Synchronous products of rewrite systems*

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet
{omartins,jalberto,narciso}@ucm.es

Facultad de Informática
Universidad Complutense de Madrid, Spain

Abstract. We present and formalize a concept of synchronous product for rewrite systems, and also a corresponding concept for general transition systems, used as semantics for the former. A series of examples shows their practical usefulness: for the strategic control of systems, and for modular specification and verification.

1 Introduction

In this paper we propose a composition of rewrite systems by means of synchronous products. We also define a corresponding synchronous product on labeled transition systems, which allows to semantically ground the construction. The concept is akin to the one from automata theory—whence it borrows its name—, but also to the composition of processes in CCS [20], to request-wait-block threads in behavioral programming [11], and to other formalisms for module composition.

Our original aim was to model check systems specified in rewriting logic [17] (particularly, in Maude [6]) and controlled by strategies. Strategies, as a means to tame—guide, control—the nondeterminism of a system, have been studied and implemented in many systems [13], and also Maude has its strategies [15]. However, the current implementation of Maude strategies is oriented to producing a set of results: the final states the strategy allows to arrive to. As it focuses on final states, and not on the process, it is not suitable for model checking. To the best of our knowledge, no existent tool allows for model checking strategically controlled systems.

We propose synchronous products for this purpose. As we show in the examples below, a strategy can be coded into a rewrite system that exerts its control through a synchronous product. When a system is required to synchronize with another, some of its actions are prevented; thus, its execution is *guided*. Then, the resulting synchronous product, being a system on its own, is amenable to being model checked like any other system. This paper focuses on the definition of the synchronous product; its use for strategic control is introduced in the examples.

* Partially supported by MINECO Spanish project StrongSoft (TIN2012–39391–C04–04), Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731), and UCM-Santander grant GR3/14.

The examples also hint at the other, more general use of the synchronous product: looking at it as parallel composition of processes, it is a promising tool for modular specification and verification.

Synchronization involves transitions and states. Transitions are synchronized by the name of the action—the label of the rule, in the case of rewrite systems. If one system includes the rewrite rule $[\ell]\, t_1 \to t_1'$ and the other $[\ell]\, t_2 \to t_2'$, with the same label ℓ, the product system includes the rule $[\ell]\, \langle t_1, t_2 \rangle \to \langle t_1', t_2' \rangle$, that represents the simultaneous execution of rule ℓ in both systems. Rules that only exist in one system can be executed by themselves, with no effect on the other side.

Also states are required to synchronize: the states simultaneously visited by the operand systems must agree on the atomic propositions they satisfy. That is, if $s_1 \models p$ and $s_2 \not\models p$ for some proposition p, then $\langle s_1, s_2 \rangle$ is not even a state in the product system.

We do not require that each state decides on each atomic proposition. So to speak, the scope of propositions is not system-wide, but state-wide. An atomic proposition that is meaningless to a given state imposes no restriction for synchronization. The reasons for this choice have to do with the implementation of strategies we have in mind and are made clear in the examples, but it also provides a natural way to model some situations: "I have a strong opinion against Hollywood movies, and won't mate anyone with opposite taste; but I haven't made up my mind on Hungarian movies, so whatever is fine."

The rest of this paper is divided into five sections. Section 2 defines the synchronous product on labeled transition systems. Section 3 focuses on the same concept for rewrite systems and on their semantics. Section 4 shows some examples. These can be seen as motivating examples, and it could be profitable to give them a glance *before* entering the technical details in Sections 2 and 3. Section 5 discusses some issues having to do with the prototype implementation of the synchronous product that we have developed in Maude. Section 6 proposes directions for future work and mentions, at the same time, related literature.

This is an extended version of a paper to appear. The Maude code for our implementation and the examples can be found at our website: `http://maude.sip.ucm.es/syncprod`. The latest version of this paper can also be downloaded there.

2 Synchronous products of transition systems

We start at the semantic level, by defining the particular kind of labeled transition systems convenient to our discussion, and showing how they can be composed by the operation we call *synchronous product*.

2.1 Transition systems

Our transition systems slightly deviate from the standard definition in that we do not require that every state decides on every atomic proposition. The reasons for this choice are hinted at in the introduction and in the examples section. Thus, a transition system is given by a tuple $\mathcal{T} = (S, \Lambda, \delta, \mathrm{AP}, \mathcal{L}^+, \mathcal{L}^-)$. As usual, there is a set S of states, an alphabet Λ of actions, a nondeterministic transition function $\delta : S \times \Lambda \to 2^S$, and a set AP of atomic propositions on states. But there are two labeling functions instead of one: $\mathcal{L}^+, \mathcal{L}^- : S \to 2^{\mathrm{AP}}$. They return the sets of propositions on which the argument state decides on the affirmative ($\mathcal{L}^+$) and on the negative ($\mathcal{L}^-$). Thus, we require that $\mathcal{L}^+(s) \cap \mathcal{L}^-(s) = \emptyset$ for each state s, but we allow that $\mathcal{L}^+(s) \cup \mathcal{L}^-(s) \neq \mathrm{AP}$. The initial state is absent from the definition; we rather consider it as an attribute of each run through the system.

2.2 Synchronous products

The synchronous product is a way to define the composition of transition systems. Synchronization happens on transitions by their actions and on states by compatible atomic propositions. Formally, given two transition systems as above, $\mathcal{T}_i = (S_i, \Lambda_i, \delta_i, \mathrm{AP}_i, \mathcal{L}_i^+, \mathcal{L}_i^-)$, for $i = 1, 2$, their *synchronous product*, denoted $\mathcal{T}_1 \parallel \mathcal{T}_2$, is another system $(S, \Lambda, \delta, \mathrm{AP}, \mathcal{L}^+, \mathcal{L}^-)$ defined as follows:

- $S := \{\langle s_1, s_2\rangle \in S_1 \times S_2 \mid \mathcal{L}_1^+(s_1) \cap \mathcal{L}_2^-(s_2) = \emptyset = \mathcal{L}_1^-(s_1) \cap \mathcal{L}_2^+(s_2)\}$. That is, the opinions of s_1 and s_2 on their common propositions must be compatible; if the state on one side decides on a certain proposition while the state on the other side doesn't care, that's OK as well.
- $\Lambda := \Lambda_1 \cup \Lambda_2$.
- Regarding δ:
  - if $\lambda \in \Lambda_1 \cap \Lambda_2$, then $\delta(\langle s_1, s_2\rangle, \lambda) := (\delta_1(s_1, \lambda) \times \delta_2(s_2, \lambda)) \cap S$;
  - if $\lambda \in \Lambda_1 \setminus \Lambda_2$, then $\delta(\langle s_1, s_2\rangle, \lambda) := (\delta_1(s_1, \lambda) \times \{s_2\}) \cap S$;
  - if $\lambda \in \Lambda_2 \setminus \Lambda_1$, then $\delta(\langle s_1, s_2\rangle, \lambda) := (\{s_1\} \times \delta_2(s_2, \lambda)) \cap S$.

