Next-preserving Branching Bisimulation

Nisansala Yatapanage[a,*], Kirsten Winter[b]

[a] *School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, UK*
[b] *School of Information Technology and Electrical Engineering The University of Queensland, Brisbane, QLD 4072, Australia*

Abstract

Bisimulations are equivalence relations between transition systems which assure that certain aspects of the behaviour of the systems are the same in a related pair. 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 *family of next-preserving branching bisimulations* to overcome this limitation.

Next-preserving branching bisimulations are parameterised with a natural number, indicating the nesting depth of the **X** operators that the bisimulation preserves, 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 parameterised relation for which we prove the preservation of CTL* formulas with an **X** operator-nesting depth that is not greater than the specified parameter. It can be shown that the family of next-preserving bisimulations constitutes a hierarchy that fills the gap between branching bisimulation and strong bisimulation.

As an example for its application we show how this definition gives rise to an advanced slicing procedure that creates a *formula-specific* slice, which constitutes a reduced model of the system that can be used as a substitute when verifying this formula. The result is a novel procedure for generating slices that are next-preserving branching bisimilar to the original model for any formula. We can assure that each slice preserves the formula it corresponds to, which renders the overall verification process sound.

Keywords: Bisimulation, transition system, temporal logic, CTL* , model checking, 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. The meaning of "observable" is relative to the type of bisimulation relation. For example, strong bisimulation [1, 2] considers all steps to be observable whereas weak forms of bisimulation relations regard silent or stuttering steps (i.e., steps that do not change the state of the model) as non-observable.

Strong bisimulation is known to preserve all temporal properties specified in the logic CTL*, the super-set of the linear and branching time temporal logics, LTL and CTL (see [3]). However, the properties that are preserved through weak forms of bisimulation 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 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* [4], a

*Corresponding Author. 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, Australia.
Email addresses: yatapanage@acm.org (Nisansala Yatapanage), kirsten@itee.uq.edu.au (Kirsten Winter)

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 forms of 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 preservation of all temporal logic properties seems contradictory and, to our knowledge, none of the existing notions of bisimulation satisfies it.

In this paper we define a novel form of bisimulation, referred to as the *family of next-preserving branching bisimulations*, which is based on van Glabbeek and Weijland's branching bisimulation [5]. The relations contained in this family are parameterised with a natural number which indicates the number of stuttering steps that are preserved at critical points in the system. This number indicates which formulas are preserved by the bisimulation. It can be shown that any formula with a nesting depth of **X** operators smaller than or equal to the specified parameter is preserved in the parameterised next-preserving bisimilar system: we give a proof that next-preserving branching bisimulation *of depth xd* preserves any CTL* formula with a **X** -nesting depth of xd or less. The next-preserving bisimulations are stronger than weak forms of bisimulation and weaker than strong bisimulation. In fact, they can be ordered into a hierarchy of bisimulations enclosed by branching bisimulation (at the weak end of the spectrum) and strong bisimulation (at the strong end). These results are useful in any application that aims for a reduction of the model, such as slicing (e.g., [4, 6]) or partial order reduction [7].

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 [8, 9]. 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., [10, 11, 12]), 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 aiming to support the user in capturing informal requirements ([13, 14]). 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 as in the original model. As a vehicle to prove the soundness of our slicing approach, the notion of next-preserving branching bisimulation with respect to a particular formula is used, which guarantees the preservation of this formula (even if it contains the next-step operator). 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 into two parts, where the first part (Section 2) constitutes the core of our work. Firstly, preliminary concepts used throughout the paper are introduced, such as doubly labelled transition systems (Section 2.1), CTL* (Section 2.2) and branching bisimulation (Section 2.3). Following from there we introduce our notion of a *family of next-preserving branching bisimulations* (Sections 2.4 and 2.5) and Section 2.6 provides the required preservation results. Section 2.7 proposes a specialisation for next-preserving bisimulations where parameterisation is extended so that a next-preserving bisimulation becomes specific for a particular formula. Section 2.8 presents the family of next-preserving bisimulations as a hierarchy that fills the gap between strong bisimulation and branching bisimulation.

In the second part of the paper, 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 [15] and gives the background. In Section 4 we demonstrate how the definition of the family of next-preserving branching bisimulations gives rise to a new sophisticated slicing procedure which generates a next-preserving bisimilar slice for a specific formula. Section 5 relates our work to other results in the literature and we conclude in Section 6.

2. Next-preserving Branching Bisimulation

This section introduces the *family of next-preserving branching bisimulations*, beginning with a brief introduction to doubly-labelled transition systems, the temporal logic CTL* and existing bisimulation relations. Following this is the main contribution of the paper, namely the definition of the *family of next-preserving branching bisimulations* and the proof that each member preserves any CTL* formula with a matching depth of **X** operators.

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 [16], which are transition systems containing labels on both edges and states. As a consequence, the results of this paper apply to Kripke structures as well as labelled transition systems.

Definition 1 (Doubly-labelled Transition System). *A doubly-labelled transition system, L^2TS, is a structure $(S, AP, I, \mathcal{L}, \mathcal{A}, \rightarrow)$ where S is a set of states, AP is a set of atomic propositions, $I \subset S$ is a set of initial states, $\mathcal{L}$ is a state labelling function (i.e., $\mathcal{L} : S \rightarrow 2^{AP}$), $\mathcal{A}$ is a set of actions which includes a special action τ (stuttering action), and $\rightarrow$: $S \times \mathcal{A} \times S$ is the transition relation. (The notation $s \xrightarrow{a} s'$ is used as a shorthand for $(s, a, s') \in \rightarrow$ and if the transition label is irrelevant it may be omitted.)*

For our purpose, we also assume the consistency of the L^2TS in the sense that the labelling of states is consistent with the transition relation. We assure this with the following consistency condition (cf. [16]).

Definition 2 (Consistent L^2TS). *An L^2TS $L = (S, AP, I, \mathcal{L}, \mathcal{A}, \rightarrow)$ is* consistent *if there exists an injective function $\alpha : (2^{AP} \times 2^{AP}) \rightarrow \mathcal{A}$ such that for any sets of atomic propositions $A, B, C : 2^{AP}$ the following holds:*

- *i.) $(s, a, t) \in \rightarrow$ implies $a = \alpha(\mathcal{L}(s), \mathcal{L}(t))$ (any action modifies the labelling of the pre-state into the labelling of the post-state of the step consistently with α.)*
- *ii.) $\alpha(A, A) = \tau$ (if the labelling of two consecutive states does not change, then the applied action is a stuttering action)*
- *iii.) $\alpha(A, B) = \alpha(A, C)$ implies $B = C$ (performing an action in a state cannot result in two different successor states)*

A consistent doubly-labelled transition system $L = (S, AP, I, \mathcal{L}, \mathcal{A}, \rightarrow)$ can be viewed as a Kripke structure $ks(L) = (S, AP, I, \mathcal{L}, \rightarrow')$ with $\rightarrow'$: $S \times S$ such that $s \rightarrow' s'$ if and only if there exists an $a \in \mathcal{A}$ such that $s \xrightarrow{a} s'$. It can also be viewed as a labelled transition system when we ignore the labels on states, i.e., $lts(L) = (S, AP, I, \mathcal{A}, \rightarrow)$. It follows that any equivalence defined on a labelled transition system $\sim_{lts}$ or a Kripke structure $\sim_{ks}$ can be lifted into an equivalence $\sim_{l^2ts}$ on a consistent doubly-labelled transition system as defined above. Hence, for any states s and t in S, $s \sim_{l^2ts} t \Leftrightarrow s \sim_{lts} t \Leftrightarrow s \sim_{ks} t$. In the following we assume consistency for any doubly-labelled transition system that is considered, without further mention.

A *run* in a doubly-labelled transition system is a sequence of states alternating with actions, starting and (if finite) ending at states, $\rho = \langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$ (or alternatively we may write $\rho = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$). A *path* on the other hand is a sequence of states, $\pi = \langle s_0, s_1, s_2 \ldots \rangle$ (alternatively, $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \ldots$), which can be derived from a run by removing all actions from the sequence. If π is derived from ρ we write $\pi = path(\rho)$. Vice versa, for each path π there exists a unique run ρ (following from function α Definition 2 being injective), referred to as $run(\pi)$. The length of a path π is given through $length(\pi)$; if π is infinite then $length(\pi) = \infty$.

The notation $\rho\langle\langle s_i]$ denotes the prefix of the run ρ ending with (and including) s_i, while $\rho[s_i\rangle\rangle$ returns the suffix of ρ starting at (and including) s_i. The same notation will be used for paths. The concatenation of two paths π_1 and π_2, where π_1 is finite, is denoted as $\pi_1 \frown \pi_2$. Any path (or run) can be split into *segments* π^i (for any $0 \leq i < n$) such that $\pi = \pi^0 \frown \pi^1 \frown \ldots \pi^n$. We indicate that a path π^i is a segment of π using the notation $\pi^i \subseteq \pi$.

The set of paths starting at a state s is denoted as $paths(s)$. A path is said to be *maximal* if either its last state, $last(\pi)$, has no possible successor defined through $\rightarrow$ (i.e., there exists no state $s' \in S$ such that $last(\pi) \rightarrow s'$), or if it ends in a cycle (i.e., it reaches a state that it has visited earlier).

We denote a transition step that performs a stuttering action as $p \dashrightarrow q$. A path consisting entirely of stuttering steps is referred to as a *stuttering path*, denoted as $p \dashrightarrow^n q$ if it consists of n stuttering steps, or $p \dashrightarrow^* q$ if it consists of an arbitrary number of stuttering steps.

*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 ([17]), 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 3 (**Syntax of CTL***).

State formulas*:*

i.) *If φ is true, then φ is a state formula.*
ii.) *If $\varphi \in AP$, then φ is a state formula.*
iii.) *If φ and ψ are state formulas, then $\neg\varphi$ and $\varphi \wedge \psi$ are state formulas.*
iv.) *If φ is a path formula, then $\mathbf{E}\,\varphi$ is a state formula.*

Path formulas*:*

i.) *If φ is a state formula, then φ is also a path formula.*
ii.) *If φ and ψ are path formulas, then $\neg\varphi$, $\varphi \wedge \psi$, $\mathbf{X}\,\varphi$ and $\varphi\,\mathbf{U}\,\psi$ are path formulas.*

Additionally, the state operator **A** is derived as follows: $\mathbf{A}\,\varphi = \neg\mathbf{E}\,\neg\varphi$. The path operators **F** and **G** are derived as follows: $\mathbf{F}\,\varphi = true\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi = \neg\mathbf{F}\,(\neg\varphi)$. The $\vee$ (disjunction) operator is derived for both state and path formulas as $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$. The semantics of CTL* is given in terms of states and maximal, but possibly finite, paths.

Definition 4 (**Semantics of CTL***). *Let T be a doubly-labelled transition system such that $T = (S, AP, I, \mathcal{L}, \mathcal{A}, \longrightarrow)$. A CTL* state formula ψ holds in a state $s \in S$, denoted $T, s \vDash \psi$ or simply $s \vDash \psi$, according to the following, where ψ_1 and ψ_2 are CTL* state formulas and φ is a path formula:*

i.) $s \vDash true$,
ii.) $s \vDash a$, *where* $a \in AP$, *iff* $a \in \mathcal{L}(s)$,
iii.) $s \vDash \neg\psi_1$ *iff* $s \nvDash \psi_1$,
iv.) $s \vDash \psi_1 \wedge \psi_2$ *iff* $s \vDash \psi_1$ *and* $s \vDash \psi_2$,
v.) $s \vDash \mathbf{E}\,\varphi$ *iff there exists a maximal path* $\pi = \langle s_0, s_1, \ldots\rangle$, *such that* $s_0 = s$ *and* $\pi \vDash \varphi$.

