Algebraic Simulations*

José Meseguer[1], Miguel Palomino[2], and Narciso Martí-Oliet[2]

[1] Computer Science Department, University of Illinois at Urbana-Champaign
[2] Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid
meseguer@cs.uiuc.edu {miguelpt,narciso}@sip.ucm.es

Abstract. Rewriting logic is a flexible and general logic to specify concurrent systems. To prove properties about concurrent systems in temporal logic, it is very useful to use *simulations* that relate the transitions and atomic predicates of a system to those of a potentially much simpler one; then, if the simpler system satisfies a property φ in a suitable temporal logic we are guaranteed that the more complex system does too. In this paper, the suitability of rewriting logic as a formal framework not only to specify concurrent systems but also to specify simulations is explored in depth. For this, increasingly more general notions of simulation (allowing stuttering) are first defined for Kripke structures, and suitable temporal logics allowing properties to be reflected back by such simulations are characterized. The paper then proves various *representability results* à la Bergstra and Tucker, showing that recursive Kripke structures and recursive simulation maps (resp. r.e. simulation relations) can always be specified in a finitary way in rewriting logic. Using simulations typically requires both model checking and theorem proving, since their correctness requires discharging proof obligations. In this regard, rewriting logic, by containing equational logic as a sublogic and having equationally-based inductive theorem proving at its disposal, is shown to be particularly well-suited for verifying the correctness of simulations.

Keywords: rewriting logic, stuttering simulations, representability results, Kripke structures, model checking.

1 Introduction

A consequence of the extraordinary growth of computer science witnessed in the last decades and of its ever greater ubiquity and influence in the daily lives of people, has been the design and implementation of increasingly complex systems to deal with the most diverse tasks. Very often their right functioning is critical because human lives depend on it (e.g. flying control systems in airplanes or maintenance systems in nuclear power plants) and, even when that is not the case, the consequences of bugs can still be catastrophic in economic terms (banking and financial systems), or can severely damage the privacy and security of individuals and institutions.

When the complexity of a software or hardware project reaches a certain point, ensuring the absence of errors becomes unfeasible; nonetheless, this should not serve as an excuse not to try to reduce its number as much as possible. In this regard, the use of *specification languages* is especially interesting, since they are flexible and powerful formalisms that can handle highly complex designs and are endowed with a precise mathematical meaning. Ideally, such languages should impose some methodology so that the number of errors would decrease

* Research supported by ONR Grant N00014-02-1-0715, NSF Grant CCR-0234524, by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130, by the Spanish projects MIDAS TIC2003-01000 and DESAFIOS TIN2006-15660-C02-01, and by Comunidad de Madrid program PROMESAS S-0505/TIC/0407.

already during the design and coding phases, while having a precise mathematical semantics should serve to help guarantee that the properties we are interested in do hold, even if we cannot assure that the system is free from other errors.

One such formalism is *rewriting logic* [33], proposed as a unifying specification logic for concurrent systems. Since its formulation in the early 90's, this logic has proved to be a very flexible formalism, not only for concurrency but also as a logical and semantic framework in which to interpret other logics and computation models [29], giving rise to a vast literature (see references in [30]). In addition, rewriting logic is the basis of the Maude declarative specification and programming language [12, 11].

For the verification of properties in rewrite theories, Maude includes a *model checker* [20] for linear temporal logic. Model checking, independently proposed in [7, 45], has proved to be one of the most successful stories in the application of formal methods to software verification in industry. But despite its success, it has a fundamental limitation: the most common model checking algorithms assume that the system under study is finite. The introduction of symbolic model checking [32] increased the size of the state spaces amenable to this technique to impressive limits, of the order of 10^{120} states. Such limits, however, are still insufficient for many industrial applications and are not directly usable for the analysis of software systems when they have an infinite number of states.

The intrinsic limitation of model checkers to deal with infinite (or just too large) systems has led many researchers to study *abstraction* techniques to overcome it. These techniques (e.g. [8, 25, 22, 27, 17]) reduce the verification of a given property in a certain system to its study in an abstract and finite version of the system.

Simulations, introduced by Park [44], are a very natural way of comparing state-based systems and are heavily used for process algebra [49] and automata [26]. In our setting, the concept of simulation generalizes the notion of abstraction. The goal of simulations is not restricted to associating finite state systems to more complex infinite state systems; it also includes relations like the simulation of an implementation of a state-based system by its specification (both possibly infinite), that shows that the implementation is correct, or the bisimulation between semantically equivalent specifications, that allows the transfer of results from one to the other; this makes simulations a flexible and very useful tool.

As a result of the comments above, it follows that a fruitful approach to the study of state-based systems consists in their mathematical formalization by means of models like Kripke structures, which allows the study of their associated properties by means of simulations that relate them to other, possibly better-known systems [5, 8, 25]. This work tries to advance two main goals along those lines: the first, to generalize the notion of simulation between Kripke structures as much as possible, and the second, to provide general representability results showing that Kripke structures and generalized simulations can be represented in rewriting logic. These two goals are themselves motivated by pragmatic reasons. The reason for trying to advance the first goal is that simulations are essential for *compositional reasoning*. A cornerstone in such reasoning is the result that simulations *reflect* temporal logic properties, that is, if we have a simulation of Kripke structures $H : \mathcal{A} \longrightarrow \mathcal{B}$ and a suitable temporal logic formula φ, then if aHb and $\mathcal{B}, b \models \varphi$, we can conclude that $\mathcal{A}, a \models \varphi$. Since this result is very powerful, there are strong reasons to generalize it: a more general notion of simulation will give it a wider applicability, even when the class of formulas φ for which it applies may have to be restricted.

Advancing the second goal is also motivated by pragmatic reasons, namely: (i) executability, (ii) ease of specification, and (iii) ease of proof. The point about (i) and (ii) is that rewriting logic is a very flexible framework, so that concurrent systems can usually be specified quite easily and at a very high level; furthermore, such specifications can be used directly to execute a system, or to reason about it, which is point (iii). Indeed, both rewriting logic and its under-

lying equational logic can be very useful for formal reasoning, since often one needs to reason beyond the propositional level. For example, even when we use a model checker to prove that an infinite state system satisfies $\mathcal{A}, a \models \varphi$ by constructing a finite state abstraction simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ and model checking that $\mathcal{B}, b \models \varphi$ for some b such that aHb, we are still left with verifying the *correctness* of H, which requires discharging proof obligations. More generally, any temporal logic deductive reasoning needs to include first-order and often inductive reasoning at the level of state predicates. This is precisely where rewriting and equational logics and their initial models supporting inductive reasoning are quite useful. In a previous paper [38, 39] we have shown the usefulness of defining abstraction simulations equationally in rewriting logic, and of using tools such as Maude's LTL model checker [20, 12] and inductive theorem prover [13] to verify properties and prove abstractions correct. The conference paper [31] further generalized [38, 39] by allowing not just the addition of equations E' to a theory (Σ, E) for abstraction purposes, thus obtaining a subtheory inclusion $(\Sigma, E) \subseteq (\Sigma, E \cup E')$, but also the use of very general theory morphisms $H : (\Sigma, E) \longrightarrow (\Sigma', E')$. This work substantially widens the results in [38, 39] and [31] and provides computability foundations for the entire approach.

We advance the first goal by generalizing simulations in three directions. First, we consider *stuttering simulations* in the sense of [5, 40, 27], which are quite general and useful to relate concurrent systems with different levels of atomicity; second, we relax the condition on preservation of atomic propositions from equality to containment; and third, we allow different alphabets AP and AP' of atomic propositions in Kripke structures $\mathcal{A}$ and $\mathcal{B}$ related by generalized stuttering simulations $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$, so that an atomic proposition $p \in AP$ is mapped by α to a *state formula* over AP'. We advance the second goal by proving several *representability results* showing that any Kripke structure (resp. any recursive Kripke structure) can be represented by a rewrite theory (resp. a recursive rewrite theory), and that any generalized simulation (resp. recursively enumerable generalized simulation) can be represented by a rewrite relation.

A categorical viewpoint is indeed the most natural to understand these generalized simulations, but as far as we know this viewpoint has not been systematically exploited before. In the conference paper [43] we treated some of these categorical aspects at the level of Kripke structures, including a classification in terms of institutions. This paper further expands some of those aspects but, while presenting also new categorical ideas and results beyond those in [43], it does not try to cover all the topics in [43].

2 Relating Kripke Structures

In this section we start by reviewing standard material on Kripke structures and temporal logic, and by recalling the notion of simulation presented in [38]. After that, we study how these ingredients can be organized in terms of categories, and how the notion of simulation can be generalized.

2.1 Transition Systems, Kripke Structures, and Temporal Logic

When reasoning about computational systems, it is convenient to abstract from as many details as possible by means of simple mathematical models that can be used to reason about them. For state-based systems we can, as a first step, represent its behavior by means of a transition system.

Definition 1. *A* transition system *is a pair $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$, where A is a set of states and $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a binary relation called the transition relation.*

A transition system, however, does not include any information about the relevant properties of the system. In order to reason about such properties it is necessary to add information about the atomic properties that hold in each state. Such atomic properties can be described by a set AP of atomic propositions.

Definition 2. *A* Kripke structure *is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, where $(A, \rightarrow_{\mathcal{A}})$ is a transition system with $\rightarrow_{\mathcal{A}}$ a* total *relation, and $L_{\mathcal{A}} : A \rightarrow \mathcal{P}(AP)$ is a labeling function associating to each state the set of atomic propositions that hold in it.*

We use the notation $a \rightarrow_{\mathcal{A}} b$ to state that $(a, b) \in \rightarrow_{\mathcal{A}}$. Note that the transition relation of a Kripke structure must be *total*, that is, for each $a \in A$ there is a $b \in A$ such that $a \rightarrow_{\mathcal{A}} b$. This is a usual requirement [9] that simplifies the definition of the semantics for temporal logics, of which Kripke structures are models. Given an arbitrary relation $\rightarrow$, we write $\rightarrow^{\bullet}$ for the total relation that extends $\rightarrow$ by adding a pair $a \rightarrow^{\bullet} a$ for each a such that there is no b with $a \rightarrow b$. A *path* in $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ such that, for each $i \in \mathbb{N}$, $\pi(i) \rightarrow_{\mathcal{A}} \pi(i+1)$. Given $n \in \mathbb{N}$, we use π^n to refer to the suffix of π starting at $\pi(n)$; explicitly, $\pi^n(i) = \pi(n+i)$, for each $i \in \mathbb{N}$.

For example, the behavior of a simple periodic system could be represented by means of a transition system with three states, s_0, s_1, and s_2, and transitions $s_i \rightarrow s_{rem(i+1,3)}$. Now, to distinguish among the different states and to reason about the system, this transition system can be extended to a Kripke structure by making explicit some atomic properties satisfied by the states, say $L(s_0) = \{\texttt{sleeping}\}$, $L(s_1) = \{\texttt{waiting}\}$, and $L(s_2) = \{\texttt{working}\}$. Note that the relevant properties may vary based on the interest at hand; thus, a less precise alternative would be $L(s_0) = L(s_1) = \{\texttt{off}\}$ and $L(s_2) = \{\texttt{on}\}$.

To specify system properties we will use the logic ACTL$^*(AP)$, which is the universally (path) quantified sublogic of the branching-time temporal logic CTL$^*(AP)$ (see for example [9, Section 3.1]). These logics are interpreted in a standard way in Kripke structures.

There are two types of formulas in CTL$^*(AP)$: state formulas, denoted by State(AP), and path formulas, denoted by Path(AP). Their syntax is given by the following mutually recursive definitions:

$$\begin{array}{ll} \text{state formulas:} & \varphi ::= p \in AP \mid \top \mid \bot \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\psi \mid \mathbf{E}\psi \\ \text{path formulas:} & \psi ::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi\,. \end{array}$$

A and **E** are respectively the universal and the existential path quantifiers, while the operators **X**, **U**, **R**, **G**, and **F** have the intuitive meanings of *next*, *until*, *release*, *henceforth*, and *eventually*. The semantics of the logic, specifying the satisfaction relations $\mathcal{A}, a \models \varphi$ and $\mathcal{A}, \pi \models \psi$ for a Kripke structure $\mathcal{A}$, an initial state $a \in A$, a state formula φ, a path π, and a path formula ψ is defined by structural induction on formulas as shown in Figure 1.

ACTL$^*(AP)$ is the restriction of CTL$^*(AP)$ to those formulas such that their negation-normal forms (with negations pushed to atoms) do not contain any existential path quantifiers. Sometimes, to avoid introducing implicitly existential quantifiers, it is more convenient to restrict ourselves to the negation-free fragment ACTL$^*\backslash\neg(AP)$ of ACTL$^*(AP)$, defined as follows:

$$\begin{array}{ll} \text{state formulas:} & \varphi ::= p \in AP \mid \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\psi \\ \text{path formulas:} & \psi ::= \varphi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi\,. \end{array}$$

We write State$\backslash\neg(AP)$ and Path$\backslash\neg(AP)$, respectively, for the sets of state and path formulas in ACTL$^*\backslash\neg(AP)$. We also write ACTL$^* \setminus \mathbf{X}(AP)$ for the fragment of the logic that does not contain the *next* operator. Note that in a very practical sense there is no real loss of generality by restricting ourselves to formulas in ACTL$^*\backslash\neg$, because we can always transform any ACTL* formula into a semantically equivalent ACTL$^*\backslash\neg$ one just by introducing new atomic propositions for the negation of the original ones.

$$
\begin{array}{lcl}
\mathcal{A}, a \models p & \iff & p \in L_{\mathcal{A}}(a) \\
\mathcal{A}, a \models \top & \iff & \textit{true} \\
\mathcal{A}, a \models \bot & \iff & \textit{false} \\
\mathcal{A}, a \models \neg\varphi & \iff & \mathcal{A}, a \not\models \varphi \\
\mathcal{A}, a \models \varphi_1 \vee \varphi_2 & \iff & \mathcal{A}, a \models \varphi_1 \text{ or } \mathcal{A}, a \models \varphi_2 \\
\mathcal{A}, a \models \varphi_1 \wedge \varphi_2 & \iff & \mathcal{A}, a \models \varphi_1 \text{ and } \mathcal{A}, a \models \varphi_2 \\
\mathcal{A}, a \models \mathbf{A}\psi & \iff & \text{for all } \pi \text{ such that } \pi(0) = a,\ \mathcal{A}, \pi \models \psi \\
\mathcal{A}, a \models \mathbf{E}\psi & \iff & \text{there exists } \pi \text{ with } \pi(0) = a \text{ such that } \mathcal{A}, \pi \models \psi \\
\mathcal{A}, \pi \models \varphi & \iff & \mathcal{A}, \pi(0) \models \varphi \\
\mathcal{A}, \pi \models \neg\psi & \iff & \mathcal{A}, \pi \not\models \psi \\
\mathcal{A}, \pi \models \psi_1 \vee \psi_2 & \iff & \mathcal{A}, \pi \models \psi_1 \text{ or } \mathcal{A}, \pi \models \psi_2 \\
\mathcal{A}, \pi \models \psi_1 \wedge \psi_2 & \iff & \mathcal{A}, \pi \models \psi_1 \text{ and } \mathcal{A}, \pi \models \psi_2 \\
\mathcal{A}, \pi \models \mathbf{X}\psi & \iff & \mathcal{A}, \pi^1 \models \psi \\
\mathcal{A}, \pi \models \psi_1 \mathbf{U} \psi_2 & \iff & \text{there exists } n \in \mathbb{N} \text{ such that } \mathcal{A}, \pi^n \models \psi_2 \text{ and, for all } m < n, \\
 & & \text{it holds that } \mathcal{A}, \pi^m \models \psi_1 \\
\mathcal{A}, \pi \models \psi_1 \mathbf{R} \psi_2 & \iff & \text{for all } n \in \mathbb{N}, \text{ either } \mathcal{A}, \pi^n \models \psi_2 \\
 & & \text{or there exists } m < n \text{ such that } \mathcal{A}, \pi^m \models \psi_1 \\
\mathcal{A}, \pi \models \mathbf{G}\psi & \iff & \text{for all } n \in \mathbb{N} \text{ it is } \mathcal{A}, \pi^n \models \psi \\
\mathcal{A}, \pi \models \mathbf{F}\psi & \iff & \text{there exists } n \in \mathbb{N} \text{ such that } \mathcal{A}, \pi^n \models \psi
\end{array}
$$

Fig. 1. CTL* semantics.

2.2 Simulations

We present a notion of simulation similar to that in [9], but somewhat more general (simulations in [9] essentially correspond to our *strict* simulations). First, we define simulations between transition systems.

Definition 3. *Given transition systems $\mathcal{A} = (\mathcal{A}, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (\mathcal{B}, \rightarrow_{\mathcal{B}})$, a* simulation of transition systems *$H : \mathcal{A} \longrightarrow \mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that if $a \rightarrow_{\mathcal{A}} a'$ and aHb then there is b' such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$.*

We say that H is a total simulation *if the relation H is total. A* map of transition systems *H is a total simulation such that H is a function.*[3] *If both H and H^{-1} are simulations, then we call H a* bisimulation*.*

We can extend a simulation of transition systems H to paths by defining $\pi H \rho$ if $\pi(i) H \rho(i)$ for each $i \in \mathbb{N}$.

Definition 4. *Given Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$, both over the same set AP of atomic propositions, an AP-*simulation *$H : \mathcal{A} \longrightarrow \mathcal{B}$ of $\mathcal{A}$ by $\mathcal{B}$ is given by a simulation $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ between the underlying transition systems such that if aHb, then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.*

*We say that H is an AP-*map *if its underlying simulation of transition systems is a map. We call H an AP-*bisimulation *if H and H^{-1} are AP-simulations. Also, we call H* strict *if aHb implies $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$. Note that an AP-bisimulation is necessarily strict.*

The fact that $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a simulation of transition systems guarantees that for each concrete path in $\mathcal{A}$ starting at a state related to one in $\mathcal{B}$ there is a path simulating it in $\mathcal{B}$. The second condition implies that a state in $\mathcal{B}$ can at best satisfy only those atomic propositions that hold in all the states in $\mathcal{A}$ that it simulates.

[3] Unless explicitly mentioned, all our functions will be total.

Notice that the definition of simulation of transition systems, and therefore that of AP-simulation, is very general, not even requiring H to be total. This leads to some perhaps unexpected consequences: for example, the empty relation is vacuously a bisimulation! The notion is natural, however, in that every AP-simulation arises from a *total* AP-simulation restricted to a certain domain of interest.

Definition 5. *Given transition systems $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is a* subsystem *of $\mathcal{B}$ if $A \subseteq B$ and $\rightarrow_{\mathcal{A}} \subseteq \rightarrow_{\mathcal{B}}$; we then write $\mathcal{A} \subseteq \mathcal{B}$. We say that a subsystem $\mathcal{A}$ is* full *in $\mathcal{B}$ if for all $a \in A$, if $a \rightarrow_{\mathcal{B}} a'$ then $a' \in A$ and $a \rightarrow_{\mathcal{A}} a'$.*

A Kripke structure $\mathcal{A}$ is a Kripke substructure *of $\mathcal{B}$ if $\mathcal{A}$'s underlying transition system is a subsystem of that of $\mathcal{B}$ and $L_{\mathcal{A}} = L_{\mathcal{B}}|_A$. It is* full *if it is so at the level of transition systems.*

Remark 1. Note that if $\mathcal{A}$ is a full Kripke substructure of $\mathcal{B}$ then the inclusion $i : \mathcal{A} \longrightarrow \mathcal{B}$ is an AP-bisimulation.

Proposition 1. *Let $H : \mathcal{A} \longrightarrow \mathcal{B}$ be an AP-simulation. Then, for any full Kripke substructure $\mathcal{B}' \subseteq \mathcal{B}$, $H^{-1}(\mathcal{B}') = (H^{-1}(B'), \rightarrow_{\mathcal{A}} \cap (H^{-1}(B') \times H^{-1}(B')), L_{\mathcal{A}}|_{H^{-1}(B')})$ is a full Kripke substructure of $\mathcal{A}$. In particular, $H^{-1}(\mathcal{B})$ is a full Kripke substructure of $\mathcal{A}$.*

Proof. We have to show that the transition relation is *total* and that $H^{-1}(\mathcal{B}')$ is full in $\mathcal{A}$. Let a be an element of $H^{-1}(B')$ such that $a \rightarrow_{\mathcal{A}} a'$ (which exists because $\rightarrow_{\mathcal{A}}$ is total). By definition, there exists $b \in B'$ such that aHb. Now, since H is a simulation, there is $b' \in B$ such that $a'Hb'$ and $b \rightarrow_{\mathcal{B}} b'$, and since $\mathcal{B}'$ is full in $\mathcal{B}$, $b' \in B'$. Hence $a' \in H^{-1}(B')$, $\rightarrow_{H^{-1}(\mathcal{B}')}$ is total, and $H^{-1}(\mathcal{B}')$ is full in $\mathcal{A}$. □

Therefore, every AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ can alternatively be seen as a total simulation $H : H^{-1}(\mathcal{B}) \longrightarrow \mathcal{B}$.

As easy consequences of the definitions we have the following results about simulations.

Lemma 1. *If $\{H_i : \mathcal{A} \longrightarrow \mathcal{B}\}_{i \in I}$ is a set of simulations of transition systems (resp. AP-simulations) then $\bigcup_{i \in I} H_i : \mathcal{A} \longrightarrow \mathcal{B}$ is a simulation of transition systems (resp. an AP-simulation).*

Corollary 1. *For any two transition systems (resp. Kripke structures) $\mathcal{A}$ and $\mathcal{B}$ there is a greatest simulation of transition systems (resp. AP-simulation) between them (that can perhaps be empty).*

Corollary 2. *For any transition system (resp. Kripke structure) $\mathcal{A}$ there is a greatest bisimulation $H : \mathcal{A} \longrightarrow \mathcal{A}$ of transition systems (resp. AP-bisimulation) and it is an equivalence relation.*

Lemma 2. *If $F : \mathcal{A} \longrightarrow \mathcal{B}$ and $G : \mathcal{B} \longrightarrow \mathcal{C}$ are simulations of transition systems (resp. AP-simulations) then $G \circ F$ is also a simulation of transition systems (resp. AP-simulation).*

The important fact about AP-simulations is that they reflect the satisfaction of appropriate classes of formulas.

Definition 6. *An AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$* reflects *the satisfaction of a formula $\varphi \in \mathrm{CTL}^*(AP)$ if either:*

- *φ is a state formula, and $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$; or*
- *φ is a path formula, and $\mathcal{B}, \rho \models \varphi$ and $\pi H \rho$ imply $\mathcal{A}, \pi \models \varphi$.*

The following theorem slightly generalizes Theorem 16 in [9]:

Theorem 1. *AP-simulations always reflect satisfaction of* $\mathrm{ACTL}^*\backslash\neg(AP)$ *formulas. In addition, strict simulations also reflect satisfaction of* $\mathrm{ACTL}^*(AP)$ *formulas.*

Proof. Let us first consider the non-strict case. Let $H : \mathcal{A} \longrightarrow \mathcal{B}$ be an AP-simulation and let $a \in A$ and $b \in B$ be such that aHb. If π is a path in $\mathcal{A}$ starting in a, it is straightforward to prove that there is a path ρ in $\mathcal{B}$ that starts in b such that $\pi H \rho$, by induction over the length of initial segments. Then, for every state formula φ and path formula ψ in $\mathrm{ACTL}^*\backslash\neg(AP)$ it can be proved by simultaneous structural induction that $\mathcal{B}, b \models \varphi$ implies $\mathcal{A}, a \models \varphi$ and that $\mathcal{B}, \rho \models \psi$ implies $\mathcal{A}, \pi \models \psi$.

