Algebraic Stuttering Simulations[*]

Narciso Martí-Oliet
Dpto. Sistemas Informáticos y Computación
Univ. Complutense
narciso@sip.ucm.es

José Meseguer
Computer Science Department
Univ. of Illinois at Urbana-Champaign
meseguer@cs.uiuc.edu

Miguel Palomino
Dpto. Sistemas Informáticos y Computación
Univ. Complutense
miguelpt@sip.ucm.es

Abstract

Rewrite theories and their associated Kripke structures constitute a flexible and executable framework in which a wide range of systems can be studied. We present a general notion of simulation between Kripke structures, study its categorical aspects, and propose rewriting logic as a framework in which these simulations can be represented. Several representability results showing that rewriting logic is indeed a suitable framework for this purpose are given, and we illustrate its use with two examples.

1 Introduction

Rewriting logic is a very flexible framework with good properties for representing many concurrent systems at a high level [16, 14]. Rewrite theories can be executed in a language such as Maude [7] and give rise to an associated Kripke structure in which properties, when the number of states is finite, can be verified using Maude's model checker.

Hence, this paper tries to advance two main goals: 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* interesting classes of 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 enormously 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 underlying equational logic can be very useful for formal reasoning, since 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 [19] we have shown the usefulness of defining

[*]Research supported by ONR Grant N00014-02-1-0715, NSF GrantCCR-0234524, and by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130; and by the Spanish projects MIDAS TIC 2003–0100, DESAFIOS TIN2006–15660–C02–01, and by Comunidad de Madrid program PROMESAS S–0505/TIC/0407.

abstraction simulations equationally in rewriting logic, and of using tools such as Maude's LTL model checker [10] and inductive theorem prover [8] to verify properties and prove abstractions correct. The paper [15] further generalized [19] 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 those previous results. The emphasis here is on *foundations*, but we have included two examples to illustrate the general methodology; we refer the reader to the extended version [20] (which also contains the proofs of the results presented here) for a discussion of the related proof obligations for our proposed simulations.

We advance the first goal by generalizing simulations in three directions. First, we consider *stuttering simulations* in the sense of [3, 22, 13], which are quite general and useful to relate concurrent systems with different levels of atomicity; second we relax the condition on preservation of atomic properties 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'. 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 [23] we treated several of these categorical aspects at the level of Kripke structures, including a classification in terms of institutions; here we expand some of those ideas. 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—in short, r.e.—generalized simulation) can be represented by a rewrite relation.

2 Relating Kripke Structures

2.1 Stuttering Simulations

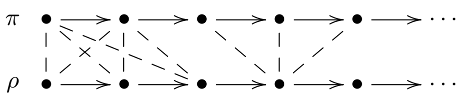
A Kripke structure over a set AP of atomic propositions consists of a transition system $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and a labeling function $L_{\mathcal{A}} : A \longrightarrow \mathcal{P}(AP)$. A path in $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ such that $\pi(i) \rightarrow_{\mathcal{A}} \pi(i+1)$ for each $i \in \mathbb{N}$. To specify system properties we use the logic $\mathrm{ACTL}^*(AP)$, which is the restriction of the branching-time temporal logic $\mathrm{CTL}^*(AP)$ (see for example [5, Sect. 3.1]) to those formulas such that their negation-normal forms (with negations pushed to atoms) do not contain any existential path quantifiers. There are two types of formulas in $\mathrm{CTL}^*(AP)$: state formulas, denoted by $\mathrm{State}(AP)$, and path formulas. The satisfaction relations are denoted by $\mathcal{A}, a \models \varphi$ 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. Sometimes, to avoid introducing implicitly existential quantifiers, it is more convenient to restrict ourselves to the negation-free fragment $\mathrm{ACTL}^*\backslash\neg(AP)$ of $\mathrm{ACTL}^*(AP)$.

