Modular specification in rewriting logic (extended version)*

Technical Report 04/17

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

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

Oct 2017

Warning

The material in this technical report has been updated and superseded by a new paper: *Compositional Specification in Rewriting Logic*, by Ó. Martín, A. Verdejo, and N. Martí-Oliet, Theory and Practice of Logic Programming 20.1 (2020), pp. 44–98 (doi:10.1017/S1471068419000425). The examples in Sections 4 and 5, however, may still be useful.

*Partially supported by MINECO Spanish project TRACES (TIN2015–67522–C3–3–R), and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

Abstract

Our aim is to bring modularity to system specification in rewriting logic. Modularity here is in the sense of decomposing a system into its functional components, coding the specification for each component as a separate system, and then assembling them back. Rewriting logic is well suited for the specification of concurrent and non-deterministic systems but, up to now, modularity could hardly be addressed within it. The base of our proposal is the operation that we call *synchronous composition.* We discuss the reasons and implications of our proposal, and formalize it for rewriting logic and also for transition structures, to be used as semantics. To show the power of our approach, we include a few small but realistic examples and a larger example on cached computer architecture.

Contents

1 Introduction **1**
- 1.1 First motivational example: mutual exclusion . . . 1
- 1.2 Second motivational example: computer architecture . . . 3
- 1.3 Our contributions in this paper . . . 4

2 Goals, ideas, and choices **5**
- 2.1 Being egalitarian . . . 5
- 2.2 What is a transition? . . . 6
- 2.3 Syncing on properties . . . 8
- 2.4 Syncing on relations other than equality . . . 9
- 2.5 Syncing states with transitions . . . 10
- 2.6 The split . . . 11
- 2.7 Topmost everyone . . . 13
- 2.8 True concurrency . . . 14
- 2.9 The whole is the sum of its parts . . . 14
- 2.10 Explicit syncing criteria . . . 15
- 2.11 Properties of the composed system . . . 16
- 2.12 Summing up . . . 16

3 Formal definitions **16**
- 3.1 A bird's-eye view of it all . . . 16
- 3.2 Transition structures . . . 18
  - 3.2.1 atEgTrStr . . . 18
  - 3.2.2 EgTrStr . . . 19
  - 3.2.3 TrStr and the split . . . 20
  - 3.2.4 Runs . . . 21
  - 3.2.5 Commutativity theorems . . . 21
- 3.3 Rewrite systems . . . 22
  - 3.3.1 atEgRwSys . . . 22
  - 3.3.2 Executability of rules in atEgRwSys . . . 25
  - 3.3.3 EgRwSys . . . 28
  - 3.3.4 RwSys . . . 29
  - 3.3.5 The split from atEgRwSys to RwSys . . . 30
  - 3.3.6 The split from EgRwSys to RwSys . . . 31
  - 3.3.7 Deductions in EgRwSys and in RwSys . . . 32
  - 3.3.8 Commutativity theorems . . . 35
- 3.4 Semantics . . . 35
  - 3.4.1 Definitions . . . 35
  - 3.4.2 Commutativity theorems . . . 36

4 Some simple but complete examples **38**
- 4.1 Two trains on a linear railway . . . 38
- 4.2 Insertion sort . . . 42
- 4.3 Dekker's algorithm . . . 44

5 A case study: cache architecture **47**

5.1 Introduction 47
5.2 The program storage 48
5.3 The core 50
5.4 Methodology note: value passing 52
5.5 The cache 53
5.6 Methodology note: logical implication 55
5.7 Methodology note: patterns as properties 55
5.8 Methodology note: until 56
5.9 The policy 57
5.10 Methodology note: synchrony as in Greek 57
5.11 Methodology note: after 58
5.12 Chips 59
5.13 The main memory 61
5.14 The computer 62
5.15 The cache-coherence protocol 64

6 Related work **66**

6.1 Process algebras 66
6.2 Automata and labelled transition structures 67
6.3 Our own previous work 68
6.4 Petri nets 68
6.5 Maude modules and asynchronous messages 69
6.6 Aspect-oriented programming 71
6.7 Behavioural programming 72

7 Future work **73**

7.1 Implementation 73
7.2 Strategies 73
7.3 Modular verification 74

8 Conclusion **74**

A All the proofs **75**

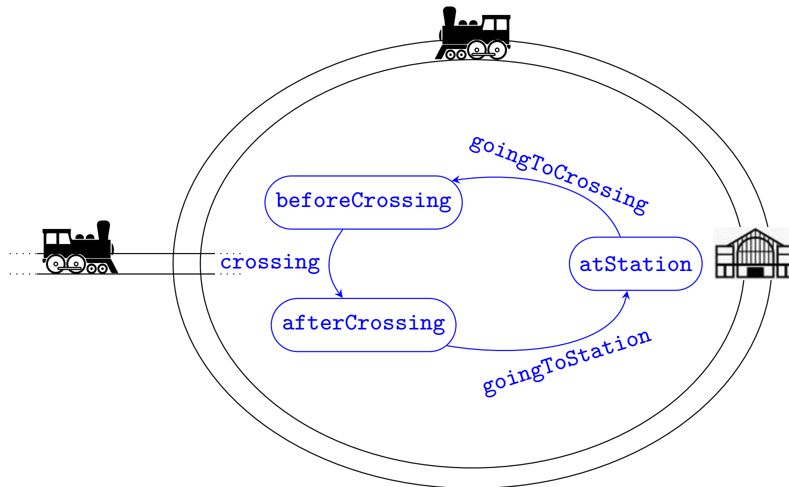
1 Introduction

To anyone in the fields of computer science and engineering, the convenience of modularity needs not be stressed. However, in the particular field of rewriting logic (started by Meseguer's paper [?]) modularity has a very restricted role in system design and specification. It is our intent in this work to discuss a means by which system modelling and specification can be accomplished in rewriting logic in a truly modular, flexible and powerful way. Our future-work list includes using this for modular verification and for the implementation of strategies. Ideally, our discussion would also be useful to other system-specification formalisms.

To motivate the need for our work, next, we discuss informally a couple of examples written in Maude-like syntax. Maude is a language based on rewriting logic, with a bunch of analysis tools around it. They are fully described in [?]. This is the language we use throughout this paper for our concrete examples, although we try to avoid syntactical nuisances and allow ourselves some liberality in notation when convenient.

1.1 First motivational example: mutual exclusion

Think of a train, a very simple model of a train, that goes round a closed railway in which there is a station and a crossing with another railway. There are three points of interest in the railway, that we use as the states for our model. There are three transitions for moving between the three states.



In Maude-like notation:

```
rl [goingToCrossing] : atStation => beforeCrossing .
rl [crossing] : beforeCrossing => afterCrossing .
rl [goingToStation] : afterCrossing => atStation .
```

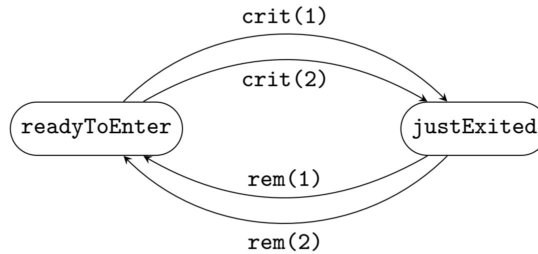
The keyword `rl` introduces a rewrite rule. The identifier in square brackets is the label of the rule. Rules describe transitions between states. To the left of the arrow (`=>`) is the origin state; to the right is the destination state.

Think, indeed, of two trains, both modelled the same, that share the piece of railway we have identified as `crossing`. Let's call these systems `TRAIN1` and

TRAIN2. This is a typical case where mutual exclusion is needed in the access to the crossing.

Modularity is desirable here. The two trains are independent systems, that are better modelled separately and combined afterwards. Also, the control to ensure mutual exclusion must be exerted on the trains from outside. From a design point of view, the model of a train must specify how a train works, what actions it is able to perform, but not any control external to the machine. From a more practical point of view, having different concerns (the workings of the trains and the control) coded into different modules eases the tasks of specification and maintenance.

This is the module we propose to control for mutual exclusion:



In Maude-like notation:

```
rl [crit(I)] : readyToEnter => justExited .
rl [rem(I)] : justExited => readyToEnter.
```

Let's call MUTEX this system. We need now to compose the three systems: TRAIN1 || TRAIN2 || MUTEX. For this composed system to work properly, we need to make sure that transition crossing in system TRAIN1 (that we write as TRAIN1.crossing) takes place in sync with (that is, simultaneously to) transition MUTEX.crit(1); and TRAIN2.crossing with MUTEX.crit(2). Certainly, transitions crit(1) and crit(2) cannot happen at the same time and, thus, mutual exclusion is ensured. This module MUTEX can be used to ensure mutual exclusion on any two systems, with appropriate syncing criteria.

This opens the door not just to modular specification, but to modular verification: given that we can assert that MUTEX satisfies mutual exclusion, that is, the LTL formula

$$\square(\neg\texttt{crit(1)} \lor \neg\texttt{crit(2)}),$$

then we can assert that TRAIN1 || TRAIN2 || MUTEX satisfies mutual exclusion in the form of the LTL formula

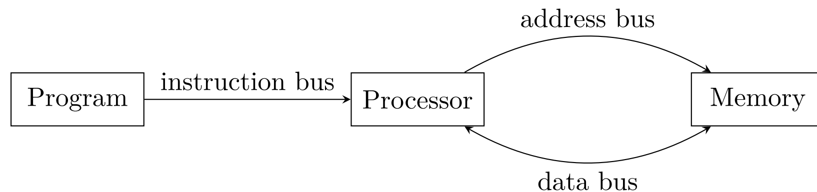
$$\square(\neg\texttt{TRAIN1.crossing} \lor \neg\texttt{TRAIN2.crossing}).$$

But there are issues to solve. We have used above terms with variables as rule labels (like crit(I)), and this is not a standard feature of rewriting logic. Also, the critical section of a system can consist of several consecutive transitions, and all of them need to be synced with the same crit(I) transition in MUTEX. All this is explored and solved below.

(A passing linguistic note. We find that the words *synchronization*, *synchronize*, *synchronously* and the rest in its syntactic family are hard. One gets tired after reading them a few times. We have to use them very often. That's why we have preferred to use when possible the much more friendly *sync*, *syncing*, *synced*, and so on.)

1.2 Second motivational example: computer architecture

We introduce now an example slightly more complex than the former one. It models a schematic computer architecture in which a processor works with an external memory for data storage and a separate memory to store the program to be executed. We can picture the complete system like this:



The processor component also includes a register for the instruction currently being executed, and another register for the last piece of data read from memory or to be written to memory.

In object-oriented and similar methodologies, the three entities—program, processor, and memory—would be coded as independent modules (classes), with internal details hidden, and with the needed operations in the interfaces to allow for the exchange of requests and data. This kind of modularity with encapsulation is hard, if ever possible, in rewriting logic.

A rewriting logic specification could go like this:

- the stored program is represented as a set of instructions, each one similar to `(1, w, 3, 7)` meaning: "instruction number 1 asks to write at memory address 3 the data 7";
- the memory is represented as a set of pairs (address, data);
- the processor stores the counter for the next instruction to be executed, plus an instruction like `(w, 3, 7)` in its instruction register, plus a piece of data in its data register.

The steps in the evolution of such a system would be represented by rewriting rules like this one, that executes a writing instruction already stored in the processor's register:

⟨Part of the `COMPUTER` specification⟩:

```
rl [execW] : Program: SomeSetOfInstructions
             Processor:
                ProgCounter: N
                Instr: (w, A, D)
                Data: D'
             Memory: (A, D'') RestOfMemory
          => Program: SomeSetOfInstructions
             Processor:
                ProgCounter: N + 1
                Instr: void
                Data: D
             Memory: (A, D) RestOfMemory .
```

Maude allows using flexible syntax; all elements in that rule that are not variables are added syntax. The rule, put in words, is saying: "If the instruction just read by the processor is a writing requirement for address `A` and data `D` (that is, `(w, A, D)`), then overwrite whichever data is currently stored in the

memory at address `A` with the new data `D`, update also the data register with `D`, and add one to the program counter." Other rules would take care of other kinds of instructions.

This is simple and useful for some goals. But it is not modular. The problem is that the rule above involves two components—the processor and the memory—and there is no way, in the setting of traditional rewriting logic, to model their behaviour independently. This is what we are after: being able to specify three separate systems for the three components of the computer, and to make them evolve in sync. In the processor, there would be a rule like this:

⟨In module `PROCESSOR`⟩:
```
rl [doingW] : ProgCounter: N
              Instr: (w, A, D)
              Data: D'
           => ProgCounter: N + 1
              Instr: void
              Data: D .
```

Then, in the memory, there would be:

⟨In module `MEMORY`⟩:
```
rl [updating] : (A, D'') RestOfMemory
             => (A, D) RestOfMemory .
```

These two rules are decoupled, and only work as needed if they are coupled back to run simultaneously; that is, if they are synced. On the processor side, the rule `doingW` does not produce any real updating—it needs cooperation from the memory. On the memory side, the rule `updating` allows to update any address to any data—but this is only useful when the processor restricts the range of values to just address `A` and data `D`.

The two rules must execute at the same time, and must exchange data to agree on the value of `D`. The means we propose below for these two needs—simultaneity and data exchange—are the same.

1.3 Our contributions in this paper

Using decoupled rules as proposed above, we specify all the capabilities of each component. The memory is capable of updating with any values of `A`, `D`, and `D''`. The processor is capable of taking its step and, if run isolated, would gladly do it pretending some writing has indeed been performed. We propose a syncing mechanism to restrict the wild capabilities of the components. It represents the wiring in real-world computers.

Such refined realism is not always needed. When it is used, each component system can be simpler, and may be given independent meaning. Our goal, again, is to provide the tools for modular design and specification in rewriting logic, keeping always in sight its future use, in particular, for the modular analysis and verification of complex systems.

We have called *synchronous composition* to the operation that allows assembling systems keeping them in sync. It is based on two ingredients: the definition of *properties* on states and transitions, and the specification of syncing criteria using the values of such properties.

This is what the reader can expect from the rest of the paper. In Section 2 we explain what our goals are and how they have driven us to particular choices and definitions. All the formalisms are in Section 3. Sometimes the formal definitions turn out to be annoyingly complex or long, and we find that the previous

explanations are needed so that the underlying ideas can be grasped. Then, in addition to the motivational examples above, Section 4 contains a few simple but complete examples for illustrative purposes, and Section 5 contains the implementation of a more complex example about cached computer architecture, aimed at showing the full power of our proposal. Of course, modularity and compositionality are by no means new ideas—Section 6 discusses related work. We expect that bringing such ideas into rewriting logic, with its strong formal basis and its existing implementations, will provide new niches of applicability and novel views on existing work. The next steps we intend to complete are discussed in Section 7, and conclusions are exposed in Section 8. All proofs are postponed to the appendix.

This is an extended version of our paper [**?**]. It contains additional examples, methodology notes and tricks, and the proofs of all the theorems. The latest version of this paper and related material can be downloaded from our website: `http://maude.sip.ucm.es/syncprod`.

2 Goals, ideas, and choices

Next we discuss the different choices we have been driven to make towards our definition of the synchronous composition operation, and the rationale behind each choice. We are interested in transition structures and rewrite systems, and the synchronous composition for both. We discuss them together. For the time being, an intuitive acceptance that rewrite rules represent transitions is enough.