  The part "$\cap S$" is needed in all cases: the resulting states must comply with the *compatibility* condition in the definition of S.
- $\mathrm{AP} := \mathrm{AP}_1 \cup \mathrm{AP}_2$.
- $\mathcal{L}^+(\langle s_1, s_2\rangle) := \mathcal{L}_1^+(s_1) \cup \mathcal{L}_2^+(s_2)$.
- $\mathcal{L}^-(\langle s_1, s_2\rangle) := \mathcal{L}_1^-(s_1) \cup \mathcal{L}_2^-(s_2)$. Thus, for example, if s_1 has an opinion on a certain proposition while s_2 doesn't, it is s_1's opinion that prevails.

The definition allows $\Lambda_1 \cap \Lambda_2 = \emptyset$ or $\mathrm{AP}_1 \cap \mathrm{AP}_2 = \emptyset$. When both intersections are empty, the two systems progress independently: no synchronization is possible at all between actions, and any state can pair with any other.

We take for granted, in the definition and in the following, that synchronization takes place on actions and propositions with *equal names*, that is, on $\Lambda_1 \cap \Lambda_2$ and $\mathrm{AP}_1 \cap \mathrm{AP}_2$. This is no loss of generality, as renaming within a system—previous to the synchronous product—is always possible and harmless. The choice of notation, $\mathcal{T}_1 \parallel \mathcal{T}_2$, without explicit mention of the actions or propositions to be synchronized, is also according to the same proviso.

By way of comparison, in CCS, actions with complementary names can synchronize or not; syncing is only mandatory for restricted actions. In our case, we have actions that *must* synchronize (because they share names) and actions that cannot synchronize; no middle ground.

2.3 Equality of transition systems

We want to prove—and make use of—this straightforward statement:

Proposition 1. *The operator* $\|$ *is commutative and associative; that is, for all* $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$*:*

$$\mathcal{T}_1 \,\|\, \mathcal{T}_2 = \mathcal{T}_2 \,\|\, \mathcal{T}_1$$

and

$$(\mathcal{T}_1 \,\|\, \mathcal{T}_2) \,\|\, \mathcal{T}_3 = \mathcal{T}_1 \,\|\, (\mathcal{T}_2 \,\|\, \mathcal{T}_3).$$

The difficulty lies in establishing when we are willing to say that two systems are *equal*. The sets Λ and AP (of actions and atomic propositions) determine the behavior of a system with respect to the synchronous product. Thus, to be considered equal, two systems must share exactly the same such sets. The only difference we allow is in the names of the states, as they do not get bound by the synchronous product. The tree structure of the transition systems must be the same, even though their nodes—the states—may have different names. More formally:

Definition 1. *Two transition systems are considered equal if they have exactly the same alphabet of actions* Λ*, the same set of propositions* AP*, and there exists a bijection between the states that preserves the transition function* δ *and the labeling functions* $\mathcal{L}^+$ *and* $\mathcal{L}^-$*.*

This is a very strict concept of equivalence, next only to exact identity. There is certainly a way in which renaming actions and propositions does not change a system. Also, it is possible to define a concept of bisimilarity through the lines of [19]. But we do neither need nor want these broader concepts of equivalence here.

In the following we consider that systems equal in the sense of Definition 1 are the same, and use the equality symbol for them.

For Proposition 1, the bijection is given by identifying $\langle s_1, s_2\rangle$ with $\langle s_2, s_1\rangle$ (as states of $\mathcal{T}_1 \,\|\, \mathcal{T}_2$ and $\mathcal{T}_2 \,\|\, \mathcal{T}_1$, resp.), and $\langle\langle s_1, s_2\rangle, s_3\rangle$ with $\langle s_1, \langle s_2, s_3\rangle\rangle$ (as states of $(\mathcal{T}_1 \,\|\, \mathcal{T}_2) \,\|\, \mathcal{T}_3$ and $\mathcal{T}_1 \,\|\, (\mathcal{T}_2 \,\|\, \mathcal{T}_3)$).

We also have these results:

Proposition 2. *Consider a transition system with exactly one state, denoted by* $\bullet$*, no actions and no propositions; that is:*

$$(\{\bullet\}, \emptyset, \emptyset, \emptyset, \bullet \mapsto \emptyset, \bullet \mapsto \emptyset).$$

It is the identity element for $\|$*.*

Proof. Identify each s with $\langle s, \bullet \rangle$ and with $\langle \bullet, s \rangle$.

Proposition 3. *The $\|$ operation is idempotent. That is:*

$$\mathcal{T} \,\|\, \mathcal{T} = \mathcal{T}.$$

Proof. Identify each $\langle s, s \rangle$ with s.

3 Synchronous products of rewrite systems

We are interested in implementing and practically using synchronous products. Thus, we now reflect the abstract definitions of the previous section into the more concrete realm of rewriting logic.

3.1 Rewrite systems

Rewriting logic takes on the concept of term rewriting and tailors it to the specification of concurrent and non-deterministic systems. It was introduced as such in [17]. Maude [6] is a language for specification and programming based on this idea. A specification in rewriting logic contains equations and rewrite rules. Equations work much like in functional programming; rules describe the way in which a system state evolves into a different one.

Maude's flavor of rewriting logic is based on order-sorted membership equational logic (see [18], for instance). Thus, a rewrite system has the form $\mathcal{R} = (\Sigma, E \cup Ax, M, R)$, where: Σ is a signature containing declarations for sorts, subsorts, and operators; E is a set of equations; Ax is a set of equational axioms for operators, such as commutativity and associativity; M is a set of membership axioms; and R is a set of rewrite rules.