For example, the behaviour of a simple periodic system could be represented by means of a transition system with four states, s_0, s_1, s_2, and s_3, and transitions $s_i \rightarrow s_{(i+1)\%3}$, $s_0 \rightarrow s_3$, and $s_3 \rightarrow s_3$. 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 $L(s_0) = \{\mathtt{sleep}\}$, $L(s_1) = \{\mathtt{wait}\}$, $L(s_2) = \{\mathtt{work}\}$, and $L(s_3) = \{\mathtt{error}\}$. The path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_0 \rightarrow s_1 \rightarrow \ldots$ satisfies the path formula $\mathbf{GF}\,\mathtt{work}$, indicating that the system always ends up doing some working. However, the state formula $\mathbf{AGF}\,\mathtt{work}$ does not hold in s_0 because a path starting at s_0 may eventually leap to s_3 and remain there.

In [19] we presented a notion of simulation similar to that in [5], but somewhat more general (simulations in [5] essentially correspond to our *strict* simulations). Here we generalize that definition in two ways. The first direction in which the original definition can be extended is that of stuttering bisimulations

[3, 22] and, more generally, stuttering simulations [13]. Our definition is closely related to the one given by Manolios [13], but with some technical and methodological differences.

For $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ transition systems and $H \subseteq A \times B$ a relation, we say that a path ρ in $\mathcal{B}$ *H-matches* a path π in $\mathcal{A}$ 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 1 ([15]) *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.

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 $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$. If H is a function we call H a* stuttering AP-map. *We call H a stuttering AP-bisimulation if H and H^{-1} are stuttering AP-simulations. We call H* strict *if aHb implies $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$.*

The fact that $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a stuttering 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 condition on properties 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.

Notice that these definitions are very general, not even requiring H to be total. This leads to some perhaps unexpected consequences: for example, the empty relation is vacuously a stuttering bisimulation of Kripke structures! The notion is natural, however, in that every stuttering AP-simulation arises from a total one restricted to a certain domain of interest.

Definition 2 *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.*

Proposition 1 *Let $H : \mathcal{A} \longrightarrow \mathcal{B}$ be a stuttering AP-simulation. 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 $\mathcal{A}$. Therefore, every stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ can alternatively be seen as a total stuttering AP-simulation $H : H^{-1}(\mathcal{B}) \longrightarrow \mathcal{B}$. Stuttering simulations of transition systems (see [13]) and of Kripke structures *compose*. Note also that the identity function $1_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{A}$ is trivially a stuttering simulation of transition systems and of Kripke structures. Therefore, transition systems together with stuttering simulations define a category **STSys**. Similarly, Kripke structures together with stuttering AP-simulations define a category $\mathbf{KSSim}_{AP}$, with two corresponding subcategories $\mathbf{KSMap}_{AP}$ and $\mathbf{KSBSim}_{AP}$ whose morphisms are, respectively, stuttering AP-maps and stuttering AP-bisimulations. There are also correspondings subcategories of *strict* stuttering simulations.

2.2 Shifting Our Ground

We also seek to generalize the definition of simulation so that Kripke structures over different sets of atomic propositions can be related. This will provide us with a very flexible way of relating Kripke structures and will allow us to gather all the previous categories $\mathbf{KSSim}_{AP}$ 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 3 ([23]) *Given a function $\alpha : AP \longrightarrow \text{State}(AP')$ 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 $L_{\mathcal{A}|_{\alpha}}(a) = \{p \in AP \mid \mathcal{A}, a \models \alpha(p)\}$.*

Proposition 2 ([23]) *Let $\alpha : AP \longrightarrow \text{State}(AP')$ be a function and let φ be a formula in $\text{CTL}^*(AP)$. 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, $\mathcal{A}, \pi \models \overline{\alpha}(\varphi) \iff \mathcal{A}|_{\alpha}, \pi \models \varphi$.*

The notation $\overline{\alpha}$ above stands for the homomorphic extension of α to formulas $\varphi \in \text{CTL}^*(AP)$. 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 $\text{State}(AP')$ as the range of the functions α is as general as possible. To deal with stuttering, however, we should restrict the range of α to formulas without the "next" operator $\mathbf{X}$. Also, note that when dealing with nonstrict simulations, since the reflected formulas will be in $\text{ACTL}^*\backslash\neg(AP)$ we will want our maps α to have their range in the negation-free fragment $\text{State}\backslash\{\neg, \mathbf{X}\}(AP')$, i.e., we will use maps $\alpha : AP \longrightarrow \text{State}\backslash\{\neg, \mathbf{X}\}(AP')$ instead. For example, let $AP = \{\texttt{init, compute}\}$ and $AP' = \{\texttt{sleep}, \texttt{wait}, \texttt{work}, \texttt{error}\}$, and $\alpha : AP \longrightarrow AP'$ given by $\alpha(\texttt{init}) = \texttt{work}$, $\alpha(\texttt{compute}) = \mathbf{AG}\,\texttt{work}$. If $\mathcal{A}$ is the previous periodic system, then $\mathcal{A}|_{\alpha}, s_2 \models \texttt{init}$ but $\mathcal{A}|_{\alpha}, s_2 \not\models \texttt{compute}$.