One message we want to convey in this section is that most choices were dictated to us by our two main goals. Our number one goal, again, is to provide flexible tools for modularity in rewriting logic. The second is to be able to use existing theoretical results and practical tools on our specifications. By practical tools we mean model checkers, state-space explorers, and execution engines, for example. Some useful theoretical results are about the computability of systems, the use of abstraction, semantics, and so on. We want to depart as little as possible from the standard definitions of rewriting logic, so that we can benefit from all the existing machinery around it.

Rewriting logic has proven useful for tasks such as the formalization of language semantics and the emulation of other logics. Our interest, however, is only focused in system modelling, specification, and analysis. This focus has also guided some of the choices described below.

2.1 Being egalitarian

Using actions for syncing, rather than states, is in many cases the natural choice, as illustrated in the two examples above—the two trains and the computer architecture. Thus, it is unfortunate that actions, or transitions, are often treated in a discriminatory fashion with respect to states. In labelled transition structures, for example, it is usually the case that we can define propositions on states, but actions are only given atomic and non-unique identifiers. In rewriting logic, states are represented by terms of any complexity, but rules are only given atomic labels. This provides little flexibility for dealing with actions.

Syncing states is also useful, so we don't want to stick to an action-only formalism. We need to be *egalitarian*. In [**?**] we already argued for the con-

venience of treating states and transitions as equals, and we proposed a kind of systems that we already called *egalitarian*. Those systems were different, though related, to the ones by that adjective in the present paper. The same point had been made before, if only partially, for instance in [?, ?, ?]. Be our task either the specification of systems or of their temporal properties, it can be made simpler and more natural with an egalitarian view. In [?] we showed that also the synchronous composition of systems benefits from being egalitarian.

From the point of view of transition structures, this means that we are going to use propositions (or rather *properties*, as defined below) both for states and for transitions. From a rewriting logic point of view, this means that we represent transitions (as well as states) by terms. Proof terms, as described in [?], can be used in rewriting logic to represent transitions. But, to be egalitarian and to achieve our goals in this paper, proof terms are not appropriate, and we need to somewhat redefine the very concept of transition, as we do next.

2.2 What is a transition?

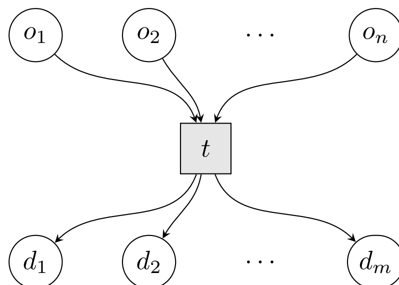
In our toy computer architecture example, consider now the way to deal with a reading instruction `(r, A)`, that is, a request to obtain the value stored in memory address `A`. In Maude-like syntax, the processor part could be written like this:

```
⟨In module PROCESSOR⟩:
   rl [doingR] : ProgCounter: N
                 Instr: (r, A)
                 Data: D
              => ProgCounter: N + 1
                 Instr: void
                 Data: D' .
```

That is, at the start, the processor has got an instruction to read the contents of memory address `A` and, after *performing* the reading, it has stored in its register the new value `D'`. Different instances of the rule represent different transitions. Again, for any of these transitions to be meaningful, they have to interact, to be synced, with some actions at the memory side, but we do not care about these right now.

The point to note here is that each transition, each instance of the rule `doingR`, represents a process of reading, that we can picture as taking place over a certain time span, and the particular value of `D'` is only available *after* the execution of such a process, not *while* executing it. Thus, `D'` cannot be an attribute of a transition represented by the rule `doingR`. It is only an attribute of the destination state. Therefore, the same transition based on `doingR` can take the system to any of a set of destination states, each with a different value for `D'`. For similar reasons, `D` needs not be known to the transition, but only to the origin state.

Such is the rationale behind our concept of *transition*. This is our proposal: whenever from any state in a set of origin states $\{o_1, \ldots, o_n\}$ a system can reach in one step any state in a set of destination states $\{d_1, \ldots, d_m\}$, it is fair to consider such step an only and same transition, irrespective of the actual origin and destination states used in each actual run of the system.



For another example, consider now the rule for updating the memory, that we wrote above and repeat now:

⟨In module `MEMORY`⟩:
```
rl [updating] : (A, D'') RestOfMemory
             => (A, D) RestOfMemory .
```

This represents an updating of the data stored in memory address `A`. The data `D''` is better seen as a property belonging to the origin state, irrelevant once the updating process starts. However, the new data `D` is better seen as a property belonging to the transition (and to the destination state as well), because the updating process needs to work with the new value since the moment it starts. Indeed, at the processor side, the rule `doingW`, the one that has to sync with `updating`, has the new value `D` available already in its origin state, as part of the instruction to be executed: `(w, A, D)`.

Transitions are, thus, freed from the usual "single origin and single destination" convention. A proof term, as defined in [?] for rewriting logic, univocally identifies a rewriting step, and contains all the information to recover the origin and destination state terms. For the rule above, a proof term has the form `updating(A, D'', RestOfMemory, D)`. This is too restrictive for us. We want to remove from a transition term all the information that belongs rather to their origin or destination states. The transition term we need in this case is `updating(A, RestOfMemory, D)`. From this term it is not possible to recover the particular origin state, but it is possible to recover the set of possible origin (and destination) states. In the case of rule `doingR` above, the transition term needed is `doingR(A, RestOfMemory)`.

We said above that it is *fair* to consider that some steps are instances of the same transition; we didn't say it is *mandatory.* There can be reasons to be otherwise. In the motivational example of Section 1.1, `crit(1)` and `crit(2)` are different transitions, even though they share their unique origin and destination states, `readyToEnter` and `justExited`. We need them to be different transitions so that each can be synced to the action of a different train. In brief: a transition must be able to take the system from any of its origin states to any of its destination states. As we refer to this requirement several times in the rest of the paper, let us give it a name: it is the "from any to any" condition for transitions. Long and funny name, but clear. It also holds for states—indeed, it is a trivial thing: for each state, any transition that takes the system to it can be followed by any transition that takes the system from it. Thus, our concept of transition is egalitarian, which is nice. But the real reason for choosing this concept of transition is that we need it for syncing to work properly.

In agreement with all this, in the egalitarian transition systems we define below, transitions are not just represented by arrows from a state to another,

but by boxes with in and out arrows, as in the diagram above. (The graphical aspect is similar to a Petri net, but the workings are different: in a Petri net, a transition fires only when all its in-places are marked, and then all its out-places get marked; in egalitarian transition structures, only one in-state needs to be *marked* for the transition to *fire*, and only one out-state gets *marked*.)

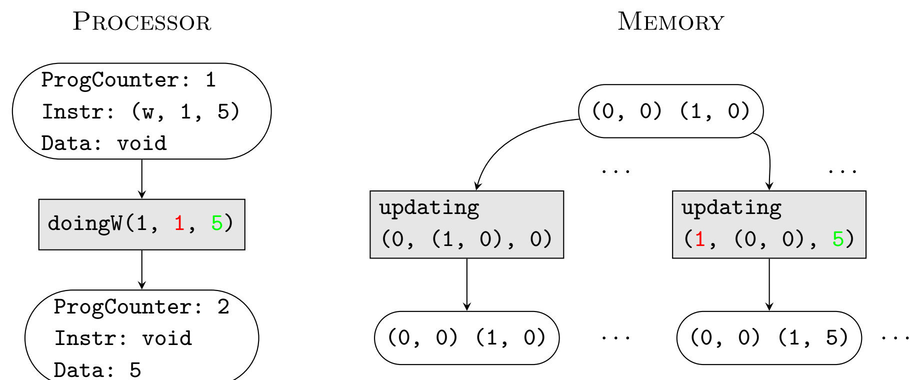
Translating all of this to rewriting logic leads us to some tweaking in the definition of the logic. It is fully described below. The idea is that, in egalitarian rewriting logic, we need to label rules, not with atomic labels, but with terms showing which parameters we have chosen to belong to the transition (always obeying the "from any to any" condition).

2.3 Syncing on properties

Consider once again the rules `doingW` and `updating` from Section 1.2, giving rise to transitions with terms of the form

`doingW(N, A, D)` and `updating(A, RestOfMemory, D)`,

respectively. Let's choose some concrete values for the initial states, as shown in this picture:



In words, the processor's state is telling:

- the next instruction to be executed is the one identified with number 1 in the program;
- the processor has already received and stored such an instruction, which is a request to write 5 at memory address 1;
- the data register is void, maybe because we have just begun executing the program.

From there, the processor can only perform the transition shown, reaching the state shown. About the memory, we assume for simplicity that it only stores two data, at addresses 0 and 1. The initial state stores two zeros. From there, the memory is ready to perform any transition following the rule `updating`. Two possible transitions and destination states are shown in the picture.

When the memory and the processor are composed and run synced, we want the memory to execute only the transition that matches the values from

the processor transition term: the `1` and the `5`. In this case, we need to sync on a pair of natural numbers. In other cases, more complex data may be needed for syncing.

Thus, we use general functions on states and on transitions. We refer to these functions as *properties*. For syncing, we need to compare the values of such properties for equality. We require their result sort to allow for such a comparison. No other requirement is made on the sorts of properties. We call *syncing criteria* to the requirements that certain properties must have equal values at both systems.

There is one more tweak here: properties may be undefined at some states and transitions. In the example above, the processor needs two properties with names, say, `addressBeingAccessed` (the 1) and `dataBeingWritten` (the 5). What can the value of `dataBeingWritten` be while the processor is reading or doing something else different from writing? That property makes no sense at those points. No value can be assigned to it. Thus, we do not require that each state and transition assigns a value to each property. When a property is undefined at a state or transition, it does not impose any conditions to sync (regarding that property—there may be others).

A very similar idea has been more thoroughly studied, in a different setting, in [**?**, **?**, **?**, **?**], under the name of *partial structures*. We also proposed something similar in [**?**].

2.4 Syncing on relations other than equality

Sometimes, other relations different from equality may be needed for syncing. For instance, we may require that the value of property p_1 in system $\mathcal{S}_1$ be less than the value of property p_2 on system $\mathcal{S}_2$. Or that p_1 is an element of the set p_2. This seems to be a good place to discuss this more technical issue. A realistic instance of this problem is in Section 5.6, as part of our example on cache architecture.

Allowing for arbitrary conditions for syncing is a bad idea. It would produce more complex specifications both syntactically and conceptually. More important is that modularity would be compromised: the composed system would not be just a straightforward sum of the components. Modular verification is based on the idea that if system $\mathcal{S}_1$ is proved to satisfy formula φ_1 and system $\mathcal{S}_2$ is proved to satisfy formula φ_2, we can deduce that the composed system satisfies a formula derived from φ_1 and φ_2. But if the composed system depends not only on the specifications of the components, but also on exotic syncing conditions, modular verification would get much more difficult. Fortunately, there is a way to handle these cases, reducing them to equality. It involves the use of families of properties, even potentially infinite ones.

Suppose that in system $\mathcal{S}_1$ we have a property p_1 whose values range over some sort C_1; and in system $\mathcal{S}_2$ we have a property p_2 whose values range over C_2. To allow state s_1 to be visited in $\mathcal{S}_1$ simultaneously with s_2 in $\mathcal{S}_2$, we require that some relation $p_1(s_1)\ R\ p_2(s_2)$ holds, writing as $p_i(s_i)$ the value of property p_i at state s_i. We could discuss the same for transitions, but let us focus on states. For example, $p_1(s_1)$ and $p_2(s_2)$ are numbers and R is $<$. Or $p_2(s_2)$ is a set, $p_1(s_1)$ is a value, and R is $\in$. We show next how to define new properties whose equality is equivalent to those relations. For that, we need to use parametric families of properties: $p(x)$, one such for each possible value of

the parameter x. Often, when we do this, the set of parameters for which the property is defined (as opposed to *declared*) is finite, even just one.

This is the way. In $\mathcal{S}_1$ we declare a family of Boolean properties $q_1(x)$. The type of the parameter x is C_2, the type returned by p_2. We define the value of property $q_1(x)$ at state s_1, based on the value of p_1, like this:

$$q_1(x)(s_1) := p_1(s_1)\ R\ x.$$

This is defined for all values of the parameter x in C_2. Then, in system $\mathcal{S}_2$, we declare the family of Boolean properties $q_2(x)$, with x again of sort C_2. At each state s_2, just one property in the family is defined, namely, when the parameter x is $p_2(s_2)$:

$$q_2(p_2(s_2))(s_2) := \text{true}.$$

All other properties in the family are undefined at s_2. We assert that requiring the equality $q_1(x)(s_1) = q_2(x)(s_2)$ for all x for which both properties are defined at those states is equivalent to requiring $p_1(s_1)\ R\ p_2(s_2)$. Indeed, the only parameter for which both families are defined on the given states s_1 and s_2 is $x = p_2(s_2)$. Undefined properties do not pose any restriction, so that equality of families reduces to one equality: $q_1(p_2(s_2))(s_1) = q_2(p_2(s_2))(s_2)$. According to the definitions of both properties, this is equivalent to $p_1(s_1)\ R\ p_2(s_2) = \text{true}$.

For a concrete example, consider again the toy computer architecture. Suppose that we need to require the following syncing condition: when the processor stores a reading or writing instruction mentioning a given memory address, we need to avoid that the memory is storing zero at this particular address. We cannot think of a good reason for such a requirement, but let us admit it. We need a property defined in `PROCESSOR` (system $\mathcal{S}_2$), that we call `address` (corresponding to p_2), that returns the address stored in the processor's instruction register. We also need a property defined in `MEMORY` (system $\mathcal{S}_1$), called `nonZeroAddressSet` (corresponding to p_1), that returns the set of addresses for which the stored value is not zero. The required syncing criterion is `address` ∈ `nonZeroAddressSet`. We use the procedure described above to state this criterion using equalities. For that, we need two families of properties, that we call `isInNonZeroAddressSetOf` and `isAddressIn` (corresponding to q_1 and q_2, respectively). We define them as infix operators, both parametric in their first argument, by these equations:

```
⟨In module MEMORY⟩:
   eq X isInNonZeroAddressSetOf S = X ∈ nonZeroAddressSet(S) .
```

and

```
⟨In module PROCESSOR⟩:
   eq address(S) isAddressIn S = true .
```

Now, syncing on

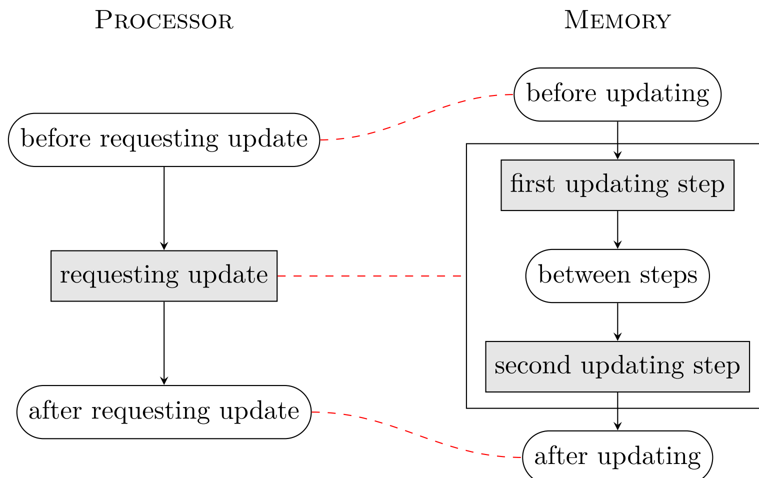
```
PROCESSOR.(X isAddressIn S) = MEMORY.(X isInNonZeroAddressSetOf S)
```

produces the desired result. A good choice of identifiers is definitely helpful here.