We are after rewrite systems whose semantics naturally yield transition systems as defined in Section 2. In particular, we need propositions on states, which are not, in principle, an ingredient of rewrite systems. Thus, we require of each rewrite system the list of properties below. Some of them are just syntactic conventions; others are deeper. Comments follow after the list. Each rewrite system $\mathcal{R} = (\Sigma, E \cup Ax, M, R)$ we use has to satisfy these properties:

- it is computable;
- it is topmost;
- all the rules in R bear labels, that is, $[\ell]\, s \to s'$ is a typical element of R;
- the sort of the terms we are interested in rewriting is called `State`;
- Σ includes a sort `Prop` of atomic propositions, all whose constructors are constants;
- $\mathcal{R}$ includes the definition of a theory of the Booleans declaring, in particular, the sort `Bool`, and constants `true` and `false`;
- Σ includes an infix relation symbol $\_ \models \_ : \texttt{State} \times \texttt{Prop} \to \texttt{Bool}$, and E includes equations defining $\models$, though not necessarily yielding `true` or `false` on every pair of arguments.

Computability of a rewrite system is formally defined, for instance, in [19]. For arbitrary sets of equations and rules, the rewriting relation between the terms of a system is undecidable. But for computable systems it is effectively decidable. The conditions are easy to meet. In Maude, all rewrite systems a sensible programmer would code are computable.

A topmost rewrite system is one in which all rewrites happen on the whole state term—not on its subterms. Formally, this is guaranteed by requiring that all rules involve terms of sort `State`, and that the sort `State` does not appear as argument in any constructor (so that no term of sort `State` can be subterm of another term of the same sort). The aim of this requirement is that all rules preserve their meaning through composition. For instance, the rule $a \to a'$ would rewrite the term $f(a)$ to $f(a')$, because a is a subterm of $f(a)$; but the composed rule $\langle a, t\rangle \to \langle a', t'\rangle$ would not rewrite the composed term $\langle f(a), s\rangle$, whatever s and t could be, because $\langle a, t\rangle$ is not a subterm of $\langle f(a), s\rangle$. Many rewrite systems are topmost or can be easily transformed into an equivalent one that is formally similar and topmost [9].[1]

The requirement that all rules have labels is natural, and not restrictive. It is convenient for the definition of the synchronous product, but it could be dropped if we agree, for instance, in that unlabeled rules do never synchronize. Note that, on the other hand, we do not prevent that several rules in the same system have the same label.

The name `State` is just a convention. This, together with the other requirements about `Prop`, `Bool`, and $\models$, is the standard way to introduce propositions in rewrite systems. It is the same setting needed to use Maude's LTL model checker. However, as we do not need that each state decides on each proposition, we do not ask that all expressions of the form $t \models p$ are equationally reducible to `true` or `false`, as is desirable for model-checking purposes.

For reasons to be made clear later, having to do with the implementation of the synchronous product (see Section 5), we want to have only a finite number of atomic propositions. The requirement that all terms of sort `Prop` are constants is a way to achieve that.

The above list of properties specifies names for sorts and constants, and they are the same for every system. This does not mean that the sorts are the same. Indeed, we interpret that any two rewrite systems have disjoint signatures. Names for sorts, constants, and the other elements must be understood within the scope, or namespace, of their respective systems. When needed, we qualify a name with a subscript showing the system it belongs to: $\texttt{State}_{\mathcal{R}}$. For implementation purposes, disjoint signatures are not the best choice, as we most probably

[1] Indeed, for any rewrite system there exists an equivalent topmost one. The Turing-completeness of term rewriting implies that for a given system it is possible to equationally define a function "next" that produces all the states accessible from a given one. Thus, the topmost rewrite system with the single rule "$s \to s'$ if $s' \in \text{next}(s)$" is topmost and equivalent to the given one. Equivalence is meant here in the strong sense of having the same state terms and the same transitions between them.

would like to have a single sort `Bool`, for instance. We take care of this practical problem in the implementation, as explained in Section 5.

3.2 Semantics

Given a rewrite system as above, $\mathcal{R} = (\Sigma, E \cup Ax, M, R)$, its semantics is a transition system, $\mathcal{T} = (S, \Lambda, \delta, \mathrm{AP}, \mathcal{L}^+, \mathcal{L}^-)$, obtained by extending the usual term-algebra semantics (see [17], for instance) in this way:

- $S := T_{\Sigma/E \cup Ax, \texttt{State}}$, the set of terms of sort `State` modulo equations (note that "being of a given sort" includes satisfying the membership axioms);
- Λ is the set of labels appearing in rules from R;
- δ is the transition relation generated by rewriting with the rules from R [17];
- $\mathrm{AP} := T_{\Sigma/E \cup Ax, \texttt{Prop}}$, the set of terms of sort `Prop` modulo equations;
- $\mathcal{L}^+(s) := \{p \in \mathrm{AP} \mid s \models p =_{E \cup Ax} \texttt{true}\}$;
- $\mathcal{L}^-(s) := \{p \in \mathrm{AP} \mid s \models p =_{E \cup Ax} \texttt{false}\}$.

3.3 The synchronous product

Given two rewrite systems as above, $\mathcal{R}_i = (\Sigma_i, E_i \cup Ax_i, M_i, R_i)$, for $i = 1, 2$, their *synchronous product*, denoted $\mathcal{R}_1 \parallel \mathcal{R}_2$, is a new rewrite system $\mathcal{R} = (\Sigma, E \cup Ax, M, R)$ as specified below.

As pointed out at the end of Section 3.1, each system works as its own namespace, so that the signatures Σ_1 and Σ_2 are naturally disjoint, as are the sets of equations, axioms, and rule labels. Equations, in particular, are included verbatim from each system into the product; any equational deduction valid in one of the systems is still valid in the product. Instead, rules from the operand systems are not included verbatim, but synchronized.

Rule labels and terms of sort `Prop` are compared by their naked names, and those names remain as such in the product system. As also discussed in Section 2.2, we assume that renaming has previously taken place if needed, so that synchronization happens on the set of rule labels and the set of atomic propositions common to both systems.