A CTL* path formula φ holds for a path $\pi = \langle s_0, s_1, \ldots\rangle$, denoted $\pi \vDash \varphi$, according to the following, where φ_1 and φ_2 are CTL* path formulas and ψ_1 is a CTL* state formula:

i.) $\pi \vDash \psi_1$ *iff* $s_0 \vDash \psi_1$,
ii.) $\pi \vDash \neg\varphi_1$ *iff* $\pi \nvDash \varphi_1$,
iii.) $\pi \vDash \varphi_1 \wedge \varphi_2$ *iff* $\pi \vDash \varphi_1$ *and* $\pi \vDash \varphi_2$,
iv.) $\pi \vDash \mathbf{X}\,\varphi_1$ *iff* $\pi[s_1\rangle\rangle \vDash \varphi_1$,
v.) $\pi \vDash \varphi_1\mathbf{U}\,\varphi_2$ *iff* $\exists\, 0 \leq j < length(\pi)$ *such that* $\pi[s_j\rangle\rangle \vDash \varphi_2$ *and* $\forall\, i$, *where* $0 \leq i < j$, $\pi[s_i\rangle\rangle \vDash \varphi_1$.

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

We denote the set of atomic propositions comprised in formula φ as $ap(\varphi)$, and refer to a restricted labelling function of T which only considers the atomic propositions used in φ, using the notation $\mathcal{L}|_\varphi : S \to 2^{AP \cap ap(\varphi)}$.

Definition 5 (**Observable actions**). *An action $a \in \mathcal{A}$ is referred to as an* observable action*, denoted as $obs(a)$, if and only if for any $s, s' \in S$ with $s \xrightarrow{a} s'$ it holds that $\mathcal{L}(s) \neq \mathcal{L}(s')$. This predicate can be restricted further to* observable with respect to φ*, denoted as $obs_\varphi(a)$, if and only if $\mathcal{L}|_\varphi(s) \neq \mathcal{L}|_\varphi(s')$.*

Every action is either observable or a stuttering action, i.e., for any $a \in \mathcal{A}$, $obs(a)$ or $a = \tau$. In the context of a specific formula φ, stuttering may be considered with respect to φ such that $\neg obs_\varphi(a)$ (and hence $\mathcal{L}|_\varphi(s) = \mathcal{L}|_\varphi(s')$) if and only if $a = \tau_\varphi$ where τ_φ denotes a stuttering action with respect to φ.

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 τ, which do not have to be matched by the other system. For many applications, these relaxed requirements are essential and weak forms of bisimulation are the only possible form of equivalence which can be established. One weak form of bisimulation, known as *branching bisimulation* ([5]), is particularly of interest as it distinguishes between two systems which perform the same observable actions 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 with and without 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$^*_{-X}$ ([16]). However, divergence-sensitive branching bisimulation as defined in ([16]) 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 [18], 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 ([5]), 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 6, which is adapted to our context from the definition in [18]. Branching bisimulation with explicit divergence preserves CTL$^*_{-X}$ ([19]). Our next-preserving branching bisimulation relation is based on branching bisimulation with explicit divergence.

Definition 6. *Let T_1 and T_2 be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \rightarrow_i)$ for $i \in \{1, 2\}$, and $S = S_1 \uplus S_2$ be the disjoint union of both sets of states. A relation is a* branching bisimulation with explicit divergence*, denoted as bb, if and only if it is symmetric and for all $s, s', t \in S$ such that $bb(s, t)$ the following holds:*

- $\mathcal{L}_1(s) = \mathcal{L}_2(t)$ *and*
- $s \xrightarrow{a} s'$ *implies* $a = \tau \wedge bb(s', t)$ *or*
  there exist $t', t'' \in S$ *such that* $t \dashrightarrow^* t' \xrightarrow{a} t''$, $bb(s, t')$, *and* $bb(s', t'')$ *and*
- *if there exists an infinite stuttering path* $s \dashrightarrow s_1 \dashrightarrow \ldots$, *such that* $bb(s_i, t)$ *for all* $i > 0$, *then there exists an infinite stuttering path* $t \dashrightarrow t_1 \dashrightarrow \ldots$, *such that* $bb(s_i, t_j)$ *for all* $i, j > 0$.

Two states s and t are branching bisimilar with explicit divergence, denoted $s \triangleq t$, iff there exists a branching bisimulation with explicit divergence, bb, such that $bb(s, t)$.

This relation can be made more specific by relating it to a specific formula φ, denoted as $\triangleq_\varphi$. To achieve this, in the above definition the occurrence of $\mathcal{L}_i$ is replaced by $\mathcal{L}_i|_\varphi$ (for $i \in \{1, 2\}$), τ by τ_φ, and any bb by bb_φ. As an effect of this, the induced equivalence relation becomes coarser as less transitions affect φ specifically. In particular, the set of actions for which $obs_\varphi(a)$ holds is smaller than the set for which $obs(a)$ holds; more steps can be considered to be τ_φ than τ. Hence more states fall into the same equivalence class.

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., [20]), it has many uses in practical applications. In particular, it is often used when specifying safety properties of the form $p \Rightarrow \mathbf{X}\, q$, 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 [21].

Unfortunately, the weak forms of bisimulation are unable to preserve properties containing the **X** operator since systems related by such forms of bisimulation 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 form of 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 action is performed or immediately before a branching point in the transition system, where an observable action 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.

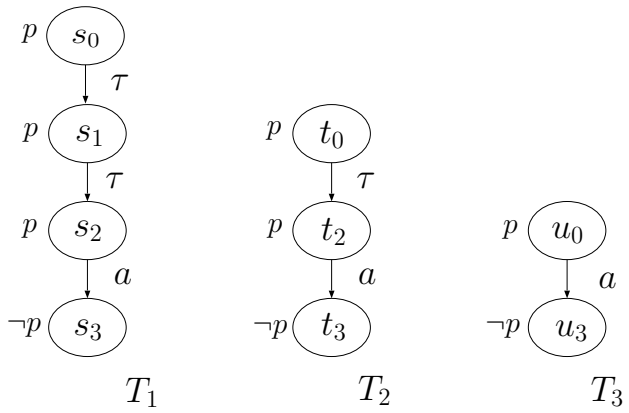

Figure 1: Two branching bisimilar transition systems performing an observable action

Example ▷ Consider the three transition systems, T_1, T_2 and T_3 shown in Figure 1. Assume that in T_1, the steps from s_0 to s_1 and from s_1 to s_2 are stuttering steps and the step from s_2 to s_3 performs an observable action (i.e., $obs(a)$). The three transition systems are branching bisimilar, as they both perform the same observable action and have the same branching structure; they could in fact stem from a reduction procedure that eliminates stuttering, i.e., T_2 results from T_1 through eliminating one stuttering step whereas T_3 results from T_1 having eliminated both stuttering steps before the observable action is performed.

Now, assume the property to be verified is $\mathbf{AX}\,p$, where p is an atomic proposition. In T_1, $s_0 \vDash \mathbf{AX}\,p$ as the system performs two stuttering steps before the observable action occurs and p becomes invalid. In T_2, $t_0 \vDash \mathbf{AX}\,p$ since there is one stuttering step before the observable action occurs. In T_3, however, $u_0 \nvDash \mathbf{AX}\,p$ as the observable action occurs immediately and invalidates p.

The validity of the property $\mathbf{AX}\,p$ changes only when the validity of p is changed at the next step. In this next step the system must perform an observable action to change the validity of p. It can be seen that at least one stuttering transition before the observable action is necessary in order to satisfy the formula. In general, the number of stuttering transitions required appears to be dependent on the number of nested **X** operators in the formula that is considered.

◁

This example demonstrates that the number of stuttering steps preceding an observable action influences the

validity of a formula containing $\mathbf{X}$. However, if T_1 contains fewer stuttering steps than the nesting depth of the formula, it is only necessary for T_2 and T_3 to precisely match the number of stuttering steps that T_1 has.

Although this example has used one particular property, most properties are influenced by the occurrence of observable actions. The only exceptions are properties which change their validity depending on the existence of a particular path. This is demonstrated by the following example.

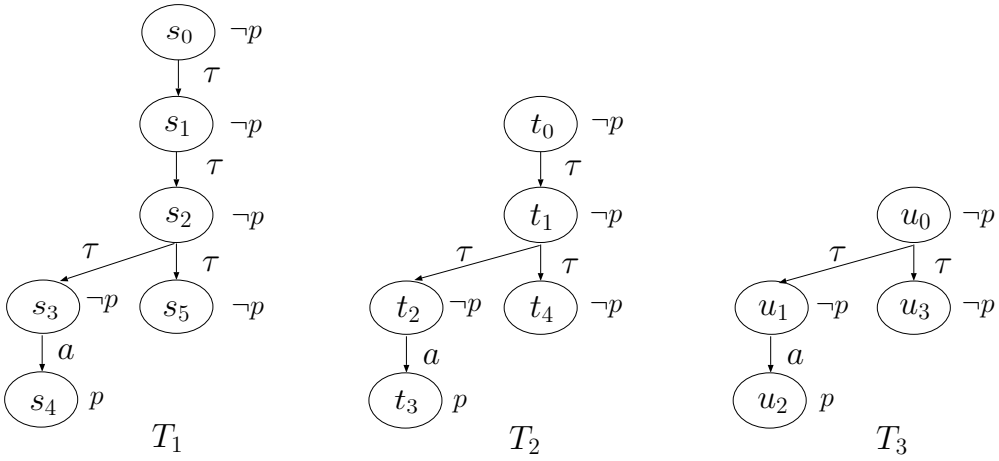

Figure 2: Three branching bisimilar transition systems with an observable action performed on one path only

Example ▷ Consider the transition systems shown in Figure 2 and assume the formula is $\mathbf{AX}\,(\mathbf{E}\,\mathbf{F}\,p)$. In T_1, there are two possible paths from s_0: the path $\langle s_0, s_1, s_2, s_3, s_4\rangle$ and the path $\langle s_0, s_1, s_2, s_5\rangle$. 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. A similarly reasoning applies to system T_2 which has one less initial stuttering step. The property holds for the initial state t_0 as its only next state is t_1 from which there exists a path (namely along t_2 and t_3) which eventually satisfies p.

However, in T_3, the formula does not hold at u_0. As in T_1 and T_2 there are still two possible paths, but this time the next step after u_0 differs on each path. On one path the next state is u_1, from which there exists a path where eventually p holds, but on the other path the next state is u_3, 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 eventually 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 u_2 and u_3 had outgoing edges looping back to u_0. ◁

This example illustrates another location at which stuttering steps may be required, namely where there is a branching point such that on one path an observable action occurs and on the other path the observable action does not occur at all or not within the same number of steps. In the following, we refer to these points in a system's behaviour as *critical branching points*. They constitute a point from which any next step is *observable*, even if it is a stuttering step and no observable action is performed, because the behaviour that is possible after this step is not the same as the behaviour that is possible before this step (i.e., at the branching point).

2.4.1. Observable steps

Both examples above together illustrate what constitutes an observable step in next-preserving branching bisimilar systems, referred to as a *next-preserving-branching observable step*. It is a step which either

- performs an *observable action* (i.e., changes the labelling of the pre-state), or
- passes a *critical branching point*, or
- performs a *stuttering step that is relevant* with respect to a particular formula.

In all cases we can see that what is satisfied in the state before the observable step might not be satisfied in the state after the observable step and vice versa. In the first case, an observable action occurs and changes the labelling of the state. In the second case, the branching structure of the possible behaviours has changed from the pre- to the post-state, which has an effect on the validity of formulas containing the **E** and **A** path quantifiers. In the third case, the number of relevant stuttering steps before an observable action or critical branching point differs and therefore the validity of a formula containing the **X** operator has changed.

