"Model transformations and properties" and "Equivalence reductions"

Vesa Luukkala, vesa.luukkala@nokia.com

November 22, 2001

Abstract

This is an essay based on Clarke's and Sclingloff's monograph on Model checking, chapters 4 and 5 (pp. 1670–1689). The chapters deal with transforming a model to a smaller one that preserves certain classes of properties. Furthermore properties are expressed as languages accepted by automatons that can be transformed to the same formalism that is used for the models. The transformations are described as preorder and equivalence relations between the original and transformed models. On the other hand, these relations can be used to define classes of properties that can be expressed using certain logics. This enables measuring the expressive and distinguishing power of these logics.

1 Introduction

This work is done for T-79.298 "Digitaalisten järjestelmien lisensiaattikurssi" autumn 2001 course at Helsinki University of Technology.

The motivation of these chapters is twofold. On one hand it is speeding up the model checking process, that is the process of determining whether a model $\mathcal{M}$ satisfies some property φ ($\mathcal{M} \models \varphi$?), by reducing the size of the model while still preserving the relevant information needed to decide the problem. How much reduction is possible is determined by the property as more expressive properties require more information in the model.

Conversely, reduction relations define not only a class models that honour the relation but also a class of properties that hold in those models. These properties can be expressed using different logics but also as languages that are accepted by certain automatons.

The "Model transformations and properties" part (chapter 4) considers properties as languages that can be translated to to the same formalism as the model. The examined relations are sequence validness and simulation and the corresponding preserved properties range from safety properties to modal box formulas.

The "Equivalence reductions" part (chapter 5) deals with bisimulation and its relation to modal logic, μ**TL** and **MSOL**. The expressability and distinguishing power of logics are defined and finally, model reduction using autobisimulation is presented.

This is written as a companion document to the original text, so numbering of theorems, examples and figures in this document follows the Clarke-Schlingloff document, unless explicitely stated otherwise. To help reading the number of the referenced item in the original text will be denoted in the margin of this text.

2 Model transformations and properties

The previous sections showed that ω-languages can be used to describe both the set of natural models for which linear temporal logic formulas and ω-automatons that accept that language. This section shows that linear temporal logic properties can be expressed using automatons and vice-versa. Logic formulas are more declarative and global way of expressing properties when compared compared to the imperative and local automaton-based description. This connection allows the designer to choose the more fitting formalism.

Furthermore the above dualism allows defining a connection between various logics and various relations on models. This translates easily to practical use: given a model the designer can transform it to a smaller one that is in chosen relation with the original and be sure that all logical properties of the original model that are preserved by the relation are present in the small model.

2.1 Models, Automata and Transition Systems

This section is based on giving a more straightforward connection between logics and automatons by noting that a Kripke model that generates a language can be structurally (thus straightforwardly) translated to a transition system that accepts the same language. This is stated in Lemma 4.1. Here **4.1**
the definition of generation of a word from the Kripke-model requires that

finite generated words are *maximal* to avoid generation of all prefixes to generated words.

Since there is an equivalent Büchi-automaton for **LTL** (Linear Time Logic) formulas, it is possible to reason about both models and linear temporal logic formulas in terms of automatons. One of the two central relations in this section – sequence validness – is defined by language inclusion:

$$\mathcal{M} \models \varphi \text{ iff } L(\mathcal{M}_{\mathcal{A}}) \subseteq L(\mathcal{M}_{\varphi})$$

Here $\mathcal{M}$ is a Kripke-model, φ is an **LTL** formula, $\mathcal{M}_{\mathcal{A}}$ is the automaton that accepts the language generated by $\mathcal{M}$ (as per lemma 4.1.) and $\mathcal{M}_{\varphi}$ is the Büchi-automaton corresponding to φ. Property φ is sequence valid in model $\mathcal{M}$ iff the behaviour of the model is restricted by the behaviour of the property.

The language inclusion problem can be transformed to language emptyness checking: $L(\mathcal{M}_{\mathcal{A}}) \cap \overline{L(\mathcal{M}_{\varphi})} = \{\}$ or $L(\mathcal{M}_{\mathcal{A}} \times \mathcal{M}_{\neg\varphi}) = \{\}$. This is the standard problem of **LTL** model checking, first a Büchi automaton is constructed both for the model $\mathcal{M}$ and the negation of the property $\neg\varphi$, next a product automaton is constructed so that it accepts an infinite word only if both of its components do. In order to ensure that accepting states of both components are certainly always visited in the product automaton, there are essentially two copies of the same product automaton. Once the accepting state of the first automaton is encountered the execution is switched to the second copy and execution is continued on that side until an accepting state is encountered signaling that the execution can again continue on the original copy. The additional labels 0,1 and 2 that are tagged to the states in the product definition in Clarke-Schlingloff allow distinguishing when either no accepting states, accepting states of component 1 or accepting states of component 2 have been encountered. This principle can be adapted for more than one components in a straightforward manner.

There is another interpretation for the sequence-validness: abstraction. φ ($\mathcal{M}_{\mathcal{S}}$) can be considered to be a "specification" and $\mathcal{M}$ ($\mathcal{M}_{\mathcal{I}}$) the "implementation", where specification is a more abstract version of the implementation. Theorem 4.2 states that on one hand if it is possible to establish a sequence- **4.2**
validness relation between two models s.t. $\mathcal{M}_I \models \mathcal{M}_S$ then *all properties* φ (**LTL** formulas) that are sequence-valid in $\mathcal{M}_S$ are also sequence valid in $\mathcal{M}_I$. This means that instead of checking whether $\mathcal{M}_I \models \varphi$ it is possible to transform $\mathcal{M}_I$ to a more abstract (and hopefully smaller) $\mathcal{M}_S$ and check the same property on that model ($\mathcal{M}_S \models \varphi$) and be sure that it holds on the less abstract model.

