Refinement Relations on Partial Specifications

by

Shiva Nejati

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Refinement Relations on Partial Specifications

Shiva Nejati

Master of Science

Graduate Department of Computer Science

University of Toronto

2003

We discuss the problem of refining partial specifications of software systems at different levels of abstraction. In general, refinement is the process of deriving an implementation from a specification and verifying the correctness of the derivation. Recently, partial specifications have been advocated for describing software systems mainly because they do not impose a commitment to all decisions made at initial stages of software development life-cycle.

Using finite-state transition systems with partial transitions and propositions as our modeling formalism, we define a refinement relation that is insensitive to finite stuttering. We refer to our proposed refinement relation as *stuttering refinement relation*. We, then, present a logical characterization of this refinement relation and describe an algorithm for computing it.

Contents

List of Figures

Chapter 1

Introduction

Reactive systems are those which maintain ongoing interactions with their environments [MP92]. Examples of reactive systems include air traffic control systems, medical systems, and network protocols. The process of developing a reactive system can be logically divided into three major phases: *specification*, *implementation*, and *verification* [Lar89]. The specification of a system is a high-level description of the requirements of that system, and the implementation is a low-level program or process that realizes the specification. Both the specification and the implementation of a reactive system are usually expressed in terms of state transition systems.

The verification phase is a *formal proof of correctness* of the implementation with respect to the specification. Many notions of correctness are based on finding a relation between the state-space of the implementation and that of the specification, and then showing that the relation preserves the logical properties of the specification in the implementation. Such a relation is referred to as a *refinement relation* [AL91].

In the existing frameworks that follow the above-mentioned paradigm, traditional formalisms (e.g. Kripke Structures, Labeled Transition Systems) are typically used for representing specifications and implementations; and the verification phase is usually based on relating an implementation to a specification through a simulation relation. In

a simulation relation, it is asserted that every behavior that is possible in the implementation of a system is also possible in its specification. In other words, the implementation is only capable of exhibiting those behaviors that are allowed by the specification; therefore, an abstract specification may allow behaviors that are not realized by a concrete implementation.

The frameworks based on traditional formalisms have three major drawbacks: Firstly, the scope of simulation-based verification is restricted to universal properties. Secondly, although the specification can fix the borders that the implementation is not permitted to cross, it cannot describe the obligations of the implementation. Thirdly, we need to use a modeling formalism that is capable of describing *partial* or *uncertain* behaviors. This is because the implementation of a large system may not be immediately derivable from the initial specification; rather, the implementation phase may consist of a series of small and successive refinement steps (the so-called stepwise-refinement [Lar89]). Therefore, there may be some incomplete or partial behaviors in the state transition systems corresponding to the intermediate refinement steps.

Partial specifications [LT88, Lar89, HJS01, HJS02] are desirable because they do not impose a commitment to all decisions made at initial stages of software development life-cycle. Moreover, partial models can distinguish between *guaranteed behaviors*, i.e. what is required, and *admissible behaviors*, i.e. what is permitted. Thus, we can enforce the realization of guaranteed behaviors in the implementation and simultaneously, ensure that the implementation does not violate admissible behaviors. The other advantage of partial modeling formalisms is that refinement relations defined over partial models preserve both existential and universal properties. Hence, the verification of an implementation against a specification can be performed in a more detailed fashion.

In this thesis, we discuss the problem of refining partial specifications of software systems at different levels of abstraction. We show how to establish a refinement relation between a specification, which is given as a partial state transition system with some

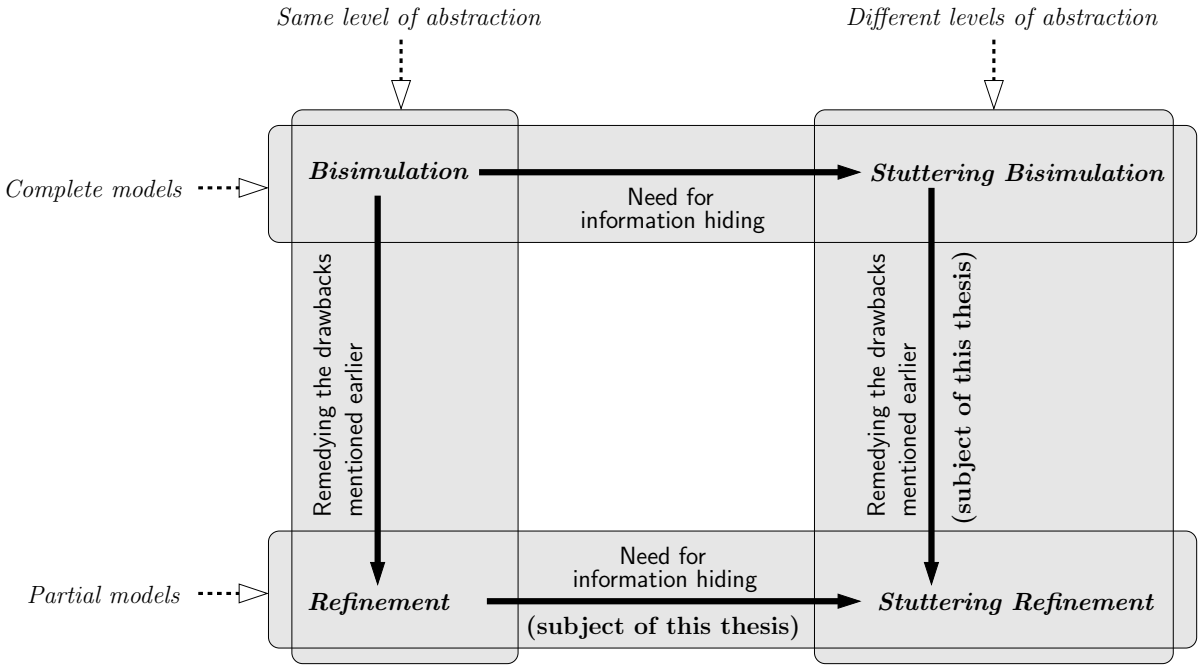

Figure 1.1: The relationships between refinement relations

variables and transitions marked as incomplete, and an implementation, which takes the form of a complete or fully-defined state transition system. In general, the verification of an implementation against a specification involves comparing a pair of models at different levels of abstraction, where a single transition in the higher-level model may correspond to several transitions in the lower-level one [AL91]. For this reason, the refinement relation that we consider is insensitive to *finite stuttering*. We refer to this refinement relation as *stuttering refinement relation*. This refinement relation is defined in such a way that the guaranteed behaviors in a specification simulate those in the refined system, and the admissible behaviors in the refined system simulate those in the specification.

Figure 1.1 illustrates the relationships between the stuttering refinement relation and some of the existing refinement relations. We briefly describe these relationships through some examples.

Figure 1.2 shows two complete state transition systems that are bisimilar. As seen in

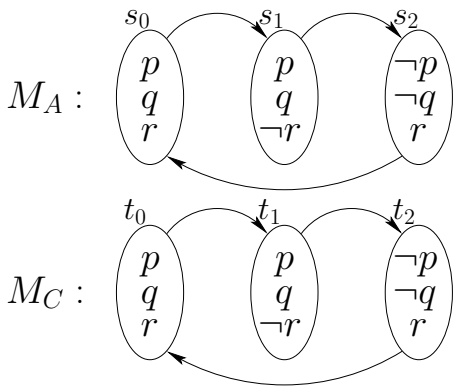

Figure 1.2: An example of bisimulation relation over complete models

the figure, the two models exhibit the same behaviors. Thus, both truth and falsehood are preserved, and therefore, the analysis can be performed in a very detailed fashion. However, in a bisimulation relation, the involved models are indeed equal. As a result, bisimulation cannot be defined over models at different levels of abstraction. Furthermore, since models are complete, they cannot describe incompleteness and uncertainty.

Figure 1.3 shows an example of refinement relation [LT88, Lar89, HJS01] over partial models. In this figure, M_A is a partial model, and M_C is a refinement of M_A. In M_A, states s_1 and s_2 have unknown variables causing M_A to have some behaviors that can neither be accepted nor refuted. In a refinement relation, it is asserted that every accepted (resp. refuted) behavior of M_A is necessarily accepted (resp. refuted) in M_C, but no assumption can be made about unknown behaviors of M_A. Therefore, refinement relation preserves both truth and falsehood. However, there are some unknown properties in M_A that have an arbitrary value in M_C. A shortcoming of refinement relations over partial models is that they assume models to be at the same level of abstraction. Therefore, the size of the abstract model is not necessarily minimal.

Figure 1.4 shows an example of stuttering bisimulation relation over complete models. As seen in the figure, M_A does not observe the variable r. Thus, there are some states and transitions in M_C that are missing in M_A. A stuttering bisimulation relation is

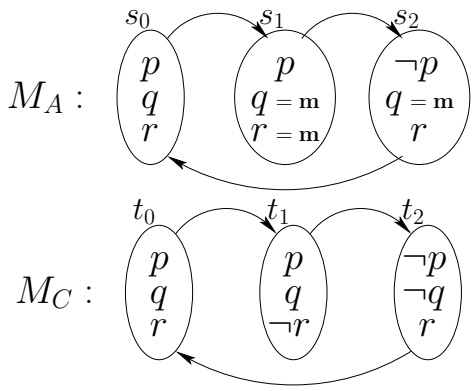

Figure 1.3: An example of refinement relation over partial models

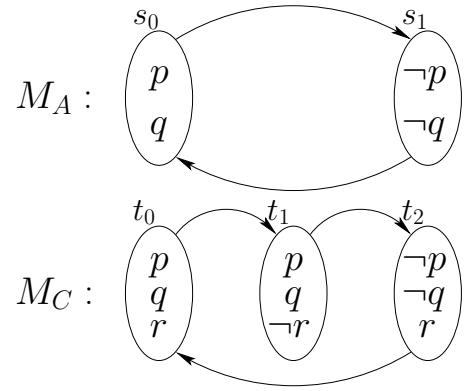

Figure 1.4: An example of stuttering bisimulation relation over complete models

a bisimulation relation that is defined over models at different levels of abstraction. However, since stuttering bisimulation is defined over complete models, we can only hide a fixed set of variables in all states of a model. More precisely, even if we want to abstract from a variable in a few states of a model, we have to hide it in all states of that model.

Figure 1.5 shows an example of stuttering refinement over partial models. This relation can be studies from two different prospectives:

- A stuttering refinement relation is a stuttering bisimulation relation that is defined over partial models. Thus, as shown in Figure 1.5, different sets of variables can be hidden in different states. For example, in state s_0, variable r is hidden, whereas,

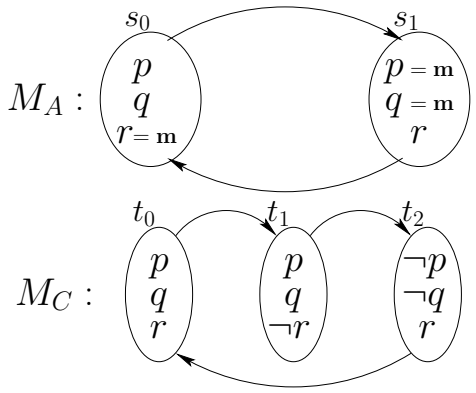

Figure 1.5: An example of stuttering refinement relation over partial models

in state s_1, variables p and q are hidden.

- A stuttering refinement relation is a refinement relation over partial models that can deal with models at different levels of abstraction. That is a state in the abstract model may be realized by a sequence of states in the concrete model. For example, in Figure 1.5, state s_0 (in M_A) is realized by a sequence, $t_0 \rightarrow t_1$, of states (in M_C).

Stuttering refinement has all the advantages of the existing refinement relations: it preserves both truth and falsehood; it relates models at different levels of abstraction; and it is defined over partial models.

1.1 Related work

The theory of *Modal Specifications* has been proposed in [LT88, Lar89]. Modal specifications have been specifically designed to provide a means for describing partial specifications. In [LT88, Lar89], Modal specifications are expressed in terms of Modal Transition Systems which are Labeled Transition Systems with two types of transitions: *may*-transitions and *must*-transitions. In Chapter 3, we transfer the results and the refinement relation in [LT88, Lar89] to a state-based setting where a simplified version of Kripke

Modal Transition Systems [HJS01] are used as the modeling formalism.

Stuttering bisimulation has originally been introduced in [BCG88]. A structural and inductive definition of stuttering bisimulation is given in [NV95]. In Chapter 4, we lift the definition of stuttering bisimulation in [NV95] from the context of traditional modeling formalisms to the context of partial modeling formalisms and obtain a divergence-sensitive refinement relation on partial specifications that takes finite stuttering into account.

1.2 Structure of the thesis

The remainder of this thesis is organized as follows: Chapter 2 covers the preliminaries including relations and functions, lattice theory, transition systems, temporal logics, and behavioral preorders and equivalences. Chapter 3 outlines the refinement relation over partial modeling formalisms that has been explained in [HJS01], and is, indeed, a state-based version of the refinement relation proposed in [LT88, Lar89]. Chapter 4 discusses our proposed notion of stuttering simulation and stuttering refinement on partial models. The chapter also includes logical characterization results of these refinement relations and presents two algorithms one of which computes stuttering simulation and the other of which computes stuttering refinement. Finally, Chapter 5 presents our conclusions and future work.

Chapter 2

Preliminaries

In this chapter, we discuss notational issues and present some background information.

2.1 Relations and functions

Let Σ_1 and Σ_2 be sets. A *binary relation* $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is defined as a set of pairs. We often write binary relations in infix notation, i.e. $(s,t) \in \mathcal{R}$ is written as $s\mathcal{R}t$. A binary relation $\mathcal{R} \subseteq \Sigma \times \Sigma$ is

- *reflexive* if $\forall s \in \Sigma \cdot s\mathcal{R}s$.
- *symmetric* if $\forall s,t \in \Sigma \cdot s\mathcal{R}t \Rightarrow t\mathcal{R}s$.
- *anti-symmetric* if $\forall s,t \in \Sigma \cdot s\mathcal{R}t \ \wedge \ t\mathcal{R}s \Rightarrow s = t$.
- *transitive* if $\forall s,t,r \in \Sigma \cdot s\mathcal{R}t \ \wedge \ t\mathcal{R}r \Rightarrow s\mathcal{R}r$.

A binary relation is a *preorder relation* if it is reflexive and transitive. A preorder relation that is also symmetric is an *equivalence relation*. A preorder relation that is anti-symmetric is a *partial order relation*. Let $\leq_1$ be a partial order on Σ_1, and let $\leq_2$ be a partial order on Σ_2. A function $f : \Sigma_1 \rightarrow \Sigma_2$ is said to be *monotonic* if $s \leq_1 t \Rightarrow f(s) \leq_2 f(t)$ for every $s,t \in \Sigma_1$.

Definition 2.1 *A binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is image finite if and only if for every $s \in \Sigma_1$, the set $\{t \in \Sigma_2 \mid s\mathcal{R}t\}$ is finite.*

2.2 Lattice theory and fixpoint theorem

Let S be a set and let $\sqsubseteq$ be a partial order on S. The pair $(S, \sqsubseteq)$ is called a *partially ordered set* or *poset*, for short. Let $(S, \sqsubseteq)$ be a poset, $T \subseteq S$, and $s, s' \in S$. When $s \sqsubseteq s'$, we say that s is *below* s' or s' is *above* s. An element $s \in S$ is a *lower bound* for T if and only if s is below all elements of T; it is an *upper bound* for T if and only if it is above all elements of T. A lower bound for T is the *greatest lower bound* (glb) for T if and only if it is above any lower bound for T. An upper bound for T is the *least upper bound* (lub) for T if and only if it is below any upper bound for T. A poset $(S, \sqsubseteq)$ is a *complete lattice* if and only if any subset T of S has an lub and a glb in S.

For a function $f : S \to S$, an element $x \in S$ is a *fixed point (fixpoint)* of f if and only if $f(x) = x$. Theorems which produce fixpoints of certain maps have been extensively used in computer science. The following theorem expresses one of the most well-known fixpoint theorems:

Theorem 2.2 (Knaster-Tarski fixpoint theorem) *[DP02] Let $(S, \sqsubseteq)$ be a complete lattice and let $f : S \to S$ a monotonic function. Then*

$$l = \bigsqcup\{x \in S \mid x \sqsubseteq f(x)\}$$

is a fixpoint of f. Further, l is the greatest fixpoint (gfp) of f. Dually, f has a least fixpoint (lfp), given by $\bigsqcap\{x \in S \mid f(x) \sqsubseteq x\}$.

2.3 Transition systems

A *transition system* is a tuple $M = (\Sigma, \rightarrow)$ consisting of a set Σ of states and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. A transition system is *total* if for every $s \in \Sigma$, there exists $t \in \Sigma$

such that $s \rightarrow t$. For the sake of brevity, we sometimes write $\forall s \rightarrow s'$ (resp. $\exists s \rightarrow s'$) when we mean $\forall s' \in \Sigma \cdot s \rightarrow s'$ (resp. $\exists s' \in \Sigma \cdot s \rightarrow s'$).

A *path* in a transition system M is an *infinite* sequence $\bar{s} = s_0 s_1 \cdots$ of states such that $\forall i \in \mathbb{N} \cdot s_i \rightarrow s_{i+1}$. A *prefix* in a transition system M is a *finite* sequence $\bar{s}_{prefix} = s_0 s_1 \cdots s_k$ $(k \geq 0)$ of states such that $\forall 0 \leq i < k \cdot s_i \rightarrow s_{i+1}$. A subsequence of $\bar{s}$ or $\bar{s}_{prefix}$ is called a *block*. A *partitioning* of $\bar{s}$ or $\bar{s}_{prefix}$ is a (finite or infinite) sequence $B_0 = \{s_0, \cdots, s_{i_0}\}, B_1 = \{s_{i_0+1} \cdots s_{i_1}\}, \cdots$ of blocks of $\bar{s}$ or $\bar{s}_{prefix}$. A partitioning of $\bar{s}_{prefix}$ is always finite. When a partitioning of $\bar{s}$ is finite, the last block of the partitioning is infinite. A path $\bar{s}$ starts with a prefix of $\bar{s}$ and continues with an infinite path called the *suffix* of $\bar{s}$. For $s \in \Sigma$, a (M, s)-*path* (or an s-*path* when M is clear from the context) is a path in M that starts from s; similarly for prefixes. We use $paths(s)$ to denote the set of all s-paths and $prefixes(s)$ to denote all the s-prefixes.

Definition 2.3 *M is called finitely branching if and only if the transition relation $\rightarrow$ is image finite.*

A transition system extended with a set Ap of atomic propositions and a labeling function $I : \Sigma \times Ap \rightarrow \{\top, \bot\}$ is a *Kripke Structure*. The labeling function specifies what propositions hold in each state. A transition system $M = (\Sigma, \rightarrow, Act)$ where Act is a set of action symbols and the transition relation $\rightarrow$ is a subset of $\Sigma \times Act \times \Sigma$ is called a *Labeled Transition System (LTS)*.

