Next-preserving Branching Bisimulation

Nisansala Yatapanage
School of Information Technology and Electrical Engineering
The University of Queensland
nisansala.yatapanage@griffithuni.edu.au

Kirsten Winter
School of Information Technology and Electrical Engineering
The University of Queensland
kirsten@itee.uq.edu.au

ITEE TECHNICAL REPORT
No. 2013-02 (October 2013)

This report has been reviewed by the Research Committee of the School of Information Technology and Electrical Engineering, the University of Queensland. Please cite as follows:


www.itee.uq.edu.au
Next-preserving Branching Bisimulation

Nisansala Yatapanage*, Kirsten Winter
School of Information Technology and Electrical Engineering
The University of Queensland,
Brisbane, QLD 4072, Australia

Abstract

Bisimulations are in general equivalence relations between transition systems which assure that certain aspects of the behaviour of the systems are the same. For many applications it is not possible to maintain such an equivalence unless non-observable (stuttering) behaviour is ignored. However, existing bisimulation relations which permit the removal of non-observable behaviour are unable to preserve temporal logic formulas referring to the next step operator. In this paper we propose a novel bisimulation relation, called next-preserving branching bisimulation, which accomplishes this, maintaining the validity of formulas with the next step, while still allowing non-observable behaviour to be reduced. Based on van Glabbeek and Weijland’s notion of branching bisimulation with explicit divergence, we define the novel relation for which we prove the preservation of full CTL*.

As an example for its application we show how this definition gives rise to an advanced slicing procedure for temporal logics, a technique in which a system model is reduced to a slice which can be used as a substitute in verification and debugging. The result is a novel procedure for generating a slice that is next-preserving branching bisimilar to the original model. Hence, we can assure that any temporal logic property is preserved in a slice that is created with respect to that property, and consequently the verification on the slice is sound.

Keywords: Bisimulation, temporal logic, model checking, transition system, CTL*, slicing, Behavior Trees

1. Introduction

A bisimulation relation defines an equivalence between two transition systems. Two models are bisimilar if their observable behaviours are indistinguishable and thus satisfy the same properties. A variety of bisimulation relations have been defined in the literature and they vary in what is considered observable and what is not. What we mean by “observable” and the choice of which properties are considered are relative to the type of bisimulation relation. For example, strong bisimulation considers all steps to be observable whereas weak bisimulation relations regard silent or stuttering steps (i.e., steps that do not change (the relevant part of) the state of the model) as non-observable.

Strong bisimulation is known to preserve all temporal properties specified in the logic CTL*, the superset of the linear and branching time temporal logics, LTL and CTL [1]. However, the properties that are preserved through a weak bisimulation relation are only those temporal properties which do not contain the temporal next-step operator X (we refer to these as CTL*−X formulas). This is due to the fact that stuttering steps can affect the validity of formulas that contain the next-step operator. On the other hand, using a strong bisimulation is not always suitable, since some applications rely on the fact that stuttering steps can be neglected. An example of this is slicing [2], a technique in which a program or model is reduced by eliminating parts which are irrelevant according to a given criterion. The sliced model and the original are related only through weak bisimulation, since the irrelevant (or stuttering) steps do not exist in the slice. Thus, the request for a bisimulation that relates a reduced model to its original and guarantees

*Corresponding Author. Tel.: +61 (0)7 3365 1651. The concepts of this paper were developed while the first author was a PhD student at the Institute for Integrated and Intelligent Systems, Griffith University.

Email addresses: nisansala.yatapanage@griffithuni.edu.au (Nisansala Yatapanage), kirsten@itee.uq.edu.au (Kirsten Winter)

October 15, 2013
preservation of all temporal logic properties seems contradictory and, to our knowledge, none of the existing notions of bisimulation satisfy it.

In this paper we define a novel form of bisimulation, called next-preserving branching bisimulation, which is based on van Glabek and Weijland’s branching bisimulation [3]. The relation is stronger than weak bisimulation and weaker than strong bisimulation, and the validity of formulas with the next step operator is preserved. We give a proof that next-preserving branching bisimulation preserves full CTL*. These results are useful in any application that aims at a reduction of the model, such as slicing, e.g., [2, 4], or partial order reduction [5].

In our context the driving force behind the definition of the next-preserving branching bisimulation was the intention of improving the slicing algorithm used for temporal logic verification, namely model checking [6, 7]. The aim of slicing is to reduce (prior to model checking) a large model to a bisimilar slice which can be model checked instead of the original model, to combat the state explosion problem inherent to model checking. Based on our past experience in the domain of safety analysis, e.g., [8, 9, 10] we found that there are many cases in which the temporal logic formula of interest includes the next-step operator, and hence these formulas must be preserved by the slice. In the second part of this paper we introduce the resulting new slicing approach for the Behavior Tree modelling notation which is a graphical formal notation that supports the user in capturing informal requirements [11, 12]. Once a model has been derived from the requirements it can be analysed using a model checker. To guarantee soundness of the approach we need to ensure that the same temporal logic formulas are satisfied in the slice that are satisfied in the original model. As a vehicle to prove the soundness of our slicing approach, the notion of next-preserving branching bisimulation is used which also guarantees the preservation of next-step formulas. Our approach for deriving a next-preserving slicing algorithm for Behavior Trees, we believe, can also be used to derive similar slicing algorithms for other notations.

The paper is organised as follows. Section 2 introduces our novel next-preserving branching bisimulation relation and provides the required preservation results. In the following sections, these results are then applied in the context of slicing for Behavior Trees. In Section 3 we provide the definitions on which basic slicing of Behavior Trees are built. This summarises the results from [13] and gives the background. In Section 4 we demonstrate how the definition of the next-preserving branching bisimulation gives rise to a new sophisticated slicing procedure that generates a next-preserving bisimilar slice. Section 5 relates our results to other results in the literature and we conclude in Section 6.

2. Next-preserving Branching Bisimulation

This section introduces the novel next-preserving branching bisimulation relation, beginning with a brief introduction to transition systems, the temporal logic CTL* and existing bisimulation relations. Following this is the main contribution of the paper, namely the definition of next-preserving branching bisimulation and the proof that it preserves full CTL*.

2.1. Doubly-labelled Transition Systems

Bisimulations are equivalence relations, normally defined either between labelled transition systems, which have labels on transitions, or Kripke structures, which have labels on states. For our purposes, we have chosen to use doubly-labelled transition systems, as introduced by de Nicola and Vaandrager in [14], which are transition systems that contain labels on both edges and states. Our proofs apply to Kripke structures as well.

A doubly-labelled transition system is a tuple \((S, AP, I, L, A, \rightarrow)\) where \(S\) is a set of states, \(AP\) is a set of atomic propositions, \(I \subseteq S\) is a set of initial states, \(L\) is a state labelling function (i.e., \(L : S \rightarrow 2^{AP}\)), \(A\) is a set of actions, and \(\rightarrow : S \times A \times S\) is the transition relation. The notation \(s \xrightarrow{a} s'\) is used as a shorthand for \((s, a, s') \in \rightarrow\), but if the transition label is irrelevant it may be omitted.

A run in a doubly-labelled transition system is a sequence of states alternating with actions, starting and (if finite) ending at states, \(\rho = \langle s_0, n_1, s_1, n_2, s_2, \ldots \rangle\). A path on the other hand is a sequence of states, \(\pi = \langle s_0, s_1, s_2 \ldots \rangle\), which can be derived from a run by removing all actions from the sequence. If \(\pi\) is derived from \(\rho\) we write \(\pi = path(\rho)\) (note that the mapping path is a bijection). We denote the set of possible successor nodes of a state \(s\) with \(succ(s)\), i.e., \(s' = succ(s)\) implies there exists a path \(\pi = (s, s', \ldots)\). The notation \(\rho[s_i]\) denotes the prefix of the run \(\rho\) ending with (and including) \(s_i\), while \(\rho[s_i]\) returns the suffix of \(\rho\) starting at (and including) \(s_i\). The same notation will be used for paths. The concatenation of two paths \(p\) and \(q\) is denoted as \(p^+q\). The set of paths starting at a state \(s\) is denoted as \(paths(s)\). A path is said to be maximal if it either terminates or ends in a cycle.
2.2. Branching Time Computation Tree Logic, CTL*

In this paper, we consider the preservation of formulas of the logic CTL*, as introduced by Emerson and Halpern [15], who used it as a vehicle for the comparison between both sub-logics LTL and CTL. Our results are therefore applicable to the sub-logics as well.

CTL* includes the linear temporal operators (X and U), as well as the temporal operators of branching-time (E and A) of the sub-logics, but it does not constrain their combination. Let AP be the set of atomic propositions, then the syntax of CTL* is defined in the following manner, distinguishing state and path formulas in the usual way:

**Definition 1 (Syntax of CTL*).**

State formulas:

   i.) If $\varphi \in AP$, then $\varphi$ is a state formula.
   
   ii.) If $\varphi$ and $\psi$ are state formulas, then $\neg \varphi$ and $\varphi \land \psi$ are state formulas.
   
   iii.) If $\varphi$ is a path formula, then $E \varphi$ is a state formula.

Path formulas:

   i.) If $\varphi$ is a state formula, then $\varphi$ is also a path formula.
   
   ii.) If $\varphi$ and $\psi$ are path formulas, then $\neg \varphi$, $\varphi \land \psi$, $X \varphi$ and $\varphi U \psi$ are path formulas.

Additionally, the state operator $A$ is derived as follows: $A \varphi = \neg E \neg \varphi$. The path operators $F$ and $G$ are derived as follows: $F \varphi = true U \varphi$ and $G \varphi = \neg F (\neg \varphi)$. The $\lor$ (disjunction) operator is derived for both state and path formulas as $\varphi \lor \psi = \neg (\neg \varphi \land \neg \psi)$. The semantics of CTL* is defined below.

**Definition 2 (Semantics of CTL*).** Let $T$ be a doubly-labelled transition system such that $T = (S, AP, I, \mathcal{L}, A, \rightarrow)$. A CTL* state formula $\psi$ holds in a state $s \in S$, denoted $T, s \models \psi$, or simply $s \models \psi$ if $T$ is clear from the context, according to the following, where $\psi_1$ and $\psi_2$ are CTL* state formulas and $\varphi$ is a path formula:

   i.) $s \models true$.
   
   ii.) $s \models a$, where $a \in AP$, iff $a \in \mathcal{L}(s)$.
   
   iii.) $s \models \neg \psi_1$ iff $s \not\models \psi_1$.
   
   iv.) $s \models \psi_1 \land \psi_2$ iff $s \models \psi_1$ and $s \models \psi_2$.
   
   v.) $s \models E \varphi$ iff there exists a path $\pi = (s_0, s_1, \ldots)$, such that $s_0 = s$ and $\pi \models \varphi$.