This is the definition of the synchronous product:

- $\Sigma := \Sigma_1 \uplus \Sigma_2 \uplus \Sigma'$, where Σ' contains:
  - declarations for sorts $\texttt{State}_{\mathcal{R}}$, $\texttt{Prop}_{\mathcal{R}}$, and $\texttt{Bool}_{\mathcal{R}}$;
  - a new constructor symbol

$$\langle \_, \_ \rangle : \texttt{State}_{\mathcal{R}_1} \times \texttt{State}_{\mathcal{R}_2} \to [\texttt{State}_{\mathcal{R}}],$$

    where the square brackets around $\texttt{State}_{\mathcal{R}}$ mean that the constructor is a partial function, that is, some pairs of the argument sorts are not elements of the result sort (because of the membership axiom described below);

  - constructors to make the set of names in the universe of $\texttt{Prop}_{\mathcal{R}}$ the union of the ones in $\texttt{Prop}_{\mathcal{R}_1}$ and $\texttt{Prop}_{\mathcal{R}_2}$, namely: a `Prop` constructor for each `Prop` constructor that appears in any system (or in both of them with the same name).
- $E := E_1 \uplus E_2 \uplus E'$, where E' contains these conditional equations (for the sake of clarity, just in this paragraph, we use subscripts i instead of $\mathcal{R}_i$, and omit the subscript $\mathcal{R}$):
  - $\langle x_1, x_2\rangle \models p = \texttt{true}$ if $x_1 \models_1 p_1 = \texttt{true}_1 \wedge \langle x_1, x_2\rangle : \texttt{State}$,
  - $\langle x_1, x_2\rangle \models p = \texttt{false}$ if $x_1 \models_1 p_1 = \texttt{false}_1 \wedge \langle x_1, x_2\rangle : \texttt{State}$,
  - $\langle x_1, x_2\rangle \models p = \texttt{true}$ if $x_2 \models_2 p_2 = \texttt{true}_2 \wedge \langle x_1, x_2\rangle : \texttt{State}$,
  - $\langle x_1, x_2\rangle \models p = \texttt{false}$ if $x_2 \models_2 p_2 = \texttt{false}_2 \wedge \langle x_1, x_2\rangle : \texttt{State}$.

  In these equations, and also below, x_1 and x_2 are variables of sorts $\texttt{State}_1$ and $\texttt{State}_2$, resp. These four equations are not, in general, disjoint. However, as explained above and formalized just below, for $\langle x_1, x_2\rangle$ to be in $\texttt{State}_{\mathcal{R}}$ (as the condition in all equations requires), the terms $x_1 \models_1 p_1$ and $x_2 \models_2 p_2$ cannot yield conflicting values. Thus, when two equations are enabled, it does not matter which one is used. (That is, the equational system by itself would not be confluent, but the membership equational system is.)
- $Ax := Ax_1 \uplus Ax_2$.
- $M := M_1 \uplus M_2 \uplus M'$, where M' contains just a membership axiom stating that $\langle x_1, x_2\rangle$ has sort $\texttt{State}_{\mathcal{R}}$ (written throughout this paper as $\langle x_1, x_2\rangle : \texttt{State}_{\mathcal{R}}$) if, for each name p that exists both in $\texttt{Prop}_{\mathcal{R}_1}$ and in $\texttt{Prop}_{\mathcal{R}_2}$, it is always the case that $x_1 \models_{\mathcal{R}_1} p$ and $x_2 \models_{\mathcal{R}_2} p$ have compatible values (where non-compatible means one `true` and the other `false`). Section 5 contains some discussion about the implementation of this axiom.
- R is composed by the following set of rules:
  - for each rule label ℓ that exists in both systems, say $[\ell]\, t_i \to t_i' \in R_i$, we have in R the conditional rule

$$[\ell]\, \langle t_1, t_2\rangle \to \langle t_1', t_2'\rangle \text{ if } \langle t_1', t_2'\rangle : \texttt{State}_{\mathcal{R}};$$

  - for each rule label ℓ that exists in R_1 but not in R_2, say $[\ell]\, t_1 \to t_1' \in R_1$, we have in R the conditional rule

$$[\ell]\, \langle t_1, x_2\rangle \to \langle t_1', x_2\rangle \text{ if } \langle t_1', x_2\rangle : \texttt{State}_{\mathcal{R}};$$

  - correspondingly for rule labels in R_2 but not in R_1.

  In all cases, the membership condition ensures that the landing state is valid. When there are no common propositions, the condition is trivially true, and we usually omit it.

Let "sem" denote the *semantics operator*, which assigns to each rewrite system a transition system as explained in Section 3.2. Let equality of transition systems be understood as defined in Section 2.3. All previous definitions have been chosen so that the following result holds.

Theorem 1. *For any rewrite systems $\mathcal{R}_1$ and $\mathcal{R}_2$, we have that*

$$\text{sem}(\mathcal{R}_1 \,\|\, \mathcal{R}_2) = \text{sem}(\mathcal{R}_1) \,\|\, \text{sem}(\mathcal{R}_2).$$

Proof. First, the alphabets of actions have to be the same. The semantic operator transforms rule labels into action identifiers, and both $\|$ operators produce unions—of rule labels in a case, of actions in the other. And that's it.

Second, the sets of atomic propositions also have to be the same. We try to reason as above. The semantic operator transforms terms of sort `Prop` into atomic propositions, and both $\|$ operators produce unions—of terms in a case, of propositions in the other. However, it is not really terms, but equational classes of terms of sort `Prop`, that are transformed into propositions. We need that the classes are preserved by the synchronous product. This is the case, as the equations and axioms concerning `Prop` in the product are just the union of those in the operand systems.

Third, we need a bijection between sets of states. The explanation for `Prop`—including the point about classes of terms—works also for `State`, even though the operation of interest here is not union, but Cartesian product (the constructor $\langle \_, \_ \rangle$ in one case, the standard set-theoretic Cartesian product in the other). The novelty is that both $\|$ operators produce subsets of the Cartesian products. The filters, in both cases, can be interpreted as the already discussed compatibility of opinions on common propositions, and they indeed are designed to be corresponding filters.

Finally, transition and labeling functions for the composition of transition systems are defined in similar ways to composed rules and the $\models$ operator for the composition of rewrite systems. □

4 Strategies and other examples

The substantive observation that we want to put forward is that composition with synchronization can be used as a means for controlling a process with another one, made up for that purpose. It is the kind of control that strategies are meant for.

We are interested in rewrite systems and their composition by means of synchronous products, but the observation is valid for other formalisms as well. Take, for instance, CCS [20]. Suppose that p is a process that can perform actions a, b, and c in some prescribed way. We want to restrict its freedom so that a and b are interleaved, irrespective of the possible occurrences of c. We can achieve so by defining the process $s := \overline{a}.\overline{b}.s$ and considering the composition $(p \,|\, s) \setminus \{a, b\}$ (let's accept we want a to happen first). Rewriting logic, with its greater complexity, provides more flexibility than CCS.