The concept of an "observable step" is not new and also inherent in the notion of branching bisimulation (although it is not called an observable step), as any bisimilar system is required to match any observable step in the original system. However, branching bisimulation considers only observable steps of the first two kinds (namely covering observable actions and critical branching points). Hence, the pre- and post-state of an observable step of the first two kinds are not branching bisimilar (according to Definition 6). We introduce the term of a bb-observable step.

Definition 7 (bb-observable step). *A step $s \to s'$ is a* branching bisimilar observable step*, or short* bb-observable step*, if $s \not\triangleq s'$. If the bb-observable step is labelled with an action a we also use the notation $obs_{bb}(a)$, i.e., $s \xrightarrow{a} s'$ and $s \not\triangleq s'$ iff $obs_{bb}(a)$ (note that $obs_{bb}(a)$ does not necessarily imply that $a \neq \tau$).*

On the other hand, the pre- and post-state of an observable step of the third kind (performing a relevant stuttering step) are not "next-preserving" bisimilar. That is, in the context of *next-preserving* branching bisimulations, we additionally have to ensure that the number of stuttering steps *preceding any bb-observable step* in the bisimilar system is sufficient. The examples above demonstrate how a violation of this condition has its effect on the validity of a given formula containing any **X** operators.

The validity of a formula depends not on the number but on the *nesting depth of* **X** *operators* in the formula. We define this notion inductively as the *xdepth of a formula* as follows. A similar definition for LTL formulas is given in [22].

Definition 8 (xdepth). *Assume ψ_1 and ψ_2 are state or path formulas and φ, φ_1 and φ_2 are path formulas. The xdepth of a CTL* formula is then given as follows,*

$xdepth(\varphi) = 0$*, where* $\varphi \in AP$*,*
$xdepth(\psi_1 \wedge \psi_2) = max(xdepth(\psi_1), xdepth(\psi_2))$*,*
$xdepth(\neg\psi_1) = xdepth(\psi_1)$*,*
$xdepth(\mathbf{E}\,\varphi) = xdepth(\varphi)$*,*
$xdepth(\varphi_1 \mathbf{U}\,\varphi_2) = max(xdepth(\varphi_1), xdepth(\varphi_2))$*,*
$xdepth(\mathbf{X}\,\varphi) = xdepth(\varphi) + 1$*.*

As could be seen in the examples above, the notion of the $xdepth$ of a formula gives rise to the number of stuttering steps required to be present in the paths of two *next-preserving* bisimilar systems in order to preserve the validity of the formula in both systems. If a behaviour has more than $xdepth(\varphi)$ stuttering steps before a bb-observable step then the validity of φ does not change along the excess stuttering steps (i.e., all stuttering steps that are further than $xdepth(\varphi)$ steps away from the bb-observable step); these stuttering steps are not observable with respect to φ. If the number of stuttering steps before a bb-observable step is less than $xdepth(\varphi)$ then the validity of φ potentially changes along one of the stuttering steps; the stuttering step is *observable* with respect to φ, or any formula with an $xdepth$ equal to $xdepth(\varphi)$. Hence, we have a dependency between the $xdepth$ (of a class of formulas) and the number of stuttering steps present before each bb-observable step in both systems. To be *next-preserving* bisimilar with respect to formulas with **X**-nesting depth *xd*, both systems have to coincide in the number of stuttering steps, but only up to *xd*.

In the following we refer to a next-preserving-branching observable step that performs a stuttering step relevant for maintaining formulas of nesting **X**-depth *xd*, i.e. the third kind of observable step listed above, as an *xd-relevant stuttering step*.

2.5. A Family of Next-preserving Branching Bisimulations

To capture the requirements on next-preserving bisimilar systems as laid out in the previous section, we parameterise the definition of a next-preserving branching bisimulation with a natural number. This leads us to a *family of bisimulations* such that each of its members is characterised by the *depth* up to which stuttering steps are considered

observable. We refer to each of the bisimulations as *next-preserving branching bisimulation of depth xd* where *xd* is a natural number.

The requirement to maintain some stuttering before some steps leads to a definition that not only considers the next step following a state (as in Definition 6) but instead a sequence of non-bb-observable stuttering steps possibly followed by a bb-observable step. Two states are next-preserving branching bisimilar if their labelling is identical and also the following conditions on every possible progression from both states are satisfied:

1. If the sequence of steps following from one of the next-preserving branching bisimilar states does not contain any npbb-observable steps (i.e., all steps are stuttering steps which constitute neither critical branching nor an *xd*-relevant stuttering step), then these steps do not need to be matched by any progression from the other next-preserving branching bisimilar state.
2. If the sequence consists of some non-bb-observable steps (i.e., stuttering steps that are not critical branching but may be *xd*-relevant stuttering steps) followed by a bb-observable step (performing an observable action or passing a critical branching point) then this bb-observable step needs to be present in some progression from the next-preserving branching bisimilar state, and also each *xd*-relevant stuttering step before the bb-observable step needs to be matched.

The definition is given as follows.

Definition 9 (Next-preserving Branching Bisimulation of depth xd over states). *Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, for $i \in \{1, 2\}$, and $S = S_1 \uplus S_2$ be the disjoint union of both sets of states. A relation over states in S is a* next-preserving branching bisimulation of depth *xd, denoted as npb_{xd}, if and only if it is symmetric and for states $s, t, \in S$ with $npb_{xd}(s, t)$ the following holds*

- $\mathcal{L}_1(s) = \mathcal{L}_2(t)$ *and*
- *on any successor states from s we pose the following conditions*

  a.) for any $s' \in S$ such that $s \dashrightarrow s'$ and $npb_{xd}(s, s')$ it follows that $npb_{xd}(s', t)$

  b.) for any $j \in \mathbb{N}$ and $s', s'' \in S$ such that $s \dashrightarrow^j s' \xrightarrow{a} s''$ and for all $s_i \in \langle s \dots s' \rangle$ $s_i \triangleq s$ and $s' \not\triangleq s''$, it follows that there exists $k \in \mathbb{N}$ and $t', t'' \in S$ such that $t \dashrightarrow^k t' \xrightarrow{a} t''$ and for all $t_i \in \langle t \dots t' \rangle$ $t_i \triangleq t$, and $npb_{xd}(s', t')$ as well as $npb_{xd}(s'', t'')$ and
  - *if $j < xd$ then $k = j$*
  - *if $j \geq xd$ then $k \geq xd$*

  and

- *if there exists an infinite stuttering path $s \dashrightarrow s_1 \dashrightarrow \dots$, such that $npb_{xd}(s_i, t)$ for all $i > 0$, then there exists an infinite stuttering path $t \dashrightarrow t_1 \dashrightarrow \dots$, such that $npb_{xd}(t_j, s_i)$ for all $i, j > 0$.*

Two states s and t are next-preserving branching bisimilar of depth *xd, denoted as $s \overset{\star}{=}_{xd} t$, if and only if there exists a next-preserving branching bisimulation of depth xd, npb_{xd}, such that $npb_{xd}(s, t)$.*

Figure 3 pictures the two cases outlined in Definition 9: On the left, if a τ step occurs after state s, which is not *xd*-relevant or passes a critical branching point, and results in a state s' which is next-preserving branching bisimilar of depth *xd* to state t, then this step does not have to be matched by a step after t (this describes the first case in the definition where $a = \tau$ and $npb_{xd}(s', t)$). On the right the second case is depicted, where any bb-observable step, (which is a step on which the performed action is observable, or a step that emanates from a critical branching point, i.e., $obs_{bb}(a)$), possibly preceded by some non-bb-observable stuttering steps, needs to be matched by the same step, also possibly preceded by some non-bb-observable stuttering steps, following state t. Of the preceding non-bb-observable stuttering steps we need to preserve those that are *xd*-relevant. In the figure, we mark the next-preserving bisimilar states with (solid-line) ellipses. As can be seen, the states before and after the stuttering sequence are in npb_{xd}. In the right graphic, a dotted-line ellipse marks the bb relation between states s' and t: for two states to be (standard) branching bisimilar, an arbitrary number of additional stuttering steps may be present (see Definition 6).

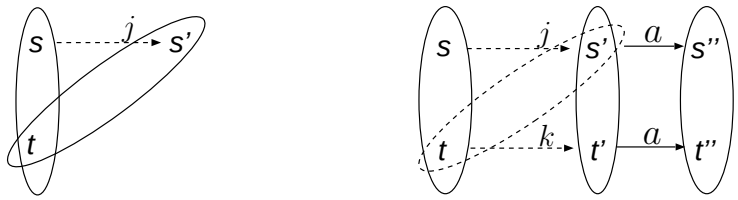

Figure 3: Two cases summarising Definition 9 with the required npb_{xd} relation between states marked by solid-line ellipses (the dotted-line ellipse indicates the bb relation)

The notion of next-preserving branching bisimulation of a particular depth can be lifted from states to paths in the obvious way. For two paths to be next-preserving bisimilar of depth *xd*, the relation requires the existence of related states on both paths, as well as the stuttering before the occurrence of a bb-observable step on both paths to be equal up to the depth *xd*.

To achieve this, we split each path into sub-paths which consist of a step performing a bb-observable step and its preceding (non-bb-observable) stuttering steps. For two paths to be next-preserving branching bisimilar it is required that each sub-path in one path is matched by a similar sub-path in the other path (and vice versa), such that it performs the same bb-observable step and mimics the same number of stuttering up to the depth *xd*.

That these matching sub-paths occur in the same order on both paths is guaranteed by the fact that we require that the matching sub-paths end in states from which the progression (i.e., the suffixes from these state) are also in the npb_{xd} relation. The definition formalises this as follows.

Definition 10 (Next-preserving Branching Bisimulation of depth xd over paths). *A path π_1 is* next-preserving branching bisimilar of depth *xd to a path π_2, denoted as $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$, if and only if*

$$(\pi_1 \stackrel{\rightarrow}{=}_{xd} \pi_2) \wedge (\pi_2 \stackrel{\rightarrow}{=}_{xd} \pi_1)$$

where $(\pi_i \stackrel{\rightarrow}{=}_{xd} \pi_j)$ with $\pi_i = \langle s_0, s_1, \ldots \rangle$ and $\pi_j = \langle t_0, t_1, \ldots \rangle$ if and only if $s_0 \stackrel{\star}{=}_{xd} t_0$ and

- *$s_0 \dashrightarrow^j s' \stackrel{a}{\rightarrow} s'' \subseteq run(\pi_i)$ with $j \geq 0$ and for all $s_i \in \langle s_0 \ldots s' \rangle$ $s_i \triangleq s_0$ and $s' \not\triangleq s''$ implies there exist $k \geq 0$ and $t', t'' \in S_j$ such that $t_0 \dashrightarrow^k t' \stackrel{a}{\rightarrow} t'' \subseteq run(\pi_j)$ and for all $t_i \in \langle t_0, \ldots, t' \rangle$ $t_i \triangleq t_0$, and $s' \stackrel{\star}{=}_{xd} t'$ and $\pi_i[s''\rangle\rangle \stackrel{\star}{=}_{xd} \pi_j[t''\rangle\rangle$ and*
  - *if $j < xd$ then $k = j$*
  - *if $j \geq xd$ then $k \geq xd$*

  or
- *if π_i is a finite or infinite stuttering path then π_j is a finite or infinite stuttering path, respectively.*

The possible scenarios for bisimilar sub-paths arising through this definition are depicted in Figure 4. On the left, the sub-path from state s to state s'', consisting of j stuttering steps before a bb-observable step is performed, where $j < xd$ (and hence all stuttering steps are *xd*-relevant), is matched by a sub-path from t to t'' containing the same number of *xd*-relevant stuttering steps. On the right, the sub-path from s to s'' includes more than or exactly xd stuttering steps (i.e., $j \geq xd$), and it has to be matched by a sub-path from t to t'' with at least xd stuttering steps before the bb-observable step occurs. Note that the ellipses around two states indicate that these states are next-preserving branching bisimilar of depth *xd* to each other. Notably, this includes the state pairs $(s, t_{(k-xd)})$, and $(s_{(j-xd)}, t)$, since those states are more than *xd* stuttering steps away from the next bb-observable step being performed (and hence the steps do not change the equivalence classes from one state to the next), and in fact it holds for all state pairs in between, i.e, $npb_{xd}(s, t_{k-n})$ for $xd \leq n < k$ and $npb_{xd}(s_{j-m}, t)$ for $xd \leq m < j$.