A CTL* path formula $\varphi$ holds for a path $\pi = (s_0, s_1, \ldots)$, denoted $\pi \models \varphi$, according to the following, where $\varphi_1$ and $\varphi_2$ are CTL* path formulas and $\psi_1$ is a CTL* state formula:

   i.) $\pi \models \psi_1$ iff $s_0 \models \psi_1$.
   
   ii.) $\pi \models \neg \varphi_1$ iff $\pi \not\models \varphi_1$.
   
   iii.) $\pi \models \varphi_1 \land \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$.
   
   iv.) $\pi \models X \varphi_1$ iff $\pi[s_1] \models \varphi_1$.
   
   v.) $\pi \models \varphi_1 U \varphi_2$ iff $\exists j > 0$ such that $\pi[s_j] \models \varphi_2$ and $\forall i, 0 \leq i < j, \pi[s_i] \not\models \varphi_1$.

The transition system $T$ satisfies a CTL* formula $\psi$, denoted by $T \models \psi$, iff $T, s_i \models \psi$, for all $s_i \in I$.

We use the notation $vars(\varphi)$ to denote the set of variables referred to in the formula $\varphi$. When a transition with action $a$ modifies a variable in $vars(\varphi)$, it is known as an observable step, denoted as $obs_{\varphi}(a)$. All other transitions are referred to as stuttering steps (w.r.t. $\varphi$). We denote stuttering steps as $p \rightarrow_{\varphi} q$ or in short $p \rightarrow q$ if the dependency on formula $\varphi$ is obvious from the context. A path consisting entirely of stuttering steps is referred to as a stuttering path, denoted as $p \rightarrow_{\varphi}^n q$ if it consists of $n$ stuttering steps, or $p \rightarrow_{\varphi}^* q$ if it consists of an arbitrary number of stuttering steps.
2.3. Bisimulation

Bisimulation establishes equivalence between two systems. It can be divided into the strong and weak forms. The strong form requires that every step made by one transition system must be matched by an identical step in the other. On the other hand, the weak forms of bisimulation allow for the presence of stuttering steps, denoted by \( \tau \), which do not have to be matched by the other system. For many applications, these relaxed requirements are essential and weak bisimulations are the only possible form of equivalence which can be established. One form of weak bisimulation, known as branching bisimulation \([3]\), is particularly of interest as it distinguishes between two systems which perform the same observable steps but have different branching structures. This property is essential for relating it to branching-time logics.

CTL\(^*\) requires a variant of branching bisimulation known as divergence-sensitive branching bisimulation. Divergence refers to infinite stuttering paths. The divergence-sensitive variant of branching bisimulation distinguishes between systems that have or do not have divergent paths. This notion is necessary in order for CTL\(^*\) properties to be preserved, as CTL\(^*\) is normally defined over maximal paths. Divergent-sensitive branching bisimulation preserves CTL\(^*\) \([14]\). However, divergence-sensitive branching bisimulation as defined in \([14]\) does not directly incorporate the notion of divergence into the definition. Instead, an extra state is created in the transition system to represent divergence and all divergent states are linked to that new state. As noted in \([16]\), using this method, livelocked states cannot be distinguished from deadlocked states. Livelocked states are ones which have self loops, while deadlocked states are ones which have no outgoing transitions. Branching bisimulation with explicit divergence, proposed by Van Glabbeek and Weijland \([3]\), incorporates the divergence requirement into the definition itself, instead of making modifications to the transition system. The definition for branching bisimulation with explicit divergence is given in Definition 3, which is adapted to our context from the definition in \([16]\). Branching bisimulation with explicit divergence preserves CTL\(^*\) \([14]\). Our next-preserving branching bisimulation relation is based on branching bisimulation with explicit divergence.

**Definition 3.** Let \( T_i \) and \( T_j \) be doubly-labelled transition systems such that \( T_i = (S_i, AP_i, I_i, L_i, A_i, \rightarrow_i) \) for \( i \in \{1, 2\} \). A relation \( R \) is a branching bisimulation with explicit divergence with respect to a CTL\(^*\) formula \( \varphi \) iff for all \( s \in S_i \) and \( t \in S_2 \) such that \( s R t \), the following holds:

- for all \( s' \in S_1 \) such that \( s \xrightarrow{\varphi} s' \) we have \( s' R t \) and
- for all \( s' \in S_1 \) and \( a \in A_i \) with \( \text{obs}_\varphi(a) \), such that \( s \xrightarrow{a} s' \) there exists \( t', t'' \in S_2 \) such that \( t \xrightarrow{\varphi} t' \xrightarrow{\varphi} t'' \), \( s R t'' \) and \( s' R t' \) and
- if there exists an infinite stuttering path \( s \xrightarrow{\varphi} s_0 \xrightarrow{\varphi} \ldots \), such that \( s_i R t_i \) for all \( i \geq 0 \), then there exists an infinite stuttering path \( t \xrightarrow{\varphi} t_0 \xrightarrow{\varphi} \ldots \), such that \( s_i R t_j \) for all \( i, j \geq 0 \).

Two states \( s \) and \( t \) are branching bisimilar with explicit divergence, denoted \( s \equiv_\varphi t \), iff there exists a branching bisimulation with explicit divergence (w.r.t. \( \varphi \)), \( R \), such that \( s R t \).

2.4. Preserving the Next Step Operator

The *next step* operator, denoted by \( X \), is used in CTL\(^*\) and its sub-logics LTL and CTL for describing properties which hold at the next step. This allows one to specify a requirement that something will occur within a certain period of time, represented by a specific number of steps in the transition system. Although some authors argue against the use of the next step operator, e.g., \([18]\), it has many uses in practical applications. In particular, it is often used when specifying safety properties of the form \( p \Rightarrow X q \), to indicate that some action must follow immediately after another. The closest alternative would be to use the eventually operator \( (F) \), but for some requirements this is too weak, as it does not provide any guarantee of the period in which something will occur. Further supporting evidence for the usefulness of \( X \) is that it occurs in many of the commonly-used safety property patterns provided in \([19]\).

Unfortunately, the weak forms of bisimulation are unable to preserve properties containing the \( X \) operator since weak bisimilar systems may differ in the number of stuttering steps. This presents a difficulty when checking properties containing \( X \), because such properties require a distinct number of steps to be performed.
Nevertheless, it is possible to preserve properties containing $X$ while still maintaining a weak bisimulation. This result arises from the observation that only the stuttering steps occurring at certain locations can influence the value of a property, so these stuttering steps are the only ones which need to be present in both systems. In particular, the only stuttering steps necessary are those that occur immediately before an observable step or immediately before a branching point in the transition system, where an observable step is possible by selecting one path but is not possible within the same number of steps along another path. Furthermore, it is only necessary to preserve a small number of stuttering steps, bounded by the nesting depth of the $X$ operators present in the formula. Next-preserving branching bisimulation preserves properties with $X$ by ensuring that the essential stuttering steps are present in both transition systems. This form of bisimulation is stronger than branching bisimulation while still being a form of weak bisimulation, as most stuttering steps can still be ignored. The following two examples demonstrate cases where stuttering steps impact the validity of the formula.

![Diagram](image_url)

Figure 1: Two branching bisimilar transition systems with an observable step

**Example** Consider the two transition systems shown in Figure 1. Assume that $t_1$ is a stuttering step and $t_2$ is an observable step. The transition systems are branching bisimilar, as they both perform the same sequence of observable steps. Now, assume the property to be verified is $AX \neg p$, where $p$ is an atomic proposition. In $T_1$, $s_0 \Vdash AX \neg p$, but in $T_2$, $s_0 \nVdash AX \neg p$. It can be seen that the stuttering transition from $s_0$ to $s_1$ is necessary in order to satisfy the formula. Suppose that in $T_1$ there are an arbitrary number of stuttering steps between $s_0$ and $s_1$. It would still only be necessary for one of the stuttering steps to be present in $T_2$, instead of all of them. The number of stuttering steps required increases according to the number of nested $X$ operators in the formula. The validity of the property $AX \neg p$ changes only when the validity of $p$ is changed at the next step, which must be observable since $p$ is an atomic proposition.

This example demonstrates that stuttering steps which lie directly before observable steps influence the validity of a formula containing $X$. However, if $T_1$ contains fewer stuttering steps than the nesting depth of the formula, it is only necessary for $T_2$ to match the number of stuttering steps that $T_1$ has.

Although this example has used one particular property, most properties are influenced by observable transitions. The only exceptions are properties which change their validity depending on the existence of a particular path. This is demonstrated by the following example.
Example Consider the transition systems shown in Figure 2 and assume the formula is $\textbf{AXE}(Fp)$. In $T_1$, there are two possible paths from $s_0$: the path $\langle s_0, s_1, s_2, s_3 \rangle$ and the path $\langle s_0, s_1, s_4 \rangle$. Thus, the formula holds at state $s_0$, because on all of the paths from $s_0$, the next state is $s_1$, and from $s_1$ there exists a path on which $p$ is eventually true. However, in $T_2$, the formula does not hold at $s_0$. As in $T_1$ there are still two possible paths, but this time the next step after $s_0$ differs on each path. On one path the next state is $s_2$, from which there exists a path where eventually $p$ holds, but on the other path the next state is $s_4$, from which there is no possible way to reach a state where $p$ holds. This problem occurs when a state has multiple paths emanating from it, where $p$ holds on some of the paths but not on others. Note that the problem would not occur if both paths were still possible after one branch was chosen, for example, if $s_3$ and $s_4$ have outgoing edges looping back to $s_0$.

![Figure 2: Two branching bisimilar transition systems with an observable step on one path only](image)

This example illustrated another location at which stuttering nodes may be required, namely where there is a branching point such that on one path an observable step occurs and on the other path the observable step does not occur within the same number of steps. This concept is represented by the condition $\text{diffPaths}$, defined below.