2.5 Syncing states with transitions

The rule `doingW` at the processor side, as it has already been discussed, must be synced with whatever actions perform the real updating at the memory side. The processor can not know precisely what is happening in the memory, and

must not care about it. In our implementation above there was just one rule for updating the memory but, if the specification of the memory were more refined, there may well be several rules involved. It must be possible to sync one transition at one side with more than one at the other side, just based on the values of properties.



In our case, the *before* states on both systems must sync; also the *after* states must sync; and the unique transition at the processor side must sync with two transitions and one state at the memory side. In particular, one process (the memory) is in a state ("between steps") while the other (the processor) is in a transition ("requesting update"). This kind of heterogeneous syncing is unavoidable.

Therefore, the norm is: either states or transitions in one system can be visited at the same time as either states or transitions in the other system if they agree on the values of their properties. In the general case, several ways of combining states or transitions at both systems are allowed by the properties. In these cases, the way to proceed is non-deterministically chosen, as usual in rewrite systems.

The final consequence is that the boundary between states and transitions in composed systems disappears or is blurred. For, what can we say about a system with two components, one in a state while the other in a transition? The composition is not purely in a state and not purely in a transition. When all the components happen to be simultaneously in states (resp., transitions), it is still correct to say that the composed system is in a pure state (resp., transition). Correct, but devoid of any importance. We become, in this way, utterly egalitarian. The result is like achieving equality among women and men by removing all sexual features. It may be boring, but it is effective.

When we want to refer to either states or transitions or any composition of them, we call them *stages*. The system composed by syncing $\mathcal{S}_1$ and $\mathcal{S}_2$ is denoted by $\mathcal{S}_1 \| \mathcal{S}_2$. A stage in it is written as $\langle g_1, g_2 \rangle$ if each g_i is a stage of $\mathcal{S}_i$.

2.6 The split

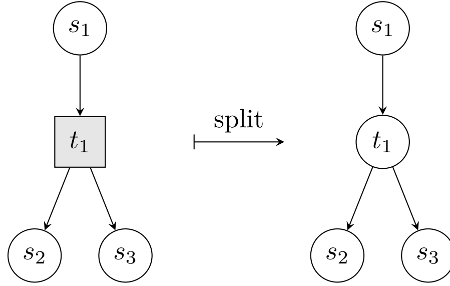
At the start of this paper, we envisioned egalitarian transition structures as represented by bipartite graphs, with states and transitions interleaved. After

the discussion just above, these structures are only valid as *atomic* components. Non-atomic structures are given as a set of atomic ones together with the required syncing criteria.

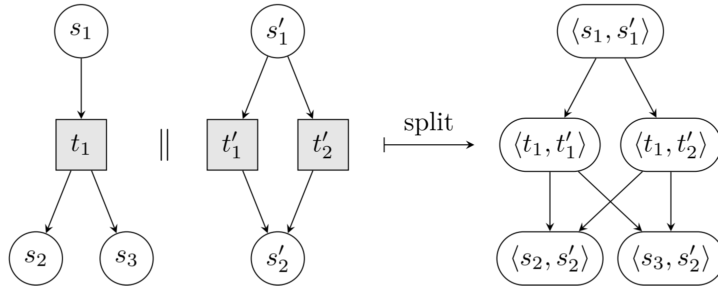
We have stepped into a problem now. Our goal for powerful modularity has driven us to use structures and systems (and sets of them) that dangerously depart from the standard ones. Our second main goal—using existing machinery—is compromised. Maude's model checker, for instance, works by traversing a Kripke structure that is a model of the specification, so it is not usable on non-standard structures. Fortunately, there is a way out; we call it the *split* operation.

Splitting applies both to transition structures and to rewrite systems, either atomic or composed. Splitting a structure or a system produces a standard one with equivalent information and behaviour. Splitting a structure or a system consists in making each state, transition, or whatever mixture of them in the original composed system into a state (of a new kind) in the resulting one. An egalitarian transition structure, through splitting, becomes a standard one. An egalitarian rewrite system, through splitting, becomes a standard rewrite system. The reason for the name *split* is that a labelled rewrite rule like $t : s \to s'$ in an egalitarian rewrite system becomes split as $s \to t$ and $t \to s'$. What was a rule label, t, becomes a state term in the split system.

The split of an atomic transition structure consists in turning transitions into new states:



The following is an example for a composed structure. So that the resulting structure is small, we are assuming that properties have been defined in both component structures that allow states to be synced only with states, and transitions only with transitions:



States like $\langle s_1, t'_1 \rangle$ are not valid in this case, according to the properties we are assuming for syncing. There are three more valid states in the result: $\langle s_1, s'_2 \rangle$, $\langle s_2, s'_1 \rangle$, and $\langle s_3, s'_1 \rangle$. These are not reachable from the ones shown.

The split operation allows us to translate any problem posed on egalitarian structures and systems to one on standard split ones. For instance, temporal properties can be translated and Maude's model checker can be used on the split system to draw conclusions about the original egalitarian one.

2.7 Topmost everyone

In rewriting logic a rule $a \to a'$ can be used to rewrite any term that contains a. For instance, the term $f(a, d)$ is rewritten to $f(a', d)$ using that rule. In some cases, however, it is convenient to ask that all rules are *topmost*, that is, that they rewrite complete state terms, and not subterms. Ours is one of those cases.

As detailed below, the synchronous composition uses the components' rules to produce the composed ones. If system A includes the rule $a \to a'$ and system B includes $b \to b'$, then the composition $A \| B$ can include (if syncing conditions allow) the rule $\langle a, b\rangle \to \langle a', b'\rangle$, to be used on composed states. But, while $f(a, d)$ can be rewritten by $a \to a'$, no term of the form $\langle f(a, d), f(b, e)\rangle$ can be rewritten by the composed rule $\langle a, b\rangle \to \langle a', b'\rangle$, because $\langle a, b\rangle$ is not a subterm of $\langle f(a, d), f(b, e)\rangle$. The way to ensure that rewrites are preserved by composition is asking for all rules to be topmost.

The paper [**?**] shows results on *completing* rules so as to transform a rewrite system into an equivalent topmost one. The idea is to substitute rule $a \to a'$ by $f(a, x) \to f(a', x)$ and any other that could be needed for other contexts in which a can occur. Many interesting rewrite systems are amenable to completion in this way.

Our view is different. Instead of completing rules, we want to decompose systems. Suppose, for example, that the states of our system are given by pairs (a, b), and that there are rules to rewrite the complete state term and also for a part of it:

$$l : (a, b) \to (a', b')$$
$$m : a \to a''$$

In many cases, this can be interpreted as a hint that a represents a meaningful component, able to evolve by itself. In consequence, we propose to decompose the system into two topmost ones: one for the a part, with rules

$$l' : a \to a'$$
$$m : a \to a''$$

and another for the b part, with rule

$$l'' : b \to b'.$$

Then, we need to add the syncing criteria to make rules l' and l'' be run only simultaneously, so as to produce the same result as the composed rule l.

Although we have not obtained theoretical results as yet, we have found this method useful when specifying our examples, and we expect it to allow us to transform many interesting systems into a synchronous composition of topmost ones.

2.8 True concurrency

True concurrency is a given in rewriting logic. When two rewrite rules refer to disjoint subterms of a state term, they can be executed at the same time. (Though many of the tools around Maude work with an interleaving semantics.) An inconvenience of topmost rewrite systems is that concurrency is prevented. As each rule uses the complete state term, there is no room left for another rule to act concurrently on a different subterm.

In our setting, as we explain throughout this paper, this is desirable, because we use these rewrite systems to represent individual and indivisible components. Concurrency is achieved by composing several components and allowing them to evolve simultaneously. Carefully chosen values for properties can mandate simultaneity of actions in different components, or can prevent it. When no restrictions are made, both concurrency and interleaving are possible. Thus, true concurrency—as opposed to interleaving semantics—is the natural choice if we are talking about actions taking place in different components. This is formally defined in Section 3.

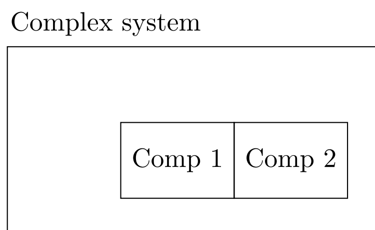
2.9 The whole is the sum of its parts

In object-oriented methodologies it is usual to design a class so that, in addition to using other classes in different ways, it adds its own code and data. Also in Maude it is possible to import a module and extend it:

```
mod A-MODULE is
   including OTHER-MODULE .
   --- code that uses or extends OTHER-MODULE
endm
```

We do not want this to happen when performing synchronous composition. A system is just the result of composing its component systems. The actions of the composed system are either individual or synced actions of the components. No new data and no new transitions are added once the synchronous composition has been produced.

In some cases, we may be tempted to specify separately just one or a few components of a complex system, while the rest is just part of the complex composed system:



Our aim for clean and complete modularity drives us to see this as the composition of three components: Comp 1 $\|$ Comp 2 $\|$ Rest. Modular verification, for instance, would be difficult otherwise.

We do admit, however, new properties to be defined for the composed system, as explain in Section 2.11.

2.10 Explicit syncing criteria

In automata theory, the synchronous product is defined so that the two automata execute the same action at the same time. Here, "the same action" means "an action with equal name." Ditto for the parallel composition of processes in CSP, and also in CCS, though with not equal but complementary names in this last case. If we need to sync action a from process P with action $\overline{b}$ from process Q, the CCS way is renaming a to b in P and composing the result with Q (that is, $P[b/a] \mid Q$), with the implicit understanding that b syncs with $\overline{b}$ just because they are actions with complementary names. For some references on automata, CSP, and CCS, see, respectively, [**?**, **?**, **?**]. Instead of relying on implicit understandings, we prefer to be explicit about which properties sync with which others in our synchronous compositions.

The way we have chosen is the following. A synchronous composition is denoted as $(\mathcal{S}_1 \| \mathcal{S}_2 \| \cdots \| \mathcal{S}_n)_Y$. It has a parameter in addition to the systems being composed: a set Y of *syncing criteria*, each of them a pair like $p_1 = p_2$ (or $\mathcal{S}_1.p_1 = \mathcal{S}_2.p_2$, if we need to be more explicit), with each p_i a property from $\mathcal{S}_i$, specifying properties to sync on.

All the following extremes are gladly accepted:

- No syncing requirements at all:

$$(\mathcal{S}_1 \| \mathcal{S}_2)_\emptyset.$$

- Syncing more than two systems at once:

$$(\mathcal{S}_1 \| \mathcal{S}_2 \| \mathcal{S}_3)_Y,$$

  where Y uses properties from the three systems. This can certainly be done in two steps, $((\mathcal{S}_1 \| \mathcal{S}_2)_{Y'} \| \mathcal{S}_3)_{Y''}$, for appropriate parameters, but sometimes it is more natural to do it at once. This has been suggested above, in Sections 1.1 (on two trains and a mutual-exclusion controller) and 1.2 (on the processor, the program, and the memory).

- The same property is used in more than one syncing criteria:

$$(\mathcal{S}_1 \| \mathcal{S}_2)_Y, \text{ where } Y \text{ includes } \mathcal{S}_1.p_1 = \mathcal{S}_2.p_2 \text{ and } \mathcal{S}_1.p_1 = \mathcal{S}_2.p'_2.$$

  Or with three systems:

$$(\mathcal{S}_1 \| \mathcal{S}_2 \| \mathcal{S}_3)_Y, \text{ where } Y \text{ includes } \mathcal{S}_1.p_1 = \mathcal{S}_2.p_2 \text{ and } \mathcal{S}_1.p_1 = \mathcal{S}_3.p_3.$$

  These can be seen as complicated ways to write $\mathcal{S}_1.p_1 = \mathcal{S}_2.p_2 = \mathcal{S}_3.p_3$. Three-way equalities like that are not accepted for the sake of simplicity in formal definitions.

- Syncing within one single system:

$$\mathcal{S}_Y, \text{ where } Y \text{ includes } \mathcal{S}.p = \mathcal{S}.p'.$$

  This may seem odd, but we use it in our example on cache architecture to implement the coherence protocol; see Section 5.15.

  Such a degenerated composition is to be interpreted like this: the system $\mathcal{S}_Y$ behaves like $\mathcal{S}$, but only is allowed to visit a state or transition if the values of p and p' are equal at it.

2.11 Properties of the composed system

Properties are defined on a system only after its states and behaviour have been specified. Suppose, for instance, that system $\mathcal{S}_1$ has been specified and, then, property p_1 has been defined on it. Same for $\mathcal{S}_2$ and p_2. We can now specify a new system

$$\mathcal{S} := (\mathcal{S}_1 \| \mathcal{S}_2)_{p_1 = p_2}.$$

This composed system $\mathcal{S}$ has no properties of its own as yet. If $\mathcal{S}$ is in turn going to be used as a component for another synchronous composition operation, properties are needed.

With modularity in mind, we request that all properties of a composed system are defined in terms of the properties of the components, not in terms of their internals. Sometimes a property p of system $\mathcal{S}$ is just inherited from one of its components:

$$p(\langle g_1, g_2 \rangle) := p_1(g_1).$$

Sometimes definitions are more complex:

$$p(\langle g_1, g_2 \rangle) := p_1(g_1) + p_2(g_2) \quad \text{if} \quad p_2(g_2) \neq 0.$$

The important condition to keep in mind is that the argument of p in the left-hand side of such defining equations cannot be more concrete than $\langle g_1, g_2 \rangle$, that is, it cannot include any details from the particular implementation used inside $\mathcal{S}_1$ and $\mathcal{S}_2$. The aim is that $\mathcal{S}_1$ can be replaced by an equivalent $\mathcal{S}'_1$ that provides the same properties and everything else works without changes.

The specification of the new properties forms a set we call Z, that we write as another subscript for the synchronous composition: $(\mathcal{S}_1 \| \mathcal{S}_2)_{Y,Z}$. More on this in the formalizations in Section 3.

2.12 Summing up

We have up to now described our goals, choices, their rationales and some of their consequences. The result is that a system specification is formed by the specifications of a series of *atomic*, topmost components. In them, we are equally interested in states and transitions. On each system, properties are defined. They work as interfaces, or ports, or handles, or wires, or communication buses, or just attributes, different metaphors being appropriate in different cases. Properties may be partial functions, they need not be defined on all states and transitions. Then, syncing criteria are specified as pairs of properties whose values, if defined, must me equal for all component systems at all times. The resulting composed systems can, in their turn, be composed. Properties are then defined for the composed systems solely based on the values of the components' properties.

This is all formalized in the next section, both for transition structures and for rewrite systems.

3 Formal definitions

3.1 A bird's-eye view of it all

We define and use below a number of structures and systems to which we give special names and symbols. The symbols are according to the BNF-like expres-

sion

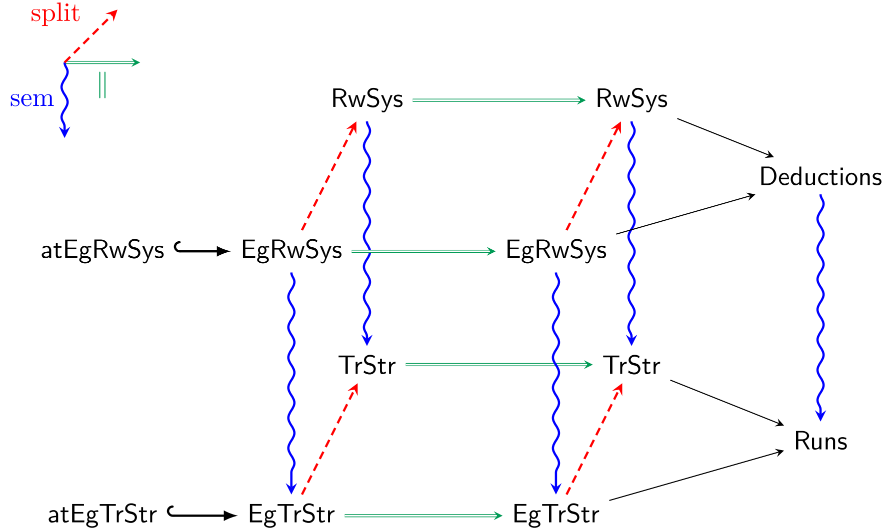
$$[[\mathsf{at}]\mathsf{Eg}]\{\mathsf{TrStr} \mid \mathsf{RwSys}\}$$

with the respective meanings

$$[[\text{atomic }]\text{egalitarian }]\{\text{transition structures} \mid \text{rewrite systems}\}.$$

Usually, we call *plain* transition structures to the elements of TrStr, to avoid ambiguity, because also the elements EgTrStr and atEgTrStr are, indeed, transition structures of a different kind. Respectively for RwSys. We also use Deductions for sets of deductions and Runs for sets of runs. All this is explained below.