In this section we show examples of synchronous products of rewrite systems; particularly, but not only, of systems made up to control others. All of them are downloadable from our website: `http://maude.sip.ucm.es/syncprod`. Many of the examples build on previous ones. Indeed, the first one does not involve any synchronization, but it uses modular specification, and serves as basis for subsequent ones.

4.1 Modular specification

Consider this sketchy implementation of a railway in Maude:

```
mod RAILWAY1 is
   including SATISFACTION .  --- declares State, Prop, and |=.
   ops waiting crossing to-station in-station from-station : -> State .
   rl [t1wc] : waiting => crossing .
   rl [t1ct] : crossing => to-station .
   rl [t1ti] : to-station => in-station .
   rl [t1if] : in-station => from-station .
   rl [t1fw] : from-station => waiting .
endm
```

We can picture it as a closed loop railway with a station and a crossing with another railway. Indeed, we model this other railway in the same way and call it `RAILWAY2`. The rule names in this new system have a `2` instead of a `1`. (Our framework does not allow for parametric modules.)

The whole system is given by `RAIL` := `RAILWAY1` ‖ `RAILWAY2`, with rules like:

```
rl [t1wc] : < waiting, T2 > => < crossing, T2 > .
```

with `T2` a variable of sort `State` in `RAILWAY2`. Even though no synchronization is possible (all rule labels are different and there are no propositions, so that we omit the trivial membership condition) the modular specification is simpler and more natural.

With this specification, both trains are allowed, but not mandated, to wait before the crossing. They need to be controlled to avoid crashes.

4.2 Safety control by synchronizing actions

We want to control the whole system so as to ensure that trains do not cross simultaneously. Consider this controller system:

```
mod SAFETY is
   including SATISFACTION .  --- declares State, Prop, and |=.
   ops none-crossing one-crossing : -> State .
   rl [t1wc] : none-crossing => one-crossing .
   rl [t2wc] : none-crossing => one-crossing .
   rl [t1ct] : one-crossing => none-crossing .
   rl [t2ct] : one-crossing => none-crossing .
endm
```

Note that the rule labels used are some of the ones appearing in `RAILWAY1` and `RAILWAY2`. The rules ensure that from state `one-crossing` only transitions out of the crossing are allowed. The system `RAIL` ‖ `SAFETY` behaves as desired.

The rules of the composed system have, for example, this shape:

```
rl [t1wc] : < < waiting, T2 >, none-crossing > =>
            < < crossing, T2 >, one-crossing > .
```

This is certainly equivalent to

```
crl [t1wc] : < waiting, T2 > => < crossing, T2 > if T2 =/= crossing .
```

But to obtain this latter one we would need to modify `RAIL`—not extending, but overwriting it. This external control is at the heart of the concept of strategy.

This example showed synchronization on actions; the next focuses on states.

4.3 Safety control by synchronizing states

Here is another way to accomplish the same effect as in the previous example. Extend the original system `RAIL` with the following lines, declaring and defining the atomic proposition `safe` to hold when at least one train is out of the crossing:

```
op safe : -> Prop .
eq < crossing, crossing > |= safe = false .
eq < T1, T2 > |= safe = true [owise] .
```

The new controller system we propose has a single state, named `o`, and no rules:

```
mod SAFETY2 is
   including SATISFACTION .  --- declares State, Prop, and |=.
   op o : -> State .
   op safe : -> Prop .
   eq o |= safe = true .
endm
```

Consider `RAIL` $\|$ `SAFETY2`. A typical rule in this composed system is

```
crl [t1wc] : < < waiting, T2 >, X > =>  < < crossing, T2 >, X >
   if < < crossing, T2 >, X > : State .
```

It is not too different from the original `t1wc`. But there is the membership axiom:

```
cmb < < T1, T2 >, o > : State if < T1, T2 > and o agree on safe .
```

The condition, of course, must be expanded to a valid Boolean expression (that we prefer to omit here) involving the satisfaction relation from both systems. As `o` is always `safe`, that entails that if `< crossing, T2 >` is not `safe`, the term `< < crossing, T2 >, o >` is not a `State`, and the rule `[t1wc]` does not apply. So, `SAFETY2` restricts `RAIL` to visit only `safe` states, as desired.

In these two simple examples, writing the control in Maude is arguably as easy as using specific strategy languages. But such languages should be profitable in many cases, and the translation from them to Maude modules is high on our to-do list.

4.4 No hiding of synchronized actions' names

Now that safety is guaranteed, experts have decided that for a better performance of the public transport network, it is worth letting two trains pass through way 1 for each one passing through way 2. This can be achieved by a synchronous product of (`RAILWAY1` $\|$ `RAILWAY2`) $\|$ `SAFETY` with this new system:

```
mod PERFORMANCE is
   including SATISFACTION .  --- declares State, Prop, and |=.
   ops 0cross 1cross 2cross : -> State .
   rl [t1wc] : 0cross => 1cross .
   rl [t1wc] : 1cross => 2cross .
   rl [t2wc] : 2cross => 0cross .
endm
```

This *accumulated control* is possible because synchronized rules in `RAIL` $\|$ `SAFETY` keep their names and are still visible from the outside. Also, this persistence of names allows for the same model checking—same formulas, same atomic propositions—to be performed on the controlled system as on the original one.

In contrast, in CCS, when two actions synchronize, they give rise to an *internal action*, represented as τ. The original actions' names are forgotten. We do not want this to happen. Consequently, in Section 2.2, we defined $\Lambda := \Lambda_1 \cup \Lambda_2$, while from a CCS point of view we should have defined $\Lambda := (\Lambda_1 \cup \Lambda_2 \cup \{\tau\}) \setminus (\Lambda_1 \cap \Lambda_2)$.

Note that the product system `SAFETY` $\|$ `PERFORMANCE` is meaningful by itself: it is a system that, when composed with any uncontrolled implementation of the railway crossing (using the same rule labels), guarantees both safety and performance.

4.5 States need not have an opinion

Let us forget for a minute about the public transport system. Suppose, in a more abstract style, that we have a Maude module called `ORIGINAL`. We don't really mind about its inner workings, but it includes rules with labels `a` and `b`. Also, atomic propositions `P` and `Q` are declared and defined on `ORIGINAL`'s states. Suppose we want to control `ORIGINAL` in the way expressed by the following ω-regular expression with tests:

$$(\mathtt{a}\,;\,\mathrm{test}(\mathtt{P})\,;\,\mathtt{b})^{\omega}.$$

That is, we want to ensure that actions `a` and `b` are interleaved and also that after performing `a` the system lands on a state satisfying `P`.