The definition of generalized stuttering simulations is now immediate.

Definition 4 ([23]) *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 $\alpha : AP \longrightarrow \text{State}\backslash\{\neg, \mathbf{X}\}(AP')$ (resp. $\alpha : AP \longrightarrow \text{State} \setminus \mathbf{X}(AP')$) and a stuttering AP-simulation (resp. strict stuttering AP-simulation) $H : \mathcal{A} \longrightarrow \mathcal{B}|_{\alpha}$.*

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.

Composition of simulations can be defined as $(\beta, G) \circ (\alpha, F) = (\overline{\beta} \circ \alpha, G \circ F)$. Using as objects pairs $(AP, \mathcal{M})$ with AP a set of atomic propositions and $\mathcal{M}$ a Kripke structure over AP, this immediately gives rise to a category **KSSim**; see [23] for a categorical discussion.

The important fact about stuttering simulations is that they reflect satisfaction of appropriate classes of formulas. Given Kripke structures $\mathcal{A}$ over AP and $\mathcal{B}$ over AP', a stuttering simulation $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ *reflects* the satisfaction of a formula $\varphi \in \text{CTL}^*(AP)$ 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 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 simulations; however, if we restrict our attention to $\text{ACTL}^* \setminus \mathbf{X}$ and $\text{ACTL}^*\backslash\{\neg, \mathbf{X}\}$, that is, the fragments of the logics that do not contain $\mathbf{X}$, formulas are reflected.

Theorem 1 *Stuttering simulations always reflect satisfaction of $\text{ACTL}^*\backslash\{\neg, \mathbf{X}\}$ formulas. Strict stuttering simulations also reflect satisfaction of $\text{ACTL}^* \setminus \mathbf{X}$ formulas.*

3 Rewriting Logic

One can distinguish two specification levels: a system specification level, in which the computational system of interest is specified, and a property specification level. These two levels correspond respectively to the specification of the transition system and the Kripke structure associated to the computational system. The main interest of rewriting logic [16] is that it provides a very flexible framework for specifying concurrent systems and for associating to them transition systems and Kripke structures. Rewriting logic is parameterized by an underlying equational logic. Here we use membership equational logic [18], an expressive extension of many-sorted equational logic that has *kinds* in addition to sorts (to simplify the presentation, we call them both *types* here), and allows subtypes defined by semantic conditions and operator overloading. Sentences are Horn clauses over atomic formulas, which include both *equations* $t = t'$ and *membership assertions* $w : s$ stating that the term w has type s.

Concurrent systems are axiomatized in rewriting logic by means of *rewrite theories* [16] of the form $\mathcal{R} = (\Sigma, E, R)$. 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 type 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 types k and k_l, $t, t' \in T_{\Sigma,k}(X)$, and $t_l, t'_l \in T_{\Sigma,k_l}(X)$ for $l \in L$. Under reasonable assumptions about E and R, rewrite theories are *executable*. Indeed, there are several rewriting logic language implementations, including CafeOBJ [11], ELAN [2], and Maude [6, 7].

Rewriting logic then has inference rules to infer all the possible concurrent computations in a system [16, 4], 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 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 $\rightarrow^1_{\mathcal{R},k}$ on $T_{\Sigma,k}$ that holds between terms $u, v \in T_{\Sigma,k}$ iff there is a proof of $u \longrightarrow v$ in which only one rewrite rule in R is applied to a single subterm. We can get a binary relation (with the same name) $\rightarrow^1_{\mathcal{R},k}$ on $T_{\Sigma/E,k}$ by defining $[u] \rightarrow^1_{\mathcal{R},k} [v]$ iff $u' \rightarrow^1_{\mathcal{R},k} v'$ for some $u' \in [u]$, $v' \in [v]$. This determines a transition system $\mathcal{T}(\mathcal{R})_k = (T_{\Sigma/E,k}, \rightarrow^1_{\mathcal{R},k})$ for each $k \in K$.