**Definition 4 (diffPaths).** Let $T$ be a doubly-labelled transition system such that $T = (S, AP, I, L, A, \rightarrow)$. For a state $s \in S$ and a CTL* path formula $\phi$, $\text{diffPaths}_\phi(s)$ iff:

\[
\exists \rho_1 = \langle s, a_1, s_1, a_2, \ldots \rangle, \exists s_k \in \text{succ}(s) \text{ such that } s_1 \neq s_k, \exists j \geq 1 \text{ and } \exists m \geq 0, \text{ where either:}
\]

- $\exists a_i \in \rho_1[\langle s_j \rangle] \text{ and } \text{obs}_\phi(a_i), \text{ where } \forall \rho_2 = \langle s, a_k, s_k, \ldots \rangle, a_i \notin \rho_2[\langle s_{k+j} \rangle], \text{ or}$

- $\forall \rho_2 = \langle s, a_k, s_k, \ldots \rangle, \exists a_i \in \rho_2[\langle s_{k+j} \rangle] \text{ where } \text{obs}_\phi(a_i) \text{ and } a_i \notin \rho_1[\langle s_j \rangle]$.

The number of $X$ operators in a formula is referred to as the $\text{xdepth}$ of the formula, which is inductively defined as follows (a similar definition for LTL formulas is given in [20]).

**Definition 5 (xdepth).** The $\text{xdepth}$ of a CTL* formula is given as follows, where $\psi_1$ and $\psi_2$ are state or path formulas and $\varphi$ and $\varphi_1$ and $\varphi_2$ are path formulas:

\[
\text{xdepth}(\varphi) = 0, \text{ where } \varphi \in \text{AP},
\]

\[
\text{xdepth}(\psi_1 \land \psi_2) = \max(\text{xdepth}(\psi_1), \text{xdepth}(\psi_2)),
\]

\[
\text{xdepth}(\lnot \psi_1) = \text{xdepth}(\psi_1),
\]

\[
\text{xdepth}(E \varphi) = \text{xdepth}(\varphi),
\]

\[
\text{xdepth}(\phi_1 U \phi_2) = \max(\text{xdepth}(\phi_1), \text{xdepth}(\phi_2)),
\]

\[
\text{xdepth}(X \varphi) = \text{xdepth}(\varphi) + 1.
\]

Using the notions of observable steps, $\text{diffPaths}$ locations, and the $\text{xdepth}$ of a formula, we define next-preserving branching bisimulation over states, paths and transition systems as follows.
**Definition 6** (Next-preserving Branching Bisimulation over states). Let $T_1$, $T_2$ be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, L_i, A_i, \rightarrow_i)$, for $i \in \{1, 2\}$. A relation $\mathcal{R}$ is a next-preserving branching bisimulation with respect to a CTL* formula $\phi$ iff

- $\mathcal{R}$ is a branching bisimulation with explicit divergence and
- for all $s \in S_1$ and $t \in S_2$ such that $s \mathcal{R} t$,
  - if $\exists s_{j-1}, s_j \in S_1$ such that $s \rightarrow^{j-1}_{j-1} \stackrel{a}{\rightarrow} s_j$ and either:
    - $\text{obs}_\phi(a)$ or
    - $\text{diffPaths}_\phi(s_{j-1})$, then:
      $\exists t_{m-1}, t_m \in S_2$ such that $t \rightarrow^k_{m-1} \stackrel{a}{\rightarrow} t_m$, where $k \geq \min(x\text{depth}(\phi), j-1)$ and $s_j \mathcal{R} t_m$.

Two states $s$ and $t$ are next-preserving branching bisimilar, with respect to a CTL* formula $\phi$, denoted $s \text{~}_{\mathcal{R}} \phi \text{~} t$, iff there exists a next-preserving branching bisimulation $\mathcal{R}$ with respect to $\phi$ such that $s \mathcal{R} t$.

**Definition 7** (Next-preserving Branching Bisimulation over paths). A path $\pi_1$ is next-preserving branching bisimilar, with respect to a formula $\phi$, to a path $\pi_2$, denoted $\pi_1 \text{~}_{\mathcal{R}} \phi \text{~} \pi_2$, iff for every state $s_i \in \pi_1$, there exists a state $t_i \in \pi_2$ such that $s_i \text{~}_{\mathcal{R}} \phi \text{~} t_i$ and vice versa.

**Definition 8** (Next-preserving Branching Bisimulation of transition systems). Let $T_1$, $T_2$ be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, L_i, A_i, \rightarrow_i)$, for $i \in \{1, 2\}$. $T_1$ and $T_2$ are next-preserving branching bisimilar, with respect to a formula $\phi$, denoted $T_1 \text{~}_{\mathcal{R}} \phi \text{~} T_2$, iff $s_0 \text{~}_{\mathcal{R}} \phi \text{~} t_0$ for all $s_0 \in I_1$ and $t_0 \in I_2$.

**2.5. Preservation of CTL***

In order to show the preservation of CTL* by next-preserving branching bisimulation, some intermediate results must first be established. The following lemma shows that if two transition systems are next-preserving branching bisimilar, for every path in one system there is guaranteed to be a next-preserving branching bisimilar path in the other system.

**Lemma 9.** Let $T_1, T_2$ be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, L_i, A_i, \rightarrow_i)$, for $i \in \{1, 2\}$. Then $T_1 \text{~}_{\mathcal{R}} \phi \text{~} T_2$ implies $\forall \pi_1 \in T_1$, $\exists \pi_2 \in T_2$ such that $\pi_1 \text{~}_{\mathcal{R}} \phi \text{~} \pi_2$.

**Proof.** This lemma can be established using induction over the length $m$ of the path $\pi_1 = (s_0, s_1, ..., s_{m-1})$. The base case where $m = 1$ is shown by demonstrating the existence of an initial state in $I_2$ for every initial state in $I_1$. From this, the inductive step is derived from Definition 6. For the full details of the proof, we refer the reader to [21].

A useful observation is that at a branching point $s_0$, when there exists a path from $s_0$ on which a formula is satisfied, while no such path is reachable from one of the successor states of $s_0$, then the condition $\text{diffpaths}$ must hold. That is, there must be an observable step on one path from $s_0$ which does not exist on the other path from $s_0$ within the same number of steps. This situation is depicted in Figure 3. If a sub-formula $\phi$ is satisfied by a state $s_j$ on the path starting at $s_1$, but not by the state $s_{k+j}$ on the other path, then $s_j$ must satisfy $\text{diffpaths}$. This implication is formalised in Lemma 10. For simple formulas containing only atomic propositions, it is easy to see that there must be an observable step on one path but not the other. As the Lemma shows inductively, more complex formulas also change their validity based on the underlying atomic propositions.

This result is necessary for preserving properties of the form $E \phi$. At state $s_0$, there are two possible paths, but at $s_1$, one path is no longer reachable. A formula $E \phi$ would be satisfied at $s_0$ but not at $s_1$. Thus, it is useful to know that such a situation would only occur on states satisfying $\text{diffpaths}$. The result is utilised in the subsequent proofs.
Let $T$ be a doubly-labelled transition system such that $T = (S, AP, I, L, A, \rightarrow)$. For a state $s_0 \in S$ and a $\text{CTL}^*$ path formula $\varphi$, if there exists a run $\rho_1 = (s_0, a_1, s_1, a_2, s_2 \ldots)$ such that for some $j \geq 0$, $\text{path}(\rho_1)[s_j] \models \varphi$ and there exists a state $s_k \in \text{succ}(s_0)$ such that $s_k \neq s_1$ and $\forall \rho_2 = (s_0, a_k, s_k, a_{k+1}, s_{k+1} \ldots)$, $\text{path}(\rho_2)[s_{k+j}] \not\models \varphi$, then $\text{diffPaths}_{\varphi}(s_0)$.

**Proof.** By structural induction over $\varphi$.

In the following, let $\varphi, \varphi_1$ and $\varphi_2$ be path formulas and $\psi, \psi_1$ and $\psi_2$ be state formulas and assume that the lemma holds for $\psi_1, \psi_2, \varphi_1$ and $\varphi_2$. Let $\rho_1 = (s_0, a_1, s_1, a_2, s_2 \ldots)$, $\rho_2 = (s_0, a_k, s_k, a_{k+1}, s_{k+1} \ldots)$, $\text{path}(\rho_1) = \pi_i$ for any $i$, and $\pi_1[s_j] \models \varphi$ and $\pi_2[s_{k+j}] \not\models \varphi$.

If $\varphi$ is a state formula: $s_j \models \varphi$ and $s_{k+j} \not\models \varphi$.

$\varphi \in AP$:

Assume that $s_0 \models \varphi$. Since the run $\rho_1$ and all runs $\rho_2$ begin in $s_0$ and $s_j \models \varphi$ but $s_{k+j} \not\models \varphi$, there must be a step $\alpha \in \rho_2[s_{k+j}]$ such that $\text{obs}_\varphi(\alpha)$, which makes the formula $\varphi$ false. Either $\alpha \not\models \rho_1[s_j]$ or there must be another observable step $\beta \in \rho_1[s_j]$ which makes $\varphi$ true and $\beta \not\models \rho_2[s_{k+j}]$. In either case, the conditions for $\text{diffpaths}$ are satisfied, so $\text{diffpaths}_{\psi}(s_0)$ holds.

If $s_0 \not\models \varphi$ we use the inverse reasoning, i.e., there must be a step $\alpha \in \rho_1[s_j]$ such that $\text{obs}_\varphi(\alpha)$, which makes the formula $\varphi$ true.

$\varphi = \psi_1 \land \psi_2$:

$s_j \models \psi_1 \land \psi_2$, so $s_j \models \psi_1$ and $s_j \models \psi_2$. Since $s_{k+j} \not\models \psi_1 \land \psi_2$, either $s_{k+j} \not\models \psi_1$ or $s_{k+j} \not\models \psi_2$. Thus, either $s_j \models \psi_1$ and $s_{k+j} \not\models \psi_1$ or the same for $\psi_2$, so by induction, $\text{diffpaths}_{\varphi}(s_0)$ holds.

$\varphi = \lnot \psi_1$:

The proof for this case is trivial. For the full details we refer the reader to [21].