Let us write this control as Maude code:

```
mod CONTROL is
   including SATISFACTION .  --- declares State, Prop, and |=.
   ops after-a after-b : -> State .
   op P : -> Prop .
   rl [a] : after-b => after-a .
   rl [b] : after-a => after-b .
   eq after-a |= P = true .
endm
```

There are two states: `after-a`, that only allows `b` as next action, and `after-b`, that only allows `a`. The controlled system `ORIGINAL` $\|$ `CONTROL` can only perform action `a` if the landing state agrees with `after-a`, that is, satisfies `P`.

The point to note is that the state `after-b` has no opinion on `P`. If it had, it would mandate the state on `ORIGINAL` to have the same opinion, which is not needed: after action `b`, it does not matter whether `P` holds or not. That is why we need to allow partial labeling in our systems, so that not every state needs to decide on every proposition, or that $\mathcal{L}^+(s) \cup \mathcal{L}^-(s)$ may be different from the whole set AP.

4.6 Instrumentation

Instrumentation is the technique of adding to the specification of a system some code in order to get information about the system's execution: number of steps, timing, sequence of actions... To some extent, it can be done by using synchronous products.

The trains are back. We want to keep track of the number of crossings for each one.

For `RAILWAY1` we propose this very simple system:

```
mod COUNT1 is
   including SATISFACTION .  --- declares State, Prop, and |=.
   including NAT .
   subsort Nat < State .
   var N : Nat .
   rl [t1wc] : N => N + 1 .
endm
```

A state of `RAILWAY1` $\|$ `COUNT1` is a pair whose second component is the counter. The initial state must be `< in-station, 0 >` (if `in-station` was the initial state for `RAILWAY1`). The same can be done to `RAILWAY2`. Then, the instrumented systems can be controlled in any of the ways described above.

4.7 Modular model checking

The module `SAFETY2`, shown in Section 4.3, when synchronized with another one, guarantees that all its states are `safe`, whatever that means in the other system. (Well, rather, that they are not un`safe`.) Seen from another point of view, it guarantees that the LTL formula $\mathbf{G}\,\mathtt{safe}$ holds.

The module `SAFETY`, shown in Section 4.2, when synchronized with another one, guarantees an interleaving between rules {`[t1wc]`, `[t2wc]`}, on the one hand, and {`[t1ct]`, `[t2ct]`}, on the other, whatever those rules do to the other system. The corresponding temporal formula—easily interpretable as mutual exclusion—can be expressed in the language of some logic of actions, like TLR* [19]. Taking `wc` as shorthand for `t1wc` $\vee$ `t2wc` and `ct` as shorthand for `t1ct` $\vee$ `t2ct`, this is the formula:

$$\mathbf{G}((\mathtt{wc} \rightarrow (\neg\mathtt{wc}\,\mathbf{U}\,\mathtt{ct})) \wedge (\mathtt{ct} \rightarrow (\neg\mathtt{ct}\,\mathbf{U}\,\mathtt{wc}))).$$

Consider also this new module:

```
mod DEKKER is
   including SATISFACTION .  --- declares State, Prop, and |=.
   sorts Waiting Turn .
   ops none-w one-w two-w : -> Waiting .
   ops turn-1 turn-2 : -> Turn .
   op (_,_) : Waiting Turn -> State .
   rl [t1wc] : (one-w, T:Turn) => (none-w, turn-2) .
   rl [t2wc] : (one-w, T:Turn) => (none-w, turn-1) .
   rl [t1wc] : (two-w, turn-1) => (one-w, turn-2) .
   rl [t2wc] : (two-w, turn-2) => (one-w, turn-1) .
   rl [t1fw] : (none-w, T:Turn) => (one-w, T:Turn) .
   rl [t2fw] : (none-w, T:Turn) => (one-w, T:Turn) .
   rl [t1fw] : (one-w, T:Turn) => (two-w, T:Turn) .
   rl [t2fw] : (one-w, T:Turn) => (two-w, T:Turn) .
endm
```

It can be used to guarantee absence of starvation. In TLR*-like syntax, it satisfies

$$\mathbf{G}(\mathtt{t1fw} \rightarrow \mathbf{F}\,\mathtt{t1wc}) \wedge \mathbf{G}(\mathtt{t2fw} \rightarrow \mathbf{F}\,\mathtt{t2wc}).$$

The product `SAFETY` $\|$ `DEKKER` implements a Dekker-style algorithm.

In all these three systems, proving the property (be it mutual exclusion or whatever) for the component that implements it is enough to prove it for the composed system. Care is needed, however, as it is not always the case that a property is preserved by composition. See discussion about modularity in Section 6.

5 Notes on the implementation

We have coded a prototype implementation of the synchronous product in Maude. It can be downloaded from our website: `http://maude.sip.ucm.es/syncprod`. It largely follows the lines of the explanations in Section 3.3. Some details, however, are worth discussing.

Maude's metalevel We want a program that takes as arguments two Maude modules and produces a new one containing their synchronous product. Our program has to handle rules, equations, labels and so on. Even complete modules have to be treated as objects by the program we seek. It turns out that the best tool for this second-order programming task is Maude itself.

The Maude language provides a bunch of metalevel functions for this purpose. The function `getRls`, to name just an example, takes as argument a module and returns its set of rules. Modules, rules, and the rest of Maude's syntactic constructs must be *meta-represented* for these metalevel functions to be able to handle them. That is, they cease to be Maude code and become terms of sorts `Module`, `Rule`, and so on. Maude provides functions to perform such meta-representation. For instance, the `upModule` function takes as argument the name of a Maude module and returns a term of sort `Module` that represents it.

We have chosen this as the natural way to the implementation. We have coded a Maude function `syncprod` that takes two terms of sort `Module` and produces another one that represents their synchronous product.

The membership axiom The need for a membership axiom in the synchronous product was explained in Section 3.3. The axiom was stated like this: "$\langle x_1, x_2 \rangle$ has sort $\texttt{State}_{\mathcal{R}}$ if, for each name p that exists both in $\texttt{Prop}_{\mathcal{R}_1}$ and in $\texttt{Prop}_{\mathcal{R}_2}$, it is always the case that $x_1 \models_{\mathcal{R}_1} p$ and $x_2 \models_{\mathcal{R}_2} p$ have compatible values". It includes a universal quantification on propositions, and this is somewhat problematic.