On the other hand the sequence-validness relation defines a class of properties: precisely those properties that are retained in models that are equivalent with respect to that relation. All properties that are expressable in **LTL** and especially as ω-regular languages are preserved by the sequence-validness relation.

2.2 Safety and Liveness Properties

LTL properties can be classified into two categories: safety and liveness properties. Safety properties state that some event never occurs and liveness properties state that some event will eventually happen. The intuition is that if a safety property is violated, there is a finite string of events that leads to this undesired state or that the violation can be detected by examining past events.

This may be easier to see in the original text if the definition there is rewritten as follows:

- φ is a *safety property*, iff for every natural model $\mathcal{M}$,

$$\mathcal{M} \not\models \varphi \text{ if } \exists i \forall \mathcal{M}' : (\mathcal{M}^{[..i]} \circ \mathcal{M}') \not\models \varphi$$

For natural models $\mathcal{M}^{[..i]}$ is the model consisting of first i points of $\mathcal{M}$, $\mathcal{M} \circ \mathcal{M}'$ is the concatenation of both models. Here the safety property is a property s.t. after certain point in execution ($\mathcal{M}^{[..i]}$) it is not possible reach a situation where the property would hold by any behaviour ($\mathcal{M}'$, including empty behaviour).

Liveness properties are also given a definition, but the neither of the following chapters deals with them aside of mention in theorem 4.3 . In short, the *decomposition theorem* states that any property can be divided into a conjunction of a safety and liveness property. $\top$ is the only property that is both a liveness and safety property. As the rest of the section deals with preservation of safety properties the function of the definition of safety and liveness formulas is to be able to recognize a safety formula. It is possible to give syntactic rules for logic formulas so that a proposition built using these rules is guaranteed to be a safety formula. Theorem 4.4 gives these syntactic rules for **LTL** formulas. Safety properties can also be characterized very naturally by past temporal logic, where all operators concern with events before the current execution time. **4.3** **4.4**

A transition relation is *image finite* if there is a finite amount of successors to every state. A transition system is *finitary* if its transition relation is image finite and the number of initial states is finite. Theorem 4.5 states **4.5**

that any finitary transition system defines a safety property, where finitary transition system covers particularly ω-regular and finite transition systems. If the restriction of image finiteness on automatons would be lifted, it would be possible to express liveness properties. For example, a transition system that has all finite paths starting from the initial state corresponds to liveness property ($\mathbf{F}^*\mathbf{X}\bot$) (see figure 4). Lemma 4.6 gives a weaker inverse statement to theorem 4.5 that says that there are corresponding transition systems for all ω-regular safety properties. fig. 4 **4.6**

Altogether, there are safety properties that can not be represented by finitary transition systems, let alone syntactically constrained **LTL** formulas of theorem 4.4. For **LTL** safety properties there exists a tableau procedure that produces a corresponding $\mathcal{M}_\varphi$ transition system; this is described in section 7.

Model checking for ω-regular properties can be done by executing the property transition system ($\mathcal{M}_\varphi$) and the model transition system ($\mathcal{M}_A$) in parallel and to decide that $L(\mathcal{M}_A) \subseteq L(\mathcal{M}_\varphi)$ every step in $\mathcal{M}_A$ must have a corresponding step in $\mathcal{M}_\varphi$. This maps straightforwardly to the context of testing an implementation, so this approach is also known as *specification based testing.*

Since safety properties are clearly a subset of properties that can be expressed (4.5, 4.6), it is possible to apply theorem 4.2 to safety properties as well, which is done in theorem 4.7 . This means that the sequence-validness relation preserves safety properties in particular, allowing them to be checked from models that have been normalized using the sequence-validness relation. **4.7**

2.3 Simulation Relations

The rest of this section describes a weaker relation than the sequence-validness relation. In presence of nondeterminism, checking of language inclusion may be hard and furthermore as sequence-validness only deals with traces of a system, reasoning about the structure of the system may be needed. The previous section considered only properties of one path throught the model, hence it is named "linear time" world. In this section the structure preserving transformations also preserve all paths through the model, so this is called "branching time" world.

This is demonstrated in figure 1 (corresponds to figure 5 in original text), where the two systems clearly have the same possible traces, but their behaviour is different. Upon start system 2 chooses nondeterministically either a branch; after executing the chosen `a`-transition, it has already chosen whether it can continue with `b` or `c`. This differs from system 1 that can execute a and still be able to perform either `b` or `c`. fig. 5

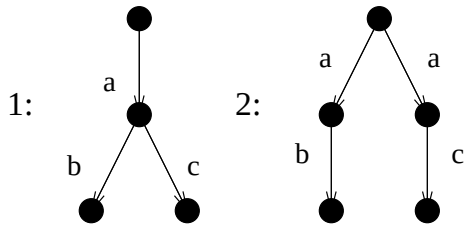

Figure 1: Two models 1 and 2 with different behaviour, yet $1 \models 2$ and $2 \models 1$. For both $\mathcal{P} = \{\}, \mathcal{R} = \{a, b, c\}$.

A straightforward relation is the *submodel* relation: $\mathcal{M}_1$ is a *submodel* of a model $\mathcal{M}_2$ ($\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$) if, intuitively, a set of states (not initial states) is removed from the original leaving a part of the original model and the transition and property relations of the new model are modified so that they don't reference the removed states. A *generated submodel* contains only those states of the model that are reachable from the initial state. All usual temporal properties [1] are preserved in generated submodel.

However, in order to preserve properties it is usually a better idea to combine states rather than delete them. This leads to *simulation relation*, which is a relation between the states in two models. $\mathcal{M}_{abstract}$ simulates $\mathcal{M}_{concrete}$ ($\mathcal{M}_{concrete} \underline{\rightarrow} \mathcal{M}_{abstract}$) when all actions of $\mathcal{M}_{concrete}$ can be matched by $\mathcal{M}_{abstract}$. A state in the abstract model may simulate several states of the concrete model and thus there may be behaviours in the abstract model that are not in the concrete model, but the abstract model is guaranteed to have all behaviours of the concrete model.