3.1 Example: Semantics of a Functional Language

In [12], a simple functional language called *Fpl* is defined along with a computation semantics and a more concrete semantics which uses an abstract machine. A state of the abstract machine is a triple $\langle S, \rho, e\rangle$, where S is a stack of values, ρ is an environment assigning values to variables, and e is an expression. A state for the computation semantics is a pair $\langle \rho, e\rangle$, with ρ an environment and e an expression. The transition relations $\langle S, \rho, e\rangle \longrightarrow \langle S', \rho', e'\rangle$ and $\langle \rho, e\rangle \longrightarrow \langle \rho', e'\rangle$ defined in [12] were translated to rewriting logic in [25]. They are summarized in App. A, in Maude notation, and we use them here to illustrate the main components of rewriting logic.

Environments in *Fpl*'s computation semantics are represented in a rewrite theory by terms of type `Env`. Similarly, there are two types 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 `<_,_> : [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 True 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 semantics of the stack machine is specified analogously.

These rewrite theories give rise to two transition systems, $\mathcal{A} = (A, \to_{\mathcal{A}})$ and $\mathcal{C} = (C, \to_{\mathcal{C}})$, for the abstract machine and the computation semantics respectively. The evaluation of a single expression in each of them requires the execution of several steps; therefore, it makes sense to study the relationship between those executions in order to prove the correctness of the implementation, and we do so in Sect. 5.

4 Specifying Kripke Structures as Rewrite Theories

4.1 Temporal Properties of Rewrite Theories

In order to associate temporal properties to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ we need to make explicit two things: the intended *type* k of states in the signature Σ, and the relevant *state predicates.* Once the type 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 only of the property specification. 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 type 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 $p : s_1 \ldots s_n \longrightarrow$ *Prop*, with *Prop* the type of propositions. If k is the type of states, the *semantics* of the state predicates Π is defined with the help of an operator $\_ \models \_ : k\ \mathit{Prop} \longrightarrow \mathit{Bool}$ in Σ' and by the equations $E \cup D$. By definition, given ground terms $u_1, \ldots, u_n$, we say that the state predicate $p(u_1, \ldots, u_n)$ *holds* in the state $[t]$ iff $E \cup D \vdash t \models p(u_1, \ldots, u_n) = \mathit{true}$.

Then, we associate to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ (with a selected type 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 $\theta(p(x_1, \ldots, x_n))$. We define $\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, \to^{1}_{\mathcal{R},k}, L_{\Pi})$, 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 *Fpl*'s computation semantics can be extended by declaring a constant `v : -> Prop` for each value v and equations (in Maude notation)

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

defining $L_{\mathcal{C}}(\langle \rho, v \rangle) = \{v\}$ and $L_{\mathcal{C}}(c)$ empty otherwise. For the abstract machine:

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

These extensions lift the transition systems $\mathcal{C}$ and $\mathcal{A}$ to the level of Kripke structures.

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 [10].

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 notion of recursive Kripke structures. We use the notion of recursive set and recursive function in the same sense as Shoenfield [24].

Definition 5 *A transition system* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ *is called* recursive *if* B *is a recursive set and there is a recursive function* $next : B \longrightarrow \mathcal{P}_{\mathrm{fin}}(B)$ *(where* $\mathcal{P}_{\mathrm{fin}}(B)$ *is the recursive set of finite subsets of* B*) such that* $a \rightarrow_{\mathcal{B}} b$ *iff* $b \in next(a)$*.*

A Kripke structure $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ *over* AP *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 $\rightarrow_{\mathcal{B}}$ is recursive, since then the set of next states of a given state would in general only be 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 [1], 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 underlying 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 finite signatures and sets of equations. In our approach, this is accomplished by specifying B as the carrier of a type 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 6 *Let* $\mathcal{R} = (\Sigma, E \cup A, R)$ *be a finitary rewrite theory such that all its rules are of the form*

