if for all $(s,t) \in R$ the following conditions hold:*

1. *whenever $s \xrightarrow{\alpha} s'$, then either there exist t', t'' such that $t \Longrightarrow t', t' \xrightarrow{\alpha} t''$ with $(s,t'), (s',t'') \in R$, or $\alpha = \tau$ and $(s',t) \in R$;*
2. *whenever $t \xrightarrow{\alpha} t'$, then either there exist s', s'' such that $s \Longrightarrow s', s' \xrightarrow{\alpha} s''$ with $(s,t'), (s',t'') \in R$, or $\alpha = \tau$ and $(s,t') \in R$.*

We write $s \approx_b t$ if there is a branching bisimulation R such that $(s,t) \in R$. □

Thus defined, it is well known that $\approx_b$ is an equivalence relation (7), here we call branching bisimilarity, and it is the largest branching bisimulation.

For states in Fig 1. $R = \{(s_2,t_2), (s_3,t_3), (s_4,t_4), (s_5,t_5)\}$ is a branching bisimulation, thus $s_3 \approx_b t_3$, as we have mentioned in the introduction. Thus $\approx_b$ is not divergence preserving because t_3 diverges while s_3 does not.

It is obvious from the definitions that every branching bisimulation is a weak bisimulation.

The following definition introduces a divergence preserving version of branching bisimulation.

DEFINITION 3.2. *(*Branching bisimulation with explicit divergence*). Let $\mathcal{A} = \langle S, A, \longrightarrow \rangle$ be an LTS, $\equiv$ be an equivalence on S. $\equiv$ is called a* branching bisimulation with explicit divergence *if $\equiv$ is a branching bisimulation and moreover whenever $s \equiv t$ then $s \Uparrow_{\equiv}$ if and only if $t \Uparrow_{\equiv}$.*

We write $s \approx_b^{\Delta} t$ if there is a branching bisimulation with explicit divergence $\equiv$ such that $s \equiv t$. □

The notion of branching bisimulation with explicit divergence was first introduced in (7) where colored trace characterization was used instead of branching bisimulation. Like $\approx^{\Delta}$, the relation $\approx_b^{\Delta}$ as defined by Definition 3.2 also needs justification. We need to prove that it is an equivalence, and is the largest branching bisimulation with explicit divergence. Although the definition in (7) has been shown to define an equivalence relation (10; 11), here we give a different justification parallel to the development of weak bisimulation with explicit divergence.

DEFINITION 3.3. *(*Complete branching bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow \rangle$ be an LTS. A binary relation $R \subseteq S \times S$ is called a* complete branching bisimulation *if it is a branching bisimulation and moreover for all $(s,t) \in R$ the following hold:*

1. *whenever $s \Longrightarrow_{\omega} D$ for some divergence set D, then $t \Longrightarrow_{\omega} E$ for some divergence set E such that for all $t' \in E$ there exists $s' \in D$ with $(s', t') \in E$;*
2. *whenever $t \Longrightarrow_{\omega} E$ for some divergence set E, then $s \Longrightarrow_{\omega} D$ for some divergence set D such that for all $s' \in D$ there exists $t' \in E$ with $(s', t') \in E$.*

We write $s \approx_{cb} t$ if there is a complete branching bisimulation R such that $(s,t) \in R$. □

THEOREM 3.4. *$\approx_{cb}$ is the largest complete branching bisimulation, and it is an equivalence relation.* □

LEMMA 3.5. *If $s \Longrightarrow t, t \Longrightarrow u$, and $s \approx_{cb} u$, then $s \approx_{cb} t$.* □

The following theorem confirms that $\approx_{cb}$ is a branching bisimulation with explicit divergence.

THEOREM 3.6. *If $s \approx_{cb} t$ and $s \Uparrow_{\approx_{cb}}$, then $t \Uparrow_{\approx_{cb}}$.* □

With this theorem we know that $\approx_{cb} \subseteq \approx_b^{\Delta}$.

Like the case for $\approx^{\Delta}$, we now turn to define inductive branching bisimulation which will be used to prove $\approx_b^{\Delta} \subseteq \approx_{cb}$ later.

DEFINITION 3.7. *(*Inductive branching bisimulation*). Let $\mathcal{A} = \langle S, A, \longrightarrow\rangle$ be an LTS. A relation R is called an* inductive branching bisimulation *if R is a branching bisimulation and moreover $R \subseteq \mathcal{I}(R)$. We write $s \approx_{ib} t$ if (s,t) is contained in an inductive branching bisimulation.* □

LEMMA 3.8. *If $\{R_i \mid i \in I\}$ is a set of inductive branching bisimulation, then $\bigcup\{R_i \mid i \in I\}$ is an inductive branching bisimulation.* □

THEOREM 3.9. *$\approx_{ib}$ is the largest inductive branching bisimulation.* □

THEOREM 3.10. *Every inductive branching bisimulation is a complete branching bisimulation.*

Proof: Let R be an inductive branching bisimulation. Then R is an inductive weak bisimulation. Then by Theorem 2.20, for all $(s,t) \in R$ it holds that whenever $s \Longrightarrow_{\omega} D$ for some divergence set D, there is a divergence set E such that $t \Longrightarrow_{\omega} E$ and whenever $t' \in E$ there exists $s' \in D$ with $(s',t') \in R$, and similar situation for any E with $t \Longrightarrow_{\omega} E$. Thus R is a complete branching bisimulation since R is a branching bisimulation. □

THEOREM 3.11. *Every branching bisimulation with explicit divergence is an inductive branching bisimulation.*

Proof: Let R be a branching bisimulation with explicit divergence. Then R is a weak bisimulation, and by Theorem 2.21 $R \subseteq \mathcal{I}(R)$, so R is an inductive branching bisimulation. □

THEOREM 3.12. *$\approx_{cb}, \approx_{ib}, \approx_b^{\Delta}$ all define the same equivalence.*

Proof: $\approx_b^{\Delta} \subseteq \approx_{ib}$ by Theorem 3.11. $\approx_{ib} \subseteq \approx_{cb}$ by Theorem 3.10. $\approx_{cb} \subseteq \approx_b^{\Delta}$ by Theorem 3.6. □

Thus, inductive branching bisimulation can be used as a verification method for establishing branching bisimilarity with explicit divergence.

3.2 Divergence Preserving Weak Bisimulation

In this subsection we will find a verification method for the usual divergence preserving weak bisimularity studied in (6). We start with the definition.