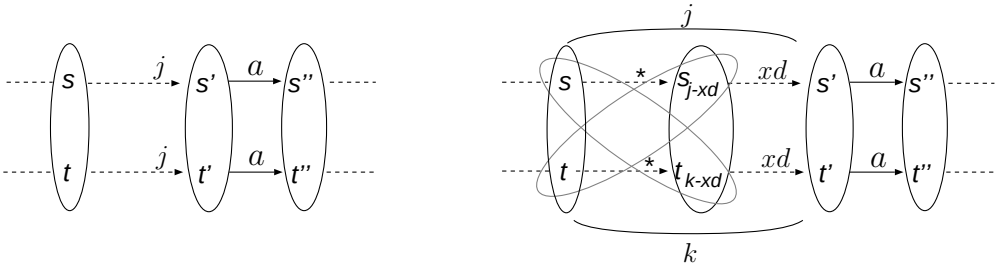

Figure 4: Two cases for matching sub-paths within two npb_{xd} paths that are covered in Definition 10

This picture invites us to an alternative formalisation of next-preserving branching bisimulation over paths in which we do not prescribe the shape of sub-paths consisting of stuttering steps followed by a bb-observable step. Instead we can characterise a path through its $\stackrel{\star}{=}_{xd}$*-induced* partitioning. Then each single partition in one path has to be matched by a partition in the next-preserving bisimilar path.

We say an $\stackrel{\star}{=}_{xd}$*-induced partitioning* of path π results in i *xd-equivalent partitions* $[\pi]^i_{xd}$, each being a maximal sub-sequence of π such that for all $s_i, s_j \in [\pi]^i_{xd}$, $s_i \stackrel{\star}{=}_{xd} s_j$. Since we require maximality it follows that for the last state in the sub-sequence, its successor, s_n, is not npbb-bisimilar to the states in the sequence (i.e., for all $s_i \in [\pi]^i_{xd}$, $s_i \stackrel{\star}{\neq}_{xd} s_n$) and hence the next step in π that follows the sub-sequence is an npbb-observable step.

Lemma 11. *If path $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$ then each xd-*equivalent partition *of π_1, $[\pi_1]^i_{xd}$ is matched by an xd-*equivalent partition *of π_2, $[\pi_2]^i_{xd}$ such that for all $s \in [\pi_1]^i_{xd}$ and $t \in [\pi_2]^i_{xd}$, $s \stackrel{\star}{=}_{xd} t$.*

Proof. As depicted in Figure 4, each sub-path in π_1 of the form $s_0 \dashrightarrow^j s' \stackrel{a}{\rightarrow} s''$ consists of the following *xd-equivalent partitions*:

- if$j \geq xd$ then
  $[\pi_1]^0_{xd} = \{s_0, \ldots, s_{j-xd}\}$, $[\pi_1]^i_{xd} = \{s_{j-xd+i}\}$ for $0 < i < xd$, $[\pi_1]^{xd}_{xd} = \{s'\}$, and $[\pi]^{(xd+1)}_{xd} = \{s''\}$
- if $j < xd$ then
  $[\pi_1]^i_{xd} = \{s_i\}$ for $0 \leq i < j$, $[\pi_1]^j_{xd} = \{s'\}$, and $[\pi_1]^{(j+1)}_{xd} = \{s''\}$.

Each sub-path in π_2, of the form $t_0 \dashrightarrow^k t' \stackrel{a}{\rightarrow} t''$ will be similarly partitioned. With Definition 9 it follows that

- if $j \geq xd$ then for all $0 \leq i \leq (xd+1)$, for all states $s \in [\pi_1]^i_{xd}$ and $t \in [\pi_2]^i_{xd}$, $s \stackrel{\star}{=}_{xd} t$,
- if $j < xd$ then for all $0 \leq i \leq (j+1)$, for all states $s \in [\pi_1]^i_{xd}$ and $t \in [\pi_2]^i_{xd}$, $s \stackrel{\star}{=}_{xd} t$.

□

The notion of next-preserving branching bisimulation can furthermore be raised to the level of systems in the following way.

Definition 12 (Next-preserving Branching Bisimulation of depth xd of transition systems). *Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, for $i \in \{1, 2\}$. T_1 and T_2 are* next-preserving branching bisimilar of depth *xd, denoted as $T_1 \stackrel{\star}{=}_{xd} T_2$, if and only if*

- *for all $s_0 \in I_1$ there exists $t_0 \in I_2$ such that $s_0 \stackrel{\star}{=}_{xd} t_0$ and*
- *for all $t_0 \in I_2$ there exists a $s_0 \in I_1$ such that $s_0 \stackrel{\star}{=}_{xd} t_0$.*

The definition of next-preserving branching bisimulation of depth *xd* is set out in such a way that standard branching bisimulation with divergence can be considered as a special case. In fact we can deduce the following correlation.

Lemma 13. *For* $xd = 0$ *the equivalence* $\stackrel{\star}{=}_{xd}$ *coincides with* $\triangleq$.

Proof. This follows obviously from Definitions 6 and 9. The additional condition on the number of stuttering steps required before the occurrence of a bb-observable step is only restricting for $xd > 0$. If $xd = 0$ then for any $j \geq xd$ it implies that $k \geq xd$, and in particular $k \geq 0$. Hence, the number of stuttering steps is unrestricted (as it is in standard branching bisimulation). □

Furthermore, our parameterised bisimulation gives rise to a hierarchy of bisimulations such that the larger the number of preserved stuttering steps, (i.e., the larger the parameter *xd*), the stronger the bisimulation. That is, any next-preserving branching bisimulation with a larger parameter includes any next-preserving branching bisimulation with a smaller parameter. The following lemma formalises this dependency.

Lemma 14. *For any* $n, m \in \mathbb{N}$ *such that* $n > m$ *it holds that* $\stackrel{\star}{=}_n \subseteq \stackrel{\star}{=}_m$.

Proof. The dependency of the depths of a next-preserving branching bisimulation is apparent in the number of stuttering steps required before the occurrence of a bb-observable step in the matching path of a bisimilar system. That is, for $s \in S_1$ and $t \in S_2$ such that $s \stackrel{\star}{=}_n t$ and bb-observable step $s' \stackrel{a}{\rightarrow} s''$ such that $s \dashrightarrow^j s' \stackrel{a}{\rightarrow} s''$ we are assured that the bb-observable step is mimicked in the progression from t and hence there exist $k \in \mathbb{N}$ and $t', t'' \in S_2$ such that $t \dashrightarrow^k t' \stackrel{a}{\rightarrow} t''$ and $j \geq n \Rightarrow k \geq n$ and $j < n \Rightarrow k = j$. If $s \stackrel{\star}{=}_m t$ then we require that $j \geq m \Rightarrow k \geq m$ and $j < m \Rightarrow k = j$.

Assume $s \stackrel{\star}{=}_n t$. Then

$j \geq n \Rightarrow k \geq n \Rightarrow k \geq m \Rightarrow s \stackrel{\star}{=}_m t$ or
$j < n \Rightarrow k = j \Rightarrow j < m$ or $m \leq j < n$
$- j < m \wedge k = j \Rightarrow s \stackrel{\star}{=}_m t$
$- m \leq j < n \wedge k = j \Rightarrow k \geq m \Rightarrow s \stackrel{\star}{=}_m t$.

□

We can also establish the following lemma which guarantees the existence of two paths being next-preserving branching bisimilar of depth *xd* within two systems that are next-preserving branching bisimilar of depth *xd*.

Lemma 15. *Let* T_1, T_2 *be doubly-labelled transition systems such that* $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, *for* $i \in \{1, 2\}$. *Then* $T_1 \stackrel{\star}{=}_{xd} T_2$ *implies that for any paths* $\pi_1 = \langle s_0, s_1, \ldots \rangle \in T_1$ *with* $s_0 \in I_1$, *there exists a path* $\pi_2 = \langle t_0, t_1, \ldots \rangle \in T_2$ *with* $t_0 \in I_2$ *such that* $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$. *If* π_1 *is maximal* π_2 *will be maximal too.*

Proof. Splitting π_1 into sub-paths $\pi_1^i = \langle s_i, \ldots s_i', s_i'' \rangle$ such that $s_i \dashrightarrow^{j_i} s_i' \stackrel{a_i}{\rightarrow} s_i''$ where $obs_{bb}(a_i)$ for any $i > 0$, possibly followed by an infinite or finite maximal stuttering sub-path, provides us with an inductive argument over the number of sub-paths within π_1.

Let $\pi[\![s]\!\rangle^n$ denote the path starting at state s that consists of n sub-paths of the form $s_i \dashrightarrow^{j_i} s_i' \stackrel{a_i}{\rightarrow} s_i''$ with $s_j \triangleq s_i$ (for all $s_j \in \langle s_1, \ldots, s_i' \rangle$ and $obs_{bb}(a_i)$ for $0 < i \leq n$, and $\pi[\![s]\!\rangle^0 = \langle s \rangle$. Furthermore, let $\pi^\tau[\![s]\!\rangle$ denote the finite but maximal or infinite stuttering paths starting from s (i.e., no bb-observable step can occur afterwards).

Any path π_1 can be described as either:

a.) $\pi_1 = \pi_1[\![s_0]\!\rangle^n$ or

b.) $\pi_1 = \pi_1[\![s_0]\!\rangle^n \frown \pi_1^\tau[\![s_n'']\!\rangle$ assuming n is finite

The proof is based on the induction over n:

- Base case: For $n = 0$, there are two cases
  a.) $\pi_1 = \langle s_0 \rangle$ and with Definitions 9 and 10 it follows that there exists $t_0 \in S_2$ such that $s_0 \stackrel{\star}{=}_{xd} t_0$ and for the path $\pi_2 = \langle t_0 \rangle$ it holds that $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$;
  b.) $\pi_1 = \pi_1^\tau[\![s_0]\!\rangle$ and with Definition 9 it follows that there exists $t_0 \in S_2$ such that $s_0 \stackrel{\star}{=}_{xd} t_0$ and if
  – $\pi_1^\tau[\![s_0]\!\rangle$ is an infinite sub-path then there exists an infinite stuttering path $\pi_2^\tau[\![t_0]\!\rangle$ such that $\pi_1^\tau[\![s_0]\!\rangle \stackrel{\star}{=}_{xd} \pi_2^\tau[\![t_0]\!\rangle$ or

– $\pi_1^\tau[s_0\rangle\!\rangle$ is a maximal but finite sub-path (i.e., it cannot be extended to perform a bb-observable step), then for all states $s_i \in \pi_1^\tau[s_0\rangle\!\rangle$ it holds that $s_i \stackrel{\star}{=}_{xd} t_0$.

- Inductive step: Assume the lemma holds for n. Again we consider two cases:

  a.) Assume for any paths $\pi_1 = \pi_1[s_0\rangle\!\rangle^n$ there exists a path $\pi_2 = \pi_2[t_0\rangle\!\rangle^n$ such that $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$. It follows that $s''_n \stackrel{\star}{=}_{xd} t''_n$ and with Definition 9 we can conclude that for any continuation $s''_n \dashrightarrow^{j_{n+1}} s'_{n+1} \stackrel{a_{n+1}}{\rightarrow} s''_{n+1}$ there exist a k_{n+1} and t'_{n+1}, t''_{n+1} such that $t''_n \dashrightarrow^{k_{n+1}} t'_{n+1} \stackrel{a_{n+1}}{\rightarrow} t''_{n+1}$ and furthermore if $j_{n+1} < xd$ then $k_{n+1} = j_{n+1}$ and if $j_{n+1} \geq xd$ then $k_{n+1} \geq xd$. From this it follows with Definition 10 that for the path $\pi_2[t_0\rangle\!\rangle^{n+1} = \pi_2[t_0\rangle\!\rangle^n \frown \pi_2^{n+1}$ (where π_2^{n+1} is the sub-path of the form $t''_n \dashrightarrow^{k_{n+1}} t'_{n+1} \stackrel{a_{n+1}}{\rightarrow} t''_{n+1}$) it holds that $\pi_1[s_0\rangle\!\rangle^{n+1} \stackrel{\star}{=}_{xd} \pi_2[t_0\rangle\!\rangle^{n+1}$.

  b.) Assume for any paths $\pi_1 = \pi_1[s_0\rangle\!\rangle^n \frown \pi_1^\tau[s''_n\rangle\!\rangle$ there exists a path π_2 such that $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$. The same argument for the case above applies for the non-stuttering sub-paths such that for any continuation $\pi_1[s_0\rangle\!\rangle^{n+1}$ there exists a sub-path $\pi_2[t_0\rangle\!\rangle^{n+1}$ such that $\pi_1[s_0\rangle\!\rangle^{n+1} \stackrel{\star}{=}_{xd} \pi_2[t_0\rangle\!\rangle^{n+1}$. With $s''_{n+1} \stackrel{\star}{=}_{xd} t''_{n+1}$ it follows from Definition 9 (similarly to the inductive base case part b) that either there exists a matching infinite stuttering sub-path $\pi_2^\tau[t''_{n+1}\rangle\!\rangle$ such that $\pi_1^\tau[s''_{n+1}\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2^\tau[t''_{n+1}\rangle\!\rangle$, and hence $\pi_1[s_0\rangle\!\rangle^{n+1} \frown \pi_1^\tau[s''_{n+1}\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_0\rangle\!\rangle^{n+1} \frown \pi_2^\tau[t''_{n+1}\rangle\!\rangle$, or otherwise if $\pi_1^\tau[s''_{n+1}\rangle\!\rangle$ is a finite maximal path then for all $s_i \in \pi_1^\tau[s''_{n+1}\rangle\!\rangle$ it holds that $s_i \stackrel{\star}{=}_{xd} t''_{n+1}$, and hence $\pi_1[s_0\rangle\!\rangle^{n+1} \frown \pi_1^\tau[s''_{n+1}\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_0\rangle\!\rangle^{n+1}$.

□

For next-preserving bisimilar paths we can also deduce that the suffix of one path, gained by omitting the first step, is next-preserving branching bisimilar of a reduced depth to the suffix of the bisimilar paths, gained by omitting the first step. This is formalised in the following lemma.

Lemma 16. *For any paths* $\pi_1 = \langle s_0, s_1, \ldots\rangle$ *and* $\pi_2 = \langle t_0, t_1, \ldots\rangle$*, if* $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$ *then* $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$.

Proof. From Definition 10 and $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$ it follows that

a.) if $s_0 \stackrel{a}{\rightarrow} s_1$ and $obs_{bb}(a)$ then $t_0 \stackrel{a}{\rightarrow} t_1$ and $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_1\rangle\!\rangle$. With Lemma 14 it follows that $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$.

b.) if $s_0 \dashrightarrow^j s' \stackrel{a}{\rightarrow} s''$ and $j > 0$ and $obs_{bb}(a)$ then there exists k and t', t'' such that $t_0 \dashrightarrow^k t' \stackrel{a}{\rightarrow} t''$ and
   – $0 < j < xd$ implies $k = j$ and $xd > 1$, and hence $(j-1) < (xd-1)$ and $(j-1) = (k-1)$ from which follows $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$,
   – $j \geq xd$ implies $k \geq xd$, and hence $(j-1) \geq (xd-1)$ and $(k-1) \geq (xd-1)$ from which follows $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$.

c.) if π_1 is a finite stuttering path, then for all $s_i \in \pi_1$ and $t_j \in \pi_2$, $s_i \stackrel{\star}{=}_{xd} t_j$. Thus, $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_1\rangle\!\rangle$ and with Lemma 14 it follows that $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$.

d.) if π_1 is an infinite stuttering path, then π_2 is also an infinite stuttering path and since $s_0 \stackrel{\star}{=}_{xd} t_0$, it follows that $s_1 \stackrel{\star}{=}_{xd} t_1$ and thus $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_1\rangle\!\rangle$. With Lemma 14 it follows that $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$.

□

Furthermore, for two next-preserving bisimilar paths we can show that for each state in one path there exist a state in the other path that is next-preserving bisimilar, and also their corresponding suffixes are next-preserving bisimilar.

Lemma 17. *For any two paths π_1 and π_2, $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$ implies that for all states $s \in \pi_1$ there exists a state $t \in \pi_2$ such that $s \stackrel{\star}{=}_{xd} t$ and $\pi_1[s\rangle\rangle \stackrel{\star}{=}_{xd} \pi_2[t\rangle\rangle$, and vice versa.*

Proof. This result follows from the construction in Lemma 11. Path π_1 can be partitioned into i maximal *xd*-equivalent partitions $[\pi_1]^i_{xd}$ and similarly path π_2 can be partitioned into i maximal *xd*-equivalent partitions $[\pi_2]^i_{xd}$. Each $[\pi_1]^i_{xd}$ is matched by $[\pi_2]^i_{xd}$ in the sense that for all $s_i \in [\pi_1]^i_{xd}$ and $t_i \in [\pi_2]^i_{xd}$, $s_i \stackrel{\star}{=}_{xd} t_i$.

Every state $s \in \pi_1$ falls into one *xd*-equivalent partition, i.e., there exists an i such that $s \in [\pi_1]^i_{xd}$. With Lemma 11 it follows that there exists an *xd*-equivalent partition in π_2, $[\pi_2]^i_{xd}$ (which is non-empty) such that $[\pi_1]^i_{xd} \stackrel{\star}{=}_{xd} [\pi_2]^i_{xd}$. We can choose any $t \in [\pi_2]^i_{xd}$ such that $s \stackrel{\star}{=}_{xd} t$.

As a consequence of Definition 10 it follows that the continuation from s in path π_1, $\pi_1[s\rangle\rangle$, may either start with some stuttering steps that are not *xd*-relevant (if $[\pi_1]^i_{xd}$ contains more than one element) or performs an npbb-observable step which leads to the next *xd*-equivalent partition $[\pi_1]^{(i+1)}_{xd}$. In both cases the suffix from t in path π_2, $\pi_2[t\rangle\rangle$, consists of matching partitions, and hence $\pi_1[s\rangle\rangle \stackrel{\star}{=}_{xd} \pi_2[t\rangle\rangle$. □

*2.6. Preservation of CTL**

The preservation of a CTL* formula depends on the nesting depth of **X** operators it contains. We can see that the standard branching bisimulation does not preserve formulas containing the **X** operator. Hence, only formulas with an $xdepth$ of 0 are preserved in a branching bisimilar system. To preserve any formula of an $xdepth$ greater than 0, the additional requirements of a next-preserving branching bisimulation of depth *xd*, where *xd* is equal to the $xdepth$ of the formula, are required.

We formulate the following theorem to establish the relation between the $xdepth$ of a formula and the parameter *xd* of the next-preserving branching bisimulation to guarantee the preservation of the formula.

Theorem 18 (Next-Preserving Branching Bisimulation of depth *xd* preserves CTL* formulas of xdepth *xd*). *For any $\psi \in$ CTL* and two doubly-labelled transition systems T_1 and T_2, such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, for $i \in \{1, 2\}$,*

$$(xdepth(\psi) = xd \ \wedge \ T_1 \stackrel{\star}{=}_{xd} T_2) \ \Rightarrow \ (T_1 \vDash \psi \Leftrightarrow T_2 \vDash \psi)$$

.

Proof. The proof is shown by induction over the formula ψ and comprises two cases: (i) if two states are next-preserving branching bisimilar of depth *xd*, they satisfy the same CTL* state formulas of xdepth *xd*, and (ii) if two paths are next-preserving branching bisimilar of depth *xd*, they satisfy the same CTL* path formulas of xdepth *xd*.

Assume $T_1 \stackrel{\star}{=}_{xd} T_2$. It is required to show that:

(i) For all state formulas $\psi \in$ CTL* with $xdepth(\psi) = xd$ and states $s \in I_1$ and $t \in I_2$,

$$s \stackrel{\star}{=}_{xd} t \Rightarrow (s \vDash \psi \Leftrightarrow t \vDash \psi)$$

(ii) For all path formulas $\varphi \in$ CTL* with $xdepth(\varphi) = xd$, states $s \in I_1$ and $t \in I_2$, and paths $\pi_1 \in paths(s)$ and $\pi_2 \in paths(t)$,

$$\pi_1 \stackrel{\star}{=}_{xd} \pi_2 \Rightarrow (\pi_1 \vDash \varphi \Leftrightarrow \pi_2 \vDash \varphi).$$

Assume that statement (i) holds for the state formulas ψ_1 and ψ_2 and statement (ii) holds for the path formulas φ_1 and φ_2.

(i) For state formulas:

$\boldsymbol{\psi \in AP}$:
Since $s \stackrel{\star}{=}_{xd} t$ implies that $\mathcal{L}_1(s) = \mathcal{L}_2(t)$, it obviously follows that $s \vDash \psi \Leftrightarrow t \vDash \psi$ for any atomic proposition ψ.

$\boldsymbol{\psi = \psi_1 \wedge \psi_2}$:
For $xdepth(\psi) = 0$ the result follows immediately from Lemma 13.
If $xdepth(\psi) > 0$ we apply the following reasoning. Let $xdepth(\psi) = n$, $xdepth(\psi_1) = n_1$ and $xdepth(\psi_2) = n_2$. From Definition 8 it follows that $n \geq n_1$ and $n \geq n_2$. Hence, $s \stackrel{\star}{=}_n t$ implies $s \stackrel{\star}{=}_{n_1} t$ and $s \stackrel{\star}{=}_{n_2} t$ (with Lemma 14). $s \vDash \psi$ if and only if $s \vDash \psi_1$ and $s \vDash \psi_2$. Since $s \stackrel{\star}{=}_{n_1} t$ and $s \stackrel{\star}{=}_{n_2} t$, with the induction assumption (i), it follows that $t \vDash \psi_1$ and $t \vDash \psi_2$, from which follows $t \vDash \psi$, so $s \vDash \psi \Leftrightarrow t \vDash \psi$.

$\boldsymbol{\psi = \neg \psi_1}$:
$s \vDash \psi$ if and only if $s \nvDash \psi_1$. From Definition 8, $xdepth(\psi) = xdepth(\psi_1)$, so from the induction assumption, $t \nvDash \psi_1$ and hence $t \vDash \neg\psi_1$. Therefore $s \vDash \psi \Leftrightarrow t \vDash \psi$.

$\boldsymbol{\psi = \mathbf{E}\,\varphi_1}$, for some path formula φ_1:
$s \vDash \psi$ if and only if there exists a maximal path $\pi_1 \in paths(s)$ such that $\pi_1 \vDash \varphi_1$. From Definition 8, $xdepth(\psi) = xdepth(\varphi_1)$, so with Lemma 15 it follows that there exists a maximal path $\pi_2 \in paths(t)$ such that $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$. From the induction assumption, $\pi_1 \vDash \varphi_1 \Leftrightarrow \pi_2 \vDash \varphi_1$ so therefore $s \vDash \psi \Leftrightarrow t \vDash \psi$.

(ii) For path formulas:

Let $\pi_1 = \langle s_0, s_1, \ldots\rangle$ where $s = s_0$ and $\pi_2 = \langle t_0, t_1, \ldots\rangle$ where $t = t_0$.

$\boldsymbol{\varphi = \varphi_1 \wedge \varphi_2}$:
The proof for this case is similar to the $\psi_1 \wedge \psi_2$ case above. Let $xdepth(\varphi) = n$, $xdepth(\varphi_1) = n_1$ and $xdepth(\varphi_2) = n_2$. From Definition 8 it follows that $n \geq n_1$ and $n \geq n_2$. Hence, $\pi_1 \stackrel{\star}{=}_n \pi_2$ implies $\pi_1 \stackrel{\star}{=}_{n_1} \pi_2$ and $\pi_1 \stackrel{\star}{=}_{n_2} \pi_2$ (with Lemma 14). With the induction assumption (ii) it follows that $\pi_1 \vDash \varphi_1 \Leftrightarrow \pi_2 \vDash \varphi_1$ and similarly for φ_2. Therefore, $\pi_1 \vDash \varphi_1 \wedge \varphi_2 \Leftrightarrow \pi_2 \vDash \varphi_1 \wedge \varphi_2$.

$\boldsymbol{\varphi = \neg \varphi_1}$:
$\pi_1 \vDash \varphi$ if and only if $\pi_1 \nvDash \varphi_1$. From Definition 8, $xdepth(\varphi) = xdepth(\varphi_1)$, so from the induction assumption, $\pi_2 \nvDash \varphi_1$ and hence $\pi_2 \vDash \neg\varphi_1$. Thus, $\pi_1 \vDash \varphi \Leftrightarrow \pi_2 \vDash \varphi$.

$\boldsymbol{\varphi = \varphi_1 \mathbf{U}\, \varphi_2}$:
$\pi_1 \vDash \varphi_1 \mathbf{U}\, \varphi_2$ if and only if there exists a j with $0 \leq j < length(\pi_1)$ such that $\pi_1[s_j\rangle\!\rangle \vDash \varphi_2$ and for all i with $0 \leq i < j$, $\pi_1[s_i\rangle\!\rangle \vDash \varphi_1$.
With Lemma 17 it follows that there exists a state $t_j \in \pi_2$ such that $t_j \stackrel{\star}{=}_{xd} s_j$ and $\pi_1[s_j\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_j\rangle\!\rangle$. With the given assumption we can deduce that $\pi_2[t_j\rangle\!\rangle \vDash \varphi_2$, using $xdepth(\varphi) \geq xdepth(\varphi_2)$ and Lemma 14. Similarly, we can reason that for all s_i ($0 \leq i < j$) there exists a state $t_i \in \pi_2$ such that $t_i \stackrel{\star}{=}_{xd} s_i$ and $\pi_1[s_i\rangle\!\rangle \stackrel{\star}{=}_{xd} \pi_2[t_i\rangle\!\rangle$, and hence, with $xdepth(\varphi) \geq xdepth(\varphi_1)$ and Lemma 14, $\pi_2[t_i\rangle\!\rangle \vDash \varphi_1$.
It remains to show that there does not exist a state $t_n \in \pi_2$, $0 \leq n < j$, such that $\pi_2[t_n\rangle\!\rangle \nvDash \varphi_1$. This follows from the concept of an xd-induced partitioning: each state in the prefix $\pi_2\langle\!\langle t_{j-1}]$ belongs to one *xd*-equivalent partition, hence $t_n \in [\pi_2]^m_{xd}$ for some m with $0 \leq m < j$, and with Lemma 11 we know that $[\pi_2]^m_{xd}$ matches some partition $[\pi_1]^m_{xd}$. With the assumption that for all i with $0 \leq i < j$, $\pi_1[s_i\rangle\!\rangle \vDash \varphi_i$, it follows also that $\pi_1[s_m\rangle\!\rangle \vDash \varphi_1$ for any $s_m \in [\pi_1]^m_{xd}$ and hence $\pi_2[t_n\rangle\!\rangle \vDash \varphi_1$.

$\boldsymbol{\varphi = \mathbf{X}\, \varphi_1}$:
$\pi_1 \vDash \varphi$ if and only if $\pi_1[s_1\rangle\!\rangle \vDash \varphi_1$; with Lemma 16, $\pi_1 \stackrel{\star}{=}_{xd} \pi_2$ implies $\pi_1[s_1\rangle\!\rangle \stackrel{\star}{=}_{(xd-1)} \pi_2[t_1\rangle\!\rangle$; since $xdepth(\varphi_1) = (xd - 1)$ it follows from the assumption (ii) that $\pi_2[t_1\rangle\!\rangle \vDash \varphi_1$ and hence $\pi_2 \vDash \varphi$.

□

2.7. Formula-specific Next-preserving Branching Bisimulations

The previous section introduces a family of next-preserving branching bisimulations in which each class is parameterised with a natural number indicating which nesting depths of **X** operators is preserved. For some applications, the notion of next-preserving bisimulation can be even more specific if we additionally parameterise the bisimulation with a sub-set of atomic propositions that are considered to be relevant. To be more specific, we can demand a model to be next-preserving bisimilar *with respect to a particular formula*. For example, in the domain of slicing it is common practice to produce a slice (which is a bisimilar system with less stuttering steps) specifically for a particular formula, which is computationally inexpensive, for the resulting model to be as small as possible.

To make the necessary changes to the definitions we need to reflect on the notion of *observable with respect to formula* φ. Generally, any change in the labelling of two states is observable. However, with respect to a formula φ only those atomic propositions are observable that are present in φ. Hence, we restrict the labelling of any state s, $\mathcal{L}(s)$ to $\mathcal{L}(s)|_\varphi$. Moreover, we extend the notion of bb-observable step $s \rightarrow s'$ with $s \not\triangleq s'$, to *bb-observable with respect to* φ, denoted as $s \not\triangleq_\varphi s'$, or if the step is labelled with an action, i.e., $s \xrightarrow{a} s'$, using the notation $obs_{\varphi\_bb}(a)$, i.e., $obs_{\varphi\_bb}(a)$ if and only if $s \not\triangleq_\varphi s'$.

The resulting definition is given as follows.

Definition 19 (Next-preserving Branching Bisimulation with respect to φ over states). *Let T_1, T_2 be doubly-labelled transition systems such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, for $i \in \{1, 2\}$, $S = S_1 \uplus S_2$ be the disjoint union of both sets of states, and φ a CTL* formula. A relation over states in S is a* next-preserving branching bisimulation with respect to φ, *denoted as npb_φ, if and only if it is symmetric and for states $s, t \in S$ with $npb_\varphi(s, t)$ the following holds*

- $\mathcal{L}_1|_\varphi(s) = \mathcal{L}_2|_\varphi(t)$ *and*
- *on any successor states from s we pose the following conditions*

  a.) for any $s' \in S$ such that $s \dashrightarrow s'$ and $npb_\varphi(s, s')$ it follows that $npb_\varphi(s', t)$

  b.) for any $j \in \mathbb{N}$ and $s', s'' \in S$ such that $s \dashrightarrow^j s' \xrightarrow{a} s''$ and for all $s_i \in \langle s \ldots s' \rangle$ $s_i \triangleq_\varphi s$ and $s' \not\triangleq_\varphi s''$, it follows that there exists $k \in \mathbb{N}$ and $t', t'' \in S$ such that $t \dashrightarrow^k t' \xrightarrow{a} t''$ and for all $t_i \in \langle t \ldots t' \rangle$ $t_i \triangleq_\varphi t$, and $npb_\varphi(s', t')$ as well as $npb_\varphi(s'', t'')$ and
  - *if $j < xdepth(\varphi)$ then $k = j$*
  - *if $j \geq xdepth(\varphi)$ then $k \geq xdepth(\varphi)$*

  and

- *if there exists an infinite stuttering path $s \dashrightarrow s_1 \dashrightarrow \ldots$, such that $npb_\varphi(s_i, t)$ for all $i > 0$, then there exists an infinite stuttering path $t \dashrightarrow t_1 \dashrightarrow \ldots$, such that $npb_\varphi(t_j, s_i)$ for all $i, j > 0$.*

Two states s and t are next-preserving branching bisimilar with respect to φ, *denoted as $s \overset{\star}{=}_\varphi t$, if and only if there exists a next-preserving branching bisimulation with respect to φ, npb_φ, such that $npb_\varphi(s, t)$.*

Next-preserving bisimulation with respect to φ over paths and whole systems can be defined in a similar fashion. Based on these new definitions we rephrase the notion of soundness in the following theorem.

Theorem 20 (Next-Preserving Branching Bisimulation with respect to φ preserves φ). *For two doubly-labelled transition systems T_1 and T_2, such that $T_i = (S_i, AP_i, I_i, \mathcal{L}_i, \mathcal{A}_i, \longrightarrow_i)$, for $i \in \{1, 2\}$ and any CTL* formula φ,*

$$T_1 \overset{\star}{=}_\varphi T_2 \;\Rightarrow\; (T_1 \vDash \varphi \Leftrightarrow T_2 \vDash \varphi)$$

The proof is similar to the proof of Theorem 18 and hence omitted here. In fact, the theorem can be generalised stating that for any CTL* formulas φ and ψ:

$$(\mathcal{L}_1|_\varphi = \mathcal{L}_1|_\psi \;\wedge\; xdepth(\varphi) = xdepth(\psi)) \;\Rightarrow\; (T_1 \overset{\star}{=}_\varphi T_2 \;\Rightarrow\; (T_1 \vDash \psi \Leftrightarrow T_2 \vDash \psi))$$

That is, two systems T_1 and T_2 for which $T_1 \overset{\star}{=}_\varphi T_2$ satisfy the same CTL* formulas that are in the same class of formulas as φ, where we consider a *class of formulas* to collect all formulas of the same $xdepth$ and containing the same atomic propositions.

2.8. A Hierarchy of Next-preserving Branching Bisimulations

The results from the previous sections lead to a hierarchy of next-preserving branching bisimulations as depicted in Figure 5.

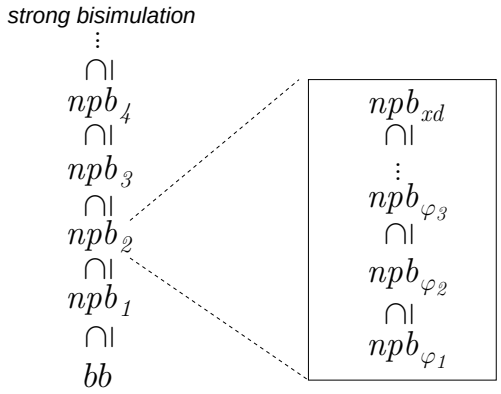

Figure 5: The hierarchy of next-preserving branching bisimulations

The order on the left depicts the hierarchy of next-preserving bisimulation of a certain depth. From Lemma 13 it is known that $bb \equiv npb_0$, which is the weakest next-preserving bisimulation in our hierarchy. Any increase in the *depth* which is taken into account results in a stronger bisimulation, as follows from Lemma 14. Eventually, the concept of next-preserving branching bisimulation coincides with strong bisimulation [1, 2], i.e., $npbb_\infty \equiv$ *strong bisimulation*.

When we zoom in to a next-preserving bisimulation of a particular depth (in the figure we chose $xd = 2$), we discover a hierarchy within this class which considers the next-preserving bisimulations with respect to particular classes of formulas. This is depicted on the right. Assuming that $xdepth(\varphi_1) = xdepth(\varphi_2) = xdepth(\varphi_3) = \ldots = xd$ and $\mathcal{L}|_{\varphi_1} \subseteq \mathcal{L}|_{\varphi_2} \subseteq \mathcal{L}|_{\varphi_3} \subseteq \ldots$ it follows that $npb_{\varphi_1} \supseteq npb_{\varphi_2} \supseteq npb_{\varphi_3}$ and for any φ_n such that $\mathcal{L}|_{\varphi_n} = \mathcal{L}$ we have $npb_{\varphi_n} \equiv npb_{xd}$.

This concludes the section on next-preserving branching bisimulations. The following section presents an application for this particular bisimulation, namely slicing.

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 [4]. It has been developed for various contexts, such as debugging and understanding of programs and for supporting program analysis. The reduced artefact, 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 [23, 24]) and lately also for a number of formal modelling notations (e.g., [25, 26, 27, 28, 29, 30, 31, 32, 33, 34]). Slicing of a formal model is of particular interest if an automated analysis of the model is intended, for example, using a model checker. Slicing as a reduction technique pushes the limits of a model checker, allowing larger models to be automatically analysed.

In [15] we have defined standard slicing for Behavior Tree models which preserves LTL_{-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 [35, 13] 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 [14] using an extension

[1] Geoff Dromey introduced the name Behavior Trees and chose the American spelling. Although this manuscript is written following the British spelling rules we respect Dromey's naming decision and deliberately sacrifice consistency at this point.

of CSP, called CSP$_\sigma$ [36]. 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 [37].

Details on the syntax of the notation and its well-formedness conditions can be found in [38]. Here we simply provide an example to give the reader a flavour of the notation. In Figure 6 an integrated tree is shown modelling the behaviour of a light switch system which can be operated manually or switched off by a timer.

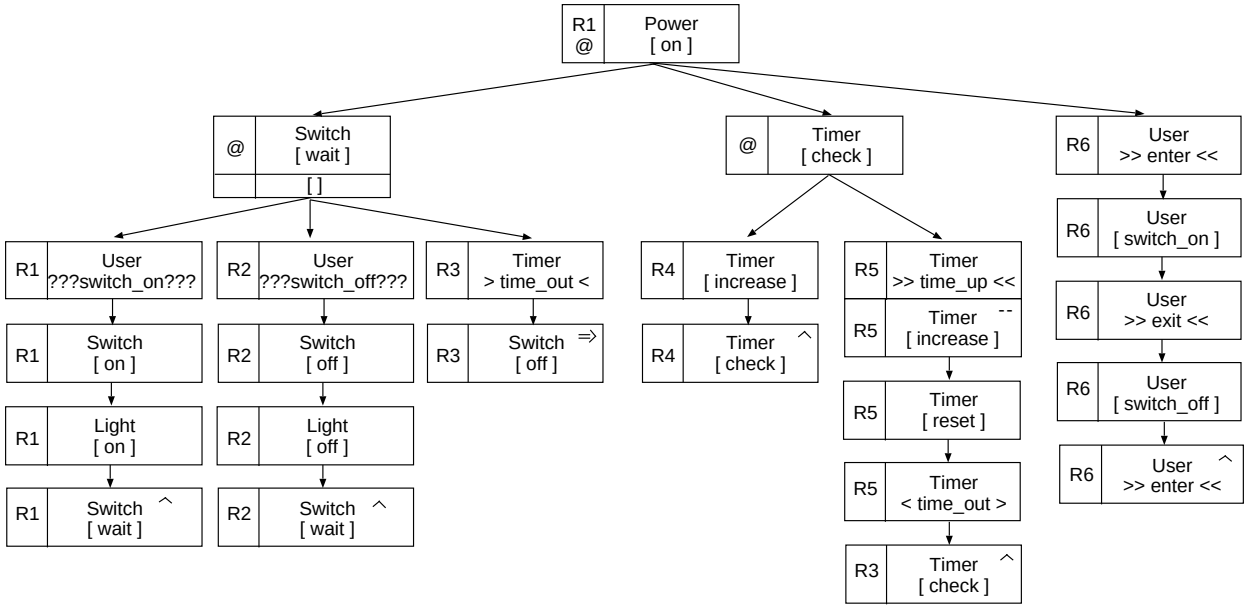

Figure 6: Example Behavior Tree modelling a hypothetical light switch system

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. 6), 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 '^'), 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 *refer-*

ence 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 indicating 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 Parameterisation. 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 parameterisation 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 $\mathcal{L}_B$. The set of actions in $\mathcal{L}_B$ 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, $s \xrightarrow{n} s'$ 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., $\mathcal{L}_B(s') = (\mathcal{L}_B(s) \oplus ctrlUpd(n)) \oplus update(n)$, where $ctrlUpd(n)$ 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[2]. All other nodes (i.e, nodes of type guard, selection, external input and internal input) function as a guard on the transition, i.e., $s \xrightarrow{n} s'$ can only occur if the pre-state s satisfies the condition imposed by the "guarding" node n, i.e., $guard(n) \in \mathcal{L}_B(s)$ 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., $\mathcal{L}_B(s') = \mathcal{L}_B(s) \oplus ctrlUpd(n)$.

[2]For two (partial) functions $f : A \to B$ and $g : A \to B$, the functional override $f \oplus g$ is defined as a function $h : A \to B$ such that $\forall a \in dom(g) \cdot h(a) = g(a) \ \wedge \ \forall b \notin dom(g) \cdot h(b) = f(b)$.

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 [39, 40]) 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 [15, 41]. The edges in the BTDG are denoted as $p \overset{d}{\rightarrowtail} q$ 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., $n_i \overset{d}{\rightarrowtail} n_{i+1}$.

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 φ. The formula φ gives rise to the *slicing criterion*, which is defined as the set of nodes which modify one of the variables included in φ. Let $Def(n)$ 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 $C_\varphi = \{n \mid \exists v \in vars(\varphi) \cdot v \in Def(n)\}$. 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 $m \notin C_\varphi$ are *stuttering* nodes. The function $obs_\varphi(n)$ 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* (with respect to φ) in $\mathcal{L}_B$. Similarly, observable nodes lead to steps performing an observable action, also referred to as *observable transitions*.

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 and – 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 outlaid above) can be further optimised using an algorithm for detecting infeasible paths (*infPath*), described in detail in [41]. 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:

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

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 [41, 15]. 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 critical branching point.

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 critical branching points. Locating observable transitions is straightforward, since these simply correspond to the observable nodes in the tree.

Finding the nodes corresponding to critical branching points requires a more careful examination of the various Behavior Tree constructs. A state before a critical branching point will 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 be classified as a *critical* branching point, one branch must lead to an observable node which is not possible on all paths via another branch, within the same number of steps (up to the nesting depth of the **X** operator in the formula considered). 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 be a critical branching point. 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 are not critical branching points, 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 preclude the execution of nodes in another branch. For example, in the BT on the left of Figure 7, if `B[f]` executes, it prevents `B?b?` and its descendent `C[c]` from executing. 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 7. 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 7 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.

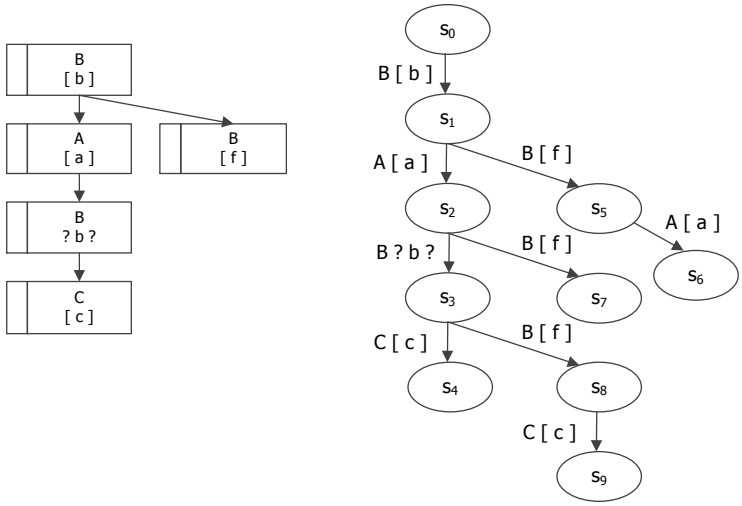

Figure 7: Example with a selection node

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 critical branching points. Recall that in the control flow graph, the false branch from an external input node loops back to the node again. Figure 8 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 be a critical branching point, since it does not have a successor from which all paths do not reach the observable node.

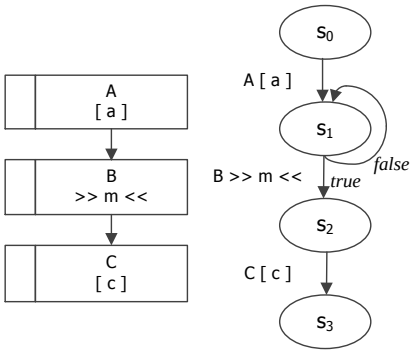

Figure 8: Example with an external input node

In summary, the only Behavior Tree constructs that produce critical branching points 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 critical branching point.

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 [41]. The rest of the algorithm searches for locations requiring extra stuttering nodes, specifically observable nodes and nodes corresponding to the steps before a critical branching point. For each of these locations, the algorithm selects suitable stuttering nodes to be returned to the slice. The algorithm maintains a set *finalSet*, which initially contains the nodes present in the normal slice. When the algorithm terminates, *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 *obs*. For each of these nodes, stuttering nodes are located, using the function *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 *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 *finalSet*. The observable node is marked as *visited* to prevent infinite traversals in later parts of the algorithm.

As discussed in the previous section, there are two situations which result in critical branching points 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 critical branching points, 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 *falseNodes*, the state (or states) in the transition system before it executes is a critical branching point, because there is a choice between a path where an observable node executes and another on which

Algorithm 1 Next-preserving slicing of Behavior Tree B

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

it does not. Therefore, the nodes which can execute immediately before each of the *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 *depChains* at line 18, which returns the set of such chains. The function *findFalseDep*, at line 19, returns the set of nodes that negate a condition which the observable node is dependent on. For example, in Figure 7, if C[c] is an observable node, then B[f] would be returned by the *findFalseDep* function.

For each of the *falseNodes*, the loop at lines 20 to 32 searches for nodes which can execute before it, given by the function *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 *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 *falseNode*. In the example, `B[b]` and `A[a]` are both such cases. For each of the nodes returned by *execBefore*, stuttering nodes are selected at lines 24 to 28 and stored in a temporary set, *tempSet*.

The process is then repeated for the new stuttering nodes, since including these nodes might result in a new critical branching point in the transition system. At each iteration, the nodes in the current working set are added to the *finalSet* and the working set is updated to *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 *finalSet* must then be re-formed into a syntactically correct Behavior Tree, according to the procedure described in [41]. 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 critical branching points.

5. Related Work

Our work builds on the definition of branching bisimulation [5], in particular its variant that takes infinitely stuttering paths into account, introduced in [5] as *divergence-sensitive branching bisimulation*. A similar bisimulation relation has also been introduced in [16] as *divergence-sensitive stuttering bisimulation*. Both notions are weak forms of bisimulation which 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 [42] define the notion of *orthogonal bisimulation equivalence*, which refines van Glabbeek and Weijland's branching bisimulation and allows a sequence of stuttering steps to be collapsed 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 [22] 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 relations preserve 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 bisimulations which are more specific and hence gives rise to an algorithmic approach to generate next-preserving branching bisimilar systems of particular depth.

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 [43] slice Object-Z for verification and in [31] 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 [44]. In their approach, the slice preserves formulas of a temporal logic which is invariant to stuttering (e.g., LTL_{-X}). Similarly, Rakow in [45] develops an approach for slicing Petri Nets which is guaranteed to preserve LTL_{-X} properties. Bordini et al. in [46] slice agent-based systems written in the AgentSpeak language and prove that their approach preserves LTL_{-X} using stuttering equivalence. Hatcliff et al. in [47] show the correctness of the Indus slicer, which slices Java programs, using a notion of projection similar to Weiser's projection [4] to demonstrate that their approach preserves LTL_{-X} properties.

Several other authors have proposed slicing approaches for reducing state explosion without providing full proofs of correctness. For example, Odenbrett et al. in [34] 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 [48]. They claim that CTL^*_{-X} properties are preserved, but the proof is left for future work. Also, Schäfer and Poetzsch-Heffter in [49] 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^* that does not contain the **U** or **X** operators, but details of the proof are not provided. Finally, van Langenhove and Hoojewijs in [50] present an approach for slicing UML models for verification. The approach is claimed to preserve LTL_{-X} properties, although again a full proof is not given.

6. Conclusion

In this paper we have proposed a novel notion of bisimulations, called a *family of next-preserving branching bisimulations*, 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. Each bisimulation in this family is parameterised by a natural number, referred to as its *depth*, which specifies the number of stuttering steps that are required to be present in both bisimilar systems. With this feature, the relation is stronger than weak forms of bisimulation but weaker than strong bisimulation. In particular, each bisimulation guarantees the preservation of temporal logic formulas containing next-step operators up to a nesting depth which coincides with the specified depth of the bisimulation.

The family of next-preserving bisimulations is useful in any application requiring 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 bisimulations, to reduce the size of models that we want to model check. The definition of the family of next-preserving branching bisimulations 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 which preserves satisfiability for any CTL* formula is novel and has not been proposed for any modelling or programming language previously.

For future work it would be interesting to investigate whether our notion of next-preserving branching bisimulation can be incorporated into the Sigref Bisimulation Tool Box [51]. This would provide us with a signature-based approach to compute a bisimilar model that preserves the next-step operator for a given class of formulas.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable and constructive feedback which prepared the ground for the final result. This work was supported by Australian Research Council (ARC) Linkage Grant LP0989643.

References

[1] D. Park, Concurrency and automata on infinite sequences, in: GI Conference on Theoretical Computer Science, Vol. 104 of Lecture Notes in Computer Science, 1981, pp. 167–183.

[2] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[3] E. A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Elsevier Science Publishers, 1990, pp. 995–1072.

[4] M. Weiser, Program slicing, in: Proc. of Int. Conf. on Software Engineering (ICSE'81), 1981, pp. 439–449.

[5] R. J. van Glabbeek, W. P. Weijland, Branching time and abstraction in bisimulation semantics, Journal of the ACM 43 (3) (1996) 555–600.

[6] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M. B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Trans. on Programming Languages and Systems 29 (5) (2007) 1–43 (Article 27).

[7] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.

[8] E. Clarke, E. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Logics of Programs, Vol. 131 of Lecture Notes in Computer Science, Springer-Verlag, 1982, pp. 52–71.

[9] J. Queille, J. Sifakis, Specification and verification of concurrent systems in cesar, in: International Symposium on Programming, Vol. 137 of Lecture Notes in Computer Science, Springer-Verlag, 1982, pp. 337–351.

[10] L. Grunske, P. A. Lindsay, N. Yatapanage, K. Winter, An automated failure mode and effect analysis based on high-level design specification with Behavior Trees, in: J. Romijn, G. Smith, J. van de Pol (Eds.), Proc. of Int Conf. on Integrated Formal Methods (IFM 2005), Vol. 3771 of Lecture Notes in Computer Science, Springer, 2005, pp. 129–149.

[11] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, P. A. Lindsay, Experience with fault injection experiments for FMEA, Software: Practice and Experience 41 (11) (2011) 1233–1258.

[12] P. Lindsay, N. Yatapanage, K. Winter, Cut set analysis using Behavior Trees and model checking, Formal Aspects of Computing 24 (2) (2012) 249–266.

[13] R. G. Dromey, Genetic design: Amplifying our ability to deal with requirements complexity, in: Scenarios: Models, Transformations and Tools, Vol. 3466 of Lecture Notes in Computer Science, Springer, 2005, pp. 95–108.

[14] R. J. Colvin, I. J. Hayes, A semantics for Behavior Trees using CSP with specification commands, Science of Computer Programming 76 (10) (2011) 891–914.

[15] N. Yatapanage, K. Winter, S. Zafar, Slicing Behavior Tree models for verification, in: Theoretical Computer Science, Vol. 323 of IFIP Advances in Information and Communication Technology, Springer-Verlag, 2010, pp. 125–139.

[16] R. de Nicola, F. Vaandrager, Three logics for branching bisimulation, Journal of the ACM 42 (2) (1995) 458–487.
[17] E. A. Emerson, J. Y. Halpern, "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic, Journal of the ACM 33 (1) (1986) 151–178.
[18] R. J. van Glabbeek, B. Luttik, N. Trčka, Branching bisimilarity with explicit divergence, Fundamenta Informaticae 93 (4) (2009) 371–392.
[19] R. J. van Glabbeek, B. Luttik, N. Trčka, Computation tree logic with deadlock detection, Logical Methods in Computer Science 5 (4:5) (2009) 1–24.
[20] L. Lamport, What good is temporal logic?, in: IFIP Congress on Information Processing, 1983, pp. 657–668.
[21] A. Antonik, M. Huth, Efficient patterns for model checking partial state spaces in CTL *intersection* LTL, Electronic Notes in Theoretical Computer Science 158 (2006) 41–57.
[22] A. Kučera, J. Strejček, The stuttering principle revisited, Acta Informatica 41 (7-8) (2005) 415–434.
[23] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.
[24] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.
[25] T. Oda, K. Araki, Specification slicing in formal methods of software development., in: Proc. of Computer Software and Applications Conference (COMSAC 93), IEEE, 1993, pp. 313–319.
[26] M. Heimdahl, M. Whalen, Reduction and slicing of heirarchical state machines, in: M. Jazayeri, H. Schauer (Eds.), Proc. of European Software Engineering Conference (ESEC '97/FSE-5), Vol. 1301 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 450–467.
[27] L. I. Millett, T. Teitelbaum, Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation, Int. Journal on Software Tools for Technology Transfer 2 (4) (2000) 343–349.
[28] V. Ganesh, H. Saidi, N. Shankar, Slicing SAL, Tech. rep., Computer Science Laboratory (1999).
[29] B. Korel, I. Singh, L. Tahat, B. Vaysburg, Slicing of state-based models, in: Proc. of Int. Conf. on Software Maintenance (ICSM 2003), IEEE, 2003, pp. 34–43.
[30] F. Wu, T. Yi, Slicing Z specifications, ACM SIGPLAN Notices 39 (8) (2004) 39–48.
[31] I. Brückner, H. Wehrheim, Slicing an integrated formal method for verification, in: K.-K. Lau, R. Banach (Eds.), Proc. of Int. Conf. on Formal Methods and Software Engineering (ICFEM'05), Vol. 3785 of Lecture Notes in Computer Science, Springer, 2005, pp. 360–374.
[32] S. Labbe, J.-P. Gallois, M. Pouzet, Slicing communicating automata specifications for efficient model reduction, in: J. Grundy, J. Han (Eds.), Proc. of Australian Software Engineering Conference (ASWEC 2007), IEEE Computer Society Press, 2007, pp. 191–200.
[33] C. Thrane, Slicing for UPPAAL, in: Ann. IEEE Conf. (Student Paper), IEEE, 2008, pp. 1–5.
[34] M. Odenbrett, V. Y. Nguyen, T. Noll, Slicing AADL specifications for model checking, in: C. Muñoz (Ed.), NASA Formal Methods, Vol. NASA/CP-2010-216215 of NASA Conference Proceedings, 2010, pp. 217–221.
[35] R. G. Dromey, From requirements to design: Formalizing the key steps, in: Proc. of Software Engineering and Formal Methods (SEFM 2003), IEEE Computer Society, 2003, pp. 2–11.
[36] R. Colvin, I. J. Hayes, CSP with hierarchical state, in: M. Leuschel, H. Wehrheim (Eds.), Proc. of Int. Conf. on Integrated Formal Methods (IFM'09), Vol. 5423 of Lecture Notes in Computer Science, Springer, 2009, pp. 118–135.
[37] K. Winter, I. J. Hayes, R. Colvin, Integrating requirements: The Behavior Tree philosophy, in: Proc. of Int. Conf. on Software Engineering and Formal Methods (SEFM 2010), IEEE Computer Society Press, 2010, pp. 41 – 50.
[38] L. Grunske, K. Winter, N. Yatapanage, Defining the abstract syntax of visual languages with advanced graph grammars-a case study based on Behavior Trees, Journal of Visual Language and Computing 19 (3) (2008) 343–379.
[39] K. J. Ottenstein, L. M. Ottenstein, The program dependence graph in a software development environment, ACM SIGSOFT Software Engineering Notes 9 (3) (1984) 177–184.
[40] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Trans. on Programming Languages and Systems 9 (3) (1987) 319–349.
[41] N. Yatapanage, Slicing Behavior Trees for verification of large systems, Ph.D. thesis, Institute for Integrated and Intelligent Systems, Griffith University, https://sites.google.com/site/nisansalayatapanage/thesis (2012).
[42] J. A. Bergstra, A. Ponse, M. B. van der Zwaag, Branching time and orthogonal bisimulation equivalence, Theoretical Computer Science 309 (1-3) (2003) 313–355.
[43] I. Brückner, H. Wehrheim, Slicing Object-Z specifications for verification, in: H. Treharne, S. King, M. C. Henson, S. A. Schneider (Eds.), Proc. of Int. Conf. on Formal Specification and Development in Z and B (ZB'05), Vol. 3455 of Lecture Notes in Computer Science, Springer, 2005, pp. 414–433.
[44] I. Brückner, Slicing concurrent real-time system specifications for verification, in: J. Davies, J. Gibbons (Eds.), Proc. of Integrated Formal Methods (IFM'07), Vol. 4591 of Lecture Notes in Computer Science, Springer, 2007, pp. 54–74.
[45] A. Rakow, Slicing Petri Nets with an application to workflow verification, in: V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, M. Bieliková (Eds.), Proc. of Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'08), Vol. 4910 of Lecture Notes in Computer Science, Springer, 2008, pp. 436–447.
[46] R. H. Bordini, M. Fisher, M. Wooldridge, W. Visser, Property-based slicing for agent verification, Journal of Logic and Computation 19 (6) (2009) 1385–1425.
[47] J. Hatcliff, M. B. Dwyer, H. Zheng, Slicing software for model construction, Higher-Order and Symbolic Computation 13 (4) (2000) 315–353.
[48] G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[49] I. Schäfer, A. Poetzsch-Heffter, Slicing for model reduction in adaptive embedded systems development, in: Proc. of Int. Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS'08), ACM, New York, NY, USA, 2008, pp. 25–32.
[50] S. van Langenhove, A. Hoogewijs, SV_tL: System verification through logic tool support for verifying sliced hierarchical statecharts, in: J. L. Fiadeiro, P.-Y. Schobbens (Eds.), Proc. on Int. Workshop on Recent Trends in Algebraic Development Techniques (WADT'06), Vol. 4409 of Lecture Notes in Computer Science, Springer, 2006, pp. 142–155.
[51] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, B. Becker, Sigref – a symbolic bisimulation tool box, in: S. Graf, W. Zhang (Eds.), Automated Technology for Verification and Analysis (ATVA 2006), Vol. 4218 of Lecture Notes in Computer Science, 2006, pp. 477–492.