- for models $\mathcal{M}_1 = (\mathcal{U}_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (\mathcal{U}_2, \mathcal{I}_2, w_2)$, a relation $H \subseteq U_1 \times U_2$ is a *simulation* between $\mathcal{M}_1$ and $\mathcal{M}_2$ if
  - $(w_1, w_2) \in H$
    initial states must simulate each other
  - $\forall p \in \mathcal{P}, u \in U_1, v \in U_2$ if $(u, v) \in H$ then $u \in \mathcal{I}_1(p)$ iff $v \in \mathcal{I}_2(p)$
    the state properties
  - $\forall u, v : (u, v) \in H$ and for all R, u' s.t. $(u, u') \in \mathcal{I}_1(R)$ there is a v' s.t. $(v, v') \in \mathcal{I}_2(R)$ and $(u', v') \in H$
    simulating system should be able to match any transition of the simulated and have the successor state simulate the successor state of the original model

[1] "Usual" probably meaning those properties that do not refer to unreachable parts of the system.

Simulation relation is a preorder (a reflexive and transitive relation) on the class of all models (4.8). If we require that there is a preorder relation in both directions ($\mathcal{M}_1 \stackrel{\rightarrow}{\preceq} \mathcal{M}_2$ and $\mathcal{M}_2 \stackrel{\rightarrow}{\preceq} \mathcal{M}_1$) then the resulting relation is reflexive as well and is thus an equivalence (in case of simulation the corresponding equivalence is bisimulation, see next section). Two models are in a simulation relation if there exists a simulation relation between the models (4.9). **4.8** **4.9**

To summarize the relation between sequence-validness, submodel and simulation relation: if $\mathcal{M}_{concrete} \sqsubseteq \mathcal{M}_{abstract}$ then $\mathcal{M}_{concrete} \stackrel{\rightarrow}{\preceq} \mathcal{M}_{abstract}$, furthermore, if $\mathcal{M}_{concrete} \stackrel{\rightarrow}{\preceq} \mathcal{M}_{abstract}$ then $\mathcal{M}_{concrete} \models \mathcal{M}_{abstract}$. If the abstract model $\mathcal{M}_{abstract}$ is deterministic (each state has at most one successor), then it is possible to to state that $\mathcal{M}_{concrete} \models \mathcal{M}_{abstract}$ iff $\mathcal{M}_{concrete} \stackrel{\rightarrow}{\preceq} \mathcal{M}_{abstract}$.

A *modal box formula* is a formula without the diamond operator ("eventually"). Rules for constructing modal box formulas are

- literals and $\top, \bot$ are allowed
- if φ, ψ are modal box formulas, then $(\varphi \wedge \psi), (\varphi \vee \psi), [R]\varphi$ are modal box formulas

Lemma 4.10 ties together simulation relation and preservation of modal box formulas (similar as 4.2 and 4.7 did for sequence-validness and LTL properties). Note, that an for existing simulation relation it is possible to reason about the validness of the formula in the model, but not the converse is not possible – see 4.13 for more. **4.10**

Simulation can characterize more expressive logics than modal box formulas, theorem 4.11 gives the same result for **ACTL** formulas. **ACTL** formulas are a subset of **CTL** formulas, so that the **CTL E** quantifier is not used and all temporal operators are prefixed by . This restricts the expressiveness to those temporal properties that are valid in all paths of the model and since simulation preserves all behaviours it is intuitive to see why **ACTL** is preserved. **4.11**

In order to be able give the converse lemma to 4.10 the models must be restricted as in the case of safety properties earlier in the linear time world (theorem 4.5). The need for this is presented in lemma 4.12 which in conjunction with figure 7 presents two systems that can not be distinguished by any modal logic formula (including modal box formulas); that is if any modal logic formula φ is sequence valid in one of the models it must be sequence valid in the other model (and conversely). Yet there are no corresponding simulation relations that should exist if converse of 4.11 would hold. Especially $\mathcal{M}_1 \stackrel{\rightarrow}{\preceq} \mathcal{M}_2$ does not hold, as each successor of a state in $\mathcal{M}_1$ should be **4.12** fig 7

mapped to some state in $\mathcal{M}_2$ and $\mathcal{M}_1$ has an infinite branch, whereas $\mathcal{M}_2$ does not have one.

Restricting the models to be image finite it is possible to give a relation between simulation and modal box formulas (in 4.13) that is a companion to 4.10, but holds in the converse case as well. Again, this means that once $\mathcal{M}_{concrete} \underline{\rightarrow} \mathcal{M}_{abstract}$ is established it is possible to determine $\mathcal{M}_{abstract} \models \varphi$ and be sure that the result is same for $\mathcal{M}_{concrete} \models \varphi$ if φ is a modal box formula. **4.13**

For deterministic finite automata there are efficient algorithms for language inclusion that can be used to check simulation for deterministic models. Below is an outline of an algorithm for creating a simulation relation $H = U_1 \times U_2$ between two nondeterministic finite models:

1. place all pairs of states with matching properties into the first iteration of the relation H^0
2. for the next iteration, place a pair of H^n to H^{n+1} if the model to be simulated has a transition that the simulating model can match – this simulated transition should end to some other pair in H^n

Since this is a finite model, eventually $H^n = H^{n+1}$ and the algorithm will stop and then the intersection of all H^n is the largest simulation relation.

3 Equivalence reductions

As noted before, equivalence is a symmetric preorder and preorders $\preceq$ can induce an equivalences $\simeq$: $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff $\mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$. Submodel ordering $\sqsubseteq$ induces isomorphism and sequence validness $\models$ induces equivalence of the generated languages. Equivalences and especially bisimulation can preserve more expressive logical properties than the preorders that have been presented in previous section. Furthermore, distinguishing power can also be thought of as an equivalence relation.