Each of the cases is immediate and we only consider some of them. For an atomic proposition p, if $\mathcal{B}, b \models p$ it follows that $p \in L_{\mathcal{B}}(b)$ and since H is a simulation, $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$ and $\mathcal{A}, a \models p$. For the cases corresponding to the operators $\top$, $\vee$, and $\wedge$ it is enough to apply the induction hypothesis. If $\mathcal{B}, b \models \mathbf{A}\psi$, then $\mathcal{B}, \rho' \models \psi$ for all paths ρ' that start in b. Let π' be a path that starts in a and, by abuse of notation, let $H(\pi')$ be a path in $\mathcal{B}$ that starts in b such that $\pi' \, H \, H(\pi')$. It follows that $\mathcal{B}, H(\pi') \models \psi$ and, by induction hypothesis, $\mathcal{A}, \pi' \models \psi$; since this holds for every path that starts in a, $\mathcal{A}, a \models \mathbf{A}\psi$. For the remaining temporal operators, the pattern is the same.

For the strict case, it is enough to show the result for formulas in negation-normal form because every formula is semantically equivalent to one of those. The proof proceeds as before with an additional case for $\neg p$, for which we have that $\mathcal{B}, b \models \neg p$ implies $p \notin L_{\mathcal{B}}(b)$ and thus, since H is strict, $p \notin L_{\mathcal{A}}(a)$ and $\mathcal{A}, a \models \neg p$. □

Corollary 3 ([9]). *If $H : \mathcal{A} \longrightarrow \mathcal{B}$ is an AP-bisimulation, then for any $\varphi \in \mathrm{CTL}^*(AP)$ and $a \in A$, $b \in B$ with aHb we have $\mathcal{A}, a \models \varphi$ iff $\mathcal{B}, b \models \varphi$.*

Note that, by Lemma 2 above, simulations of transition systems and of Kripke structures *compose*. Note also that the identity function $1_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{A}$ is trivially a simulation of transition systems and of Kripke structures. Therefore, transition systems together with their simulations define a category **TSys**. Similarly, Kripke structures together with AP-simulations define a category[4] $\mathbf{KSim}_{AP}$, with two corresponding subcategories $\mathbf{KMap}_{AP}$ and $\mathbf{KBSim}_{AP}$ whose morphisms are, respectively, AP-maps and AP-bisimulations. There is also of course a subcategory $\mathbf{KSim}^{\mathrm{str}}_{AP}$ of strict AP-simulations, and corresponding subcategories $\mathbf{KMap}^{\mathrm{str}}_{AP}$ and $\mathbf{KBSim}^{\mathrm{str}}_{AP} = \mathbf{KBSim}_{AP}$. Note that if H is an isomorphism in $\mathbf{KSim}_{AP}$ then it must be a map and a bisimulation. Note, finally, that the mapping $(A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}}) \mapsto (A, \rightarrow_{\mathcal{A}})$ extends to a forgetful functor $\mathbf{TS} : \mathbf{KSim}_{AP} \longrightarrow \mathbf{TSys}$.

2.3 Shifting One's Ground

We are interested in a more general definition of simulation, one in which Kripke structures over different sets AP and AP' of atomic propositions can be related. This provides a very flexible way of relating Kripke structures and will allow us to gather all the previous categories $\mathbf{KSim}_{AP}$ into a single one. First we need the following definition to translate the properties of a Kripke structure to a different set of atomic propositions.

Definition 7. *Given a function $\alpha : AP \longrightarrow \mathrm{State}(AP')$ and a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ over AP', we define the* reduct Kripke structure $\mathcal{A}|_\alpha = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}|_\alpha})$ *over AP, with labeling function $L_{\mathcal{A}|_\alpha}(a) = \{p \in AP \mid \mathcal{A}, a \models \alpha(p)\}$.*

[4] A categorically-oriented reader may recognize the category of Kripke structures and AP-simulations as the category of *partial* morphisms associated to the category of Kripke structures and *total* AP-simulations by the choice of full Kripke substructures as subobjects.

The definition of α is extended in the expected, homomorphic way to formulas $\varphi \in \mathrm{CTL}^*(AP)$, replacing each atomic proposition p occurring in φ by $\alpha(p)$; we denote the formula resulting from this translation by $\overline{\alpha}(\varphi)$. We then have the following result.

Proposition 2. *Let $\alpha : AP \longrightarrow \mathrm{State}(AP')$ be a function and let $\varphi \in \mathrm{CTL}^*(AP)$. Then, for all Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ over AP', states $a \in A$, and paths π:*

- *if φ is a state formula, $\mathcal{A}, a \models \overline{\alpha}(\varphi) \iff \mathcal{A}|_\alpha, a \models \varphi$, and*
- *if φ is a path formula, $\mathcal{A}, \pi \models \overline{\alpha}(\varphi) \iff \mathcal{A}|_\alpha, \pi \models \varphi$.*

Proof. We prove both statements simultaneously by induction on the structure of formulas. The result follows by definition of $\mathcal{A}|_\alpha$ if p is an atomic proposition, and it is trivial for $\top$ and $\bot$. For $\mathbf{A}\varphi$, it is

$$\begin{aligned} \mathcal{A}, a \models \overline{\alpha}(\mathbf{A}\varphi) &\iff \mathcal{A}, \pi \models \overline{\alpha}(\varphi) \text{ for all paths } \pi \text{ starting at } a \\ &\iff \mathcal{A}|_\alpha, \pi \models \varphi \quad \text{for all paths } \pi \text{ starting at } a \\ &\iff \mathcal{A}|_\alpha, a \models \mathbf{A}\varphi \end{aligned}$$

where the first equivalence holds because $\overline{\alpha}(\mathbf{A}\varphi) = \mathbf{A}\overline{\alpha}(\varphi)$ and the second one because of the induction hypothesis. For $\mathbf{F}\varphi$,

$$\begin{aligned} \mathcal{A}, \pi \models \overline{\alpha}(\mathbf{F}\varphi) &\iff \mathcal{A}, \pi^n \models \overline{\alpha}(\varphi) \text{ for some } n \in \mathbb{N} \\ &\iff \mathcal{A}|_\alpha, \pi^n \models \varphi \quad \text{for some } n \in \mathbb{N} \\ &\iff \mathcal{A}|_\alpha, \pi \models \mathbf{F}\varphi \end{aligned}$$

where the first equivalence holds because $\overline{\alpha}(\mathbf{F}\varphi) = \mathbf{F}\overline{\alpha}(\varphi)$ and the second equivalence follows from the induction hypothesis. The proof proceeds analogously for the rest of temporal logic operators. □

Note that it makes no sense to map an atomic proposition, which is a state formula, to an arbitrary CTL* formula that may turn out to be a path formula. Therefore, the choice of $\mathrm{State}(AP')$ as the range of the functions α is as general as possible. Also, note that when dealing with non-strict simulations, since the reflected formulas will be in $\mathrm{ACTL}^* \backslash \neg(AP)$, we will want our functions α to have their range in the negation-free fragment $\mathrm{State}\backslash\neg(AP')$, i.e., we will use functions $\alpha : AP \longrightarrow \mathrm{State}\backslash\neg(AP')$ instead.

The definition of generalized simulations is now immediate.

Definition 8. *Given a Kripke structure $\mathcal{A}$ over a set AP of atomic propositions and a Kripke structure $\mathcal{B}$ over a set AP', a* simulation *(resp.* strict simulation*) $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ consists of a function $\alpha : AP \longrightarrow \mathrm{State}\backslash\neg(AP')$ (resp. $\alpha : AP \longrightarrow \mathrm{State}(AP')$) and an AP-simulation (resp. strict AP-simulation) $H : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$. We call (α, H) a* map *(resp.* strict map*) or a* bisimulation *if H is so in the category $\mathbf{KSim}_{AP}$ (resp. $\mathbf{KSim}^{\mathrm{str}}_{AP}$).*

Proposition 3. *If $(\alpha, F) : \mathcal{A} \longrightarrow \mathcal{B}$ and $(\beta, G) : \mathcal{B} \longrightarrow \mathcal{C}$ are simulations, then their composition $(\beta, G) \circ (\alpha, F) = (\overline{\beta} \circ \alpha, G \circ F)$ is also a simulation.*

Proof. Assuming that AP is the set of atomic propositions of $\mathcal{A}$, we have to check that $G \circ F : \mathcal{A} \longrightarrow \mathcal{C}|_{\overline{\beta}\circ\alpha}$ is an AP-simulation. Let $a \in A$ and $c \in C$ be such that $a(G \circ F)c$; then there is $b \in \mathcal{B}$ such that aFb and bGc.

Let us first check that $G \circ F$ is a simulation of the underlying transition systems. If $a \rightarrow_{\mathcal{A}} a'$, since $F : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$ is an AP-simulation there is $b' \in B$ such that $a'Fb'$ and $b \rightarrow_{\mathcal{B}} b'$; now, analogously, there is $c' \in C$ such that $b'Gc'$ and $c \rightarrow_{\mathcal{C}} c'$. We then have $a'(G \circ F)c'$ with $c \rightarrow_{\mathcal{C}} c'$ as required.

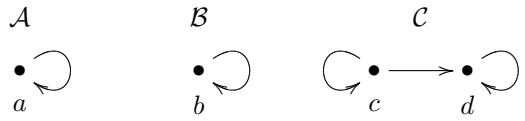
Now, let $p \in L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c)$:

$$\begin{array}{rcll} p \in L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c) & \iff & \mathcal{C}, c \models \overline{\beta}(\alpha(p)) & \text{(by definition)} \\ & \iff & \mathcal{C}|_\beta, c \models \alpha(p) & \text{(by Proposition 2)} \\ & \implies & \mathcal{B}, b \models \alpha(p) & \text{(by Theorem 1)} \\ & \iff & \mathcal{B}|_\alpha, b \models p & \text{(by Proposition 2)} \\ & \implies & \mathcal{A}, a \models p & (F \text{ is an } AP\text{-simulation}) \\ & \iff & p \in L_{\mathcal{A}}(a). & \end{array}$$

That is, $L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c) \subseteq L_{\mathcal{A}}(a)$ and $G \circ F : \mathcal{A} \longrightarrow \mathcal{C}|_{\overline{\beta}\circ\alpha}$ is an AP-simulation. □

Therefore, using as objects pairs $(AP, \mathcal{A})$ with AP a set of atomic propositions and $\mathcal{A}$ a Kripke structure over AP, this immediately gives rise to categories **KSim**, **KMap**, and **KBSim**. Again, if (α, H) is an isomorphism in **KSim** then H must be a map and a bisimulation.

Note, however, that strict simulations *do not* compose. In the above proof, if F and G were strict the last implication would become an equivalence, but the first one would not. For example, consider the following three Kripke structures over the same set $AP = \{p\}$ of atomic propositions, with $L_{\mathcal{A}}(a) = L_{\mathcal{B}}(b) = L_{\mathcal{C}}(c) = \{p\}$ and $L_{\mathcal{C}}(d) = \emptyset$:



Now, if we define $\alpha(p) = \mathbf{AG}p$, $\beta(p) = p$, $f(a) = b$, and $g(b) = c$, it is easy to check that $(\alpha, f) : (AP, \mathcal{A}) \longrightarrow (AP, \mathcal{B})$ and $(\beta, g) : (AP, \mathcal{B}) \longrightarrow (AP, \mathcal{C})$ are strict simulations but $(\overline{\beta} \circ \alpha, g \circ f) : (AP, \mathcal{A}) \longrightarrow (AP, \mathcal{C})$ is not: $p \notin L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c)$ because $\mathcal{C}, c \not\models \mathbf{AG}p$. Clearly, the reason behind this lies in the fact that α maps an atomic proposition to an arbitrary state formula. Strict AP-simulations compose because the elements they relate satisfy exactly the same set of atomic propositions; now that we are shifting our ground and transforming atomic propositions into general state formulas it would be necessary for related elements to satisfy those, which in general is not the case.

Thus, if one were interested in having strict simulations that could be composed the range of the function α would have to be restricted. It would be enough to require α to map atomic propositions only to atomic propositions, but a mild generalization is actually possible: α's range has to be of the form $\mathrm{Bool}(AP)$, the state formulas in $\mathrm{ACTL}^*(AP)$ that do not contain the operator $\mathbf{A}$ (that is, the Boolean expressions over AP). Then, composition of strict simulations is a consequence of the following specialization of Theorem 1.

Proposition 4. *Strict AP-simulations always preserve formulas in* $\mathrm{Bool}(AP)$*, in addition to reflecting them.*

To simplify notation, from now on we will write $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ instead of $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ except in those cases where it could lead to confusion.

Definition 9. *Given Kripke structures $\mathcal{A}$ over AP and $\mathcal{B}$ over AP', a simulation $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$* reflects *the satisfaction of a formula $\varphi \in \mathrm{CTL}^*(AP)$ if either:*

- *φ is a state formula, and $\mathcal{B}, b \models \overline{\alpha}(\varphi)$ and aHb imply that $\mathcal{A}, a \models \varphi$; or*
- *φ is a path formula, and $\mathcal{B}, \rho \models \overline{\alpha}(\varphi)$ and $\pi H \rho$ imply that $\mathcal{A}, \pi \models \varphi$.*

The main results that we had for Kripke structures over a fixed set of atomic propositions extend naturally to generalized simulations.

Theorem 2. *Simulations always reflect satisfaction of* $\mathrm{ACTL}^*\backslash\neg$ *formulas. In addition, strict simulations also reflect satisfaction of* ACTL^* *formulas.*

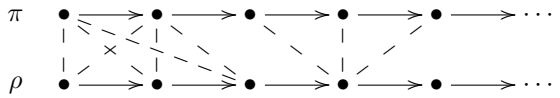
Proof. The result is a consequence of Proposition 2 and Theorem 1. □

2.4 Stuttering Simulations

Another direction in which the original definition of simulation can be extended is that of stuttering bisimulations [5, 40] and, more generally, stuttering simulations [27].

Definition 10. *Let* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ *and* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ *be transition systems and let* $H \subseteq A \times B$ *be a relation. Given a path* π *in* $\mathcal{A}$ *and a path* ρ *in* $\mathcal{B}$*, we say that* ρ H-matches π *if there are strictly increasing functions* $\alpha, \beta : \mathbb{N} \longrightarrow \mathbb{N}$ *with* $\alpha(0) = \beta(0) = 0$ *such that, for all* $i, j, k \in \mathbb{N}$*, if* $\alpha(i) \leq j < \alpha(i+1)$ *and* $\beta(i) \leq k < \beta(i+1)$*, it holds that* $\pi(j) H \rho(k)$*.*

For example, the following diagram shows the beginning of two matching paths, where related elements are joined by dashed lines and $\alpha(0) = \beta(0) = 0$, $\alpha(1) = 2$, $\beta(1) = 3$, $\alpha(2) = 5$, etc.



Definition 11. *Given transition systems* $\mathcal{A}$ *and* $\mathcal{B}$*, a* stuttering simulation of transition systems $H : \mathcal{A} \longrightarrow \mathcal{B}$ *is a binary relation* $H \subseteq A \times B$ *such that if* aHb*, then for each path* π *in* $\mathcal{A}$ *starting at* a *there is a path* ρ *in* $\mathcal{B}$ *starting at* b *that* H*-matches* π*.*

If H *is a function we say that* H *is a* stuttering map of transition systems*. If both* H *and* H^{-1} *are stuttering simulations, then we call* H *a* stuttering bisimulation*.*

Stuttering simulations of transition systems compose [27] and together with transition systems define a category that we denote **STSys**.

The extension to Kripke structures is immediate:

Definition 12. *Given Kripke structures* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ *and* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ *over* AP*, a* stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *is a stuttering simulation of transition systems* $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ *such that if* aHb *then* $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$*. We call the stuttering* AP*-simulation* strict *if* aHb *implies* $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$*.*

Again, stuttering AP-simulations compose and define a category $\mathbf{KSSim}_{AP}$ with corresponding subcategories of strict and stuttering AP-maps.

Our definition of stuttering simulation is closely related to the one given by Manolios [27, 28], but with some technical and methodological differences. He defines such simulation on a single set obtained by forming the disjoint union of the two Kripke structures, and has two different ingredients: the simulation relation and a refinement map which "borrows" for the source structure the labeling information from the target structure.

As it happened for AP-simulations, every stuttering AP-simulation arises from a total one.

Proposition 5. *Let* $H : \mathcal{A} \longrightarrow \mathcal{B}$ *be a stuttering* AP*-simulation. Then, for any full Kripke substructure* $\mathcal{B}' \subseteq \mathcal{B}$*, the triple* $H^{-1}(\mathcal{B}') = (H^{-1}(B'), \rightarrow_{\mathcal{A}} \cap (H^{-1}(B') \times H^{-1}(B')), L_{\mathcal{A}}|_{H^{-1}(B')})$ *is a full Kripke substructure of* $\mathcal{A}$*. In particular,* $H^{-1}(\mathcal{B})$ *is a full Kripke substructure of* $\mathcal{A}$*.*

Proof. We have to show that the transition relation is total and that $H^{-1}(\mathcal{B}')$ is full in $\mathcal{A}$. Let a be an element of $H^{-1}(B')$ such that $a \rightarrow_{\mathcal{A}} a'$ (which exists because $\rightarrow_{\mathcal{A}}$ is total), and let π be a path in $\mathcal{A}$ such that $\pi(0) = a$ and $\pi(1) = a'$. By definition, there exists $b \in B'$ such that aHb. Now, since H is a stuttering AP-simulation, there is a path ρ in $\mathcal{B}$ starting at b that H-matches π, and since $\mathcal{B}'$ is full in $\mathcal{B}$, ρ is actually a path in $\mathcal{B}'$. Since ρ H-matches π, it is $a'H\rho(i)$ for some i; hence $a' \in H^{-1}(B')$, $\rightarrow_{H^{-1}(\mathcal{B}')}$ is total, and $H^{-1}(\mathcal{B}')$ is full in $\mathcal{A}$. □

The definition of when a simulation reflects the satisfaction of a formula has to be slightly modified in this new context for the case of paths.

Definition 13. *A stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$* reflects *the satisfaction of a formula $\varphi \in \mathrm{CTL}^*(AP)$ if either:*

- *φ is a state formula, and $\mathcal{B}, b \models \varphi$ and aHb imply that $\mathcal{A}, a \models \varphi$; or*
- *φ is a path formula, and $\mathcal{B}, \rho \models \varphi$ and ρ H-matches π imply that $\mathcal{A}, \pi \models \varphi$.*

It is clear that the *next* operator $\mathbf{X}$ of temporal logic is not reflected by stuttering AP-simulations; however, if we restrict our attention to $\mathrm{ACTL}^* \backslash \mathbf{X}(AP)$ and $\mathrm{ACTL}^* \backslash \{\neg, \mathbf{X}\}(AP)$, that is, the fragments of the logics that do not contain $\mathbf{X}$, formulas are reflected. In practice, the elimination of the operator $\mathbf{X}$ is not a great loss since, as argued in [24], interesting properties are not so much concerned about what happens in the next step as to what eventually happens. Also, the notion of "next step" assumes a fixed notion of *atomic* transition, whereas one of the important roles played by stuttering simulations is that they can relate systems having different levels of computational granularity, so that what is an atomic transition in one system may correspond to a sequence of transitions in the other system.

Theorem 3. *Stuttering AP-simulations always reflect satisfaction of $\mathrm{ACTL}^* \backslash \{\neg, \mathbf{X}\}(AP)$ formulas. In addition, strict stuttering AP-simulations also reflect satisfaction of formulas in $\mathrm{ACTL}^* \setminus \mathbf{X}(AP)$.*

Proof. Let $H : \mathcal{A} \longrightarrow \mathcal{B}$ be a stuttering AP-simulation and assume that aHb and that ρ H-matches π through α and β; we proceed by induction on the structure of state and path formulas.

For an atomic proposition p, if $\mathcal{B}, b \models p$ then $p \in L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$, and thus $\mathcal{A}, a \models p$. The result is trivial for $\top$ and $\bot$. For a state formula $\mathbf{A}\varphi$, if $\mathcal{B}, b \models \mathbf{A}\varphi$ then $\mathcal{B}, \rho' \models \varphi$ for all paths ρ' in $\mathcal{B}$ starting at b. Let then π' be a path in $\mathcal{A}$ starting at a and, by abuse of notation, write $H(\pi')$ for one of its H-matching paths in $\mathcal{B}$ starting at b. Then, $\mathcal{B}, H(\pi') \models \varphi$ and, by induction hypothesis, $\mathcal{A}, \pi' \models \varphi$, and therefore $\mathcal{A}, a \models \mathbf{A}\varphi$.

The result for the logical operators $\vee$ and $\wedge$, for state and path formulas, follows straightforwardly from the induction hypothesis.

If $\mathcal{B}, \rho \models \mathbf{F}\varphi$ then there exists $n \in \mathbb{N}$ such that $\mathcal{B}, \rho^n \models \varphi$. Let i be the unique natural number such that $\beta(i) \leq n < \beta(i+1)$. Then $\rho^{\beta(i)}$, but also ρ^n, H-match $\pi^{\alpha(i)}$ and, by induction hypothesis, $\mathcal{A}, \pi^{\alpha(i)} \models \varphi$ and therefore $\mathcal{A}, \pi \models \mathbf{F}\varphi$.

If $\mathcal{B}, \rho \models \varphi_1 \mathbf{U} \varphi_2$, there exists $n \in \mathbb{N}$ such that $\mathcal{B}, \rho^n \models \varphi_2$ and, for all $m < n$, $\mathcal{B}, \rho^m \models \varphi_1$. Let i be the unique natural number such that $\beta(i) \leq n < \beta(i+1)$. Then ρ^n H-matches $\pi^{\alpha(i)}$ and, by induction hypothesis, $\mathcal{A}, \pi^{\alpha(i)} \models \varphi_2$. Let $m < \alpha(i)$. If j is the unique natural number such that $\alpha(j) \leq m < \alpha(j+1)$, since α is strictly increasing it must be $j < i$, and since β is also strictly increasing, $\beta(j) < \beta(i) \leq n$ and thus $\mathcal{B}, \rho^{\beta(j)} \models \varphi_1$. Since $\rho^{\beta(j)}$ H-matches π^m, $\mathcal{A}, \pi^m \models \varphi_1$ by induction hypothesis and therefore $\mathcal{A}, \pi \models \varphi_1 \mathbf{U} \varphi_2$.

The proofs for $\mathbf{R}$ and $\mathbf{G}$ are similar.

In the case of strict stuttering AP-simulations it is enough to consider only formulas in negation-normal form. The proof proceeds exactly as above but we have to consider the additional case in which the formula is of the form $\neg p$. In this case, if $\mathcal{B}, b \models \neg p$ then $p \notin L_{\mathcal{B}}(b)$ and, since H is strict, $p \notin L_{\mathcal{A}}(a)$ and $\mathcal{A}, a \models \neg p$ as required. □

Corollary 4. *If $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a stuttering AP-bisimulation, then for any $\varphi \in \mathrm{CTL}^* \setminus \mathbf{X}(AP)$ and $a \in A$, $b \in B$ with aHb we have $\mathcal{A}, a \models \varphi$ iff $\mathcal{B}, b \models \varphi$.*

Proof. It is essentially like that for Theorem 3. The only new case corresponds to the existential quantifier $\mathbf{E}$, which follows because now given a path ρ in $\mathcal{B}$ we can always find a path π in $\mathcal{A}$ such that π H-matches ρ.

Finally, we can combine both extensions of the notion of simulation for Kripke structures (stuttering and shifting one's ground) into a single definition.

Definition 14. *Given a Kripke structure $\mathcal{A}$ over a set AP of atomic propositions and a Kripke structure $\mathcal{B}$ over a set AP', a* stuttering simulation *(resp.* strict stuttering simulation*)* $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ *consists of a function $\alpha : AP \longrightarrow \mathrm{State}\backslash\{\neg, \mathbf{X}\}(AP')$ (resp. $\alpha : AP \longrightarrow \mathrm{State} \setminus \mathbf{X}(AP')$) and a stuttering AP-simulation (resp. strict stuttering AP-simulation) $H : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$.*

Note that we have restricted the range of α by forbidding the use of $\mathbf{X}$. This is in correspondence with the fact that the *next* operator is useless in the presence of stuttering. The notions of stuttering map, strict stuttering map, and bisimulation are defined in the expected way. Again, we will usually write $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ instead of $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$.