This diagram shows the whole set of structures and systems with their related maps:



Slanted dashed arrows represent the several concepts of split. Double horizontal arrows represent synchronous composition of systems or structures. Downward snake arrows represent semantic maps. (Runs and deductions can be seen as providing trace semantics, but the assignment of a transition structure is usually seen as *the* semantics of a rewrite system.) Atomic systems and structures are treated as particular cases in all the definitions in this section; thus, we do not need to show in the diagram explicit arrows for the split, semantics, and composition of atomic systems and structures.

It is argued below that there is a commutative diagram in each of the nine different faces of that polyhedron. (The interior vertical face also counts, but it is the same as the left one.) These are nine theorems to be stated, explained, and proved. All this commutativity has practical consequences. For example, we can use model checking on the runs of some structure (at the lower right-hand corner of the figure) to modularly verify a composed rewrite system (at the upper left-hand corner), with complete flexibility on the path we choose from corner to corner.

3.2 Transition structures

3.2.1 atEgTrStr

We denote by atEgTrStr the class of *atomic egalitarian transition structures*. They are the basic building blocks for all our transition structures. An element of atEgTrStr contains two kinds of nodes: states and transitions. They can only occur interleaved. We call *stages* to either states or transitions, and use variables typically called g, with or without subscripts or superscripts, to range over them.

Formally, an element of atEgTrStr is given by a tuple $\mathcal{T} = (S, T, \to, P, g^0)$, where:

- S is the set of states;
- T is the set of transitions;
- $\to \subseteq (S \times T) \cup (T \times S)$ is the bipartite adjacency relation;
- P is the set of properties—partial functions defined on stages with different codomains: $p : S \cup T \nrightarrow C_p$;
- $g^0 \in S \cup T$ is the initial stage.

The adjacency relation allows for several arrows in and out of a transition, as well as a state. Also, the egalitarian goal mandates that not only an initial state is possible, but also an initial transition. Think of it as if we start studying the system when it is already doing something. Each property $p \in P$ can have a different codomain C_p. We pose no restrictions on property's codomains, except that their elements can be tested for equality, because this is what we need for syncing.

The set P of properties is allowed to be infinite, but with an important remark: the only practical way of declaring and defining an infinite set of properties is that they be parametric, that is, $q : A \to P$, for a (possibly) infinite set of parameters A, so that $q(a)$ denotes a different property for each $a \in A$. For example, a memory storage can define a different property for each memory address it stores. We used parametric properties in Section 2.4 and use them again several times in the examples below.

So that our definitions are as close as possible to an eventual implementation, we make this requirement now:

> The set P of properties is composed of a finite (maybe empty) set of individual properties plus a finite (maybe empty) set of parametric properties.

Sets of properties are meant to be compared for syncing. Having an infinity of values to compare is problematic in practice. Thus, we have new requirements in the next section.

In some cases, initial stages are unimportant. Also, properties are only needed if the structures are going to be composed. Thus, by a slight abuse of language, we accept as elements of atEgTrStr structures in which no P or no g^0 are provided. Ditto for all the transition structures and rewrite systems discussed in this paper.

3.2.2 EgTrStr

The class atEgTrStr is not closed under the synchronous composition operation, because combinations arise that are not properly states nor transitions. That is why we need the more general EgTrStr. The class EgTrStr is the smallest one that includes atEgTrStr and is closed under synchronous composition. Thus, an element of EgTrStr either is in atEgTrStr or has the form $\mathcal{T} = (\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z}$, with $n \geq 1$ and with each $\mathcal{T}_i$ being in EgTrStr itself. Each $\mathcal{T}_i$ may be, but need not be, in atEgTrStr. The set Y contains the syncing criteria, and Z contains the definition of the properties for $\mathcal{T}$; both are described below.

Remember that in atEgTrStr a *stage* is either a state or a transition. For $\mathcal{T} = (\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in$ EgTrStr, a *stage* is a tuple $\langle g_1, \ldots, g_n \rangle$, where each g_i is a stage of $\mathcal{T}_i$, that is, each g_i is a tuple in itself, unless $\mathcal{T}_i \in$ atEgTrStr. We call them *stages* because they represent stages in the evolution of a (composed) system, and because, like on a theatrical stage, there may be several actors (components) performing at the same time and relating to each other in different ways.

As we have already been doing, we use the name of a structure or a system as a prefix whenever we need to state explicitly where an element belongs or where it was initially declared. Thus, we write $\mathcal{T}.p$ and $\mathcal{T}.g_1$ for a property and a stage from structure $\mathcal{T}$.

The set Z contains the definitions of the properties of $\mathcal{T}$ based on the properties of the components. See the remarks in Section 2.11. Each element of Z is written as an assignment $p := F(p', p'', \ldots)$, where p is the name of the new property for $\mathcal{T}$, the arguments p', p'', etc. are properties from some of the components (that is, $\{p', p'', \ldots\} \subseteq \bigcup_i \mathcal{T}_i.P$), and F is some functional expression providing the recipe to compute the value of p based on the values of p', p'',... at each given stage. Remember that a property needs not be defined at all stages; if some of F's arguments are not defined at a given stage, neither is p. The set of properties defined by $\mathcal{T}.Z$ is denoted as $\mathcal{T}.P$, to keep it coherent with the atomic case.

The syncing criteria in Y are pairs of properties: $Y \subseteq \bigcup_{i,j=1}^{n} \mathcal{T}_i.P \times \mathcal{T}_j.P$. Instead of (p_i, p_j), we write each such pair as $p_i = p_j$, or $\mathcal{T}_i.p_i = \mathcal{T}_j.p_j$. The idea is that the structure $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z}$ only visits stages that are compatible with respect to Y, that is, that satisfy all the syncing criteria in Y. Formally, a stage $\langle g_1, \ldots, g_n \rangle$ is *compatible* wrt Y if for each criterion $p_i = p_j \in Y$, where p_i is a property of $\mathcal{T}_i$ and p_j is a property of $\mathcal{T}_j$, we have that $p_i(g_i) = p_j(g_j)$ when both properties are defined. In those cases, we say that the stage $g = \langle g_1, \ldots, g_n \rangle$ is compatible or, with the same meaning, that the stages $g_1, \ldots, g_n$ are compatible, in both cases wrt Y.

That the stages g_i and g_j are compatible is a necessary but not sufficient condition for them to be visited at the same time, because there can be other allowed combinations. For instance, the stage g'_j in $\mathcal{T}_j$ may also be compatible with g_i. Each run of the system must non-deterministically choose one of the allowed ways to go on. It is always possible to define the values of properties in a tighter way so as to achieve exactly what we need.

For parametric families of properties, a syncing criterion $q_1 = q_2$ means equality for each value of their parameter, that is, $q_1(x)(g_1) = q_2(x)(g_2)$ for each x for which both $q_i(x)(g_i)$ and $q_j(x)(g_j)$ are defined. When the parameter

is allowed to take on an infinite set of values, problems certainly arise. So, we require:

> Whenever a syncing criterion using parametric properties $q_i = q_j \in Y$ needs to be tested at stages g_i in $\mathcal{T}_i$ and g_j in $\mathcal{T}_j$, we assume that only for a finite amount of parameters x are both $q_i(x)(g_i)$ and $q_j(x)(g_j)$ defined.

This is usually not a problem in practice. For example, to make it possible for a processor to access the contents of memory at a given address a, we would declare a parametric property taking the address as parameter. At the memory side it would be defined for all addresses; at the processor side, however, it would only be defined for the address in which the processor in interested at each stage. See examples in Section 5.

The initial stage of $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in \mathsf{EgTrStr}$ can be straightforwardly defined as $g^0 = \langle \mathcal{T}_1.g^0, \ldots, \mathcal{T}_n.g^0 \rangle$. We always need to assume that the initial stage is compatible wrt Y.

It is also natural to define an adjacency relation in $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in \mathsf{EgTrStr}$ to describe the possible ways in which it can evolve in time. We say that (compatible) stages $g = \langle g_1, \ldots, g_n \rangle$ and $g' = \langle g'_1, \ldots, g'_n \rangle$ are adjacent, and write $g \to g'$, if for one or more $i \in \{1, \ldots, n\}$ we have that $g_i \to g'_i$ (recursively defined) and for the rest $g_i = g'_i$. The "one or more" part in the definition means that we allow several components to evolve in parallel.

3.2.3 TrStr and the split

An element of EgTrStr is a set of structures, only the basic ones being elementary graphs, together with instructions (criteria) on how to make them evolve in sync. Next, we define a new kind of structures, that we call just *transition structures* or, to avoid ambiguity, *plain transition structures*; we denote their class by TrStr. Plain transition structures are always single graphs. TrStr is closed under the appropriate notion of synchronous composition. We show below a translation from atEgTrStr and EgTrStr to TrStr, that we call *split*, that preserves all the information in the original structure. The importance of the split boils down to the fact that these non-egalitarian structures in TrStr are standard and well-known, or nearly so—they are almost Kripke structures.

A plain transition structure is, in essence, a graph with no multiple edges, whose nodes we interpret as states, and with properties defined on them. It is very much like a Kripke structure (as described, for instance, in [?]), except that we define properties on the states, instead of Boolean propositions. Formally, a plain transition structure is given by a tuple $(S, \to, P, s^0)$, where S is the set of states, $\to$ the adjacency relation, P the set of properties, and s^0 the initial state. Each property is a partial function from S to any codomain that admits comparison for equality.

In the previous section we have defined stages in EgTrStr, initial ones, adjacency relations among them, and properties defined on them. We can make all this into an element of TrStr through the translation that we call *split*. Formally:

- Given $\mathcal{T} = (S, T, \to, P, g^0) \in \mathsf{atEgTrStr}$, its split is $\mathrm{split}(\mathcal{T}) = (S \cup T, \to, P, g^0) \in \mathsf{TrStr}$.

- Given $\mathcal{T} = (\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in \mathsf{EgTrStr}$, its split is $\mathrm{split}(\mathcal{T}) = (G, \rightarrow, P, g^0) \in \mathsf{TrStr}$, where G is the set of stages of $\mathcal{T}$ compatible wrt Y, and $\rightarrow$, P and g^0 are the adjacency relation, the properties (defined in Z), and the initial stage for $\mathcal{T}$.

We can define a synchronous composition for TrStr that reflects the one for EgTrStr. Namely, given $\mathcal{T}_i = (S_i, \rightarrow_i, P_i, s_i^0) \in \mathsf{TrStr}$, for $i = 1, \ldots, n$, with $n \geq 1$, and given Y and Z as above, we define $\mathcal{T} = (\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z}$ as $\mathcal{T} = (S, \rightarrow, P, s^0)$, where

- $S \subseteq S_1 \times \cdots \times S_n$ is the set of tuples of states compatible wrt Y;
- $\langle s_1, \ldots, s_n \rangle \rightarrow \langle s'_1, \ldots, s'_n \rangle$ iff for each $i = 1, \ldots, n$ we have either $s_i \rightarrow_i s'_i$ or $s_i = s'_i$, with at least one instance of $s_i \rightarrow_i s'_i$;
- P is the set of properties as specified in Z; and
- $s^0 = \langle s_1^0, \ldots, s_n^0 \rangle$.

Although the operators "$\|$" for EgTrStr and for TrStr reflect each other (see Theorem 2), they are of a different nature: it is a constructor in EgTrStr, but an evaluable operator in TrStr. In other words, $(\mathcal{T}_1 \| \mathcal{T}_2)_{Y,Z}$ is an irreducible expression in EgTrStr, but can be reduced in TrStr as described in the previous lines of this paragraph.

3.2.4 Runs

For $(S, T, \rightarrow, P, g^0) \in \mathsf{atEgTrStr}$, or for $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in \mathsf{EgTrStr}$, or for $(S, \rightarrow, P, s^0) \in \mathsf{TrStr}$, we define a *run* in them as a sequence $g^0, g^1, \ldots$ of stages (or a sequence $s^0, s^1, \ldots$ of states, for TrStr) such that the initial stage (resp., state) of the sequence is the initial one for the given structure and, for each i, we have $g^i \rightarrow g^{i+1}$ (resp., $s^i \rightarrow s^{i+1}$) using the adjacency relation from the given structure. A run can be finite or infinite.

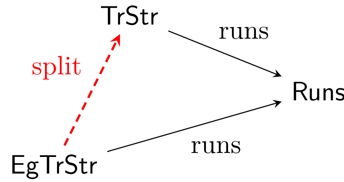
For each structure $\mathcal{T}$ of one of those kinds, a set of runs, denoted as $\mathrm{runs}(\mathcal{T})$, can be associated to it. We represent as Runs the class whose elements are sets of runs:

$$\mathrm{runs} : \mathsf{EgTrStr} \cup \mathsf{TrStr} \longrightarrow \mathsf{Runs}.$$

3.2.5 Commutativity theorems

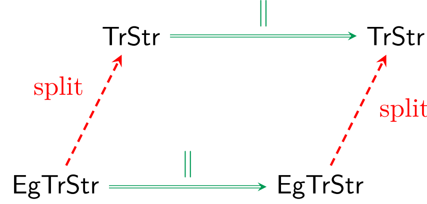
The two polygons on the lower floor of the figures in Section 3.1 are commutative diagrams. Let's state this as theorems. The proofs of these and the rest of theorems stated in this paper are in Appendix A.

Theorem 1. *The following diagram is commutative:*



That is, given an element of EgTrStr and its split, their sets of runs are the same. For model checking, for instance, we need to explore all runs in the state space of an egalitarian transition structure. Thanks to this theorem we can perform such an exploration, equivalently, on the more standard split structure.

Theorem 2. *The following diagram is commutative:*



This allows, for example, to perform verification on split($\mathcal{T}_1$) and split($\mathcal{T}_2$) (at the upper left-hand corner) and draw conclusions valid for $(\mathcal{T}_1 \| \mathcal{T}_2)_{Y,Z}$ (at the lower right-hand corner). Combining this theorem with the previous one, such a verification, in particular, can be model checking.

3.3 Rewrite systems

3.3.1 atEgRwSys

We slightly modify the standard definition of rewrite system to accommodate in it transitions of the kind we are interested in. We call the result *atomic egalitarian rewrite systems* and denote their class by atEgRwSys. The meaning of the adjectives *egalitarian* and *atomic* has already been discussed. Topmost rewrite systems defined in the standard way are atomic in our sense. Non-atomic systems are defined below as the synchronous composition of atomic ones.