3.1 Bisimulations (p-morphisms)

Bisimulation $\underline{\leftrightarrow}$ is an equivalence relation between universes of two Kripke-models $(\mathcal{U}_1, \mathcal{I}_1, w_1), (\mathcal{U}_2, \mathcal{I}_2, w_2)$:

- $w_1 \underline{\leftrightarrow} w_2$
- if $u \underline{\leftrightarrow} v$ then $u \in \mathcal{I}_1(p)$ iff $v \in \mathcal{I}_2(p)$

- if $u \underline{\leftrightarrow} v$ and $(u, u') \in \mathcal{I}_1(R)$ then there exists v' s.t. $(v, v') \in \mathcal{I}_2(R)$ and $u' \underline{\leftrightarrow} v'$
- if $u \underline{\leftrightarrow} v$ and $(v, v') \in \mathcal{I}_2(R)$ then there exists u' s.t. $(u, u') \in \mathcal{I}_1(R)$ and $u' \underline{\leftrightarrow} v'$

Two models $\mathcal{M}_1, \mathcal{M}_2$ are bisimilar if there is a bisimulation between them. Figure 9 gives an example of bisimilar models which demonstrate the properties of bisimilar models given in fact 5.1. Figure 2 presents three bisimilar models. For example state $s1$ of A is bismilar to state $s1$ of B: in A it is possible to $s1 \xrightarrow{a} s4$, in B it is possible to $s1 \xrightarrow{a} s4$ and states $s4$ of A and $s4$ of B are again bisimilar. Note that the path $s1 \xrightarrow{a} s2 \xrightarrow{b} s3 \xrightarrow{b} s5 \xrightarrow{a} s7 \xrightarrow{b} s6 \xrightarrow{b} s1$ in model A is simulated by $s1 \xrightarrow{a} s2 \xrightarrow{b} s3 \xrightarrow{b}$ in model B, so bisimulation preserves all paths. Model C is otherwise similar to B, but the two states $s3, s5$ have been merged as bisimilar. Another property of bisimulation is that for bisimilar states the amount of transitions leaving the state doesn't need to be the same, but the set of transition names leaving the states must match.

fig 9

5.1

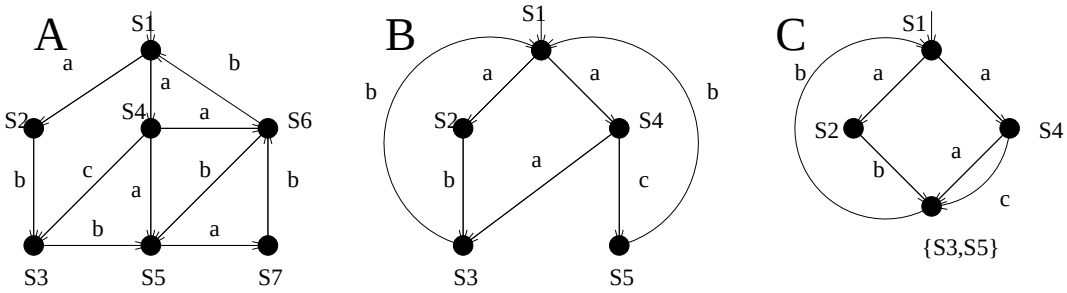

Figure 2: Three bisimilar models. $\mathcal{R} = \{a, b, c\}, \mathcal{P} = \{\}$

The appendix A gives more examples of bisimulation of models in figure 2 and executions of a bisimulation algorithm, which may be useful in understanding bisimulation.

Note that although there exists a simulation relation between two models in both directions, it does not necessarily imply existence of a bisimulation between the models. This can be seen in the two models in figure 10, the difference in bisimulation can be noticed in the additional branch in the system on the right side: bisimulation requires that b transition should be possible after an a transition (see appendix A). In case of simulation the requirement is that the other system should have some path that will execute b after a.

fig 10

Models $\mathcal{M}_1$ and $\mathcal{M}_2$ are are *equivalent with respect to the logic* $\mathbf{L}$ ($\mathcal{M}_1 \equiv_{\mathbf{L}} \mathcal{M}_2$) if for all well formed formulas of $\mathbf{L}$ it holds that $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$. $\equiv_{\mathbf{L}}$ is an equivalence relation. The relation $\equiv_{\mathbf{FOL}}$ for First Order Logic is the elementary equivalence.

Lemma 5.2 gives the relation between bisimulation and modal logic: bisimilar models are modally equivalent. The converse requires image finiteness of the models: image finite models are modally equivalent iff they are bisimilar (theorem 5.3). And again, this enables replacing models with (smaller) bisimilar ones so that all properties that can be expressed using modal logic are preserved. **5.2** **5.3**

If the models are restricted even more it is possible to reason about more expressive logics. Lemma 5.4 gives a connection between sequence-validness and positive $\mu\mathbf{TL}$ (μ calculus) formulas for finite models. Since modal logic is a sublanguage of $\mu\mathbf{TL}$, models that are equivalent w.r.t. monotonic $\mu\mathbf{TL}$ are also modally equivalent. This leads to theorem 5.5 that states that finite models that are bisimilar are also monotonic $\mu\mathbf{TL}$-equivalent. **5.4** **5.5**

Since bisimilarity is used to relate both $\equiv_{\mathbf{ML}}$ and $\equiv_{\mu\mathbf{TL}}$ (5.2, 5.5) it can be said (corollary 5.6) that if it is possible to distinguish finite models using a $\mu\mathbf{TL}$ formula, then it is also possible to distinguish the models using a modal logic formula, even though modal logic is less expressive than μ-calculus. A similar result for equal distinguishing power can be found for $\mathbf{CTL}^*$ and $\mathbf{CTL}$ as $\mathbf{CTL}^*$ can be translated to $\mu\mathbf{TL}$. **5.6**

3.2 Distinguishing Power and Ehrenfeucht-Fraïsse Games

As there are logics that have different expressiveness kbut still have the same distinguishing power, it is useful to define these concepts. Logic **L2** is *at least as expressive as* **L1** iff for any formula $\varphi_1 \in \mathbf{L1}$ there exists a formula $\varphi_2 \in \mathbf{L2}$ s.t. for all models $\mathcal{M}$: $\mathcal{M} \models \varphi_1$ iff $\mathcal{M} \models \varphi_2$ (this could also be phrased **L1** is *at most as expressive as* **L2**). **L1** and **L2** *have the same expressive power* if **L1** is at least as expressive as **L2** and vice versa – for each formula in one logic there is an equivalent one in the second logic.