The set of terms of sort `Prop` in a system can be infinite, and each term can be arbitrarily complex. We need an effective way of deciding the axiom's condition based on such a set of terms. We have decided to restrict our implementation to a finite number of `Prop`s. As was explained in Section 3, we require our systems to have only constants as `Prop`s. That way, the list is surely finite and can be traversed in a standard recursive way.

Name clashes We discussed in Section 3.1 that names `State`, `Prop`, `Bool`, and so on are required to appear in each operand system, and in the resulting system as well. In the theoretical description we assumed each occurrence of them to be qualified by their different scopes. In practice, there are three cases to be considered:

- Sorts such as `Bool` and `Nat`, and their operators, are most probably going to be defined and used in the same way in every system. Keeping several copies of them would not harm, but is pointless.
- The sort `State` for the resulting system is defined as pairs of operand `State`s. Thus, all three `State` sorts need to be present in the resulting system, with different names. The same applies to the operator `|=`, whose definition uses the corresponding operators from each system.
- The sort `Prop` is somewhat special in that we identify elements with the same name in the three systems. Having just one sort `Prop` makes things easier.

This is what our implementation does: First, for each operand module, it renames its sort `State` to `StateModName`, if `ModName` is the name of the module; also, it renames the satisfaction symbol from `|=` to `|=ModName`. Once this is done for both operand modules, the union of them is computed, thus leaving only one sort `Prop`, and also one sort `Bool`, and so on. A fresh sort `State` and a fresh operator `|=` are then declared. The just mentioned union affects declarations and equations, but not rules, that are individually computed in their composed forms. The membership axiom is finally put in place as explained.

Full Maude As explained in Section 5, we have coded a function `syncprod`. However, it can only be invoked at the metalevel, that is, feeding it not with two Maude modules, but with two objects of sort `Module`. A decent implementation must allow for a simpler use, like writing `including MODULE1 || MODULE2` in the import section of a module. For those acquainted with Maude, the tool of choice for such a task is Full Maude.

Full Maude [7,6] is a re-implementation of the Maude interpreter using Maude itself. It is adaptable and extensible, and allows the definition of new module expressions, as we need. We have extended Full Maude to include an operator `||` on modules to represent the synchronous product. A specification containing `including MODULE1 || MODULE2` can refer to any of the constructs of the synchronous product, like pairs of states, propositions inherited from the operand systems, and so on. For example, this is taken from the examples available at http://maude.sip.ucm.es/syncprod:

```
(mod RAIL is
   including RAILWAY1 * RAILWAY2 .
   op safe : -> Prop .
   eq < crossing, crossing > |= safe = false .
   eq < T1:StateRAILWAY1, T2:StateRAILWAY2 > |= safe = true [owise] .
endm)
```

6 Related and future work

This paper contains definitions and motivating examples. As such, it is the ground on which interesting work is still to be done. Let's be more concrete.

Strategies. We expect to be able to develop automatic translations from some strategy languages—starting from simple regular expressions—to equivalent Maude modules. We also expect that, in some cases, we will need restrictions on the systems to be controlled or on the strategies to be applied for our technique to work.

From its origin in games, the concept of strategy, under different names and in different flavors, has become pervasive, particularly in relation to rewriting (see the recent and excellent survey [13]). Maude [6] includes flexible strategies for the evaluation of terms (like lazy, innermost and so on), and external implementations have been proposed in [15] and in [23]. ELAN [3], Tom [2], and Stratego [24] include them built-in. They also appear in graph rewriting systems (see references in [13] and also [21], where they are called just *programs*). In theorem provers, they allow the user to guide the system towards the theorem, or they represent the whole proof once found. Our implementation will be tested against these uses, to see which ones it is able to handle.

The very concept of strategy that is useful to us needs elucidation. Is it just a path filter, that prunes out some branches from the original system? Or should we allow for a strategy to introduce new behaviors? A valuable feature of runtime verification tools, for instance, is their ability to take the system to a safe state when something dangerous is found.

Model checking. Since the foundational book [4] proposed the verification of a system by exhaustively visiting all its possible states and paths, the literature on model checking has become huge. We are interested in model checking strategically controlled systems. Once the concept of control through synchronous products is in place, existing tools can be used on the resulting system (particularly, for us, Maude's LTL model checker [8]). We need to explore ways to make this whole process easier, specially through the use of Full Maude.

The nearest works to this we are aware of are GP2, that includes Hoare-style verification in the context of graph rewriting [22], and the BPmc prototype tool for model checking behavioral programs in Java [10].

Runtime verification. The controlled and the controller systems are run side by side in the synchronous product, the one accompanying the other. This is very much the idea of runtime verification (see [12], for instance), and our examples on railway safety can easily be seen from this point of view. Additional ingredients of runtime verification, like drawing conclusions about the entire system by inspecting just a run, and taking the system to a safe state whenever danger is found, can probably be accommodated into our framework with uncertain ease.

Modularity. Modular systems are easier to write, read, and verify. For the writing phase, the separation of concerns among modules has great simplifying power: one module implements the base system, another ensures mutual exclusion, and yet another deals just with starvation. Compare to this scheme to present mutual exclusion algorithms:

```
loop {
   get permission to enter
   perform critical-section actions
   notify exit
   perform non-critical-section actions
}
```

This specification, once expanded, mixes the proper actions of the system with the mutual exclusion control. An attractive possibility is that of providing the specifier with a library of pre-manufactured modules ready to be used (through synchronous product) for specific tasks. For ensuring mutual exclusion, for instance, one could readily choose among `SAFETY` or `SAFETY2` or some other. All these possibilities are illustrated in the examples section.

Model checking modular systems can be more efficient, given that the state space of the composed system is of the order of the product of the individual sizes. Also, the library modules referred to above would be verified once and for ever, removing the need to verify those properties for each individual system that uses them.

But some care is needed. The system `SAFETY` (see example in Section 4.2) satisfies mutual exclusion; the system `RAIL` satisfies its negation; the product system `RAIL` $\|$ `SAFETY` preserves that property from `SAFETY`, not from `RAIL`. The precise reasons for this behavior are described in the literature (see next paragraph). We need to take them into account for modular verification.