An element of atEgRwSys is given by a tuple $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R)$ that satisfies all the requirements stated below, where:

- $(\Sigma, \leq)$ is the partial order of sorts and subsorts;
- Ω is a set of operator declarations like

  $$f : S_1 \times \cdots \times S_n \to S$$

  for $S_1, \ldots, S_n, S$ sorts in Σ;
- E is a set of (possibly conditional) equations;
- A is the set of equational attributes for the operators declared in Ω: `comm`, `assoc`, and so on;
- M is a set of (possibly conditional) membership axioms;
- R is a set of (possibly conditional) rewrite rules like

  $$t : s \to s' \text{ if } C$$

  where t is the label term and $C = \bigwedge_{j=1}^{m} C_j$ an ordered list of (equational, membership, matching, and rewrite) conditions.

The requirements and assumptions are the following:

- There are two sorts named `State` and `Trans`, whose terms represent states and transitions, respectively, and a supersort named `Stage` comprising both of them and nothing else.

- There is a constant `init` of sort `Stage` to represent the initial state or transition.

- The system must be topmost. This requires, first, that constructors are declared in such a way that no term of sort `State` or `Trans` can include a subterm of the same sort. Second, both sides of each rule, s and s', are terms of sort `State`, and t, the label of the rule, is a term of sort `Trans`.

- There is a sort named `Ppty` to represent properties. The sort `Ppty` is only to be used as a supersort of all property sorts. No terms can be declared of sort `Ppty`, but only of some of its subsorts. If it were a class in object-oriented methodologies, it would be called *abstract*. All subsorts of `Ppty` have names ending in `Ppty`; for example, `NatPpty` would be a subsort for properties whose values are natural numbers.

- Properties were formalized as functions in our transition structures, but are not represented by operators here, but by elements of some subsort of `Ppty`. The value of property p at stage g is written as $p\,@\,g$. Thus, there is an overloaded, infix operator @ (read as "at") for each subsort of `Ppty`. Namely, if `XPpty < Ppty`, the operator must be declared as

$$@ : \texttt{XPpty} \times \texttt{Stage} \to \texttt{X},$$

  for the appropriate sort `X`.

- The terms involved in an equation or in a membership axiom can be of whatever sort, including `State`, `Trans`, and `Ppty`.

- The system $\mathcal{R}$ includes a theory of the Booleans, declaring, in particular, the sort `Bool` and constants `true` and `false`.

- The values of properties must be comparable for equality. That is, for each subsort $\texttt{XPpty} < \texttt{Ppty}$, with $@ : \texttt{XPpty} \times \texttt{Stage} \to \texttt{X}$, there is in Ω an overloaded, infix, Boolean operator $= : \texttt{X} \times \texttt{X} \to \texttt{Bool}$. The equations in E must be able to reduce (modulo A) any expression $x_1 = x_2$, for x_1 and x_2 of sort `X`, to either `true` or `false`, and effectively so, because we need it for syncing.

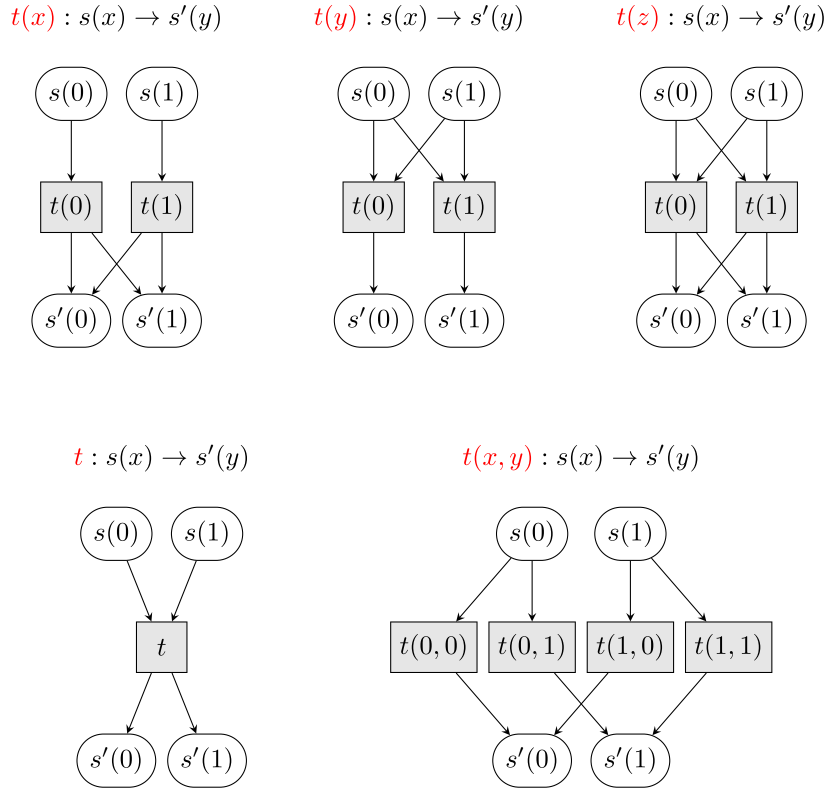
- We discussed in Section 2.3 that some properties may be undefined at some stages. Thus, the operator @ defines indeed a partial function: $@ : \texttt{XPpty} \times \texttt{Stage} \nrightarrow \texttt{X}$. We use the following convention:

  > Property p is considered undefined at stage g iff the expression $p\,@\,g$ cannot be reduced using $E \cup A$ to an expression not containing the operator @.

  This allows the definition of properties to be based on the values of others, in such a way that undefinedness is passed on.

- We require each element of the tuple defining $\mathcal{R}$ to be finite—which does not prevent any sort to have an infinite universe, of course. This corresponds to the requirement that we made for atEgTrStr in Section 3.2.1, that the sets of individual and parametric properties be finite.

The main point where we depart from the standard definition is that we label rules with terms instead of atomic identifiers. The choice of the label and, in particular, of the variables in the label term is a concern for the specifier, with semantic consequences. For a schematic illustration, consider the rule $s(x) \to s'(y)$, where x and y are the only variables in s and s', resp. We depict below the semantics for five possible rule labels. In each label, we show explicitly all variables involved. We assume that the variables x, y, and z range over $\{0, 1\}$.



Focusing only on states, abstracting transitions away, the result is in all cases the same: from any of $s(0)$ or $s(1)$, in one step, any of $s'(0)$ or $s'(1)$ can be reached. However, the importance of transitions in modelling systems is one of the ideas put forward in this work. It is specially discussed in Sections 2.1 and 2.2.

Fresh variables in the label term are acceptable the same as in the destination state term. The motivational example in Section 1.1 includes a rule with fresh variables in its label term. Another example was briefly discussed in Section 2.2: when the same operation in the memory can be asked by different processors,

the identity of the processor to which the memory is serving can naturally be an attribute of the transition, and not of any state of the memory.

Also, consider this new example. States are now pairs of natural numbers and we need a rule able to take any pair to another with the same difference. That is, we need something to the effect of

$$(a, b) \to (c, d) \;\text{ if }\; a - b = c - d.$$

The defining characteristic of each such transition is the difference, so it makes sense to choose it as parameter for the rule label:

$$l(n) : (a, b) \to (c, c + n) \;\text{ if }\; n := a - b,$$

where l is a constructor of `Trans`. But still better may be

$$l(a - b) : (a, b) \to (c, c + a - b),$$

with the same meaning and no need for fresh variables in the label.

If a rule like, say, "$f(x) : s \to s'$ if C" is present in our system, then a declaration for the operator f is needed, showing that it takes an argument and returns a `Trans`. Then, $f(a)$ is a value of sort `Trans`, and it is valid, for instance, to include in Ω a constant b of sort `Trans` and in E an equation stating that $b = f(a)$. All this is standard for other sorts, and we need it to be allowed for `Trans` as well.

A rule $t : s \to s'$ can be applied to a given state term s_0 if there is a substitution σ for the variables in the rule such that $s_0 = \sigma(s)$. Then, the transition term is $\sigma(t)$, and the destination state term is $\sigma(s')$. For conditional rules, σ must provide also substitutions for fresh variables in the conditions, like in standard rewrite systems.

3.3.2 Executability of rules in atEgRwSys

The theoretical definition of rewrite system allows for some features that can make the system non-executable. This section discusses how to ensure executability. In short: we need that fresh variables in the label term and in the destination state term are also in the conditions of the rule, and in a way in which they can be instantiated by matching (maybe in a non-deterministic way), that is, on the left-hand side of matching conditions or in the right-hand side of rewrite conditions.

For standard rewrite systems (that is, with atomic rule labels), these requirements are thoroughly described in [**?**, Sect. 4.6 & 6.3] under the name of *admissibility conditions*. For the sake of completeness, we rephrase them here. For a standard-setting rewrite rule $l : s \to s'$ if $\bigwedge_{j=1}^{m} C_j$:

1. all variables in s' must also occur in s or in some condition, that is, $\mathrm{vars}(s') \subseteq \mathrm{vars}(s) \cup \bigcup_{j=1}^{m} \mathrm{vars}(C_j)$;
2. all variables occurring in an equational condition $u_i = u_i'$, or in a membership condition $u_i : S$, or in the right-hand side of a matching condition $u_i := u_i'$, or in the left-hand side of a rewrite condition $u_i \to u_i'$, must also occur in s or in previous conditions, that is, in $\mathrm{vars}(s) \cup \bigcup_{j=1}^{i-1} \mathrm{vars}(C_j)$;

3. the left-hand sides of matching conditions and the right-hand sides of rewrite conditions must be *patterns*. A pattern, roughly speaking, is a term that can be equationally reduced only at single variables. The aim is that, in a matching condition like $f(x) := g(s)$, the term $f(x)$ cannot be equationally reduced to something different, because that would modify the possibilities of matching. For the formal definition, see [**?**, p. 66].

As well as for rules, all this must also hold for equations and membership axioms, except that these cannot include rewrite conditions. Equations and membership axioms in atEgRwSys are the same as in the standard setting, so they are not further discussed here.

Our needs are a little more complex than in the standard setting, because we have to take care of variables in rule labels. The label term t can contain variables not in s, and also s' can contain fresh variables, and all of them need to be instantiated when first found. Also, we need to add a new requirement: that rule labels always instantiate to transition terms satisfying the “from any to any” condition (as discussed in Section 2.2). Intuitively speaking, once the transition t has fired, the system forgets which particular state s it comes from and, thus, the destination state s' cannot depend on s. In a conditional rule, some of the conditions are on the firing of the transition, on it be allowed to start executing, and they may involve variables from s and t; others are conditions to finish the execution and to reach a given destination state, and may involve variables from t and s'. Conditions that relate variables in s and in s' are not acceptable unless they are also in t.

To achieve all that, the admissibility conditions for atEgRwSys are the same as for the standard setting, as rephrased above, except the first item, that needs to be more elaborate, as follows. The idea is that a variable x whose value is drawn from s by a matching condition like, say, $f(x) := g(s)$ can be considered, in a way, a variable *belonging* to s. Thus, in addition to the sets of variables occurring literally in s and in t, that we represent respectively as $\mathrm{vars}(s)$ and $\mathrm{vars}(t)$, we define now *extended* sets of variables of s and of t in a rule $t : s \to s'$ if $\bigwedge_{j=1}^{m} C_j$, that we represent as $\mathrm{vars}^*(s)$ and $\mathrm{vars}^*(t)$. These extended sets are computed incrementally, going through the list of conditions in order, from left to right. At the start, $\mathrm{vars}^*(s) := \mathrm{vars}(s)$ and $\mathrm{vars}^*(t) := \mathrm{vars}(t)$. Then, for each matching condition $u_i := u'_i$, if $\mathrm{vars}(u'_i) \subseteq \mathrm{vars}^*(s)$, then we add $\mathrm{vars}(u_i)$ to $\mathrm{vars}^*(s)$. The same for t. And correspondingly for rewrite conditions. Equational and membership conditions do not modify the extended sets. Now, the first admissibility condition for atEgRwSys is:

1. $\mathrm{vars}(t) \subseteq \mathrm{vars}^*(s)$ and $\mathrm{vars}(s') \subseteq \mathrm{vars}^*(t)$.

Conditions directly relating variables in s and in s', like $f(x) = f'(x')$, with $x \in \mathrm{vars}(s)$ and $x' \in \mathrm{vars}(s')$, are still acceptable as long as some earlier condition puts x or x' also in $\mathrm{vars}^*(t)$, something like $g(x) := g'(t)$.

The following two theorems show that the admissibility conditions for atEgRwSys are nice.

Theorem 3. *1. The standard-setting rule*

$$l : s \to s' \ \text{ if } \ \bigwedge_{j=1}^{m} C_j$$

has the same transition semantics as the atEgRwSys *rule*

$$l(\overline{v}) : s \to s' \text{ if } \bigwedge_{j=1}^{m} C_j,$$

turning l from an atomic label into an operator, and where $\overline{v}$ is the tuple with all the variables in s, in s', and in all the conditions C_j. The term $l(\overline{v})$ is called a proof term *in standard rewriting logic.*

2. *For that case, the admissibility conditions for rules in* atEgRwSys *are equivalent to the standard ones.*

Theorem 4. *The "from any to any" condition for rule labels is implied by the admissibility conditions for* atEgRwSys*.*

Let's discuss some simple consequences. Our framework needs to allow a transition to start executing even when there is no guarantee that some destination state can be reached, because the behaviour of some other system synced to this one may depend on it. Thus, even a rule like

$$l : x \to y \text{ if } y := 0 \wedge y = 1,$$

with l a constant, that obviously cannot reach a destination state, must be allowed to start executing for any value of x. In cases when no destination state can be reached, the system gets stuck in a transition. This must be considered as good or as bad as getting stuck in a state.

Conditions involving variables shared by s, t, and s' must be considered conditions both on the firing and on the finishing of a transition. Consider an extreme example with no variables at all in the condition: $l : a \to b$ if $0 = 1$. Imagine that this trivially false condition is considered only a condition on the firing of the transition, that is, to prevent the rule from starting execution. Imagine also that another rule is included in the same system: $l : a \to c$. It is the same constant label, so the two rules are defining the same transition: a transition that goes from a to either b (if conditions allowed) or c. The latter rule allows to start executing transition l and the former rule would allow it to finish on b, if we considered the condition only on the firing.

Another consequence of our admissibility conditions is that a good-looking rule like $l : x \to x + 1$, where l is a constant, is not admissible, because x occurs in the right-hand side term but not in the label term. This reflects the fact that the rule does not satisfy the "from any to any" condition as stated in Section 2.2. The transition l would take 1 to 2 and 5 to 6 and, being the same transition, should be able to take also 1 to 6 and 5 to 2. The rule $l(x) : x \to x + 1$ is admissible.

In addition to the admissibility conditions, and indeed previous to them, other requirements represent basic needs for a rewrite system to even be meaningful. The problem is that, for arbitrary sets of equations and rules, the rewriting relation between the terms of a system (that is, whether a term s can be rewritten to s' by means of a transition with term t) is undecidable. In [**?**], for instance, conditions are stated on how to make that relation effectively decidable. A system that satisfies those conditions is called a *computable system*. The three conditions, stated here in a rather simplistic way, are:

- equality modulo a set A of equational axioms (like commutativity, associativity...) is decidable;
- equations must be ground Church-Rosser and ground terminating (modulo A);
- rules must be ground coherent wrt the equations (modulo A).

These conditions are easy to meet. Usually, the rewrite systems a sensible programmer would code are computable. Only the third condition concerns rules and, thus, only it needs to be adapted to our setting.