Proposition 6. *If $(\alpha, F) : \mathcal{A} \longrightarrow \mathcal{B}$ and $(\beta, G) : \mathcal{B} \longrightarrow \mathcal{C}$ are stuttering simulations, then $(\beta, G) \circ (\alpha, F) = (\overline{\beta} \circ \alpha, G \circ F)$ is also a stuttering simulation.*

Proof. Assume that $\mathcal{A}$ is a Kripke structure over the set of atomic propositions AP: we have to check that $G \circ F : \mathcal{A} \longrightarrow \mathcal{C}|_{\overline{\beta}\circ\alpha}$ is a stuttering AP-simulation. F and G are stuttering simulations of the underlying transition systems and therefore, as proved in [27], its composition is also a stuttering simulation of transition systems. Let now $a \in A$ and $c \in C$ be such that $a(G \circ F)c$, and let $p \in L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c)$. Then, there exists $b \in B$ such that aFb and bGc, and we have the following chain of implications:

$$\begin{array}{rll}
p \in L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c) & \iff & \mathcal{C}, c \models \overline{\beta}(\alpha(p)) \text{ (by definition)} \\
& \iff & \mathcal{C}|_\beta, c \models \alpha(p) \quad \text{(by Proposition 2)} \\
& \implies & \mathcal{B}, b \models \alpha(p) \quad \text{(by Theorem 3)} \\
& \iff & \mathcal{B}|_\alpha, b \models p \quad \text{(by Proposition 2)} \\
& \implies & \mathcal{A}, a \models p \quad (F \text{ is a stuttering } AP\text{-simulation}) \\
& \iff & p \in L_{\mathcal{A}}(a).
\end{array}$$

That is, $L_{\mathcal{C}|_{\overline{\beta}\circ\alpha}}(c) \subseteq L_{\mathcal{A}}(a)$ and $G \circ F : \mathcal{A} \longrightarrow \mathcal{C}|_{\overline{\beta}\circ\alpha}$ is a stuttering AP-simulation. □

Therefore, we have a category **KSSim** of Kripke structures and stuttering simulations, with corresponding subcategories for stuttering AP-simulations, maps, bisimulations. As we noted when we first presented generalized simulations in Section 2.3, in order for strict stuttering simulations to compose it is necessary that the range of the function α be $\mathrm{Bool}(AP)$.

The following theorem generalizes all previous results about simulations reflecting the satisfaction relation.

Theorem 4. *Stuttering simulations always reflect satisfaction of* $\mathrm{ACTL}^*\backslash\{\neg, \mathbf{X}\}$ *formulas. In addition, strict stuttering simulations also reflect satisfaction of* $\mathrm{ACTL}^* \setminus \mathbf{X}$ *formulas.*

Proof. It is a consequence of Theorem 3 and Proposition 2. □

Remark 2. Actually, this result is true even if we allow general functions of the form $\alpha : AP \to \mathrm{State}\backslash\neg(AP')$ in the definition of stuttering simulations; the restriction to formulas without the *next* operator is only necessary for the composition to be well-defined.

Corollary 5. *If $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ is a stuttering bisimulation, then for any $\varphi \in \mathrm{CTL}^* \setminus \mathbf{X}(AP)$ and $a \in A$, $b \in B$ with aHb we have $\mathcal{A}, a \models \varphi$ iff $\mathcal{B}, b \models \overline{\alpha}(\varphi)$.*

Definition 12 characterizes stuttering simulations in terms of infinite paths. In [27], an alternative, more finitary characterization, called well-founded simulation, is presented, which can also be adapted to our framework.

Definition 15. *Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be transition systems. A relation $H \subseteq A \times B$ is a* well-founded simulation of transition systems *from $\mathcal{A}$ to $\mathcal{B}$ if there exist functions $\mu : A \times B \longrightarrow W$ and $\mu' : A \times A \times B \longrightarrow \mathbb{N}$, with $(W, <)$ a well-founded order, such that whenever aHb and $a \rightarrow_{\mathcal{A}} a'$, either:*

1. *there is b' such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, or*
2. *$a'Hb$ and $\mu(a', b) < \mu(a, b)$, or*
3. *there is b' such that $b \rightarrow_{\mathcal{B}} b'$, aHb', and $\mu'(a, a', b') < \mu'(a, a', b)$.*

Remark 3. Note that if H is a function only conditions (1) and (2) apply and the function μ' is not necessary.

Definition 16. *Given Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over AP, a relation $H \subseteq A \times B$ is a* well-founded AP-simulation *if H is a well-founded simulation of transition systems and $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$ whenever aHb.*

Then we have the following important theorem, which can be proved by adapting the proof in [27] to our setting.

Theorem 5. *Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ be Kripke structures over AP, and $H \subseteq A \times B$. Then, H is a well-founded AP-simulation iff it is a stuttering AP-simulation.*

3 Membership Equational Logic and Rewriting Logic

When specifying a system, one can distinguish two specification levels:

- a *system specification* level, in which the computational system of interest is specified, and
- a *property specification* level, in which the relevant properties are specified.

The system itself will typically be some kind of *transition system* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$. However, to be able to talk about system properties we may need to make explicit some state predicates in such a system. Such predicates may not belong to the original system specification: they may just be needed in order to interpret relevant properties. Such an interpretation of atomic propositions will typically be made by adding a labeling function $L : A \rightarrow \mathcal{P}(AP)$ to our transition system, thus obtaining a *Kripke structure* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$. The property specification level as such, will then typically correspond to the different temporal logic formulas φ (or formulas in some other logic) that such a system is then supposed to satisfy, according to a satisfaction relation $\mathcal{A}, a \models \varphi$.

The main interest of rewriting logic [33] is that it provides a very flexible framework for the *system-level specification* of concurrent systems, as witnessed by the numerous references in [30]. Rewriting logic is parameterized by an underlying equational logic: in this paper we use membership equational logic, whose main features we now review. As we shall see, both transition systems and Kripke structures can be naturally specified in a high level way as rewrite theories. The property specification level will correspond to those temporal logic formulas that the system so specified satisfies.

3.1 Membership Equational Logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0$ **if** $A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership assertion, and X is a set of K-kinded variables containing all the variables in the A_i. A *theory* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ. We write $(\Sigma, E) \vdash \phi$, or just $E \vdash \phi$ if Σ is clear from the context, to denote that (Σ, E) entails the sentence ϕ in the proof system of membership equational logic [35].

A Σ-*algebra* A consists of a set A_k for each $k \in K$, a function $A_f : A_{k_1} \times \ldots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, whereas elements in a kind not having a sort are "undefined" or "error" elements. A theory (Σ, E) has an initial model $T_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [4, 35] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, initial and free algebras, and theory morphisms.

3.2 Rewriting Logic

Concurrent systems are axiomatized in rewriting logic by means of *rewrite theories* [33] of the form $\mathcal{R} = (\Sigma, E, R, \phi)$. The set of states is described by a membership equational theory (Σ, E) as the algebraic data type $T_{\Sigma/E,k}$ associated to the initial algebra $T_{\Sigma/E}$ of (Σ, E) by the choice of a kind k of states in Σ. The system's *transitions* are axiomatized by the *conditional rewrite rules* R, which are of the form

$$\lambda : (\forall X)\, t \longrightarrow t' \textbf{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t'_l\,,$$

with λ a label, $p_i = q_i$ and $w_j : s_j$ atomic formulas in membership equational logic for $i \in I$ and $j \in J$, and for appropriate kinds k and k_l, $t, t' \in T_{\Sigma,k}(X)$, and $t_l, t'_l \in T_{\Sigma,k_l}(X)$ for $l \in L$. The last component $\phi : \sigma \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ is a function assigning to each $f : k_1 \ldots k_n \longrightarrow k$ in Σ a set $\phi(f) = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ of *frozen* argument positions, so that for $f(t_1, \ldots, t_n)$ it is forbidden to rewrite with R at any subterm position t_j with $j \in \phi(f)$. This function will be omitted if $\phi(f) = \emptyset$ for all f.

Rewriting logic has inference rules to infer all the possible concurrent computations in a system [33, 6], in the sense that, given two states $[u], [v] \in T_{\Sigma/E,k}$, we can reach $[v]$ from $[u]$ by some possibly complex concurrent computation iff we can prove $u \longrightarrow v$ in the logic; we denote this provability relation by $\mathcal{R} \vdash u \longrightarrow v$. In particular we can easily define the *one-step* $\mathcal{R}$-*rewriting relation*, which is a binary relation $\to^1_{\mathcal{R},k}$ on $T_{\Sigma,k}$ that holds between terms $u, v \in T_{\Sigma,k}$ iff there is a one-step proof of $u \longrightarrow v$, that is, if there is a proof in which only one rewrite rule in R is applied to a single subterm. We can get a binary relation (with the same name) $\to^1_{\mathcal{R},k}$ on $T_{\Sigma/E,k}$ by defining $[u] \to^1_{\mathcal{R},k} [v]$ iff $u' \to^1_{\mathcal{R},k} v'$ for some $u' \in [u]$, $v' \in [v]$. This defines a transition system $\mathcal{T}(\mathcal{R})_k = (T_{\Sigma/E,k}, (\to^1_{\mathcal{R},k})^\bullet)$ for each $k \in K$.

Under reasonable assumptions about E and R, rewrite theories are *executable*. Indeed, there are several rewriting logic language implementations, including CafeOBJ [21], ELAN [3], and Maude [10–12]. From an operational viewpoint, the set of equations is divided into a set A of equational axioms for some of the operators in the signature, for which there exists a finitary A-matching algorithm, and a set E that will always be considered to be a set of simplifying (oriented) equations modulo A. For a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$ to be executable the equations E have to be (ground) Church-Rosser and terminating modulo A, and the rules R have to be (ground) *coherent* [51] relative to the equations E modulo A. The last condition means that for each ground term t, whenever we have $t \rightarrow^1_{\mathcal{R}} u$ we can always find a one-step rewrite $can_{E/A}(t) \rightarrow^1_{\mathcal{R}} v$ such that $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$, where $can_{E/A}(t)$ denotes the canonical form of t after simplification with the equations E modulo A, which by the Church-Rosser and termination assumptions exists and is unique modulo A. This implies that $(\rightarrow^1_{\mathcal{R},k})^\bullet$ is a computable binary relation on $T_{\Sigma/E\cup A,k}$, since we can decide $[t]_{E\cup A} \rightarrow^1_{\mathcal{R}} [u]_{E\cup A}$ by enumerating the finite set of all one-step $\mathcal{R}$-rewrites modulo A of $can_{E/A}(t)$, and for any such rewrite, say v, we can decide $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$.

3.3 Example: Semantics of a Functional Language

In [23], a simple functional language called *Fpl* is defined along with three different semantics. In Section 5.3.1 we will use this language and two of its semantics to illustrate simulations, but now we consider it just for the purpose of showing how systems are specified in rewriting logic.

We consider the computation semantics: a state is a pair $\langle \rho, e \rangle$, with ρ an environment and e an expression. Environments are represented in a rewrite theory by terms of sort `Env`. Similarly, there are two sorts to represent numerical and Boolean expressions, `NExp` and `BExp`, together with several operators, like `_+_ : NExp NExp -> NExp` to represent addition, or `If_Then_Else_ : BExp NExp NExp -> NExp` for conditional expressions, where the underbars are placeholders for the arguments. Finally, states are constructed with operators `<_,_> : Env NExp -> State` and `<_,_> : Env BExp -> State`. In this particular example no equations are needed. Then, the transitions of the system are given by rewrite rules like

```
rl [IfRc] : < rho, If T Then e Else e' > => < rho, e > .
```

that specifies the behavior of the `If` expression when its condition is true. (`IfRc` is the label, and `crl` would be used to introduce a conditional rule.) The complete set of rules of the rewrite theory can be found in Section 5.3.1 (see Figure 2 on page 27), and it gives rise to a transition system $\mathcal{C} = (C, \rightarrow_{\mathcal{C}})$.

4 Specifying Kripke Structures as Rewrite Theories

4.1 Temporal Properties of Rewrite Theories

In order to be able to associate temporal properties to a rewrite theory $\mathcal{R} = (\Sigma, E, R, \phi)$ we first need to make explicit two things: the intended *kind* k of states in the signature Σ, and the relevant *state predicates* on which any such temporal properties will be based, so that they can be interpreted in our system.

Once the kind k is fixed, the transitions between states are given by $\mathcal{T}(\mathcal{R})_k$. In general, however, the state predicates need not be part of the system specification but may only be needed for property verification purposes when we want to show that some temporal logic properties are satisfied. We can assume that they have been defined by means of equations D

in a *protecting* theory extension $(\Sigma', E \cup D)$ of (Σ, E); that is, the extension is conservative in the sense that the unique Σ-homomorphism $T_{\Sigma/E} \longrightarrow T_{\Sigma'/E\cup D}|_\Sigma$ should be bijective at each sort in Σ. We also assume that $(\Sigma', E \cup D)$ contains the theory $BOOL$ of Boolean values in *protecting* mode. Furthermore, we assume that the syntax defining the state predicates consists of a subsignature $\Pi \subseteq \Sigma'$ of operators, with each $p \in \Pi$ a different state predicate symbol that can be *parameterized*, that is, p need not be a constant, but can in general be an operator $p : s_1 \dots s_n \longrightarrow Prop$. If k is the kind of states, the *semantics* of the state predicates Π is defined with the help of an operator $\_ \models \_ : k\ Prop \longrightarrow Bool$ in Σ' and by the equations $E \cup D$. By definition, given ground terms $u_1, \dots, u_n$, we say that the state predicate $p(u_1, \dots, u_n)$ *holds* in the state $[t]$ iff

$$E \cup D \vdash t \models p(u_1, \dots, u_n) = true\,.$$

Then, we associate to a rewrite theory $\mathcal{R} = (\Sigma, E, R, \phi)$ (with a selected kind k of states and with state predicates Π) a Kripke structure whose atomic propositions are specified by the set $AP_\Pi = \{\theta(p) \mid p \in \Pi,\ \theta \text{ ground substitution}\}$, where by convention we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \dots, x_n))$. We define

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet, L_\Pi)$$

where

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid \theta(p) \text{ holds in } [t]\}\,.$$

For example, if we consider as the set of atomic propositions the set of all possible values, the rewrite theory specifying the computation semantics of the *Fpl* language in Section 3.3 can be extended by declaring a constant `v : -> Prop` for each value v and equations

```
eq (< rho, v > |= w) = true if v = w .
```

that define $L_\mathcal{C}(\langle \rho, v \rangle) = \{v\}$ and $L_\mathcal{C}(c)$ empty otherwise.

4.2 General Representability Results

What is the point of using rewrite theories to specify Kripke structures? It is a *logical* point: in this way, we have at our disposal two logics to specify a system and its predicates, namely, membership equational logic to specify the data type of states and its atomic propositions, and rewriting logic to specify the system's transitions. This is quite useful for reasoning about the properties of a system so specified. For example, when doing deductive reasoning about temporal logic properties we can use a host of inductive equational techniques combined with temporal logic reasoning to prove that certain formulas hold. Likewise, for model checking it is possible to specify at a high level many different Kripke structures as rewrite theories and (assuming finitary reachability) to model check their properties in a tool like Maude's LTL model checker [20, 12].

What is the *generality* of rewriting logic to specify Kripke structures? That is, can we specify in this way any Kripke structure that we may care about? The answer is *yes*. Furthermore, if the Kripke structure is *recursive*, then the corresponding rewrite theory will be finitary and also recursive in a suitable sense.

This brings us to the notions of recursive transition system and Kripke structure. We use the notion of recursive set and recursive function in the same sense as Shoenfield [48].

Definition 17. *A transition system $\mathcal{B} = (B, \rightarrow_\mathcal{B})$ is called* recursive *if B is a recursive set and there is a recursive function $next : B \longrightarrow \mathcal{P}_{\text{fin}}(B)$ (where $\mathcal{P}_{\text{fin}}(B)$ is the recursive set of finite subsets of B) such that $a \rightarrow_\mathcal{B} b$ iff $b \in next(a)$.*

Definition 18. *A Kripke structure* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ *is called* recursive *if* $(B, \rightarrow_{\mathcal{B}})$ *is a recursive transition system, AP is a recursive set, and the function* $\hat{L}_{\mathcal{B}} : B \times AP \longrightarrow Bool$ *mapping a pair* (a, p) *to* ***true*** *if* $p \in L_{\mathcal{B}}(a)$ *and to* ***false*** *otherwise, is recursive.*

The above notions of recursive transition system and recursive Kripke structure capture the intuition of systems for which we can effectively determine in a finite number of steps all the one-step successors of a given state. This is a stronger notion than just requiring that the transition relation $\rightarrow_{\mathcal{B}}$ be recursive, since then the set of next states of a given state would in general only be recursively enumerable (in short, r.e.). Note that being a recursive Kripke structure is a necessary condition for effectively model checking the satisfaction of temporal logic formulas in an initial state. In general, however, recursiveness is not a sufficient condition for effective model checking unless the set of states reachable from the given initial state is *finite*.

By a well-known metatheorem of Bergstra and Tucker [2], recursive sets and recursive functions coincide with those sets and functions that can be specified by a finite signature Σ and a finite set of Church-Rosser and terminating equations E. The underlying carrier sets of the initial algebra $T_{\Sigma/E}$ are the desired recursive sets, and the operations of the algebra provide the recursive functions. In the context of Kripke structures, this means that if $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ is a recursive Kripke structure, then B, AP, and $\hat{L}_{\mathcal{B}}$ can always be specified by a finite signature and set of equations. In our approach, this is accomplished by specifying B as the carrier of a kind k of an initial algebra $T_{\Sigma/E}$ with Σ finite and E Church-Rosser and terminating, and specifying $\hat{L}_{\mathcal{B}}$ (which is denoted $\_ \models \_$ in our terminology) in an also Church-Rosser and terminating protecting extension $(\Sigma', E \cup D) \supseteq (\Sigma, E)$ in which the state predicates Π have been specified.

What about the specification of the transition relation $\rightarrow_{\mathcal{B}}$? Here is where rewrite theories come in.

Definition 19. *Let* $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$ *be a finitary rewrite theory such that all its rules are of the form*

$$\lambda : (\forall X)\, t \longrightarrow t' \ \textbf{if} \ \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \,, \qquad (\dagger)$$

with $\bigcup_i (vars(p_i) \cup vars(q_i)) \cup \bigcup_j vars(w_j) \cup vars(t') \subseteq vars(t)$*, or more generally, the rules* (†) *are* admissible *in the sense of [10]; that is, any extra variables not in* $vars(t)$ *can only be introduced incrementally by "matching equations"*[5] *in the condition, so that they are all instantiated by matching.*

We call $\mathcal{R}$ recursive *if:*

1. *there exists a* matching algorithm modulo *the equational axioms*[6] A*;*
2. *the equational theory* $(\Sigma, E \cup A)$ *is (ground)* Church-Rosser and terminating modulo A *[19]; and*
3. *the rules* R *are (ground)* coherent *[51] relative to the equations* E *modulo* A*.*

[5] A *matching equation*, denoted $p := q$, in the above condition is such that p is a term involving only constructor symbols so that, for θ a ground irreducible substitution, $\theta(p)$ is also ground irreducible. We allow extra variables in p, but when solving a condition in which q is instantiated by a substitution μ we can instantiate the extra variables in p by matching p to $can_E(\mu(q))$. An example of a matching condition is given in the text that follows.

[6] In the rewriting logic language Maude, the axioms A for which the rewrite engine supports matching modulo are any combination of *associativity*, *commutativity*, and *identity* axioms for different binary operators.

The last condition means that no rewrites are lost by reducing a term to its (unique modulo A) canonical form $can_{E/A}(t)$ with respect to E before applying any of the rules.

Note first of all that if $\mathcal{R}$ is a recursive rewrite theory, then for any kind k the transition relation $\rightarrow^1_{\mathcal{R},k} \subseteq T_{\Sigma/E,k} \times T_{\Sigma/E,k}$ is recursive. Indeed, given $[u], [v] \in T_{\Sigma/E,k}$, by coherence we have

$$u \rightarrow^1_{\mathcal{R},k} v \iff \text{there exists } w \text{ such that } can_{E/A}(u) \rightarrow^1_{\mathcal{R},k} w \text{ and } can_{E/A}(w) = can_{E/A}(v)\,.$$

Therefore, to decide if $[u] \rightarrow^1_{\mathcal{R},k} [v]$ we first reduce u to its canonical form $can_{E/A}(u)$, and then try to match any rule (†) in R to a subterm of $can_E(u)$. For each such matching substitution θ modulo A, we then try to find a substitution ρ modulo A extending θ to the variables in $vars(t') \setminus vars(t)$ (which may not be empty for admissible rules) and such that

$$E \vdash \bigwedge_{i \in I} \rho(p_i) = \rho(q_i) \wedge \bigwedge_{j \in J} \rho(w_j) : s_j\,,$$

which is a decidable problem given the assumption that E is Church-Rosser and terminating. Because of the assumption that the extra variables in $vars(t') \setminus vars(t)$ are all introduced incrementally in "matching equations" in the condition, and the existence of a matching algorithm modulo A, there is only a *finite* number $\rho_1, \ldots, \rho_n$ of substitutions extending θ and satisfying the rule's condition, and can be computed. For example, to compute the successors of the term $f(a)$ by the rule

$$(\forall \{x, y, z\})\, f(x) \longrightarrow g(x, y, z) \textbf{ if } h(y, z) := h(x, b)\,,$$

where h is a binary operator with a commutativity attribute and $h(y, z) := h(x, b)$ is a matching equation, the substitution $\theta = \{x \mapsto a\}$ would be first obtained. Variables y and z would not be then instantiated, but since the rule is admissible and we are assuming that we have an algorithm for matching modulo commutativity, from the instantiation of $h(y, z) := h(x, b)$ with θ it turns out that there are only two possible ways of assigning values to y and z, giving rise to the substitutions $\rho_1 = \{x \mapsto a, y \mapsto a, z \mapsto b\}$ and $\rho_2 = \{x \mapsto a, y \mapsto b, z \mapsto a\}$.

The next states for rule (†) are then effectively describable as the canonical forms of the one-step rewrites in which the subterm $\theta(t)$ of $can_{E/A}(u)$ is replaced by $\rho_i(t')$. Therefore, we have a recursive function $next_{\mathcal{R}} : T_{\Sigma/E \cup A,k} \longrightarrow \mathcal{P}_{\text{fin}}(T_{\Sigma/E \cup A,k})$.

As a consequence, if $\mathcal{R}$ is recursive then $\mathcal{T}(\mathcal{R})_k = (T_{\Sigma/E \cup A,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet)$ is a recursive transition system. In addition, if the extension $(\Sigma', E \cup D) \supseteq (\Sigma, E)$ is protecting with $E \cup D$ Church-Rosser and terminating, then $\mathcal{K}(\mathcal{R}, k)_\Pi$ is a recursive Kripke structure.

The converse also holds. We state and prove this result for recursive Kripke structures, but the result and proof hold *a fortiori* for recursive transition systems.

Theorem 6. *Let $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ be a recursive Kripke structure. Then there is a recursive rewrite theory $\mathcal{R}$ with state predicates Π corresponding to those of $\mathcal{B}$ and a kind k such that $\mathcal{B}$ is isomorphic to $\mathcal{K}(\mathcal{R}, k)_\Pi$.*

Proof. The case when B and the set Π of atomic predicates are finite admits a much simpler proof. We do in detail the hardest case in which both B and Π are countably infinite. Without loss of generality we may assume that $B = \Pi = \mathbb{N}$. Also, without loss of generality we may represent the recursive function $next : B \longrightarrow \mathcal{P}_{fin}(B)$ as a recursive function $next : \mathbb{N} \longrightarrow \mathbb{N}$, using the fact that there is a recursive isomorphism $\mathcal{P}_{fin}(\mathbb{N}) \cong \mathbb{N}$ mapping each finite set $\{n_1, \ldots, n_k\}$ with $n_1 > \cdots > n_k$ to the number $2^{n_1} + \cdots + 2^{n_k}$, and mapping $\emptyset$ to 0 (see [46, Section 5.6]). By using 0 as *false* and $s(0)$ as *true*, we may also assume without loss of generality that the labeling function $L_{\mathcal{B}}$ is represented as a recursive function $label : \mathbb{N} \times \mathbb{N} \longrightarrow$

$\mathbb{N}$. For later use we also include the recursive predicate $even : \mathbb{N} \longrightarrow \mathbb{N}$, mapping odd numbers to 0 and even numbers to $s(0)$, and the "division by two" recursive function $\_/2 : \mathbb{N} \longrightarrow \mathbb{N}$. All these operations determine a computable Σ-algebra structure, say B, on the natural numbers. By the Bergstra-Tucker theorem [2], there is a supersignature $\Sigma' \supseteq \Sigma$ and a finite set E of confluent and terminating equations such that we have an isomorphism $T_{\Sigma'/E}|_\Sigma \cong B$, where $T_{\Sigma'/E}|_\Sigma$ denotes the reduct of $T_{\Sigma'/E}$ as a Σ-algebra. Furthermore, the construction of $T_{\Sigma'/E}$ in [2] ensures that the canonical term algebra $Can_{\Sigma'/E}$, whose data elements are the ground terms in E-canonical form and is isomorphic to $T_{\Sigma'/E}$, has $\mathbb{N}$ (in Peano notation) as its set of underlying elements. That is, we have, not just an isomorphism, but (assuming B is also in Peano notation) an identity of computable Σ-algebras $Can_{\Sigma'/E}|_\Sigma = B$. We now extend the unsorted signature Σ' (whose only sort we may call *Nat*) to a many-sorted[7] signature by adding two new sorts *Set* and *System*, and adding the following signature Ω with a new constant $\emptyset : Set$, and additional new operations:

$$\begin{array}{l}
\langle _ \rangle : Nat \longrightarrow System \\
\{_\} : Nat \longrightarrow Set \\
_ \cup _ : Set \times Set \longrightarrow Set \\
decode : Nat \longrightarrow Set \\
map.s : Set \longrightarrow Set
\end{array}$$

We can then define the following equational theory $(\Sigma' \cup \Omega, E \cup G \cup A)$ with: (i) A the associativity and commutativity axioms for $\_ \cup \_$, together with the axiom of $\emptyset$ as its identity element; and (ii) G the equations (for x, y variables of sort *Nat*, and S a variable of sort *Set*):

$$\begin{array}{l}
decode(x) = \emptyset \textbf{ if } x = 0 \\
decode(x) = map.s(decode(x/2)) \textbf{ if } s(y) := x \wedge even(y) = 0 \\
decode(x) = \{0\} \cup map.s(decode(y/2)) \textbf{ if } s(y) := x \wedge even(y) = s(0) \\
map.s(\emptyset) = \emptyset \\
map.s(\{x\} \cup S) = \{s(x)\} \cup map.s(S)
\end{array}$$

First of all, note that since none of the operations in Ω has sort *Nat* and none of the equations in $A \cup G$ has sort *Nat*, we obviously have $T_{\Sigma' \cup \Omega/E \cup G \cup A}|_{\Sigma'} \cong T_{\Sigma'/E}$. Second, it is easy to show that the equations G are quasi-decreasing modulo A (see [41] for a precise definition of this notion) and therefore terminating modulo A. Third, we have:

Lemma 3. *Whenever we have $\Sigma \cup \Omega$-terms t, u, v such that (viewing the equations E and G as rewrite rules) $t \longrightarrow_{E/A} u \longrightarrow_{G/A} v$, then there exists a $\Sigma \cup \Omega$-term w such that $t \longrightarrow_{G/A} w \longrightarrow^*_{E/A} v$.*

Proof. Observe that this situation can only arise for terms of sort *Set*. Then consider that: (i) any such term is always of the form $\theta(C)$ with C an Ω-term, and $dom(\theta) = \{x_1, \ldots, x_n\}$ a set of variables of sort *Nat* which do not appear repeated in C, furthermore, $\theta(x_i)$ is always a Σ'-term; (ii) rewrites of $\theta(C)$ with E modulo A do not change C, but can only change θ to, say, θ'; (iii) since all left-hand sides in G are Ω-terms, rewrites of $\theta(C)$ with G modulo A always happen inside C; and (iv) since E is confluent and terminating, and the rules G form a strongly deterministic 3-CTRS (see [41]) where the conditions only involve Σ'-terms, if the condition for the application of a rule in G holds for a substitution θ and $\theta \longrightarrow^*_{E/A} \theta'$, then the same condition also holds for θ'. □

[7] By identifying its sorts with kinds, we can view any many-sorted signature as the special case of a signature in membership equational logic with only kinds and no sorts.

As a consequence of Lemma 3, $\longrightarrow_{G/A}$ "quasi-commutes" with $\longrightarrow_{E/A}$ in the sense of [1]. But then, by the fact that both $\longrightarrow_{E/A}$ and $\longrightarrow_{G/A}$ are terminating, and Lemmas 1–2 in [1], we have that $\longrightarrow_{G\cup E/A}$ is terminating.

To see that $\longrightarrow_{G\cup E/A}$ is also confluent, we can proceed as follows. First, note that all conditional critical pairs for G are joinable (the only conditional critical pairs arise for *decode*, and they are all infeasible, since the conditions of the rules for *decode* are mutually exclusive). Therefore, by Theorem 7.3.2 in [41], $\longrightarrow_{G/A}$ is confluent. To obtain the confluence of $\longrightarrow_{G\cup E/A}$ from those of $\longrightarrow_{E/A}$ and $\longrightarrow_{G/A}$, first observe that by Lemma 3 above and Lemma 1 in [1], the relation $(G/A)/(E/A) = \longrightarrow^*_{E/A} \circ \longrightarrow_{G/A} \circ \longrightarrow^*_{E/A}$ is terminating. Note also the trivial identity $\longrightarrow^*_{G\cup E/A} = \longrightarrow^*_{E} \circ ((G/A)/(E/A))^*$. We can now obtain the desired confluence of $\longrightarrow_{G\cup E/A}$ by case analysis on the two rewrites using $\longrightarrow^*_{G\cup E/A}$ that must be joined. The case where both are of the form $\longrightarrow^*_{E/A}$ follows from the confluence of E. The case where one is of the form $\longrightarrow^*_{E/A}$ and another of the general from $\longrightarrow^*_{G\cup E/A}$ follows easily from Lemma 3 and the following additional lemma:

Lemma 4. *Whenever we have $\Sigma\cup\Omega$-terms t, u, v such that $u \;{}_{E/A}\!\longleftarrow t \longrightarrow_{G/A} v$, then there exists a $\Sigma\cup\Omega$-term w such that $u \longrightarrow_{G/A} w \;{}^{*}_{E/A}\!\longleftarrow v$.*

Proof. As in Lemma 3 above, this can only happen if t is of the form $t = \theta(C)$, with C an Ω-term and $dom(\theta) = \{x_1, \ldots, x_n\}$ a set of variables of sort *Nat* which do not appear repeated in C, and $\theta(x_i)$ a Σ'-term for $1 \leq i \leq n$. Since rewrites of $\theta(C)$ with E modulo A do not change C, but can only change θ to, say, θ', in the rewrite $t \longrightarrow_{E/A} u$ there will be exactly one variable, say x_k, such that $\theta(x_k) \longrightarrow_{E/A} r$, so θ' agrees with θ everywhere except for mapping x_k to r, and we have $u = \theta'(C)$. The key point now is to realize that the left-hand side of the rule in G used in the rewrite $t \longrightarrow_{G/A} v$, which is an Ω-term, still matches the context C in the exact same redex position in the term u, with the only difference that the substitution that now has to be used to check the rule's condition (if it has a condition) is θ' instead of θ. But since E is confluent and terminating, and the rules G form a strongly deterministic 3-CTRS where the conditions only involve Σ'-terms, if the condition for the application of a rule in G holds for substitution θ and $\theta \longrightarrow_{E/A} \theta'$, then the same condition also holds for θ'. The desired term w is then the term obtained by rewriting u using the given rule in G at the exact same redex position in C and matching substitution θ'. It is then trivial to show that we also have $v \longrightarrow^*_{E/A} w$. □

Finally, the case where both rewrites are of the form $((G/A)/(E/A))^*$ can be proved by an easy Noetherian induction on the relation $(G/A)/(E/A)$, using Lemma 3 to fill in all the diamonds in the Noetherian induction step.

In summary, therefore, we have constructed a confluent and terminating equational theory $(\Sigma' \cup \Omega, E \cup G \cup A)$ such that $Can_{\Sigma'\cup\Omega/E\cup G\cup A}|_{\Sigma} = B$, where B is the computable algebra corresponding to the state set of our original Kripke structure and including the *next* and *label* functions. So all we have left to do is to extend $(\Sigma' \cup \Omega, E \cup G \cup A)$ to a rewrite theory $\mathcal{R} = (\Sigma' \cup \Omega, E \cup G \cup A, R, \phi)$ which specifies a Kripke structure isomorphic to $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$. This is now easy. The kind/sort of states is *System*, and R consists of the single rule:

$$\langle x \rangle \longrightarrow \langle y \rangle \textbf{ if } \{y\} \cup S := decode(next(x))$$

which, up to the bijective change of representation $\langle x \rangle \mapsto x$, exactly captures the transition relation $\rightarrow_{\mathcal{B}}$. We still need to check that the rules R are ground coherent with respect to the equations. We define ϕ so that all arguments are frozen for all operators, except for the successor function, where $\phi(s) = \emptyset$. Ground coherence is now easy to check, because the only ground terms that can make an R-transition are terms of the form $\langle t \rangle$, with t of sort *Nat*. But

then, of course, if t' is the E-canonical form of t, both $decode(next(t))$ and $decode(next(t'))$ have the same canonical form; which easily yields the desired coherence, since any element $\{w\}$ that we could choose by associative-commutative matching in some partially simplified version of the term $decode(next(t))$ has a canonical form appearing among the elements of the set that is the canonical form of $decode(next(t'))$. □

At the price of allowing infinite signatures and dropping computability, there is a general representability result stating that *any* transition system and *any* Kripke structure can be modeled in rewriting logic. Indeed, given a transition system $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ we can define $\mathcal{R}_{\mathcal{B}}$ with a single kind $State$, $\Sigma_{nil,State} = B$ and rules $a \longrightarrow b$ iff $a \rightarrow_{\mathcal{B}} b$. And if we extend $\mathcal{B}$ to a Kripke structure, the labeling function $L_{\mathcal{B}}$ can be modelled with equations $(a \models p) = true$ if $p \in L_{\mathcal{B}}(a)$, and $(a \models p) = false$ if $p \notin L_{\mathcal{B}}(a)$.

The interesting point, however, is not whether we can or cannot represent any Kripke structure: we always can. Instead, the point is that we have a general way of defining any *recursive* Kripke structure by means of a *finitary* rewrite theory that is Church-Rosser, terminating, and coherent.

As we will see in Section 5, similar remarks apply to simulations as well. Thus, we will have categories of rewrite theories that can represent all transition systems (resp. Kripke structures) and all simulations between them, and categories of recursive rewrite theories that can represent all recursive transition systems (resp. Kripke structures) and all recursive maps between them.

5 Algebraic Simulations

We have already noted that, in order to reason about computational systems, these can be abstractly described by means of transition systems and Kripke structures. As explained in the previous sections, rewriting logic can be used to specify both kinds of structures in a natural and modular way. Our goal now is to study how to relate different rewrite theories and how to lift to this specification level all the previous results about simulations of Kripke structures. For this, we consider four increasingly more general ways of defining simulations for rewrite theories specifying a concurrent system:

1. The easiest way of defining a simulation map for a rewrite theory (Σ, E, R, ϕ) is by means of an *equational abstraction* [38, 39], which consists in simply adding new equations, say E', to get a quotient system specified by $(\Sigma, E \cup E', R, \phi)$.
2. The previous method can be generalized by considering, instead of just theory inclusions $(\Sigma, E) \subseteq (\Sigma, E \cup E')$, arbitrary *theory interpretations* $H : (\Sigma, E) \longrightarrow (\Sigma', E')$ allowing arbitrary transformations on the data representation of states. We gave a presentation of these in [31].
3. A third alternative consists in defining a simulation *map* between rewrite theories $\mathcal{R}$ and $\mathcal{R}'$ directly at the level of their associated Kripke structures by means of *equationally-defined functions*.
4. Finally, the most general case is obtained by defining arbitrary simulations between rewrite theories $\mathcal{R}$ and $\mathcal{R}'$ by means of *rewrite relations*.

For each of the increasingly more general ways above of defining simulations, there are of course associated *correctness conditions* that must be verified. For equational abstractions they are considered in detail in [38, 39] and for theory interpretations in [31]; here we give a more comprehensive account of case (2), and study also the remaining cases (3) and (4).

5.1 Simulation Maps as Equationally-Defined Functions

In this section we spell out the details for the categories mentioned at the end of Section 4.2. Let us first consider transition systems. For that, we define a category **SRWTh** whose objects are pairs $(\mathcal{R}, k)$, with $\mathcal{R}$ a rewrite theory and k a distinguished kind in $\mathcal{R}$. Objects in the subcategory **RecSRWTh** are also pairs $(\mathcal{R}, k)$ but now, since we are interested in recursive structures, we require the rewrite theory $\mathcal{R}$ to be recursive.

What about morphisms? At the end of Section 4.2 we showed that any transition system can be defined in rewriting logic. Likewise, any stuttering map of transition systems (and in particular any non-stuttering one) $h : \mathcal{A} \longrightarrow \mathcal{B}$ can be equationally defined in a protecting extension of $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{R}_{\mathcal{B}}$ by simply adding an equation $h(a) = b$ for each $a \in A$ which is mapped to b by h; therefore, the following definition does not involve any loss of generality.

Definition 20. *A morphism* $(\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$ *in* **SRWTh**, *called an* algebraic stuttering map of transition systems, *is a stuttering map* $h : \mathcal{T}(\mathcal{R}_1)_{k_1} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{k_2}$ *such that there exists a protecting theory extension* (Ω, G) *containing the equational parts of* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *in which* h *can be equationally defined through an operator* $h : k_1' \longrightarrow k_2'$ *(where the primes indicate the corresponding names for the disjoint copies of the kinds).*

Note that we only require the existence of (Ω, G); we do not need to choose any particular such extension to define the category. Morphisms in **RecSRWTh** are defined similarly, but now we further require h to be defined by a finite set of Church-Rosser and terminating equations in a *finitary* extension (Ω, G).

We can now show that the construction defined in Section 3.2 that associates a transition system to a rewrite theory with a chosen kind of states is actually a *functor*. More precisely, we define $\mathcal{T} : \mathbf{SRWTh} \longrightarrow \mathbf{STSys}$ as follows:

- for objects, $\mathcal{T}(\mathcal{R}, k) = \mathcal{T}(\mathcal{R})_k$;
- for morphisms $h : (\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$, $\mathcal{T}(h) = h$.

Let us denote by **RecSTSys** the category whose objects are recursive transition systems and whose morphisms $h : \mathcal{A} \longrightarrow \mathcal{B}$ are stuttering maps of transition systems such that h is recursive; the following result is an immediate consequence of the definitions.

Proposition 7. *The functor* $\mathcal{T}$: **SRWTh** $\longrightarrow$ **STSys** *is surjective on objects, full, and faithful, with the obvious restriction for non-stuttering maps. Similarly,* $\mathcal{T}$: **RecSRWTh** $\longrightarrow$ **RecSTSys** *is surjective on objects up to isomorphism, full, and faithful (again, with the obvious restriction). Graphically:*

$$\begin{array}{ccc} \mathbf{RecSRWTh} & \hookrightarrow & \mathbf{SRWTh} \\ \downarrow{\scriptstyle \mathcal{T}} & & \downarrow{\scriptstyle \mathcal{T}} \\ \mathbf{RecSTSys} & \hookrightarrow & \mathbf{STSys} \end{array}$$

Let us now turn our attention to Kripke structures. For that we need to consider a theory $BOOL_{\models}$ extending $BOOL$ with two new kinds, *State* and *Prop*, and a new operator $\_ \models \_ : State\ Prop \longrightarrow Bool$.

Objects in the category $\mathbf{SRWTh}_{\models}$ will be rewriting logic specifications of Kripke structures, and arrows will define stuttering maps between them. As already explained, we have to specify both the transition system and the semantics of atomic propositions. Therefore, objects in $\mathbf{SRWTh}_{\models}$ will be pairs consisting of a rewrite theory specifying the underlying transition system, and an equational theory specifying the relevant atomic propositions. We will add, however, a third component whose purpose will be to distinguish the chosen kind of states and also to make sure that the theory $BOOL$ remains fixed along simulations. More precisely, objects in $\mathbf{SRWTh}_{\models}$ are given by triples $(\mathcal{R}, (\Sigma', E \cup D), J)$ where:

1. $\mathcal{R} = (\Sigma, E, R, \phi)$ is a rewrite theory specifying the transition system.
2. $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ is a protecting theory extension, containing and protecting also the theory $BOOL$ of Booleans, that defines the atomic propositions satisfied by the states. We define $\Pi \subseteq \Sigma'$ as the subsignature of operators of coarity $Prop$.
3. $J : BOOL_{\models} \longrightarrow (\Sigma', E \cup D)$ is a membership equational theory morphism [35] that selects the distinguished kind of states $J(State)$, and such that: (i) it is the identity when restricted to $BOOL$, (ii) $J(Prop) = Prop$, and (iii) $J(\_ \models \_ : State\ Prop \rightarrow Bool) = \_ \models \_ : J(State)\ Prop \rightarrow Bool$.

As explained in [38, 39] we can assume, without loss of generality, that $\mathcal{R}$ is $J(State)$-deadlock free, that is, that the relation $\rightarrow^1_{\mathcal{R},J(State)}$ is total.

Objects in the subcategory $\mathbf{RecSRWTh}_{\models}$ are also triples $(\mathcal{R}, (\Sigma', E \cup D), J)$ but now we require the rewrite theory $\mathcal{R}$ to be recursive and the protecting extension $(\Sigma', E \cup D) \supset (\Sigma, E)$ to be finitary, Church-Rosser, and terminating.

What about morphisms? Again, any stuttering (and non-stuttering) map of Kripke structures $(\alpha, h) : \mathcal{A} \longrightarrow \mathcal{B}$ can be equationally defined in a protecting extension of $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{R}_{\mathcal{B}}$, so the following definition does not involve any loss of generality.

Definition 21. *A morphism* $(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *in* $\mathbf{SRWTh}_{\models}$, *called an* algebraic stuttering map, *is a pair* (α, h) *such that:*

1. $(\alpha, h) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ *is a stuttering map of Kripke structures.*
2. *There exists a theory extension* (Ω, G) *containing and protecting disjoint copies of* $(\Sigma'_1, E_1 \cup D_1)$ *and* $(\Sigma'_2, E_2 \cup D_2)$ *in which* α *and* h *can be equationally defined through operators* $\alpha : Prop_1 \longrightarrow StateForm_2$ *and* $h : J_1(State)_1 \longrightarrow J_2(State)_2$ *in* Ω*; the subscripts 1, 2 indicate the corresponding names for the disjoint copies of the kinds, and* $StateForm_2$ *is a new kind for representing state formulas over* $Prop_2$.

Note again the existential quantifier for the extension (Ω, G).

Since, by the general representability result, we can always find an extension (Ω, G) in which the functions α and h can be equationally defined, the category is well-defined, because for each composition we can do the same.

The important point is that if α and h are *recursive*, and the structures $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ and $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ are objects in $\mathbf{RecSRWTh}_{\models}$, then by the metaresult of Bergstra and Tucker [2] we can always find a *finitary* extension (Ω, G) that is both protecting of the pieces and Church-Rosser and terminating, and in which both α and h can be specified by means of Church-Rosser and terminating equations. Therefore, we define morphisms in $\mathbf{RecSRWTh}_{\models}$, called *recursive algebraic stuttering maps*, to be pairs (α, h) as before, but now with the extra requirement that both α and h can be defined by means of Church-Rosser and terminating equations in the extension (Ω, G).

Of course, all these constructions can also be applied to non-stuttering simulations, leading to categories $\mathbf{RWTh}_{\models}$ and $\mathbf{RecRWTh}_{\models}$ with subcategory inclusions:

$$\begin{array}{ccc} \mathbf{RecRWTh}_{\models} & \hookrightarrow & \mathbf{RecSRWTh}_{\models} \\ \downarrow & & \downarrow \\ \mathbf{RWTh}_{\models} & \hookrightarrow & \mathbf{SRWTh}_{\models} \end{array}$$

We can now show that the construction defined in Section 4.1 that associates a Kripke structure to a rewrite theory with a chosen kind of states and chosen state predicates is a functor. More precisely, we define $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$ as follows:

– for objects, $\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$;
– for morphisms $(\alpha, h) : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$, $\mathcal{K}(\alpha, h) = (\alpha, h)$.

Now, if we denote with **RecKSMap** the category whose objects are recursive Kripke structures and whose morphisms are stuttering maps $(\alpha, h) : \mathcal{A} \longrightarrow \mathcal{B}$ such that α and h are both recursive functions, the previous discussion can be summarized as the following result:

Proposition 8. *The functor* $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$ *is surjective on objects, full, and faithful, with the obvious restrictions for non-stuttering maps. Similarly,* $\mathcal{K} : \mathbf{RecSRWTh}_{\models} \longrightarrow \mathbf{RecKSMap}$ *is surjective on objects up to isomorphism, full, and faithful (again, with the obvious restrictions). Graphically:*

$$\begin{array}{ccc} \mathbf{RecSRWTh}_{\models} & \hookrightarrow & \mathbf{SRWTh}_{\models} \\ \downarrow \mathcal{K} & & \downarrow \mathcal{K} \\ \mathbf{RecKSMap} & \hookrightarrow & \mathbf{KSMap} \end{array}$$

The fact that $\mathcal{K}$ is surjective on objects, full, and faithful constitutes a *general representability result*, stating that *all* (resp. *all recursive*) Kripke structures and stuttering maps can be represented by rewrite theories and equationally-defined functions (resp. recursive rewrite theories and recursive equationally-defined functions).

An interesting question is how to verify that an algebraic stuttering simulation is correct, that is, identifying a set of proof obligations ensuring that equationally-defined functions α and h define in fact an algebraic stuttering simulation; some such criteria are discussed in Section 6.

5.2 Simulations as Rewrite Relations

The previous construction, though already very general and applicable to many situations, restricts us to work only with functions. This drawback can be avoided by a simple extension of the ideas introduced above. Let us consider only the case of Kripke structures, bearing in mind that everything applies to transition systems as well by just forgetting about the additional structure given by the atomic propositions.

We define a category $\mathbf{SRelRWTh}_{\models}$ whose objects are those of $\mathbf{SRWTh}_{\models}$ and with arrows as described in the following definition.

Definition 22. *A morphism* $(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *in the category* $\mathbf{SRelRWTh}_{\models}$*, called an* algebraic stuttering simulation*, is a pair* (α, H) *such that:*

1. (α, H) *is a stuttering simulation of Kripke structures* $(\alpha, H) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$.
2. *There exists a rewrite theory extension* $\mathcal{R}_3$ *containing and protecting disjoint copies of* $(\Sigma'_1, E_1 \cup D_1, R_1)$ *and* $(\Sigma'_2, E_2 \cup D_2, R_2)$ *in which* α *can be equationally-defined through an operator* $\alpha : Prop_1 \longrightarrow StateForm_2$*, and* H *is defined by rewrite rules involving an operator* $H : J_1(State)_1\ J_2(State)_2 \longrightarrow Bool$ *such that* xHy *iff* $\mathcal{R}_3 \vdash H(x, y) \longrightarrow true$*. Here the subscripts 1, 2 indicate the corresponding names for the disjoint copies of the kinds, and* $StateForm_2$ *is a new kind for representing state formulas over* $Prop_2$.

The subcategory $\mathbf{RecSRelRWTh}_{\models}$ of recursive rewrite theories and *r.e. algebraic stuttering simulations* is defined analogously, but we now require the theory extension $\mathcal{R}_3$ to be finitary and *admissible* in the sense of [10]. That is, $\mathcal{R}_3$ satisfies requirements similar to those of a recursive rewrite theory, but the conditions of the rules can now contain rewrites as long

as the only new variables in their left-hand sides not present in the rule's left-hand side are contained in the right-hand sides of previous rewrite conditions (in a left-to-right order of the conditions), or in matching equational conditions. Note that, due to the Turing-complete nature of rewriting, this is equivalent to requiring the relation H to be r.e.

Remark 4. It is worth mentioning that we only consider recursive functions in $\mathbf{RecSRWTh}_{\models}$, whereas we now allow arbitrary r.e. relations in $\mathbf{RecSRelRWTh}_{\models}$. This seems a natural extension to us, since in general the composition of recursive relations is not recursive, whereas the composition of r.e. relations is r.e.

Let us denote by **RecKSSim** the category of recursive Kripke structures and stuttering simulations $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ such that α is recursive and H is r.e. The forgetful functor $\mathcal{K}$ is extended in the obvious way to the new categories, and we have the following result.

Proposition 9. *With the above definitions,* $\mathcal{K} : \mathbf{SRelRWTh}_{\models} \longrightarrow \mathbf{KSSim}$ *is surjective on objects, full, and faithful, and* $\mathcal{K} : \mathbf{RecSRelRWTh}_{\models} \longrightarrow \mathbf{RecKSSim}$ *is surjective on objects up to isomorphism, full, and faithful. Graphically:*

$$\begin{array}{ccc} \mathbf{RecSRelRWTh}_{\models} & \hookrightarrow & \mathbf{SRelRWTh}_{\models} \\ \downarrow \mathcal{K} & & \downarrow \mathcal{K} \\ \mathbf{RecKSSim} & \hookrightarrow & \mathbf{KSSim} \end{array}$$

This is the most general representability result possible for stuttering simulations as we have defined them. It shows that we can represent both Kripke structures and stuttering simulations in rewriting logic, and can use rewriting logic and membership equational logic to reason about them.

5.3 Some Examples

5.3.1 Semantics of a Functional Language. As mentioned in Section 3.3, a simple functional language called *Fpl* is defined in [23] along with three different semantics: a quite abstract evaluation semantics, a computation semantics, and a more concrete semantics which uses a stack machine.

The executable specification in Maude of those three semantics is described in [50]. The evaluation semantics is very abstract and uninteresting from a transition system point of view: all expressions are evaluated in a single step. However, the other two semantics are much more concrete, so that the evaluation of a single expression requires the execution of several steps. Therefore, it makes sense to study the relationship between the executions in each of them and express their agreement by means of a stuttering simulation.

A state of the stack machine, using Maude syntax, is a triple `< ST, rho, e >`, where `ST` is a stack of values, `rho` is an environment assigning values to variables, and `e` is an expression. A state for the computation semantics is a pair `< rho, e >`, with `rho` an environment and `e` an expression. The transition relations

`< ST, rho, e >` $\longrightarrow$ `< ST', rho', e'>` and `< rho, e >` $\longrightarrow$ `< rho',e' >`

defined in [23] were translated to rewriting logic in [50]. They appear in Figures 2 and 3, using Maude notation. Following [50], the right-hand side of the transition relation in the computation semantics, which only contains the numerical value in [23], is extended so as to be able to define a transition system. Unlike [23, 50], we do not consider functions.

These definitions give rise to two transition systems, $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ and $\mathcal{C} = (C, \rightarrow_{\mathcal{C}})$, for the stack machine and the computation semantics respectively. To prove the correctness of the stack machine implementation relative to the computation semantics we show that there exists a recursive algebraic stuttering simulation of transition systems $h : \mathcal{S} \longrightarrow \mathcal{C}$.

Intuitively, `< empty, rho, e >`, where `empty` is used both to represent the empty stack and the environment that associates no value to any of the variables, should be related to `< rho, e >`. Consider this derivation:

```
< empty, empty, 2 + 3 > →S < empty, empty, 2 . 3 . + >
                        →S < 2, empty, 3 . + >
                        →S < 3 . 2, empty, + >
                        →S < 5, empty, empty >
```

The second, third, and fourth states in the derivation carry exactly the same information as the first one, though in a different order. The rules used to reach them are examples of what are called *analysis rules* in [23]. It seems appropriate, then, to relate them to the same state as the first one, namely `< empty, 2 + 3 >`. The situation is different for the last state: some information has been lost, and it seems more appropriate to relate this state to `< empty, 5 >`. This last step is an example of an *application rule*.

So we define $h : \mathcal{S} \longrightarrow \mathcal{C}$ by $h(a) =$ `< rho, e >` if a can be obtained from `< empty, rho, e >` by zero or more applications of the analysis rules for the stack machine together with `Valm` and `Locm2`. Note that h is a function precisely because not all of the rules can be applied. Also, h is partial: it is only defined for reachable states, which constitute a full substructure of $\mathcal{S}$ where h is total (recall the discussion before Definition 5 on page 6).

Alternatively, by "undoing" the steps taken by the rules, h can be defined by means of the following set of equations.

```
eq [Base] : h(< empty, rho, e>) = < rho, e > .
eq [Opm1] : h(< ST, rho, e . e' . op . C >) = h(< ST, rho, e op e' . C >) .
eq [Opm1] : h(< ST, rho, be . be' . bop . C >) = h(< ST, rho, be bop be' . C >) .
eq [Ifm1] : h(< ST, rho, be . if(e, e') . C >) =
            h(< ST, rho, If be Then e Else e' . C >) .
eq [Locm1] : h(< ST, rho, e. <x, e'> . C >) =
             h(< ST, rho, let x = e in e' . C >) .
eq [Notm1] : h(< ST, rho, be . not. C >) = h(< ST, rho, Not be . C >) .
eq [Eqm1] : h(< ST, rho, e . e' . equal . C >) =
            h(< ST, rho, Equal(e, e') . C >) .
eq [Locm2] : h(< ST, (x, v) . rho, e . pop . C >) =
             h(< v . ST, rho, < x, e > . C >) .
ceq [Valm] : h(< v . ST, rho, C >) = h(< ST, rho, v . C >) if not(enabled(C)) .
ceq [Valm] : h(< bv . ST, rho, C >) = h(< ST, rho, bv . C >) if not(enabled(C)) .
```

The auxiliary predicate `enabled` used in `Valm` checks that none of the other equations can be applied.

Lemma 5. *If* $h($`< ST, rho, e . C >`$) =$ `< rho, e' >`*, then there exists a position* p *in* `e'` *such that* $\mathtt{e}'|_p = \mathtt{e}$ *and, if* `e` *is not a value, then it is a subexpression that can be reduced in* `e'` *in the next step with the rules of the computation semantics.*

Proof. Note that the transition relation $\rightarrow_{\mathcal{S}}$ is deterministic and that, given a state `< ST, rho, C >`, there is a single way of undoing all the steps to reach a state of the form `< empty, rho, e >`. Therefore, for the purpose of the proof we consider the equations defining h to be oriented rules and proceed by induction on the number of steps used to reach `< rho, e' >`.

– Computation semantics for *Fpl* arithmetic expressions.

```
rl [VarRc] : < rho, x > => < rho, rho(x) > .
rl [OpRc] : < rho, v op v' > => < rho, Ap(op,v,v') > .
crl [OpRc] : < rho, e op e' > => < rho', e'' op e' >
             if < rho, e > => < rho', e'' > .
crl [OpRc] : < rho, e op e' > => < rho', e op e'' >
             if < rho, e' > => < rho', e'' > .
crl [IfRc] : < rho, If be Then e Else e' > => < rho', If be' Then e Else e' >
             if < rho, be > => < rho', be' > .
rl [IfRc] : < rho, If T Then e Else e' > => < rho, e > .
rl [IfRc] : < rho, If F Then e Else e' > => < rho, e' > .
crl [LocRc] : < rho, let x = e in e' > => < rho', let x = e'' in e' >
              if < rho, e > => < rho', e'' > .
rl [LocRc] : < rho, let x = v in e' > => < rho, e'[v / x] > .
```

– Computation semantics for *Fpl* Boolean expressions.

```
rl [BVarRc] : < rho, bx > => < rho, rho(bx) > .
rl [BOpRc] : < rho, bv bop bv' > => < rho, Ap(bop,bv,bv') > .
crl [BOpRc] : < rho, be bop be' > => < rho', be'' bop be' >
              if < rho, be > => < rho', be'' > .
crl [BOpRc] : < rho, be bop be' > => < rho', be bop be'' >
              if < rho, be' > => < rho', be'' > .
crl [NotRc] : < rho, Not be > => < rho', Not be' >
              if < rho, be > => < rho', be' > .
rl [NotRc] : < rho, Not T > => < rho, F > .
rl [NotRc] : < rho, Not F > => < rho, T > .
crl [EqRc] : < rho, Equal(e,e') > => < rho, Equal(e'',e') >
             if < rho, e > => < rho', e'' > .
crl [EqRc] : < rho, Equal(e,e') > => < rho, Equal(e,e'') >
             if < rho, e' > => < rho', e'' > .
crl [EqRc] : < rho, Equal(v,v') > => < rho, T > if v = v' .
crl [EqRc] : < rho, Equal(v,v') > => < rho, F > if v =/= v'.
```

Fig. 2. Semantics rules for *Fpl* computation semantics.

When the number of steps is 1 we have $h(\texttt{< empty, rho, e >}) \rightarrow \texttt{< rho, e >}$ and the result is trivial. Assume that n is greater than 1; we distinguish cases according to the equation (seen as a rule) used for the first step.

– If the first equation `Opm1` has been applied,

$$h(\texttt{< ST, rho, e1 . e2 . op . C >}) \rightarrow h(\texttt{< ST, rho, e1 op e2 . C >})\,.$$

By induction hypothesis, there is a position p such that $\texttt{e}'|_p$ is `e1 op e2` and then our required position is $p.1$. In addition, since `e1 op e2` is not a value it can be reduced, which implies that `e'` is actually `e1 op e2` and thus `e1` can also be reduced if it is not a value. The same reasoning applies to the other `Opm1`, `Ifm1`, `Notm1`, and `Eqm1` equations.

– If `Locm1` has been applied,

$$h(\texttt{< ST, rho, e1 . < x, e2 > . C >}) \rightarrow h(\texttt{< ST, rho, let x = e1 in e2 . C >})\,.$$

By induction hypothesis, $\texttt{e}'|_p$ is `let x = e1 in e2` and we can take $p.1$ as the desired position.

– Analysis rules for the stack machine.

```
rl [Opm1] : < ST, rho, e op e' . C > => < ST, rho, e . e' . op . C > .
rl [Opm1] : < ST, rho, be op be' . C > => < ST, rho, be . be' . bop . C > .
rl [Ifm1] : < ST, rho, If be Then e Else e' . C > =>
            < ST, rho, be . if(e, e') . C > .
rl [Locm1] : < ST, rho, let x = e in e' . C > =>
             < ST, rho, e . < x, e' > . C > .
rl [Notm1] : < ST, rho, Not be . C > => < ST, rho, be . not . C > .
rl [Eqm1] : < ST, rho, Equal(e, e') . C > => < ST, rho, e . e' . equal . C > .
```

– Application rules for the stack machine.

```
rl [Opm2] : < v' . v . ST, rho, op . C > => < Ap(op,v,v') . ST, rho, C > .
rl [Opm2] : < bv' . bv . ST, rho, bop . C > => < Ap(bop,bv,bv') . ST, rho, C > .
crl [Varm] : < ST, rho, x . C > => < v . ST, rho, C >
             if v := lookup(rho,x) .
crl [Varm] : < ST, rho, bx . C > => < bv . ST, rho, C >
             if bv := lookup(rho,bx) .
rl [Valm] : < ST, rho, v . C > => < v . ST, rho, C > .
rl [Valm] : < ST, rho, bv . C > => < bv . ST, rho, C > .
rl [Notm2] : < T . ST, rho, not . C > => < F . ST, rho, C > .
rl [Notm2] : < F . ST, rho, not . C > => < T . ST, rho, C > .
crl [Eqm2] : < v . v' . ST, rho, equal . C > => < T . ST, rho, C >
             if v = v' .
crl [Eqm2] : < v . v' . ST, rho, equal . C > => < F . ST, rho, C >
             if v =/= v' .
rl [Ifm2] : < T . ST, rho, if(e, e') . C > => < ST, rho, e . C > .
rl [Ifm2] : < F . ST, rho, if(e, e') . C > => < ST, rho, e' . C > .
rl [Locm2] : < v . ST, rho, < x, e > . C > => < ST, (x,v) . rho, e . pop . C > .
rl [Pop] : < ST, (x,v) . rho, pop . C > => < ST, rho, C > .
```

Fig. 3. Semantics rules for *Fpl* stack machine.

– For `Locm2`,

$$h(\texttt{< ST, (x,v). rho, e . pop . C >}) \to h(\texttt{< v . ST, rho, < x, e > . C >}) \to h(\texttt{< ST, rho, v . < x, e > . C >}) \to h(\texttt{< ST, rho, let x = v in e . C >}).$$

By induction hypothesis, $\mathtt{e}'|_p$ is `let x = v in e` and we can take $p.3$.

– For `Valm`, we have $h(\texttt{< v . ST, rho, e . C >}) \to h(\texttt{< ST, rho, v . e . C >})$. Now, the only rules that can be applied to the last term are `Opm1` and `Eqm1`; `Valm` is not a valid alternative because it would give rise to three consecutive expressions, which is not possible since there are no ternary operators. Assume that `Eqm1` is used (analogously for the two `Opm1` rules): `C` is of the form `equal . C'` and $h(\texttt{< ST, rho, v . e . equal . C' >}) \to h(\texttt{< ST, rho, Equal(v,e) . C' >})$. Now, by induction hypothesis, $\mathtt{e}'|_p$ is `Equal(v,e)` and the required position is $p.2$. □

Theorem 7. *The function $h : \mathcal{S} \longrightarrow \mathcal{C}$ defines a recursive algebraic stuttering simulation of transition systems.*

Proof. We will use the finitary characterization of stuttering simulations given in Definition 15. Since h is a (partial) function, it is only necessary to define a function $\mu : S \times C \longrightarrow \mathbb{N}$,

which we do by defining $\mu(a, c)$ as the length of the longest path starting at a that only uses analysis rules, `Valm`, or `Locm2`.

Assume that $a \rightarrow_{\mathcal{S}} a'$ and that $h(a) = c$. If a' has been obtained by applying an analysis rule, `Valm`, or `Locm2`, then $h(a') = c$ and $\mu(a', c) < \mu(a, c)$. Otherwise, we must find a c' such that $c \rightarrow_{\mathcal{C}} c'$ and $h(a') = c'$; we distinguish cases depending on the rule used.

- `Opm2`. In this case, a is `< v' . v . ST, rho, op . C >` and therefore $h(a)$ is equal to $h($`< ST, rho, v op v' . C >`$) =$ `< rho, e >` where, by Lemma 5, there is a position p in `e` such that $\mathtt{e}|_p$ is `v op v'` and `v op v'` is a subexpression of `e` that can be reduced by the rules of the computation semantics in the next step. We can then take c' to be `< rho, e[Ap(op,v,v')]`$_p$`>`. Similarly for `Notm2`, `Eqm2`, and `Ifm2`.
- `Varm`. Then a must be equal to `< ST, rho, x . C >` and $h(a)$ to `< rho, e >` with $\mathtt{e}|_p = \mathtt{x}$ an expression in `e` that can be reduced. Thus, we can take c' to be `< rho, e[rho(x)]`$_p$`>`.
- `Pop`. In this case a must be of the form `< ST, (x,v) . rho, pop . C >`. The only equation that applies to $h(a)$ is `Valm`, and therefore there exists a value `v'` such that `ST` is `v' . ST'`. Applying now the other equations it turns out that $h(a)$ is equal to $h($`< ST', rho, let x = v in v' . C >`$)$, that has to be equal to `< rho, e >` with $\mathtt{e}|_p =$ `let x = v in v'` a subexpression of `e` that can be reduced. We now take c' to be `< rho, e[v']`$_p$`>`.

Therefore, the conditions of Definition 15 are satisfied and, by Theorem 5, h is a stuttering simulation of transition systems. It is also clear that the equations above defining h are Church-Rosser and terminating, and therefore h is a recursive algebraic stuttering simulation of transition systems. □

Note that h is not a bisimulation. In the computation semantics, for an expression `e op e'` we can choose whether to evaluate `e` before `e'` or vice versa, whereas the stack machine always evaluates `e` first. That means that, for example, the transition

$$\texttt{< empty, (1 + 2) + (3 + 4) >} \rightarrow_{\mathcal{C}} \texttt{< empty, (1 + 2) + 7 >}$$

cannot be simulated by the stack machine.

The simulation h can be lifted to the level of Kripke structures. For that, we consider as the set AP of atomic propositions the set of all possible values, and extend the transition systems $\mathcal{S}$ and $\mathcal{C}$ with $L_{\mathcal{S}}($`< empty, rho, v >`$) = L_{\mathcal{S}}($`< v, rho, empty >`$) = \{$`v`$\}$, $L_{\mathcal{C}}($`< rho,v >`$) = \{$`v`$\}$ and both $L_{\mathcal{S}}(a)$ and $L_{\mathcal{C}}(c)$ are empty otherwise. Then, by the preservation result in Theorem 3, for all expressions `e` and environments `rho`,

$$\mathcal{C}, \texttt{< rho, e >} \models \mathbf{AF}\texttt{v} \quad \Longrightarrow \quad \mathcal{S}, \texttt{< empty, rho, e >} \models \mathbf{AF}\texttt{v}\,.$$

That is, $\mathcal{S}$ is a correct implementation of $\mathcal{C}$.

5.3.2 A Communication Protocol Example. If a communication mechanism does not provide reliable, in-order delivery of messages, it may be necessary to generate this service using the given unreliable basis. In [34] it is shown how this might be done, and we slightly adapt here the proposed solution. Both the sender and the receiver keep a counter for synchronization purposes; the sender releases a message together with such number (rule `send`) and does not send another message until it receives an acknowledgment by the receiver. This is the complete Maude module providing the rules implementing this idea.

```
mod PROTOCOL is
  protecting NAT .
```

```
protecting QID .

sorts Object Msg Config .
subsort Object Msg < Config .

op null : -> Config .
op __ : Config Config -> Config [assoc comm id: null] .

sorts Elem List Contents .
subsort Elem < Contents List .

op empty : -> Contents .
ops a b c : -> Elem .
op nil : -> List .
op _:_ : List List -> List [assoc id: nil] .

op to:_(_,_) : Qid Elem Nat -> Msg .
op to:_ack_ : Qid Nat -> Msg .

op <_: Sender | rec:_, sendq:_ , sendbuff:_, sendcnt:_ > :
   Qid Qid List Contents Nat -> Object .

--- rec is the receiver, sendq is the outgoing queue, sendbuff
--- is either empty or the current data, sendcnt is the sender
--- sequence number

op <_: Receiver | sender:_, recq:_, reccnt:_ > :
   Qid Qid List Nat -> Object .

--- sender is the sender, recq is the incoming queue,
--- and reccnt is the receiver sequence number

vars S R : Qid .     vars M N : Nat .
var E : Elem .       var L : List .
var C : Contents .

--- rules for the sender

rl [produce-a] :
   < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
   < S : Sender | rec: R, sendq: L : a, sendbuff: a, sendcnt: N + 1 > .
rl [produce-b] :
   < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
   < S : Sender | rec: R, sendq: L : b, sendbuff: b, sendcnt: N + 1 > .
rl [produce-c] :
   < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
   < S : Sender | rec: R, sendq: L : c, sendbuff: c, sendcnt: N + 1 > .
rl [send] : < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N >
         => < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N >
            (to: R (E,N)) .
rl [rec-ack] :
   < S : Sender | rec: R, sendq: L, sendbuff: C, sendcnt: N >
   (to: S ack M) =>
   < S : Sender | rec: R, sendq: L,
                  sendbuff: (if N == M then empty else C fi),
```

```
                    sendcnt: N > .

  --- rule for the receiver

  rl [receive] :
     < R : Receiver | sender: S, recq: L, reccnt: M > (to: R (E,N)) =>
     (if N == M + 1 then
        < R : Receiver | sender: S, recq: L : E, reccnt: M + 1 >
      else
        < R : Receiver | sender: S, recq: L, reccnt: M >
      fi)
     (to: S ack N) .
endm
```

Under reasonable fairness assumptions (namely, the receiver won't wait indefinitely for an available message), these definitions will generate a reliable, in-order communication mechanism from an unreliable one. The fault modes of the communication channel can be explicitly modeled as in the following Maude module.

```
mod PROTOCOL-FAULTY is
  including PROTOCOL .

  op <_: Destroyer | sender:_, rec:_, cnt:_, cnt':_, rate:_ > :
     Qid Qid Qid Nat Nat Nat -> Object .

  var M : Msg .       vars K N N' : Nat .
  var E : Elem .      vars S R D : Qid .

  rl [destroy1] :
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
     (to: R (E,N)) =>
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N', rate: K > .
  rl [destroy2] :
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
     (to: R ack N) =>
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N', rate: K > .
  rl [limited-injury] :
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': 0, rate: K > =>
     < D : Destroyer | sender: S, rec: R, cnt: s(N), cnt': K, rate: K > .
endm
```

Messages may be destroyed by objects of class `Destroyer`. The first counter represents the identifying number of the messages they can destroy, and the second one represents how many more messages with that number they are still allowed to remove. The attribute `rate` is used to reset the value of `cnt'` once it reaches zero.

To check if messages are delivered in the correct order, we define a state predicate `prefix(S,R)` that holds for a sender `S` and receiver `R` whenever the queue associated to `R` is a prefix of that associated to `S`. This is done, both for `PROTOCOL` and `PROTOCOL-FAULTY`, by means of the following operator:

```
  op prefix : Qid Qid -> Prop .

  var CO : Config .
```

```
eq < S : Sender | rec: R, sendq: L1 : L2, sendbuff: C, sendcnt: N >
   < R : Receiver | sender: S, recq: L1, reccnt: M >
   CO |= prefix(S, R) = true .
```

The new system will satisfy the same correctness conditions as PROTOCOL regardless of messages being destroyed or arriving out of order. In particular, the initial state

```
eq init = < 'A : Sender | rec: 'B, sendq: nil, sendbuff: empty, sendcnt: 0 >
          < 'B : Receiver | sender: 'A, recq: nil, reccnt: 0 > .
```

should satisfy the formula $\mathbf{AG}\,\mathtt{prefix}(\mathtt{'A}, \mathtt{'B})$. To prove it we define a stuttering simulation

$$H : \mathcal{K}(\mathtt{PROTOCOL\text{-}FAULTY}, \mathtt{Config})_{\Pi} \longrightarrow \mathcal{K}(\mathtt{PROTOCOL}, \mathtt{Config})_{\Pi}\,,$$

where Π only contains the state predicate prefix. Given configurations (states) a and b respectively in PROTOCOL-FAULTY and PROTOCOL, aHb iff:

- b is obtained from a by removing all objects of class Destroyer, or
- there exists a' such that $a'Hb$ and a can be obtained from a' by the rules that belong only to PROTOCOL-FAULTY.

We can define H as a rewrite relation in an admissible rewrite theory extending PROTOCOL and PROTOCOL-FAULTY. In this specification, the kinds of states have been renamed as Config1 and Config2, and removeD and messages are auxiliary functions that, given a configuration, remove all objects of class Destroyer and return all messages in it, respectively.

```
op H : Config1 Config2 -> Bool .

op undo-d1 : Qid Elem Nat -> Msg .
op undo-d2 : Qid Nat -> Msg .
op undo-injury : -> Msg .

rl [destroy1-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N' > undo-d1(R,E,N) =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N') > (to: R (E,N)) .
rl [destroy2-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N' > undo-d2(R,N) =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N') > (to: R ack N) .
rl [limited-injury-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: s(N), cnt': K, rate: K >
                                                    undo-injury =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': 0 > .

crl H(C, C') => true if removeD(C) = C' .
crl H(C, C') => true if M (to: R (E,N)) := messages(C') /\
                        (to: R (E,N)) in messages(C) = false /\
                        C undo-d1(R,E,N) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if M (to: R ack N) := messages(C') /\
                        (to: R ack N) in messages(C) = false /\
                        C undo-d2(R,E) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if C undo-injury => C'' /\ H(C'', C') => true .
```

Theorem 8. *$H : \mathcal{K}(\mathtt{PROTOCOL\text{-}FAULTY}, \mathtt{Config})_{\Pi} \longrightarrow \mathcal{K}(\mathtt{PROTOCOL}, \mathtt{Config})_{\Pi}$ is an r.e. algebraic stuttering simulation.*

Proof. H so defined clearly preserves the atomic propositions, because the value of the sender's and the receiver's queues, `sendq` and `recq`, are not changed. Let R_1 be the set of rules in `PROTOCOL` and let R_2 be those added in `PROTOCOL-FAULTY`, and define $\mu(a, b)$ to be the length of the longest rewrite sequence starting at a using rules in R_2. Note that this is well-defined because R_2 is terminating. If aHb and $a \rightarrow^1_{R_1} a'$ then, since the `Destroyer` class plays no role in R_1, it is $b \rightarrow^1_{R_1} b'$ with $a'Hb'$. And if $a \rightarrow^1_{R_2} a'$, by definition of H it is $a'Hb$ and $\mu(a', b) < \mu(a, b)$. Because of rule `send` there are no deadlocks in the system and hence these two alternatives cover all possibilities. Therefore, by Theorem 5, H is a stuttering Π-simulation. And since the rules above defining H in rewriting logic are admissible, H is an r.e. algebraic stuttering simulation. □

By Theorem 3, the existence of H shows that if $\mathbf{AG}\,\texttt{prefix('A, 'B)}$ holds in `PROTOCOL` then it must also hold in `PROTOCOL-FAULTY`. But we have not proved yet that the property holds in `PROTOCOL`. For that, in [39, 42] a finite abstraction

$$G : \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_{\Pi} \longrightarrow \mathcal{K}(\texttt{ABS-PROTOCOL}, \texttt{Config})_{\Pi}$$

is defined for the case of two processes, and the fact that messages are delivered in order is model checked in `ABS-PROTOCOL`; by composing G with H this also proves that the same property is true in `PROTOCOL-FAULTY`.

5.3.3 A Simple Pipelined Machine. We consider here an example adapted from [27] about the correctness of a pipelined machine.

The specification used to prove the correctness is an instruction set architecture (ISA). An ISA state is a triple consisting of a program counter, a register file, and a memory where (only) instructions are stored. Instructions consist of an operation code, the target register to which the instruction applies, and two source registers; there are operation codes for addition, subtraction, and a "do nothing" instruction called noop. In each step, the machine executes the instruction pointed to by the program counter, and updates the program counter and the register file accordingly.

We can represent an ISA machine in rewriting logic by means of a protecting theory extension $\mathcal{R}_{\text{ISA}}$ of the natural numbers used to represent the registers and their values. To represent all elements of the machine we need the following operators:

```
subsort Register < RegFile .

op {_,_,_} : ProgramCounter RegFile Memory -> StateISA .
op inst : OpCode Nat Nat Nat -> Instruction .
ops add sub noop : -> OpCode .
op reg : Nat Nat -> Register .
op _;_ : RegFile RegFile -> RegFile [assoc comm] .
op update : RegFile Instruction -> RegFile .
op cell : Nat Instruction -> MemCell .
op _:_ Memory Memory -> Memory .
op applyOp : OpCode Nat Nat -> Nat .
op getValue : RegFile Nat -> Nat .
```

Then, its behavior is governed by the rule

```
rl { PC, RF, cell(PC, I) : M } => { PC + 1, update(RF, I), cell(PC, I) : M } .
```

and the equations

```
eq update(reg(R1, V1) ; RF, inst(OC, R1, R2, R3)) =
   reg(R1, applyOP(OC, getValue(reg(R1, V1) ; RF, R2),
                       getValue(reg(R1, V1) ; RF, R3))) ; RF .
eq getValue(reg(R, V) ; RF, R) = V .
eq applyOp(+, N1, N2) = N1 + N2 .
eq applyOp(*, N1, N2) = N1 * N2 .
```

Following [27], we focus on the transition relation, forgetting about the atomic propositions.

The ISA is implemented by a micro architecture (MA) machine, a pipelined machine with three stages. An MA machine state is a 5-tuple consisting of a program counter, a register file, a memory, and two latches. During the fetch stage, the instruction pointed to by the program counter is stored in the first latch. During the set-up stage, the instruction in the first latch is passed to the second one together with the values for the source registers. In the write-back stage, the instruction in the second latch is executed and the register file is updated.

Again, the MA can be represented in rewriting logic as a theory $\mathcal{R}_{\mathrm{MA}}$ protecting the theory of the natural numbers. The operators needed include the ones introduced above except for the constructor `{_,_,_}`, together with the following ones:

```
subsort Instruction < Latch1 .

op {_,_,_,_,_} : ProgramCounter RegisterFile Memory Latch1 Latch2 -> StateMA .
op empty1 : -> Latch1 .
op empty2 : -> Latch2 .
op latch : OpCode Nat Nat Nat -> Latch2 .
op nextPC : StateMA -> ProgramCounter .
op nextRF : StateMA -> RegisterFile .
op nextM : StateMA -> Memory .
op nextL1 : StateMA -> Latch1 .
op nextL2 : StateMA -> Latch2 .
op stalled : Latch1 Latch2 -> Bool .
```

The behavior of the machine is given by the rule

```
rl S => { nextPC(S), nextRF(S), nextM(S), nextL1(S), nextL2(S) } .
```

where `S` is a variable of sort `StateMA`. If no stall occurs the program counter is incremented; otherwise it remains fixed. A stall happens when both latches are nonempty and the target register of the second latch is one of the source registers in the first one.

```
eq nextPC({ PC, RF, M, L1, L2 }) = PC if stalled(L1, L2) = true .
eq nextPC({ PC, RF, M, L1, L2 }) = PC + 1 if stalled(L1, L2) = false .
eq stalled(empty1, L2) = false .
eq stalled(L1, empty2) = false .
eq stalled(inst(OC, R1, R2, R3), latch(OC', R, N1, N2)) =
   (R == R2) or (R == R3) .
```

The memory remains fixed throughout the execution

```
eq nextM({ PC, RF, M, L1, L2 }) = M .
```

The register file is updated with the value resulting from executing the instruction in the second latch, whenever this is not empty:

```
eq nextRF({ PC, RF, M, L1, empty }) = RF .
eq nextRF({ PC, reg(R, V) ; RF, M, L1, latch(OC, R, N1, N2) }) =
   reg(R, applyOp(OC, N1, N2)) ; RF .
```

If there is no stall, the first latch is updated by fetching from the memory the instruction pointed to by the program counter.

```
ceq nextL1({ PC, RF, cell(PC, I) : M, L1, L2 }) = I
    if stalled(L1, L2) = false .
eq nextL1({ PC, RF, M, L1, L2 }) = L1 if stalled(L1, L2) = true .
```

Similarly, if there is no stall the second latch is updated by passing the instruction in the first one together with the values in the source registers.

```
ceq nextL2({ PC, RF, M, empty1, L2 }) = empty2 if stalled(L1, L2) = false .
ceq nextL2({ PC, RF, M, inst(OC, R1, R2, R3), L2 }) =
    inst(OC, R1, getValue(RF, R2), getValue(RF, R3))
    if stalled(L1, L2) = false .
eq nextL2({ PC, RF, M, L1, L2 }) = empty2 if stalled(L1, L2) = true .
```

We have just given the specifications in rewriting logic of both ISA and MA. Now we want to relate them by means of a recursive algebraic stuttering map of transition systems $(\mathcal{R}_{\mathrm{MA}}, [\mathtt{State}]_{\mathrm{MA}}) \longrightarrow (\mathcal{R}_{\mathrm{ISA}}, [\mathtt{State}]_{\mathrm{ISA}})$. Note the direction of the arrow, from the implementation to the specification.

Instructions in ISA are executed immediately, while in MA they go through three different stages; therefore, the simulation will necessarily be a stuttering one. Given an MA machine state, to get an ISA state we just have to forget the information in the latches. Note, however, that the program counter in the MA machine points to the next instruction to be fetched, so that now that we are removing the instructions already fetched in the latches we have to decrease the program counter accordingly.

The simulation can then be specified in the disjoint union of the rewrite theories $\mathcal{R}_{\mathrm{ISA}}$ and $\mathcal{R}_{\mathrm{MA}}$. The program counter is appropriately updated by an operator

```
op commit : StateRA -> ProgramCounter .
```

defined by the equations

```
eq commit({ PC, RF, M, empty1, empty2}) = PC .
eq commit({ PC, RF, M, inst(OC, R1, R2, R3), empty2}) = PC - 1 .
eq commit({ PC, RF, M, empty1, latch(OC, R, N1, N2) }) = PC - 1 .
eq commit({ PC, RF, M, inst(OC, R1, R2, R3), latch(OC, R, N1, N2) }) = PC - 2 .
```

Finally, the map is defined through an operator

```
op sim : StateMA -> StateISA .
```

and the equation

```
eq sim({ PC, RF, M, L1, L2 }) = { commit({ PC, RF, M, L1, L2 }, RF, M) } .
```

That this indeed defines a recursive algebraic stuttering map of transition systems follows by adapting the proof in [27].

Theorem 9. *The operator* `sim` *specifies a recursive algebraic stuttering map of transition systems,* $sim : \mathcal{T}(\mathcal{R}_{\mathrm{MA}})_{\mathrm{State}_{\mathrm{MA}}} \longrightarrow \mathcal{T}(\mathcal{R}_{\mathrm{ISA}})_{\mathrm{State}_{\mathrm{ISA}}}$.

5.4 Theoroidal Simulation Maps

At the beginning of Section 5 we indicated that the equational simulations introduced in [38, 39] can be generalized either by considering theory interpretations or, more generally, equationally defined functions or rewrite relations. The previous sections have been devoted to present the more general of these cases. However, it is not always necessary to use that greater generality: there are many interesting examples that can be explained by means of just theory interpretations, as first presented in [31]; to them we turn now our attention.

5.4.1 Generalized Signature Morphisms. The first thing to do is to make precise the meaning of *theory interpretation*. The idea is to use the standard concepts of signature and theory morphism. However, as we shall see in some of the examples below, the usual definition of signature morphism is sometimes not expressive enough. For this reason we introduce the following generalization of the concept of signature morphism in which a kind or an operator can be *erased*.

Definition 23. *Given two membership equational signatures* $\Sigma = (K, \Sigma, S)$ *and* $\Sigma' = (K', \Sigma', S')$*, a* generalized signature morphism $H : \Sigma \longrightarrow \Sigma'$ *is specified by:*

- *partial functions* $H : K \longrightarrow K'$ *and* $H : S \longrightarrow S'$ *such that, for all sorts* $s \in \Sigma$*, if* $H(s)$ *is defined so is* $H([s])$ *and* $H([s]) = [H(s)]$*.*
- *a partial function* H *assigning, to each* $f \in \Sigma_{k_1 \ldots k_n, k}$ *such that* $H(k)$ *is defined, a* Σ'*-term* $H(f)$ *of kind* $H(k)$ *such that* $vars(H(f)) \subseteq \{x_{i_1} : H(k_{i_1}), \ldots, x_{i_m} : H(k_{i_m})\}$*, where* $k_{i_1}, \ldots, k_{i_m}$ *is the (possibly empty) subsequence of* $k_1, \ldots, k_n$ *determined by those* k_i *such that* $H(k_i)$ *is defined. Otherwise, if* $H(k)$ *is undefined, so is* $H(f)$*.*

All standard constructions and results about signature morphisms apply to these generalized ones as well. Given $H : \Sigma \longrightarrow \Sigma'$ and a Σ'-algebra A, its reduct $U_H(A)$ over Σ is defined by:

- For each kind k, $U_H(A)_k = A_{H(k)}$ if $H(k)$ is defined; otherwise $U_H(A)_k = \{*\}$.
- For each sort s, $U_H(A)_s = A_{H(s)}$ if $H(s)$ is defined; otherwise $U_H(A)_s = \{*\}$.
- For each operator $f : k_1 \ldots k_n \longrightarrow k$, if $k_{i_1}, \ldots, k_{i_m}$ is the subsequence of those kinds in $k_1, \ldots, k_n$ for which H is defined,

$$U_H(A)_f(a_1, \ldots, a_n) = A_{H(f)}(a_{i_1}, \ldots, a_{i_m})\,;$$

otherwise

$$U_H(A)_f(a_1, \ldots, a_n) = *\,.$$

Given generalized signature morphisms $F : \Sigma \longrightarrow \Sigma'$ and $G : \Sigma' \longrightarrow \Sigma''$, their composition $G \circ F$ is defined for a kind k only if both $F(k)$ and $G(F(k))$ are defined, and then it is $(G \circ F)(k) = G(F(k))$; analogously for a sort s and an operator f.

Generalized signature morphisms can also be extended homomorphically to terms, but note that for t of kind k, if $H(k)$ is not defined then $H(t)$ is not defined either. This translation extends to formulas in the expected way, where by convention $H(t = t') = H(t : s) = \top$ if H is not defined for the kind of t (which is the same as that of t' and s). Our desired general notion of "theory interpretation" is then captured by the following:

Definition 24. *Given two membership equational theories* (Σ, E) *and* (Σ', E')*, a* generalized theory morphism *(resp. a* generalized theory morphism with initial semantics*)* $H : (\Sigma, E) \longrightarrow (\Sigma', E')$ *is a generalized signature morphism* $H : \Sigma \longrightarrow \Sigma'$ *such that for each* $\varphi \in E$*,* $E' \vdash H(\varphi)$ *(resp.* $T_{\Sigma'/E'} \models H(\varphi)$*).*

Note that, since $T_{\Sigma'/E'} \models E'$, each generalized theory morphism is *a fortiori* a generalized theory morphism with initial semantics, but not conversely. For example, if (Σ, E) is the theory with one sort, *Nat*, a binary operator $+$, and the equation $(\forall\{x, y : Nat\})\, x+y = y+x$, (Σ', E') is the usual equational definition of addition in Peano arithmetic, and H is the obvious signature inclusion, then we have $T_{\Sigma'/E'} \models (\forall\{x, y : Nat\})\, x + y = y + x$, but $E' \not\vdash (\forall\{x, y : Nat\})\, x + y = y + x$.

Again, generalized theory morphisms compose and, together with membership equational theories, give rise to a category $\mathbf{GTh}_{\mathrm{MEL}}$.

The new feature of generalized signature morphisms, which is inherited by generalized theory morphisms, is that kinds and operators can be removed. This could have been "implemented" using the standard notion of theory morphism in the following alternative manner:

Proposition 10. *A generalized theory morphism $H : T \longrightarrow T'$ is the same thing as an ordinary theory morphism $H : T \longrightarrow T' \oplus ONE$, where $\oplus$ denotes coproduct of theories, and ONE is a theory with a single kind $[One]$ and sort One, a constant $*$ of that kind, and the equation $(\forall\{x\})\, x = *$.*

Proof. Leaving a kind or sort undefined in a generalized signature morphism corresponds respectively to mapping it to $[One]$ or *One* in $T' \oplus ONE$, while leaving the image of an operator undefined corresponds to mapping it to the term $*$. □

Note that there is an equivalence of categories between the models of T' and those of $T' \oplus ONE$, because, even though we have introduced a new kind $[One]$, all its elements are collapsed by the equation $(\forall\{x\})\, x = *$ to the constant $*$ and can play no distinguished role.

Example. A special case of generalized theory morphisms are the projections from n-tuples to m-tuples, with $m < n$. Consider a theory *3-TUPLE* for triples with kinds *3-Tuple*, *Elt@x*, *Elt@y*, *Elt@z*, an operator $\langle \_, \_, \_ \rangle$: *Elt@x Elt@y Elt@z* $\longrightarrow$ *3-Tuple*, projection operators p_1, p_2, and p_3, and the obvious equations. Similarly, the theory *2-TUPLE* has kinds *2-Tuple*, *Elt@x*, *Elt@z*, an operator $\langle \_, \_ \rangle$: *Elt@x Elt@z* $\longrightarrow$ *2-Tuple*, corresponding projection operators p_1 and p_2, and the equations for pairing. Projecting from a triple to a pair by projecting out the second component can be represented by the generalized theory morphism H : *3-TUPLE* $\longrightarrow$ *2-TUPLE* mapping the kinds *Elt@x* and *Elt@z* to themselves, *3-Tuple* to *2-Tuple*, and the operator $\langle \_, \_, \_ \rangle$ to the term $\langle x_1 : Elt@x, x_3 : Elt@z \rangle$; the images of the kind *Elt@y* and the operator p_2 are left undefined.

5.4.2 Simulation Maps as Generalized Theory Morphisms. We now have all the ingredients needed to define a category $\mathbf{SRWThHom}_{\models}$ in which stuttering maps are specified by theory interpretations. Objects in $\mathbf{SRWThHom}_{\models}$ are the same as those in $\mathbf{SRWTh}_{\models}$ (see Section 5.1), that is, triples $(\mathcal{R}, (\Sigma', E'), J)$ satisfying all requirements on page 22. A morphism

$$H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$$

in $\mathbf{SRWThHom}_{\models}$ is a *generalized signature morphism* $H : \Sigma_1 \cup \Pi_1 \longrightarrow \Sigma_2 \cup \Pi_2$ such that:

1. $H \circ J_1 = J_2$ (so that *BOOL* is preserved and states in $\mathcal{R}_1$ are mapped to states in $\mathcal{R}_2$).
2. $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ is a generalized morphism of membership equational theories with initial semantics, so that we have a unique Σ_1-homomorphism

$$\eta^H : T_{\Sigma_1/E_1} \longrightarrow U_H(T_{\Sigma_2/E_2}) : [t] \mapsto [H(t)]\,.$$

3. (Preservation of transitions.) $\eta^H_{J_1(State)} : \mathcal{T}(\mathcal{R}_1)_{J_1(State)} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{J_2(State)}$, the component corresponding to the kind $J_1(State)$ in η^H mapping $[t]$ to $[H(t)]$, is a stuttering map of transition systems.
4. (Preservation of predicates.) For each $t \in T_{\Sigma_1, J_1(State)}$ and state predicate $p(u_1, \ldots, u_n)$, $H(p(u_1, \ldots, u_n))$ is a state predicate and we have

$$E_2 \cup D_2 \vdash (H(t) \models H(p(u_1, \ldots, u_n))) = true \implies E_1 \cup D_1 \vdash (t \models p(u_1, \ldots, u_n)) = true\,.$$

Since H cannot map a state predicate to an arbitrary formula, the problem mentioned in Section 2.3 does not arise and we can analogously construct a subcategory $\mathbf{SRWThHom}^{\mathrm{str}}_{\models}$ of strict maps. The definition is exactly the same except for item (4), where the implication must actually be an equivalence. Similarly, to get a category $\mathbf{RWThHom}_{\models}$ of non-stuttering maps we simply replace condition (3) by the requirement that, for all $t, t' \in T_{\Sigma_1, J_1(\mathrm{State})}$:

$$t \rightarrow^1_{\mathcal{R}_1, J_1(State)} t' \implies H(t) \rightarrow^1_{\mathcal{R}_2, J_2(State)} H(t')\,.$$

That H so constrained indeed gives rise to a map of Kripke structures is shown in Proposition 11 below. For that, let us define a functor $\mathcal{K} : \mathbf{SRWThHom}_{\models} \longrightarrow \mathbf{KSMap}$ as follows:

- for objects, $\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$;
- for morphisms $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ we define $\mathcal{K}(H) = (H|_{\Pi_1}, \eta^H_{J_1(State)})$, where $H|_{\Pi_1}$ is the restriction of H to the state predicates Π_1.

Proposition 11. *With the above definitions,* $\mathcal{K} : \mathbf{SRWThHom}_{\models} \longrightarrow \mathbf{KSMap}$ *is a functor with restrictions* $\mathcal{K} : \mathbf{SRWThHom}^{\mathrm{str}}_{\models} \longrightarrow \mathbf{KSMap}^{\mathrm{str}}$ *and* $\mathcal{K} : \mathbf{RWThHom}_{\models} \longrightarrow \mathbf{KMap}$.

Proof. $\mathcal{K}$ is well-defined on objects, and it is immediate to see that it preserves identities and composition of morphisms; the only thing we need to check is that, for all H, $\mathcal{K}(H)$ is indeed a map of Kripke structures.

Let then $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ be a morphism in $\mathbf{SRWThHom}_{\models}$. By item (3) above, $\eta^H_{J_1(State)} : \mathcal{T}(\mathcal{R}_1)_{J_1(State)} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{J_2(State)}$ is a stuttering map of transition systems. To show preservation of predicates, let $p(u_1, \ldots, u_n) \in L_{\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}|_{H|_{\Pi_1}}}([H(t)])$. By definition of the reduct of a Kripke structure, we have $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}, [H(t)] \models H(p(u_1, \ldots, u_n))$ which, by definition of $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ and condition (4) in the definition of morphisms in $\mathbf{SRWThHom}_{\models}$, implies that $p(u_1, \ldots, u_n)$ belongs to $L_{\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}}([t])$, as required. It is clear that if H belongs to $\mathbf{SRWThHom}^{\mathrm{str}}_{\models}$ the converse is also true and $\mathcal{K}(H)$ is a strict map.

Finally, for the second restriction mentioned in the statement of the proposition, let $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ be a morphism in $\mathbf{RWThHom}_{\models}$. We have to show that $\mathcal{K}(H) = (H|_{\Pi_1}, \eta^H_{J_1(State)})$ is a map from $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ to $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$, that is, that $\eta^H_{J_1(State)}$ is a Π_1-map from $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ to the reduct $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}|_{H|_{\Pi_1}}$. Let $[t] \rightarrow [t']$ be a transition in $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$. By the deadlock-freedom assumption (recall the definition of objects in $\mathbf{SRWTh}_{\models}$ in Section 5.1), this means that $t_0 \rightarrow^1_{\mathcal{R}_1, J_1(State)} t'_0$ for some $t_0 \in [t]$, $t'_0 \in [t']$. Since H preserves rewrites, $H(t_0) \rightarrow^1_{\mathcal{R}_2, J_2(State)} H(t'_0)$, and therefore $[H(t)] \rightarrow [H(t')]$ in $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$. Preservation of predicates is proved as before. □

An important consequence of this result and Theorems 2 and 4 is the following:

Theorem 10. *Given a morphism* $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *in* $\mathbf{SRWThHom}_{\models}$, $\mathbf{SRWThHom}^{\mathrm{str}}_{\models}$, *or* $\mathbf{RWThHom}_{\models}$ *and a formula* φ *in* $\mathrm{ACTL}^* \backslash \{\neg, \mathbf{X}\}(\Pi_1)$, $\mathrm{ACTL}^* \setminus \mathbf{X}(\Pi_1)$, *or* $\mathrm{ACTL}^* \backslash \neg(\Pi_1)$, *respectively, if* $H(\varphi)$ *holds in* $\mathcal{K}(\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *then* φ *holds in* $\mathcal{K}(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1)$.

Note that there exists an obvious inclusion functor $\mathbf{SRWThHom}_{\models} \hookrightarrow \mathbf{SRWTh}_{\models}$. Similarly, a category $\mathbf{RecSRWThHom}_{\models}$ of recursive morphisms can be defined with an inclusion functor $\mathbf{RecSRWThHom}_{\models} \hookrightarrow \mathbf{RecSRWTh}_{\models}$.

The present lifting of Kripke structures to the framework of rewriting logic can be represented graphically with the following commutative diagram. In it, the horizontal arrows between categories associated to Kripke structures are inclusions, and those that map to categories associated to transition systems are the expected forgetful functors.

$$\begin{array}{ccccccc}
\mathbf{SRWThHom}_{\models} & \longrightarrow & \mathbf{SRWTh}_{\models} & \longrightarrow & \mathbf{SRelRWTh}_{\models} & \longrightarrow & \mathbf{SRWTh} \\
\downarrow \mathcal{K} & & \downarrow \mathcal{K} & & \downarrow \mathcal{K} & & \downarrow \mathcal{T} \\
\mathbf{KSMap} & \longrightarrow & \mathbf{KSMap} & \longrightarrow & \mathbf{KSSim} & \longrightarrow & \mathbf{STSys}
\end{array}$$

There are of course similar diagrams involving recursive structures.

5.5 Examples of Simulations as Theoroidal Morphisms

5.5.1 Predicate Abstraction. Simulations are useful to define abstractions that allow studying the properties of a complex system using a simpler one. A particular instance of the methodology of abstraction is *predicate abstraction* [22, 15, 47, 18]. Under this approach, the abstract domain is a Boolean algebra over a set of assertions and the abstraction function, typically as part of a Galois connection, is symbolically constructed as the conjunction of all expressions satisfying a certain condition, which is typically proved using theorem proving. We now show how predicate abstractions can be understood as an instance of our notion of algebraic simulation.

Let us first focus on the transition relation. Given a computational system, a set $\phi_1, \ldots, \phi_n$ of predicates over the states determines an abstraction function mapping a state S to the Boolean tuple $\langle \phi_1(S), \ldots, \phi_n(S) \rangle$. Let us assume that the transitions of the system are specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ whose kind of states is *State*. Then, if $\mathcal{R}$ is *State*-encapsulated with constructor $st : k_1 \ldots k_m \longrightarrow State$ (that is, among all operators in Σ the kind *State* only appears in the operator *st*, and only as its coarity), the above predicate abstraction can be represented in rewriting logic by means of a rewrite theory $\mathcal{R}_A = (\Sigma_A, E_A, R_A)$ where:

- Σ_A contains Σ and the signature of *BOOL*, together with a new kind *BState*, a new operator $bst : Bool^n \longrightarrow BState$ and, for each predicate ϕ_i, $1 \leq i \leq n$, an operator $p_i : State \longrightarrow Bool$ to represent it. We then have a signature morphism $H : \Sigma \longrightarrow \Sigma_A$ that maps the kind *State* to *BState*, the constructor *st* to the term

$$bst(p_1(st(x_1, \ldots, x_m)), \ldots, p_n(st(x_1, \ldots, x_m))),$$

  and is the identity everywhere else.
- E_A contains $H(E)$ and the equations in *BOOL*, together with equations for $p_1, \ldots, p_n$ specifying the predicates $\phi_1, \ldots, \phi_n$.
- $R_A = H(R)$.

Then, by construction, $H : (\Sigma, E) \longrightarrow (\Sigma_A, E_A)$ is a theory morphism such that $t \rightarrow^1_{\mathcal{R}, State} t'$ implies $H(t) \rightarrow^1_{\mathcal{R}_A, BState} H(t')$, thus preserving the transition relation.

We can now turn our attention to the preservation of properties. Graphically, the relationship between the different theories involved is depicted in the following diagram,

$$\begin{array}{ccc} (\Sigma, E) & \hookrightarrow & (\Sigma', E \cup D) \\ {\scriptstyle H}\downarrow & & \downarrow \\ (\Sigma_A, E_A) & \hookrightarrow & (\Sigma'_A, E_A \cup D_A) \end{array}$$

where $(\Sigma', E \cup D)$ is the equational theory specifying the properties of the given system, and $(\Sigma'_A, E_A \cup D_A)$ is the theory we have to associate to $\mathcal{R}_A$ defining its atomic propositions.

The syntax for the state predicates q (that we assume are constants) in the original system is given in a subsignature Π of Σ'. It is usually the case that for each of these q one of the predicates ϕ_i in the basis defining the abstraction has the meaning "the state S satisfies q." Let $q_1, \ldots, q_k$ be the state predicates in Π. We assume $k \leq n$, and that each q_j, $1 \leq j \leq k$, corresponds to the predicate ϕ_j in the basis of the abstraction (but in general we may have $n > k$, with predicates $\phi_{k+1}, \ldots, \phi_n$ not having a counterpart in Π). That is, for a ϕ_j with a corresponding q_j in Π, its specification in E_A through $p_j(S)$ is thus essentially the same (modulo renaming) as that of $S \models q_j$ in D, so that $E \cup D \vdash (S \models q_j) = \mathit{true} \iff E_A \vdash p_j(S) = \mathit{true}$. Then, for the abstraction we use the same set of state predicates Π and they are specified in a theory extension $(\Sigma_A, E_A) \subseteq (\Sigma'_A, E_A \cup D_A)$, with $\Sigma'_A = \Sigma_A \cup \Sigma'$ and D_A containing, for each q_j in Π with associated ϕ_j, the equation

$$(\forall \{x_1, \ldots, x_n\})\, (\mathit{bst}(x_1, \ldots, x_j, \ldots, x_n) \models q_j) = x_j\,.$$

Let us extend H to $\Sigma \cup \Pi$ by mapping each state predicate to itself. Thus, for all ground terms t of kind *State* and state predicates q_j, if $E_A \cup D_A \vdash (H(t) \models q_j) = \mathit{true}$ then, by the equation defining q_j in $E_A \cup D_A$ and since $H(t) = \mathit{bst}(p_1(t), \ldots, p_n(t))$, we have $E_A \cup D_A \vdash p_j(t) = \mathit{true}$ and even $E_A \vdash p_j(t) = \mathit{true}$ because p_j is completely specified in E_A. And hence, due to the relation between the equations defining $p_j(S)$ and $S \models q_j$, $E \cup D \vdash (t \models q_j) = \mathit{true}$ holds and preservation of predicates is guaranteed.

Finally, we can put all the pieces together and summarize the previous discussion as follows.

Theorem 11. *Let a concurrent system be specified as an object $(\mathcal{R}, (\Sigma', E \cup D), J)$ of the category* $\mathbf{SRWThHom}_{\models}$*, where $\mathcal{R}$ is $J(\mathit{State})$-encapsulated, and let $\phi_1, \ldots, \phi_n$ be a set of predicates over the kind $J(\mathit{State})$, with each state predicate $q_j \in \Pi$ (we assume that all such q_j are constants) corresponding to a ϕ_j, $1 \leq j \leq k$. The result of applying predicate abstraction is the system given by $(\mathcal{R}_A, (\Sigma'_A, E_A \cup D_A), J_A)$, where $(\Sigma'_A, E_A \cup D_A)$ and $\mathcal{R}_A$ are defined as explained above, and where $J_A(\mathit{State}) = \mathit{BState}$. Then, with these definitions, $H : (\mathcal{R}, (\Sigma', E \cup D), J) \longrightarrow (\mathcal{R}_A, (\Sigma'_A, E_A \cup D_A), J_A)$ is an arrow in* $\mathbf{SRWThHom}_{\models}$*, where H is the signature morphism $\Sigma \cup \Pi \longrightarrow \Sigma'_A \cup \Pi$.*

The bakery protocol. Let us illustrate these ideas by outlining how they apply to the bakery protocol. This is an infinite state protocol that achieves mutual exclusion between processes by dispensing a number to each process and serving them in sequential order according to the number they hold. For the case of two processes, the transitions can be specified in rewriting logic, using Maude syntax, by the following theory $\mathcal{R} = (\Sigma, E, R)$:

```
mod BAKERY is
  protecting NAT .
  sorts Mode State .
  ops sleep wait crit : -> Mode .
```

```
  op st : Mode Nat Mode Nat -> State .
  op initial : -> State .
  vars P Q : Mode .
  vars X Y : Nat .

  eq initial = st(sleep, 0, sleep, 0) .

  rl [p1_sleep] : st(sleep, X, Q, Y) => st(wait, s Y, Q, Y) .
  rl [p1_wait] : st(wait, X, Q, 0) => st(crit, X, Q, 0) .
  crl [p1_wait] : st(wait, X, Q, Y) => st(crit, X, Q, Y) if not (Y < X) .
  rl [p1_crit] : st(crit, X, Q, Y) => st(sleep, 0, Q, Y) .
  rl [p2_sleep] : st(P, X, sleep, Y) => st(P, X, wait, s X) .
  rl [p2_wait] : st(P, 0, wait, Y) => st(P, 0, crit, Y) .
  crl [p2_wait] : st(P, X, wait, Y) => st(P, X, crit, Y) if Y < X .
  rl [p2_crit] : st(P, X, crit, Y) => st(P, X, sleep, 0) .
endm
```

States are represented by terms of sort `State`, which are constructed by a 4-tuple operator `<_,_,_,_>`; the first two components describe the status of the first process (the mode it is currently in, and its priority as given by the number according to which it will be served), and the last two components the status of the second process. The rules describe how each process passes from being sleeping to waiting, from waiting to its critical section, and then back to sleeping.

The properties are defined in a theory extension $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ that simply adds four constants `1wait`, `1crit`, `2wait`, and `2crit` to Σ to characterize when the first and second processes are in `wait` or `crit` mode, together with the obvious equations:

```
  eq (st(wait, X, N, Y) |= 1wait) = true .
  eq (st(sleep, X, N, Y) |= 1wait) = false .
  eq (st(crit, X, N, Y) |= 1wait) = false .
  eq (st(P , X, wait, Y) |= 2wait) = true .
  eq (st(P , X, sleep, Y) |= 2wait) = false .
  eq (st(P , X, crit, Y) |= 2wait) = false .
  eq (st(crit , X, Q, Y) |= 1crit) = true .
  eq (st(sleep, X, Q, Y) |= 1crit) = false .
  eq (st(wait, X, Q, Y) |= 1crit) = false .
  eq (st(P , X, crit, Y) |= 2crit) = true .
  eq (st(P , X, sleep, Y) |= 2crit) = false .
  eq (st(P , X, wait, Y) |= 2crit) = false .
```

For this protocol, we might be interested in verifying the safety property $\mathbf{AG}\neg(\texttt{1crit} \wedge \texttt{2crit})$.

We will use the following set of seven predicates to define the predicate abstraction:

$$\begin{array}{ll} \phi_1(\texttt{st(M, X, N, Y)}) \iff \texttt{M == wait} & \phi_5(\texttt{st(M, X, N, Y)}) \iff \texttt{X == 0} \\ \phi_2(\texttt{st(M, X, N, Y)}) \iff \texttt{M == crit} & \phi_6(\texttt{st(M, X, N, Y)}) \iff \texttt{Y == 0} \\ \phi_3(\texttt{st(M, X, N, Y)}) \iff \texttt{N == wait} & \phi_7(\texttt{st(M, X, N, Y)}) \iff \texttt{X < Y} \\ \phi_4(\texttt{st(M, X, N, Y)}) \iff \texttt{N == crit} & \end{array}$$

Intuitively, we only care whether the processes are in `wait` or `crit` mode, whether their counters are equal to zero, and which counter is greater.

Note that the state predicates in the signature correspond to predicates 1–4. In terms of the notation used above, q_1 would be `1wait` and it would be associated to ϕ_1, q_2 would be `1crit` and would be associated to ϕ_2, and q_3 and q_4 would be `2wait` and `2crit`, associated to ϕ_3 and ϕ_4, respectively. Now, the abstract rewrite theory $\mathcal{R}_A = (\Sigma_A, E_A, R_A)$ is constructed by adding to $\mathcal{R}$:

– Operators `p1 : State -> Bool`, ..., `p7 : State -> Bool`, together with a new kind `BState` and the constructor for abstract states

```
op bst : Bool Bool Bool Bool Bool Bool Bool -> BState .
```

This determines the signature morphism H, that maps the constructor operator `st` to the term

```
bst(p1(st(M, X, N, Y)),...,p7(st(M, X, N, Y)))
```

– Equations associated to pi specifying ϕ_i for $i = 1, \ldots, 7$. Since predicates $\phi_1, \ldots, \phi_4$ correspond to the atomic propositions, their defining equations are "the same":

```
eq p1(st(wait, X, N, Y)) = true .
eq p1(st(sleep, X, N, Y)) = false .
eq p1(st(crit, X, N, Y)) = false .
eq p2(st(wait, X, N, Y)) = false .
eq p2(st(sleep, X, N, Y)) = false .
eq p2(st(crit, X, N, Y)) = true .
...
```

The three remaining equations are also immediate:

```
eq p5(st(M, X, N, Y)) = (X == 0) .
eq p6(st(M, X, N, Y)) = (Y == 0) .
eq p7(st(M, X, N, Y)) = (Y < X) .
```

– The translation of the rules in R by the signature morphism H. In particular, the two rules introduced before become:

```
rl bst(p1(st(M, X, sleep, Y)), ..., p7(st(M, X, sleep, Y))) =>
   bst(p1(st(M, X, wait, Y)), ..., p7(st(M, X, wait, s(X)))) .
crl bst(p1(st(M, X, wait, Y)), ..., p7(st(M, X, wait, Y))) =>
    bst(p1(st(M, X, crit, Y)), ..., p7(st(M, X, crit, Y)))
    if Y < X .
```

Finally, we have to write the equations in D_A defining the atomic propositions in the abstract model, which is straightforward.

```
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 1wait) = B1 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 1crit) = B2 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 2wait) = B3 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 2crit) = B4 .
```

By construction, this model is a predicate abstraction with respect to the basis $\phi_1, \ldots, \phi_7$ of the bakery protocol, in which the desired property can be model checked.

It is worth pointing out that this algebraic method of defining predicate abstractions cannot be expressed within the framework of [38, 39], because the specification of the predicates ϕ_i requires, in general, to introduce auxiliary operators and thus a different signature $\Sigma_A \neq \Sigma$. Also, the resulting rewrite theory *is not executable* in general. This means that it cannot be directly used in a tool like the Maude model checker [20, 12]. Predicate abstraction can be considered as a particular instance of our framework of algebraic simulations from a conceptual or foundational point of view, which is still quite useful because it provides a justification for the method within our framework. Current approaches to predicate abstraction do not work directly with the minimal transition relation (described in our account by $\mathcal{R}_A$). Instead, they compute a safe *approximation* of $\mathcal{R}_A$ by discharging some proof obligations.

5.5.2 A Fairness Example. In many situations we are interested in the behavior of a system under certain *fairness* assumptions, like requiring that a rule is eventually applied if it is always enabled from a certain point on. Currently, the capability of focusing only on those paths satisfying the fairness requirements is not supported by the Maude LTL model checker. However, all is not lost since we can illustrate the use of theoroidal (bi)simulation maps to reason about fairness. The treatment can be made for very general classes of rewrite theories, and for quite flexible notions of fairness [36] (see also [37], where a general method to reduce action-based formulas such as fairness to state-based formulas in LTL or CTL* is given). Here, we limit ourselves to illustrating some of the key ideas, including the use of theoroidal maps, by means of a simple communication protocol example. Note also that the same idea can be used for the representation and study of labeled transition systems in rewriting logic.

Consider a system consisting of a sender, a channel, and a receiver. The goal is to send a multiset of numbers (in arbitrary order) from the sender to the receiver through the channel. The channel can at any time contain several of these numbers. Besides the normal send and receive actions, the channel may stall an arbitrary number of times in sending some data. We can model the states of such a system by means of the signature

```
ops snd ch rcv : Nat -> Configuration .
op null : -> Configuration .
op __ Configuration Configuration -> Configuration [assoc comm id: null] .
```

where the operator `__` (juxtaposition notation) denotes multiset union and satisfies the equations of associativity and commutativity, and has `null` as its identity element. For example, the term

```
snd(7) snd(3) snd(7) ch(2) ch(3) rcv(1) rcv(9)
```

describes a state in which `3` and two copies of `7` have not yet been sent, `2` and another copy of `3` are in the channel, and `1` and `9` have been received. The behavior of the system is specified by the following three rewrite rules:

```
var N : Nat .

rl [send] : snd(N) => ch(N) .
rl [stall] : ch(N) => ch(N) .
rl [receive] : ch(N) => rcv(N) .
```

Is this system terminating? Not without extra assumptions, since the `stall` rule could be applied forever. To make it terminating it is enough to assume the following "weak fairness" property about the `receive` rule, described by the formula

$$\textit{wf-receive} = \mathbf{FG}\,\texttt{enabled-receive} \rightarrow \mathbf{GF}\,\texttt{taken-receive}\,,$$

that is, if eventually the `receive` rule becomes continuously enabled in a path, then it is taken infinitely often. Specifying the `enabled-receive` predicate equationally is quite easy (we just need to have some value in the channel) but the specification of the `taken-receive` predicate is more elusive. For example, does the `taken-receive` predicate hold of the state described above? We don't know; maybe the last action was receiving the value `1`, in which case it would hold, but it could instead have been stalling on `3`, or sending `2`, and then it wouldn't. Here is where a theory transformation corresponding to a theoroidal map, and allowing us to define a bisimilar system where the `taken-receive` predicate can be defined, comes in. The new theory extends the above signature with the following new sorts and operators:

```
ops send stall receive * : -> Label .
op {_|_} : Configuration Label -> State .
```

that is, a state now consists of a configuration-label pair, indicating the last rule that was applied. Since initially no rule has been applied, we add the label `*` for all initial states. The rules of the transformed theory are now:

```
var Conf : Configuration .
var L : Label .

rl [send] : { Conf snd(n) | L } => { Conf ch(n) | send } .
rl [stall] : { Conf ch(n) | L } => { Conf ch(n) | stall } .
rl [receive] : { Conf ch(n) | L } => { Conf rcv(n) | receive } .
```

We can then define the predicates `enabled-send`, `enabled-receive`, and `taken-receive` by the equations

```
eq ({ Conf snd(N) | L } |= enabled-send) = true .
eq ({ Conf ch(N) | L } |= enabled-receive) = true .
eq ({ Conf | receive } |= taken-receive) = true .
```

Then the fair termination property can be defined by the following formula, which indeed holds in the Kripke structure associated to this transformed theory for any initial state:

$$\mathbf{A}(\textit{wf-receive} \rightarrow \mathbf{F}(\neg\texttt{enabled-send} \wedge \neg\texttt{enabled-receive}))\,.$$

Let $(\Sigma_{Comm}, E_{Comm})$ denote the underlying equational theory of our original rewrite theory, and let $(\Sigma_{LComm}, E_{Comm})$ denote that of the transformed theory (it has the same equations E_{Comm}). We can define a generalized theory morphism $H : (\Sigma_{LComm}, E_{Comm}) \longrightarrow (\Sigma_{Comm}, E_{Comm})$ as follows. The sorts, implicit kinds, and operators in Σ_{Comm} are mapped identically to themselves; the sort `State` is mapped to `Configuration`; and the sort `Label` is not mapped anywhere; the operator `{_|_}` is mapped to the variable `Conf` of sort `Configuration`; finally, the label constants are not mapped anywhere. Now, let Π_0 consist of the predicates `enabled-send` and `enabled-receive`, which in the original theory are defined by the equations

```
eq (Conf snd(N) |= enabled-send) = true .
eq (Conf ch(N) |= enabled-receive) = true .
```

Then, if *Comm* and *LComm* denote our rewrite theories, H induces a theoroidal *bisimulation* (and thus, strict) map of Kripke structures

$$H : \mathcal{K}(\mathit{LComm}, [\texttt{State}])_{\Pi_0} \longrightarrow \mathcal{K}(\mathit{Comm}, [\texttt{Configuration}])_{\Pi_0}.$$

Furthermore, in the case of *LComm* we can extend Π_0 to Π by adding the `taken-receive` predicate, so that fair termination can be properly specified and verified.

6 Proving Algebraic Simulations Correct

In this section we discuss methods to show that a given theory morphism or equationally-defined function indeed specifies a simulation between two computational systems. We start by considering the simpler case of non-stuttering maps, and move to stuttering ones after that.

6.1 Preservation of the Transition Relation in $\mathbf{RWThHom}_{\models}$ and $\mathbf{RWTh}_{\models}$

Let us start by considering the category $\mathbf{RWThHom}_{\models}$. A simple criterion for a generalized theory morphism $H : \mathcal{R}_1 \longrightarrow \mathcal{R}_2$ to actually preserve the transition relation is to check that, for each rule $t \longrightarrow t'$ **if** C in $\mathcal{R}_1$, a corresponding rule $H(t) \longrightarrow H(t')$ **if** $H(C)$ exists in $\mathcal{R}_2$. This requirement, however, is too strong in many cases.

Another possibility is to use theorem proving. In Maude there is available the ITP tool [14], an inductive theorem prover written in Maude itself that mechanizes the proof of sentences in membership equational logic. Unfortunately, for the time being, the ITP does not allow us to reason about rewrite rules. However, using the constructions explained in [6] we can still do that reasoning in an indirect way. In [6], to every rewrite theory $\mathcal{R}$ a membership equational theory *Reach*($\mathcal{R}$) is associated, with sorts Ar_k, Ar^1_k, and operators $\_ \to \_$ for each kind k in $\mathcal{R}$, and such that $\mathcal{R} \vdash t \longrightarrow t'$ iff *Reach*($\mathcal{R}$) $\vdash (t \to t') : Ar_k$, and $t \to^1_{\mathcal{R},k} t'$ iff *Reach*($\mathcal{R}$) $\vdash (t \to t') : Ar^1_k$. Based on this result, the following proposition offers a criterion for checking whether the transition relation is preserved.

Proposition 12. *Let $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ and $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$ be rewrite theories and let $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ be a theory morphism with initial semantics such that, for any $f \in \Sigma_1$, the term $H(f)$ does not have multiple occurrences of a single variable. Let T be a membership equational theory extending the disjoint union of (Σ_1, E_1) and (Σ_2, E_2) in a protecting mode with operators and sentences defining $\to^1_{\mathcal{R}_1,k}$ and $\to^1_{\mathcal{R}_2,k}$ as sorts $Ar1^1_k$ and $Ar2^1_k$, and in which the morphism H is equationally specified through a family of operators h. Then, if for all rules $(\forall X)\, t \longrightarrow t'$ **if** C in $\mathcal{R}_1$ with t of kind k we can inductively prove*

$$T \vdash_{ind} (\forall X)\, (h(t) \to h(t')) : Ar2^1_{H(k)} \textbf{ if } C^\sharp\,,$$

(where $C^\sharp$ is like C but with all rewrites $t \longrightarrow t'$ replaced by $(t \to t') : Ar1_k$), it follows that for all kinds k in $\mathcal{R}_1$ and $t, t' \in T_{\Sigma_1,k}$, we can inductively prove

$$t \to^1_{\mathcal{R}_1,k} t' \implies H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')\,.$$

Proof. Assume that $t \to^1_{\mathcal{R}_1,k} t'$. Then, either there is a rule $(\forall X)\, l \longrightarrow r$ **if** C in $\mathcal{R}_1$ and a substitution θ such that $E_1 \vdash t = \theta(l)$, $E_1 \vdash t' = \theta(r)$ and $\theta(C)$ holds in $\mathcal{R}_1$, or t is $f(t_1, \ldots, t_i, \ldots, t_n)$, t' is $f(t_1, \ldots, t'_i, \ldots, t_n)$ and $t_i \to^1_{\mathcal{R}_1,k_i} t'_i$. In the first case, by the way T has been constructed we have $T \vdash \theta(C^\sharp)$ and from the hypothesis it follows that $T \vdash (h(t) \to h(t')) : Ar2^1_{H(k)}$, which implies $H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')$. In the second case, by induction hypothesis $H(t_i) \to^1_{\mathcal{R}_2,H(k_i)} H(t'_i)$ and, by our assumption that $H(f)$ has no repeated variables, $H(t) = H(f)(H(t_1), \ldots, H(t_i), \ldots, H(t_n)) \to^1_{\mathcal{R}_2,H(k_i)} H(f)(H(t_1), \ldots, H(t'_i), \ldots, H(t_n)) = H(t')$. □

Note that this proposition is still valid even if H is just an arbitrary function, as long as it can be equationally defined. Therefore, the result also applies to morphisms in $\mathbf{RWTh}_{\models}$, and obviously to those in $\mathbf{RecRWThHom}_{\models}$ and $\mathbf{RecRWTh}_{\models}$.

6.1.1 Example: A Simple Protocol.

Let us illustrate this idea with an example. Consider the following protocol adapted from [16]:

```
mod PROTOCOL is
  protecting NAT .
  sorts State Mode .
  ops think eat : -> Mode .
```

```
  op st : Mode Mode Nat -> State .
  op odd : Nat -> Bool .

  vars M N : Mode .  var X : Nat .

  eq odd(0) = false .
  eq odd(s(X)) = not(odd(X)) .

  crl st(think, N, X) => st(eat, N, X) if odd(X) = true .
  rl  st(eat, N, X) => st(think, N, 3 * X + 1) .
  crl st(M, think, X) => st(M, eat, X) if odd(X) = false .
  crl st(M, eat, X) => st(M, think, X quo 2) if odd(X) = false .
endm
```

Following [16], this specification can be thought of as a protocol controlling the mutually exclusive access to a common resource of two concurrent processes, modelling the behavior of two mathematicians, corresponding to the first two components in a state represented by `st(M, N, X)`. They alternate phases of "thinking" and "eating," regulated by the current value `X` of the third component of the state: if `X` is odd, then the first mathematician has the right to enjoy the meal, otherwise, the turn corresponds to the second one. After finishing the eating phase, each mathematician leaves the dining room and modifies the value of `N` in his own fashion.

Consider now the following module, purported to specify a correct abstraction of the system specified by `PROTOCOL`, that replaces the third argument of the state by its parity.

```
mod PROTOCOL-ABS is
  sorts State Mode Parity .
  ops think eat : -> Mode .
  ops o e : -> Parity .
  op st : Mode Mode Parity -> State .

  vars M N : Mode .

  rl st(think, N, o) => st(eat, N, o) .
  rl st(eat, N, o) => st(think, N, e) .
  rl st(eat, N, e) => st(think, N, o) .
  rl st(M, think, e) => st(M, eat, e) .
  rl st(M, eat, e) => st(M, think, e) .
  rl st(M, eat, e) => st(M, think, o) .
endm
```

The theory T in Proposition 12 corresponding to these two modules is then as follows. (Actually, since the modules are quite simple, the theory below is a simplification of the theory that would result from applying the general construction given in [6].)

```
fmod ABSTRACTION is protecting NAT .
  sorts AR1 AR1? AR2 AR2? .
  sorts Mode1 Mode2 State1 State2 Parity .

  subsort AR1 < AR1? .
  subsort AR2 < AR2? .

  op odd : Nat -> Bool .
```

```
  ops think1 eat1 : -> Mode1 [ctor] .
  op st1 : Mode1 Mode1 Nat -> State1 [ctor] .

  ops think2 eat2 : -> Mode2 [ctor] .
  ops o e : -> Parity [ctor] .
  op st2 : Mode2 Mode2 Parity -> State2 [ctor] .

  op _->_ : State1 State1 -> AR1? [ctor] .
  op _->_ : State2 State2 -> AR2? [ctor] .

  op abs : State1 -> State2 .
  op absMode : Mode1 -> Mode2 .

  vars M1 N1 : Mode1 .
  vars M2 N2 : Mode2 .
  vars X Y : Nat .

  eq odd(0) = false .
  eq odd(s(X)) = not(odd(X)) .

  eq absMode(think1) = think2 .
  eq absMode(eat1) = eat2 .

  ceq abs(st1(M1, N1, X)) = st2(absMode(M1), absMode(N1), o)
      if (odd(X) = true)  .
  ceq abs(st1(M1, N1, X)) = st2(absMode(M1), absMode(N1), e)
      if (odd(X) = false) .

  cmb st1(think1, N1, X) -> st1(eat1, N1, X) : AR1 if (odd(X) = true) .
  mb st1(eat1, N1, X) -> st1(think1, N1, (3 * X) + 1) : AR1  .
  cmb st1(M1, think1, X) -> st1(M1, eat1, X) : AR1 if (odd(X) = false) .
  cmb st1(M1, eat1, X) -> st1(M1, think1, X quo 2) : AR1 if (odd(X) = false) .

  mb st2(think2, N2, o) -> st2(eat2, N2, o) : AR2 .
  mb st2(eat2, N2, o) -> st2(think2, N2, e) : AR2 .
  mb st2(eat2, N2, e) -> st2(think2, N2, o) : AR2 .
  mb st2(M2, think2, e) -> st2(M2, eat2, e) : AR2 .
  mb st2(M2, eat2, e) -> st2(M2, think2, e) : AR2 .
  mb st2(M2, eat2, e) -> st2(M2, think2, o) : AR2 .
endfm
```

Note that each of the rules in the original two specifications has become a membership assertion defining one of the two types `AR1` or `AR2`.

Then, we can prove with the ITP that the transition given by the fourth rule in `PROTOCOL` is preserved in `PROTOCOL-ABS`, which is expressed as follows:

```
(goal abstract4 : ABSTRACTION |- A{ M1:Mode1 ; X:Nat }
      (((odd(X:Nat)) = (false)) =>
      ((abs(st1(M1:Mode1, eat1, X:Nat)) ->
        abs(st1(M1:Mode1, think1, X:Nat quo 2))) : AR2)) .)
```

The proof for the other rules is similar. Full details can be found in [42].

Note that Proposition 12 is not very useful when the condition C contains rules. Consider for example a rewrite theory $\mathcal{R}_1$ with two unary operators f_1 and g_1, and the rule $f_1(x) \to g_1(y)$ **if** $x \to y$. Let us write $\mathcal{R}_2$ for the rewrite theory that is obtained from $\mathcal{R}_1$ by renaming

f_1 and g_1 as f_2 and g_2. $\mathcal{R}_1$ and $\mathcal{R}_2$ are clearly related by a theory morphism H (just a renaming) and the transition relation is trivially preserved. However, to prove that using the previous result we would have to show

$$T \vdash_{ind} (\forall x, y)\, (h(f_1(x)) \to h(g_1(x))) : Ar2^1_{H(k)} \textbf{ if } (x \to y) : Ar1_k\,,$$

which requires the use of some kind of induction hypothesis on $(x \to y) : Ar1_k$, which is not available. Fortunately, many specifications do not require the use of rules in the conditions; in particular, recursive rewrite theories belong to this class (recall Definition 19).

6.2 Preservation of the Transition Relation in $\mathbf{SRWThHom}_{\models}$ and $\mathbf{SRWTh}_{\models}$

The definition of stuttering simulations requires the satisfaction of a property involving infinite paths, which in general is not easy to check. In Section 2.4 we presented an alternative characterization that can also be used for morphisms between rewrite theories; however, we would obviously rather have conditions that apply directly to the sets of equations and rules of the rewrite theories.

The idea is as follows. Assume that we have rewrite theories $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ and $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$, and a generalized theory morphism with initial semantics $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$. For H to be a stuttering map it is enough to show that some of the rules in R_1 give rise to rewrite steps in $\mathcal{R}_2$ while the rest amount to stuttering when translated. That is, R_1 can be decomposed as the disjoint union of R_1' and R_1'' so that if $t \to^1_{R_1',k} t'$ then $H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')$, and if $t \to^1_{R_1'',k} t'$ then $H(t)$ and $H(t')$ can be proved to be equal in $\mathcal{R}_2$. There is only one detail missing: in order to avoid infinite stuttering we have to require R_1'' to be terminating. This idea is formalized in the following proposition.

Proposition 13. *Let $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ and $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$ be rewrite theories and let $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ be a generalized theory morphism with initial semantics such that, for any $f \in \Sigma_1$, the term $H(f)$ does not have multiple occurrences of a single variable. Let T be a membership equational theory extending the disjoint union of (Σ_1, E_1) and (Σ_2, E_2) with operators and sentences defining $\to^1_{\mathcal{R}_1,k}$ and $\to^1_{\mathcal{R}_2,k}$ as sorts $Ar1^1_k$ and $Ar2^1_k$, and in which the morphism H is equationally specified through a family of operators h. Assume that R_1 is the disjoint union of R_1' and R_1'', with R_1'' terminating modulo E_1. Then, if for all rules $(\forall X)\, t \longrightarrow t'$ **if** C in R_1' with t of kind k we can inductively prove*

$$T \vdash_{ind} (\forall X)\, (h(t) \to h(t')) : Ar2^1_{H(k)} \textbf{ if } C^\sharp\,,$$

*(where $C^\sharp$ is like C but with all rewrites $t \longrightarrow t'$ replaced by $(t \to t') : Ar1_k$), and for all rules $(\forall X)\, t \longrightarrow t'$ **if** C in R_1'' with t of kind k,*

$$T \vdash_{ind} (\forall X)\, (h(t) = h(t')) \textbf{ if } C^\sharp\,,$$

it follows that each path in $\mathcal{R}_1$ is H-matched by a path in $\mathcal{R}_2$.

Proof. Let π be a path $t = t_0 \to_{\mathcal{R}_1} t_1 \to_{\mathcal{R}_1} t_2 \to_{\mathcal{R}_1} \cdots$ starting at $t \in T_{\Sigma_1,k_1}$: we have to prove that there is a path ρ in $\mathcal{R}_2$ starting at $H(t)$ that H-matches π. For this, define $\alpha(0) = 0$, and $\alpha(i+1)$ to be the first position in π greater than $\alpha(i)$ that results from the application of a rule in R_1'; since R_1'' is terminating, α is well-defined and strictly increasing. Then, define ρ by $\rho(i) = H(t_{\alpha(i)})$. It turns out that $t_{\alpha(i)}$ reaches $t_{\alpha(i+1)}$ after a finite number of rewrites in R_1'' and a single transition in R_1': $t_{\alpha(i)} \to^1_{R_1'',k} u_1 \to^1_{R_1'',k} u_2 \to^1_{R_1'',k} \cdots \to^1_{R_1'',k} u_n \to^1_{R_1',k} t_{\alpha(i+1)}$. By the assumptions of the proposition, applying the same reasoning as in the proof of Proposition 12 we have that $H(u_n) \to^1_{\mathcal{R}_2,H(k)} H(t_{\alpha(i+1)}) = \rho(i+1)$. Analogously, $t \to^1_{R_1'',k} t'$

implies $E_2 \vdash H(t) = H(t')$ and hence $\rho(i) = H(t_{\alpha(i)})$, $u_1, \ldots, u_n$ are all provably equal in E_2. It follows that $\rho(i) \rightarrow^1_{\mathcal{R}_2, H(k)} \rho(i+1)$ and therefore ρ is a valid path in $\mathcal{R}_2$ and it H-matches π by construction. □

As in the case of non-stuttering simulations, this proposition applies not only to theory morphisms but to any equationally definable function, and thus it provides a criterion to check preservation of transitions in $\mathbf{SRWTh}_{\models}$ as well.

6.2.1 Example: A readers-writers system. Consider the following specification of a readers-writers system.

```
mod R&W is
  protecting NAT .
  sort Config .
  op <_,_> : Nat Nat -> Config .  --- readers/writers

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

A state is represented by a pair `< R, W >` indicating the number `R` of readers and the number `W` of writers accessing the critical resource. Readers and writers can leave the resource at any time, but writers can only gain access if nobody else is using it, and readers only if there are no writers.

Now, consider the following implementation of the system in which readers and writers "ask for permission" before entering the critical section.

```
mod R&W-STUTTERING is
  protecting NAT .
  sorts Key Config .

  ops reader writer empty : -> Key .
  op <_,_,_> : Nat Nat Key -> Config .

  vars R W : Nat .
  var K : Key .

  rl < 0, 0, empty > => < 0, 0, writer > .
  rl < R, s(W), K > => < R, W, K > .
  rl < R, 0, empty > => < R, 0, reader > .
  rl < s(R), W, K > => < R, W, K > .
  rl < R, W, writer > => < R, s(W), empty > .
  rl < R, W, reader > => < s(R), W, empty > .
endm
```

The third argument of the tuple of a state indicates whether a `reader` or a `writer` has asked for permission to enter the critical section; it takes the value `empty` if no request has been made.

We can show that R&W-STUTTERING is a correct implementation of R&W by constructing a stuttering map of transition systems

$$h : \mathcal{T}(\texttt{R\&W-STUTTERING})_{\texttt{Config}} \longrightarrow \mathcal{T}(\texttt{R\&W})_{\texttt{Config}} .$$

For that, if the theory T in Proposition 13 renames the sort Config in R&W and R&W-STUTTERING respectively as Config1 and Config2, the stuttering map h can be equationally defined in T as follows. (Again, it is a simplification of the construction in [6].)

```
fmod R&W-SIMULATION is
  protecting NAT .

  sorts AR1 AR1? AR2 AR2? .
  sorts Config1 Key Config2 .

  subsort AR1 < AR1? .
  subsort AR2 < AR2? .

  op <_,_> : Nat Nat -> Config1 [ctor] .  --- readers/writers

  ops reader writer empty : -> Key [ctor] .
  op <_,_,_> : Nat Nat Key -> Config2 [ctor] .

  op _->_ : Config1 Config1 -> AR1? [ctor] .
  op _->_ : Config2 Config2 -> AR2? [ctor] .

  op h : Config2 -> Config1 .

  vars R W : Nat .
  var K : Key .

  eq h(< R, W, empty >) = < R, W > .
  eq h(< R, W, reader >) = < s(R), W > .
  eq h(< R, W, writer >) = < R, s(W) > .

  mb < 0, 0 > -> < 0, s(0) > : AR1 .
  mb < R, s(W) > -> < R, W > : AR1 .
  mb < R, 0 > -> < s(R), 0 > : AR1 .
  mb < s(R), W > -> < R, W > : AR1 .

  mb < 0, 0, empty > -> < 0, 0, writer > : AR2 .
  mb < R, s(W), K > -> < R, W, K > : AR2 .
  mb < R, 0, empty > -> < R, 0, reader > : AR2 .
  mb < s(R), W, K > -> < R, W, K > : AR2 .
  mb < R, W, writer > -> < R, s(W), empty > : AR2 .
  mb < R, W, reader > -> < s(R), W, empty > : AR2 .
endfm
```

Let R_1 be the first four rules in R&W-STUTTERING and R_2 the remaining two. R_1 is terminating, because the total number of readers, writers, and "emptys" decreases. Now, consider the first rule in R_1. Recall that the rewrite relation in R&W is represented by a sort AR1 in T. Then, for instance for the second rule, we can prove

```
(goal abstract2 : R&W-SIMULATION |- A{ R:Nat ; W:Nat ; K:Key }
      ((h(< R:Nat, s(W:Nat), K:Key >) -> h(< R:Nat, W:Nat, K:Key >)) : AR1) .)
```

and similarly for the other rules in R_1. In the case of R_2, we can show that

```
(goal stutt1 : R&W-SIMULATION |- A { R:Nat ; W:Nat }
      ((h(< R:Nat, W:Nat, reader >)) = (h(< s(R:Nat), W:Nat, empty >))) .)
```

and

```
(goal stutt2 : R&W-SIMULATION |- A { R:Nat ; W:Nat }
      ((h(< R:Nat, W:Nat, writer >)) = (h(< R:Nat, s(W:Nat), empty >))) .)
```

Therefore, the conditions in Proposition 13 are satisfied and h is an algebraic stuttering map of transition systems.

6.3 Preservation of Atomic Propositions

In order to show preservation of atomic propositions it is convenient to assume that they are completely specified, in the sense that we can always prove whether they are *true* or *false* with respect to any particular state. For the case of decidable properties, we can make this assumption without loss of generality. We then have the following result, similar to the ones for preservation of the transition relation.

Proposition 14. *Let* $(\Sigma_1, E_1) \subseteq (\Sigma_1', E_1 \cup D_1)$ *and* $(\Sigma_2, E_2) \subseteq (\Sigma_2', E_2 \cup D_2)$ *be the equational theories corresponding to two objects in* $\mathbf{SRWThHom}_{\models}$*, and let* $H : \Sigma_1 \cup \Pi_1 \longrightarrow \Sigma_2 \cup \Pi_2$ *be a generalized signature morphism such that* $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ *is a theory morphism. Let* T *be a membership equational theory extending the disjoint union of* $(\Sigma_1', E_1 \cup D_1)$ *and* $(\Sigma_2', E_2 \cup D_2)$ *in which the morphism* H *is equationally specified through a family of operators* h*. Then, if we can prove*

$$T \vdash_{ind} (\forall X)\, (h(t) \models h(p)) = \mathit{false} \textbf{ if } C$$

for all equations $(\forall X)\, (t \models p) = \mathit{false}$ **if** C *in* D_1*, it follows that, for ground terms* t' *and* p'*,*

$$E_2 \cup D_2 \vdash (h(t') \models h(p') = \mathit{true}) \implies E_1 \cup D_1 \vdash (t' \models p' = \mathit{true})\,.$$

Furthermore, if instead we can prove

$$T \vdash_{ind} (\forall X)\, (x \models p) = (h(x) \models h(p))$$

then the above implication becomes an equivalence.

Proof. By our assumption about the completeness of the specifications it holds that either $E_1 \cup D_1 \vdash (t \models p) = \mathit{true}$ or $E_1 \cup D_1 \vdash (t \models p) = \mathit{false}$. But this second case cannot happen unless $\mathit{true} = \mathit{false}$ in $E_2 \cup D_2$, because we can prove by structural induction and using the hypothesis of the proposition that $E_1 \cup D_1 \vdash (t \models p) = \mathit{false}$ implies $E_2 \cup D_2 \vdash (h(t) \models h(p)) = \mathit{false}$.

It is also clear that the equality at the end of the proposition implies that h is strict, with which we have the stated equivalence. □

Again, this result also applies to $\mathbf{SRWTh}_{\models}$.

6.3.1 Example: A revisited protocol. For the protocol of the "thinking mathematicians" of Section 6.1, the atomic propositions are specified as follows:

```
eq st1(think1, N1, X) |= nmexcl1 = true .
eq st1(M1, think1, X) |= nmexcl1 = true .
eq st1(eat1, eat1, X) |= nmexcl1 = false .

eq st2(think2, N2, P) |= nmexcl2 = true .
eq st2(M2, think2, P) |= nmexcl2 = true .
eq st2(eat2, eat2, P) |= nmexcl2 = false .
```

Then, the conditions we have to check are of the form:

```
(goal abstract : ABSTRACTION |- A{ X:Nat }
      ((abs(st1(eat1, eat1, X:Nat)) |= nmexcl2) = (false)) .)
```

Similarly, for strictness we would have to show that

```
(goal abstract-st : ABSTRACTION |- A{ S1:State1 }
      ((S1:State1 |= nmexcl1) = (abs(S1:State1) |= nmexcl2)) .)
```

which can also be proved with the ITP.

7 Concluding Remarks

We have presented a quite general notion of stuttering simulation between Kripke structures that relaxes the requirements on preservation of state predicates both in not requiring identical preservation and in allowing formulas to be translated. We have also proved general representability results showing that both Kripke structures and their simulations can be fruitfully represented in rewriting logic. There are different ways of representing these notions in rewriting logic, ranging from equational abstractions to algebraic stuttering simulations: equational abstractions were considered in [38, 39] and theoroidal maps in [31]. Here, besides giving a fuller account of theoroidal maps, we have focused on the remaining cases of equationally-defined abstraction maps and simulations as rewrite relations, illustrating their use and giving sufficient conditions to discharge their associated proof obligations.

A particular instance of our methodology is the technique of predicate abstraction; the theory transformation explained in this paper produces a non-executable theory in general, though. We have also developed a prototype using Maude that implements the algorithm in [15], and have used it to abstract some of the examples considered here [42]. The prototype makes heavy use of Maude's ITP to build the abstract transition relation.

Future research directions include: a continued quest for even more general simulations and for related preservation and compositionality techniques; proof methods and tool support to prove simulations correct; and experimentation and case studies.

Acknowledgments. We thank Pete Manolios for very helpful discussions on stuttering simulations. We also thank Jan Bergstra and John Tucker for suggesting to us the use of the encoding of finite sets as numbers in [46] as a key step in the proof of Theorem 6.

References

1. L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction - CADE-8*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 1986.
2. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *7th International Conference on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1980.
3. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.
4. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
5. M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
6. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.4), 2008. `http://maude.cs.uiuc.edu/maude2-manual/`.
12. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude — A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
13. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000. `http://maude.csl.sri.com/papers`.
14. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Special issue with extended versions of selected papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.
15. M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998.
16. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291, 1997.
17. S. Das. *Predicate Abstraction*. PhD thesis, Department of Electrical Engineering, Stanford University, Dec. 2003.
18. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification. 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999.
19. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.
20. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA'02, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

21. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
22. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification. 9th International Conference, CAV'97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
23. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.
24. L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668. North-Holland, 1983.
25. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–36, 1995.
26. N. Lynch and F. Vaandrager. Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.
27. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, Aug. 2001.
28. P. Manolios. A compositional theory of refinement for branching time. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods. 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.
29. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay, editor, *Handbook of Philosophical Logic. Second Edition*, volume 9, pages 1–81. Kluwer Academic Press, 2002.
30. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
31. N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro, P. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques. 17th International Workshop, WADT 2004. Barcelona, Spain, March 27-30, 2004. Revised Selected Papers*, volume 3423 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2005.
32. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Press, 1993.
33. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
34. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
35. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3 - 7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
36. J. Meseguer. Localized fairness: A rewriting semantics. In J. Giesl, editor, *Rewriting Techniques and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19–21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2005.
37. J. Meseguer. The temporal logic of rewriting: A gentle introduction. In P. Degano, R. D. Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 354–382. Springer, 2008.
38. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.
39. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.
40. K. S. Namjoshi. A simple characterization of stuttering bisimulation. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science. 17th Conference, Kharagpur, India, December 18 - 20, 1997. Proceedings*, volume 1346 of *Lecture Notes in Computer Science*, pages 284–296. Springer, 1997.

41. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
42. M. Palomino. *Reflexión, abstracción y simulación en la lógica de reescritura*. PhD thesis, Universidad Complutense de Madrid, Spain, Mar. 2005. `http://maude.sip.ucm.es/~miguelpt/`.
43. M. Palomino, J. Meseguer, and N. Martí-Oliet. A categorical approach to Kripke structures and simulations. In J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. Rutten, editors, *Algebra and Coalgebra in Computer Science. First International Conference, CALCO 2005, Swansea, UK, September 2005. Proceedings*, volume 3629 of *Lecture Notes in Computer Science*, pages 313–330. Springer, 2005.
44. D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.
45. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6–8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
46. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, 1967.
47. H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification. 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer, 1999.
48. J. R. Shoenfield. *Degrees of Unsolvability*. North-Holland, 1971.
49. R. J. van Glabbeek. The linear time-branching time spectrum I: The semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.
50. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Technical Report 134-03, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2003. `http://eprints.ucm.es/4888/`.
51. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.