Much work already exists on modular model checking and verification. Our definition of the synchronous product is partially based on the "composition of modules" from [14]. See also [5] for a different point of view. But not many tools allow for them and, to the best of our knowledge, no implementation on rewriting logic has been developed. Ideally, advances in this direction will be useful to other verification tools and frameworks.

Behavioral programming. Based on the idea that a system can be decomposed into several synchronized threads, each of them implementing a behavior of the system, behavioral programming [11] bears many similarities with our proposal. Formally, it uses the *request-wait-block* paradigm. According to it, at each synchronization point, each thread declares three sets of events: the ones it requests (it needs one of them to go on), the ones it does not request, but wants to be informed when they happen, and the ones it blocks. An external scheduler chooses one event requested by some thread and blocked by none, and so the system goes on to the next synchronization point.

Our actions or rules are such events. The actions a component system is able to perform from a state can be seen as requested or waited-for, according to whether other individual actions are available. Blocking also happens in

synchronized rewrite systems: as rules with the same name must execute at the same time, when a state is not able to perform a given rule that exists in its system, it actually blocks that rule in the other systems. Although there is not a perfect fit between the formalizations, the resulting settings are very similar. The examples found in [10,11] are easily translatable to synchronized Maude modules. One of the favorite examples in papers on behavioral programming consists of implementing the rules for the game of tic-tac-toe and their *strategies* as independent synchronized threads.

The strong formal basis of rewriting logic and of the synchronous product could benefit behavioral programming research. Also, the already large literature on behavioral programming must contain inspiration for our future work.

Generalizations. Abandoning the "equal names" convention and relying on general relations to specify synchronization requirements is an obvious item in our to-do list. Also useful can be the introduction of some kind of identity or null transition, so that a rule can synchronize to it (that is, individually execute) or with some other actual rule. Relaxing the requirement of having only a finite number of `Prop`s is worth considering as well.

Quite interesting seems to allow for propositions on transitions in the way described in our paper [16]. The above setting only takes into account the label of the rule being executed. But one of the strengths of rewriting logic is that, not only states, but also transitions are represented by terms (on a extended signature, in the case of transitions). We should profit from this and ask transitions to synchronize on their common propositions. This would allow a rule to synchronize with another depending on their variable instantiations. The temporal logic TLR* [19] is designed to draw profit from transition terms (typically called *proof terms*) to specify more powerful temporal properties. The paper [1] describes a model checker for LTLR, the linear subset of TLR*.

References

1. Bae, K., Meseguer, J.: The linear temporal logic of rewriting Maude model checker. In: Ölveczky, P.C. (ed.) Rewriting Logic and its Applications. 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 208–225. Springer (2010), http://dx.doi.org/10.1007/978-3-642-16310-4_14
2. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on java. In: Baader, F. (ed.) Term Rewriting and Applications: 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007. Proceedings. pp. 36–47. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73449-9_5
3. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E.: ELAN from a rewriting logic point of view. Theoretical Computer Science 285(2), 155–185 (2002), http://dx.doi.org/10.1016/S0304-3975(01)00358-9
4. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)

5. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Fourth Annual Symposium on Logic in Computer Science, 1989. LICS '89, Proceedings. pp. 353–362 (Jun 1989)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007), http://dx.doi.org/10.1007/978-3-540-71999-1
7. Durán, F., Meseguer, J.: The Maude specification of Full Maude (Feb 1999), http://maude.cs.uiuc.edu/papers, manuscript, Computer Science Laboratory, SRI International
8. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002. Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187. Elsevier (2004), http://dx.doi.org/10.1016/S1571-0661(05)82534-4
9. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4533, pp. 153–168. Springer (2007), http://dx.doi.org/10.1007/978-3-540-73449-9_13
10. Harel, D., Lampert, R., Marron, A., Weiss, G.: Model-checking behavioral programs. In: Proceedings of the Ninth ACM International Conference on Embedded Software. pp. 279–288. EMSOFT '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2038642.2038686
11. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM 55(7), 90–100 (Jul 2012), http://doi.acm.org/10.1145/2209249.2209270
12. Havelund, K., Roşu, G.: Monitoring programs using rewriting. In: Feather, M.S., Goedicke, M. (eds.) Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, Coronado Island, San Diego, CA, USA, November 26-29, 2001. pp. 135–143. IEEE Computer Society (2001), http://dx.doi.org/10.1109/ASE.2001.989799
13. Kirchner, H.: Rewriting strategies and strategic rewrite programs. In: MartÃ-Oliet, N., Ã–lveczky, P.C., Talcott, C. (eds.) Logic, Rewriting, and Concurrency, Lecture Notes in Computer Science, vol. 9200, pp. 380–403. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-23165-5_18
14. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to modular model checking. ACM Trans. Program. Lang. Syst. 22(1), 87–128 (Jan 2000), http://doi.acm.org/10.1145/345099.345104
15. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for Maude strategies. In: Roşu, G. (ed.) Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008. Electronic Notes in Theoretical Computer Science, vol. 238(3), pp. 227–247. Elsevier (2009), http://dx.doi.org/10.1016/j.entcs.2009.05.022
16. Martín, O., Verdejo, A., Martí-Oliet, N.: Egalitarian state-transition systems (2016), http://maude.sip.ucm.es/syncprod, submitted
17. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992), http://dx.doi.org/10.1016/0304-3975(92)90182-F

18. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3-7, 1997, Selected Papers. Lecture Notes in Computer Science, vol. 1376, pp. 18–61. Springer (1997), http://dx.doi.org/10.1007/3-540-64299-4_26
19. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 5065, pp. 354–382. Springer (2008), http://dx.doi.org/10.1007/978-3-540-68679-8_22
20. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer Berlin / Heidelberg (1980), http://dx.doi.org/10.1007/3-540-10235-3
21. Plump, D.: The design of GP 2. In: Escobar, S. (ed.) Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011. EPTCS, vol. 82, pp. 1–16 (2011), http://dx.doi.org/10.4204/EPTCS.82.1
22. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
23. Roldán, M., Durán, F., Vallecillo, A.: Invariant-driven specifications in Maude. Science of Computer Programming 74(10), 812–835 (2009), http://dx.doi.org/10.1016/j.scico.2009.03.003
24. Visser, E.: A survey of rewriting strategies in program transformation systems. Electronic Notes in Theoretical Computer Science 57, 109 – 143 (2001), http://www.sciencedirect.com/science/article/pii/S1571066104002701, WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming