Synchronous products of rewrite systems*

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet

Technical Report 02/16

Departamento de Sistemas Informáticos y Computación
Facultad de Informática, Universidad Complutense de Madrid, Spain

Jun 2016

{omartins,jalberto,narciso}@ucm.es

* Partially supported by MINECO Spanish projects StrongSoft (TIN2012–39391–C04–04) and TRACES (TIN2015–67522–C3–3–R), Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731), and UCM-Santander grant GR3/14.

Abstract. We present a concept of module composition for rewrite systems that we call synchronous product, and also a corresponding concept for doubly labeled transition systems (as proposed by De Nicola and Vaandrager) used as semantics for the former. In both cases, synchronization happens on states and on transitions, providing in this way more flexibility and more natural specifications. We describe our implementation in Maude, a rewriting logic-based language and system. A series of examples shows their use for modular specification and hints at other possible uses, including modular verification.

Table of Contents

1 Introduction

In this paper we propose a composition of rewrite systems [21] by means of synchronous products with the aim of using it for modular specification of systems. We also define a synchronous product for doubly labeled transition systems (L^2TS) as defined in [7]. We use the latter to semantically ground the former.

Our concept of synchronous product is akin to the one from automata theory, whence it borrows its name, but also to the composition of processes in CCS [24], to request-wait-block threads in behavioral programming [13], and to other formalisms for module composition. Most of these formalisms rely on action identifiers for synchronization, that is, actions with the same name in both component systems execute simultaneously. Some, like [17], synchronize states: the ones simultaneously visited by the component systems must agree on the atomic propositions they satisfy: if $s_1 \models p$ and $s_2 \models \neg p$ for some proposition p, then $\langle s_1, s_2 \rangle$ is not even a state of the composed system.

As explained in several papers [19,23,15,8], state-only based or action-only based settings are often not enough for a natural specification of systems and temporal properties. In some cases, we are interested in the propositions of the states; in other cases, it is the action that took the system to that state that matters. In many cases, the combined use of propositions on states and on transitions results in more natural formulas. For instance, the formula

$$\Box \Diamond \texttt{enabled-a} \rightarrow \Box \Diamond \texttt{taking-a}$$

(from [23]) expresses fairness for action `a`: if action `a` is infinitely often enabled, then it is infinitely often taken. Here, `enabled-a` is a property of states (that they allow the execution of `a` on them), but `taking-a` is a property of the transition taking place. In the same spirit, this paper suggests that composition of modules is better approached by synchronizing both states and actions. The papers [19,23] show how it is always possible to *cook* a system so that all relevant information about transitions is included in states. Thus, strictly speaking, action synchronization is not needed, but is most convenient.

L^2TSs are a kind of amalgamation of LTSs (labeled transition systems) and Kripke structures: they label states with sets of propositions (as Krikpe structures do), and transitions with action identifiers (as LTSs do). They are state- and action-based, and are appropriate for our discussion.

The theoretical contribution of this paper is the definition of the synchronous product for both L^2TSs and rewrite systems. States synchronize based on their atomic propositions, and transitions based on their action identifiers or rule labels. We show how rewrite systems (and their synchronous product) can be given semantics on L^2TSs (and their own synchronous product).

As a more practical contribution, the aim of our definitions is to allow the modular specification of rewrite systems. This is shown in the examples. We foresee that this would make modular verification possible. Also, as a composed system only has the behaviors that are possible in both component systems, it can be used as a means to control a system with another one tailored for

that purpose. We see this as a possible implementation of strategies for rewrite systems—and one suited to modular verification. These two possibilities are work in progress and are just hinted at in the examples.

The rest of this paper is divided into six sections. Section 2 recalls L^{2}TSs and defines their synchronous product. Section 3 focuses on the synchronous product for rewrite systems and on their semantics. Section 4 shows some examples of modular specification. 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. Section 7 summarizes the conclusions of the paper.

This is an extended version of [20]. 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 L^{2}TSs

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

2.1 L^{2}TS: Doubly Labeled Transition Systems

Doubly labeled transition systems were introduced by De Nicola and Vaandrager in [7] with the aim of comparing properties of Kripke structures and of labeled transition systems (LTSs). Indeed, L^{2}TSs join in a single object the characteristics of these different structures. That is, their states are labeled by sets of atomic propositions (the ones that hold true in the state) and their transitions are labeled by action identifiers. The original definition from [7] includes invisible actions, but we will not need them.

Formally, an L^{2}TS is defined as a tuple $(S, \Lambda, \rightarrow, \mathrm{AP}, L)$, where S is a set of states, Λ is an alphabet of action identifiers, $\rightarrow \subseteq S \times \Lambda \times S$ is a transition relation (denoted as $s \xrightarrow{\lambda} s'$), AP is a set of atomic propositions, and $L : S \rightarrow 2^{\mathrm{AP}}$ is a labeling function, that assigns to each state the atomic propositions that hold true on it.

2.2 Synchronous Products

The synchronous product of two systems is a way to make them evolve in parallel, making sure that they agree at each step and in every moment. Given two L^{2}TSs $\mathcal{L}_i = (S_i, \Lambda_i, \rightarrow_i, \mathrm{AP}_i, L_i)$, we define next their synchronous product $\mathcal{L}_1 \parallel \mathcal{L}_2 = (S, \Lambda, \rightarrow, \mathrm{AP}, L)$. The synchronization is specified by relating properties and actions common to both structures, that is, existing with the same name in both. For a state $s_1 \in S_1$ to be visited by $\mathcal{L}_1$ at the same time as $s_2 \in S_2$ is visited by $\mathcal{L}_2$ it is necessary that, for each common atomic proposition $p \in \mathrm{AP}_1 \cap \mathrm{AP}_2$, we have that p holds for s_1 iff it holds for s_2; more

formally: $L_1(s_1) \cap \mathrm{AP}_2 = L_2(s_2) \cap \mathrm{AP}_1$. We denote this by $s_1 \approx s_2$ and say that s_1 and s_2 are *compatible* or that the pair $\langle s_1, s_2 \rangle$ is compatible. For a transition $s_1 \xrightarrow{\lambda_1}_1 s_1'$ to occur in $\mathcal{L}_1$ simultaneously with $s_2 \xrightarrow{\lambda_2}_2 s_2'$ in $\mathcal{L}_2$ it is necessary that $\lambda_1 = \lambda_2$ (in addition to $s_1 \approx s_2$ and $s_1' \approx s_2'$). However, actions existing only in one of the systems can execute by themselves. This is the definition of $\mathcal{L}_1 \parallel \mathcal{L}_2 = (S, \Lambda, \rightarrow, \mathrm{AP}, L)$:

- $S := S_1 \times S_2$;
- $\Lambda := \Lambda_1 \cup \Lambda_2$;
- regarding transitions (assuming $s_1 \approx s_2$):
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2' \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $s_2 \xrightarrow{\lambda} s_2'$ and $s_1' \approx s_2'$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2 \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $\lambda \notin \Lambda_2$ and $s_1' \approx s_2$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1, s_2' \rangle$ iff $s_2 \xrightarrow{\lambda} s_2'$ and $\lambda \notin \Lambda_1$ and $s_1 \approx s_2'$;
- $\mathrm{AP} := \mathrm{AP}_1 \cup \mathrm{AP}_2$;
- $L(\langle s_1, s_2 \rangle) := L_1(s_1) \cup L_2(s_2)$.

Some notes on the definition and its consequences are in order:

- We let the space state S include non-compatible pairs. However, only transitions going into compatible states are allowed, so that all states reachable from a compatible initial state are compatible.
- The resulting composed system includes all the propositions and action identifiers from both component systems (we take their unions), but for synchronization only the ones that are common are taken into account (their intersections).
- Renaming of propositions and actions in a structure can be done with no harm to get equal names in both structures as needed for synchronization.
- When the two systems being composed have no common propositions and no common actions ($\mathrm{AP}_1 \cap \mathrm{AP}_2 = \emptyset$ and $\Lambda_1 \cap \Lambda_2 = \emptyset$), they progress with no consideration to each other: any state can pair with any other, and each action is executed by itself.
- A system controls the actions the other one can perform. Consider the situation where the composed system is in state $\langle s_1, s_2 \rangle$ (with $s_1 \approx s_2$) and $\mathcal{L}_1$ can execute action λ from s_1 ($s_1 \xrightarrow{\lambda} s_1'$). There are three possibilities to consider in $\mathcal{L}_2$:
  - if $\lambda \notin \Lambda_2$, the action can be run in $\mathcal{L}_1$ by itself: $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2 \rangle$ (provided $s_1' \approx s_2$);
  - if $\lambda \in \Lambda_2$, but it cannot be executed from s_2, then λ cannot be executed in the composed system at the moment;
  - if $\lambda \in \Lambda_2$ and can be executed from s_2 in $\mathcal{L}_2$ ($s_2 \xrightarrow{\lambda} s_2'$), then λ can only be executed simultaneously in both systems: $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2' \rangle$ (provided $s_1' \approx s_2'$).

3 Synchronous Products of Rewrite Systems

Our aim is to implement and practically use synchronous products for modular specification. Thus, we now reflect the abstract definitions above in the more concrete realm of rewrite systems.

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 by Meseguer in [21]. Maude [5] is a language (and system) 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 equational logic—membership equational logic indeed [22], but we are not using such a feature in this paper. Thus, a rewrite system has the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$, 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; and R is a set of labeled rewrite rules of the form $[\ell]\, s \to s'$.

In Section 3.2 below, we show a way to compose and synchronize rewrite systems. Synchronization on states happens on coincidence on their common propositions. For that to be meaningful, we need a way to handle propositions, which are not, in principle, an ingredient of rewrite systems. Thus, we require of each rewrite system $\mathcal{R} = (\Sigma, E \cup Ax, R)$ the following:

- the sort in Σ that represents the states of the system is called `State`;
- Σ includes a sort `Prop`, representing atomic propositions, composed by a finite amount of constants (this requirement is needed in Section 3.2 to define the synchronous product);
- $\mathcal{R}$ includes the definition of a theory of the Booleans declaring, in particular, the sort `Bool`, and the constants `true` and `false`;
- Σ includes an infix relation symbol $\_ \models \_ : \texttt{State} \times \texttt{Prop} \to \texttt{Bool}$, and E includes equations that completely define $\models$, that is, each expression $s \models p$ is reduced to `true` or `false` by $E \cup Ax$.

These conventions are a standard way to introduce propositions in rewrite systems. It is the setting needed to use Maude's LTL model checker [10], for instance. However, we are using propositions only for synchronization. Even if model checking were performed on any of these systems, the propositions used for that need not be the same ones used for synchronization.

We have one additional technical requirement: the system has to be *topmost*. 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` is not an argument

of 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 rules preserve their meaning after composition. For instance, the non-topmost 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 [11]. (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 sense of having the same state terms and the same transitions between them.)

If we intend not only to specify systems but also to run them, we need to require that our rewrite systems be *computable*. Computability of a rewrite system is formally defined, for instance, in [23]. For arbitrary sets of equations and rules, the one-step rewriting relation between the terms of a system (that is, whether a term s can be rewritten to s' by applying some rule of the system) is undecidable. But for computable systems it is effectively decidable. The conditions are easy to meet. Usually, the rewrite systems a sensible programmer would code are computable.

3.2 Synchronous Products

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

A technical detail is needed about names and namespaces. The conditions in Section 3.1 require that each system includes some sorts and operators: `State`, $\models$, and so on. This does not mean that sorts with the same name in different systems are the same sort. Indeed, we consider that each system has implicit its own namespace. Names for sorts, constants, and the other elements must be understood within the namespace of their respective systems. When needed, we qualify a name with a prefix showing the system it belongs to or where it originated: $\mathcal{R}$.`State`. We omit the prefix whenever there is no danger of confusion. Sometimes we say that something is true "in $\mathcal{R}$" to avoid cluttering the text with prefixes for each element that would need it.

We refer as *naked names* to the ones without the qualifying prefixes. These are needed for synchronization, as it is done on coincidence of naked names, and those names remain as such in the product system, with different qualification. For instance, the value of $\mathcal{R}_1.p$ has to be the same as the one of $\mathcal{R}_2.p$ and both give rise to $\mathcal{R}.p$.

With this convention about namespaces, 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 synchronous product,

according to the definition below; any equational deduction valid in one of the systems is still valid in the product. Rules, instead, are not included verbatim from the component systems, but synchronized as formalized below.

As also mentioned in Section 2.2, we assume that renaming has previously taken place as needed, so that synchronization happens on the set of rule labels and the set of atomic propositions whose naked names are common to both systems.

This is the rather long definition of the synchronous product:

- $\Sigma := \Sigma_1 \uplus \Sigma_2 \uplus \Sigma'$, where Σ' contains:
  - declarations for sorts $\mathcal{R}.\texttt{State}$ and $\mathcal{R}.\texttt{Prop}$;
  - declarations for $\mathcal{R}.\texttt{Bool}$, $\mathcal{R}.\texttt{true}$, and $\mathcal{R}.\texttt{false}$;
  - a declaration for the operator $\mathcal{R}.\models : \mathcal{R}.\texttt{State} \times \mathcal{R}.\texttt{Prop} \to \mathcal{R}.\texttt{Bool}$;
  - a new constructor symbol $\langle \_, \_ \rangle : \mathcal{R}_1.\texttt{State} \times \mathcal{R}_2.\texttt{State} \to \mathcal{R}.\texttt{State}$;
  - a set of declarations for operators to make $\mathcal{R}.\texttt{Prop}$ the union of $\mathcal{R}_1.\texttt{Prop}$ and $\mathcal{R}_2.\texttt{Prop}$, that is:

    $$\{\mathcal{R}.p : \mathcal{R}.\texttt{Prop} \mid \mathcal{R}_1.p : \mathcal{R}_1.\texttt{Prop} \in \Sigma_1 \text{ or } \mathcal{R}_2.p : \mathcal{R}_2.\texttt{Prop} \in \Sigma_2 \text{ or both}\};$$

  - a declaration for the predicate: $\mathcal{R}.\approx : \mathcal{R}_1.\texttt{State} \times \mathcal{R}_2.\texttt{State} \to \mathcal{R}.\texttt{Bool}$.
- $E := E_1 \uplus E_2 \uplus E'$, where E' contains:
  - equations for a theory of the Booleans;
  - equations to reduce $s_1 \approx s_2$ to $\texttt{true}$ in $\mathcal{R}$ iff ($s_1 \models p = \texttt{true}$ in $\mathcal{R}_1 \iff s_2 \models p = \texttt{true}$ in $\mathcal{R}_2$, for every proposition whose naked name p exists in both systems), and to $\mathcal{R}.\texttt{false}$ otherwise;
  - for each p such that $\mathcal{R}_1.p : \mathcal{R}_1.\texttt{Prop} \in \Sigma_1$, the equation:

    $$\langle x_1, x_2 \rangle \ \mathcal{R}.\models \mathcal{R}.p \quad = \quad x_1 \ \mathcal{R}_1.\models \mathcal{R}_1.p,$$

  - for each p not in the previous item but such that $\mathcal{R}_2.p : \mathcal{R}_2.\texttt{Prop} \in \Sigma_2$, the equation:

    $$\langle x_1, x_2 \rangle \ \mathcal{R}.\models \mathcal{R}.p \quad = \quad x_2 \ \mathcal{R}_2.\models \mathcal{R}_2.p.$$

  In these equations x_1 and x_2 are variables of sorts $\mathcal{R}_1.\texttt{State}$ and $\mathcal{R}_2.\texttt{State}$, respectively. Because of the conditions on the rules below, only compatible pairs $\langle x_1, x_2 \rangle$ are reachable. And only for such pairs we will need to use some of the last two equations above. Thus, for a proposition p whose naked name exists in both systems, we have arbitrarily but harmlessly chosen to use the value from the first system.
- $Ax := Ax_1 \uplus Ax_2$.
- R is composed of the following set of rules:
  - for each rule label ℓ that exists in both systems, say $[\ell]\, s_i \to s_i' \in R_i$, we have in R the conditional rule $[\ell]\, \langle s_1, s_2 \rangle \to \langle s_1', s_2' \rangle$ if $s_1' \approx s_2'$;
  - for each rule label ℓ that exists in R_1 but not in R_2, say $[\ell]\, s_1 \to s_1' \in R_1$, we have in R the conditional rule $[\ell]\, \langle s_1, x_2 \rangle \to \langle s_1', x_2 \rangle$ if $s_1' \approx x_2$ (with x_2 a variable of sort $\mathcal{R}_2.\texttt{State}$);
  - correspondingly for rule labels in R_2 but not in R_1.

In these three kinds of rules, the condition guarantees that only compatible states are reached.

Several items above include universal quantification on atomic propositions. This could be problematic, and it is the reason why we require the sorts `Prop` to consist only of a finite amount of constants.

3.3 Semantics

Given a rewrite system as above, $\mathcal{R} = (\Sigma, E \cup Ax, R)$, its semantics is an $\mathrm{L}^2\mathrm{TS}$ $\mathcal{L} = (S, \Lambda, \rightarrow, \mathrm{AP}, L)$, based on the usual term-algebra semantics (see [21], for instance) in this way:

- $S := T_{\Sigma/E \cup Ax, \texttt{State}}$, the set of terms of sort `State` modulo equations;
- Λ is the set of rule labels in R;
- $\rightarrow$ corresponds to the transition relation generated by rewriting with the rules from R [21], that is, $s \xrightarrow{\lambda} s'$ iff there is a rule in R with label λ that allows rewriting s to s' in one step within $\mathcal{R}$;
- $\mathrm{AP} := T_{\Sigma/E \cup Ax, \texttt{Prop}}$, the set of terms of sort `Prop` modulo equations;
- $L(s) := \{p \in \mathrm{AP} \mid s \models p = \texttt{true} \text{ modulo } E \cup Ax\}$.

Let "sem" denote the *semantics operator*, which assigns to each rewrite system an $\mathrm{L}^2\mathrm{TS}$ as just explained. All previous definitions have been chosen so that the following result holds.

Theorem. *For any rewrite systems $\mathcal{R}_1$ and $\mathcal{R}_2$, we have that* $\mathrm{sem}(\mathcal{R}_1 \,\|\, \mathcal{R}_2)$ *is isomorphic to* $\mathrm{sem}(\mathcal{R}_1) \,\|\, \mathrm{sem}(\mathcal{R}_2)$. *The isomorphism is in the sense that there exist bijections between their sets of states, between their sets of actions, and between their sets of atomic propositions that preserve the transition relation and the labeling.*

The proof is in Appendix **??**.

4 Examples

We present examples of synchronous products of rewrite systems. Many of them show systems made up to control others. They are coded in Maude [5], the rewriting based language and system we have used to develop our implementation of the synchronous product. They should be easily understood by anyone acquainted with rewriting logic or algebraic programming. All the examples are downloadable from our website: `http://maude.sip.ucm.es/syncprod`. Many of the examples build on previous ones. The first one involves no synchronization, but it uses modular specification, and serves as basis for subsequent ones.

4.1 Modular Specification: Two Railways

Consider this sketchy implementation of a railway in Maude:

```
mod RAILWAY1 is
   including BOOL .
   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
```

Modules `BOOL` and `SATISFACTION` are conveniently predefined in Maude. We can picture the system 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 `RAILWAY2.State`. No synchronization is possible, because all rule labels are different and there are no propositions, but the modular specification is simpler and more natural than a monolithic one would be.

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 Synchronizing Actions: Safety Control

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 BOOL .
   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 this 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. The advantage of modularity, in this case, is that it allows an external, non-intrusive control.

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

4.3 Synchronizing States: Alternative Safety Control

In more complex implementations of the RAIL system, controlling all the ways in which trains can get into the crossing can be involved. For instance, both trains could be allowed to move into the crossing at the same time, so that controlling individual isolated movements as above would not be enough. In such cases, it can be easier to base the control on the states.

We extend the 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:

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

The new controller system we propose, SAFETY2, has a single state, named o, that satisfies the proposition safe, and no rules:

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

Consider RAIL-EXT ‖ SAFETY2. A typical rule in this composed system is

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

It is not too different from the previous t1wc, except for the compatibility condition. As o is always safe, also < crossing, T2 > must be safe for the rule to be applied. So, SAFETY2 restricts RAIL-EXT to visit only safe states, as desired.

Note again the advantage of a modular specification: once RAIL-EXT is given, we can easily choose the control that fits better, either SAFETY or SAFETY2 or some other given module with the same purpose.

4.4 Repeated Composition: Controlling Performance

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 RAIL ‖ SAFETY with this new system:

```
mod PERFORMANCE is
   including BOOL .
   including SATISFACTION .
   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.

Note that the product `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 Instrumentation: Counting Crossings

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, etc. To some extent, it can be done by using synchronous products.

This time we want to keep track of the number of crossings for each train. For `RAILWAY1` we propose this very simple system:

```
mod COUNT1 is
   including BOOL .
   including SATISFACTION .
   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.

4.6 Separation of Concerns: Dekker's Algorithm

Consider this new module:

```
mod DEKKER is
   including BOOL .
   including SATISFACTION .
   sorts Waiting Turn .
   ops 0w 1w 2w : -> Waiting .
   ops t1 t2 : -> Turn .
   op (_,_) : Waiting Turn -> State .
   var T : Turn .
   rl [t1wc] : (1w,T)  => (0w,t2) .   rl [t2wc] : (1w,T)  => (0w,t1) .
   rl [t1wc] : (2w,t1) => (1w,t2) .   rl [t2wc] : (2w,t2) => (1w,t1) .
   rl [t1fw] : (0w,T)  => (1w,T) .    rl [t2fw] : (0w,T)  => (1w,T) .
```

```
   rl [t1fw] : (1w,T)  => (2w,T) .    rl [t2fw] : (1w,T)  => (2w,T) .
endm
```

This module can be used to ensure absence of starvation in the controlled system, that is, that no process waits indefinitely. The `Waiting` component of the state stores how many processes are waiting to enter the critical section: both, one, or none. The `Turn` component stores whose turn is next, in case both processes are waiting (if only one process is waiting, it can just go on).

Usual presentations of Dekker's algorithm also include mutual exclusion control. Our module `DEKKER` does not control when processes exit the critical section so it cannot ensure mutual exclusion by itself. In our case, the combined control is achieved by the product `SAFETY` $\|$ `DEKKER`. Separation of different concerns in different modules is made possible by the synchronous product construction.

4.7 State and Rule Synchronization: Two Trains in a Linear Railway

As an example that sometimes synchronization is convenient on states and on transitions in the same system, consider this one, taken from [6], told again in terms of train traffic. There is a single linear railway divided into tracks, and there are two trains going along it from track to track, always in the same direction—to the right, say. Each train can move at any time from one track to the next, but they can never be at the same time on the same track. Thus, whenever the trains are in adjacent tracks, only the rightmost one can move. This is the specification for the train on the left:

```
mod LTRAIN is
   including BOOL .
   including SATISFACTION .
   including NAT .
   subsort Nat < State .
   var Track : Nat .
   rl [lmove] : Track => Track + 1 .
endm
```

The one on the right is specified in module `RTRAIN` which is the same as above except that the rule is called `rmove`. The controller we need has to detect when the trains are in adjacent tracks, and this is a property on the states of `LTRAIN` $\|$ `RTRAIN`. To make the control possible, we extend this composed system with the declaration of the proposition `adjacent`:

```
mod TRAINS-EXT is
   including LTRAIN || RTRAIN .
   op adjacent : -> Prop .
   vars T T' : Nat .
   eq < T, T + 1 > |= adjacent = true .
   eq < T, T' > |= adjacent = false [owise] .
endm
```

The controller is this:

```
mod CONTROL is
   including BOOL .
   including SATISFACTION .
```

```
    ops adj nonadj : -> State .
    var S : State .
    rl [lmove] : nonadj => nonadj .
    rl [lmove] : nonadj => adj .
    rl [rmove] : S => nonadj .
    op adjacent : -> Prop .
    eq adj |= adjacent = true .
    eq nonadj |= adjacent = false .
endm
```

Only the movement of the train on the left can take the system to a configuration with adjacency. When it does, the controller remembers it in its state, and the next movement can only be made by the train on the right. Note that both kinds of synchronization, on states and on transitions, are present in this example, and that using only one of them would result in a more involved specification.

5 Notes on the Implementation

Our prototype implementation of the synchronous product in Maude can be downloaded from our website: http://maude.sip.ucm.es/syncprod. Appendix **??** contains brief instructions for using it. The implementation largely follows the explanations in Section 3.2. Some details, however, could be appreciated by those familiar with Maude or rewriting logic.

Choice of Tools. 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 Maude itself is a very convenient tool for this second-order programming task.

Rewriting logic is reflective, and that implies in particular that constructs of the language can be represented and handled as terms [5]. Maude provides a set 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 also provides functions to perform such meta-representation. We have chosen this as the natural way to the implementation. We have coded a Maude function `syncprod` that receives two terms of sort `Module` and produces one representing their synchronous product.

But that function can only be invoked at the metalevel, feeding it with two terms of sort `Module`, not with two Maude modules. A decent implementation must allow a simpler use. For those acquainted with Maude, the tool of choice for such a task is Full Maude. Full Maude [9,5] 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 module 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.

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 its namespace. 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 `ModName.State`, 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, their union 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. (To be completely true, the dots cause some problems with Maude's syntax, so we have omitted them: `ModNameState` and `ModName|=`.)

6 Related and Future Work

Some of the proposals of this paper set the ground on which interesting work is already being done. Let's be more concrete.

Egalitarian Synchronization. In [19] we presented a class of transition systems called *egalitarian structures*. They are egalitarian in the sense that they treat states and transition as equals. In particular, they allow using atomic propositions on transitions. That paper also showed how rewrite systems are egalitarian in nature, because transitions are represented by proof terms in the same way as states are represented by terms of the appropriate sort.

As pointed in the introduction and also in [19], the expression of temporal properties by formulas benefits from an egalitarian view. Composition of systems should benefit in the same way. An egalitarian synchronous product would allow transitions to synchronize not just on labels, but on their common propositions (depending, in particular, on variable instantiations).

Strategies. The examples have shown how it is possible to control a system with another one made up for that purpose. It is fair to call *strategic* this kind of control. Indeed, we see the synchronous product as a means to implement strategies for rewrite systems. As also shown in [19], strategies can also benefit from an egalitarian treatment. We expect to be able to develop automatic translations from some strategy languages to equivalent Maude modules, although the precise power of such a technique is still to be seen.

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 [16]). Maude [5] includes flexible strategies for the evaluation of terms (like lazy, innermost and so on), and external implementations have been proposed in [18] and in [27]. ELAN [2], Tom [1], and Stratego [28] include strategies built-in. They also appear in graph rewriting systems (see references in [16] and also [25], where they are called just *programs*). The same concept is used in theorem provers: it allows the user to guide the system towards the theorem, or to represent the whole proof once found.

Modularity for Specification and Verification. 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, another deals just with starvation.

Model checking [3] performed in a modular way can be more efficient, given that the size of the state space of the composed system is of the order of the product of the individual sizes. An attractive possibility is that of providing the specifier with a library of pre-manufactured and pre-verified 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. Care is needed, however, as it is not always the case that a composed system preserves the properties of the components.

Much work already exists on modular model checking and verification, but not many tools allow for it and, to the best of our knowledge, no implementation on rewriting logic has been developed. The papers [17,4], among many others, show techniques for drawing conclusions compositionally. Adapting such techniques to our framework is pending work.

Composition of modules can generate new deadlocks in cases where the components do not agree on a common next step. The system `SAFETY2` from Section 4.3 is a very simple example: as it constrains the base system to visit only `safe` states, absence of new deadlocks is only guaranteed assuming that in the base system, `RAIL-EXT`, a `safe` state is always reachable in one step. This is the same assume-guarantee paradigm proposed in [17] for modular model checking.

We are particularly interested in model checking strategically controlled systems. Once the concept of control through synchronous products is in place, existing tools can be used, ideally in a modular way (particularly, for us, Maude's LTL model checker [10]). The nearest works on this we are aware of are GP 2,

that includes Hoare-style verification in the context of graph rewriting [26], and the BPmc prototype tool for model checking behavioral programs in Java [12].

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 [14], for instance), and our examples on railway safety can 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.

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 [13] 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 an event requested by some thread and blocked by none, and so the system goes on to the next synchronization point. Although there is not a perfect fit between their formalization and ours, the resulting settings are very similar, and the examples in [12,13] are easily translatable to synchronized Maude modules.

7 Conclusions

The concept of synchronous product can be extended from automata theory to the specification of systems, where it represents composition of modules. It can be equivalently defined on abstract transition systems (namely, L^2TSs) and on rewrite systems. For more flexible and natural specifications, it is possible and convenient to synchronize at the same time on states and on transitions. We have used atomic propositions to synchronize states, but just rule labels (or action names) for transitions. We intend to generalize this in the near future.

The examples (to be run in our implementation in Maude) show how the synchronous product makes modular specifications easier in rewriting logic. We expect that it will also make possible the implementation of some kind of strategies and the modular verification of systems, even after they have been controlled by strategies. All this is work in progress.

References

1. 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

2. 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
3. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
4. 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)
5. 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
6. De Nicola, R., Fantechi, A., Gnesi, S., Ristori, G.: An action-based framework for veryfying logical and behavioural properties of concurrent systems. Computer Networks and ISDN Systems 25(7), 761 – 778 (1993), http://www.sciencedirect.com/science/article/pii/0169755293900478
7. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (Mar 1995), http://doi.acm.org/10.1145/201019.201032
8. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes. Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer (1990)
9. Durán, F., Meseguer, J.: The Maude specification of Full Maude (Feb 1999), http://maude.cs.uiuc.edu/papers, manuscript, Computer Science Laboratory, SRI International
10. 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
11. 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
12. 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
13. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM 55(7), 90–100 (Jul 2012), http://doi.acm.org/10.1145/2209249.2209270
14. 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
15. Kindler, E., Vesper, T.: ESTL: A temporal logic for events and states. In: Desel, J., Silva, M. (eds.) Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98 Lisbon, Portugal, June 22–26, 1998 Proceedings. pp. 365–384. Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg (1998), http://dx.doi.org/10.1007/3-540-69108-1_20

16. Kirchner, H.: Rewriting strategies and strategic rewrite programs. In: Martí-Oliet, N., Ölveczky, 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
17. 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
18. 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
19. Martín, Ó., Verdejo, A., Martí-Oliet, N.: Egalitarian State-Transition Systems. In: Lucanu, D. (ed.) Proceedings of the Eleventh International Workshop on Rewriting Logic and its Applications, WRLA 2016, Eindhoven, The Netherlands, April 2-3, 2016. Lecture Notes in Computer Science, vol. 9942, pp. 98–117. Springer (2016), http://dx.doi.org/10.1007/978-3-319-44802-2_6
20. Martín, Ó., Verdejo, A., Martí-Oliet, N.: Synchronous Products of Rewrite Systems. In: Artho, C., Legay, A., Peled, D. (ed.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 141–156. Springer (2016), http://dx.doi.org/10.1007/978-3-319-46520-3_10
21. 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
22. 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
23. 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
24. 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
25. 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
26. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
27. 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
28. 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,