Logic **L2** is *at least as distinguishing as* **L1** if any two models that are inequivalent w.r.t. **L1** are inequivalent w.r.t. **L2** – or iff $\mathcal{M}_1 \equiv_{\mathbf{L2}} \mathcal{M}_2$ implies $\mathcal{M}_1 \equiv_{\mathbf{L1}} \mathcal{M}_2$. **L1** and **L2** *have the same distinguishing power* if **L1** is at least as distinguishing as **L2** and vice versa – or iff for all models it holds that $\mathcal{M}_1 \equiv_{\mathbf{L2}} \mathcal{M}_2$ iff $\mathcal{M}_1 \equiv_{\mathbf{L1}} \mathcal{M}_2$.

Fact 5.7 states that the expressiveness is a finer equivalence than distinguishability: that is distinguishability considers more logics to be the same when compared to expressiveness. **5.7**

A formula φ is *preserved under bisimulations* if for all models $\mathcal{M}_1 \underline{\leftrightarrow} \mathcal{M}_2$

it holds that $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$. A logic $\mathbf{L}$ is bisimulation invariant, if all well formed formulas of $\mathbf{L}$ are preserved under bisimulations.

As per theorem 5.2 modal logics are bisimulation invariant, but this also holds for monotonic $\mu\mathbf{TL}$ (lemma 5.8). Furthermore bisimulation can be used to define a connection between modal logics and first order logic: only those first order formulas that are preserved under bisimulations can be translated to modal logic (theorem 5.9). This result can be extended to for second order logics and $\mu\mathbf{TL}$ as theorem 5.10. **5.8** **5.9** **5.10**

A convinient way of imagining bisimulations (and equivalences w.r.t. other logics) is Ehrenfeucht-Fraïsse game. The game has two players Ann and Bob, each having an unlimited number of identified pieces: $a_0, a_1, \ldots$ and $b_0, b_1, \ldots$. The game is played on two Kripke-structures, where Ann must show that the models are not bisimilar and Bob must show that they are bisimilar. The game begins so that both place their first pieces on initial states of different models. If the proposition labels on the models are not matching Bob loses and the boards are not bisimilar. The game continues as follows:

- Ann places her $(i+1)$th piece on either of the boards honouring the transition relation w.r.t placed pieces, that is, the piece in the predecessor state of the $(i+1)$th piece must have label a_i or b_i.
- Bob has to match Ann's move on the other board by locating the ith piece on that board and placing the piece honouring the transition relation.
- if Bob can not match Ann's move he loses, if he can play the game forever he wins.

Ann *can force a win within n rounds* if she can place her piece such that Bob loses immediately or after $n-1$ rounds. Ann has a *winning strategy* if there is n s.t. she can force a win – Bob has a winning strategy if Ann does not have one. Theorem 5.11 relates Ann's and Bob's winning strategies to existence of bisimulation relation between the game board models. The rules of the game can be easily modified to capture the notion of equivalence with respect to other logics. **5.11**

3.3 Auto-bisimulations and Paige-Tarjan Algorithm

So far the bisimulation relation has been between two models. However it is possible that a state in a model is bisimilar to another point in the same

model. For a simple example see figure 3: both states can bisimulate the other state.

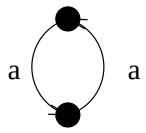

Figure 3: A model with both states bisimilar to each other.

The existence of these *auto-bisimulations* can be exploited for a bisimulation minimization algorithm that searches for states that are auto-bisimilar and combines them. By the definition of bisimulation it can be stated that the union of auto-bisimulations is again an auto-bisimulation (lemma 5.12). **5.12**
For each auto-bisimulation there exists greatest equivalence relation $\equiv$ that includes the auto-bisimulation ($\underline{\leftrightarrow} \subseteq \equiv$) and is also an auto-bisimulation. A *quotient* of a model is another model where the states have been grouped into equivalent classes and the transition relation has been modified accordingly. Theorem 5.13 states that if the equivalence relation $\equiv$ is an auto-bisimulation, **5.13**
then $\mathcal{M} \underline{\leftrightarrow} \mathcal{M}^{\equiv}$.

The quotient of the model with respect to its largest autobisimulation is the minimal representation of the model.

In order to create an algorithm for calculating autobisimulations some technical definitions are needed – below are intuitive explanations for these

- for any set of points $P \subseteq U < R > P$ gives the set of those nodes that have a transition to set P
- given a partition U to equivalence classes, a component $w^{\equiv}$ is *uniform* if the nodes in partition have the same labeling (propositions).
- a component $w^{\equiv}$ is *stable* w.r.t set P if it either is possible to access some states of P from partition $w^{\equiv}$ or it is not possible to access any state of P from $w^{\equiv}$
- a partition is *stable* if all components are uniform and stable w.r.t other partitions

Theorem 5.14 states that the coarsest stable partition (the partition with most states in it) is the largest auto-bisimulation. A sketch for a bisimulation minimization algorithm is below: **5.14**

- To construct the coarsest stable partition:

- start with a trivial partition of one component
- repeat until no new partitions are created and choose nondeterministically
  * 1. choose a component $w_0^{\equiv}$ and a proposition $p \in \mathcal{P}$
    2. split $w_0^{\equiv}$ to two uniform partitions in which the other partition has property p
  * 1. choose components $w_0^{\equiv}, w_1^{\equiv}$ and $R \in \mathcal{R}$
    2. split $w_0^{\equiv}$ to be stable w.r.t. $w_1^{\equiv}$ to those that have a transition to $w_1^{\equiv}$ and to those that have not

In practice the more sophisticated Paige-Tarjan algorithm can compute the same in $\mathcal{O}(m \cdot \log n)$, where n is number of points in model and m is the number of partitions in the result. See appendix for an implementation of the Paige-Tarjan algorithm.

A Additional examples

Once Ocaml code in appendix B has been evaluated in the interpreter it is possible to give other values than those that are in the code. Currently the code contains model A of figure 2. If the code is compiled as such the execution of the algorighm will be annotated:

```
 > ocamlc -o bisim bisim.ml
 > ./bisim | grep =
=== Initial partition:{{s2,s7,}{s1,s4,s5,}{s3,s6,}}
 == C is now:{{s1,s2,s3,s4,s5,s6,s7,}}
 == F is now:{{s2,s7,}{s1,s4,s5,}{s3,s6,}}
 == Refined C to: {{s2,s7,}{s1,s3,s4,s5,s6,}}
  = Refining F:{{s2,s7,}{s1,s4,s5,}{s3,s6,}}
  = F after splitting with a: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with b: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with c: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == Refined F to: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == C is now:{{s2,s7,}{s1,s3,s4,s5,s6,}}
 == F is now:{{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == Refined C to: {{s1,s5,}{s2,s7,}{s3,s4,s6,}}
  = Refining F:{{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with a: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with b: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with c: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
```

```
 == Refined F to: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == C is now:{{s1,s5,}{s2,s7,}{s3,s4,s6,}}
 == F is now:{{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == Refined C to: {{s1,s5,}{s2,s7,}{s3,s6,}{s4,}}
  = Refining F:{{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with a: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with b: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
  = F after splitting with c: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == Refined F to: {{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
 == C is now:{{s1,s5,}{s2,s7,}{s3,s6,}{s4,}}
 == F is now:{{s1,s5,}{s2,s7,}{s4,}{s3,s6,}}
```

The final C and F both contain the largest autobisimulation state partitions. If the `grep` is omitted even more diagnostic of the algorithm execution can be seen.

In order to enter other models perhaps the easiest way is to use the interpreter create new initial partitions. Once the program has been interpreted it is possible to give the following commands to enter model B of figure 2:

```
(* We need only 5 states *)
# let u_b = add_states 5;;
val u_b : States.t = <abstr>
(* The transition relation encoding the model *)
# let b_trans = add_trans_rel [
("a",("s1","s2")); ("a",("s1","s4")); ("b",("s2","s3"));
("a",("s4","s3")); ("c",("s4","s5")); ("b",("s3","s1"));
("b",("s5","s1"));];;
val a_trans : Trans_rel.t = <abstr>
(* The initial partitions *)
# let b_init_coarse = Partition.add u_b (Partition.empty);;
val b_init_coarse : Partition.t = <abstr>
# let b_init_f = Partition.add u_b (Partition.empty);;
val b_init_f : Partition.t = <abstr>
(* Now the algorithm can be executed: *)
# bisim b_init_coarse b_init_f r b_init_coarse b_trans;;
...
 == F is now:{{s1,}{s2,}{s3,s5,}{s4,}}
- : Partition.t = <abstr>
```

Here are the definitions for the systems in original papers figure 10:

```
let r2 = Transitions.add "a" (Transitions.add "b" (Transitions.empty));;
```

```
let u_left = add_states 4;;
let u_right = add_states 5;;
let left_trans = add_trans_rel [
("a",("s1","s2")); ("a",("s2","s3"));("b",("s2","s4"));];;
let right_trans = add_trans_rel [
("a",("s1","s2")); ("a",("s2","s3"));("b",("s2","s4"));
("a",("s1","s5")); ("a",("s5","s4"));];;
let left_i_c = Partition.add u_left (Partition.empty);;
let right_i_c = Partition.add u_right (Partition.empty);;
let left_i_f = Partition.add u_left (Partition.empty);;
let right_i_f = Partition.add u_right (Partition.empty);;
# bisim left_i_c left_i_f r2 left_i_c left_trans;;
 == F is now:{{s1,}{s2,}{s3,s4,}}
# bisim right_i_c right_i_f r2 right_i_c right_trans;;
 == F is now:{{s1,}{s2,}{s3,s4,}{s5,}}
```

B Bisimulation implementation

This section gives Ocaml source code for implementation of the Paige-Tarjan bisimulation algorithm and example execution of the algorithm. This may be helpful in understanding both the bisimulation relation and the execution of the algorithm.

```
(*
 * To calculate strong autobisimulation relation
 * using Clarke-Schlinghoff presentation of
 * the Paige-Tarjan algorithm (p. 1690).
 *
 * Does not handle state properties and only
 * produces the smallest state partition,
 * also the transition relation is not
 * modified.
 *
 * This is in Caml, an ML dialect.
 * Compiler & environment available at
 * http://www.ocaml.org
 *
 * To compile: ocamlc -o bisim bisim.ml
 *)
```

```
open Set;;
open Printf;;

module Transitions = Set.Make ( struct
  type t = string
  let compare = compare
end)

module States = Set.Make ( struct
  type t = string
  let compare = compare
end)

(* ("trans_name",(start_state,target_state)) *)
module Trans_rel = Set.Make ( struct
  type t = (string * ( string * string))
  let compare = compare
end)
module Partition = Set.Make ( struct
type t = States.t
let compare = compare
      end)

(* Helper function for entering state names *)
let rec add_states n  =
  if n = 0 then
    States.empty
  else
    States.add ("s" ^ (string_of_int n)) (add_states (n - 1));;

(* Helper function that creates a set out of a list *)
let rec add_trans_rel list  =
  match list with
      [] -> Trans_rel.empty
    | x::the_rest ->
Trans_rel.add x (add_trans_rel the_rest);;

let print_set set  =
    let print_item item =
      print_string (item ^ ",") in
      print_string "{";
```

```
      States.iter print_item set;
      print_string "}";;

let print_partition partition =
  print_string "{";
  Partition.iter print_set partition;
  print_string "}";;

(*
 * Take in a set of transition relations
 * 'trans_rel' (elements of form ("a",("s1","s2")))
 * and return a set of states that contains
 * the starting states of th4e transitions
 * (for the example item above it would be 's1').
 *)
let rec starting_state_to_set trans_rel =
  (* returns the source target state of 'item'  *)
  let starting item =
    (fst (snd item)) in
    if Trans_rel.is_empty trans_rel then
      States.empty
    else
      let item = Trans_rel.choose trans_rel in
States.add (starting item)
  (starting_state_to_set (Trans_rel.remove item trans_rel));;

(*
 * The set of 'trans predecessors',
 * those states that can take transition
 * 'trans' to reach 'set' according to
 * 'relation'.
 *
 * Corresponds to <R>U in the text.
 *
 * prec "a" u trans;;
 * prec "a" (Partition.choose c) trans;;
 *)
let prec trans_name set relation =
  (* true if the tag of 'item' is equal 'trans' *)
  let trans_matches item =
    trans_name = (fst item) in
```

```
  (* true if the target state of 'item' is in 'set' *)
  let target_in item =
    (States.mem (snd (snd item)) set) in
    starting_state_to_set
      (Trans_rel.filter target_in
 (Trans_rel.filter trans_matches relation));;

(*
 * initial_partition r u trans;;
 *)
let initial_partition transitions set relation =
  let rec initial_iter result trans_names =
    match trans_names with
[] -> result
      | x::the_rest ->
  let temp = prec x set relation in
    initial_iter (temp::result) the_rest in
    initial_iter [] transitions;;

(*
 * Split each set of a partition w.r.t some
 * transition and returns a partition with
 * the splitted sets.
 * Note that here we remove the empty set
 * from the refined partition after each refinition --
 * the algorithm does not mention that case and
 * it probably does not matter, but it looks
 * cleaner that way.
 *)
let rec split_all_sets trans partition relations results =
  if Partition.is_empty partition then
    results
  else
    let one_partition = Partition.choose partition in
      ignore(printf " partitioning with %s " trans);
      print_set one_partition;
      let pred = prec trans one_partition relations in
      let new_1 = States.inter one_partition pred
      and new_2 = States.diff one_partition pred in
ignore(printf " to ");
print_set new_1;
```

```
print_set new_2; print_newline ();
let results = Partition.add new_2 (Partition.add new_1 results) in
  split_all_sets trans (Partition.remove one_partition partition)
    relations
  (Partition.remove States.empty results);;

(*
 * Performs the initial partitioning.
 *
 *)
let rec initial_partition transitions init_partitions relations =
  if Transitions.is_empty transitions then
    init_partitions
  else
    let trans = Transitions.choose transitions in
      let new_partition = split_all_sets trans init_partitions relations
    Partition.empty in
initial_partition (Transitions.remove trans transitions)
  new_partition relations;;

let rec choose_from_fine bigger partition =
    ignore(printf "     try choosing one from F:");
    print_partition partition; print_newline();
  if Partition.is_empty partition then
    States.empty
  else
    let candidate = Partition.choose partition in
      if States.subset candidate bigger then
begin
  ignore(printf "     chose ");
  print_set candidate; print_newline ();
  candidate;
end
      else
choose_from_fine bigger (Partition.remove candidate partition);;

(*
 * Refinement for the coarse partition.
 * This corresponds to choosing w and w1
 * in the algorithm. Returns both w and
 * w1.
```

```
 *)
let rec choose_from_coarse coarse fine =
  let filtered_coarse = Partition.diff coarse fine in
    ignore(printf "     try choosing one from C:");
    print_partition filtered_coarse; print_newline();
    if Partition.is_empty filtered_coarse then
      (States.empty, States.empty)
    else
      let candidate_in_coarse = Partition.choose filtered_coarse in
ignore (printf "     chose "); print_set candidate_in_coarse;
print_newline ();
let candidate_in_fine = choose_from_fine candidate_in_coarse fine in
  if States.is_empty candidate_in_fine then
    choose_from_coarse (Partition.remove candidate_in_coarse coarse) fine
  else
    (candidate_in_coarse, candidate_in_fine);;

(*
 * Splits the w0 partition to four components
 *)
let split_into_four trans w0 w1 w2 trans_rel =
  let part_w1 = prec trans w1 trans_rel
  and  part_w2 = prec trans w2 trans_rel in
  let one = States.inter (States.inter w0 part_w1) part_w2
  and two = States.diff (States.inter w0 part_w1) part_w2