DEFINITION 3.13. *Let $\mathcal{A} = \langle S, A \longrightarrow\rangle$ be an LTS. A binary relation R is called a divergence preserving weak bisimulation if it is a weak bisimulation and moreover for all $(s,t) \in R$ it holds that $s \Uparrow$ if and only if $t \Uparrow$.*

We write $s \approx^{\Uparrow} t$ if there is a divergence preserving weak bisimulation R such that $(s,t) \in R$. □

This is also a definition with an underlying monotonic function, and it is routine to prove that $\approx^{\Uparrow}$ is an equivalence relation, and that it is a divergence preserving weak bisimulation (in fact the biggest one).

Here is a good point to compare distinguishing strengths of the equivalences studied so far.

THEOREM 3.14. *$\approx_b^{\Delta} \subseteq \approx^{\Delta} \subseteq \approx^{\Uparrow}$, and the inclusions are strict.*

Proof: The first inclusion simply follows from the fact that every branching bisimulation is a weak bisimulation. Clearly it is strict because branching bisimulation is strictly stronger then weak bisimulation. The second inclusion follows from Theorem 2.12. To see that it is strict, note that in Fig 1., $t_3 \approx^{\Uparrow} u_3$, while $t_3 \not\approx^{\Delta} u_3$, because $u_3 \Uparrow_{\approx^{\Delta}}$ only holds for u_3. □

Let s be a state s, $trace(s)$ and $\omega\text{-}trace(s)$ be the sets of finite and infinite traces from s, i.e.

$$trace(s) = \{Act(\sigma) \mid \sigma \text{ is a finite run from s}\},$$
$$\omega\text{-}trace(s) = \{Act(\sigma) \mid \sigma \text{ is an infinite run from s}\}.$$

For two states s,t, we say that they are trace equivalent, written $trace(s) \approx trace(t)$, if whenever $l \in trace(s)$ then there is $m \in trace(t)$ such that $\widehat{l} = \widehat{m}$, and vise-versa. We say that they are ω-trace equivalent, written $\omega\text{-}trace(s) \approx \omega\text{-}trace(t)$, if whenever $l \in \omega\text{-}trace(s)$ then there is $m \in \omega\text{-}trace(t)$ such that $\widehat{l} = \widehat{m}$, and vise-versa.

It is obvious that $\approx$ preserves trace equivalence. $\approx^{\Uparrow}$ also preserves ω-trace equivalence.

THEOREM 3.15. *If $s \approx^{\Uparrow} t$, then s and t are ω-trace equivalent as well as trace equivalent.* □

$\approx^{\Uparrow}$ is not feasible to verify by using Definition 3.13, because the definition needs to ask an oracle to know if $s \Uparrow$. Thus we will find a simpler verification method for it. For that we introduce the following definition.

DEFINITION 3.16. *Let $\mathcal{A} = \langle S, A \longrightarrow\rangle$ be an LTS. A generalized inductive weak bisimulation is a pair (D,R) where D is a divergence set, R is a weak bisimulation, and moreover whenever $(s,t) \in R$ then either there exist $s',t' \in D$ such that $s \Longrightarrow s', t \Longrightarrow t'$, or $(s,t) \in \mathcal{I}(R)$.* □

Our aim is to establish generalized inductive weak bisimulation as a verification method for $\approx^{\Uparrow}$.

LEMMA 3.17. *Let R be a weak bisimulation. Then for all $(s,t) \in \mathcal{I}(R)$ the following hold: if $(s,t) \in R$, and $s \Uparrow$, then there exist s',t' such that $s \Longrightarrow s'$, $t \stackrel{\tau}{\Longrightarrow} t'$, $s' \Uparrow$ and $(s',t') \in R$.*

Proof: Easily follows from Lemma 2.19. □

THEOREM 3.18. *Let D be a divergence set, and (D,R) a generalized inductive weak bisimulation. Then R is a divergence preserving weak bisimulation.*

Proof: We will show that R is a divergence preserving weak bisimulation. Suppose $(s,t) \in R$, and $s \Uparrow$. Since (D,R) is a generalized inductive weak bisimulation, so either there exist $s',t' \in D$ such that $s \Longrightarrow s', t \Longrightarrow t'$ or $(s,t) \in \mathcal{I}(R)$. In the first case, since D is a divergence set, clearly $t \Uparrow$. In the second case, we can apply Lemma 3.17 to obtain s',t' such that $s \Longrightarrow s'$, $t \stackrel{\tau}{\Longrightarrow} t'$, $s' \Uparrow$ and $(s',t') \in R$. Then we can repeat the process for $(s',t') \in R$ with $s' \Uparrow$. Continue with this process, either it stops after finite steps to obtain $t \Uparrow$ or it goes to infinity to obtain $t \Uparrow$. □

We write $s \Downarrow$ if s is not divergent.

LEMMA 3.19. *If $s \approx^{\Uparrow} t$ and $s \Downarrow, t \Downarrow$, then $(s,t) \in \mathcal{I}(\approx^{\Uparrow})$.*

Proof: Define a binary relation $\succ$ on pairs of states such that $(s,t) \succ (s',t')$ if and only if $s \Downarrow, t \Downarrow$ and either $s \stackrel{\tau}{\longrightarrow} s'$ and $t = t'$ or $t \stackrel{\tau}{\longrightarrow} t'$ and $s = s'$. Then $\succ$ clearly is a well founded relation.

Now let $s \approx^{\Uparrow} t$ and $s \Downarrow, t \Downarrow$, we prove $(s,t) \in \mathcal{I}(\approx^{\Uparrow})$ by well founded induction on $\succ$. Suppose $s \stackrel{\tau}{\longrightarrow} s'$, then since $\approx^{\Uparrow}$ is a bisimulation, there exists t' such that $t \Longrightarrow t'$ with $s' \approx^{\Uparrow} t'$. For this t', either $t \stackrel{\tau}{\Longrightarrow} t'$, or $t = t'$, and in this case $(s,t) \succ (s',t')$, and $s' \Downarrow, t' \Downarrow$, then by the induction hypothesis $(s',t') \in \mathcal{I}(\approx^{\Uparrow})$. In the same way we can show that if $t \stackrel{\tau}{\longrightarrow} t'$ then either $s \stackrel{\tau}{\Longrightarrow} s'$ for some s' with $s' \approx^{\Uparrow} t'$, or $s \approx^{\Uparrow} t'$ and $(s,t') \in \mathcal{I}(\approx^{\Uparrow})$. Hence $(s,t) \in \mathcal{I}(\approx^{\Uparrow})$. □

```
     type Node = { val: Val,
                   next: ptr_to Node}
     Top: ptr_to Node
     type Op=enum{NONE,PUSH,POP}
     type opInfo={op:OP,
                   node:ptr_to Node}
     opInfos: array[numprocs] of opInfo
     collision: array[size] of ProcessId

     push(v: Val) =
A0   n: ptr_to Node := newNode()
A1      n-> val := v
A2      info: opInfo :=(PUSH,n)
A3   loop
A4   if tryPush(n) then exit
A5   if tryElimination(& info) then exit
A6   endloop
A7   return

     tryPush(n:ptr_to Node):boolean=
C1   ss:ptr_to Node := Top
C2   n->next:= ss
C3   return CAS(&Top, ss, n)
```

```
     pop(): Val =
B0   info: opInfo:=(POP,null)
B1   loop
B2      if tryPop(info.node) then exit
B3      if tryEliminate(&info) then exit
B4   endloop
B5   if info.node=null then
B6       return empty
B7   else
B8       v: Val:=info.node-> val
B10     return v
B11  fi

     tryPop(n:ptr_to Node): boolean=
D1   ss: ptr_to Node:= Top
D2   if ss=null then
D3      n:=null
D4      return true
D5   else
D6       n:= ss
D7       ssn: ptr_to Node:= ss-> next
D8       return CAS(&Top, ss, ssn)
D9   fi
```

Figure 3. Pseudo-code for HSY collision stack

THEOREM 3.20. *Let s, t be two states. Then $s \approx^{\Uparrow} t$ if and only if there is a generalized inductive weak bisimulation (D, R) such that $(s, t) \in R$.*

Proof: Suppose (D, R) is a generalized inductive weak bisimulation and $(s, t) \in R$, then by Theorem 3.18 $s \approx^{\Uparrow} t$.

Let $D = \{s \mid s \Uparrow\}$, then by Lemma 3.19, whenever $s \approx^{\Uparrow} t$ then either $s, t \in D$, or $(s, t) \in \mathcal{I}(\approx^{\Uparrow})$. Thus $(D, \approx^{\Uparrow})$ is a generalized inductive weak bisimulation. So if $s \approx^{\Uparrow} t$ then there is a generalized inductive weak bisimulation $(D, \approx^{\Uparrow})$ such that $(s, t) \in \approx^{\Uparrow}$. □

With Theorem 3.20, generalized inductive weak bisimulation can be used as a verification method for $\approx^{\Uparrow}$. That is to say, in order to show $s \approx^{\Uparrow} t$, one just needs to find a generalized inductive bisimulation (D, R) with $(s, t) \in R$.

4. Proving Correctness of a Concurrent Stack

In this section we will use the inductive weak bisimulation method to prove the correctness of a concurrent stack implementation, the so-called HSY collision stack named by the initials of the authors (8).

A stack is often implemented as an object with a pointer variable `Top` pointing to the top node of a linked list. When such a stack allows concurrent calls by different client programs, the pointer `Top` and the linked list are exposed to concurrent access of different programs. In this case care must be taken so that the proper structure of the stack is maintained.

In Figure 3(together with Figure 4) is the pseudo-code for the HSY collision stack algorithm taken from (9). HSY algorithm uses the following observation: if a `push` followed by a `pop` are performed on a stack, the data structure's state does not change. This means that if one can cause pairs of pushes and pops to meet and pair up in separate locations, the threads can exchange values without having to touch a centralized structure since they have "eliminated" each other's effect on it (the process is also called "collision"). Elimination is implemented by using a collision array in which threads pick random locations in order to try and collide. Pairs of threads that "collide" in some location run through a lock-free synchronization protocol, and all such disjoint collisions can be performed in parallel. With these ideas, the push method creates a new node (line `A0,A1,A2`) and then it repeatedly tries to swing the `Top` from the current top-of-stack to its `next` field by the `CAS` (compare and set) statement (line `A4`), or tries to collide with a pop call of another thread in the collision array (line `A5`). The method returns until either it has pushed the node into the stack by a successful `CAS` or it has completed collision with another thread. The pop method follows a similar process. To simplify the matter when a node is popped there is no `free` statement to release the memory of the node, and we assume that there is a garbage collector to manage memory usage.

In order to show the correctness of the above algorithm, we use the pseudo-code of a simple algorithm shown in Figure 5 (with program labels `E0,E1,...,F0,F1,...`) which uses atomic block brackets `atomic`$\{\ldots\}$ which requires non-interrupted execution of the operations between the brackets. The description with atomic blocks is easily seen to be correct stack operations under concurrent access. On the one hand, the non-interrupted execution of the operations in the atomic blocks makes it clear that the correct structure of the linked list is maintained. On the other hand all the operations in the entire description are simple ones without looping, so the methods are clearly terminating which implies progress condition. In fact it can be formally stated and proved that the program is linearizable to a stack specification and that it satisfies the lock-free progress condition, where linearizability, first introduced in (13), is a widely used correctness condition for concurrent objects.

While the HSY stack implementation raises obvious concerns of correctness because it allows concurrent access of the central

```
      tryElimination(pinfo: ptr_to opInfo): boolean =
X1    opInfos[mypid]:= *pinfo
X2    pos := getPosition()
X3    repeat him:= collision[pos]
X4    until CAS(collision[pos], him, mypid)
X5    qinfo:= opInfos[him]
X6    if qinfo.op+pinfo->op= PUSH + POP then
X7        if CAS(&opInfos[myid], *pinfo, (NONE,pinfo->node)) then
X8            if CAS(&opInfos[him], qinfo, (NONE, pinfo->node)) then
X9                  pinfo->node := qinfo.node
X10                 return true
X11           else
X12                 return false
X13           fi
X14       else
X15           pinfo->node := opInfos[myid].node
X16           return true
X17       fi
X18   fi
X19   delay
X20   if !CAS(&opInfos[myid],*pinfo,(NONE,pinfo->node)) then
X21         pinfo->node := opInfos[myid].node
X22         return true
X23   else
X24         return false
X25   fi
```

Figure 4. Pseudo-code for HSY collision stack (continued)

linked list. With all the loop statements whether a call will be returned eventually also becomes a question. Our aim here is to show that the HSY stack implementation has the same behavior w.r.t. $\approx^{\Delta}$ as the description using atomic blocks by establishing an inductive weak bisimulation between them. Now, by Theorem 3.14 and Theorem 3.15, $\approx^{\Delta}$ preserves trace equivalence, and according to (15) trace equivalence preserves linearizability, then if the program in Figure 5 is linearizable so is the program in Figure 3. Also by Theorem 3.14 and Theorem 3.15, $\approx^{\Delta}$ preserves ω-trace equivalence, and according to (16) ω-trace equivalence preserves lock-free progress condition, then if the program in Figure 5 satisfies lock-free progress condition so does the program in Figure 3. Thus the correctness of the program with atomic block structures could ensure the correctness of the HSY stack.

In order to establish the wanted inductive weak bisimulation, we need an LTS to describe the object systems. To simplify the matter, we assume that we are dealing with a concurrent system in which the number of concurrent threads is fixed at n. In the case of dynamic creation of threads the verification principle is the same, the only difference is that we need a slightly more general representation for configurations. Under this assumption, the concurrent object has n threads running in parallel sharing the common heap H and memory M, and it has configurations of the form

$$\langle H, M, (t_1, l_1, m_1), \ldots, (t_n, l_n, m_n)\rangle,$$

where $(t_1, l_1, m_1), \ldots, (t_n, l_n, m_n)$ are the states of concurrent treads with t_i being the thread names and l_i the program counters (labels) of the thread and m_i the local memories. We assume that in each such configuration, the treads are either running the code labeled `A,B,C,D,X` or running the code labeled `E` and `F`. This assumption is obviously valid for any configuration that is reachable from the initial configurations that will be introduced shortly. We call the former type AB configuration, and the latter type EF configuration. The transition relation between configurations is completely determined by the following rule: if a thread has a transition $\langle H, M, (t_i, l_i, m_i)\rangle \xrightarrow{\alpha} \langle H', M', (t_i, l_i', m_i')\rangle$, then the configuration $\langle H, M, (t_1, l_1, m_1)\ldots(t_i, l_i, m_i)\ldots\rangle$ has an α transition to $\langle H', M', (t_1, l_1, m_1)\ldots(t_i, l_i', m_i')\ldots\rangle$, i.e. the resulting configuration changed the shared state and the local state of the thread without changing the local states of other threads. In other words, this rule declares that the transitions of a configuration is determined by the interleaving of the transitions of the threads. The transitions of the threads are described in the following sections.

First, the transitions for the program of Figure 5 are listed in Figure 6. `EFidle` as a program label means that at the moment the thread is not executing any method of the stack, in other words the thread is executing the client code. For memory m, we write $m[\mathtt{v} \mapsto d]$ for the modified memory where the value of `v` is changed to d and the values of other variables remain the same as in m.

For example the first line of Figure 6 says that when thread t is not executing any method of the stack, the method push can be invoked with a parameter d, and then the configuration changes so that the local memory is modified by saving d at the local variable `v`, and the thread is ready to execute the first line of method push labeled $\mathtt{E}_0$. The second line of Figure 6 says that when thread t is at label $\mathtt{E}_0$, after one step internal execution a node is created with the pointer value q pointing to it, and the heap H is expanded with the new node, and q is saved in the local variable `n`, and after the execution the thread has arrived at label `E1`.

```
     type Node = { val: Val,
                   next: ptr_to Node}
     Top: ptr_to Node

     push(v: Val) =
E0   n: ptr_to Node := newNode()
E1   atomic {
E2      n-> val := v
E3      n->next := Top
E4      Top := n
E5   }
E6   return
```

```
     pop(): Val =
F0   atomic {
F1      n:ptr_to Node:= Top
F2      if n <> null then
F3        {Top:=n->next
F4        v: Val :=n-> val
F5         }
F6   }
F7   if n = null then
F8      return empty
F9   else
F10     return v
F11  fi
```

Figure 5. Pseudo-code for lock-free stack specification using atomic blocks

$$
\begin{array}{lcl}
\langle H, M, (t, \texttt{EFidle}, m)\rangle & \xrightarrow{(t,\text{call push}(d))} & \langle H, M, (t, \texttt{E0}, m[\texttt{v} \mapsto d])\rangle \\
\langle H, M, (t, \texttt{E0}, m)\rangle & \xrightarrow{\tau} & \langle H \uplus [q \mapsto \{\texttt{Val} : \bot, \texttt{next} : \texttt{null}\}], \\
 & & \quad M, (t, \texttt{E1}, m[\texttt{n} \mapsto q])\rangle \\
\langle H[q \mapsto \{\texttt{Val} : d, \texttt{next} : r\}], & & \langle H[q \mapsto \{\texttt{Val} : m(\texttt{v}), \texttt{next} : M(\texttt{Top})\})], \\
\quad M, (t, \texttt{E1}, m[\texttt{n} \mapsto q])\rangle & \xrightarrow{\tau} & \quad M[\texttt{Top} \mapsto q], (t, \texttt{E6}, m[\texttt{n} \mapsto q])\rangle \\
\langle H, M, (t, \texttt{E6}, m)\rangle & \xrightarrow{(t,\text{ret}()\texttt{push})} & \langle H, M, (t, \texttt{EFidle}, m)\rangle \\
\langle H, M, (t, \texttt{EFidle}, m)\rangle & \xrightarrow{(t,\text{call pop}())} & \langle H, M, (t, \texttt{F0}, m)\rangle \\
\langle H, M, (t, \texttt{F0}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{F7}, m[\texttt{n} \mapsto null])\rangle \quad (M(\texttt{Top}) = null) \\
\langle H, M, (t, \texttt{F7}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{F8}, m)\rangle \qquad (m(\texttt{n}) = null) \\
\langle H, M, (t, \texttt{F8}, m)\rangle & \xrightarrow{(t,\text{ret}(\texttt{empty})\texttt{pop})} & \langle H, M, (t, \texttt{EFidle}, m)\rangle \\
\langle H[p \mapsto \{\texttt{Val} : d, \texttt{next} : q\}], & & \langle H, M[\texttt{Top} \mapsto q], \\
\quad M[\texttt{Top} \mapsto p], (t, \texttt{F0}, m)\rangle & \xrightarrow{\tau} & \quad (t, \texttt{F7}, m[\texttt{v} \mapsto d, \texttt{n} \mapsto p])\rangle \\
\langle H, M, (t, \texttt{F7}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{F10}, m)\rangle \qquad (m(\texttt{n}) \neq null) \\
\langle H, M, (t, \texttt{F10}, m)\rangle & \xrightarrow{(t,\text{ret}(m(\texttt{v}))\texttt{pop})} & \langle H, M, (t, \texttt{EFidle}, m)\rangle
\end{array}
$$

Figure 6. Transitions for thread t executing E and F

The third line represents a long execution sequence in one step, because this sequence is assumed atomic. The effect of this sequence is that the shared variable `Top` is updated, the new node is inserted into the linked list represented by H. The rest of the transitions for the program in Figure 6 can be derived by referring to the code in Figure 5 in the same way.

Similar thing happens in Figure 7 which shows examples of the transitions for HSY collision stack. Particularly interesting is how the `CAS` expression is interpreted by the transition rules. Informally $\texttt{CAS}(\&\texttt{Top}, \texttt{ss}, \texttt{n})$ compares the value in `ss` with the value in `Top`, and if they are the same then update the content of `Top` with the value in `n` and the `CAS` expression evaluates to true, and if they are not the same then no change is made to the contents of the variables and the `CAS` expression evaluates to false. The transitions concerning this part are the 5th rule (for the true case) and the 6th rule (for the false case) in Figure 7. Since we assume that the program is running under a good garbage collector, we do not need to concern the so-called ABA problem which may arise when the memory for nodes is recycled. More transitions for HSY stack can be found in Figure 8.

After settling the transitions, we need to specify the relation R which we will show to be an inductive weak bisimulation. First of all, R should relate the initial configuration of the HSY concurrent stack implementation (using ϵ for the initial empty heap)

$$\langle \epsilon, M, (t_1, \texttt{ABidle}, m_1), \ldots, (t_n, \texttt{ABidle}, m_n)\rangle,$$

to the initial configuration of the abstract concurrent stack

$$\langle \epsilon, M, (t_1, \texttt{EFidle}, m_1) \ldots (t_n, \texttt{EFidle}, m_n)\rangle,$$

where $M, m_1, \ldots, m_n$ are all initial memories which are undefined anywhere (we could have used different memories according to whether they are used in an AB configuration or EF configuration with the same analysis process, since the former code includes all the variables used in the latter.)

To specify the other elements of R, first note that we only need to consider configurations which are reachable from the two initial configurations by the transitions. Let $Cong$ denote the set of all such reachable configurations. Second, note that the states of individual threads can be classified into the following five kinds:

$$
\begin{array}{lcl}
\langle H, M, (t, \texttt{ABidle}, m)\rangle & \xrightarrow{(t,\text{call push}(d))} & \langle H, M, (t, \texttt{A0}, m[\texttt{v} \mapsto d])\rangle \\
\langle H, M, (t, \texttt{A0}, m)\rangle & \xrightarrow{\tau} & \langle H \uplus [q \mapsto \{\texttt{Val} : \bot, \texttt{next} : null\}], M, (t, \texttt{A1}, m[\texttt{n} \mapsto q])\rangle \\
\langle H, M, (t, \texttt{A2}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{A3}, m[\texttt{info} \mapsto \texttt{PUSH}, m(\texttt{n})])\rangle \\
\langle H, M, (t, \texttt{A3}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{A4}, m)\rangle \\
\langle H, M[\texttt{Top} \mapsto p], (t, \texttt{C3}, m[\texttt{n} \mapsto q, \texttt{ss} \mapsto p])\rangle & \xrightarrow{\tau} & \langle H, M[\texttt{Top} \mapsto q], (t, \texttt{A7}, m[\texttt{n} \mapsto q, \texttt{ss} \mapsto p])\rangle \\
\langle H, M[\texttt{Top} \mapsto r], (t, \texttt{C3}, m[\texttt{ss} \mapsto p])\rangle & \xrightarrow{\tau} & \langle H, M[\texttt{Top} \mapsto r], (t, \texttt{A5}, m[\texttt{ss} \mapsto p])\rangle \quad (p \neq r) \\
\langle H, M[\texttt{Top} \mapsto r], (t, \texttt{A7}, m)\rangle & \xrightarrow{(t,\text{ret()push})} & \langle H, M, (t, \texttt{ABidle}, m)\rangle \\
\langle H, M, (t, \texttt{A5}, m)\rangle & \xrightarrow{\tau} & \langle H, M, (t, \texttt{AX1}, m[\texttt{pinfo} \mapsto m(\texttt{info})])\rangle
\end{array}
$$

Figure 7. Transitions for thread t executing HSY program

1. *idle states*, in which neither push nor pop is invoked. Those are states of the form $\langle H, M, (t, \texttt{EFidle}, m)\rangle$ or $\langle H, M, (t, \texttt{ABidle}, m)\rangle$.
2. *pre-linearization push states*, in which the thread has invoked a push method but the call has not taken any effect on the shared memory M and H yet. Those are states of the form $\langle H, M, (t, l, m)\rangle$ where l is one of $\{\texttt{E0}, \texttt{E1}, \texttt{A0} - \texttt{A6}, \texttt{AX1} - \texttt{AX8}, \texttt{AX11} - \texttt{AX13}, \texttt{AX17} - \texttt{AX20}, \texttt{AX23} - \texttt{AX25}\}$. (The double labels such as $\texttt{AX1}, \texttt{BX1}$ are the labels for the collision procedure, they represent that the thread is a call from push and pop respectively. Could have put $\texttt{A}, \texttt{B}$ in front of $\texttt{C}, \texttt{D}$ as well, but that is unnecessary because without them would not cause any confusion. A range $\texttt{AX11} - \texttt{AX13}$ means $\texttt{AX11}, \texttt{AX12}, \texttt{AX13}$ etc.)
3. *post-linearization push states*, in which the thread has made a push method call and the call has taken effect on the shared memory M and H, but the call has not been returned yet. Those are states of the form $\langle H, M, (t, l, m)\rangle$ where l is one of $\{\texttt{E6}, \texttt{A7}, \texttt{AX9}, \texttt{AX10}, \texttt{AX14} - \texttt{AX16}, \texttt{AX21}, \texttt{AX22}\}$.
4. *pre-linearized pop states*, in which the thread has invoked a pop method but the call has not taken any effect on the shared memory M and H. Those are states of the form $\langle H, M, (t, l, m)\rangle$ where l is one of $\{\texttt{F0}, \texttt{B0} - \texttt{B4}, \texttt{BX1} - \texttt{BX8}, \texttt{BX11} - \texttt{BX13}, \texttt{BX17} - \texttt{BX20}, \texttt{BX23} - \texttt{BX25}\}$.
5. *post-linearization pop states*, in which the thread has made a pop method call and the call has taken effect on the shared memory M and H, but the call has not been returned yet. Those are states of the form $\langle H, M, (t, l, m)\rangle$ where l is one of $\{\texttt{F7}, \texttt{F8}, \texttt{F10}, \texttt{B5} - \texttt{B10}, \texttt{BX9}, \texttt{BX10}, \texttt{BX14} - \texttt{BX16}, \texttt{BX21}, \texttt{BX22}\}$.

Now R is defined as a binary relation such that a pair of configurations $\langle H, M, (t_1, l_1, m_1) \dots (t_n, l_n, m_n)\rangle$ and $\langle H', M', (t_1, l'_1, m'_1) \dots (t_n, l'_n, m'_n)\rangle$ is related in R if and only if the following hold:

1. $\langle H, M, (t_1, l_1, m_1) \dots (t_n, l_n, m_n)\rangle$ is a type AB configuration, and $\langle H', M', (t_1, l'_1, m'_1) \dots (t_n, l'_n, m'_n)\rangle$ is a type EF configuration, and they are both in $Cong$.
2. $H = H', M(\texttt{Top}) = M'(\texttt{Top})$.
3. for each i, (t_i, l_i, m_i) and (t_i, l'_i, m'_i) satisfy one of the following conditions

idle: both in idle states;

push: both in pre-linearization push states or both in post-linearization push states, $m_i(\texttt{n}) = m'_i(\texttt{n}), m_i(\texttt{v}) = m'_i(\texttt{v})$;

pre-pop: both in pre-linearization pop states;

post-pop: both in post-linearization pop states, $m_i(\texttt{ss}) = m'_i(\texttt{n}), m_i(\texttt{v}) = m'_i(\texttt{v})$.

Obviously R contains the pair of initial configurations. Our remaining task is to show that R is an inductive weak bisimulation.

It is routine and tedious to check that R is a weak bisimulation, which we will skip. To show that R is an inductive weak bisimulation, we still need to establish $R \subseteq \mathcal{I}(R)$, which we sketch as follows.

It is not difficult to observe that the code for HSY stack contains a double loop for the procedure push: the code from A3 to A6 is the outer loop and that of X3, X4 is the inner loop. It also contains a similar double loop for the procedure pop. Now we can define a well-founded order between two type AB configurations such that $\Gamma > \Gamma'$ if and only if $\Gamma \xrightarrow{\tau} \Gamma'$ where the τ transition is either caused by some thread had a failed execution of CAS at X4 or caused by some thread had executed X3. To see that $>$ is well-founded, suppose there is an infinite decreasing sequence $\Gamma_1 > \dots \Gamma_i > \dots$, then the sequence must be purely caused by a number of threads looping between lines X3, X4, which is obviously not possible. Hence $>$ is well-founded. We can define another well-founded order $\succ$ such that $\Gamma \succ \Gamma'$ if and only if $\Gamma \xrightarrow{\tau} \Gamma'$ where the τ transition is not caused by any threads' successful execution of CAS at C3, D8. To see that $\succ$ is well-founded, suppose there is an infinite decreasing sequence $\Gamma_1 \succ \dots \Gamma_i \succ \dots$, then the sequence must be caused by a number of threads looping between the lines A3 and A6 or B1 and B4 (the outer loop), since $>$ is well-founded and the inner loop cannot produce such infinite sequence.

Now, for all $(\Gamma_1, \Gamma_2) \in R$, we can prove $(\Gamma_1, \Gamma_2) \in \mathcal{I}(R)$ by well-founded induction on $\succ$.

Suppose $(\Gamma_1, \Gamma_2) \in R$ with $\Gamma_1 \xrightarrow{\tau} \Gamma'_1$. Then according to the transition rule for configurations, there is (t_i, l_i, m_i) in Γ_1 such that $(t_i, l_i, m_i) \xrightarrow{\tau} (t_i, l'_i, m'_i)$. We discuss in two cases. The first case is that l_i is one of C3, D8 and $m_i(\texttt{ss}) = M(\texttt{Top})$, then the CAS statement must be successful, and it is not difficult to see that the corresponding (t_i, l''_i, m''_i) in Γ_2 has a sequence of τ-transitions such that $\Gamma_2 \xRightarrow{\tau} \Gamma'_2$ for some Γ'_2 with $(\Gamma'_1, \Gamma'_2) \in R$. If the condition for the first case is not true, then obviously $\Gamma_1 \succ \Gamma'_1$. In this case, since R is a weak bisimulation, there is Γ'_2 with $\Gamma_2 \Longrightarrow \Gamma'_2$ and $(\Gamma'_1, \Gamma'_2) \in R$. If $\Gamma'_2 \neq \Gamma_2$, then we have $\Gamma_2 \xRightarrow{\tau} \Gamma'_2$. If $\Gamma'_2 = \Gamma_2$, then we have $(\Gamma'_1, \Gamma_2) \in R$ and $\Gamma_1 \succ \Gamma'_1$, by the induction hypothesis we know $(\Gamma_1, \Gamma_2) \in \mathcal{I}(R)$. Thus $(\Gamma_1, \Gamma_2) \in \mathcal{I}(R)$, and we completed the proof (we do not need to check the τ transitions for Γ_2 since obviously any τ transition sequence from Γ_2 is finite so that the matching can be trivially found).

To summarize, we have successfully carried out the verification of HSY stack implementation by establishing an inductive weak bisimulation between it and a specification using atomic block structures. Then it follows that, by the discussion earlier, the im-

$\langle H[q \mapsto \{\texttt{Val} : d, \texttt{next} : r\}],$
$\quad M, (t, \texttt{A1}, m[\texttt{n} \mapsto q])\rangle \xrightarrow{\tau} \langle H[q \mapsto \{\texttt{Val} : m(\texttt{v}), \texttt{next} : r\}],$
$\quad M, (t, \texttt{A2}, m[\texttt{n} \mapsto q])\rangle$

$\langle H, M, (t, \texttt{A4}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{C1}, m)\rangle$

$\langle H, M, (t, \texttt{C1}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{C2}, m[\texttt{ss} \mapsto M(Top)])\rangle$

$\langle H[q \mapsto \{\texttt{Val} : m(\texttt{v}), \texttt{next} : r\}],$
$\quad M, (t, \texttt{C2}, m[\texttt{n} \mapsto q])\rangle \xrightarrow{\tau} \langle H[q \mapsto \{\texttt{Val} : m(\texttt{v}), \texttt{next} : m(ss)\}],$
$\quad M, (t, \texttt{C3}, m[\texttt{n} \mapsto q])\rangle$

$\langle H, M, (t, \texttt{AX1}, m)\rangle \xrightarrow{\tau} \langle H, M[\texttt{Opinfos}[\texttt{mypid}] \mapsto m(pinfo)], (t, \texttt{AX2}, m)\rangle$

$\langle H, M, (t, \texttt{AX2}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX3}, m[\texttt{pos} \mapsto getPosition()])\rangle$

$\langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto q],$
$\quad (t, \texttt{AX3}, m[\texttt{him} \mapsto x])\rangle \xrightarrow{\tau} \langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto q],$
$\quad (t, \texttt{AX4}, m[\texttt{him} \mapsto q])\rangle$

$\langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto q],$
$\quad (t, \texttt{AX4}, m[\texttt{him} \mapsto q])\rangle \xrightarrow{\tau} \langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto mypid],$
$\quad (t, \texttt{AX5}, m)\rangle$

$\langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto q],$
$\quad (t, \texttt{AX4}, m[\texttt{him} \mapsto r])\rangle \xrightarrow{\tau} \langle H, M[\texttt{Collision}[\texttt{pos}] \mapsto q],$
$\quad (t, \texttt{AX3}, m[\texttt{him} \mapsto x])\rangle \quad (q \neq r)$

$\langle H, M, (t, \texttt{AX5}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX6}, m[\texttt{qinfo} \mapsto M(\texttt{Opinfos}[\texttt{him}])])\rangle$

$\langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_1\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M, (t, \texttt{AX6}, m[\texttt{pinfo} \mapsto (e_1, p), \texttt{qinfo} \mapsto (e_2, q)])\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX7}, m)\rangle \quad (e_1 + e_2 = PUSH + POP)$

$\langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_1\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M, (t, \texttt{AX6}, m[\texttt{pinfo} \mapsto (e_1, p), \texttt{qinfo} \mapsto (e_2, q)])\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX20}, m)\rangle \quad (e_1 + e_2 \neq PUSH + POP)$

$\langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e, p)],$
$\quad (t, \texttt{AX7}, m[\texttt{pinfo} \mapsto (e, p)])\rangle \xrightarrow{\tau} \langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (NONE, p)],$
$\quad (t, \texttt{AX8}, m[\texttt{pinfo} \mapsto (e, p)])\rangle$

$\langle H[q \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e, q)],$
$\quad (t, \texttt{AX8}, m[\texttt{pinfo} \mapsto (e, q)])\rangle \xrightarrow{\tau} \langle H[q \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (NONE, q)],$
$\quad (t, \texttt{AX9}, m[\texttt{pinfo} \mapsto (e, q)])\rangle$

$\langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_1\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M, (t, \texttt{AX9}, m[\texttt{pinfo} \mapsto (e_1, p), \texttt{qinfo} \mapsto (e_2, q)])\rangle \xrightarrow{\tau} \langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_1\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M, (t, \texttt{AX10}, m[\texttt{pinfo} \mapsto (e_1, q), \texttt{qinfo} \mapsto (e_2, q)])\rangle$

$\langle H, M, (t, \texttt{AX10}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{A7}, m)\rangle$

$\langle H, M[\texttt{Opinfos}[\texttt{mypid}] \mapsto \texttt{r}],$
$\quad (t, \texttt{AX8}, m[\texttt{qinfo} \mapsto q])\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{A3}, m)\rangle \quad (r \neq q)$

$\langle H, M[\texttt{Opinfos}[\texttt{mypid}] \mapsto r],$
$\quad (t, \texttt{AX7}, m[\texttt{pinfo} \mapsto p])\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX15}, m)\rangle \quad (r \neq p)$

$\langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e_1, p)],$
$\quad (t, \texttt{AX15}, m[\texttt{pinfo} \mapsto (e_2, q)])\rangle \xrightarrow{\tau} \langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e_1, p)],$
$\quad (t, \texttt{AX16}, m[\texttt{pinfo} \mapsto (e_2, p)])\rangle$

$\langle H, M, (t, \texttt{AX16}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{A7}, m)\rangle$

$\langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e, p)],$
$\quad (t, \texttt{AX20}, m[\texttt{pinfo} \mapsto (e, p)])\rangle \xrightarrow{\tau} \langle H[p \mapsto \{\texttt{Val} : f, \texttt{next} : g\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (NONE, p)],$
$\quad (t, \texttt{AX24}, m[\texttt{pinfo} \mapsto (e, p)])\rangle$

$\langle H, M, (t, \texttt{AX24}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{A3}, m)\rangle$

$\langle H, M[\texttt{Opinfos}[\texttt{mypid}] \mapsto r],$
$\quad (t, \texttt{AX20}, m[\texttt{pinfo} \mapsto p])\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{AX21}, m)\rangle \quad (r \neq p)$

$\langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_1\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e_1, p)],$
$\quad (t, \texttt{AX21}, m[\texttt{pinfo} \mapsto (e_2, q)])\rangle \xrightarrow{\tau} \langle H[p \mapsto \{\texttt{Val} : f_1, \texttt{next} : g_2\}, q \mapsto \{\texttt{Val} : f_2, \texttt{next} : g_2\}],$
$\quad M[\texttt{Opinfos}[\texttt{mypid}] \mapsto (e_1, p)],$
$\quad (t, \texttt{AX22}, m[\texttt{pinfo} \mapsto (e_2, p)])\rangle$

$\langle H, M, (t, \texttt{AX22}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{A7}, m)\rangle$

$\langle H, M, (t, \texttt{ABidle}, m)\rangle \xrightarrow{(t, \text{call pop}())} \langle H, M, (t, \texttt{B0}, m)\rangle$

$\langle H, M, (t, \texttt{B0}, m)\rangle \xrightarrow{\tau} \langle H \uplus [q \mapsto \{\texttt{Val} : \bot, \texttt{next} : null\}],$
$\quad M, (t, \texttt{B1}, m[\texttt{info} \mapsto (POP, q)])\rangle$

$\langle H, M, (t, \texttt{B1}, m)\rangle \xrightarrow{\tau} \langle H, M, (t, \texttt{B2}, m)\rangle$

$\langle H[q \mapsto \{\texttt{Val} : f, \texttt{next} : g\}], M,$
$\quad (t, \texttt{B2}, m[\texttt{info} \mapsto (POP, q)])\rangle \xrightarrow{\tau} \langle H[q \mapsto \{\texttt{Val} : f, \texttt{next} : g\}], M,$
$\quad (t, \texttt{D1}, m[\texttt{n} \mapsto q])\rangle$

Figure 8. More transitions for thread t executing HSY-collision stack

plementation has the same linearizability property as well as lock-free progress property as the specification. This case study shows that our method of inductive weak bisimulation is not over restrictive. An important point to note is that we cannot use inductive branching bisimulation to carry out this verification, since branching bisimulation equivalence does not hold between the two initial configurations because of the possibility of collision operation in HSY stack.

5. Related Works

The divergence preserving weak bisimilarity has been studied in many works (5; 6), with idea tracing back to Hennessy and Plotkin (1). In these works emphasis was given to how to define a bisimulation equivalence which preserves divergence (sometimes overloaded with under specification), little were considered how to verify such an equivalence with a bisimulation technique. (14) devised an equational inference system for divergence preserving weak bisimularity which is complete for regular processes.

The works on branching bisimulation (7; 10; 11) provide important foundation for our work. Branching bisimilarity with explicit divergence, first introduced by van Glabbeek and Weijland in (7), motivated the weak bisimilarity with explicit divergence studied in this work. Because the useful tool of colored trace is not available in this case, properties of weak bisimilarity with explicit divergence is more difficult to obtain than corresponding properties for branching bisimilarity with explicit divergence. In (10) some relational characterizations are studied for branching bisimilarity with explicit divergence. However these characterizations are not feasible as a verification method because they all rely on checking complicated conditions relating to divergence.

In (12), Namjoshi proposed a verification method based on well-founded induction to verify equality with respect to stuttering equivalence, which is the branching bisimilarity with explicit divergence in the setting of Kripke structure. This method is, among those in the literature, the closest one to the inductive method proposed in this paper in that when doing verification it also relies only on simple observations of transitions. However since it mixes two induction procedures into one well-founded order (one for divergence sensitivity one for stuttering), it is unclear how to use it for equivalences other than stuttering equivalence (branching bisimilarity with explicit divergence).

In verification of concurrent objects, progress properties such as liveness and termination preserving are of great interest, and have been studied in various works (18; 16; 17). In (18), Gotsman and Yang showed that lock-freedom with linearizability implies termination sensitive contextual refinement. The other direction was proved by Liang *et al.* in (16), which makes it possible to verify lock-freedom using proof methods for contextual refinement. In (17), by using the idea of rely-guarantee reasoning, Liang *et al.* proposed a simulation relation RGSim-T as a method to verify termination-preserving refinement of concurrent programs which supports compositional verification. Our work in this paper differs from these works in that we are using and developing the bisimulation method of concurrency theory to study correctness with progress for concurrent objects, following the idea proposed in (19).

Last but not the least, Yang *et al.* proposed a method in (19) to verify concurrent objects, which uses branching bisimilarity to quotient the state space of the object system in order to reduce its size. Being the first one, as far as we know, to advocate the idea of using branching bisimilarity with explicit divergence to prove linearizability and progress of concurrent data structures, this work provides a great source of inspiration for our work. Our work is making an effort to improve the theoretical foundation that supports the idea. In particular, the inductive branching bisimulation introduced in our paper could provide a useful tool in using branching bisimilarity with explicit divergence to verify correctness of concurrent data structures as proposed in (19).

6. Conclusion

In this paper, we introduced a divergence preserving weak bisimulation equivalence called weak bisimilarity with explicit divergence, which can be verified by a method called inductive weak bisimulation which generalizes the weak bisimulation method. As an application example we use inductive weak bisimulation to prove that the HSY collision stack implementation and the stack specification described using atomic block structures are equivalent under weak bisimilarity with explicit divergence, thus proved the linearizability and lock-free progress property of the collision stack implementation at the same time. This example shows that the method is not over restrictive. It is interesting to note that the collision stack implementation is not equivalent to the stack specification described using atomic block structures under branching bisimilarity with explicit divergence. Hence weak bisimilarity with explicit divergence is a more applicable tool for establishing system corrrectness compared to branching bisimilarity with explicit divergence.

The inductive method for weak bisimilarity with explicit divergence turns out to be a general idea. We have also shown that it can be adapted to provide verification methods for branching bisimilarity with explicit divergence and the divergence preserving weak bisimilarity.

The introduced weak bisimilarity can be characterized by a modal logic, and there is a partition algorithm for equality checking of weak bisimilarity with explicit divergence for finite state systems. These are standard results for a bisimulation equivalence relation, and will be reported in detail in other works due to the shortage of space and limited relevance to the inductive method.

We hope that this inductive verification method could open up new possibilities for deciding equality of infinite state systems, such as equality of BPA processes under branching bisimilarity with explicit divergence. Also the new notion of complete weak (branching) bisimulation seems to be very helpful in theoretical analysis. We hope that it could be useful in future works.

Acknowledgments

We would like to thank Xiaoxiao Yang for directing our attention to divergence preservation bisimulation relations, and for her work on the topic which makes a starting point of our work. We thank the anonymous referees whose suggestions improved the presentation. This work is supported by NSFC-61161130530, NSFC-91418204, NSFC-61672504, NSFC-6147474, the 973 Program of China under Grant No. 2014CB340701.

References

[1] M. Hennessy and G. Plotkin. A term model for CCS, Lecture notes in computer science, Vol.88, Springer-Verlag, 1980.

[2] David Park: Concurrency and automata on infinite sequences. Lecture Notes in Computer Science 104,1981. Proceedings of 5th GI Conference.

[3] J. van Benthem. Modal Logic and Classic Logic. Bibliopolis, 1983.

[4] R. Milner: A complete axiomatisation for observational congruence of finite-state behaviours. Inf. Comput. 81(1989)227-247.

[5] D.J. Walker. Bisimulation and divergence, Information and Computation, vol. 85, pp. 212-241, 1990.

[6] Rob J. van Glabbeek: The Linear Time - Branching Time Spectrum II. CONCUR 1993: 66-81.

[7] Rob J. van Glabbeek, Peter Weijland: Branching time and abstraction in bisimulation semantics. J. ACM 43(3):555-600.1996

[8] D. Hendler, N. Shavit, and L. Yerushalmi: A scalable lock-free stack algorithm. In SPAA 2004: Proceedings of the sixteenth annual ACM symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain, pages 206-215, 2004

[9] R. Colvin, L. Groves. A Scalable lock-free stack algorithm and its verification. fifth IEEE International conference on software engineering and formal methods. 2007

[10] Rob J. van Glabbeek, Bas Luttik, Nikola Trcka: Branching Bisimilarity with Explicit Divergence. Fundam. Inform. 93(4): 371-392. 2009

[11] Rob J. van Glabbeek, Bas Luttik, Nikola Trcka: Computation tree logic with deadlock detection. Logical Methods in Computer Science. Vol.5(4:5) 2009, pages 1-24.

[12] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In 17th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), volume 1346 of Lecture Notes in Computer Science, pages 284-296.

[13] M. Herlihy, J. Wing, H. Hermanns: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463-492(1990)

[14] M. Lohrey, P. R. D'Argenio, H. Hermanns: Axiomatising divergence. Inf. Comput. 203(2005)115-144.

[15] I. Filipovic, P. O'Hearn, N.Rinetzky, H. Yang: Abstraction for concurrent objects. Theor. Comput. Sci. (2010).

[16] Hongjin Liang, Jan Hoffmann, Xinyu Feng, Zhong Shao: Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. CONCUR 2013: 227-241.

[17] Hongjin Liang, Xinyu Feng, Zhong Shao: Compositional verification of termination-preserving refinement of concurrent programs. CSL-LICS 2014: 65:1-10.

[18] Alexey Gotsman, Hongseok Yang: Liveness-Preserving Atomicity Abstraction. ICALP (2) 2011: 453-465.

[19] X. Yang, J. Katoen, H. Lin, H. Wu: Proving linearizability via branching bisimulation. CoRR abs/1609.07546(2016)