Transition systems have been extensively used in systems specification. In general, there are two popular approaches to specification: *state-based* and *action-based*. In the state-based approach, an execution of a system is viewed as a sequence of states in which every state is an assignment of values to some set of propositions. The action-based approach views an execution as a sequence of actions. Kripke Structures fall into the former approach, while LTSs fall into the latter.

A *Kripke Modal Transition System (KMTS)* [HJS01] is a transition system

$M = (\Sigma, \rightarrow^{must}, \rightarrow^{may}, I^{must}, I^{may}, Ap)$ where both $M_{must} = (\Sigma, \rightarrow^{must}, I^{must}, Ap)$ and $M_{may} = (\Sigma, \rightarrow^{may}, I^{may}, Ap)$ are Kripke Structures subject to the constraints that $\rightarrow^{must} \subseteq \rightarrow^{may}$ and $\forall p \in Ap \cdot I^{must}(s,p) \Rightarrow I^{may}(s,p)$. Our definition of KMTS is a simplified version of that introduced in [HJS01]. In [HJS01], KMTSs are *modal Doubly Labeled Transition Systems* [NV95]. Doubly labeled Transition Systems combine the features of Labeled Transitions Systems and Kripke Structures. In our definition, however, KMTSs are *modal Kripke Structures*. More precisely, in our definition, the set of action symbols, Act, does not exist.

Let $M = (M_{must}, M_{may})$ be a KMTS. A path in M_{must} is called a *must-path* in M. Dually, a path in M_{may} is called a *may-path* in M. Since $M_{must} \subseteq M_{may}$, every must-path is a may-path as well, but not vice versa; similarly for prefixes and transitions.

The following two relations define two preorder relations over the set of states in Kripke Structures and KMTSs based on the labeling functions:

Definition 2.4 *Let $M = (\Sigma, \rightarrow, I, Ap)$ be a Kripke Structure, and let $s, t \in \Sigma$. The preorder relation $\mathcal{R}_0 \subseteq \Sigma \times \Sigma$ is a binary relation such that $s\mathcal{R}_0 t$ if and only if:*

- $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t,p) = \top$

Definition 2.5 *Let $M = (M_{must}, M_{may})$ be a KMTS, and let $s, t \in \Sigma$. A preorder relation $\preceq_0 \subseteq \Sigma \times \Sigma$ is a binary relation such that $t \preceq_0 s$ if and only if:*

- $\forall p \in Ap \cdot I^{must}(s,p) = \top \Rightarrow I^{must}(t,p) = \top$
- $\forall p \in Ap \cdot I^{may}(t,p) = \top \Rightarrow I^{may}(s,p) = \top$

2.4 Temporal logics

A major ingredient of every formal method is its approach to expressing system properties. In reactive systems, we typically use *temporal logics* to express systems properties.

Examples of temporal logics include μ-calculus [Koz83], CTL* [EH86], CTL [CES86], and LTL [Pnu77]. μ-calculus, CTL, and CTL* are branching-time logics, while LTL is a linear-time logic. In this thesis, we use CTL to express properties of Kripke Structures and KMTSs. CTL is able to express both *universal* properties, i.e. properties that have to hold along all paths, and *existential* properties, i.e. properties that have to hold along some path, as well as *safety* properties, i.e. nothing bad may happen, and *liveness* properties, i.e. something good has to happen.

Definition 2.6 *The abstract syntax of the logic CTL is inductively defined as follows:*

$$\begin{aligned}\phi \;::=\; & \top \mid \bot \mid p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid EX\phi \mid AX\phi \mid \\ & EF\phi \mid AF\phi \mid E[\phi U \phi] \mid A[\phi U \phi] \mid EG\phi \mid AG\phi\end{aligned}$$

where $p \in Ap$.

The following definition describes the semantics of CTL on Kripke Structures:

Definition 2.7 *For a Kripke Structure M and a CTL formula ϕ, $[\![\phi]\!] \subseteq \Sigma$ is defined as follows:*

$$\begin{aligned}
[\![\top]\!] &\triangleq \Sigma \\
[\![\bot]\!] &\triangleq \emptyset \\
[\![p]\!] &\triangleq \{s \in \Sigma \mid I(s,p) = \top\} \\
[\![\phi_1 \wedge \phi_2]\!] &\triangleq [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![\phi_1 \vee \phi_2]\!] &\triangleq [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
[\![\neg\phi]\!] &\triangleq \Sigma \setminus [\![\phi]\!] \\
[\![EX\phi]\!] &\triangleq \{s \in \Sigma \mid \exists s' \in \Sigma \cdot (s \rightarrow s' \ \wedge \ s' \in [\![\phi]\!])\} \\
[\![EG\phi]\!] &\triangleq \{s \in \Sigma \mid \exists \bar{s} \in \mathit{paths}(s) \cdot \forall s_i \in \bar{s} \cdot s_i \in [\![\phi]\!]\} \\
[\![E[\phi U \psi]]\!] &\triangleq \{s \in \Sigma \mid \exists \bar{s}_{\mathit{prefix}} \in \mathit{prefixes}(s) \cdot \exists s_j \in \bar{s}_{\mathit{prefix}} \cdot (s_j \in [\![\psi]\!] \wedge \\
&\qquad \forall s_i \in \bar{s}_{\mathit{prefix}} \cdot (i < j \Rightarrow s_i \in [\![\phi]\!]))\}
\end{aligned}$$

The remaining operators are defined as follows:

$$
\begin{aligned}
[\![A[\phi U \psi]]\!] &\triangleq [\![\neg E[\neg\psi U \neg\phi \wedge \neg\psi] \wedge \neg EG\neg\psi]\!] \\
[\![AX\phi]\!] &\triangleq [\![\neg EX\neg\phi]\!] \\
[\![AF\phi]\!] &\triangleq [\![A[\top U \phi]]\!] \\
[\![EF\phi]\!] &\triangleq [\![E[\top U \phi]]\!] \\
[\![AG\phi]\!] &\triangleq [\![\neg EF\neg\phi]\!]
\end{aligned}
$$

The following definition gives the semantics for CTL on KMTSs. This semantics coincides exactly with the multi-valued semantics of CTL proposed in [CDEG03] when the truth set is the 3-valued Kleene logic [Kle52].

Definition 2.8 *Let $M = (M_{must}, M_{may})$ be a KMTS, and let ϕ be a CTL formula. We define $[\![\phi]\!]^{\top} \subseteq \Sigma$, i.e. the set of states that satisfy ϕ, and $[\![\phi]\!]^{\perp} \subseteq \Sigma$, i.e. the set of states that refute ϕ, as follows:*

$$
\begin{aligned}
[\![\top]\!]^{\top} &\triangleq \Sigma \\
[\![\bot]\!]^{\top} &\triangleq \emptyset \\
[\![p]\!]^{\top} &\triangleq \{s \in \Sigma \mid I^{must}(s,p) = \top\} \\
[\![\phi_1 \wedge \phi_2]\!]^{\top} &\triangleq [\![\phi_1]\!]^{\top} \cap [\![\phi_2]\!]^{\top} \\
[\![\phi_1 \vee \phi_2]\!]^{\top} &\triangleq [\![\phi_1]\!]^{\top} \cup [\![\phi_2]\!]^{\top} \\
[\![\neg\phi]\!]^{\top} &\triangleq [\![\phi]\!]^{\perp} \\
[\![EX\phi]\!]^{\top} &\triangleq \{s \in \Sigma \mid \exists s' \in \Sigma \cdot (s \rightarrow^{must} s' \ \wedge \ s' \in [\![\phi]\!]^{\top})\} \\
[\![EG\phi]\!]^{\top} &\triangleq \{s \in \Sigma \mid \exists \bar{s} \in \textit{must-paths}(s) \cdot \forall s_i \in \bar{s} \cdot s_i \in [\![\phi]\!]^{\top}\} \\
[\![E[\phi U \psi]]\!]^{\top} &\triangleq \{s \in \Sigma \mid \exists \bar{s}_{\textit{prefix}} \in \textit{must-prefixes}(s) \cdot \exists s_j \in \bar{s}_{\textit{prefix}} \cdot (s_j \in [\![\psi]\!]^{\top} \wedge \\
&\qquad \forall s_i \in \bar{s}_{\textit{prefix}} \cdot (i < j \Rightarrow s_i \in [\![\phi]\!]^{\top}))\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\top]\!]^{\perp} &\triangleq \emptyset \\
[\![\perp]\!]^{\perp} &\triangleq \Sigma \\
[\![p]\!]^{\perp} &\triangleq \{s \in \Sigma \mid I^{may}(s,p) = \perp\} \\
[\![\phi_1 \wedge \phi_2]\!]^{\perp} &\triangleq [\![\phi_1]\!]^{\perp} \cap [\![\phi_2]\!]^{\perp} \\
[\![\phi_1 \vee \phi_2]\!]^{\perp} &\triangleq [\![\phi_1]\!]^{\perp} \cup [\![\phi_2]\!]^{\perp} \\
[\![\neg\phi]\!]^{\perp} &\triangleq [\![\phi]\!]^{\top} \\
[\![EX\phi]\!]^{\perp} &\triangleq \{s \in \Sigma \mid \forall s' \in \Sigma \cdot (s \rightarrow^{may} s' \Rightarrow s' \in [\![\phi]\!]^{\perp})\} \\
[\![EG\phi]\!]^{\perp} &\triangleq \{s \in \Sigma \mid \forall \bar{s}_{prefix} \in \textit{may-prefixes}(s) \cdot \exists s_i \in \bar{s}_{prefix} \cdot s_i \in [\![\phi]\!]^{\perp}\} \\
[\![E[\phi U \psi]]\!]^{\perp} &\triangleq \{s \in \Sigma \mid \forall \bar{s} \in \textit{may-paths}(s) \cdot (\forall s_j \in \bar{s} \cdot s_j \in [\![\psi]\!]^{\perp} \vee \\
&\qquad \forall s_j \in \bar{s} \cdot (s_j \in [\![\psi]\!]^{\top} \Rightarrow \exists s_i \in \bar{s} \cdot (i < j \wedge s_i \in [\![\phi]\!]^{\perp})))\}
\end{aligned}
$$

Let $c \in \{\top, \perp\}$. The remaining operators are defined as follows:

$$
\begin{aligned}
[\![A[\phi U \psi]]\!]^{c} &\triangleq [\![\neg E[\neg\psi U \neg\phi \wedge \neg\psi] \wedge \neg EG\neg\psi]\!]^{c} \\
[\![AX\phi]\!]^{c} &\triangleq [\![\neg EX\neg\phi]\!]^{c} \\
[\![AF\phi]\!]^{c} &\triangleq [\![A[\top U \phi]]\!]^{c} \\
[\![EF\phi]\!]^{c} &\triangleq [\![E[\top U \phi]]\!]^{c} \\
[\![AG\phi]\!]^{c} &\triangleq [\![\neg EF\neg\phi]\!]^{c}
\end{aligned}
$$

ACTL, i.e. universal CTL, and ECTL, i.e. existential CTL, are subsets of CTL in the former of which the only allowed path quantifier is A and in the latter of which the only allowed path quantifier is E. Furthermore, negation exists in neither ACTL nor ECTL.

In Chapter 4, we discuss the motivation for dropping the next-time operator from temporal logics. The *nextless* fragment of CTL, denoted CTL_{-X}, is the set of all CTL formulas that do not contain EX or AX operators.

2.5 Behavioral preorders and equivalences

Let $M = (\Sigma,\ \rightarrow,\ I,\ Ap)$ be a Kripke Structure, and let $s, t \in \Sigma$. The preorder relation $\mathcal{R} \subseteq \Sigma \times \Sigma$ is a *simulation* relation [Mil89] such that $s\mathcal{R}t$ if and only if:

1. $\forall p \in Ap \cdot I(s,p) = I(t,p)$

2. $\forall s' \in \Sigma \cdot (s \rightarrow s' \Rightarrow \exists t' \in \Sigma \cdot (t \rightarrow t' \wedge s' \mathcal{R} t'))$

Let $M = (\Sigma,\ \rightarrow,\ I,\ Ap)$ be a Kripke Structure, and let $s, t \in \Sigma$. The equivalence relation $\equiv \subseteq \Sigma \times \Sigma$ is a *bisimulation* relation [Mil89] such that $s \equiv t$ if and only if:

1. $\forall p \in Ap \cdot I(s,p) = I(t,p)$

2. $\forall s' \in \Sigma \cdot (s \rightarrow s' \Rightarrow \exists t' \in \Sigma \cdot (t \rightarrow t' \wedge s' \equiv t'))$

3. $\forall t' \in \Sigma \cdot (t \rightarrow t' \Rightarrow \exists s' \in \Sigma \cdot (s \rightarrow s' \wedge s' \equiv t'))$

States s and t are said to be *bisimilar* if and only if they satisfy the same CTL formulas [HM85, BCG88]. A state s is *similar* to a state t if and only if any ACTL formula that holds in t also holds in s [HM85, BCG88]. In other words, CTL is a *logical characterization* of bisimulation, and ACTL is a *logical characterization* of simulation. It should be pointed out that bisimulation (resp. simulation) can be logically characterized by CTL (resp. ACTL) *only* when the underlying transition system is *finitely branching*.

Chapter 3

Refinement on Partial Specifications

Development by sound refinement steps is an important method for showing the correctness of an implementation of a system with respect to its specification. Various notions of behavioral refinement have been proposed. In branching-time approaches, refinement relations fall into two categories: equivalence relations (e.g. bisimulation) and preorder relations (e.g. simulation). This categorization reflects an interesting trade-off between the expressive power of the logic that characterizes a refinement relation and the flexibility of the refinement relation in providing a reasonably good state space reduction.

The analysis of software systems can be performed in a sufficiently detailed way if conducted within a bisimulation-equivalence class. This is because bisimulation equivalence can be characterized by CTL* which is a very expressive logic capable of describing both liveness and safety properties. However, bisimulation leaves no room for state space reduction because the behaviors of the abstract specification and the concrete implementation of the system must be bisimilar. Simulation-based analysis, in contrast, is restricted to a smaller fragment of logical properties because it cannot preserve both safety and liveness properties. But simulation has the flexibility required for state space reduction. Therefore, at one extreme, we have bisimulation with fairly limited state space reducibility but powerful property preservation; and at the other extreme, we have

simulation with the opposite features.

In this chapter, we outline a refinement relation that is somewhere in between these two extremes. This refinement relation, which has been proposed in [HJS01], is defined in such a way that it preserves both safety and liveness properties and also achieves a good state space reduction. The key to making this possible is to represent abstract systems using partial models rather than conventional classical models. We use a simplified version of Kripke Modal Transition Systems [HJS01] as the underlying basis for our partial modeling formalism. Other examples of partial models are χKripke Structures [CDEG03], Modal Transition Systems (MTSs) [LT88, Lar89], and Partial Kripke Structures (PKSs) [BG99]. Since the above-mentioned partial modeling formalisms can be translated into one another [GJ03], the refinement relation presented in this chapter can be adapted to any of these formalisms in more-or-less the same way.

3.1 Simulation on Kripke Structures

In this section, simulation relation and its logical characterization are reviewed. The results of this section are used in the proof of the logical characterization theorem of the refinement relation described in the next section.

In this chapter, all preorder relations, i.e. simulations and refinements, are defined between two states of a single model M, but intuitively, we expect a refinement relation to be defined between the states of two different models. This does not pose a problem because the same definitions can be used for relating the states of a model M_1 to those of another model M_2 through constructing the disjoint union $M_1 \oplus M_2$, and formulating the refinement relation within $M_1 \oplus M_2$ in such a way that the refinement relation is a subset of $\Sigma_1 \times \Sigma_2$.

Definition 3.1 (Simulation on Kripke Structures) *[HJS01] Let $M = (\Sigma, \rightarrow, I, Ap)$ be a Kripke Structure, and let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a binary relation such that $\forall s, t \in \Sigma \cdot\ s\mathcal{R}t$*

if and only if:

1. $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t,p) = \top$

2. $\forall s' \in \Sigma \cdot (s \rightarrow s' \Rightarrow \exists t' \in \Sigma \cdot (t \rightarrow t' \ \wedge \ s' \mathcal{R} t'))$

Then, $\mathcal{R}$ is called a simulation relation. The largest simulation relation is denoted $\mathcal{R}_{sim}$.

In Definition 3.1, a state s is simulated by a state t if every atomic proposition that is true in s is also true in t, but in the conventional definition of simulation (given in the Preliminaries chapter), the set of atomic propositions that are true in s must be equal to the set of atomic propositions that are true in t.

In general, two states s and t are different according to some relation $\mathcal{R}$ if there is some logical property ϕ in a logic L such that ϕ distinguishes s from t and L is the logic that characterizes $\mathcal{R}$. As we will see later, the logic characterizing the simulation relation $\mathcal{R}_{sim}$ does not have negation. Therefore, a state s is similar to a state t when the set of positive atomic propositions of s is a subset of the set of positive atomic propositions of t.

The intuition behind simulation is that a transition from a state s can be mimicked by another transition from a state t when $s \mathcal{R}_{sim} t$. The relation $\mathcal{R}_{sim}$ is typically used to refine a specification *Spec* into an implementation *Imp*. More precisely, an implementation *Imp* refines a specification *Spec* if *Spec* simulates *Imp*. Thus, every behavior exhibited by *Imp* is permitted by *Spec*, or alternatively, the set of behaviors of *Imp* is a subset of the set of behaviors of *Spec*.

The simulation relation $\mathcal{R}_{sim}$ can logically be characterized by a propositional modal logic, denoted $L_\diamond$ [Pnu86]. The logic $L_\diamond$, which is only capable of expressing existential properties, has the following abstract syntax:

$$\phi ::= \top \mid p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid EX\phi$$

where $p \in Ap$.

For any Kripke Structure M and any property $\phi \in L_\diamond$, we use $[\![\phi]\!]$ to denote the set of states that satisfy ϕ. In other words, for a state s and a property ϕ, $s \vDash \phi \Leftrightarrow s \in [\![\phi]\!]$ and $s \nvDash \phi \Leftrightarrow s \notin [\![\phi]\!]$.

Theorem 3.2 (Logical characterization of $\mathcal{R}_{sim}$) *[HJS01] $s\mathcal{R}_{sim}t$ if and only if* $\forall \phi \in L_\diamond \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$.

Proof

$\Rightarrow$ Let ϕ be a $L_\diamond$ formula. We prove the lemma by induction on the structure of ϕ.

- $\phi = p$

$$
\begin{aligned}
& s \in [\![p]\!] \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& I(s,p) = \top \\
\Rightarrow\ & \text{(by the assumption that } s\mathcal{R}_{sim}t\text{)} \\
& I(t,p) = \top \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& t \in [\![p]\!]
\end{aligned}
$$

- $\phi = \varphi \wedge \psi$

$$
\begin{aligned}
& s \in [\![\varphi \wedge \psi]\!] \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& s \in ([\![\varphi]\!] \cap [\![\psi]\!]) \\
\Rightarrow\ & \text{(by the definition of } \cap\text{)} \\
& s \in [\![\varphi]\!] \wedge s \in [\![\psi]\!] \\
\Rightarrow\ & \text{(by the inductive hypothesis)} \\
& t \in [\![\varphi]\!] \wedge t \in [\![\psi]\!] \\
\Rightarrow\ & \text{(by the definition of } \cap\text{)} \\
& t \in ([\![\varphi]\!] \cap [\![\psi]\!]) \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& t \in [\![\varphi \wedge \psi]\!]
\end{aligned}
$$

-$\phi = EX\varphi$

$s \in [\![EX\varphi]\!]$

$\Rightarrow$ (by Definition 2.7)

$s \in \{s \in \Sigma \mid \exists s' \in \Sigma \cdot (s \rightarrow s' \ \wedge \ s' \in [\![\varphi]\!])\}$

$\Rightarrow$ (by the assumption that $s\mathcal{R}_{sim}t$, and by the inductive hypothesis)

$t \in \{t \in \Sigma \mid \exists t' \in \Sigma \cdot (t \rightarrow t' \ \wedge \ t' \in [\![\varphi]\!])\}$

$\Rightarrow$ (by Definition 2.7)

$t \in [\![EX\varphi]\!]$

$\Leftarrow$ Assume that $\forall \phi \in L_\diamond \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$. We show $s\mathcal{R}_{sim}t$. Since $\mathcal{R}_{sim}$ is the largest simulation relation, we need to prove that the pair (s,t) is an element of *some* simulation relation $\mathcal{R} \subseteq \mathcal{R}_{sim}$. We define a binary relation $\mathcal{R}'$ as follows: $s\mathcal{R}'t$ if and only if $\forall \phi \in L_\diamond \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$. We prove that the conditions of Definition 3.1 are satisfied by $\mathcal{R}'$.

Condition 1: Follows from the fact that $Ap \subset L_\diamond$.

Condition 2: Suppose $s \rightarrow s'$. We show that there exists t' such that $t \rightarrow t'$ and $s'\mathcal{R}'t'$. Suppose this is not the case. Consider the set C of successors of t. Because the model is finitely branching, C is finite. We distinguish the following two cases:

1. C is empty. Let φ be a $L_\diamond$ formula such that $s' \in [\![\varphi]\!]$. Thus, $s \in [\![EX\varphi]\!]$. Since $s\mathcal{R}'t$, we conclude that $t \in [\![EX\varphi]\!]$. But t does not have any outgoing transitions; therefore, $t \notin [\![EX\varphi]\!]$ - Contradiction.
2. C is not empty, say $C = \{t'_1, t'_2, \cdots, t'_n\}$ with $n \geq 1$. For every $t'_i \in C$, we have $(s', t'_i) \notin \mathcal{R}'$. Thus, by the definition of $\mathcal{R}'$, there exist formulas $\varphi_1, \cdots, \varphi_n \in L_\diamond$ such that $s' \in [\![\varphi_i]\!] \wedge t'_i \notin [\![\varphi_i]\!]$ for every $1 \leq i \leq n$. But, this implies $s \in [\![EX(\varphi_1 \wedge \cdots \wedge \varphi_n)]\!]$, while $t \notin [\![EX(\varphi_1 \wedge \cdots \wedge \varphi_n)]\!]$; therefore, $(s,t) \notin \mathcal{R}'$ - Contradiction.

■

Dually, we can prove that:

Theorem 3.3 *$s\mathcal{R}_{sim}t$ if and only if $\forall\varphi \in L_{\Box} \cdot t \in [\![\varphi]\!] \Rightarrow s \in [\![\varphi]\!]$.*

$L_{\Box}$ [Pnu86] is the dual of $L_{\Diamond}$ and can only express universal properties. The abstract syntax of $L_{\Box}$ is as follows:

$$\phi ::= \top \mid p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid AX\phi$$

where $p \in Ap$.

Thus, when a state s is related to a state t by the relation $\mathcal{R}_{sim}$, the universal properties that have been proven to be correct in t are preserved in s, and the existential properties that evaluate to true in s are true when evaluated in t. More precisely, the relation $\mathcal{R}_{sim}$ preserves existential properties in one direction and universal properties in the other. A major drawback in any kind of analysis based on $\mathcal{R}_{sim}$ is that the scope of verification is limited to universal properties (or existential properties in the reverse direction). In order to capture both universal and existential properties, a refinement relation over partial models has been proposed in [HJS01]. This refinement relation is the subject of the next section.

3.2 Refinement on Kripke Modal Transition Systems

In this section, we review the refinement relation over Kripke Modal Transition Systems (or partial models, in general) proposed in [HJS01]. We assume that every KMTS is total. This means that each state has at least one outgoing *must* or *may*-transition. However, M_{must} is not necessarily total.

In a KMTS, M_{must} represents those behaviors of the system that are guaranteed to be present in the final implementation, while M_{may} represents those behaviors that are not yet confirmed and may not make it to the final implementation.

In order to reason about KMTSs, we have to define the semantics of logical properties over them. In classical models, we can always prove either ϕ or $\neg\phi$. Hence, in classical reasoning, $s \notin [\![\phi]\!]$ if and only if $s \in [\![\neg\phi]\!]$. In partial models, in contrast to classical models, if $s \in [\![\neg\phi]\!]$ then $s \notin [\![\phi]\!]$, but not vice versa. Particularly, we may fail to prove a property ϕ because we do not have enough information about it, but this does not imply that we can prove $\neg\phi$. More precisely, when there is no proof for property ϕ, it does not mean that we have proved $\neg\phi$.

In partial models, we use $[\![\phi]\!]^{\top}$ to denote the set of states that satisfy a property ϕ, and $[\![\phi]\!]^{\bot}$ to denote the set of states that refute ϕ. To prove ϕ in a state s, we have to show that $s \in [\![\phi]\!]^{\top}$, and to refute ϕ in s, we have to show that $s \in [\![\phi]\!]^{\bot}$. Clearly, the value of ϕ is unknown in s if $s \notin [\![\phi]\!]^{\top}$ and $s \notin [\![\phi]\!]^{\bot}$.

We augment the logic $L_{\diamond}$ with negation and introduce another logic denoted PML [Var97]. PML can be extended with a fixpoint operator to form a modal fixpoint logic, referred as μ-calculus [Koz83]. The abstract syntax of PML is as follows:

$$\phi ::= \top \mid p \mid \phi_1 \wedge \phi_2 \mid EX\phi \mid \neg\phi$$

where $p \in Ap$.

The operators $\bot$, $\vee$, and AX are derived as $\neg\top$, $\neg\phi_1 \wedge \neg\phi_2$, and $\neg EX\neg\phi$, respectively.

Definition 3.4 (Refinement) *[HJS01] Let $M = (\Sigma, \rightarrow^{must}, \rightarrow^{may}, I^{must}, I^{may}, Ap)$ be a KMTS, and let $\preceq \subseteq \Sigma \times \Sigma$ be a binary relation such that $\forall s, t \in \Sigma \cdot t \preceq s$ if and only if:*

1. $\forall p \in Ap \cdot I^{must}(s,p) = \top \Rightarrow I^{must}(t,p) = \top$
2. $\forall p \in Ap \cdot I^{may}(t,p) = \top \Rightarrow I^{may}(s,p) = \top$
3. $\forall s' \in \Sigma \cdot (s \rightarrow^{must} s' \Rightarrow \exists t' \in \Sigma \cdot (t \rightarrow^{must} t' \ \wedge \ t' \preceq s'))$
4. $\forall t' \in \Sigma \cdot (t \rightarrow^{may} t' \Rightarrow \exists s' \in \Sigma \cdot (s \rightarrow^{may} s' \ \wedge \ t' \preceq s'))$

Then, $\preceq$ is called a refinement relation. The largest refinement relation is denoted $\preceq_{ref}$.

Notice that if $t \preceq_{ref} s$, then $s\mathcal{R}_{sim}t$ in M_{must} and $t\mathcal{R}_{sim}s$ in M_{may}, but not vice versa.

The refinement relation $\preceq_{ref}$ derives an implementation *Imp* from a partial specification *Spec* in such a way that the set of behaviors guaranteed in *Spec* is a subset of the set of behaviors of *Imp*, and the set of behaviors admissible in *Spec* includes the set of behaviors of *Imp*. In other words, $\preceq_{ref}$ is capable of relating an implementation to a specification in such a way that all behaviors guaranteed by the specification are exhibited by the implementation and all behaviors of the implementation are admitted by the specification.

It has been shown [HJS01] that $\preceq_{ref}$ can be logically characterized by a modal μ-calculus logic [Koz83] which is a very expressive logic and important logics like CTL* can be embedded into it [Dam94]. As mentioned earlier, the logic PML can itself be extended with a fixpoint operator to give a modal μ-calculus logic. Thus, we prove the following logical characterization theorem for PML.

Theorem 3.5 (Logical characterization of $\preceq_{ref}$) *[HJS01] $t \preceq_{ref} s$ if and only if*

$$\forall \phi \in PML \cdot s \in [\![\phi]\!]^{\top} \Rightarrow t \in [\![\phi]\!]^{\top} \ \wedge \ s \in [\![\phi]\!]^{\perp} \Rightarrow t \in [\![\phi]\!]^{\perp}$$

Proof

$\Rightarrow$ Let $t \preceq_{ref} s$ and let $\phi \in PML$.

Suppose $s \in [\![\phi]\!]^{\top}$. Then, $s \in [\![\phi]\!]$ in M_{must}. Since $s\mathcal{R}_{sim}t$ in M_{must}, by Theorem 3.2, $t \in [\![\phi]\!]$ in M_{must} as well. Thus, $t \in [\![\phi]\!]^{\top}$.

Suppose $s \in [\![\phi]\!]^{\perp}$. Then, $s \notin [\![\phi]\!]$ in M_{may}. Since $t\mathcal{R}_{sim}s$ in M_{may}, by Theorem 3.2, $t \notin [\![\phi]\!]$ in M_{may} as well. Thus, $t \in [\![\phi]\!]^{\perp}$.

$\Leftarrow$ Assume that $\forall \phi \in PML \cdot s \in [\![\phi]\!]^{\top} \Rightarrow t \in [\![\phi]\!]^{\top}$ and $s \in [\![\phi]\!]^{\perp} \Rightarrow t \in [\![\phi]\!]^{\perp}$. We show that $t \preceq_{ref} s$. Since $\preceq_{ref}$ is the largest refinement relation, all we need to prove is that the pair (t, s) is an element of *some* refinement relation $\preceq \subseteq \preceq_{ref}$. We define a binary relation $\preceq'$ as follows: $s \preceq' t$ if and only if $\forall \phi \in PML \cdot s \in [\![\phi]\!]^{\top} \Rightarrow t \in [\![\phi]\!]^{\top}$

and $s \in [\![\phi]\!]^{\perp} \Rightarrow t \in [\![\phi]\!]^{\perp}$. We prove that the conditions of Definition 3.4 are satisfied by $\preceq'$.

Conditions 1 and 2: Follow from the fact that $Ap \subset PML$.

Condition 3: Suppose $s \rightarrow^{must} s'$. We show that there exists t' such that $t \rightarrow^{must} t'$ and $t' \preceq' s'$. Suppose this is not the case. Consider the set C of all states that can be reached by a single *must*-transition from t. Since we assumed that KMTSs are finitely branching, C is finite. We distinguish the following two cases:

1. C is empty. Let φ be a PML formula such that $s' \in [\![\varphi]\!]^{\top}$. Therefore, $s \in [\![EX\varphi]\!]^{\top}$. Since $t \preceq' s$, we conclude that $t \in [\![EX\varphi]\!]^{\top}$. But, t does not have any outgoing *must*-transition; therefore, $t \notin [\![EX\varphi]\!]^{\top}$ - Contradiction.
2. C is not empty, say $C = \{t'_1, t'_2, \cdots, t'_n\}$ with $n \geq 1$. By the assumption that we made, we have $t'_i \not\preceq' s'$ for every $t'_i \in C$. Thus, by the definition of $\preceq'$, there exist formulas $\varphi_1, \cdots \varphi_n \in PML$ such that $(s' \in [\![\varphi_i]\!]^{\top} \wedge t'_i \notin [\![\varphi_i]\!]^{\top})$ $\bigvee$ $(s' \in [\![\varphi_i]\!]^{\perp} \wedge t'_i \notin [\![\varphi_i]\!]^{\perp})$. We replace every formula φ_i with a formula $\varphi'_i \in PML$ such that if $s \in [\![\varphi_i]\!]^{\perp}$ then $\varphi'_i = \neg\varphi_i$, otherwise $\varphi'_i = \varphi_i$. But this implies $s \in [\![EX(\varphi'_1 \wedge \cdots \wedge \varphi'_n)]\!]^{\top}$, while $t \notin [\![EX(\varphi_1 \wedge \cdots \wedge \varphi_n)]\!]^{\top}$; therefore, $t \not\preceq' s$ - Contradiction.

Condition 4: This condition can be proved in the same as the previous condition.

■

Chapter 4

Stuttering Refinement on Partial Specifications

The notion of a *step* made by a system has an important place in behavioral modeling. From a system's point of view, a step is the next moment at which some internal or external state variable changes. Such a step is referred as a *system* step in [Dam96]. However, for an external observer, a system step is often non-existent because a change in the internal variables of the system is not externally observable. From an observer's point of view, a step can be distinguished only when some external variable changes in such a way that the value of at least one logical property of the system is affected. Therefore, how a step is interpreted by an external observer depends on two factors: first, the expressive power of the logic used for describing the properties of the system; and second, the observable propositions of the system. The observer's interpretation of a step is referred as a *logical* step in [Dam96].

Figure 4.1 illustrates the difference between a system step and a logical step on a simple example. Each transition of the system shown in Figure 4.1(a) represents a system step. The set of observable variables of the system shown in Figure 4.1(a) is $\{p, r\}$, and the system properties are expressed in CTL_{-X}. Under these assumptions, we

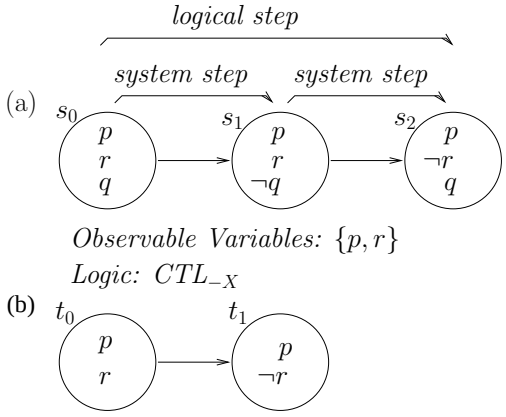

Figure 4.1: System step versus. logical step

can only distinguish one logical step. This logical step is shown in Figure 4.1(b). It is easy to check that every CTL$_{-X}$ property that evaluates to true or false in the system shown in Figure 4.1(a) has the same truth value when evaluated in the system shown in Figure 4.1(b), and vice versa. In other words, no CTL$_{-X}$ property can distinguish the systems in Figures 4.1(a) and 4.1(b). However, if we choose CTL as the underlying logic, or if we assume that the set of observable variables is $\{p, q, r\}$, then the systems in Figures 4.1(a) and 4.1(b) are not logically equivalent, anymore. As this example shows, the distinguishing power of CTL is more than that of CTL$_{-X}$.

Many temporal logics include a next operator which describes that some property holds in the next state along a path. The next state is usually the state that can be reached by taking a single system step. Therefore, the next operator may refer to a step that is not externally observable. This is usually undesirable. For example, in Figure 4.1(b), we do not consider the transition from s_0 to s_1 because it does not make sense to abstract from the variable q while still considering its changes. However, the property EXr distinguishes between s_0 and t_0 because the transition from s_0 to s_1 is missing at t_0. Therefore, the next operator may produce different results when applied to systems that are identical to an external observer.

Lamport was among the pioneers to argue [Lam83] that the next operator should not be used in specifications because it makes the logic more expressive than what it should really be. In fact, the next operator describes some unnecessary details about the behaviors usually ignored in high-level abstractions of a specification. More intuitively, the next operator may provide some information about system steps, while what we are really interested in is high-level logical steps. In [Lam83], Lamport simply drops the next operator from the set of operators of the underlying logic. This approach is closely related to the field of behavioral equivalences and is discussed in this chapter. Other solutions to this problem involve parameterizing the next operator and changing its interpretation. For example, in [PC02], a new next operator $\uparrow$ is defined in such a way that $\uparrow A$ indicates a change in the value of an observable formula A.

A similar problem is caused by *silent moves* in action-based approaches. A silent move represents a pause or a period of time during which a system is idle according to an external observer. Silent moves in action-based approaches are analogous to system (but not to a logical) steps in state-based approaches. Some of the proposed action-based behavioral relations ignore silent moves in specifications. Examples of such behavioral relations are *observational* or *weak* bisimulation relations [Mil89]. In [vG90], it has been argued that *branching bisimulation* is the coarsest possible equivalence relation that ignores silent moves and yet respects the branching structure of a system.

A state-based approach to *stuttering bisimulation* over Kripke Structures has been introduced by Browne, Clarke, and Grumberg [BCG88]. They prove that states related by a stuttering equivalence relation satisfy the same CTL^*_{-X} properties. In [NV95], branching bisimulation over Labeled Transition Systems and stuttering equivalence over Kripke Structures have been compared and a precise connection between them has been established. Consequently, it has been shown that branching bisimulation over Labeled Transition Systems can also be characterized by CTL^*_{-X} properties.

The notions of step and stuttering relation have been studied extensively in the liter-

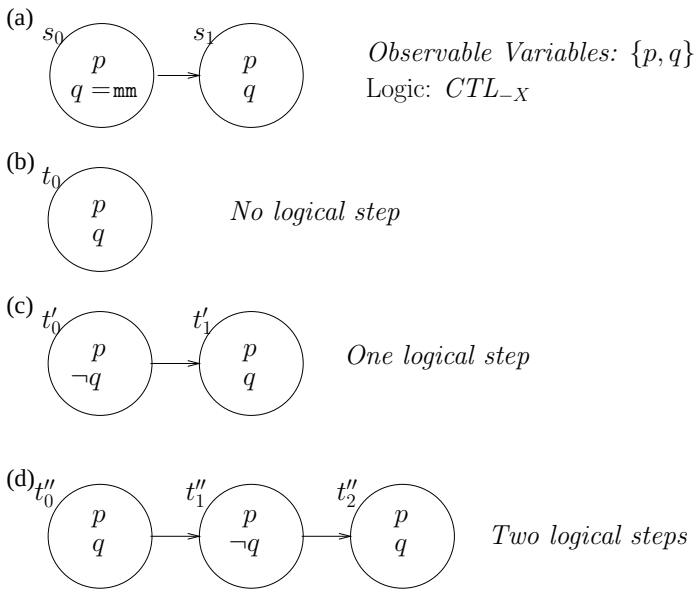

Figure 4.2: Logical steps in partial models

ature; however, to the best of our knowledge, these notions have never been elaborated in the context of partial systems. In classical systems, a logical step ensures an observable change in the value of some formula from true to false or false to true. In partial systems, formulas may evaluate to unknown as well as true or false, and a change from unknown to true or false does not necessarily create a logical step.

In Figure 4.2, a partial system (Figure 4.2(a)) and three different possible refinements (Figures 4.2(b), 4.2(c), and 4.2(d)) of it are shown. According to the refinement in Figure 4.2(b), s_0 and s_1 in Figure 4.2(a) are just two repetitions of t_0 in Figure 4.2(b). Therefore, no logical step exists. However, in the refinement shown in Figure 4.2(c), one logical step between t'_0 and t'_1 can be identified. Figure 4.2(d) shows yet another possible refinement, but this time, two logical steps can be identified, one from t''_0 to t''_1 and the other from t''_1 to t''_2.

This example illustrates that different possible refinements induce different logical steps in a partial system. In this chapter, we define a preorder relation that relates a partial system to all its possible refinements with different sets of logical steps. We refer

to such a preorder relation as a *stuttering refinement relation.*

4.1 Stuttering simulation on Kripke Structures

In this section, we define a notion of stuttering simulation on Kripke Structures. Like in the previous chapter, we define refinement relations between the states of a single model M. For showing a refinement relation between the states of two different models M_1 and M_2, we can construct the disjoint union, $M_1 \oplus M_2$, and formulate the refinement relation within $M_1 \oplus M_2$ in such a way that the refinement relation is a subset of $\Sigma_1 \times \Sigma_2$.

Definition 4.1 (Divergence-blind stuttering simulation) *Let $M = (\Sigma, \rightarrow, I, Ap)$ be a Kripke Structure, and let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a binary relation such that $\forall s, t \in \Sigma \cdot s\mathcal{R}t$ if and only if:*

1. $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t,p) = \top$
2. *For every $s \rightarrow s_1$, there exists $t_0 = t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_{m-1} \rightarrow t_m$ such that $m \geq 0, s\mathcal{R}t_i$ for every $i < m$, and $s_1\mathcal{R}t_m$*

Then, $\mathcal{R}$ is a divergence-blind stuttering simulation. The largest divergence-blind stuttering simulation is denoted $\mathcal{R}_{dbs}$.

The second condition of Definition 4.1 describes that if state s can make a transition to a state s_1, then state t can make a transition sequence of length zero or more to some state t_m such that s_1 is related to t_m and s is related to every state that lies on the prefix from t to t_m (excluding t_m). In the remainder of this chapter, whenever we write $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_{m-1} \rightarrow t_m$, we mean a transition sequence of length zero or more, i.e. $m \geq 0$.

Definition 4.1 is a weaker version of the definition of divergence-blind equivalence relation proposed in [NV95]. In our definition, a state s is related to a state t when the set of atomic propositions that are true in s is a *subset* of the set of atomic propositions

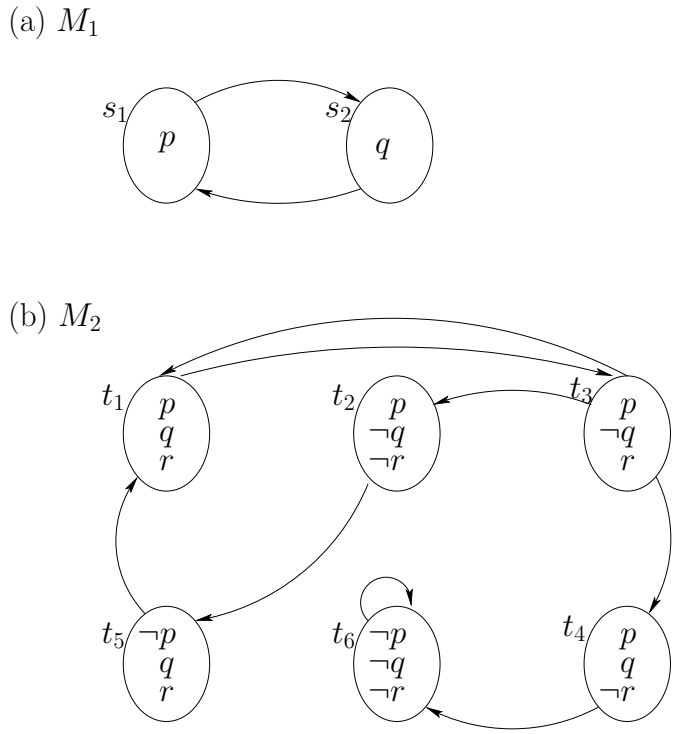

Figure 4.3: An example of stuttering simulation

that are true in t; while in [NV95], s is related to t when the set of atomic propositions that are true in s is *equal* to the set of atomic propositions that are true in t. Thus, $\mathcal{R}_{dbs}$ is a preorder relation, whereas the relation proposed in [NV95] is an equivalence relation.

Example 4.2 Figure 4.3 illustrates an example of stuttering simulation relation between two Kripke Structures M_1 and M_2. The set of atomic propositions in both M_1 and M_2 is $\{p, q, r\}$. The value of each proposition not shown in the states of M_1 is either false or unknown. In this figure, there are some pairs $(s, t) \in \Sigma_1 \times \Sigma_2$ for which $s\mathcal{R}_{dbs}t$. Here, we only elaborate two cases:

1. $s_1\mathcal{R}_{dbs}t_3$: because p is true in state t_3, and there exists a transition sequence $t_3 \rightarrow t_2 \rightarrow t_5$ corresponding to the transition $s_1 \rightarrow s_2$ such that $s_1\mathcal{R}_{dbs}t_2$ and $s_2\mathcal{R}_{dbs}t_5$.

2. $s_1\mathcal{R}_{dbs}t_4$: because p is true in t_4, and there exists a transition sequence t_4 of length

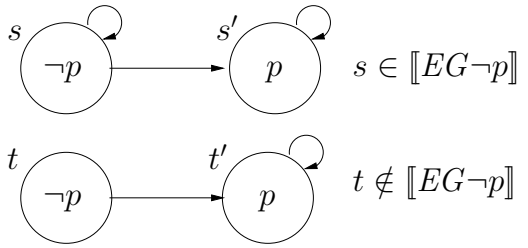

Figure 4.4: An example of dbs-simulation

zero corresponding to the transition $s_1 \rightarrow s_2$ such that $s_1 \mathcal{R}_{dbs} t_4$ and $s_2 \mathcal{R}_{dbs} t_4$.

If states s and t cannot be distinguished by any CTL$_{-X}$ formula, then $s \mathcal{R}_{dbs} t$. The converse, however, is not true. For example, in Figure 4.4, $s \mathcal{R}_{dbs} t$ because $\mathcal{R}_{dbs}$ maps the infinite path $s \rightarrow s \rightarrow s \rightarrow \cdots$ to the single state t (Definition 4.1, Condition 2). Thus, $s \in [\![EG\neg p]\!]$, whereas $t \notin [\![EG\neg p]\!]$. In Figure 4.4, state s is a divergent state because it occurs on an infinite path such that all the states on that path are related to a single state t. As this example shows, $\mathcal{R}_{dbs}$ cannot distinguish the divergent state s from the non-divergent state t.

Different approaches to characterizing divergence-blind and divergence-sensitive stuttering relations have been investigated in the literature. In [NV95], the interpretation of path formulas is changed in such a way that the existential (E) and the universal (A) operators quantify over finite prefixes as well as infinite paths. Under the assumption made there, $EG\neg p$ holds in state t of Figure 4.4 because t itself is a finite prefix. Having applied this adjustment, it is possible to show that CTL$_{-X}$ properties logically characterize $\mathcal{R}_{dbs}$. Furthermore, in [NV95], a definition has been proposed for *divergence-sensitive stuttering equivalence*. This definition is, in fact, a divergence-blind stuttering equivalence that is defined over Kripke Structures extended with a *live-locked* state, i.e. a state with only one outgoing self-loop transition. A Kripke Structure can be extended with a live-locked state by adding a transition from every divergent state to the live-locked state. In [NV95], it is proved that a divergence-blind stuttering equivalence relation de-

fined over a Kripke Structure M becomes a divergence-sensitive stuttering equivalence over the Kripke Structure M that is extended with a live-locked state.

In [NV95], the problem to tackle is finding a stuttering equivalence relation. Thus, divergent states are known before-hand. There, divergent states are those located on a cycle of states all of which have the same set of atomic propositions. In our case, however, the divergent states are not known before-hand because when a single state t is related to a path $\bar{s}$ with respect to $\mathcal{R}_{dbs}$, we can only conclude that the set of positive atomic propositions of t includes the set of positive atomic propositions of every state on $\bar{s}$. Therefore, in general, no assumption can be made about the set of atomic propositions of the states on $\bar{s}$. More precisely, we cannot conclude that the states on $\bar{s}$ have the same atomic propositions. This makes it impossible to use the solution proposed in [NV95] for converting $\mathcal{R}_{dbs}$ to a divergence-sensitive stuttering relation.

In [Nam97], a formulation of stuttering bisimulation, called *well-founded bisimulation*, has been proposed. The definition of well-founded bisimulation is similar to that of bisimulation in the sense that in each step, we only need to match single transitions. This is in contrast to the definition of stuttering bisimulation where finite sequences of states are matched in each step. Well-founded bisimulation is a divergence-sensitive stuttering bisimulation because it ensures that the length of every sequence of stuttering states is finite. Therefore, a finite sequence of stuttering states is never mapped to an infinite one. The definition of well-founded bisimulation is suitable for establishing stuttering bisimulation relations; however, it is difficult to use this definition to prove the logical results that characterize stuttering bisimulation.

In [Dam96], Dams has also worked on this problem. His definition of divergence-sensitive stuttering equivalence is an adaptation of the definition of dbs-equivalence. There, he adds a new condition (Definition 4.13, Condition 3) to the definition of dbs-equivalence. In his definition, a state s is equal to a state t with respect to a stuttering-sensitive equivalence relation when the following condition holds: "s is located on an

infinite path $\bar{s}$ such that all the states on $\bar{s}$ are equal if and only if t is located on an infinite path $\bar{t}$ such that all the states on $\bar{t}$ are equal". Here, we also add a new condition to the definition of $\mathcal{R}_{dbs}$ (Definition 4.1) to define the divergence-sensitive stuttering simulation relation.

Definition 4.3 (Divergence-sensitive stuttering simulation) *Let $M = (\Sigma, \rightarrow, I, Ap)$ be a Kripke Structure, and let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a binary relation such that $\forall s, t \in \Sigma \cdot s\mathcal{R}t$ if and only if:*

1. $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t,p) = \top$
2. *For every $s \rightarrow s_1$, there exists $t_0 = t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_{m-1} \rightarrow t_m$ such that $s\mathcal{R}t_i$ for every $i < m$ and $s_1\mathcal{R}t_m$*
3. $\exists \bar{s} \in paths(s) \cdot \forall s_i \in \bar{s} \cdot s_i\mathcal{R}t \Rightarrow \exists t' \cdot (t \rightarrow t' \wedge \exists s' \in \bar{s} \cdot s'\mathcal{R}t')$

Then, $\mathcal{R}$ is a divergence-sensitive stuttering simulation. The largest divergence-sensitive stuttering simulation is denoted $\mathcal{R}_{stut}$.

The third condition in the above definition states that if there exists a path $\bar{s}$ emanating from state s such that all the states on $\bar{s}$ are related to state t, then t has to have some successor t' such that some state s' on $\bar{s}$ is related to t'.

Example 4.4 In Figure 4.3, $s_1\mathcal{R}_{dbs}t_4$ and $s_2\mathcal{R}_{dbs}t_4$; hence, there exists an infinite path $\bar{s}$ such that all the states on $\bar{s}$ are related to t_4. But t_4's successor, i.e. t_6, is not related to any state on $\bar{s}$, i.e. s_1 and s_2. Therefore, $(s_1, t_4) \notin \mathcal{R}_{stut}$ and $(s_2, t_4) \notin \mathcal{R}_{stut}$.

Example 4.5 In Figure 4.3, $s_1\mathcal{R}_{dbs}t_1$ and $s_2\mathcal{R}_{dbs}t_1$; hence, there exists an infinite path $\bar{s}$ such that all the states on $\bar{s}$ are related to t_1. However, $t_1 \rightarrow t_3$ and $s_1\mathcal{R}_{stut}t_3$; therefore, $s_1\mathcal{R}_{stut}t_1$ and $s_2\mathcal{R}_{stut}t_1$.

The following definition, which has been adapted from [BCG88], lifts the notion of divergence-sensitive stuttering simulation from states to paths.

Definition 4.6 *A path $\bar{s} \in paths(s)$ (resp. a prefix $\bar{s}_{prefix} \in prefixes(s)$) is related to a path $\bar{t} \in paths(t)$ (resp. a prefix $\bar{t}_{prefix} \in prefixes(t)$) with respect to a preorder relation $\mathcal{R}$ if and only if there exists a partitioning $B_0, B_1, B_2, \cdots$ of $\bar{s}$ (resp. $\bar{s}_{prefix}$), and a partitioning $C_0, C_1, C_2, \cdots$ of $\bar{t}$ (resp. $\bar{t}_{prefix}$) such that*

$$\forall j \in \mathbb{N} \cdot (B_j \neq \emptyset \wedge C_j \neq \emptyset \wedge \forall s' \in B_j \cdot \forall t' \in C_j \cdot s' \mathcal{R} t')$$

The following lemma extends $\mathcal{R}_{stut}$ to paths. We prove this lemma with the help of König Lemma, using an argument similar to the one given in [BCG88].

Lemma 4.7 *If $s \mathcal{R}_{stut} t$, then $\forall \bar{s} \in paths(s) \cdot \exists \bar{t} \in paths(t) \cdot \ \bar{s} \mathcal{R}_{stut} \bar{t}$.*

Proof Let M be a Kripke Structure, and let $\bar{s}$ be an infinite (M, s)-path. From Definition 4.3, it follows that for every prefix $\bar{s}_{prefix}$ of $\bar{s}$, there exists a prefix $\bar{t}_{prefix}$ such that $\bar{s}_{prefix} \mathcal{R}_{stut} \bar{t}_{prefix}$:

$$s \mathcal{R}_{stut} t \Rightarrow \forall \bar{s}_{prefix} \text{of } \bar{s} \cdot \exists \bar{t}_{prefix} \in prefixes(t) \cdot \bar{s}_{prefix} \ \mathcal{R}_{stut} \ \bar{t}_{prefix} \tag{4.1}$$

Using an argument based on König Lemma, we now prove that there exists a path $\bar{t}$ corresponding to $\bar{s}$ such that $\bar{s} \mathcal{R}_{stut} \bar{t}$. We then show that $\bar{t}$ has to be infinite.

We construct a tree T rooted at t in such a way that $t_0 t_1 t_2 \cdots t_n$ is a path in T if and only if

- $t_0 = t$, and
- there exists a prefix $\bar{t}_{prefix} \in prefixes(t)$ and a prefix $\bar{s}_{prefix}$ of $\bar{s}$ such that the following three conditions are satisfied.

  1. $\bar{t}_{prefix}$ is partitioned by a sequence

     $$C_0 = \{t_0 u_1 \cdots u_p\}, C_1 = \{t_1 v_1 \cdots v_q\}, \cdots, C_n = \{t_n z_1 \cdots z_w\}$$

     of blocks.

2. $\bar{s}_{prefix}$ is partitioned by a sequence $B_0, B_1, \cdots, B_n$ of blocks.

3. $\forall 0 \leq i \leq n \cdot B_i \neq \emptyset \wedge C_i \neq \emptyset \wedge \forall s' \in B_i \cdot \forall t' \in C_i \cdot s' \mathcal{R}_{stut} t'$.

Intuitively, every path in T corresponds to a partitioning of some $\bar{t}_{prefix}$ such that $\bar{s}_{prefix}$ $\mathcal{R}_{stut}$ $\bar{t}_{prefix}$ and $\bar{s}_{prefix}$ is a prefix of $\bar{s}$. For every path in T, the partitioning of its corresponding t-prefix, i.e. $\{B_0, B_1, \cdots, B_n\}$, is denoted $\mathcal{B}$ and the partitioning of its corresponding s-prefix, i.e. $\{C_1, \cdots, C_n\}$, is denoted $\mathcal{C}$.

Suppose that the number of nodes in T is infinite. Since we assumed that the number of states in the underlying model M is finite, T is finitely branching. Therefore, by König Lemma, there exists an infinite path through T. Thus, for this path the number of blocks in both $\mathcal{B}$ and $\mathcal{C}$ is infinite. Hence, there is an infinite path $\bar{t}$ corresponding to $\bar{s}$.

Suppose that the number of nodes in T is finite, say n'. Thus, in T, there is no path of length $n' + 1$. Therefore, for every path in T, the number of blocks in both $\mathcal{B}$ and $\mathcal{C}$ is finite. For every path in T, we extend the last block of $\mathcal{B}$, i.e. B_n, to cover all the remaining states of $\bar{s}$. This makes $\mathcal{B}$ a partitioning of the infinite path $\bar{s}$ for every path in T. If we prove that there exists some path in T such that $\forall s' \in B_n \cdot \forall t' \in C_n \cdot s' \mathcal{R}_{stut} t'$, then we are done. Suppose this is not the case. Thus, for every path in T, there exists a state $s' \in B_n$ and a state $t' \in C_n$ such that $(s', t') \notin \mathcal{R}_{stut}$. This implies that there exists a prefix of $\bar{s}$ without any corresponding t-prefix, this contradicts the premise 4.1.

The above argument implies that for every infinite path $\bar{s}$, there exists a corresponding *finite* or *infinite* path $\bar{t}$. Now, we show that $\bar{t}$ has to be infinite:

First, if the partitioning of $\bar{s}$ has an infinite number of blocks, then so does the partitioning of $\bar{t}$, meaning that $\bar{t}$ is infinite.

Otherwise, suppose that the partitioning of $\bar{s}$ has a finite number of blocks $\mathcal{B} = \{B_0, B_1, \cdots, B_n\}$. Thus, the partitioning of $\bar{t}$ has a finite number of blocks $\mathcal{C} = \{C_0, C_1, \cdots, C_n\}$ as well. If $\bar{t}$ is finite, all C_i's are finite. But, B_n is infinite because $\bar{s}$ is. Without loss of generality, assume that C_n has a single state t_l. We have just shown that $\forall s_i \in B_n \cdot s_i \mathcal{R}_{stut} t_l$. Hence, by Definition 4.3 (Condition 3), there exists

some state t_{l+1} such that $t_l \rightarrow t_{l+1}$ and some state $s' \in B_n$ such that $s'\mathcal{R}_{stut}t_{l+1}$. This contradicts the finiteness of $\bar{t}$; thus, $\bar{t}$ must be infinite. ∎

Lemma 4.8 *If* $s\mathcal{R}_{stut}t$, *then* $\forall\phi \in ECTL_{-X} \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$.

Proof By induction on the structure of ϕ. If ϕ is a propositional formula of the form p or $\phi_1 \wedge \phi_2$, then the lemma follows from Theorem 3.2.

It suffices to prove the lemma for EG and EU because EG and EU form an adequate set for $ECTL_{-X}$.

Notice that according to the semantics of EG and EU, a witness for the property $EG\phi$ is an *infinite* path, while a witness for the property $E[\varphi U\psi]$ is a *finite* prefix.

- $\phi = EG\phi$:

$$
\begin{aligned}
& s \in [\![EG\phi]\!] \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& \exists\bar{s} \in paths(s) \cdot \forall s_i \in \bar{s} \cdot s_i \in [\![\phi]\!] \\
\Rightarrow\ & \text{(by Lemma 4.7, and } s\mathcal{R}_{stut}t\text{)} \\
& \exists\bar{t} \in paths(t) \cdot \forall t_j \in \bar{t} \cdot \exists s_i \in \bar{s} \cdot s_i\mathcal{R}_{stut}t_j \\
\Rightarrow\ & \text{(by the inductive hypothesis)} \\
& \exists\bar{t} \in paths(t) \cdot \forall t_j \in \bar{t} \cdot t_j \in [\![\phi]\!] \\
\Rightarrow\ & \text{(by Definition 2.7)} \\
& t \in [\![EG\phi]\!]
\end{aligned}
$$

- $\phi = E[\varphi U\psi]$:

  Let $\bar{s}_1$ be an s-prefix witnessing $E[\varphi U\psi]$, and let $\bar{t}_1$ be a t-prefix such that $\bar{s}_1\mathcal{R}_{stut}\bar{t}_1$, and let $B_0, B_1, \cdots B_n$, (resp. $C_0, C_1, \cdots, C_n$) be the partitioning of $\bar{s}_1$ (resp. $\bar{t}_1$).

  Since $\bar{s}_1$ witnesses $E[\varphi U\psi]$, there exists $s_i \in \bar{s}_1$ such that $s_i \in [\![\psi]\!]$ and $\forall s_j \in \bar{s}_1 \cdot j < i \Rightarrow s_j \in [\![\varphi]\!]$. Now, suppose that B_k is the first block in the partitioning of $\bar{s}_1$ such that $s_i \in B_k$. Therefore, there exists a block C_k in the partitioning of $\bar{t}_1$ such that $\forall t' \in C_k \cdot s_i\mathcal{R}_{stut}t'$. Thus, by the inductive hypothesis,

$\forall t' \in C_k \cdot t' \in [\![\psi]\!]$. Moreover, $\forall s_j \in B_{k'} \cdot k' < k \Rightarrow s_j \in [\![\varphi]\!]$. Since $\bar{s}_1 \mathcal{R}_{stut} \bar{t}_1$, we have $\forall t' \in C_{k'} \cdot k' < k \Rightarrow \exists s_j \in B_{k'} \cdot s_j \mathcal{R}_{stut} t'$. Therefore, by the inductive hypothesis, $\forall t' \in C_{k'} \cdot k' < k \Rightarrow t' \in [\![\varphi]\!]$. Hence, $\exists \bar{t}_1 \in \mathit{prefixes}(t) \cdot \exists t_{i'} \in \bar{t}_1 \cdot t_{i'} \in [\![\psi]\!], \forall j' < i' \cdot t_{j'} \in [\![\varphi]\!]$, which provides a witness for $t \in [\![E[\varphi U \psi]]\!]$.

■

Taking advantage of the same technique used in [Dam96], we prove the following lemma:

Lemma 4.9 *If* $\forall \phi \in ECTL_{-X} \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$, *then* $s \mathcal{R}_{stut} t$.

Proof Assume that $\forall \phi \in ECTL_{-X} \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$. We show that $s \mathcal{R}_{stut} t$. Since $\mathcal{R}_{stut}$ is the largest stuttering simulation relation, it is sufficient to prove that the pair (s, t) is an element of *some* stuttering simulation $\mathcal{R} \subseteq \mathcal{R}_{stut}$. We define a binary relation $\mathcal{R}'$ as follows: $s \mathcal{R}' t$ if and only if $\forall \phi \in ECTL_{-X} \cdot s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$. We now prove that $\mathcal{R}'$ is a stuttering simulation by showing that the conditions of Definition 4.3 are satisfied by $\mathcal{R}'$.

Condition 1: Follows from the fact that $Ap \subset ECTL_{-X}$.

Condition 2: Suppose $s \rightarrow s_1$. We want to show that there exists $t \rightarrow t_1 \rightarrow \cdots \rightarrow t_m$ such that $s \mathcal{R}' t_i$ for every $i < m$ and $s_1 \mathcal{R}' t_m$. Suppose this is not the case. Consider a set C:

$$\begin{aligned} C \;=\; & \{\langle t_0 = t, \cdots, t_{m-1}, t_m \rangle \mid t_0 \rightarrow \cdots \rightarrow t_{m-1} \rightarrow t_m \wedge s \mathcal{R}' t_0 \cdots s \mathcal{R}' t_m \wedge \\ & (s_1, t_0) \notin \mathcal{R}' \cdots (s_1, t_m) \notin \mathcal{R}' \wedge \forall t_i, t_j \cdot i \neq j \Rightarrow t_i \neq t_j \} \end{aligned}$$

Intuitively, a tuple $\langle t_0, \cdots, t_{m-1}, t_m \rangle$ is in C if the transition sequence $t_0 \rightarrow \cdots \rightarrow t_m$ is a t-prefix, $\bar{t}_{prefix}$, where every state on $\bar{t}_{prefix}$ is distinct and every state on $\bar{t}_{prefix}$ is related to s but not to s_1.

Since the number of states in the underlying Kripke Structure is finite, and no redundancy is permitted in the elements of C, C is finite. Notice that C is non-

empty because $\langle t \rangle \in C$. Let $C = \{\langle t_1^1, \cdots, t_{m_1-1}^1, t_{m_1}^1 \rangle, \cdots, \langle t_1^n, \cdots, t_{m_n-1}^n, t_{m_n}^n \rangle\}$ with $n \geq 1$. For every $1 \leq i \leq n$, we distinguish two cases:

1. There is no state $t_{m_i+1}^i$ such that $t_{m_i}^i \rightarrow t_{m_i+1}^i$ and $(s, t_{m_i+1}^i) \notin \mathcal{R}'$. In this case, we choose $\varphi_i' = \top$.

2. There exists some state $t_{m_i+1}^i$ such that $t_{m_i}^i \rightarrow t_{m_i+1}^i$ and $(s, t_{m_i+1}^i) \notin \mathcal{R}'$. Therefore, by the definition of $\mathcal{R}'$, we can choose $\varphi_i' \in ECTL_{-X}$ such that $s \in [\![\varphi_i']\!] \wedge t_{m_i+1}^i \notin [\![\varphi_i']\!]$; moreover, since $s\mathcal{R}'t_j^i$, we have $t_i^j \in [\![\varphi_i']\!]$ for every $1 \leq j \leq m_i$.

Therefore, there exists a formula φ', i.e. $\varphi' = \varphi_1' \wedge \cdots \wedge \varphi_n'$, such that s satisfies φ'. Furthermore, for every tuple e in C, all the states of e satisfy φ' but the successors of the last state of e do not satisfy φ'.

Since $(s_1, t_j^i) \notin \mathcal{R}'$ for any $1 \leq i \leq n$ and $1 \leq j \leq m_i + 1$, we choose $\varphi_{ij}'' \in ECTL_{-X}$ in such a way that $s_1 \in [\![\varphi_{ij}'']\!] \wedge t_j^i \notin [\![\varphi_{ij}'']\!]$ for every $1 \leq i \leq n$ and $1 \leq j \leq m_i + 1$. If some $t_{m_i+1}^i$ does not exists, we assume $\varphi_{i,m_i+1}'' = \top$.

Thus, there exists a formula φ'', i.e. $\varphi'' = \bigwedge_{1 \leq i \leq n}(\bigwedge_{1 \leq j \leq m_i+1} \varphi_{ij}'')$, such that s_1 satisfies φ'', but for every tuple e in C, none of the states of e or the successors of the last state of e satisfy φ''.

Hence, $s \in [\![E[\varphi' U \varphi'']]\!]$ because $s \in [\![\varphi']\!]$, and $s_1 \in [\![\varphi'']\!]$. But, every t-path contains some $\langle t_1^i, \cdots t_{m_i}^i \rangle$, and for every $t_j^i \in \langle t_1^i, \cdots t_{m_i}^i \rangle$, we have $t_j^i \in [\![\varphi']\!]$, $t_j^i \notin [\![\varphi'']\!]$, and for every successor $t_{m_i+1}^i$ of the last state of $\langle t_1^i, \cdots t_{m_i}^i \rangle$, we have $t_{m_i+1}^i \notin [\![\varphi']\!] \wedge t_{m_i+1}^i \notin [\![\varphi'']\!]$. Thus, we conclude that $t \notin [\![E[\varphi' U \varphi'']]\!]$. This implies that there is some formula ϕ such that $s \in [\![\phi]\!] \wedge t \notin [\![\phi]\!]$. By the definition of $\mathcal{R}'$, we have $(s, t) \notin \mathcal{R}'$ - Contradiction.

Condition 3: Suppose there exists $\bar{s} \in \mathit{paths}(s)$ such that $\forall s_i \in \bar{s} \cdot s_i \mathcal{R}' t$. We want to show that there is some $s' \in \bar{s}$ and some $t \rightarrow t'$ such that $s' \mathcal{R} t'$. Suppose this is not

the case. We distinguish two cases:

1. t does not have any outgoing transition. Thus, $s \in [\![EG\top]\!]$. But, $t \notin [\![EG\top]\!]$ because t does not have any outgoing transition - Contradiction.

2. t has some outgoing transition. Then, every t-path contains a state from the set $C = \{t'' | t \rightarrow t'' \wedge \forall s' \in \bar{s} \cdot (s', t'') \notin \mathcal{R}'\}$. Since t has some outgoing transition, and the number of states in the underlying Kripke Structure is finite, C is nonempty and finite. Thus, $C = \{t''_1, t''_2, \cdots, t''_n\}$ with $n \geq 1$.

   By the assumption that we made, we have $(s_j, t''_i) \notin \mathcal{R}'$ for every $t''_i \in C$ and every $s_j \in \bar{s}$. Thus, by the definition of $\mathcal{R}'$: $\exists \varphi_{ij} \in ECTL_{-X} \cdot s_j \in [\![\varphi_{ij}]\!] \wedge t''_i \notin [\![\varphi_{ij}]\!]$. Hence, for every $t''_i \in C$, there exists a formula φ_i, i.e. $\varphi_i = \bigvee_{s_j \in \bar{s}} \varphi_{ij}$, such that $t''_i \notin [\![\varphi_i]\!]$ but $\forall s_j \in \bar{s} \cdot s_j \in [\![\varphi_i]\!]$. Thus, $s \in [\![EG \bigwedge_{1 \leq i \leq n} \varphi_i]\!]$, but as every t-path contains some t''_i such that $t''_i \notin [\![\varphi_i]\!]$, we have $t \notin [\![EG \bigwedge_{1 \leq i \leq n} \varphi_i]\!]$. This implies that there is some formula ϕ such that $s \in [\![\phi]\!] \wedge t \notin [\![\phi]\!]$. By the definition of $\mathcal{R}'$, we have $(s, t) \notin \mathcal{R}'$ - Contradiction.

■

Example 4.10 In Figure 4.3, $s_1 \mathcal{R}_{stut} t_1$ and $s_1 \in [\![EGp \vee q]\!]$. Therefore, by Lemma 4.8, we can conclude that $t_1 \in [\![EGp \vee q]\!]$.

The following definition is a variant of the definition of stuttering simulation (Definition 4.3).

Definition 4.11 *Let $M = (\Sigma, \rightarrow, I, Ap)$ be a Kripke Structure, and let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a binary relation such that $\forall s, t \in \Sigma \cdot s \mathcal{R} t$ if and only if* [1]*:*

1. $\forall p \in Ap \cdot I(s, p) = \top \Rightarrow I(t, p) = \top$

[1] Recall Definition 2.4

2. *For every* $s \rightarrow s_1$, *there exists* $t_0 = t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_{m-1} \rightarrow t_m$ *such that* $s\mathcal{R}t_{m-1}, s\mathcal{R}_0 t_i$ *for every* $i < m-1$, *and* $s_1\mathcal{R}t_m$

3. $\exists \bar{s} \in paths(s) \cdot \forall s_i \in \bar{s} \cdot s_i\mathcal{R}t \Rightarrow \exists t' \cdot (t \rightarrow t' \wedge \exists s' \in \bar{s} \cdot s'\mathcal{R}t')$

Then, $\mathcal{R}$ *is a divergence-sensitive stuttering simulation. The largest such relation is denoted* $\mathcal{R}_{stut1}$.

At first glance, the above definition may seem weaker than Definition 4.3, but Lemma 4.12 shows that both definitions, in fact, result in the same preorder relation.

Lemma 4.12 $\mathcal{R}_{stut} = \mathcal{R}_{stut1}$.

Proof

$\Rightarrow$ Obvious.

$\Leftarrow$ Suppose $s\mathcal{R}_{stut1}t$ for states $s, t \in \Sigma$. We prove that $s\mathcal{R}_{stut}t$. Since $\mathcal{R}_{stut}$ is the largest stuttering simulation relation, it suffices to show that the pair (s,t) is an element of *some* stuttering simulation relation $\mathcal{R}' \subseteq \mathcal{R}_{stut}$. Consider a binary relation $\mathcal{R}'$ as follows: $s\mathcal{R}_{stut1}t \Rightarrow s\mathcal{R}'t$. We argue that $\mathcal{R}'$ satisfies the conditions of Definition 4.3.

Condition 1: Follows from Definition 4.11, Condition 1.

Condition 2: Suppose $s \rightarrow s_1$. By Definition 4.11, there exists $t_0 = t \rightarrow t_1 \rightarrow \cdots \rightarrow t_m$ such that $s\mathcal{R}'t_{m-1}$, $s\mathcal{R}_0 t_i$ for every $i < m-1$, and $s_1\mathcal{R}'t_m$. We prove that $s\mathcal{R}'t_i$ for every $i < m$ by induction on i. For $i = m-1$, we have $s\mathcal{R}'t_{m-1}$. Suppose $s\mathcal{R}'t_{i+1}$. We have to prove $s\mathcal{R}'t_i$. We show that $\mathcal{R}'$ satisfies the conditions of Definition 4.3.

Condition 1: $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t_i,p) = \top$. This follows from the fact that $s\mathcal{R}_0 t_i$.

Condition 2: Suppose $s \rightarrow s_1$. By the inductive hypothesis, there exists a sequence

$t_{i+1} \rightarrow \cdots \rightarrow t_m$ of states such that $s\mathcal{R}'t_j$ for $i+1 < j < m$ and $s_1\mathcal{R}'t_m$. Since $s\mathcal{R}'t_{i+1}$, there exists a sequence $t_i \rightarrow t_{i+1} \rightarrow \cdots \rightarrow t_m$ of states such that $s\mathcal{R}'t_j$ for $i < j < m$ and $s_1\mathcal{R}'t_m$.

Condition 3: Follows from $s\mathcal{R}'t_{i+1}$.

Condition 3: Follows from Definition 4.11, Condition 3.

■

As a result of Lemma 4.12, for every state t_i on a sequence $\bar{t}_{prefix} = t_0 \rightarrow \cdots \rightarrow t_{m-1}$ of states, we have $s\mathcal{R}_{stut}t_i$ if $s\mathcal{R}_{stut}t_{m-1}$ and $s\mathcal{R}_0t_j$ for every $t_j \in \bar{t}_{prefix}$. An interesting case of Definition 4.11 is when t_{prefix} forms a cycle. By Lemma 4.12, in order to prove that $s\mathcal{R}_{stut}t_i$ for every state t_i on a cycle $\bar{t}_{cycle}$, it suffices to show $s\mathcal{R}_{stut}t$ for a single state t on $\bar{t}_{cycle}$, and $s\mathcal{R}_0t'$ for other states t' on $\bar{t}_{cycle}$.

In the remainder of this section, we prove that for a symmetric relation $\mathcal{R}$, Definition 4.3 coincides with the definition of stuttering equivalence proposed in [Dam96]. The definition of stuttering equivalence [Dam96] for Kripke Structures is as follows:

Definition 4.13 (Stuttering equivalence) *[Dam96] Let $\equiv \subseteq \Sigma \times \Sigma$ be a **symmetric** relation such that $\forall s, t \in \Sigma \cdot s \equiv t$ if and only if:*

1. $\forall p \in Ap \cdot I(s,p) = I(t,p)$
2. *For every $s \rightarrow s_1$, there exists $t_0 = t \rightarrow \cdots \rightarrow t_{m-1} \rightarrow t_m$ such that $t \equiv t_i$ for every $i < m$ and $s_1 \equiv t_m$*
3. *s occurs on an infinite path $\bar{s}$ such that $\forall s_i \in \bar{s} \cdot s \equiv s_i$ if and only if t occurs on an infinite path $\bar{t}$ such that $\forall t_i \in \bar{t} \cdot t \equiv t_i$*

Then, $\equiv$ is a divergence-sensitive stuttering equivalence. The largest stuttering equivalence is denoted by $\equiv_{stut}$.

We prove that $\equiv_{stut}$ is equal to the following definition:

Definition 4.14 *Consider a* ***symmetric*** *relation* $\equiv \subseteq \Sigma \times \Sigma$ *such that* $\forall s, t \in \Sigma \cdot s \equiv t$ *if and only if:*

1. $\forall p \in Ap \cdot I(s,p) = \top \Rightarrow I(t,p) = \top$
2. *For every* $s \rightarrow s_1$, *there exists* $t_0 = t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_{m-1} \rightarrow t_m$ *such that* $s \equiv t_i$ *for every* $i < m$ *and* $s_1 \equiv t_m$
3. $\exists \bar{s} \in paths(s) \cdot \forall s_i \in \bar{s} \cdot s_i \equiv t \Rightarrow \exists t' \cdot (t \rightarrow t' \wedge \exists s' \in \bar{s} \cdot s' \equiv t')$

Then, $\equiv$ *is a divergence-sensitive stuttering equivalence. The largest such stuttering equivalence is denoted by* $\equiv_{stut1}$.

The following lemma states that both of the above definitions result in the same equivalence relation.

Lemma 4.15 *Both* $\equiv_{stut}$ *and* $\equiv_{stut1}$ *induce the same equivalence relation.*

Proof

$\Leftarrow$ Let $s \equiv_{stut1} t$. We define an equivalence relation $\equiv'$ as follows: $s \equiv_{stut1} t \Rightarrow s \equiv' t$. We prove that $\equiv'$ satisfies the conditions of Definition 4.13.

Condition 1: Follows from Definition 4.14, Condition 1 because $\equiv_{stut}$ is symmetric.

Condition 2: Follows from Definition 4.14, Condition 2.

Condition 3: Suppose s is on an infinite path $\bar{s}$ such that $\forall s_i \in \bar{s} \cdot s_i \equiv' s$. Since $s \equiv' t$, we have $\forall s_i \in \bar{s} \cdot s_i \equiv' t$. By Definition 4.14, we conclude that $\exists t' \cdot \exists s' \in \bar{s} \cdot t \rightarrow t' \wedge s' \equiv' t'$. Since $\forall s_i \in \bar{s} \cdot s_i \equiv' t$, we have $\exists t \cdot t \rightarrow t' \wedge t \equiv' t'$. Since the number of states is finite, there eventually exists a cycle $t \rightarrow t_1 \rightarrow \cdots \rightarrow t_m$ $(t = t_m)$, denoted $\bar{t}_{cycle}$, such that $\forall t_i \in \bar{t}_{cycle} \cdot t_i \equiv' t$. Therefore, there exists an infinite path $\bar{t}$ such that $\forall t_i \in \bar{t} \cdot t \equiv' t_i$.

$\Rightarrow$ Let $s \equiv_{stut} t$. We define an equivalence relation $\equiv'$ as follows: $s \equiv_{stut} t \Rightarrow s \equiv' t$. We prove that $\equiv'$ satisfies the conditions of Definition 4.14.

Condition 1: Follows from Definition 4.13, Condition 1.

Condition 2: Follows from Definition 4.13, Condition 2.

Condition 3: Suppose that $\exists \bar{s} \in \mathit{paths}(s) \cdot (\forall s_i \in \bar{s} \cdot s_i \equiv' t)$. Since $s \equiv' t$, we have $\forall s_i \in \bar{s} \cdot s_i \equiv' s$. By Definition 4.13, there exists some path $\bar{t}$ such that $\forall t_i \in \bar{t} \cdot t_i \equiv' t$. Therefore, $\exists t' \cdot t \rightarrow t' \ \wedge \ t' \equiv' t$. Since $t \equiv' s$, we have $\exists t' \cdot \exists s' \in \bar{s} \cdot t \rightarrow t' \ \wedge \ t' \equiv' s'$.

■

4.2 An algorithm for stuttering simulation

In this section, we propose an algorithm for computing $\mathcal{R}_{stut}$ by modifying the stuttering equivalence computation algorithm given in [GV90, BFH+92] so as to produce a preorder relation rather than an equivalence relation.

In the rest of this section, let $M_1 = (\Sigma_1, \ \rightarrow_1, \ I_1, \ Ap)$ and $M_2 = \ (\Sigma_2, \ \rightarrow_2, \ I_2, \ Ap)$ be Kripke Structures. Further, let variables s and t range over Σ_1 and Σ_2, respectively. We will hereafter drop the subscripts of $\rightarrow_1$, $\rightarrow_2$, I_1, and I_2 wherever there is no ambiguity. We want to find all pairs (s, t) of states such that $s\mathcal{R}_{stut}t$. In order to achieve this, we first give an algorithm for finding $\mathcal{R}_{dbs}$, (Definition 4.1). Then, we show that by adding a simple step to the algorithm for $\mathcal{R}_{dbs}$, we obtain an algorithm for $\mathcal{R}_{stut}$.

Definition 4.16 *Let $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ be a preorder relation. A devision set* [2] *$P_{\mathcal{R}}$ is defined as follows:*

$$P_{\mathcal{R}} \triangleq \{B_{s_i} \mid s_i \in \Sigma_1; B_{s_i} \neq \emptyset; \forall t \in \Sigma_2 \cdot s_i \mathcal{R} t \Leftrightarrow t \in B_{s_i}\}$$

[2] Division sets in this algorithm are analogous to partition sets in [GV90].

The elements of a division set are called *blocks*.

Let $\mathcal{R}, \mathcal{R}' \subseteq \Sigma_1 \times \Sigma_2$. A division set $P_{\mathcal{R}} = \{B_{s_i} \mid s_i \in \Sigma_1\}$ *refines* a division set $P_{\mathcal{R}'} = \{B'_{s_i} \mid s_i \in \Sigma_1\}$ if $\forall s_i \in \Sigma_1 \cdot B_{s_i} \subseteq B'_{s_i}$. A division set $P_{\mathcal{R}'}$ is *coarser* than a division set $P_{\mathcal{R}}$ if and only if $P_{\mathcal{R}}$ refines $P_{\mathcal{R}'}$.

Example 4.17 It can be verified that $P_{\mathcal{R}_{dbs}}$ refines $P_{\mathcal{R}_0}$. This is because for every $B'_{s_i} \in P_{\mathcal{R}_0}$ and $B_{s_i} \in P_{\mathcal{R}_{dbs}}$, we have $B_{s_i} \subseteq B'_{s_i}$. Letting $t \in B_{s_i}$, we get $s_i \mathcal{R}_{dbs} t$ by Definition 4.16. Then, by Definition 4.1, we get $s_i \mathcal{R}_0 t$. Thus, $t \in B'_{s_i}$.

Definition 4.18 *Let $P_{\mathcal{R}}$ be a division set, and let $B_{s_i}, B_{s_j} \in P_{\mathcal{R}}$. We define $pos(B_{s_i}, B_{s_j})$ as the set of all those states in B_{s_i} from which, after some initial stuttering, a state in B_{s_j} can be reached:*

$$pos(B_{s_i}, B_{s_j}) \triangleq \{t \in B_{s_i} \mid \exists n \geq 0 \cdot \exists t_0 = t \rightarrow \cdots \rightarrow t_n \cdot \ \forall i < n \cdot t_i \in B_{s_i} \wedge t_n \in B_{s_j}\}$$

We are now ready to give the algorithm for computing $\mathcal{R}_{dbs}$. This algorithm, which has been shown in Figure 4.5, starts with an initial division set P_0. Then, in every iteration of the while-loop (lines 3-14), it refines the division set P into a finer division set P'. The algorithm terminates when the division set produced in a step k is equal to that produced in step $k-1$. In each iteration of the while-loop (lines 3-14), every $B_{s_i} \in P'$ is replaced with the set $\bigcap_{\forall s_i \rightarrow s_j} pos(B_{s_i}, B_{s_j})$. If some block B_{s_i} turns out to be empty (line 7) in an iteration of the while-loop, then the algorithm will terminate.

Before stating the proof of correctness and termination of the algorithm in Figure 4.5, we give some preliminary definitions.

Definition 4.19 *The function $\mathbb{F} : \mathcal{P}(\Sigma_1 \times \Sigma_2) \rightarrow \mathcal{P}(\Sigma_1 \times \Sigma_2)$ is defined as follows:*

$$\begin{aligned}\mathbb{F}(X) \triangleq \{(s,t) \in X \mid \ & (s,t) \in \mathcal{R}_0 \ \wedge \ \forall s' \in \Sigma_1 \cdot s \rightarrow s' \Rightarrow \exists t_0 = t \rightarrow t_1, \cdots, \rightarrow t_m \cdot \\ & (s, t_i) \in X \text{ for every } i < m \wedge (s', t_m) \in X\}\end{aligned}$$

1: $P' = P = P_0$
2: $stable = \bot$
3: **while not** *stable* **do**
4: **for** $B_{s_j} \in P'$ **do**
5: **for** every $s_i \in \Sigma_1$ such that $s_i \rightarrow s_j$ **do**
6: $B_{s_i} = pos(B_{s_i}, B_{s_j})$
7: **if** $B_{s_i} = \emptyset$ **then exit**
8: **end for**
9: **end for**
10: **if** $P = P'$ **then**
11: $stable = \top$
12: **end if**
13: $P = P'$
14: **end while**

Figure 4.5: An algorithm for $\mathcal{R}_{dbs}$

Since $\mathcal{R}_{dbs}$ is the largest divergence-blind stuttering simulation relation defined over $\Sigma_1 \times \Sigma_2$, we have:

$$\mathcal{R}_{dbs} = \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a divergence-blind stuttering simulation relation}\}$$

It is easy to prove that:

$$\forall X \subseteq \Sigma_1 \times \Sigma_2 \cdot (\mathbb{F}(X) = X \Leftrightarrow X \text{ is a divergence-blind stuttering simulation relation})$$

Furthermore, $\mathbb{F}$ is monotonic over $(\mathcal{P}(\Sigma_1 \times \Sigma_2), \subseteq)$, and $(\mathcal{P}(\Sigma_1 \times \Sigma_2), \subseteq)$ is a complete lattice with $\cap$ and $\cup$ as *glb* and *lub* operators. Thus, by Knaster-Tarski Theorem [DP02], we can conclude that $\mathcal{R}_{dbs}$ is the greatest fixpoint of $\mathbb{F}$. Finally, by the finiteness of Σ_1 and Σ_2:

$$\mathcal{R}_{dbs} = \mathbb{F}^n(\Sigma_1 \times \Sigma_2) \text{ for some } n \geq 0$$

The number of iterations, i.e. n, is bounded by the length of the longest acyclic path between the states of M_1. Thus, n is bounded by the number of states in M_1, i.e. $n \leq |\Sigma_1|$.

Definition 4.20 *Let $P_{\mathcal{R}}$ be a division set. A set $X_{P_{\mathcal{R}}} \subseteq \Sigma_1 \times \Sigma_2$ is the corresponding tuple set of $P_{\mathcal{R}}$ if:*

$$X_{P_{\mathcal{R}}} = \{(s,t) \mid s \in \Sigma_1; t \in B_s\}$$

Lemma 4.21 *If $P_0 = P_{\mathcal{R}_0}$, then the algorithm shown in Figure 4.5 computes $P_{\mathcal{R}_{dbs}}$, and the while-loop of the algorithm (lines 3-14) terminates after at most $|\Sigma_1|$ iterations.*

Proof Let $X_{P'}$ be the corresponding tuple set of the division set P' on line 10. We prove that $X_{P'} = \mathbb{F}(X_P)$ where X_P is the corresponding tuple set of the division set P on line 10.

$X_{P'} = \{(s,t) | B_s \in P'; t \in B_s\}$

$\Rightarrow$ (by Algorithm in Figure 4.5)

$B_s \in P' \Rightarrow B_s = \bigcap\{pos(B_s^{old}, B_{s'}) | s \rightarrow s'; B_s^{old} \in P; B_{s'} \in P\}$

$\Rightarrow$ (by Definition 4.18)

$B_s \in P' \Rightarrow B_s = \{t \in B_s^{old} \mid \forall s \rightarrow s' \cdot \exists n \geq 0 \cdot \exists t_0 = t \rightarrow \cdots \rightarrow t_n \cdot \forall i < n \cdot t_i \in B_s^{old} \wedge t_n \in B_{s'}\}$

$\Rightarrow$ (by Definition 4.20)

$X_{P'} = \{(s,t) | \forall s \rightarrow s' \cdot \exists n \geq 0 \cdot \exists t_0 = t \rightarrow \cdots \rightarrow t_n \cdot \forall i < n \cdot (s,t_i) \in X_P \wedge (s',t_n) \in X_P\}$

$\Rightarrow$ (by Definition 4.19 and by the assumption that $P_0 = P_{\mathcal{R}_0}$)

$X_{P'} = \mathbb{F}(X_P)$

Hence, the for-loop (lines 4-9) computes $\mathbb{F}$. Thus, the final devision set $P_{\mathcal{R}_{dbs}}$ is computed after at most $|\Sigma_1|$ iterations of the while-loop in Figure 4.5. ∎

Now, we present an efficient implementation of the for-loop (lines 4-9) that runs in $O(m_1 \times m_2)$ time. Therefore, the overall running time of the algorithm is $O(n_1 \times m_1 \times m_2)$. Some of the steps in our implementation are similar to those in the implementation of the algorithm proposed in [GV90].

We refer to the transitions between the states in a same block as *internal* transitions and those between the states in different blocks as *external* transitions. Assume that there exists a cycle $\bar{t}_{cycle}$ consisting of only internal transitions in some block B_{s_i} of the initial division set $P_{\mathcal{R}_0}$. If we prove that there exists a block $B'_{s_i} \in P_{\mathcal{R}_{dbs}}$ such that $t_i \in B'_{s_i}$ for some state $t_i \in \bar{t}_{cycle}$, then, by Lemma 4.12, all the states on $\bar{t}_{cycle}$ are present in B'_{s_i}. In other words, we only need to show *some* state of $\bar{t}_{cycle}$ is present in a block of the final division set in order to be able to infer that *all* states of $\bar{t}_{cycle}$ are present in that block. Therefore, as a preprocessing step, we look for cycles consisting of internal transitions in every $B_{s_i} \in P_{\mathcal{R}_0}$ and collapse these cycles to one state. More importantly, it is sufficient to implement this algorithm for the case where blocks contain no cycles consisting of internal transitions.

For $B_{s_i} \in P$, we define the set $bottom(B_{s_i})$ consisting of the states that have no successor in B_{s_i} as follows [GV90]:

$$bottom(B_{s_i}) \triangleq \{t \in B_{s_i} \mid \forall t' \in \Sigma_2 \cdot (t \rightarrow t' \Rightarrow t' \notin B_{s_i})\}$$

The following lemma expresses an important observation that has been exploited in the implementation of our algorithm.

Lemma 4.22 *Given a pair of blocks B_{s_i} and B_{s_j}, for every state $t \in bottom(B_{s_i})$, we have $t \in pos(B_{s_i}, B_{s_j})$ if and only if either $t \in B_{s_j}$ or there exists a state t' such that $t \rightarrow t'$ and $t' \in B_{s_j}$.*

The basic idea of the algorithm is shown in Figure 4.6. In order to compute $pos(B_{s_i}, B_{s_j})$ for blocks B_{s_i} and B_{s_j}, we check every $t \in bottom(B_{s_i})$ to see if either $t \in B_{s_j}$ or there exists some state $t' \in B_{s_j}$ such that $t \rightarrow t'$. Since there is no cycle of internal transitions in B_{s_i}, there must be a path of internal transitions from every *non-bottom* state in B_{s_i} to some *bottom* state in B_{s_i}. Hence, if for every *bottom* state in B_{s_i}, we show that some state in B_{s_j} is reachable by taking *at most* one step, then we can conclude that from *every* state in B_{s_i}, after some initial stuttering steps, some state in B_{s_j} is reachable.

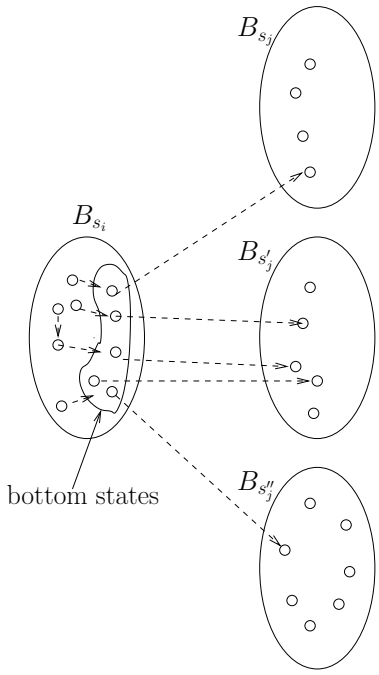

Figure 4.6: Blocks and bottom states

In our algorithm, for every pair of states s_i and s_j such that $s_i \rightarrow s_j$, we compute $pos(B_{s_i}, B_{s_j})$. To do this, we scan the bottom states in B_{s_i}. If we cannot reach a state in B_{s_j} from a bottom state in B_{s_i} by taking at most one step, then we remove that bottom state from B_{s_i}. After scanning all the bottom states in B_{s_i}, if some bottom state has been removed, then we scan the *non-bottom* states in B_{s_i} and remove those non-bottom states from which none of the states in B_{s_j} or current bottom states in B_{s_i} can be reached.

To improve the running time of the algorithm, we need some preprocessing steps. The implementation of the algorithm includes four preprocessing steps. Here, we explain these preprocessing steps, the required data structures, and the main part of the algorithm.

In the implementation of the algorithm, for each block, state, and transition, there is a corresponding record of type block, state, and transition, respectively. In the arguments given hereafter in this section, we will not distinguish between transitions, states, blocks, etc. and the data structures representing them. We assume that for each of the Kripke

structures M_1 and M_2, there is a linked list of the states of that Kripke Structure (see Figure 4.7). In M_1, each state s_i points to the list of its predecessors, denoted $pre(s_i)$, and in M_2, each state t_i points to the list of its successors, denoted $succ(t_i)$. Furthermore, in M_2, each state contains a pointer to an array of length n_1 called `block-trace`. For a state t_i, t_i.`block-trace`$[j]$ is true if and only if $t_i \in B_{s_j}$. We use t_i.`block-trace` to refer to the block-trace array that belongs to state t_i [3].

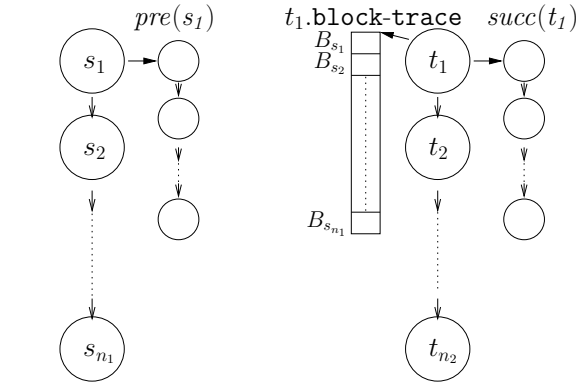

Figure 4.7: Data structures for Kripke Structures M_1 and M_2

The first preprocessing step is constructing the initial division set $P_{\mathcal{R}_0}$. This can be done by scanning all states $s_i \in \Sigma_1$ and accumulating in B_{s_i} all the states $t_j \in \Sigma_2$ such that $s_i \mathcal{R}_0 t_j$. The fragment of code that constructs the initial division set $P_{\mathcal{R}_0}$ is shown in Figure 4.8.

Example 4.23 If we execute `construct-P`$_0$ on the Kripke Structures in Figure 4.3, we obtain $B_{s_1} = \{t_1, t_2, t_3, t_4\}$ and $B_{s_2} = \{t_1, t_4, t_5\}$. Moreover, the `block-trace` arrays of the states of M_1 are as follows:

t_1.`block-trace` $= \{1 \mapsto \top, 2 \mapsto \top\}$, t_2.`block-trace` $= \{1 \mapsto \top, 2 \mapsto \bot\}$,

t_3.`block-trace` $= \{1 \mapsto \top, 2 \mapsto \bot\}$, t_4.`block-trace` $= \{1 \mapsto \top, 2 \mapsto \top\}$,

[3] It is obvious that . binds tighter than [ ].

1: **proc** `construct-P`$_0$
2: **for** $s_i \in \Sigma_1$ **do**
3: **for** $t_j \in \Sigma_2$ **do**
4: **if** $s_i \mathcal{R}_0 t_j$ **then**
5: add t_j to the block B_{s_i}
6: t_j.`block-trace`$[i] = \top$.
7: **end if**
8: **end for**
9: **end for**

Figure 4.8: Procedure for computing $P_{\mathcal{R}_0}$

t_5.`block-trace` $= \{1 \mapsto \bot, 2 \mapsto \top\}$, t_6.`block-trace` $= \{1 \mapsto \bot, 2 \mapsto \bot\}$.

After executing `construct-P`$_0$, we have a linked list of blocks such that each block in the linked list points to the list of the states belonging to that block.

In the next preprocessing step, we remove cycles consisting of only internal transitions in the blocks of the initial division set P_0. This can be done by a standard depth first search algorithm (cf. e.g. [AHU74]).

Example 4.24 In block B_{s_1} of the Kripke Structures in Figure 4.3, there exists a cycle consisting of states t_1 and t_3. We collapse this cycle into a state $t_{1'}$. Therefore, we get $B_{s_1} = \{t_{1'} = (t_1, t_3), t_2, t_4\}$ and $B_{s_2} = \{t_1, t_4, t_5\}$.

Next, we run the procedure `divide-bottom-nonbottom`, given in Figure 4.9, to separate non-bottom states from bottom states in every block. After executing `divide-bottom-nonbottom`, each block B_{s_i} points to the list of bottom states and the list of non-bottom states in B_{s_i}. Moreover, each non-bottom state points to the list of internal transitions emanating from that state. Since we assumed there is no cycle of internal transitions in the initial blocks, we can order the list of non-bottom states in

```
1: proc divide-bottom-nonbottom
2: for s_i ∈ Σ_1 do
3:   for t_j ∈ B_{s_i} do
4:     bottom = ⊤
5:     for t_k ∈ succ(t_j) do
6:       if t_k.block-trace[i] then
7:         add t_j → t_k to the list of internal transitions of t_j
8:         bottom = ⊥
9:       end if
10:    end for
11:    if bottom then
12:      add t_j to the list of bottom states of B_{s_i}
13:    end if
14:  end for
15: end for
```

Figure 4.9: Procedure for partitioning the set of states into bottom and non-bottom

each block in such a way that whenever $t \rightarrow t'$ for non-bottom states $t, t' \in B_{s_i}$, t' comes after t in the list of B_{s_i}'s non-bottom states.

Example 4.25 Figure 4.11 illustrates the blocks of the Kripke Structures in Figure 4.3 after preprocessing steps. In $B_{s_1} = \{t_{1'} = (t_1, t_3), t_2, t_4\}$, states t_2 and t_4 are the bottom states as they do not have any successor states in B_{s_1}; and transitions $t_3 \rightarrow t_2$ and $t_3 \rightarrow t_4$ are the internal transitions of B_{s_1}. Since t_1 and t_3 have collapsed to $t_{1'}$, the transition from t_3 to t_2 (resp. t_4) is represented as a transition from $t_{1'}$ to t_2 (resp. t_4). In $B_{s_2} = \{t_1, t_5, t_4\}$, states t_1 and t_4 are the bottom states; and the transition from t_5 to t_1 is the only internal transition.

In the last preprocessing step, we determine the external transitions that end in B_{s_i}.

```
1: proc external-transitions
2: for s_j ∈ Σ_1 do
3:   for s_i ∈ pre(s_j) do
4:     for t_k ∈ B_{s_i} do
5:       for t_l ∈ succ(t_k) do
6:         if t_l.block-trace[j] then
7:           add t_k → t_l to the list of external transitions of B_{s_j}.
8:         end if
9:       end for
10:    end for
11:  end for
12: end for
```

Figure 4.10: Procedure for finding external transitions

This can be done by executing the procedure given in Figure 4.10. Notice that when executed on a block B_{s_j}, `external-transition` adds a transition $t_k \rightarrow t_l$ to the list of the external transitions of B_{s_j} if $t_k \in B_{s_i}$, $t_l \in B_{s_j}$, and $s_i \rightarrow s_j$. This is because, we do not have to consider the external transitions from the states of B_{s_i} to the states of B_{s_j} unless $s_i \rightarrow s_j$. Thus, we can restrict the list of the external transitions of each block to only those external transitions required for the computation of the relation $\mathcal{R}_{dbs}$. This helps reduce the running time of our algorithm.

Example 4.26 In the Kripke Structures shown in Figure 4.3, there are two external transitions that end in B_{s_1}: one from t_5 to t_1, and the other from t_1 (in B_{s_2}) to t_3. Since t_1 and t_3 have collapsed to $t_{1'}$, the external transitions of B_{s_1} are represented as $t_5 \rightarrow t_{1'}$ and $t_1 \rightarrow t_{1'}$. Similarly, there are two external transitions in B_{s_2}: one from t_2 to t_5, and the other from t_3 to t_4 shown as a transition from $t_{1'}$ to t_4.

After doing the preprocessing steps, we will have a linked list of blocks as illustrated

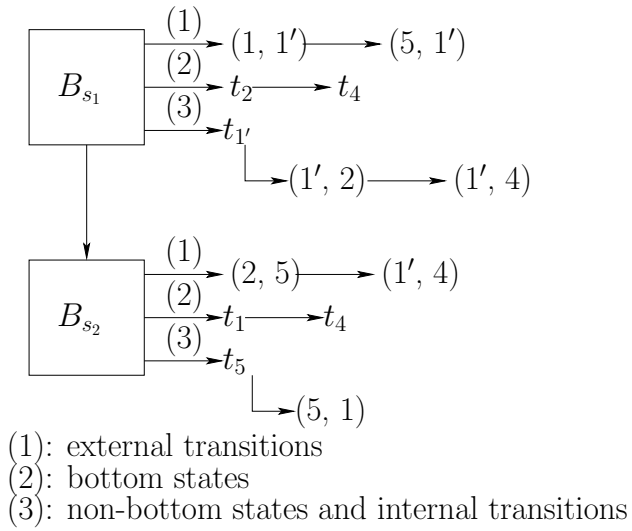

Figure 4.11: Blocks after preprocessing steps

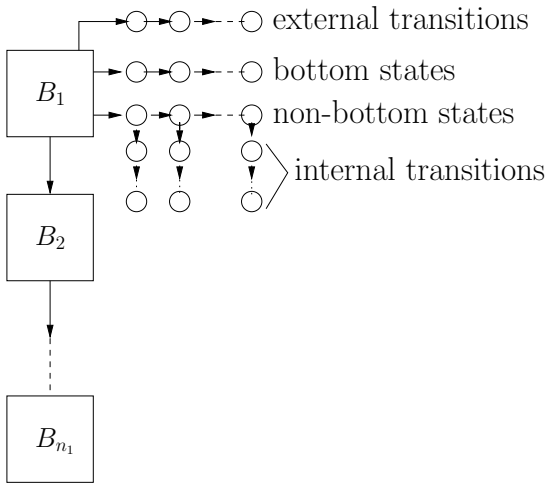

Figure 4.12: Data structure for blocks

in Figure 4.12. Each block B_{s_i} points to:

1. a list of the external transitions that end in B_{s_i},
2. a list of bottom states in B_{s_i}, and
3. a list of non-bottom states in B_{s_i}.

Each non-bottom state t points to the list of the internal transitions emanating from t. Each external transition contains two pointers: on pointing to its starting state and the other to its target state. Each state t, irrespective of being bottom or non-bottom, points to both its corresponding state in the linked list M_2 (Figure 4.7) and the block to which it belongs. Each state s in the linked list M_1 (Figure 4.7) contains a pointer to its block. In each block, each state t has an auxiliary boolean flag which is initially set to zero.

We now describe how to perform the main part of the algorithm, i.e. the for-loop (lines 4-9, Figure 4.5), in $O(m_1 \times m_2)$ time. Let B_{s_j} be the first block in the linked list of the blocks. As noted earlier, B_{s_j} has a pointer to the list of all external transitions that end in B_{s_j}. We start by scanning the list of external transitions that end in B_{s_j}. When an external transition is visited, the flag of the starting state is raised. After having scanned all the external transitions ending in B_{s_j}, we consider the list of all the predecessors of state s_j, which we earlier denoted $pre(s_j)$. Let s_i be the head record in the linked list $pre(s_j)$. We scan the list of bottom states in B_{s_i}. If there is some state $t \in bottom(B_{s_i})$ whose flag has not been raised, then we check $t.\texttt{block-trace}[j]$. If $t.\texttt{block-trace}[j] = \perp$, then, by Lemma 4.22, $t \notin pos(B_{s_i}, B_{s_j})$. Thus, we remove t from B_{s_i}.

Once scanning the list of bottom states in B_{s_i} is completed, we check to see if any bottom states have been removed. If some bottom state has been removed, then we scan the list of non-bottom states in B_{s_i}, and perform the following: if for some non-bottom state $t \in B_{s_i}$, the flag has not been raised and if none of the internal transitions of t lead to an existing state in B_{s_i}, then we distinguish two cases:

1. $t.\texttt{block-trace}[j] = \perp$: we remove t from B_{s_i}.

2. $t.\texttt{block-trace}[j] = \top$: we remove t from the list of non-bottom states in B_{s_i} and add it to the list of bottom states in B_{s_i}.

After carrying out all the activities described above for B_{s_i}, we consider the next block referred to by the next state in the list $pre(s_j)$ and repeat the above steps again. When we have dealt with all the elements of the list $pre(s_j)$, we apply the same procedure to the next block in the list of blocks.

Example 4.27 We explain how to compute $\mathcal{R}_{dbs}$ for the Kripke Structures in Figure 4.3. We first scan the list of all external transitions in B_{s_1}, i.e. $(1, 1')$ and $(5, 1')$ (see Figure 4.11), and raise the flags of t_1 and t_5 (in B_{s_2}). Then, we consider block B_{s_2} (because s_2 is the predecessor of s_1) and scan the list of its bottom states. The flag of t_1 has been raised, but not that of t_4. However, $t_4.\texttt{block-trace}[1] = \top$ meaning that t_4 is present in B_{s_1}. Therefore, $pos(B_{s_2}, B_{s_1})$ is equal to B_{s_2}.

We follow the same procedure for B_{s_2}. Since there is an external transition from t_2 (in B_{s_1}) to t_5 (in B_{s_2}), and t_4 is present in B_{s_2}, we do not remove any states from B_{s_1} either. Since no changes have occurred in this iteration of the algorithm, the blocks are stable. Thus, the algorithm terminates and yields the following relation:

$$\mathcal{R}_{dbs} = \{(s_1, t_1), (s_1, t_2), (s_1, t_3), (s_1, t_4), (s_2, t_1), (s_2, t_4), (s_2, t_5)\}$$

Definition 4.28 *Let $\bar{s}$ be an infinite path in a Kripke Structure M, let $\mathcal{R} \subseteq \mathcal{R}_{dbs}$, and let t be a state in a Kripke Structure M' such that $\forall s_i \in \bar{s} \cdot s_i \mathcal{R} t$. Then, we refer to t as an infinite stuttering state.*

To compute $\mathcal{R}_{stut}$, we should add an extra step for detecting the infinite stuttering states to the algorithm. To achieve this, during the execution of the main part of the algorithm, i.e. the for-loop (lines 4-9, Figure 4.5), when we are scanning the states of some block B_{s_i} such that $s_i \in pre(s_j)$, if for some $t \in B_{s_i}$:

1. the flag of t has not been raised,

2. none of its internal transitions lead to an existing state in B_{s_i}, and

3. t.`block-trace`$[j] = \top$;

we add a tuple (i, j, t) to a linked list named *infinite-stuttering*. Clearly, for a state t that satisfies the above-mentioned conditions, we have $t \in \mathit{bottom}(B_{s_i})$ and $t \in \mathit{bottom}(B_{s_j})$. Thus, an element (i, j, t) in the *infinite-stuttering* list indicates that there exist $s_i, s_j \in \Sigma_1$ and $t \in \Sigma_2$ such that $s_i \rightarrow s_j \ \wedge \ s_i \mathcal{R} t \ \wedge \ s_j \mathcal{R} t$, but there exists no state t' such that $t \rightarrow t' \ \wedge (s_i \mathcal{R} t' \ \vee \ s_j \mathcal{R} t')$.

In each iteration of the while-loop (lines 3-14, in Figure 4.5), we look for cycles $s_{i_1} \rightarrow s_{i_2} \rightarrow \cdots \rightarrow s_{i_m}$ $(s_{i_1} = s_{i_m})$ such that $(i_j, i_{j+1}, t) \in$ *infinite-stuttering* for some $t \in \Sigma_2$ and for every $1 \leq j < m$. If such a cycle exists, then t is an infinite stuttering state because there exists an infinite path $\bar{s}_{cycle}$ such that $\forall s_i \in \bar{s}_{cycle} \cdot s_i \mathcal{R} t$. Moreover, t is a bottom state in every block B_{s_i} such that $s_i \in \bar{s}_{cycle}$. Thus, t does not have any successor t' such that $s' \mathcal{R} t'$ for some $s' \in \bar{s}_{cycle}$. Therefore, we remove t from every block B_{s_i} when $s_i \in \bar{s}_{cycle}$.

Example 4.29 We show how to compute $\mathcal{R}_{stut}$ for the Kripke Structures in Figure 4.3. As mentioned in Example 4.27, after scanning the external transitions in B_{s_1}, the flag of t_4 will not have been raised. But since t_4 is present in B_{s_1}, we do not remove it from B_{s_2}. Moreover, t_4 is a bottom state in both B_{s_1} and B_{s_2}. Therefore, we add a tuple $(1, 2, t_4)$ to the *infinite-stuttering* list. Similarly, after scanning the external transitions in B_{s_2}, we add another tuple $(2, 1, t_4)$ to the *infinite-stuttering* list. Since states s_1 and s_2 are located on a cycle, we have to remove state t_4 from both B_{s_1} and B_{s_2}. Figure 4.13 illustrates blocks B_{s_1} and B_{s_2} after removing t_4. Finally, we obtain

$$\mathcal{R}_{stut} = \{(s_1, t_1), (s_1, t_2), (s_1, t_3), (s_2, t_1), (s_2, t_5)\}$$

In the rest of this section, we prove that the running time of the algorithm shown in Figure 4.5 is $O(n_1 \times m_1 \times m_2)$, and that its space complexity is $O(m_1 \times m_2)$. Assume

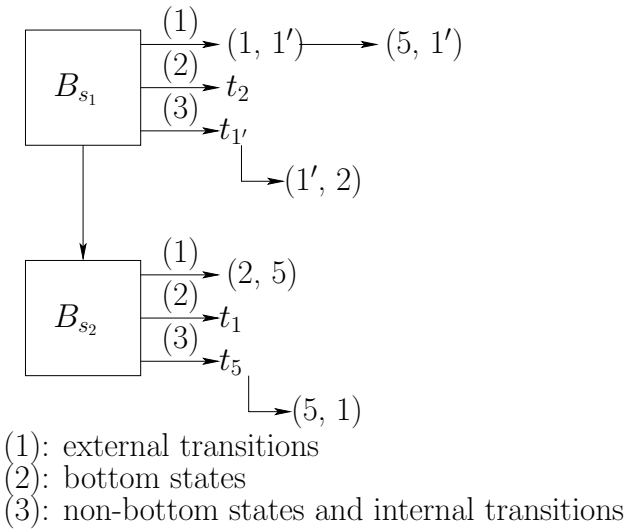

Figure 4.13: Blocks after computing $\mathcal{R}_{stut}$

that $|\Sigma_1| = n_1$, $|\Sigma_2| = n_2$, the number of transitions in M_1 is m_1, and the number of transitions in M_2 is m_2. We compute the running time and the space complexity of the algorithm step by step.

For the preprocessing steps, we have:

1. the complexity of `construct-P0` is $O(n_1 \times n_2)$;

2. the complexity of `divide-bottom-nonbottom` is $O(m_2 \times n_1)$;

3. the complexity of `external-transition` is $O(m_1 \times m_2)$; and

4. the complexity of finding cycles by a depth first search algorithm in the blocks of the initial devision set is $O(m_2)$ per block, and therefore, $O(n_1 \times m_2)$ for all blocks.

The next step is computing the main part of the algorithm, i.e. the for-loop (lines 4-9, in Figure 4.5). In the main part of the algorithm, for each block B_{s_j}, we scan the list of external transitions that end in B_{s_j}. Remember that for each block, we consider only those external transitions whose starting states are in some block B_{s_i} such that $s_i \rightarrow s_j$.

Thus, the number of external transitions ending in B_{s_j} is bounded by $m_2 \times d_{s_j}$ where d_{s_j} is the fan-in of s_j.

Then, we scan the list of bottom states in all blocks B_{s_i} such that $s_i \rightarrow s_j$. In some cases, we also have to scan the list of non-bottom states in B_{s_i}. Since there are no cycles of internal transitions in any block, and the list of non-bottom states in each block is ordered, the complexity of scanning and removing non-bottom states for every B_{s_i} is $O(m_2)$. Thus, the complexity of scanning bottom and non-bottom states in $\{B_{s_i} | s_i \rightarrow s_j\}$ is $O(m_2 \times d_{s_j})$. Therefore, the running time of performing this procedure for every block B_{s_j} is $O(n_1 \times d_{s_j} \times m_2) = O(m_1 \times m_2)$. Hence, the running time of the algorithm for computing $\mathcal{R}_{dbs}$ (Figure 4.5) is $O(n_1 \times m_1 \times m_2)$.

The computation of $\mathcal{R}_{stut}$ has an extra step which is finding the cycles in M_1 and removing infinite stuttering states. Here, we give an upper-band for this step, but we conjecture that this upper bound can be improved significantly. The complexity of finding cycles in M_1 is $O(m_1)$. In the worst case, for every state $t \in \Sigma_2$, we have to find cycles of states $s \in \Sigma_1$. This can be done in $O(m_1 \times n_2)$. Since for every state s, we have at most n_2 stuttering states, the running time of finding cycles and removing the infinite-stuttering states is bounded by $O(m_1 \times {n_2}^2)$. Thus, the running time of the algorithm for computing $\mathcal{R}_{stut}$ is bounded by $O(m_1 \times {n_2}^2 + n_1 \times m_1 \times m_2) = O(max(n_1, n_2) \times m_1 \times m_2)$.

In the end, the space complexity of the algorithm is $O(m_1 \times m_2)$ because:

1. the space complexity of M_1 is $O(m_1)$;

2. the space complexity of M_2 is $O(m_2 + n_1 \times n_2)$ because each state has an array *block-trace* of length n_1; and

3. the space complexity of the list of blocks is $O(m_2 \times m_1)$.

4.3 Stuttering refinement on Kripke Modal Transition Systems

In this section, we propose a refinement relation over Kripke Modal Transition Systems that is insensitive to finite stuttering. We assume every KMTS $M = (M_{must}, M_{may})$ is total. This means that each state has at least one outgoing must or may-transition. However, M_{must} is not necessarily total.

Definition 4.30 (Stuttering refinement) *Let $M = (\Sigma, \rightarrow^{must}, \rightarrow^{may}, I^{may}, I^{must}, Ap)$ be a KMTS. We define a binary relation $\preceq \subseteq \Sigma \times \Sigma$ such that $\forall s, t \in \Sigma \cdot t \preceq s$ if and only if:*

1. $\forall p \in Ap \cdot I^{must}(s,p) = \top \Rightarrow I^{must}(t,p) = \top$
2. $\forall p \in Ap \cdot I^{may}(t,p) = \top \Rightarrow I^{may}(s,p) = \top$
3. *For every $s \rightarrow^{must} s_1$, there exists $t_0 = t \rightarrow^{must} t_1 \rightarrow^{must} t_2 \rightarrow^{must} \cdots t_{m-1} \rightarrow^{must} t_m$ such that $\wedge\ t_i \preceq s$ for every $i < m$ and $t_m \preceq s_1$*
4. *For every $t \rightarrow^{may} t_1$, there exists $s_0 = s \rightarrow^{may} s_1 \rightarrow^{may} s_2 \rightarrow^{may} \cdots s_{m-1} \rightarrow^{may} s_m$ such that $t \preceq s_i$ for every $i < m$ and $t_1 \preceq s_m$*
5. $\exists \bar{s} \in \textit{must-paths}(s) \cdot \forall s_i \in \bar{s} \cdot t \preceq s_i \Rightarrow \exists t' \cdot (t \rightarrow^{must} t' \wedge \exists s' \in \bar{s} \cdot t' \preceq s')$
6. $\exists \bar{t} \in \textit{may-paths}(t) \cdot \forall t_i \in \bar{t} \cdot t_i \preceq s \Rightarrow \exists s' \cdot (s \rightarrow^{may} s' \wedge \exists t' \in \bar{t} \cdot t' \preceq s')$

Then, $\preceq$ is a stuttering refinement relation. The largest refinement relation is denoted $\preceq_{stut}$.

Example 4.31 Figure 4.14 illustrates two KMTSs M_1 and M_2. By Definition 4.30, it is easy to check that $t_1 \preceq_{stut} s_1$ and $t_1 \preceq_{stut} s_2$.

Example 4.32 In Figure 4.14, states s_1 and s_2 form an infinite *must-path*, and s_2 together with its self-loop transition form an infinite *may-path*. Since $t_1 \preceq_{stut} s_1$ and $t_1 \preceq_{stut} s_2$, by Definition 4.30, there has to be a must transition from t_1 to some t' that refines either s_1 or s_2. It is clear from Figure 4.14 that t_1 has a must-transition to t_3 and that $t_3 \preceq_{stut} s_1$. But since the self-loop transition of s_2 is not a must-transition, t_1's successors do not have to refine s_2.

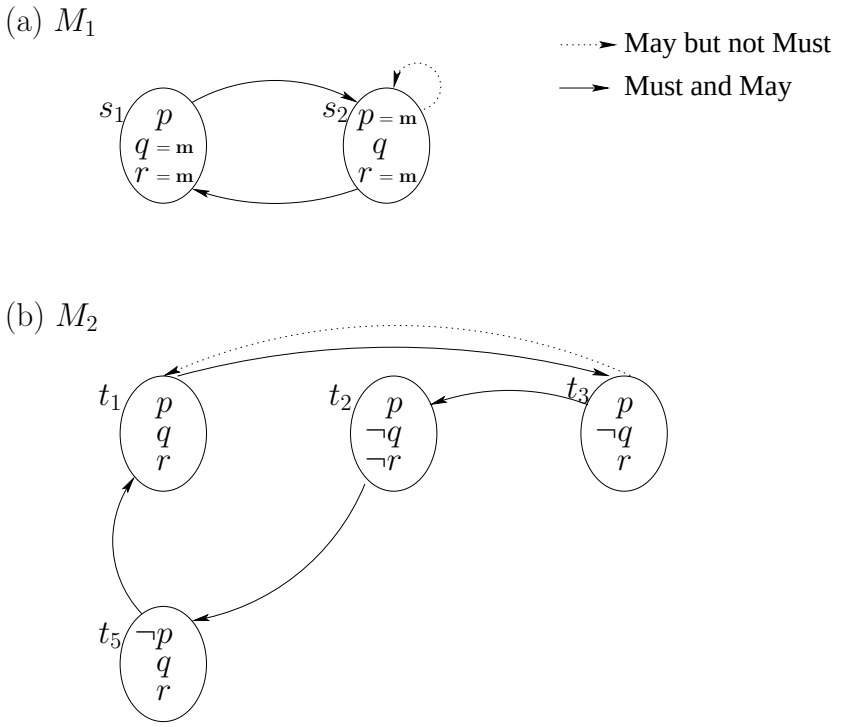

Figure 4.14: An example of stuttering refinement

The conditions of Definition 4.30 are similar to those of Definition 4.3. An immediate result of Definition 4.3, which is given by Lemma 4.7, is that for every infinite path emanating from state s, there exists a corresponding infinite path emanating from state t. Definition 4.30 implies that for every infinite *must*-path emanating from s, there exists an infinite *must*-path emanating from t; and for every infinite *may*-path emanating from t, there exists an infinite *may*-path emanating from s; but, an infinite *may*-path emanating

from s is not necessarily mapped to an infinite *may* or *must*-path emanating from t.

Although Definition 4.30 does not force the infinite *may*-paths emanating from s to be simulated by infinite paths emanating from t, we can still prove that the relation $\preceq_{stut}$ preserves CTL_{-X} properties. The crucial point is that an infinite *may*-path cannot be a witness for the truth of an existential property. For example, in Figure 4.14, s_2 has an infinite *may*-path such that all the states on that path are labeled with q; but, $s_2 \notin [\![EGq]\!]^{\top}$. Moreover, since universal operators range over *all* *may*-paths, and *may*-paths in M_2 are simulated by *may*-paths in M_1, the universal properties that are true in M_1 are true in M_2 as well. Therefore, in order to preserve both existential and universal properties, infinite *may*-paths in M_1 are not required to be simulated by infinite paths in M_2.

Theorem 4.33 (Logical characterizaionf of $\preceq_{stut}$) *$t \preceq_{stut} s$ if and only if*

$$\forall \phi \in CTL_{-X} \cdot s \in [\![\phi]\!]^{\top} \Rightarrow t \in [\![\phi]\!]^{\top} \text{ and } s \in [\![\phi]\!]^{\perp} \Rightarrow t \in [\![\phi]\!]^{\perp}$$

We consider two stuttering simulation relations $\mathcal{R}^{must}_{stut}$ and $\mathcal{R}^{may}_{stut}$ over M_{must} and M_{may}, respectively. Clearly, $t \preceq_{stut} s$ implies $s\mathcal{R}^{must}_{stut}t$ and $t\mathcal{R}^{may}_{stut}s$, but not vice versa.

Proof (*sketch*)

$\Rightarrow$ Let ϕ be a CTL_{-X} property. Proof by induction.

- $\phi = p, \neg\varphi, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2$: Obvious.
- $\phi = EG\varphi$ or $\phi = E[\varphi U \psi]$: Suppose $s \in [\![\phi]\!]^{\top}$ in a KMTS M. Then, $s \in [\![\phi]\!]^{\top}$ in M_{must}. Since $s\mathcal{R}^{must}_{stut}t$, by Lemma 4.8, $t \in [\![\phi]\!]^{\top}$ in M_{must} as well. Thus, $t \in [\![\phi]\!]^{\top}$ in M. Now, suppose $s \in [\![\phi]\!]^{\perp}$ in M. Then, $s \in [\![\phi]\!]^{\perp}$ in M_{may}. Since $t\mathcal{R}^{may}_{stut}s$, by Lemma 4.8, $t \in [\![\phi]\!]^{\perp}$ in M_{may} as well. Thus, $t \in [\![\phi]\!]^{\perp}$ in M.

$\Leftarrow$ The lemma can be proved in the same way as Lemma 4.9. There is only a slight difference: in Lemma 4.9, when $(s,t) \notin \mathcal{R}'$ for states s and t, we conclude that $\exists \phi \in ECTL_{-X} \cdot s \in [\![\phi]\!]^{\top} \wedge t \in [\![\phi]\!]^{\perp}$. But in this lemma, when $s \not\preceq' t$, we conclude

that $\exists \varphi \in CTL_{-X} \cdot (s \in [\![\varphi]\!]^{\top} \wedge t \in [\![\varphi]\!]^{\bot}) \bigvee (s \in [\![\varphi]\!]^{\bot} \wedge t \in [\![\varphi]\!]^{\top})$. We replace every formula $\varphi \in CTL_{-X}$ with a formula $\varphi' = \neg\varphi$ if $s \in [\![\varphi]\!]^{\bot}$. The rest of the proof will be the same as that of Lemma 4.9.

■

Example 4.34 In Figure 4.14, $t_1 \preceq_{stut} s_1$ and $s_1 \in [\![AGp \vee q]\!]^{\top}$. By Theorem 4.33, we conclude that $t_1 \in [\![AGp \vee q]\!]^{\top}$.

4.4 An algorithm for stuttering refinement

In this section, let M_1 and M_2 be Kripke Modal Transition Systems, and let variables s and t range over Σ_1 and Σ_2, respectively. We want to find all pairs (s,t) of states such that $t \preceq_{stut} s$. We first present a fixpoint characterization of the relation $\preceq_{stut}$. We define the functions $\mathbb{F}^{may}, \mathbb{F}^{must} : \mathcal{P}(\Sigma_1 \times \Sigma_2) \to \mathcal{P}(\Sigma_1 \times \Sigma_2)$ as follows:

$$\begin{aligned}
\mathbb{F}^{must}(X) \triangleq \{(s,t) \in X \mid\ & \forall s' \in \Sigma_1 \cdot s \rightarrow^{must} s' \Rightarrow \exists t_0 = t \rightarrow^{must} t_1, \cdots, \rightarrow^{must} t_m \cdot \\
& (s,t_i) \in X \text{ for every } i < m \ \wedge\ (s',t_m) \in X\} \\
\mathbb{F}^{may}(X) \triangleq \{(s,t) \in X \mid\ & \forall t' \in \Sigma_2 \cdot t \rightarrow^{may} t' \Rightarrow \exists s_0 = s \rightarrow^{may} s_1, \cdots, \rightarrow^{may} s_m \cdot \\
& (s_i,t) \in X \text{ for every } i < m \ \wedge\ (s_m,t') \in X\}
\end{aligned}$$

It can be seen from the above definitions that $\mathbb{F}^{may}$ and $\mathbb{F}^{must}$ are similar to the function $\mathbb{F}$ defined in Definition 4.19.

We now define a function $\mathbb{G} : \mathcal{P}(\Sigma_1 \times \Sigma_2) \to \mathcal{P}(\Sigma_1 \times \Sigma_2)$ as follows:

$$\mathbb{G}(X) \triangleq \{(s,t) \in X \mid (s,t) \in \mathbb{F}^{must}(X) \ \wedge\ (s,t) \in \mathbb{F}^{may}(X)\}$$

Since $\mathbb{F}^{must}$ and $\mathbb{F}^{may}$ are monotonic over the complete lattice $(\mathcal{P}(\Sigma_1 \times \Sigma_2), \subseteq)$, the function $\mathbb{G}$ is also monotonic over $(\mathcal{P}(\Sigma_1 \times \Sigma_2), \subseteq)$. The relation $\preceq_{stut}$ is the largest stuttering refinement relation defined over $\Sigma_1 \times \Sigma_2$; thus, $\preceq_{stut}$ is the greatest fixpoint of $\mathbb{G}$. Since the number of states in each M_1 and M_2 is finite, we have:

$$\preceq_{stut} = \mathbb{G}^n(X_0) \text{ for some } n \geq 0$$

1: $P'^{may} = P^{may} = P_0^{may}$

2: $P^{must} = P'^{must} = P_0^{must}$

3: $stable = \bot$

4: **while not** *stable* **do**

5: refine P'^{must}

6: refine P'^{may}

7: $P^{must} =$ `update-P`$'^{must}$`-P`$'^{may}$

8: $P^{may} =$ `update-P`$'^{may}$`-P`$'^{must}$

9: **if** $(P^{must} = P'^{must} \wedge P^{may} = P'^{may})$ **then**

10: $stable = \top$

11: **end if**

12: $P'^{must} = P^{must}$

13: $P'^{may} = P^{may}$

14: **end while**

Figure 4.15: An algorithm for computing $\preceq_{stut}$

where $X_0 = \{(s,t) \mid t \preceq_0 s\}$ [4]. Notice that this time the number of iterations is bounded by $max(|\Sigma_1|, |\Sigma_2|)$. The algorithm for computing $\preceq_{stut}$ is shown in Figure 4.15. It starts with the initial division sets $P_0^{must} = \{B_{s_i} \mid s_i \in \Sigma_1; B_{s_i} \neq \emptyset; \forall t \in \Sigma_2 \cdot t \preceq_0 s_i \Leftrightarrow t \in B_{s_i}\}$ and $P_0^{may} = \{B_{t_i} \mid t_i \in \Sigma_2; B_{t_i} \neq \emptyset; \forall s \in \Sigma_1 \cdot t_i \preceq_0 s \Leftrightarrow s \in B_{t_i}\}$. In each iteration of the while-loop (lines 4-14, Figure 4.15), the operations in the for-loop (lines 4-9) of the algorithm in Figure 4.5 is executed two times: first (refine P'^{must}, on line 5), M_1 (resp. M_2) of the algorithm in Figure 4.5 is assumed to be $M_{1,must}$ (resp. $M_{2,must}$); second (refine P'^{may} on line 6), M_1 (resp. M_2) of the algorithm in Figure 4.5 is assumed to be $M_{2,may}$ (resp. $M_{1,may}$). At the end of each iteration (lines 7-8), the resulting division sets P'^{must} and P'^{may} are updated. The procedure for updating the division sets P'^{must} and

[4] Recall Definition 2.5

P'^{may} is shown in Figure 4.16. The procedure update-P$'^{may}$-P$'^{must}$ in Figure 4.16 scans the states $s_i \in \Sigma_1$ and for every state t_j such that s_i is in block B_{t_j}, it checks to see whether t_j is also present in block B_{s_i} or not. If t_j is not in B_{s_i}, it removes s_i from B_{t_j}. Conversely, the procedure update-P$'^{must}$-P$'^{may}$ scans the states in Σ_2 and removes $t_i \in \Sigma_2$ from block B_{s_j} if t_i is in B_{s_j}, but s_j is not in B_{t_i}.

In the previous section, it was shown that the running time of the algorithm for refining the division sets P'^{must} and P'^{may} is $O(m_1 \times m_2)$. The order of the algorithm for updating the division sets is $O(n_1 \times n_2 \times m_2 + n_2 \times n_1 \times m_1)$. Therefore, the running time of every iteration of the while-loop (lines 4-14) of the algorithm in Figure 4.15 is $O(n_1 \times n_2 \times max(m_1, m_2) + m_1 \times m_2)$. Since the number of iterations of the while-loop is $max(n_1, n_2)$, the running time of the algorithm for computing $\preceq_{stut}$ is

$$O\left(max(n_1, n_2) \times (n_1 \times n_2 \times max(m_1, m_2) + m_1 \times m_2)\right)$$

```
1: proc update-P'^may_-P'^must
2: for s_i ∈ Σ_1 do
3:   if s_i.block-trace[j] then
4:     if not t_j.block-trace[i] then
5:       remove state s_i from block B_{t_j}
6:     end if
7:   end if
8: end for
9: proc update-P'^must_-P'^may
10: for t_i ∈ Σ_2 do
11:   if t_i.block-trace[j] then
12:     if not s_j.block-trace[i] then
13:       remove state t_i from block B_{s_j}
14:     end if
15:   end if
16: end for
```

Figure 4.16: Procedure for updating P'^{may} and P'^{must}

Chapter 5

Conclusions

Partial modeling formalisms have been advocated for specifying software systems at early stages of software development mainly because they can describe the uncertainty and incompleteness involved in initial software specifications. Partial models can also capture the notion of incremental development of software systems. Recently, partial models have been successfully used for deriving small-size abstractions from complete and fully defined models [BG99, BG00]. Since the size of the state-space of a partial model is typically less than that of its corresponding complete model, partial models are more amenable to model checking techniques. For this reason, it is important to have a behavioral relation that can relate a small-size abstract partial model to a fairly large complete model in such a way that the logical properties of the abstract model are preserved in the complete model.

In this thesis, we developed a refinement relation over partial models that is insensitive to stuttering. Comparing to the previously proposed relations for partial models, we believe that our stuttering refinement relation is considerably better for reducing the number of states because it can deal with models that are at different levels of abstraction. Thus, it can map a finite sequence of states to a single state, and more importantly, it still results in the same logical characterization as those given by the existing refinement

relations.

We also proposed algorithms for computing stuttering simulation and stuttering refinement. The idea behind of these two algorithms is essentially the same as that behind the partition refinement algorithm [GV90]. The running time of our algorithm is polynomial in the size of the state space of the input models.

We plan to continue this work in the following directions:

- Obtaining tighter complexity bounds for the proposed algorithm.
- Investigating other possible techniques for computing stuttering simulation and stuttering refinement relations.
- Conducting case studies to demonstrate the efficiency and practicality of our refinement relations.
- Using stuttering refinement checking to speed up model checking techniques.

Bibliography

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Massachusetts, 1974.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BG99] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, LNCS 1633, pages 274–287, Trento, Italy, 1999. Springer.

[BG00] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In C. Palamidessi, editor, *Proceedings of 11th International Conference on Concurrency Theory (CONCUR'00)*, LNCS 1877, pages 168–182, University Park, PA, USA, August 2000. Springer.

[CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology*, January 2003. (Accepted for publication.).

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, February 1994.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technische Universiteit Eindhoven, 1996.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.

[GJ03] P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*, LNCS 2575, pages 206–222. Springer, 2003.

[GV90] J. F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 626–638. Springer-Verlag New York, Inc., 1990.

[HJS01] M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of 10th Euro-*

pean Symposium on Programming (ESOP'01), LNCS 2028, pages 155–169. Springer, 2001.

[HJS02] M. Huth, R. Jagadeesan, and D. Schmidt. A domain equation for refinement of partial systems. Submitted to Mathematical Structures in Computer Science, 2002.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32(1):137–161, 1985.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:334–354, 1983.

[Lam83] L. Lamport. What good is temporal logic? In *Proceedings of Cong. IFIP 83*, pages 657–667, Paris, North Holland, 1983.

[Lar89] K. G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 232–246. Springer, 1989.

[LT88] K.G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press, 1988.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer Verlag, 1992.

[Nam97] K. S. Namjoshi. A simple characterization of stuttering bisimulation. *LNCS 1346*, pages 284–296, 1997.

[NV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.

[PC02] D. O. Păun and M. Chechik. On closure under stuttering. *Formal Aspects of Computing*, 2002. to appear.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, LNCS 224, pages 510–584. Springer Verlag, 1986.

[Var97] M. Y. Vardi. Why is modal logic so robustly decidable? Technical Report TR97-274, 12 1997.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *Proceedings on Theories of concurrency : unification and extension*, pages 278–297. Springer-Verlag New York, Inc., 1990.