  and three = States.inter (States.diff w0 part_w1) part_w2
  and four = States.diff (States.diff w0 part_w1) part_w2 in
    (one, two, three, four);;

(*
 * Split all the sets in 'f' w.r.t trans using w1 and w2
 *)
let rec four_split_all_sets trans f trans_rel w1 w2 result =
  if Partition.is_empty f then
    result
  else
    let w0 = Partition.choose f in
      ignore(printf "        now splitting ");
      print_set w0; print_newline();
    let (one, two, three, four) = split_into_four trans w0 w1 w2 trans_rel
    in
```

```
      ignore(printf "          one:   "); print_set one;
      print_newline();
      ignore(printf "          two:   "); print_set two;
      print_newline();
      ignore(printf "          three: "); print_set three;
      print_newline();
      ignore(printf "          four:  "); print_set four;
      print_newline();
    let new_F = Partition.remove w0 result in
    let new_F = Partition.add four (Partition.add three
(Partition.add two
   (Partition.add one new_F)))
    in
      (* ignore(printf "   F now:\n"); print_partition new_F; *)
      if Partition.mem States.empty new_F then
four_split_all_sets
  trans (Partition.remove w0 f) trans_rel w1 w2
  (Partition.remove States.empty new_F)
      else
result;;

(*
 * The loop for the
 * for all R \in \mathcal{R} and w_0 \in F do
 * part of the algorithm.
 *)
let rec main_partition transitions old_partition relations w1 w2 =
  if Transitions.is_empty transitions then
    old_partition
  else
    (* For all transitions \in r *)
    let trans = Transitions.choose transitions in
      (* For all w0 \in F with transition trans *)
      ignore(printf "       splitting w.r.t %s\n" trans);
      let new_partition = four_split_all_sets trans old_partition relations
    w1 w2 Partition.empty in
ignore(printf "  = F after splitting with %s: " trans);
print_partition new_partition; print_newline();
main_partition (Transitions.remove trans transitions)
  new_partition relations w1 w2;;
```

```
(*
 * Simulates Partition.remove.
 * A stupid auxiliary function that should be done
 * at the module level. My ML ineptitude. It seems
 * that Partition.remove does not find the set
 * unless it is the first one in the set. Must be
 * related to non-working of Partition.mem
 *)
let rec rem_set_from_partition set partition result =
  if Partition.is_empty partition then
    result
  else
    let from_p = Partition.choose partition in
      if States.equal set from_p then
rem_set_from_partition set (Partition.remove from_p partition) result
      else
rem_set_from_partition set
  (Partition.remove from_p partition)
  (Partition.add from_p result);;

(*
 * Simulates Partition.mem
 * A stupid auxiliary function that should be done
 * at the module level. My ML ineptitude.  It seems
 * that Partition.remove does not find the set
 * unless it is the first one in the set.
 *)
let rec set_in_partition set partition =
  if Partition.is_empty partition then
    false
  else
    let from_p = Partition.choose partition in
      if States.equal set from_p then
true
      else
set_in_partition set
  (rem_set_from_partition from_p partition (Partition.empty));;

(*
 * Simulates Partition.equal
 * See above for comments on 'rem_set_from_partition'
```

```
 * and 'set_in_partition set partition'.
 * Compare two partitions, return true if
 * both partitions have the same sets.
 * This really should be given to the
 * module as an the compare function.
 *)
let rec compare_partitions a b =
  if Partition.is_empty a then
    if Partition.is_empty b then
      (* If both are empty we are done *)
      true
    else
      false
  else
    let from_a = Partition.choose a in
      if Partition.is_empty b then
false
      else
if set_in_partition from_a b then
  begin
    (*
     * Both partitions have the same item,
     * remove the item from both and continue
     *)
    compare_partitions
      (rem_set_from_partition from_a a Partition.empty)
      (rem_set_from_partition from_a b Partition.empty)
  end
else
  false;;

(*
 * The main loop of the whole algorithm
 * while C \neq F do
 *)
let rec refine c f transition_rel transition_names =
  ignore(printf " == C is now:"); print_partition c; print_newline();
  ignore(printf " == F is now:"); print_partition f; print_newline();
  if (compare_partitions c f) then
    f
  else
```

```
      let (w, w1) = choose_from_coarse c f in
      let w2 = States.diff w w1 in
      let refined_C = Partition.remove w c in
      let refined_C = Partition.add w2 (Partition.add w1 refined_C) in
ignore(printf " == Refined C to: " );
print_partition refined_C; print_newline();
ignore(printf "  = Refining F:" );
print_partition f; print_newline();
let refined_F = main_partition transition_names
  f transition_rel w1 w2 in
ignore(printf " == Refined F to: " );
print_partition refined_F; print_newline ();
  refine refined_C refined_F transition_rel transition_names;;

(*
 * bisim c f r c trans;;
 *)
let bisim c f transition_names all_states transition_rel =
  let initial_F = initial_partition
    transition_names all_states transition_rel in
    ignore(printf "=== Initial partition:" );
    print_partition initial_F; print_newline();
    refine c initial_F transition_rel transition_names;;

(* Below are examples of use *)

(* Transition names *)
let r = Transitions.add "a"
  (Transitions.add "b" (Transitions.add "c" (Transitions.empty)));;

(* State names from 1 to 7 *)
let u = add_states 7;;

(* The transition relation *)
let a_trans = add_trans_rel [
  ("a",("s1","s2")); ("a",("s1","s4")); ("b",("s2","s3"));
  ("b",("s3","s5")); ("a",("s5","s4")); ("a",("s5","s7"));
  ("b",("s7","s6")); ("c",("s4","s3")); ("a",("s4","s6"));
  ("b",("s6","s1"))];;

(* The initial partitions *)
```

```
let a_init_coarse = Partition.add u (Partition.empty);;
let a_init_f = Partition.add u (Partition.empty);;

(*
 * The initial values
# States.elements (List.hd (Partition.elements a_init_coarse));;
- : States.elt list = ["s1"; "s2"; "s3"; "s4"; "s5"; "s6"; "s7"]
 *)

(*
 * Now its possible to say:
 * bisim a_init_coarse a_init_f r a_init_coarse a_trans;;
 *)

(*
 * Note that the call to bisim is hardwired here -
 * the values must be changed in code in order
 * for the compiled code to have other behaviour.
 *)
if !Sys.interactive
then ()
else ignore(bisim a_init_coarse a_init_f r a_init_coarse a_trans);;
```