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

with $\bigcup_i vars(p_i) \cup vars(q_i) \cup \bigcup_j vars(w_j) \subseteq vars(t) \cup vars(t')$*, and where either* $vars(t') \subseteq vars(t)$*, or, more generally, the rules* $(\dagger)$ *are* admissible *in the sense of [6]; that is, any extra variables not in* $vars(t)$ *can only be introduced incrementally by "matching equations" 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*[1] *A;*
2. *the equational theory* $(\Sigma, E \cup A)$ *is (ground)* Church-Rosser and terminating modulo *A [9]; and*
3. *the rules R are (ground)* coherent *[26] relative to the equations E modulo A.*

Notice that the rules that specify *Fpl*'s semantics in our example (see App. A) are admissible, and that only those labeled `Varm` contain matching equations (introduced with `:=`).

[1] 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 t to its (unique modulo A) canonical form $can_{E/A}(t)$ with respect to E before applying any of the rules. Then, if $\mathcal{R}$ is a recursive rewrite theory, it can be proven that 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 $\mathcal{T}(\mathcal{R})_k$ is recursive and if, in addition, 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, that is, each recursive system and Kripke structure can be specified this way. For lack of space we omit the proofs of these results and refer the reader to [20] for a detailed discussion.

At the price of allowing infinite signatures and losing computability, there is obviously a general representability result stating that *any* Kripke structure can be modeled in rewriting logic.

5 Algebraic Stuttering 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 stuttering simulations of Kripke structures. In previous works we did it by means of equational abstractions [19] and theoroidal morphisms [15]. Here we substantially extend those previous notions by considering two increasingly more general ways of defining stuttering simulations for rewrite theories that specify a concurrent system. In both ways we represent Kripke structures as rewrite theories; in the first case we represent stuttering *maps* as equationally defined *functions*, and in the second, more general case, we represent stuttering *simulations* as *rewrite relations*.

5.1 Stuttering Maps as Equationally Defined Functions

We define a category $\mathbf{SRWTh}_{\models}$ of rewrite theories that specify Kripke structures and stuttering *maps*, with the maps specified as equationally defined functions. We also define a subcategory $\mathbf{RecSRWTh}_{\models}$ where everything is recursive. For that we need to consider a theory $BOOL_{\models}$ extending $BOOL$ with two new types, *State* and *Prop*, and a new operator $\_ \models \_ : \mathit{State}\ \mathit{Prop} \longrightarrow \mathit{Bool}$.

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, there are both a system and a property specification levels involved in the description of a Kripke structure, namely the transition system level, and the level in which propositions are added. 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 type 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. $\mathcal{R} = (\Sigma, E, R)$ 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 [18] that selects the distinguished type of states $J(\mathit{State})$ and such that: (i) it is the identity when restricted to $BOOL$, (ii) $J(\mathit{Prop}) = \mathit{Prop}$, and (iii) $J(\_ \models$

$\_ : \mathit{State}\ \mathit{Prop} \to \mathit{Bool}) = \_ \models \_ : J(\mathit{State})\ \mathit{Prop} \to \mathit{Bool}$.

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 $(\Sigma', E \cup D) \supseteq (\Sigma, E)$ to be Church-Rosser and terminating.

What about morphisms? At the end of Sect. 4.2 we showed that any Kripke structure can be defined in rewriting logic. Likewise, it is obvious that any 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 $\mathcal{R}_{\mathcal{B}}$. Therefore, without loss of generality we can focus our attention on those stuttering maps that can be equationally defined. 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. (α, h) is a stuttering map $(\alpha, h) : \mathcal{K}(\mathcal{R}_1, J_1(\mathit{State}))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(\mathit{State}))_{\Pi_2}$.

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 $\alpha : \mathit{Prop}_1 \longrightarrow \mathit{StateForm}_2$ and $h : J_1(\mathit{State})_1 \longrightarrow J_2(\mathit{State})_2$ in Ω; the subscripts 1, 2 indicate the corresponding names for the disjoint copies of the types, and $\mathit{StateForm}_2$ is a new type for representing state formulas over Prop_2.

Note the existential quantifier in the second item; therefore, we do not need to choose any particular such extension to define the category: existence is enough. Since, by the general representability result, we can always find an extension (Ω, G) in which such a function h can be equationally defined, the category is well-defined, because for each composition we can do the same, perhaps with an extension unrelated to the extensions for each of the morphisms being composed.

The important point is that if α and h are *recursive*, and $\mathcal{K}(\mathcal{R}_1, J_1(\mathit{State}))_{\Pi_1}$ and $\mathcal{K}(\mathcal{R}_2, J_2(\mathit{State}))_{\Pi_2}$ are objects in $\mathbf{RecSRWTh}_{\models}$, then by the metaresult of Bergstra and Tucker [1] 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).

We can now show that the construction defined in Sect. 4.1 that associates a Kripke structure to a rewrite theory with a chosen type of states and chosen state predicates is actually a *functor*. More precisely, we define $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$, where $\mathbf{KSMap}$ is the global category of Kripke structures and stuttering maps, as follows:

- for objects, $\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(\mathit{State}))_{\Pi}$;

- 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 $\mathbf{RecKSMap}$ 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, we have the following result.

Proposition 3 *The functor $\mathcal{K}$ from $\mathbf{SRWTh}_{\models}$ to $\mathbf{KSMap}$ is an equivalence of categories. Similarly, $\mathcal{K} : \mathbf{RecSRWTh}_{\models} \longrightarrow \mathbf{RecKSMap}$ is also an equivalence.*

The fact that $\mathcal{K}$ is an equivalence of categories is 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).

5.2 The Example Revisited

To prove the correctness of the abstract machine implementation relative to the computation semantics we show that there exists a recursive algebraic stuttering map h between them.

Intuitively, $\langle \mathit{empty}, \rho, e\rangle$ should be related to $\langle \rho, e\rangle$. Consider this derivation:

$$\begin{aligned}\langle \mathit{empty}, \mathit{empty}, 2+3\rangle &\rightarrow_{\mathcal{A}} \langle \mathit{empty}, \mathit{empty}, 2.3.+\rangle \\ &\rightarrow_{\mathcal{A}} \langle 2, \mathit{empty}, 3.+\rangle \\ &\rightarrow_{\mathcal{A}} \langle 3.2, \mathit{empty}, +\rangle \\ &\rightarrow_{\mathcal{A}} \langle 5, \mathit{empty}, \mathit{empty}\rangle\end{aligned}$$

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 [12]. It seems appropriate, then, to relate them to the same state as the first one, namely $\langle \mathit{empty}, 2{+}3\rangle$. The situation is different for the last state: some information has been lost, and it seems more appropriate to relate this state to $\langle \mathit{empty}, 5\rangle$. This last step is an example of an *application rule*.

So we define $h : \mathcal{A} \longrightarrow \mathcal{C}$ by $h(a) = \langle \rho, e\rangle$ if a can be obtained from $\langle \mathit{empty}, \rho, e\rangle$ by zero or more applications of the analysis rules for the abstract machine together with `Valm` and `Locm2` (see App. A). Note that h is partial: it is only defined for reachable states, which constitute a full substructure of $\mathcal{A}$ where h is total.

Alternatively, h can be defined by means of the following set of equations.

```
eq [Base] : h(< empty,rho,e>) = < rho,e > .
eq [Opm1] : h(< S,rho,e . e' . op . C >) =
            h(< S,rho,e op e' . C >) .
eq [Opm1] : h(< S,rho,be . be' . bop . C >) =
            h(< S,rho,be bop be' . C >) .
eq [Ifm1] : h(< S,rho,be . if(e, e') . C >) =
      h(< S,rho,If be Then e Else e' . C >) .
eq [Locm1] : h(< S,rho,e. <x, e'> . C >) =
      h(< S,rho,let x = e in e' . C >) .
eq [Notm1] : h(< S,rho,be . not. C >) =
             h(< S, rho, Not be . C >) .
eq [Eqm1] : h(< S,rho,e . e' . equal . C >) =
            h(< S,rho,Equal(e, e') . C >) .
eq [Locm2] :
      h(< S,(x, v) . rho,e . pop . C >) =
      h(< v . S,rho,<x, e> . C >) .
eq [Valm] : h(< v . S,rho,C >) =
      h(< S,rho,v . C >) if not(enabled(C)) .
eq [Valm] : h(< bv . S,rho,C >) =
      h(< S,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 1 *If $h(\langle S, \rho, e.C\rangle) = \langle \rho, e'\rangle$, then there is a position p in e' such that $e'|_p = e$ and, if e is not a value, then e is a subexpression that can be reduced in e' by the rules of the computation semantics in the next step.*

Theorem 2 *$h : \mathcal{A} \longrightarrow \mathcal{C}$ is a recursive algebraic stuttering map.*

Then, by the preservation result in Theorem 1, for all expressions e and environments ρ, we have that $\mathcal{C}, \langle \rho, e\rangle \models \mathbf{AF}v$ implies $\mathcal{A}, \langle \mathit{empty}, \rho, e\rangle \models \mathbf{AF}v$. That is, $\mathcal{A}$ is a correct implementation of $\mathcal{C}$. Note that h is not a bisimulation. In the computation semantics, for an expression $e\ \mathit{op}\ e'$ we can choose whether to evaluate e before e' or vice versa, whereas the abstract machine always evaluates e first. That means that, for example, the transition $\langle \mathit{empty}, (1+2)+(3+4)\rangle \rightarrow \langle \mathit{empty}, (1+2)+7\rangle$ cannot be simulated by the abstract machine.

5.3 Stuttering Simulations as Rewrite Relations

The notion of map in Sect. 5.1, though already very general and applicable to many situations, is unsatisfactory in the sense that it restricts us to work only with functions. This drawback can be avoided by a simple extension of the ideas introduced above.

We define a category $\mathbf{SRelRWTh}_{\models}$ with the same objects as $\mathbf{SRWTh}_{\models}$. Now, 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 $\mathbf{SRelRWTh}_{\models}$, called an *algebraic stuttering simulation*, is a pair (α, H) such that:

1. (α, H) is a stuttering simulation of Kripke structures (α, H) :

$\mathcal{K}(\mathcal{R}_1, J_1(\mathit{State}))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(\mathit{State}))_{\Pi_2}$.

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 $\alpha : \mathit{Prop}_1 \longrightarrow \mathit{StateForm}_2$, and H is defined by rewrite rules through an operator $H : J_1(\mathit{State})_1\ J_2(\mathit{State})_2 \longrightarrow \mathit{Bool}$ such that xHy iff $\mathcal{R}_3 \vdash H(x, y) \longrightarrow \mathit{true}$.

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 [6]. That is, $\mathcal{R}$ satisfies requirements similar to those for a recursive rewrite theory, but the conditions of the rules can now contain rewrites as long as they do not have new variables on their lefthand sides. Note that this is equivalent to requiring H to be r.e.

It is worth mentioning that when we work with functions in $\mathbf{RecSRWTh}_{\models}$ we only consider recursive functions, whereas we 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.

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 4 *With the above definitions, the functor* $\mathcal{K} : \mathbf{SRelRWTh}_{\models} \longrightarrow \mathbf{KSSim}$ *is an equivalence of categories, and so is* $\mathcal{K} : \mathbf{RecSRelRWTh}_{\models} \longrightarrow \mathbf{RecKSSim}$

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.

5.4 A Communication Protocol Example

To emphasize the usefulness of the compositional approach, we illustrate it with an example that shows how a system can be successively abstracted until a finite one that can be model checked is reached.

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 [17] it is shown how this might be done and we slightly adapt it here, in a module `PROTOCOL`, to consider a system with messages `a`, `b`, and `c` of type `Elem`.

The system consists of a “soup” of type `Config` of senders, receivers, and messages, built with the following constructors (`Qid` is a type of quoted identifiers):

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

--- rec is the receiver, sendq is the
--- outgoing queue, sendbuff is either empty
--- or the current data, sendcnt is the
--- sender sequence number
op <_: Sender | rec:_, sendq:_ , sendbuff:_,
   sendcnt:_ > : Qid Qid List Contents Nat
                 -> Object .

--- sender is the sender, recq is the
--- incoming queue, and reccnt is the
--- receiver sequence number
op <_: Receiver | sender:_, recq:_,
   reccnt:_ > : Qid Qid List Nat -> Object .
```

The sender produces a message (we omit the rules for `b` and `c`) and keeps on broadcasting it, together with an identifying number, until it receives an acknowledgment from the receiver.

```
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 [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 > .
```

The receiver, in turn, waits for a message and sends an acknowledgment upon reception.

```
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) .
```

Under reasonable fairness assumptions, 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 using a `Destroyer` object in a module `PROTOCOL-FAULTY`.

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

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 > .
```

Messages may be erased by objects of class `Destroyer`. The first counter represents the identifying number of the messages they can destroy, and the second one is the number of remaining messages with that number that 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)` in Π which holds for a sender `S` and a receiver `R` whenever the queue associated to `R` is a prefix of that of `S`. This is done in both modules by means of:

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

eq < S : Sender | rec: R, sendq: L1 : L2,
     sendbuff: C, sendcnt: N >
   < R : Receiver | sender: S, recq: L1,
     reccnt: M >
   CO:Config |= 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, it should satisfy the formula **AG** `prefix('A, 'B)` for the initial state:

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

For a proof, we define a stuttering simulation

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

Given configurations (states) a in `PROTOCOL-FAULTY` and b in `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 that extends `PROTOCOL` and `PROTOCOL-FAULTY` as follows. In this specification, the types 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 3 *H : $\mathcal{K}($*`PROTOCOL-FAULTY`*$,$* `Config`*$)_\Pi \longrightarrow \mathcal{K}($*`PROTOCOL`*$,$*`Config`*$)_\Pi$ is an r.e. algebraic stuttering simulation.*

By Theorem 1, the existence of H shows that if **AG** `prefix('A, 'B)` is true in `PROTOCOL` then it must also hold in `PROTOCOL-FAULTY`; however, we must first prove that, and `PROTOCOL` is still an infinite state system. But `PROTOCOL` is now amenable to the equational techniques described in [19] and we can obtain a finite abstraction

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

by simply adding some equations to `PROTOCOL` (see [21] for the details of a very similar abstraction). Since `ABS-PROTOCOL` is finite we can prove the property using Maude's model checker; by composing G with H this also proves that the same property is true in `PROTOCOL-FAULTY`.

6 Concluding Remarks

We have introduced 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 presented general representability results showing that both Kripke structures and their simulations can be fruitfully represented in rewriting logic. As witnessed by the numerous references in [14], rewriting logic is a very flexible framework for the specification of a wide range of systems; we believe that the ideas presented in this work, and in particular the compositional approach, can be useful in order to deal with the complexity of those systems and to reduce them to finite ones where properties can be proved using Maude's model checker.

References

[1] 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, *ICALP'80*, LNCS 81, pages 76–90. Springer, 1980.

[2] 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.

[3] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[4] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten et al, editors, *ICALP 2003*, LNCS 2719, pages 252–266. Springer, 2003.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] 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.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.3). `http://maude.cs.uiuc.edu/manual/`, 2007.

[8] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Selected papers from PROLE 2005.

[9] 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.

[10] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *WRLA'02*, ENTCS 71. Elsevier, 2002.

[11] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[12] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.

[13] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, Aug. 2001.

[14] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[15] N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro et al, editors, *WADT 2004*, LNCS 3423, pages 126–143. Springer, 2005.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[17] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha et al, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[18] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT'97*, LNCS 1376, pages 18–61. Springer, 1998.

[19] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *CADE-19*, LNCS 2741, pages 2–16. Springer, 2003.

[20] J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. `http://maude.sip.ucm.es/~miguelpt/`, 2007.

[21] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. Extended version. Submitted. `http://maude.sip.ucm.es/~miguelpt/`, 2007.

[22] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In S. Ramesh and G. Sivakumar, editors, *FSTTCS'97*, LNCS 1346, pages 284–296. Springer, 1997.

[23] M. Palomino, J. Meseguer, and N. Martí-Oliet. A categorical approach to Kripke structures and simulations. In J. L. Fiadeiro et al, editors, *CALCO 2005*, LNCS 3629, pages 313–330. Springer, 2005.

[24] J. R. Shoenfield. *Degrees of Unsolvability.* North-Holland, 1971.

[25] 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.

[26] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

A Semantics of *Fpl*

Computation semantics for *Fpl*. (There is an analogous set of rules for Boolean expressions [20].)

```
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] > .
```

Analysis rules for the abstract 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 abstract 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, 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 > .
```