$\varphi = E \varphi_1$:

Since $s_j \models E \varphi_1$, there exists a path $\pi_3$ starting from $s_j$, such that $\pi_3 \models \varphi_1$. Since $s_{k+j} \not\models E \varphi_1$, for all paths $\pi_4$ starting at $s_{k+j}$, $\pi_4 \not\models \varphi_1$. Therefore, there exists a path $\pi_5 = (s_0, s_1, \ldots, s_{j-1}) \pi_3$ such that $\pi_5[s_j] \models \varphi_1$ and a path $\pi_6 = (s_0, s_k, \ldots, s_{k+j-1}) \pi_4$ for every $\pi_4$ starting at $s_{k+j}$, such that $\pi_6[s_{k+j}] \not\models \varphi_1$. Therefore, from the induction assumption, $\text{diffPaths}_{\psi_1}(s_0)$ holds. Since all nodes that are observable with respect to $\varphi_1$ are observable with respect to $E \varphi_1$, $\text{diffPaths}_{\varphi}(s_0)$ holds.

If $\varphi$ is a path formula:

$\varphi = \varphi_1 \land \varphi_2$:

This case is proved in a similar fashion to the $\psi_1 \land \psi_2$ case above. For the full details we refer the reader to [21].

$\varphi = \lnot \varphi_1$:

The proof for this case is trivial. For the full details we refer the reader to [21].
\[ \varphi = X \varphi_1: \]
\[ \pi_1(s_1) \models X \varphi_1 \text{ and } \pi_2(s_{k+1}) \not\equiv X \varphi_1 \implies \pi_1(s_{j+1}) \models \varphi_1 \text{ and } \pi_2(s_{k+j+1}) \not\equiv \varphi_1. \]
By induction using the formula \( \varphi_1 \), \( \text{diffPaths}_{\varphi_2}(s_0) \) holds. Since all nodes that are observable with respect to \( \varphi_1 \) are observable with respect to \( X \varphi_1, \text{diffPaths}_{\varphi}(s_0) \) holds.

\[ \varphi = \varphi_1 U \varphi_2: \]
Since \( \pi_1(s_1) \models \varphi_1 \ U \varphi_2, 3 \nu \) such that \( \nu \geq j \) and \( \pi_1(s_{nu}) \not\equiv \varphi_1 \). Additionally, for all \( w \) such that \( j \leq w < v, \)
\[ \pi_1(s_{wu}) \not\equiv \varphi_1. \]
Since \( \pi_2(s_{k+y}) \not\equiv \varphi_1 \ U \varphi_2, \) there are two possibilities. The first possibility is that there exists a \( y \)
such that \( y \geq j \) and \( \pi_2(s_{k+y}) \not\equiv \varphi_2 \), and for all \( j < y' < y, \)
\[ \pi_2(s_{k+y'}) \not\equiv \varphi_2 \] (i.e., \( s_{k+y} \) is the first state along \( \pi_2(s_{k+y}) \)) that satisfies \( \varphi_2 \), and there exists an \( x \) such that \( j \leq x < y \) and \( \pi_2(s_{k+x}) \not\equiv \varphi_1 \). The second possibility is
that there does not exist a \( y \) such that \( y \geq j \) and \( \pi_2(s_{k+y}) \not\equiv \varphi_2 \).
Assume the first case holds. Then, \( \forall w \) such that \( j \leq w < v, \) it holds that \( \pi_1(s_{wu}) \models \varphi_1 \). If \( y \leq v, \) then since
\[ j \leq x < y, \] it holds that \( \pi_1(s_{x}) \models \varphi_1. \]
Since \( \pi_2(s_{k+x}) \not\equiv \varphi_1 \), this implies \( \text{diffPaths}_{\varphi_2}(s_0) \). Otherwise, if \( y > v, \)
then \( \pi_1(s_{nu}) \not\equiv \varphi_2 \) but \( \pi_2(s_{k+y}) \not\equiv \varphi_2 \), so \( \text{diffPaths}_{\varphi_2}(s_0) \). Since all nodes that are observable with respect to \( \varphi_1 \) or \( \varphi_2 \) are observable with respect to \( \varphi_1 U \varphi_2, \text{diffPaths}_{\varphi}(s_0) \) holds.
Assume the second case holds. Then, \( \pi_1(s_{nu}) \models \varphi_2 \) but since there does not exist a \( y \) such that \( y \geq j \), where
\[ \pi_2(s_{k+y}) \not\equiv \varphi_2 \], then \( \pi_2(s_{k+y}) \not\equiv \varphi_2 \), so \( \text{diffPaths}_{\varphi_2}(s_0) \). Since all nodes that are observable with respect to \( \varphi_2 \) are observable with respect to \( \varphi_1 U \varphi_2, \text{diffPaths}_{\varphi}(s_0) \) holds.

For formulas with \( X \) to be preserved by two paths, sub-formulas must only change their validity after the same number of steps on both paths. Next-preserving branching bisimulation between paths guarantees this by ensuring that the correct number of stuttering steps are present whenever a sub-formula changes its validity from one state to the next. This can only occur due to an observable step changing the underlying atomic propositions in the formula, or at a \( \text{diffPaths} \) state. Assume there are two next-preserving bisimilar paths \( \pi_1 \) and \( \pi_2 \), as depicted in Figure 4. If the step from \( s_{n-1} \) to \( s_n \) is observable or \( s_{n-1} \) is a \( \text{diffPaths} \) state, then there must be a minimum of \( \text{xdepth}(\varphi) \) or \( n-1 \) steps in the second path before the state \( t_{m-1} \), according to Definition 6. However, if the sub-formula \( \varphi_1 \) itself contains one or more \( X \) operators then the crucial observable step or \( \text{diffPaths} \) state, which is responsible for the change in validity of the formula, lies further along on the path. In this case we relate the number of stuttering steps, required before the state where the validity of the sub-formula changes, to the difference between the \( \text{xdepth} \) of the formula and the \( \text{xdepth} \) of the sub-formula.

For instance, suppose that there are two stuttering steps between \( s_0 \) and \( s_{n-1} \) and that \( \varphi = X X \varphi_1 \) with \( \varphi_1 = X p \). Assume that \( s_{n-1} \models \varphi_1 \) but \( s_n \not\models \varphi_1 \). Then, the observable step would be between \( s_n \) and \( s_{n+1} \) and \( s_{n+1} \models p \), while \( s_{n+2} \not\models p \). In this case, from Definition 6, there would be at least 3 \( \text{xdepth}(\varphi) = 3 \) since \( \varphi = X X \varphi_1 \) or \( n \) steps before \( t_m \). From the state before which \( \varphi_1 \) changes, i.e. \( s_{n-1} \), there are guaranteed to be at least \( m-1 \) or 2 steps (the difference between \( \text{xdepth}(\varphi_1) \) and \( \text{xdepth}(\varphi) \)) before \( t_{m-1} \). This holds in general: at any state before a sub-formula \( \varphi_1 \) changes its validity, there are guaranteed to be \( \text{xdepth}(\varphi) - \text{xdepth}(\varphi_1) \) stuttering steps before its equivalent state on the other path. This result arises inductively from Definition 6, as shown by Lemma 11.

**Figure 4**: The two paths for Lemma 11

**Lemma 11.** For a path formula \( \varphi \) and two paths \( \pi_1 = (s_0, s_1, \ldots) \) and \( \pi_2 = (t_0, t_1, \ldots) \), where \( \pi_1 =_\prec \pi_2 \) and
\[ \exists s_n, t_m \text{ such that } s_n =_\prec t_m \text{ and } s_n \rightarrow^{n-1} s_{n-1} \text{ and for some formula } \varphi_1, \text{ which is a sub-formula of } \varphi, \text{ if } \pi_1(s_{n-1}) \equiv \varphi_1 \text{ and } \pi_1(s_n) \not\equiv \varphi_1, \text{ where } n > 1, \]
then \( t_0 \rightarrow^k t_{m-1} \), where \( k \geq \min(\text{xdepth}(\varphi) - \text{xdepth}(\varphi_1), n - 1) \).
Assume in the following that \( s \) with respect to \( \phi \) formulas \( \psi_1 \) and \( \psi_2 \).

Proof. By structural induction over \( \phi \). Assume the lemma holds for the state formulas \( \psi_1 \) and \( \psi_2 \) and the path formulas \( \phi_2 \) and \( \phi_3 \).

Consider the case where the step from \( s_{n-1} \) to \( s_n \) is observable with respect to \( \phi \), or \( \text{diffpaths}_{\phi}(s_{n-1}) \). Then, since \( s_n \models \phi \), by Definition 6, \( t_0 \rightarrow s^k \ t_{m-1} \), where \( k \geq \min(xdepth(\phi), n - 1) \). Since \( xdepth(\phi) \geq xdepth(\phi_1) \), we know that \( k \geq \min(xdepth(\phi) - xdepth(\phi_1), n - 1) \), as required.

Otherwise, if the step from \( s_{n-1} \) to \( s_n \) is stuttering with respect to \( \phi \), then we reason in the following way.

Assume in the following that \( \phi_1 \) is a state formula.

\( \phi_1 \in AP \):

The step from \( s_{n-1} \) to \( s_n \) must be observable with respect to \( \phi_1 \), since \( \phi_1 \) becomes false. Therefore, it is observable with respect to \( \phi \) and the same reasoning as above holds.

\( \phi_1 = \psi_1 \land \psi_2 \):

This case is trivial, using similar reasoning to the conjunction cases in the previous proof. For the full details we refer the reader to [21].

\( \phi_1 = \neg \psi_1 \):

The proof for this case is trivial. For the full details we refer the reader to [21].

\( \phi_1 = E \phi_2 \):

Since \( s_{n-1} \models E \phi_2 \) but \( s_n \not\models E \phi_2 \), there are two cases. The first case is that there exists a path \( \pi_3 \in \text{paths}(s_{n-1}) \) such that \( \pi_3 \models \phi_2 \) but \( \forall \pi_4 \in \text{paths}(s_n) \), \( \pi_4 \not\models \phi_2 \). Applying Lemma 10 to the run corresponding to path \( \pi_3 \), using \( s_{n-1} \) as the \( s_0 \) in Lemma 10 and with \( j = 0 \), gives \( \text{diffpaths}_{\phi_2}(s_{n-1}) \). Therefore, since \( s_n \models \phi \), by Definition 6, \( t_0 \rightarrow^k t_{m-1} \), where \( k \geq \min(xdepth(\phi), n - 1) \). Since \( xdepth(\phi) \geq xdepth(\phi_1) \), then \( xdepth(\phi) \geq xdepth(\phi_1) \), so \( k \geq \min(xdepth(\phi) - xdepth(\phi_1), n - 1) \), as required.

The second case is that \( s_{n-1} \models \phi_2 \) but \( s_n \not\models \phi_2 \). From induction using \( \phi_2 \), \( t_0 \rightarrow^k t_{m-1} \), where \( k \geq \min(xdepth(\phi) - xdepth(\phi_1), n - 1) \), as required.

Assume in the following that \( \phi_1 \) is a path formula.

\( \phi_1 = \phi_2 \land \phi_3 \):

This case is trivial, using similar reasoning to the conjunction cases in the previous proof. For the full details we refer the reader to [21].

\( \phi_1 = \neg \phi_2 \):

The proof for this case is trivial. For the full details we refer the reader to [21].

\( \phi_1 = X \phi_2 \):

The step from \( s_{n-1} \) to \( s_n \) is not observable by assumption, so \( s_0 \rightarrow^* s_n \). Furthermore, \( \pi_1[s_{n-1}] \models X \phi_2 \) but \( \pi_1[s_n] \not\models X \phi_2 \), which implies that \( \pi_1[s_{n-1}] \models \phi_2 \) and \( \pi_1[s_{n+1}] \not\models \phi_2 \). Using this and \( s_n \models t_m \) along with the induction assumption that this lemma holds for \( \phi_2 \), gives \( t_0 \rightarrow^r t_m \), where \( r \geq \min(xdepth(\phi) - xdepth(\phi_2), n) \).

Since \( xdepth(\phi_1) = xdepth(\phi_2) + 1 \), the statement \( xdepth(\phi) = xdepth(\phi_2) + 1 \) can be reduced to \( xdepth(\phi) = xdepth(\phi_2) + 1 \). Thus, \( t_0 \rightarrow^k t_{m-1} \), where \( k \geq \min(xdepth(\phi) - xdepth(\phi_1), n - 1) \).

\( \phi_1 = \phi_2 U \phi_3 \):

\( \pi_1[s_{n-1}] \models \phi_2 U \phi_3 \) but \( \pi_1[s_n] \not\models \phi_2 U \phi_3 \). Since \( \pi_1[s_{n-1}] \models \phi_2 U \phi_3 \), there must be a state \( s_v \in \pi_1[s_{n-1}] \) such that for all \( i \), where \( n - 1 \leq i < v \), \( \pi_1[s_i] \models \phi_2 \) and \( \pi_1[s_n] \models \phi_3 \). However, since \( \pi_1[s_n] \not\models \phi_2 U \phi_3 \), the state \( s_v \) cannot be at or after \( s_n \), as otherwise \( s_v \) would satisfy the formula. Therefore, \( s_v \) must be \( s_{n-1} \), so \( s_{n-1} \models \phi_3 \) and \( s_n \not\models \phi_3 \). Using this and the induction assumption that this lemma holds for \( \phi_3 \) gives \( t_0 \rightarrow^k t_{m-1} \), where \( k \geq \min(xdepth(\phi) - xdepth(\phi_3), n - 1) \). Since \( xdepth(\phi_1) \geq xdepth(\phi_3) \) it follows that \( k \geq \min(xdepth(\phi) - xdepth(\phi_1), n - 1) \), as required.

\( \square \)
Theorem 12 is the main result of this section, which demonstrates that two transition systems which are next-preserving branching bisimilar preserve the same \( \text{CTL}^* \) formulas. The proof is divided into two sections: a proof that (i) if two states are next-preserving branching bisimilar, they preserve \( \text{CTL}^* \) state formulas and (ii) if two paths are next-preserving branching bisimilar, they preserve \( \text{CTL}^* \) path formulas. Recall that one of the \( \text{CTL}^* \) state formulas is \( E \varphi \), where \( \varphi \) is a path formula. This is where the result of Lemma 9 is used, to show that there will always exist a matching path in the other system. The second section, concerning the proof of the preservation of path formulas, is where the \( \textbf{X} \) operator is examined, since it is a path operator. This section utilises the results of Lemma 11, as explained previously.

**Theorem 12 (Next-Preserving Branching Bisimulation preserves full \( \text{CTL}^* \)).** For two doubly-labelled transition systems \( T_1 \) and \( T_2 \) such that \( T_i = (S_i, AP_i, I_i, L_i, A_i, \rightarrow_{i}) \), for \( i \in \{1, 2\} \), and for all \( \psi \in \text{CTL}^* \),

\[
T_1 \models_{\psi} T_2 \Rightarrow (T_1 \models \psi \iff T_2 \models \psi).
\]

**Proof.** Assume \( T_1 \models_{\psi} T_2 \). For this, it is required to show that:

(i) For all state formulas \( \psi \in \text{CTL}^* \) and states \( s_i \in S_1 \) and \( t_i \in S_2 \), \( s_i \models_{\psi} t_i \Rightarrow (s_i \models \psi \Leftrightarrow t_i \models \psi) \) and

(ii) For all path formulas \( \varphi \in \text{CTL}^* \) and paths \( \pi_1 \) and \( \pi_2 \), \( \pi_1 \models_{\varphi} \pi_2 \Rightarrow (\pi_1 \models \varphi \Leftrightarrow \pi_2 \models \varphi) \).

Assume that statement (i) holds for the state formulas \( \psi_1 \) and \( \psi_2 \) and statement (ii) holds for the path formulas \( \varphi_1 \) and \( \varphi_2 \).

(i) For state formulas:

\[
\psi = \psi_1 \land \psi_2:
\]

Since next-preserving branching bisimulation implies branching bisimilarity with explicit divergence (Definition 6), \( \psi \) is preserved by \( \equiv \).

\[
T_1 \models \psi \iff s_0 \models \psi_1 \text{ and } s_0 \models \psi_2.
\]

Since the sets of observable steps with respect to \( \psi_1 \) and \( \psi_2 \) are subsets of the set of observable steps with respect to \( \psi \), by Definition 6, \( \equiv_\psi \) implies \( \equiv_{\psi_1} \) and \( \equiv_{\psi_2} \). Therefore, if \( s_0 \models \psi_1 \), then \( t_0 \models \psi_1 \) and similarly for \( \psi_2 \). From this it follows that \( t_0 \models \psi_1 \land \psi_2 \), and hence \( T_2 \models \psi \).

\[
\psi = \neg \psi_1:
\]

This case holds using a similar reasoning to the previous case. For the full details we refer the reader to [21].

\[
\psi = E \varphi_1, \text{ for some path formula } \varphi_1:
\]

\[
T_1 \models \psi \iff \text{there exists a path } \pi_1 \in \text{path} \{s_0\} \text{ such that } \pi_1 \models \varphi_1.
\]

Since the set of observable steps with respect to \( \varphi_1 \) is the same as the set of the observable steps with respect to \( \psi \), by Definition 6, \( \equiv_\psi \) implies \( \equiv_{\varphi_1} \). With Lemma 9 it follows that there exists a path \( \pi_2 \in \text{path} \{t_0\} \) such that \( \pi_2 \models \varphi_1 \), which is equivalent to \( T_2 \models E \varphi_1 \).

(ii) For path formulas:

Let \( \pi_1 = \langle s_0, s_1, \ldots \rangle \) and \( \pi_2 = \langle t_0, t_1, \ldots \rangle \) and assume wlog. that \( \pi_1 \models \varphi \).

\[
\varphi = \varphi_1 \land \varphi_2:
\]

The proof for this case is similar to the \( \psi_1 \land \psi_2 \) case above, using the fact that \( \equiv_\varphi \) implies \( \equiv_{\varphi_1} \) and \( \equiv_{\varphi_2} \). From the induction assumption, this gives \( \pi_1 \models \varphi_1 \Leftrightarrow \pi_2 \models \varphi_1 \) and similarly for \( \varphi_2 \). Therefore, \( \pi_2 \models \varphi_1 \land \varphi_2 \).

\[
\varphi = \neg \varphi_1:
\]

The proof for this case is similar to the previous case. For the full details we refer the reader to [21].
\[ \varphi = \varphi_1 \cup \varphi_2. \]

Since the sets of observable steps with respect to \( \varphi_1 \) and \( \varphi_2 \) are subsets of the set of observable steps with respect to \( \varphi \), by Definition 6, \( \equiv_{\varphi} \) implies \( \equiv_{\varphi_1} \) and \( \equiv_{\varphi_2} \). Since \( \pi_1 \models \varphi_1 \cup \varphi_2 \), there must exist a state \( s_v \) on path \( \pi_1 \) such that all the states up to \( s_v \) satisfy \( \varphi_1 \) and \( s_v \) \( \models \varphi_2 \). Since \( \pi_1 \equiv_{\varphi_2} \pi_2 \), there must be a state \( t_w \) on path \( \pi_2 \) such that \( s_v \equiv_{\varphi_2} t_w \) and \( \pi_2[t_w] \models \varphi_2 \). For all steps \( s_i \) before \( s_v \), there must be a \( t_6 \) on path \( \pi_2 \) such that \( s_i \equiv_{\varphi_1} t_6 \) and \( \pi_2[t_6] \models \varphi_1 \). Therefore, \( \pi_2 \models \varphi_1 \cup \varphi_2 \).

\[ \varphi = X \varphi_1: \]

Proof by contradiction. Assume that \( \pi_1 \models X \varphi_1 \) and \( \pi_2 \not\models X \varphi_1 \), so \( \pi_1[s_1] \not\models \varphi_1 \) and \( \pi_2[t_1] \not\models \varphi_1 \). By the assumption, \( \pi_1 \equiv_{\varphi} \pi_2 \). Since for any step \( a \) we know that \( \text{obs}_{\varphi_1}(a) \Leftrightarrow \text{obs}_{\varphi}(a) \) (i.e., the set of observable steps w.r.t. \( \varphi_1 \) is identical to the observable steps w.r.t. \( \varphi \)), we know by Definition 6 that \( \equiv_{\varphi} \) implies \( \equiv_{\varphi_1} \). Thus, there must be a state \( s_j \) on path \( \pi_1 \) with \( s_j \equiv_{\varphi_1} t_1 \). With the assumption that Theorem 12 holds for the subformula \( \varphi_1 \) we know that \( \pi_1[s_j] \not\models \varphi_1 \) and that \( s_j \) is the first state after \( s_1 \) in which \( \varphi_1 \) does not hold, i.e., \( \pi_1[s_{j-1}] \models \varphi_1 \). This means there will be zero or more stuttering steps from \( s_0 \) to \( s_{j-1} \) and then an observable step to \( s_j \), i.e., there exists \( j \geq 1 \) and a step \( a \) and such that \( s_0 \xrightarrow{\varphi_1} s_{j-1} \xrightarrow{a} s_j \) with \( \text{obs}_{\varphi_1}(a) \). Using Lemma 11 there must be \( k \) stuttering steps before \( t_0 \) such that \( k \geq \min(x\text{depth}(\varphi) - x\text{depth}(\varphi_1), j-1) \). Since \( \varphi = X \varphi_1 \), \( x\text{depth}(\varphi) - x\text{depth}(\varphi_1) = 1 \), and \( x \geq \min(1, j-1) \). Since \( t_0 \) is the first state in \( \pi_2 \) we also know that \( k = 0 \). Therefore \( j = 1 \) and \( s_j = s_1 \) with \( \pi_1[s_1] \models \varphi_1 \) which contradicts the assumption that \( \pi_1[s_j] \not\models \varphi_1 \).

This concludes the preservation result for next-preserving branching bisimilar systems. In the following sections we use this new notion of bisimulation to derive a novel slicing algorithm (for Behavior Trees) which creates next-preserving branching bisimilar slices. With the result above this slicing approach is sound for all \( \text{CTL}^* \) formulas.

3. Slicing Behavior Tree models

Slicing is a technique for reducing the size of a model, which may be a program or any (formal) description of a system’s behaviour [2]. It has been developed for various contexts, such as debugging and understanding of programs and for supporting program analysis. The reduced artifact, called a slice, can be computed algorithmically with little computational effort, even for very large models. The technique utilises flow information of the model and dependencies between its elements to eliminate parts which are not relevant to a specific criteria, called the slicing criteria. Slicing has been defined for various programming languages (for an overview see [22, 23]) and lately also for a number of formal modelling notations (e.g., [24, 25, 26, 27, 28, 29, 30, 31, 32, 33]). Slicing of a formal model is of particular interest if an automated analysis of the model is intended, using for example a model checker. Slicing as a reduction technique pushes the limits of a model checker, allowing larger models to be automatically analysed.

In [13] we have defined standard slicing for Behavior Tree models which preserves \( \text{LTL}_{\neg X} \) formulas (i.e., linear temporal formulas that do not contain the \( X \) operator). In this section we briefly introduce the Behavior Tree notation and the basic concepts of standard slicing for this language.

3.1. The Behavior Tree Notation

The Behavior Tree\(^1\) (BT) notation [34, 11] is a graphical notation to capture the functional requirements of a system provided in natural language. The semantics of the notation has been formalised in [12] using an extension of CSP, called \( \text{CSP}_\pi \) [35]. The novel (and from a modeller’s perspective appealing) idea of BTs is that the notation allows the user to merge sub-trees on a graphical level, and hence supports the integration of single requirements (each modelled by a sub-tree) into a full model. A formal discussion on the integration process can be found in [36].

Details on the syntax of the notation and its well-formedness conditions can be found in [37]. Here we simply provide an example to give the reader a flavour of the notation. In Figure 5 an integrated tree is shown modelling the behaviour of a light switch system which can be operated manually or switched off by a timer.

\(^{1}\)Geoff Dromey introduced the name Behavior Trees and chose the American spelling. Although this manuscript is written following the Australian spelling rules we respect Dromey’s naming decision and deliberately sacrifice consistency at this point.
Integration. The example is fictive but we assume that the system’s requirements were given as six individual requirements (labelled as R1 to R6), each of which is modelled as a sub-tree (the labels in the nodes of each sub-tree refer to the corresponding requirement label). These sub-trees are then merged at nodes they have in common and hence form a single (integrated) tree, modelling the system as a whole.

Components. The system is comprised of five components (Power, Switch, Light, Timer, and User) and receives external input from an unspecified environment. The behaviour of each component is captured via nodes and their types (specified as bracketing symbols, e.g., < >) such that a sequence of nodes can refer to different components. This way the model can be captured closer to common functional requirements given in natural language.

Branching Structure. Our example model consists of three main threads operating concurrently: the Switch thread (on the left in Fig. 5), the Timer thread (in the middle), and the User thread (on the right). The Switch thread models three alternative behaviours of which only one is executed: either the user is executing a switch (i.e., the first two branches are guarded by a guard node which becomes satisfied as soon as the User changes into the corresponding state), or a time-out occurs, modelled by the third branch which is guarded by an internal event sent from the Timer. In each case the state of Switch and Light is set correspondingly as a consequence. Alternatively, behaviour can also be guarded by a selection node which specifies a condition on the state of a component which needs to be satisfied in order to pass control to the successor node. If the condition is not satisfied (i.e., the component is not in this state) the behaviour terminates at this node.

Flags. The leaf nodes of most branches are reversion nodes (carrying the flag ‘ˆ’), modelling that the behaviour reverts back to the matching node and thus repeats the sub-tree. A matching node is a node that has the same component name, the same type and qualifier (e.g., Switch, [ ] and wait). As a side-effect of reversion all parallel threads are terminated (if they have branched off below the matching node) as to prevent infinite thread creation. As the behaviour of the time-out case is the same as in the situation where the user switched the light off, we use a reference node (carrying the flag ‘=>’) which substitutes the sub-tree below the matching node. Nodes in different parallel threads can also carry a synchronisation flag (marked by ‘=’) which enforces the synchronisation of the threads. This feature, however, has not been used in our model.

The Timer thread is modelled abstractly here with two parallel threads, one repeatedly increasing an abstract counter which is not specified here, and one awaiting an external input that indicates that the set maximum time has expired. Once this external input occurs a kill node, carrying the ‘−−’ flag, terminates the parallel counting thread (i.e., the thread that follows the node matching the kill node). Note that these first two nodes of the thread are linked
and build an atomic block which specifies that the execution occurs in one atomic step. After the counting thread is killed, the timer resets, sends an internal time_out event (read by the Switch thread) and reverts back to the root of the Timer thread.

The User thread models toggling behaviour, which is triggered by an external input indicating a Person entering or exiting the room.

Tags and Parametrisation. Besides component names, types, qualifier and flags, a node might also carry a tag (e.g., R1) which indicates the requirement (from a given requirements document) it models. The symbol @ can be used to indicate points of integration. The syntax also provides constructs for parametrisation of sub-trees in a BT such that the sub-tree is to be performed on all members (do forall) or one member (do forone) of a reference set S. This feature, however, has not been used in the example above.

The system modelled here is a hypothetical one which could have been modelled in many different ways and on various levels of abstraction. Our aim here was to give a concise example which covers most of the common features of the BT syntax, namely various node types (state realisation, guard, selection, internal input, internal output, external input and external output), flags that manipulate the control flow (reversion, reference, synchronisation, thread kill), as well as concurrent and alternative branching, and atomic blocks.

3.2. Behavior Tree Models as Transition Systems

BT models can obviously be interpreted as transition systems in the sense that the nodes prescribe possible transitions from pre- to post-states. States are understood in the usual way as evaluations of state variables. The state variables of a BT model represent the components and the possible values represent the “states” that a component might “realise” (e.g., the state realisation node Switch[on] can be interpreted as the assignment Switch := on). Additional variables are used for marking the position in the BT and thus for capturing the control flow, such as program counters and Boolean flags to indicate sending or receiving of messages.

Since in BTs we encounter both the notion of labelling of states by variable/value pairs as well as the labelling of transitions by actions (i.e., BT nodes), we base our results on the concept of doubly-labelled transition systems, as described in the previous section. We denote the doubly-labelled transitions system that represents a BT model B with the state labelling function $L_B$. The set of actions in $L_B$ is given through the nodes (or atomic blocks) in B. Each of these nodes (or atomic blocks) change at least the program counter variable associated with the node’s thread. Some nodes also change other state variables, namely those nodes of type state realisation, internal input or output, and external output. That is, $s \xrightarrow{a} s'$ can occur if $s$ is a possible pre-state of node n (i.e., the value of the program counter of n’s thread in s corresponds to the position of n in the BT) and will lead to an updated post-state $s'$, i.e., $L_B(s') = (L_B(s) \oplus ctrlUpd(pd(n))) \oplus update(n)$, where $ctrlUpd(pd(n))$ specifies the update of the program counter variable of the thread to which node n belongs, $update(n)$ specifies the variable changes specified by n and $\oplus$ denotes a functional override. All other nodes (i.e., nodes of type guard, selection, external input and internal input) function as a guard on the transition, i.e., $s \xrightarrow{guard} s'$ can only occur if the pre-state s satisfies the condition imposed by the “guarding” node n, i.e., $guard(n) \in L_B(s)$ where $guard(n)$ specifies the condition on the state variables other than the program counter imposed by node n. When the transition occurs, the state remains unchanged apart from the program counter variable for that thread, indicating that the control flow has progressed beyond node n, i.e., $L_B(s') = L_B(s) \oplus ctrlUpd(pd(n))$.

3.3. Standard Slicing of Behavior Trees

Slicing of BTs is performed in a similar manner as program slicing. As a first step we create a control flow graph of the BT model (CFG-BT) which indicates the flow of control between single nodes. The CFG-BT is generated from the BT model by adding the following edges and nodes to the BT model:

- an edge from each reversion and reference node to its matching target node;
- a terminal node and a false edge for each selection node such that the new edge leads to the terminal node (modelling termination in the case where the selection is not satisfied);
- a false edge from each guard and input event node that loops back to itself (modelling blocking behaviour when the guard is not satisfied or the input event has not yet occurred).
Control flow graphs of BTs are different to control flow graphs for programs in that a node can have more than two successors (due to concurrent and alternative branching structures). Moreover, threads are not necessarily synchronised at the beginning and the end. A terminal node thus only terminates the behaviour of one thread only (while other threads might still be running). Therefore, a CFG-BT can have multiple terminal nodes. We consider threads to start at the root node of the BT and thus all threads contain the same initial nodes.

### 3.3.1. Node Dependencies

In a second step a Behavior Tree Dependence Graph (BTDG) is generated from a given CFG-BT. A BTDG (similar to a Program Dependence Graph [38, 39]) is a graph in which all (BT) nodes are linked according to their dependencies. The dependencies specifically defined for BTs are control dependence (cd), data dependence (dd), message dependence (md) (between internal input and output nodes), synchronisation dependence (sd) (between synchronisation nodes), interference dependence (id) (a dependence across parallel threads), and termination dependence (td) (between a node which terminates another thread, such as a reversion or thread kill node, and a node in the thread to be terminated). The definitions of these dependencies, as well as examples, can be found in [13, 21]. The edges in the BTDG are denoted as \( p \xrightarrow{d} q \) where \( d \in \{ cd, dd, md, sd, id \} \), indicating that node \( q \) depends on node \( p \) and the dependency is of type \( d \). A path in a BTDG is a sequence of nodes such that for every pair of consecutive nodes \( n_i \) and \( n_{i+1} \) there is a dependency of some type \( d \), i.e., \( n_i \xrightarrow{d} n_{i+1} \).

### 3.3.2. The Slicing Criterion

For each BT we will compute only one general BTDG which is independent of the property to be verified but covers all dependencies between all components and attributes. From this general BTDG a slice is created with respect to a specific temporal logic formula \( \varphi \). The formula \( \varphi \) gives rise to the slicing criterion, which is defined as the set of nodes which modify one of the variables included in \( \varphi \). Let \( \text{Def}(n) \) be the set of all components (or component attributes) that are defined or modified at node \( n \) (through a state realisation), then we define the slicing criterion as

\[
C_\varphi = \{ n \mid \exists v \in \text{vars}(\varphi) \cdot v \in \text{Def}(n) \}.
\]

The nodes in the criterion set form the starting points for simultaneous backwards traversals in the dependence graph. Using each of the criterion nodes as a starting point, the dependence graph is traversed in reverse, collecting all nodes that are encountered via dependence edges. The algorithm checks whether a node has previously been encountered before adding it to the slice, in order to prevent infinite cycles caused by cyclic or symmetric dependencies. The set of nodes encountered by the simultaneous traversals of the dependence graph is referred to as the slice set.

The second phase involves identifying which reversion and reference nodes to add back into the slice. These reversion and reference nodes are then used as the starting points for another reverse exploration of the dependence graph, in order to locate any further dependencies. Finally, the nodes collected in the slice set so far are re-formed into a syntactically correct Behavior Tree, forming the slice.

### 3.3.3. Observable and Stuttering Nodes

The nodes in the slicing criterion \( n \in C_\varphi \) are referred to as observable nodes whereas all nodes \( n \notin C_\varphi \) are stuttering nodes. The function \( \text{obs}_\varphi(n) \) returns true if and only if the node \( n \) is observable. Note that the backwards traversal in the dependence graph reaches stuttering as well as observable nodes, so while all observable nodes must be included in the slice, stuttering nodes may or may not be included in the slice.

Each stuttering node in a BT \( B \) gives rise to a stuttering step (wrt. \( \varphi \)) in \( L_B \). Similarly, observable nodes lead to observable steps.

### 3.3.4. A Layered Approach to Slicing

Due to the fact that the interference dependency relation is intransitive, the traversal through the BTDG leads to slices that include nodes along infeasible paths in the BTDG. That is, there is no true dependency to these nodes\(^2\) and

---

\(^2\)If there exists nodes \( p, q, r \) such that \( p \xrightarrow{id} q \) and \( q \xrightarrow{id} r \) then \( p \xrightarrow{id} r \) only if there exists a possible execution which performs \( p, q, \) and \( r \) in sequence [21, 40]. In some cases the execution of \( p \) might exclude the execution of \( r \), e.g., when both nodes are in alternative branches rather than parallel branches.
– although there is no harm in including them – the slice can be reduced by eliminating the nodes along infeasible paths. Thus, a slice $S_s$ that is created using the standard slicing technique (as outlined above) can be further optimised using an algorithm for detecting infeasible paths ($infPath$), described in detail in [21]. We denote this process as $infPath(S_s) = S_{inf}$.

For formulas which contain the next-step operator $X$, we propose a new algorithm in this paper which modifies a slice in such a way that the satisfiability of all CTL* formulas is preserved, including formulas containing the $X$ operator. We call this algorithm next-preserving slicing, denoted as function $nsp$. This algorithm can be applied to standard slices or optimised slices, i.e., $nsp(S_s) = S_{nsp}$ or $nsp(infPath(S_s)) = nsp(S_{inf}) = S_{insp}$. Applied to a BT model $B$ we can depict the following chain of steps if all the procedures are applied:

$$B \xrightarrow{\text{standard}} S_s \xrightarrow{\text{infPath}} S_{inf} \xrightarrow{nsp} S_{insp}$$

More details on our approach of standard slicing of BTs, including the full definitions of the dependencies and a proof of soundness, can be found in [21, 13]. The next section describes the next-preserving slicing function.

4. Next-preserving Slicing of BT models

The standard BT slicing procedure described in the previous section only preserves CTL*$_X$. This section presents a slicing algorithm which preserves properties containing the $X$ operator. The technique is derived from the given definition of next-preserving branching bisimulation, by ensuring that stuttering nodes are retained at places in the BT where it is required by the relation. In order to find these locations, the nodes corresponding to observable transitions must be identified, as well as nodes that may execute before a state satisfying the $diffpaths$ condition.

4.1. Where to place stuttering nodes

The first step is to determine the locations at which these stuttering nodes must be placed. From the definition of next-preserving branching bisimulation, stuttering nodes may need to be placed before observable transitions and states satisfying $diffpaths$. Locating observable transitions is straightforward, since these simply correspond to the observable nodes in the tree.

Finding the nodes corresponding to $diffpaths$ states requires a more careful examination of the various Behavior Tree constructs. A state satisfying $diffpaths$ is required to have at least two branches emanating from it. Branching only occurs in the transition system when there is non-determinism in the Behavior Tree. Furthermore, to satisfy $diffpaths$, one branch must lead to an observable node which is not possible on all paths via another branch, within the same number of steps. The Behavior Tree constructs to be considered in more detail are alternative branching, concurrent branching and conditional nodes. The remaining constructs, state realisations, thread kills, reversions, reference and synchronisation nodes, can only result in non-determinism when executed in parallel, so are covered by the discussion on concurrent branching in Section 4.1.2.

4.1.1. Alternative Branching

This is perhaps the most obvious Behavior Tree construct which exhibits non-deterministic behaviour, since one branch is chosen and the others are terminated. Since there may be an observable node in one branch that is unreachable on the other branch, the state immediately before a set of alternative branches may satisfy $diffpaths$. Note that the exception is if the root nodes of each of the alternative branches are selections or guards with mutually exclusive conditions, in which case the choice is deterministic.

4.1.2. Concurrent Branching

Concurrent branches may also have non-determinism, arising from the interleaved execution of nodes in different threads. Hence, any node in a concurrent branch is a branching point of behaviour and there is a non-deterministic choice before every step. In most cases such states do not satisfy $diffpaths$, since after executing a node in one thread, it is still possible to execute the nodes in the other threads.

There are cases, however, where the execution of one or more nodes in one branch may cause the early termination of the other thread by rendering a selection criterion false. For example, in the BT on the left of Figure 6, if $B[f]$
executes, it causes $B?b?$ to evaluate to false and the thread to terminate. Hence, the descendent $C[c]$ is not executed. Another case is where one branch contains a thread kill node that terminates the other branch. In general, the cases of interest are when an observable node is transitively dependent on a conditional node, which is in turn dependent on a node in a parallel thread that negates the condition, or when an observable node is in a thread that may be terminated by a thread kill in another thread. Note that even though reversions terminate parallel threads, after reverting, the control flow can still reach those threads when they are re-started.

4.1.3. Conditional nodes

It may appear that conditional nodes produce branching in the transition system, since they produce branching in the corresponding control flow graph. However, this is not the case. At any particular state, the values of the atomic propositions are already known. Therefore, if the next node which is to execute is a conditional node, it is already known whether or not the condition is satisfied by that state, so there is no non-determinism involved. For example, consider the Behavior Tree shown on the left of Figure 6. The selection $B?b?$ may or may not hold, depending on whether or not the state realisation $B[f]$ has executed. That is, the location between $A[a]$ and $B?b?$ corresponds to multiple states in the transition system. The diagram on the right of Figure 6 shows the corresponding transition system. Both states $s_2$ and $s_6$ correspond to the location between $A[a]$ and $B?b?$ in the Behavior Tree. At $s_2$, the component $B$ is still set to $b$, so the condition is satisfied, whereas at $s_6$, the component $B$ is now set to $f$, so the condition is not satisfied. At both states, the values of the components are known, so there is no non-determinism.

The exception is external input nodes, which are controlled by the external environment and thus may or may not hold regardless of the current state of the system. Nevertheless, even external input nodes do not result in diffpaths states. Recall that in the control flow graph, the false branch from an external input node loops back to the node again. Figure 7 shows an example Behavior Tree with an external input node and the corresponding transition system. Assume that $C[c]$ is an observable node. Therefore, after taking the false branch, it is still possible to take the true branch on a subsequent iteration. Even though there is one possible path consisting of an infinite loop of false steps, the state $s_1$ will not satisfy diffpaths, since it does not have a successor from which all paths do not reach the observable node.
In summary, the only Behavior Tree constructs that produce \textit{diffpaths} states in the transition system are the nodes followed immediately by alternative branches which are not guarded by mutually exclusive selections and nodes in concurrent branches which prevent nodes in other branches from executing.

To produce a next-preserving bisimilar slice we need to add stuttering nodes before each of the nodes that are either observable or positioned before a \textit{diffpaths} state.

4.2. Slicing Algorithm

The following algorithm produces a next-preserving slice from a given Behavior Tree $B$. The algorithm starts by creating the normal slice $S$, according to the algorithm given in [21]. The rest of the algorithm searches for locations requiring extra stuttering nodes, specifically observable nodes and nodes corresponding to the steps before a \textit{diffpaths} state. For each of these locations, the algorithm selects suitable stuttering nodes to be returned to the slice. The algorithm maintains a set $\text{finalSet}$, which initially contains the nodes present in the normal slice. When the algorithm terminates, $\text{finalSet}$ contains the nodes which belong to the new next-preserving slice, consisting of the normal slice set with additional stuttering nodes as required.

The set of observable nodes are identified at Line 5, using the function $\text{obs}$. For each of these nodes, stuttering nodes are located, using the function $\text{findStuttering}$. This function identifies suitable stuttering nodes which can execute immediately before the given node and returns the correct number of them according to the $\text{xdepth}$ of the formula and the number of stuttering nodes which were present in the original BT. These stuttering nodes are then added to the slice $\text{finalSet}$. The observable node is marked as $\text{visited}$ to prevent infinite traversals in later parts of the algorithm.

As discussed in the previous section, there are two situations which result in \textit{diffpaths} states in the transition system: alternative branches and nodes in concurrent branches which prevent an observable node from executing.

Lines 11 to 15 repeat the process of selecting stuttering nodes for all the nodes which are root nodes of an alternative branch. Although there are cases where alternative branching groups do not result in \textit{diffpaths}, i.e. when the conditions of the root nodes are mutually exclusive, all alternative branching nodes are included. Computing each condition is undecidable in general, so it is not possible to identify which groups of alternative branches have mutually exclusive conditions. Including user input at this stage could produce a smaller slice by eliminating such cases.

From line 17, the algorithm locates the nodes which require stuttering steps in concurrent branches. The goal is to locate any nodes that can prevent an observable node from executing (i.e., falsifies the execution condition). For each of these nodes, referred to as $\text{falseNodes}$, the state (or states) in the transition system before it executes satisfies \textit{diffpaths}, because there is a choice between a path where an observable node executes and another on which it does not. Therefore, the nodes which can execute immediately before each of the $\text{falseNodes}$ are locations where extra stuttering is required. Lines 17 to 34 find these locations in the following fashion: For every observable node, the algorithm traverses each chain of dependencies from that node in the dependency graph, using the function $\text{depChains}$ at line 18, which returns the set of such chains. The function $\text{findFalseDep}$, at line 19, returns the set of nodes that
Algorithm 1 Next-preserving slicing of Behavior Tree $B$

1: Compute the normal slice $S$ of $B$
2: $x = x_{\text{depth}}(\phi)$
3: $\text{finalSet} = \{\}$
4: for all $n$ in $\text{nodes}(B)$ such that $\text{obs}(n)$ do
5: \hspace{4em} $\text{stuttSet} = \text{findStuttering}(n, x)$
6: \hspace{4em} Mark $n$ as visited
7: \hspace{4em} $\text{finalSet} = \text{finalSet} \cup \text{stuttSet}$
8: end for
9:
10: for all $n$ in $\text{nodes}(B)$ such that $n$ is the root node of an alternative branch do
11: \hspace{4em} $\text{stuttSet} = \text{findStuttering}(n, x)$
12: \hspace{4em} Mark $n$ as visited
13: \hspace{4em} $\text{finalSet} = \text{finalSet} \cup \text{stuttSet}$
14: end for
15:
16: for all $n$ in $\text{nodes}(B)$ such that $\text{obs}(n)$ do
17: \hspace{4em} for all $d$ in $\text{depChains}(n)$ do
18: \hspace{8em} $\text{falseNodes} = \text{findFalseDep}(n, d)$
19: \hspace{8em} for all $m$ in $\text{falseNodes}$ do
20: \hspace{12em} $\text{workingSet} = \{m\}$
21: \hspace{12em} while $\text{workingSet} \neq \{\}$ do
22: \hspace{16em} $\text{tempSet} = \{\}$
23: \hspace{16em} for all $p$ in $\text{workingSet}$ such that $\text{execBefore}(p, \text{falseNode})$ and $p$ is not marked as visited do
24: \hspace{20em} $\text{stuttSet} = \text{findStuttering}(p, x)$
25: \hspace{20em} $\text{tempSet} = \text{tempSet} \cup \text{stuttSet}$
26: \hspace{20em} Mark $p$ as visited
27: \hspace{20em} end for
28: \hspace{16em} $\text{finalSet} = \text{finalSet} \cup \text{workingSet}$
29: \hspace{16em} $\text{workingSet} = \text{tempSet}$
30: \hspace{12em} end while
31: \hspace{8em} end for
32: \hspace{4em} end for
33: end for
34: end for

negate a condition which the observable node is dependent on. For example, in Figure 6, if $C[c]$ is an observable node, then $B[I]$ would be returned by the $\text{findFalseDep}$ function.

For each of the $\text{falseNodes}$, the loop at lines 20 to 32 searches for nodes which can execute before it, given by the function $\text{execBefore}$ at line 24. The function returns the parent, as well as any nodes in parallel threads which are able to execute immediately prior to the $\text{falseNode}$. This is computed by exploring the dependency graph to ensure that there are no dependencies preventing the parent/parallel node from executing before the $\text{falseNode}$. In the example, $B[b]$ and $A[a]$ are both such cases. For each of the nodes returned by $\text{execBefore}$, stuttering nodes are selected at lines 24 to 28 and stored in a temporary set, $\text{tempSet}$.

The process is then repeated for the new stuttering nodes, since including these nodes might result in a new $\text{diffpaths}$ state in the transition system. At each iteration, the nodes in the current working set are added to the $\text{finalSet}$ and the working set is updated to $\text{tempSet}$, the set of additional stuttering steps found. When a fixed point is reached and there are no more unexplored nodes in the working set, the loop ends.

The nodes in the $\text{finalSet}$ must then be re-formed into a syntactically correct Behavior Tree, according to the procedure described in [21]. The resulting slice is next-preserving branching bisimilar to the original BT as follows from its construction, since stuttering steps were inserted before all observable nodes and states satisfying $\text{diffpaths}$. 
5. Related Work

Our work builds on the definition of branching bisimulation [3], in particular its variant that takes infinitely stuttering paths into account, introduced in [3] as divergence-sensitive branching bisimulation. A similar bisimulation relation has also been introduced in [14] as divergence-sensitive stuttering bisimulation. Both notions are weak bisimulations that disregard the number of stuttering steps in a system (and are hence suitable for reduction techniques such as slicing or partial order reductions) but take into account the branching structure of a system (hence suitable for linear and branching time temporal logics). Both bisimulation relations, however, do not preserve formulas containing the next-step temporal operator.

Bergstra et al. in [41] define the notion of orthogonal bisimulation equivalence, which refines van Glabbeek and Weijland’s branching bisimulation and allows to collapse a sequence of stuttering steps into one stuttering step to compress internal activity for the purpose of abstraction. The stuttering actions are thus not totally discarded; hence this notion provides a step into the same direction as our approach. The modal logic that characterises the equivalence relation, however, does not contain the next-step operator (only one stuttering action is preserved) and therefore would not be suitable for our purpose.

Kučera and Strejček in [20] characterise the temporal logic LTL in terms of the nesting depths of the two modalities X (the next-step operator) and U (the until operator) which leads to a hierarchy of LTL formulas. They formulate a stuttering theorem which principally provides a similar result to our work using the notion of \((m, n)\)–stutter closedness of languages for LTL formulas containing at most \(m\) U operators and at most \(n\) X operators. From this theorem one can conclude that if \(n\) is the nesting depth of the X operator in the formula then at least \(n\) stuttering steps have to remain in each path, and if \(m\) is the nesting depth of the U operator in the formula then at least \(m\) copies of repeated sub-paths have to remain in each path. However, their paper is mainly interested in theoretical aspects of LTL. Our work on the other hand is based on a branching bisimulation and thus the relation preserves formulas of linear and branching-time temporal logics. Furthermore, our results are driven by the application which lead to a definition of next-preserving branching bisimulation relation which is more specific and hence gives rise to an algorithmic approach to generate next-preserving branching bisimilar systems.

Slicing has been applied to other languages in the context of temporal logic model checking where preservation of temporal formulas is required. However, only few approaches are formalised and include the full proof of correctness.

For instance, Brückner and Wehrheim in [42] slice Object-Z for verification and in [30] extend the approach to CSP-OZ, a language that combines CSP and Object-Z. A further extension to CSP-OZ-DC, a combination of CSP, Object-Z and Duration Calculus, is given in [43]. In their approach, the slice preserves formulas of a temporal logic which is invariant to stuttering (e.g., LTL\(\neg\ X\)). Similarly, Rakow in [44] develops an approach for slicing Petri Nets which is guaranteed to preserve LTL\(\neg\ X\) properties. Bordini et al. in [45] slice agent-based systems written in the AgentSpeak language and prove that their approach preserves LTL\(\neg\ X\) using stuttering equivalence. Hatcliff et al. in [46] show the correctness of the Indus slicer, which slices Java programs, using a notion of projection similar to Weiser’s projection [2] to demonstrate that their approach preserves LTL\(\neg\ X\) properties.

Several other authors have proposed slicing approaches for reducing state explosion without providing full proofs of correctness. For example, Nguyen et al. in [33] present an approach for slicing AADL (Architecture Analysis and Design Language) specifications in order to reduce them prior to translation into Promela, the input language of the SPIN model checker [47]. They claim that CTL\(\neg\ X\) properties are preserved, but the proof is left for future work. Also, Schäfer and Poetzsch-Heffter in [48] slice specifications of adaptive systems as part of the MARS framework. They use consistent bisimulation to show that the approach preserves a variant of CTL\(\neg\) that does not contain the U or X operators, but details of the proof are not provided. Finally, van Langenhove and Hooijewijs in [49] present an approach for slicing UML models for verification. The approach is claimed to preserve LTL\(\neg\ X\) properties, although again a full proof is not given.

6. Conclusion

In this paper we have proposed a novel bisimulation relation, called next-preserving branching bisimulation, which takes into account the branching structure of paths and disregards stuttering steps along the paths unless they are significant with respect to a temporal logic formula. With this feature the relation is stronger than weak forms of bisimulation but weaker than strong bisimulation. In particular, the relation guarantees the preservation of temporal
logic formulas containing the next-step operator. We provided a proof that the new bisimulation preserves full CTL*.

The relation is useful in any application which requires formulas with the next-step operator but in which strong bisimulation is too restrictive.

As an example of such an application, we designed a novel slicing approach based on next-preserving branching bisimulation, to reduce the size of models that we want to model check. The definition of next-preserving branching bisimulation gives rise to an algorithm for slicing in that it advises which stuttering steps can be sliced away and which need to be preserved. The relation can then be utilised to prove correctness of the approach. Few approaches have been published that provide a full proof of correctness of their slicing technique. None of these preserve next-step formulas. To our knowledge our approach to slicing that preserves full CTL* is novel and has not been proposed for any modelling or programming language before.

References