In the standard setting, coherence means that if a rewrite is possible from a state term s to another s', then from any term in the equational class $[s]_{E \cup A}$ a rewrite is possible to a term in the equational class $[s']_{E \cup A}$. This allows an execution engine to work by, first, reducing the current term to its normal form with respect to the equations E modulo A and, then, rewriting from the normal form. See [**?**, Sect. 6.3] for complete explanations.

Once again, we need to take care of our complex rule labels. Transition terms can be reduced by equations, in the same way as state terms can. A policy like the one described in the previous paragraph for state terms seems appropriate: as soon as a transition is fired, its term is computed and immediately reduced to its normal form. Thus, we need coherence in the two phases:

- If a transition with term t can be fired from a state with term s, then from any state term in $[s]_{E \cup A}$ a transition can be fired whose term is in $[t]_{E \cup A}$.
- If a transition with term t can reach a state with term s', then from any transition term in $[t]_{E \cup A}$ a state can be reached whose term is in $[s']_{E \cup A}$.

These two complementary conditions are required for a system in atEgRwSys to be considered *computable*.

3.3.3 EgRwSys

The class atEgRwSys is not closed under the synchronous composition operation, because combinations arise that are not properly states nor transitions. That is why we need the more general EgRwSys. To a large extent, what we are defining and discussing in this section and the next ones about rewrite systems is a translation of what we have already defined and discussed about transition structures.

The class EgRwSys is the smallest one that includes atEgRwSys and is closed under synchronous composition. Thus, an element of EgRwSys either is in atEgRwSys or has the form $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z}$, with $n \geq 1$ and with each $\mathcal{R}_i$ being in EgRwSys itself. Each $\mathcal{R}_i$ may be, but needs not be, in atEgRwSys.

The parameter Y, the syncing criteria, consists of pairs of properties (or of parametric families of properties), exactly as in Section 3.2.2. The stages of $\mathcal{R}$ are written as $g = \langle g_1, \dots, g_n \rangle$, where each g_i is a stage of $\mathcal{R}_i$. As above, the component stages must be compatible wrt Y for g to be considered a stage of $\mathcal{R}$.

The parameter Z, the specification of the properties of the resulting system, is an example of a simple idea with an involved formalization. The idea is that Z declares the properties for $\mathcal{R}$ (so that it can be used as a component in turn)

and defines them solely based on the properties of the components $\mathcal{R}_1, \ldots, \mathcal{R}_n$. For an atomic system, the definition of the properties is based on the internals of the specification of the system; in a composed system, however, the definition of the properties must not care about the internals of each component, but only about their properties. Formally, however, also the declarations of stage and property sorts are needed. We call *interface* of a system to all the declarations (but no definitions) other systems may need to know in order to sync with it. It is formalized next.

We want to allow complex definitions in Z, including, for example, auxiliary functions and sorts. Thus, we choose to formalize Z as a membership equational system $Z = (\Sigma', \leq', \Omega', E' \cup A', M')$. This system has to include the interfaces from the component systems and create a new interface for $\mathcal{R}$.

Thus, the interface of a system in EgRwSys (even if in atEgRwSys) consists of:

- the declaration of the sort `Stage`;
- the declarations of the sort `Ppty` and all its subsorts;
- the declarations of all the properties (that is, operators whose resulting sort is a subsort of `Ppty`);
- for each property, the declaration of its result sort and its constructors (for instance, the sort of natural numbers if using some `NatPpty` property);
- for each parametric property, the declaration of its argument sorts and their constructors;
- for each property subsort, the declaration of the @ operator.

For atomic systems, all those elements are included in its specification (according to the definition of atEgRwSys). Non-atomic systems, however are not rewrite systems in the standard sense. So $\mathcal{R}$ has neither sorts nor operators by itself. The interface is part of Z, not $\mathcal{R}$. By a slight abuse of terminology, however, we consider it the interface of $\mathcal{R}$. The resulting properties, in particular, are to be considered $\mathcal{R}$'s properties.

In spite of all the complexities above, the definition of new properties is usually represented by simple equations. For instance, assuming p is a Boolean property in $\mathcal{R}$ and the p_i are integer properties in the respective $\mathcal{R}_i$, the following can be in $\mathcal{R}.Z$:

$$p \,@\, \langle g_1, g_2 \rangle := (p_1 \,@\, g_1 - p_2 \,@\, g_2 = 1).$$

3.3.4 RwSys

Next, we define a new kind of rewrite systems, that we call just *rewrite systems* or, to avoid ambiguity, *plain rewrite systems*; we denote their class by RwSys. The class RwSys is closed under the appropriate notion of synchronous composition. While an element of EgRwSys is a set of rewrite systems, only the basic ones being atomic, all elements of RwSys are, in some sense, atomic. We will show a translation from atEgRwSys and EgRwSys to RwSys, that we call *split*, that preserves all the information in the original system. The importance of the split boils down to the fact that the resulting non-egalitarian systems in

RwSys are standard, or nearly so, and we can use existing tools on them. This corresponds to the split translation already defined for transition structures.

Formally, an element of RwSys, $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R)$, is a standard rewrite system as first defined in [?], to which we require that:

- it includes a sort named `State`,
- it is topmost, with `State` being its top sort,
- its rules have no labels.

The absence of labels has the effect of making transitions identifiable just by their origin and destination states, thus merging all edges between two given states into one. We can add to the definition all the machinery about properties, if we intend to compose them.

The concepts of coherence, admissibility, and others, can be used for RwSys as originally defined for standard rewrite systems.

We can define a synchronous composition operation for RwSys that reflects the one for EgRwSys. Namely, given $\mathcal{R}_i = (\Sigma_i, \leq_i, \Omega_i, E_i \cup A_i, M_i, R_i) \in$ RwSys, we define its composition $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z} \in$ RwSys. The parameters Y and Z are like the ones for EgRwSys.

The definition of $\mathcal{R}$ happens to be formally almost identical to the one for defining the split of a general EgRwSys in Section 3.3.6, the only change needed being the replacement of all occurrences of split($\mathcal{R}$) by just $\mathcal{R}$ in the text of the definition. Thus, we do not repeat it here. This, by the way, has the effect of turning trivial the proof of Theorem 8.

The same remark from the end of Section 3.2.3 applies here. Namely, although the operators "$\|$" for EgRwSys and for RwSys reflect each other (see Theorem 8), they are of a different nature: it is a constructor in EgRwSys, but an evaluable operator in RwSys. In other words, $(\mathcal{R}_1 \| \mathcal{R}_2)_{Y,Z}$ is an irreducible expression in EgRwSys, but can be reduced in RwSys.

3.3.5 The split from atEgRwSys to RwSys

Given $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R) \in$ atEgRwSys, we define next split($\mathcal{R}$) $= (\Sigma', \leq, \Omega, E \cup A, M, R') \in$ RwSys. This system needs a top sort split($\mathcal{R}$).`State`, according to the definition of RwSys. This, however, is not the sort $\mathcal{R}$.`State` from $\mathcal{R}$, but rather corresponds to the top sort in $\mathcal{R}$, that is, to $\mathcal{R}$.`Stage`. (Terminology may be a little confusing here.) As a consequence, Σ' is identical to Σ except that it includes a new sort split($\mathcal{R}$).`State` as a synonym for $\mathcal{R}$.`Stage`. Apart from this, the only other change from $\mathcal{R}$ to split($\mathcal{R}$) is in the rules and their labels. The new set of rules R' is given by the following. For each conditional rule

$$t : s \to s' \text{ if } C$$

in R, we include in R' the two unlabelled conditional rules

$$s \to t \text{ if } C'$$
$$t \to s' \text{ if } C''$$

where the ordered list of conditions $C = \bigwedge_{j=1}^{m} C_j$ has been divided into two ordered sublists, C' and C'', not necessarily disjoint, in this way:

- a condition C_j in C is put into C' iff $\mathrm{vars}(C_j) \subseteq \mathrm{vars}^*(s)$.
- a condition C_j in C is put into C'' iff $\mathrm{vars}(C_j) \subseteq \mathrm{vars}^*(t)$.

Here, $\mathrm{vars}^*(s)$ and $\mathrm{vars}^*(t)$ are the extended sets of variables defined in Section 3.3.2. When $\mathrm{vars}(C_j) \subseteq \mathrm{vars}^*(s) \cap \mathrm{vars}^*(t)$, the condition C_j is put into both C' and C'' (for reasons explained in Section 3.3.2). The conditions in C' are on the firing of the transition; the ones in C'' are to finish executing. The two resulting rules are also called the *split* of the original rule.

As one more piece of notation, we write $C \simeq C' \wedge C''$ meaning that the ordered list of conditions C is the result of shuffling and simplifying the ordered lists C' and C''. That is: C' and C'' are ordered sublists of C, keeping the same ordering, and all conditions in C are either in C' or in C'' or in both, but not necessarily all conditions in C' come before the ones in C'' when seen in C.

The following two theorems show that the split operation behaves nicely.

Theorem 5. *A rewrite system $\mathcal{R}$ in* atEgRwSys *satisfies the admissibility conditions for* atEgRwSys *iff* $\mathrm{split}(\mathcal{R})$ *satisfies the admissibility conditions for* RwSys.

Theorem 6. *Given $\mathcal{R}$ in* atEgRwSys*, its set of rules is coherent with respect to its set of equations (according to the definition at the end of Section 3.3.2) iff* $\mathrm{split}(\mathcal{R})$ *is coherent with respect to the same equations (according to the standard definition).*

3.3.6 The split from EgRwSys to RwSys

Given $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z} \in$ EgRwSys, its split is $\mathrm{split}(\mathcal{R}) = (\Sigma, \leq, \Omega, E \cup A, M, R) \in$ RwSys, defined next. Remember that Z is a membership equational system $Z = (\Sigma', \leq', \Omega', E' \cup A', M')$.

The formal definition of the split is long, but its essence is simple. It defines a state of the split system as a stage in the original (that is, a tuple of the components' stages, provided they are compatible), and creates rules for any possible concurrent execution of rules of the components. Many points here are recursive, with the definitions for atEgRwSys providing base cases. Now, this is the definition of $\mathrm{split}(\mathcal{R})$:

- Σ is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).\Sigma$, and
  - Σ' from Z (including, in particular, the sort $\mathrm{split}(\mathcal{R})$.`State`);
- $\leq$ is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).\leq$, and
  - $\leq'$ from Z;
- Ω is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).\Omega$,
  - Ω' from Z (including, in particular, declarations for the new properties),

- the tuple operator to build states: $\langle _ \rangle : \Pi_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).\mathtt{State} \to \mathrm{split}(\mathcal{R}).\mathtt{State}$,
- the @ operator for properties and states of the split system, and
- a constant `init` of sort $\mathrm{split}(\mathcal{R}).\mathtt{State}$;

- E is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).E$,
  - E' from Z, and
  - an equation $\mathrm{split}(\mathcal{R}).\mathtt{init} = \langle \mathrm{split}(\mathcal{R}_1).\mathtt{init}, \ldots, \mathrm{split}(\mathcal{R}_n).\mathtt{init} \rangle$;
- A is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).A$, and
  - A' from Z;
- M is composed by:
  - $\bigcup_{i=1}^{n} \mathrm{split}(\mathcal{R}_i).M$,
  - M' from Z, and
  - a single new conditional membership axiom stating that a tuple $\langle s_1, \ldots, s_n \rangle \in \Pi_i \mathrm{split}(\mathcal{R}_i).\mathtt{State}$ is in $\mathrm{split}(\mathcal{R}).\mathtt{State}$ if it satisfies all the syncing criteria in Y, that is, for each $\mathcal{R}_j.p_j = \mathcal{R}_k.p_k \in Y$, there is a condition $p_j \,@\, s_j = p_k \,@\, s_k$ in the membership axiom; for syncing criteria involving parametric properties, we have assumed that only a finite set of them are defined at each stage, so they can be made into a finite list of conditions;
- R is obtained in this way: for each non-empty subset $N \subseteq \{1, \ldots, n\}$ and for each possible way to choose a rule from each $\mathrm{split}(\mathcal{R}_i)$ with $i \in N$, say "$s_i \to s'_i$ if C_i" from $\mathrm{split}(\mathcal{R}_i).R$, we add to $\mathrm{split}(\mathcal{R}).R$ the rule

  $$s \to s' \text{ if } \bigwedge_{i \in N} C_i \wedge s : \mathtt{State} \wedge s' : \mathtt{State},$$

  denoting $s = \langle s_1, \ldots, s_n \rangle$ and $s' = \langle s'_1, \ldots, s'_n \rangle$, and where $s_i = s'_i$ for $i \notin N$. The two membership conditions are needed so that only actual states are rewritten, and then only to other states; that is, tuples of states that are not compatible wrt Y are not considered.

3.3.7 Deductions in EgRwSys and in RwSys

Given a rewrite system of any of the three kinds we are considering, a *deduction* in it is a sequence of stages (states, for RwSys) whose first element is `init` and, then, each element can be deduced from the previous one in a single rewriting step using the system's rules. (In this context, we could appropriately use the name *rewrite theory* instead of *rewrite system*.)

For each rewrite system $\mathcal{R}$, a set of deductions, denoted as $\mathrm{deds}(\mathcal{R})$, can be associated to it. We represent as Deductions the class whose elements are sets of deductions:

$$\mathrm{deds} : \mathsf{EgRwSys} \cup \mathsf{RwSys} \longrightarrow \mathsf{Deductions}.$$

We define below when a deduction is *valid* in a system. Validity, as we are about to define it, does not take into account the initial stage (or state) of the system. Out of all the sequences valid in $\mathcal{R}$, the set deds($\mathcal{R}$) includes only the ones whose first element is `init`.

Deductions in atEgRwSys. Let $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R)$ be an element of atEgRwSys. The rules below formalize the concept of logical deduction valid in $\mathcal{R}$. Given our choices, a sequence of transitions alone would be as incomplete as a sequence of states alone; a complete deduction needs both. Also, we allow a deduction to start and end either with a state or with a transition. We write $\mathcal{R} \vdash s, t, s'$ to denote that such sequence is a valid deduction in the rewrite system $\mathcal{R}$. Infinite sequences are also valid as deductions.

Variables named α, β, γ range over sequences of stages (even empty ones), g ranges over stages, s over states, and t over transitions. These are the rules for validity of a deduction in a system, that is, for $\mathcal{R} \vdash \alpha$:

rewriting This reflects the ways a rule is used.

$$1) \quad \frac{\begin{array}{c}\text{“}t : s \to s' \text{ if } C\text{” is a rule in } R \\ C \simeq C' \wedge C'' \text{ (as in Section 3.3.5)} \\ \sigma \text{ is a substitution} \\ \text{all conditions in } \sigma(C') \text{ are satisfied in } \mathcal{R}\end{array}}{\mathcal{R} \vdash \sigma(s), \sigma(t)}$$

$$2) \quad \frac{\begin{array}{c}\text{“}t : s \to s' \text{ if } C\text{” is a rule in } R \\ C \simeq C' \wedge C'' \text{ (as in Section 3.3.5)} \\ \sigma \text{ is a substitution} \\ \text{all conditions in } \sigma(C'') \text{ are satisfied in } \mathcal{R}\end{array}}{\mathcal{R} \vdash \sigma(t), \sigma(s')}$$

The substitution σ assigns a ground term over $\mathcal{R}$'s signature to each variable in the rule. For equational and matching conditions, satisfaction means that the equalities can be proved from $E \cup A$; for membership conditions, it means that they can be deduced from M; for rewrite conditions like $u \to u'$, it means that a deduction is possible in $\mathcal{R}$ of the form $\sigma(u), \alpha, \sigma(u')$ according to the same deduction rules we are describing. If all the conditions in C are satisfied, from the two conclusions, $\mathcal{R} \vdash \sigma(s), \sigma(t)$ and $\mathcal{R} \vdash \sigma(t), \sigma(s')$, the composed one, $\mathcal{R} \vdash \sigma(s), \sigma(t), \sigma(s')$, can be obtained through the rule “concatenation” below.

equality This represents the fact that rewriting with rules is performed on equational classes of terms. If the deduction is valid with a member of the class, so is it with any other member.

$$\frac{E \cup A \vdash g = g' \qquad \mathcal{R} \vdash \alpha, g, \beta}{\mathcal{R} \vdash \alpha, g', \beta}$$

concatenation Two valid deductions can be concatenated, provided one ends where the other begins.

$$\frac{\mathcal{R} \vdash \alpha, g \qquad \mathcal{R} \vdash g, \beta}{\mathcal{R} \vdash \alpha, g, \beta}$$

In [?] and [?] some axioms are included so as to identify deductions that are "essentially the same." We argue that we do not need them—at least not at this point. There are two groups of such axioms to discuss. One group formalizes the fact that we are interested in working in equational classes. That is, that α, g, β and α, g', β represent the same deduction if $g = g'$ can be proved in $E \cup A$ (thus, the rule *equality* does not really produce *new* deductions). This, however, is a semantic equivalence, and we formalize it when we define the semantics of a deduction as a run formed by equivalence classes (see Definition 4 in Section 3.4). This parallels how the semantics of a rewrite system uses equivalence classes of terms as states for the resulting transition structure (see Definitions 1, 2, and 3 in Section 3.4).

The second group of axioms formalizes the fact that two independent rewrites in disjoint subterms can be performed in any order, or simultaneously, with the same final effect. However, this is not convenient for the modelling of systems, and this is one of our tenets in this work. Two different ways to go from some state to some other may be equivalent from a proof-theoretic point of view, but they represent different evolutions of the system, and must be considered different. A model checker, for example, must analyse both of them.

Deductions in EgRwSys. Let $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z}$ be an element of EgRwSys. A deduction is a sequence of (compatible) stages in $\mathcal{R}$ starting with the initial one. A deduction in $\mathcal{R}$ has the shape

$$\langle g_1^0, \ldots, g_n^0 \rangle,\ \langle g_1', \ldots, g_n' \rangle,\ \langle g_1'', \ldots, g_n'' \rangle, \ldots$$

where from an element of the sequence to the next some of the component stages may remain unchanged, but not all of them (for instance, g_i'' can be equal to g_i' for a few i's). Such a deduction is valid in $\mathcal{R}$ iff each individual sequence, after removing repetitions, is a valid deduction in the component system. That is, if $g_i^0, g_i^1, \ldots$ is the result of removing duplicate consecutive elements in $g_i^0, g_i', g_i'', \ldots$, then we have

$$\mathcal{R}_i \vdash g_i^0, g_i^1, \ldots$$

for all $i = 1, \ldots, n$.

Deductions in RwSys. Let $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R)$ be an element of RwSys. Deductions in $\mathcal{R}$ are defined by the same rules above, except that we replace the two called *rewriting* by this one:

rewriting This reflects the way a rule is used.

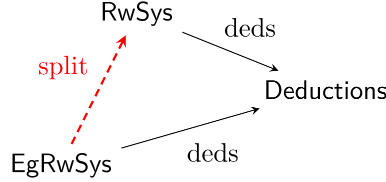
$$\frac{\begin{array}{c}\text{“}g \to g' \text{ if } C\text{” is a rule in } R\\ \sigma \text{ is a substitution}\\ \text{all conditions in } \sigma(C) \text{ are satisfied in } \mathcal{R}\end{array}}{\mathcal{R} \vdash \sigma(g), \sigma(g')}$$

In RwSys there is only one way to go from a state to an adjacent one. Thus, the mere sequence of states in a deduction is unambiguous. We keep in this rule variables named g and g', for homogeneity with the other rules not repeated here, but in RwSys they represent states.

3.3.8 Commutativity theorems

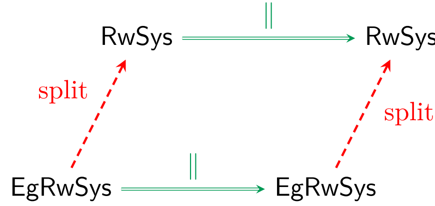
The two polygons on the upper floor of the figure in Section 3.1 are commutative diagrams. Let's state this as theorems.

Theorem 7. *The following diagram is commutative:*



That is, given an element of EgRwSys and its split, their sets of deductions are the same. Thanks to this theorem, the study of deductions in an egalitarian rewrite system can be equivalently performed in the more standard setting of its split system.

Theorem 8. *The following diagram is commutative:*



This allows, for example, to perform verification on $\mathrm{split}(\mathcal{R}_1)$ and $\mathrm{split}(\mathcal{R}_2)$ (at the upper left-hand corner) and draw conclusions valid for $(\mathcal{R}_1 \| \mathcal{R}_2)_{Y,Z}$ (at the lower right-hand corner).

3.4 Semantics

3.4.1 Definitions

Definition 1. *Given* $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R) \in \mathsf{atEgRwSys}$, *we define* $\mathrm{sem}(\mathcal{R}) = (S, T, \to, P, g^0) \in \mathsf{atEgTrStr}$ *by:*

- *S is the set of ground terms of sort `State` in $\mathcal{R}$ modulo $E \cup A$, that is, $T_{\Omega/E\cup A,\mathtt{State}}$;*
- *T is the set of ground terms of sort `Trans` in $\mathcal{R}$ modulo $E \cup A$, that is, $T_{\Omega/E\cup A,\mathtt{Trans}}$;*
- *$\to$ is the adjacency relation defined by R, namely, for each $[a]_{E\cup A}$ of sort `State`, if there is a rule "$t : s \to s'$ if C" in $\mathcal{R}$, with $C \simeq C' \wedge C''$ as in Section 3.3.5, and a substitution σ such that $\sigma(s) \in [a]_{E\cup A}$ and all the conditions $\sigma(C')$ (resp., $\sigma(C'')$) hold, then the adjacency relation includes $[a]_{E\cup A} \to [\sigma(t)]_{E\cup A}$ (resp., $[\sigma(t)]_{E\cup A} \to [\sigma(s')]_{E\cup A}$);*
- *for each ground term (modulo $E \cup A$) of sort `Ppty` in $\mathcal{R}$ there is a property in P whose values are obtained through application of the operator @, being undefined if such an application cannot be reduced using $E \cup A$ to a term not containing @;*

- $g^0 = [\texttt{init}]_{E \cup A}$.

Definition 2. *Given* $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z} \in \mathsf{EgRwSys}$*, we define* $\mathrm{sem}(\mathcal{R})$ *as*

$$(\mathrm{sem}(\mathcal{R}_1) \| \cdots \| \mathrm{sem}(\mathcal{R}_n))_{Y,\mathrm{sem}(Z)} \in \mathsf{EgTrStr}.$$

The parameter Y needs not be modified from $\mathcal{R}$ to its semantics. On the contrary, the parameter Z needs to be adapted in a way that may well be called a semantic transformation. Thus, for $Z = (\Sigma', \leq', \Omega', E' \cup A', M')$, we define $\mathrm{sem}(Z)$ as the set of property definitions (that is, function definitions) that result from the usual term-rewriting semantics for equational systems.

Definition 3. *Given* $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R) \in \mathsf{RwSys}$*, we define* $\mathrm{sem}(\mathcal{R}) = (S, \rightarrow, P, s^0) \in \mathsf{TrStr}$ *by:*

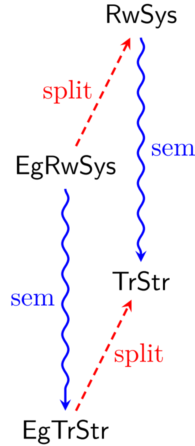
- *S is the set of ground terms of sort* `State` *in $\mathcal{R}$, that is,* $T_{\Omega/E \cup A, \texttt{State}}$*;*
- *$\rightarrow$ is the rewriting relation defined by R, namely, for each $[a]_{E \cup A}$ of sort* `State`*, if there is a rule "$s \rightarrow s'$* if *C" in $\mathcal{R}$ and a substitution σ such that $\sigma(s) \in [a]_{E \cup A}$ and all the conditions $\sigma(C)$ hold, then the adjacency relation includes $[a]_{E \cup A} \rightarrow [\sigma(s')]_{E \cup A}$;*
- *for each ground term (modulo $E \cup A$) of sort* `Ppty` *in $\mathcal{R}$ there is a property in P whose values are obtained through application of the operator @, being undefined if such an application cannot be reduced using $E \cup A$ to a term not containing @;*
- $s^0 = [\texttt{init}]_{E \cup A}$.

Definition 4. *Given* $\mathcal{R} \in \mathsf{EgRwSys} \cup \mathsf{RwSys}$ *and a deduction in it,* $\mathcal{R} \vdash g^0, g^1, \ldots,$ *the semantics for such a deduction is the run* $[g^0]_{E \cup A}, [g^1]_{E \cup A}, \ldots$ *in* $\mathrm{sem}(\mathcal{R})$.

3.4.2 Commutativity theorems

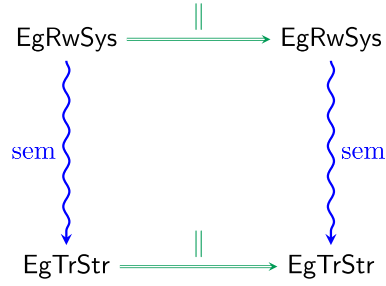
All the five different vertical faces in the figure in Section 3.1 are commutative diagrams. Let's state them as independent theorems.

Theorem 9. *The following diagram is commutative:*

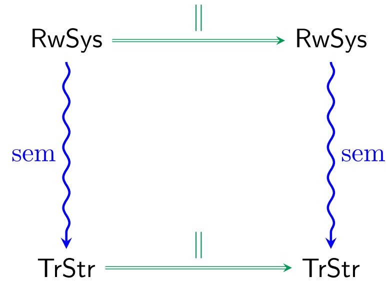


That is, the semantics for split systems faithfully represent the semantics for egalitarian ones.

Theorem 10. *The following diagram is commutative:*

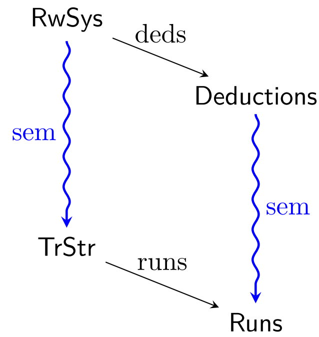


Theorem 11. *The following diagram is commutative:*

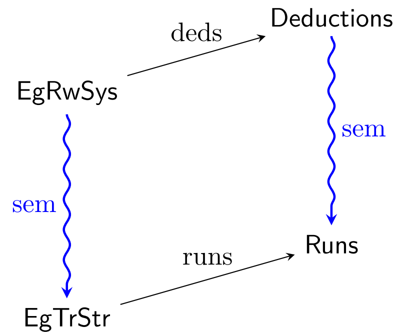


Thus, when drawing conclusions for a rewrite system based on its semantics, we can do it component-wise.

Theorem 12. *The following diagram is commutative:*



Theorem 13. *The following diagram is commutative:*



As a consequence of these two last theorems, model checking and theorem proving can be interchangeably used to draw conclusions on rewrite theories and transition structures.

4 Some simple but complete examples

We show now the specification of three example systems, small but completely developed. We use the language Maude, described in [?], with the addition of the syntactic constructs for synchronous composition that we intend to implement in the near future. The aim of these examples is to show the way to think and use our tools for specifying realistic systems.

There are a few declarations of sorts and operators that are used in all the examples. So as to avoid repeating them, we assume they are included in a common module that is implicitly and silently imported whenever needed. To begin with, we declare four sorts and some subsort relations among them:

⟨In the common module⟩:

```
sorts State Trans Stage .
subsorts State Trans < Stage .
sort Ppty .
```

Also, whenever we need a property subsort, we use it assuming it has been declared, together with its @ operator. For example, for a property whose values range over the natural numbers, we use the sort named `NatPpty` and assume the following has been imported:

⟨Also in the common module⟩:

```
sort NatPpty .
subsort NatPpty < Ppty .
op _@_ : NatPpty Stage -> Nat .
```

The keyword `op` in Maude introduces the declaration of an operator. In this case, `@` is declared infix, with the two underscores marking the places where arguments must be written. The declaration of the `init` constant is also in the common module:

⟨Also in the common module⟩:

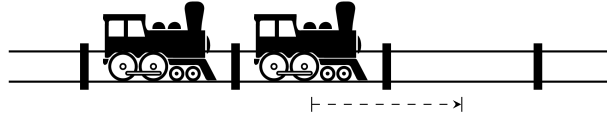
```
op init : -> Stage .
```

All this helps keeping our code clean and focused.

4.1 Two trains on a linear railway

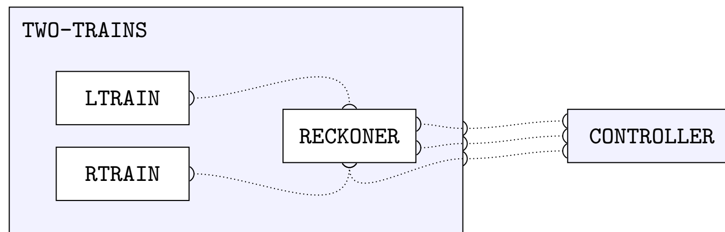
This example is inspired in one from [?]. Two trains move independently on the same linear railway divided in track sections. They both move rightwards, one

track section at a time. The train on the left is not allowed to reach the one on the right, so that, when the trains are in consecutive track sections, the right one has to move.



The two trains are modelled by identical modules. We leave the control of the movements to an external module, so that the model of each train needs not care about the existence of another train. Moreover, the behaviour of a train is likely to be the same no matter which track section it is sitting on, so we prefer that sections are not part of the model of each train, but of a different system that we call the *reckoner*.

We structure the complete system in two levels. First, the two trains, `LTRAIN` and `RTRAIN`, and the `RECKONER` are combined into a module called `TWO-TRAINS`; then, in the second level, this system is combined with the `CONTROLLER`. Graphically:



Each system is represented as a box. Bump-shaped connectors on the edges represent the properties each system provides. Dotted-line wires represent syncing. The reckoner, for example, needs to sync with the trains to know when each is moving. The controller sees the system `TWO-TRAINS` as a black box that provides three properties to sync on. Inside the system `TWO-TRAINS`, we see that its three properties are originally defined as properties of the reckoner.

The translation of that picture into Maude is:

```
⟨In module TWO-TRAINS⟩:
   pr LTRAIN || RTRAIN || RECKONER
   sync on LTRAIN.isMoving = RECKONER.isLMoving
        /\ RTRAIN.isMoving = RECKONER.isRMoving .
```

And, then:

```
mod CONTROLLED-TRAINS is
   pr TWO-TRAINS || CONTROLLER
   sync on CONTROLLER.areConsec = TWO-TRAINS.areConsec
        /\ CONTROLLER.doMove = TWO-TRAINS.isSomeMoving
        /\ CONTROLLER.doMoveR = TWO-TRAINS.isRMoving .
endm
```

All this is explained and completed in the rest of this section. The keyword `pr` is short for *protecting*, a way to import modules in Maude. Syncing criteria are specified after the keywords `sync on` (which is not standard Maude). They correspond to the set we called Y above, and to dotted lines in the picture. The definition of new properties, corresponding to the set we called Z, is discussed below.

The systems that model the two trains are identical, with a single state that we call `o` and a single transition that we call `m`:

⟨In modules `LTRAIN` and `RTRAIN`⟩:
```
op o : -> State .
op m : -> Trans .
rl [m] : o => o .
```

We assume both trains start static:

⟨Also in modules `LTRAIN` and `RTRAIN`⟩:
```
eq init = o .
```

The reckoner needs to know when the trains are moving. For that, each train module has a Boolean property `isMoving`, and the reckoner has two properties called `isLMoving` and `isRMoving`, so as to *receive* the information from each train. The Boolean property `isMoving` is declared and defined by two equations in this way:

⟨Also in modules `LTRAIN` and `RTRAIN`⟩:
```
op isMoving : -> BoolPpty .
eq isMoving @ m = true .
eq isMoving @ o = false .
```

It is defined as true at transitions and false at states. This can seem natural for a property called `isMoving`, but it is not necessary. If the specification were more refined, for instance so as to allow a train to *do something* while staying on the same track section, there would be a transition corresponding to that, and `isMoving` would be false at that transition.

As for the `RECKONER`, it keeps in its state the distance (number of track sections) between the trains—an integer number. It has three rules, corresponding to the movement of each train and of both at the same time. The simultaneous movement of the two trains is not considered in [?], the inspiration for this example, but it is welcome in our setting.

⟨In module `RECKONER`⟩:
```
subsort Int < State .
ops ml mr mlr : Int -> Trans .
var D : Int .
rl [ml(D)] : D => D - 1 .
rl [mr(D)] : D => D + 1 .
rl [mlr(D)] : D => D .
```

Transitions built with `ml` allow the distance to become zero, and even negative. This is just a reckoner, so it has nothing to say about it; the controller needs to take care.

Anything positive is acceptable as initial state. For instance:

⟨Also in module `RECKONER`⟩:
```
eq init = 1 .
```

The properties of the `RECKONER` are defined like this:

⟨Also in module `RECKONER`⟩:
```
ops isLMoving isRMoving : -> BoolPpty .
eq isLMoving @ ml(D) = true .
eq isLMoving @ mr(D) = false .
eq isLMoving @ mlr(D) = true .
eq isLMoving @ D = false .
eq isRMoving @ ml(D) = false .
eq isRMoving @ mr(D) = true .
eq isRMoving @ mlr(D) = true .
eq isRMoving @ D = false .
```

Pure states of `TWO-TRAINS` have the form `< o, o, D >`, for some integer value of `D`. The composed stage `< m, o, ml(2) >`, for instance, represents the left train moving while the right one is static. After this movement the distance between the trains will be 1.

Let's turn the focus to the controller. Its task is to ensure that, when the trains are in consecutive track sections, the right one moves, alone or otherwise. When the trains are not in consecutive sections, any one can move, or both. We propose syncing on three criteria: the first for the controller to be aware of consecutive-trains situations; the second so that the controller can command some train to move; the third so that the controller can command the right train to move. All these properties have not been defined yet. The final, composed system, where these properties are used, was presented above, and we repeat it here:

```
mod CONTROLLED-TRAINS is
   pr TWO-TRAINS || CONTROLLER
   sync on CONTROLLER.areConsec = TWO-TRAINS.areConsec
        /\ CONTROLLER.doMove = TWO-TRAINS.isSomeMoving
        /\ CONTROLLER.doMoveR = TWO-TRAINS.isRMoving .
endm
```

The word *command* used in the previous paragraph must be correctly understood. Syncing works in a symmetrical way, so any system can be seen as commanding the other. Intuitively, in this case, the controller will set its property `doMove` to true in some situations, and this will make mandatory for the trains to execute some action for which `isSomeMoving` is true. This is what we dub as the controller commanding the trains.

It often happens that controllers have complete meanings by themselves, and can be applied to different base systems. This is the case with the one we are describing now. In short, the controller's task is to detect when the base system is in a particular state and command that only a particular action be allowed next. The right train must move when the left one is next to it. Only deposits are allowed in a bank account whose balance is zero. Defensive moves are advised when our king is in trouble. Seen in this way, the names of the properties for the controller would be better chosen agnostic: instead of `areConsec`, use `isMarkedState`; instead of `doMoveR`, use `doMarkedAction`; instead of `doMove`, use `doAnyAction`. For the time being, however, we keep using train-related names.

Before going to the specification of the controller, let us define the three properties needed in `TWO-TRAINS`. All of them are Boolean and can be originally defined for the reckoner. Indeed, one of the properties, `isRMoving`, has already been defined (and used to sync with `RTRAIN`). The other two are defined like this:

```
⟨Also in module RECKONER⟩:
   ops areConsec isSomeMoving : -> BoolPpty .
   var S : State .
   var T : Trans .
   eq areConsec @ 1 = true .
   eq areConsec @ S = false [otherwise] .
   eq isSomeMoving @ S = false .
   eq isSomeMoving @ T = true .
```

The property `areConsec` needs not be defined while the trains are moving (and it would not make sense).

The three properties are now transferred from `RECKONER` to `TWO-TRAINS` with these statements:

⟨Also in module `TWO-TRAINS`⟩:

```
ops areConsec isSomeMoving isRMoving : -> BoolPpty .
var LT : LTRAIN.Stage .
var RT : RTRAIN.Stage .
var RE : RECKONER.Stage .
eq areConsec @ < LT, RT, RE > = RECKONER.areConsec @ RE .
eq isSomeMoving @ < LT, RT, RE > = RECKONER.isSomeMoving @ RE .
eq isRMoving @ < LT, RT, RE > = RECKONER.isRMoving @ RE .
--- or, equivalently:
--- eq isRMoving @ < LT, RT, RE > = RTRAIN.isMoving @ RT .
```

This represents the set we called Z during our theoretical discussions.

The controller has two states: one, `c`, to represent trains in consecutive track sections; the other, `n`, for the rest. A different transition is needed from each state, so that the one leaving from `c` commands (through syncing) a movement of the right train. No more refinement is needed.

⟨In module `CONTROLLER`⟩:

```
ops c n : -> State .
ops fromC fromN : -> Trans .
rl [fromC] : c => c .
rl [fromC] : c => n .
rl [fromN] : n => c .
rl [fromN] : n => n .
```

Note that the first two rules define the same transition, as do the last two.

Finally, this is the definition of the properties for the controller that we have already used to sync with `TWO-TRAINS`:

⟨Also in module `CONTROLLER`⟩:

```
ops areConsec doMove doMoveR : -> BoolPpty .
var S : State .
var T : Trans .
eq areConsec @ c = true .
eq areConsec @ n = false .
eq doMove @ S = false .
eq doMove @ T = true .
eq doMoveR @ fromC = true .
```

When moving away from a non-consecutive state, any movement of the trains is valid. Thus, `doMoveR` must be left undefined at `fromN`. Setting it to false would prevent the movement of the right train.

4.2 Insertion sort

This is an example proposed in [?] and [?]. Its aim is to implement a particular sorting algorithm (namely, insertion sort) as a strategy, or control, applied to a base system that just allows for the swapping of cells in an array. Contrary to the previous example, in this one we are designing and coding bottom-up. This is our implementation of arrays:

⟨In module `ARRAY`⟩:

```
subsort Set{Pair{Nat}} < State .
op swap : Pair{Nat} Pair{Nat} Set{Pair{Nat}} -> Trans .
vars I J V W : Nat .
var Rest : Set{Pair{Nat}} .
rl [swap((I, V), (J, W), Rest)] : (I, V) (J, W) Rest
                                => (I, W) (J, V) Rest .
```

Thus, an array is a set of cells, each cell represented as a pair of natural numbers: the first element of each pair is the index of the cell; the second is the contents. We use empty syntax for sets, that is, union is represented by mere juxtaposition of subsets and individual elements. We assume that the set of indices in each array starts at 1, ends at 100 (a completely arbitrary choice), and has no holes. Note that the rule `swap` allows swapping any two cells, even though they may be already sorted.

The strategy needs some properties as *handles* into this system to control it. The following two are quite natural:

```
⟨Also in module ARRAY⟩:
   sort Pair? .
   subsort Pair{Nat} < Pair? .
   op none : -> Pair? .
   op indSwapping : -> Pair?Ppty .
   op ar[_]<=ar[_] : Nat Nat -> BoolPpty .
```

The property `indSwapping` returns two numbers with the indices of the cells that are swapping their values. It only makes sense when a swapping is indeed taking place. We let it be `none` otherwise (that is, at a state), so as to easily differentiate swapping from non-swapping sections in the system. We use the sort `Pair?`, that includes pairs of natural numbers and the constant `none`. Using that property, a controller can command a swapping by setting a property (to be synced with `indSwapping`) to the appropriate pair of indices in the appropriate situations, and to `none` in the rest.

The second property, `ar[_]<=ar[_]`, is doubly parametric. It informs on how the contents of two cells compare, given their indices. The name we have chosen for the property is just fancy syntax. These are the definitions of these two properties:

```
⟨Also in module ARRAY⟩:
   var S : State .
   eq indSwapping @ swap((I, V), (J, W), Rest) = (I, J) .
   eq indSwapping @ S = none .
   eq ar[I]<=ar[J] @ (I, V) (J, W) Rest = V <= W .
```

We turn now to the implementation of our insertion-sort algorithm. As usual, it needs two pointers: one, called `C`, to the element that is `C`urrently going leftwards, looking for its place; the other, `N`, pointing to the `N`ext element to be processed. And as usual we need two movements of the pointers, represented by two rules: one, that we label as `left`, to move `C` one position to the left (that will be accompanied by a swap of cells in the array); the other, labelled as `next`, to proceed to the next element waiting.

Which of the rules, `left` or `next`, must be executed in the strategy at a given time depends on a property of the current state of the array: whether the element pointed by `C` has already reached its final position, that is, whether it is greater than the element to its left. This is a property of the array that the controller needs to know. Thus, in our example, the state will include, in addition to the two pointers, `C` and `N`, a Boolean, an answer to the question "Is the element pointed by `C` already in place?". The rule `left` is executed when that value is false, and the rule `next` when true:

```
⟨In module INSERTION⟩:
   op (_,_,_) : Nat Nat Bool -> State .
   ops left next : Nat Nat -> Trans .
   vars C N : Nat .
```

```
vars B B' : Bool .
crl [left(C, N)] : (C, N, false) => (sd(C, 1), N, B)
   if C > 1
   /\ B B' := true false .
crl [next(C, N)] : (C, N, true) => (N, s N, B)
   if N <= 100
   /\ B B' := true false .
```

The function `sd` is the symmetric difference for natural numbers, and `s` is the successor. The matching condition `B B' := true false` is a way to express that the value of `B` is non-deterministically chosen to be true or false. It will be determined when syncing with the base array system, as coded next.

The properties we need and the initial state are in the following piece of code. The property `doSwap` is meant to *tell* the base system which cells must be swapped at each step, according to the current value of the pointer `C`. The property `hasArrived` is rather for *receiving* information on whether the cell pointed by `C` is already in place.

```
⟨Also in module INSERTION⟩:
   var S : State .
   op doSwap : -> Pair?Ppty .
   op hasArrived : Nat -> BoolPpty .
   eq doSwap @ left(C, N) = (sd(C, 1), C) .
   eq doSwap @ next(C, N) = none .
   eq doSwap @ S = none .
   eq hasArrived(C) @ (C, N, B) = B .
   eq init = (1, 2, true)
```

That initial state, `(1, 2, true)`, makes the system execute `next(1, 2)`, leaving cell 1 untouched, and go to state `(2, 3, B)`, with the appropriate value of `B` according to the array.

Finally, the complete system is:

```
mod INSERTION-SORT is
   var C : Nat .
   pr ARRAY || INSERTION
   sync on INSERTION.doSwap = ARRAY.indSwapping
        /\ INSERTION.hasArrived(C) = ARRAY.ar[C - 1]<=ar[C] .
endm
```

Also this controller, `INSERTION`, can be seen as a particularization of a general means of finding appropriate pairs of elements in a collection: for each element in turn, find the immediately smaller one. Selection sort can be dubbed as another particular case: for each element in turn, find the immediately larger one. Also: for a collection of socks, find pairs of the same colour, by considering each one in turn and going through the ones already considered without success. Again, controllers have meaning by themselves.

4.3 Dekker's algorithm

Consider this minimalistic specification of a mutual exclusion algorithm:

```
⟨In module MUTEX⟩:
   op rem : -> State .
   op crit : Nat -> Trans .
   rl [crit(1)] : rem => rem .
   rl [crit(2)] : rem => rem .
```

There is a single state, `rem`, with two looping transitions, `crit(1)` and `crit(2)`. This control is to be applied to two base systems some of whose actions need

exclusivity. The two base systems must define properties to allow syncing, so that the critical section of the first base system executes at the same time as `crit(1)`, and respectively for the second system and `crit(2)`.

These are appropriate properties for `MUTEX`:

⟨Also in module `MUTEX`⟩:
```
op doIt : Nat -> BoolPpty .
var N : Nat .
var G : Stage .
eq doIt(N) @ crit(N) = true .
eq doIt(N) @ G = false [otherwise] .
```

This is only meaningful for `N` in $\{1, 2\}$. As initial state we choose:

⟨Also in module `MUTEX`⟩:
```
eq init = rem .
```

That means that the base systems must both start outside their critical sections.

In Section 1.1 we sketched a different `MUTEX` system, with two `rem` and two `crit` transitions. Any particular specification is equally valid as long as it provides two properties `doIt(1)` and `doIt(2)` that do not get true at the same time.

Next we show a one-rule adaptation of Dekker's algorithm (to guarantee absence of starvation in a set of processes). Usual presentations of Dekker's algorithm involve only two processes; we allow any given set of whatever number of processes. Usually mutual exclusion is also guaranteed by Dekker's algorithm; our implementation of it does not address the mutual exclusion problem, only absence of starvation. In the particular case of two processes, for instance, our algorithm would gladly allow both of them to do their thing at the same time. In case we need mutual exclusion, or resources are limited and do not allow for so many processes running in parallel, another controller would be needed to take care of it, like `MUTEX`. More on this below.

For our implementation, we identify processes with natural numbers from 1; therefore, sets of processes are sets of natural numbers. We use in the conditions some mathematical notation that we explain below:

⟨In module `DEKKER`⟩:
```
op (_,_,_,_) : Nat Set{Nat} Set{Nat} Set{Nat} -> State .
  --- an implementation of sets of natural numbers is assumed
op step : Nat Set{Nat} Set{Nat} Set{Nat}
              Set{Nat} Set{Nat} Set{Nat} -> Trans .
vars T T' : Nat .
vars W A O W' A' O' W2A A2O O2W : Set{Nat} .
crl [step(T, W, A, O, W2A, A2O, O2W)] : (T, W, A, O)
                                    => (T', W', A', O')
   if W2A ⊆T W
   /\ A2O ⊆ A
   /\ O2W ⊆ O
   /\ W2A ∪ A2O ∪ O2W ≠ ∅
   /\ W' := (W \ W2A) ∪ O2W
   /\ A' := (A \ A2O) ∪ W2A
   /\ O' := (O \ O2W) ∪ A2O
   /\ T' := nextTurn(T, W2A) .
```

The last three components of the state term—`W`, `A`, and `O`—are a partition of the set of processes: `W` is the set of processes that are `Waiting`; `A` is the set of processes that are being `Attended`; `O` is the set of processes that are busy in `Other` things. We require that, indeed, these three sets form a partition of the set of process identifiers in the initial state. Then, the rules guarantee that this property keeps holding.

The set W2A contains the processes that were waiting and begin to be attended after the current step. Similarly for A2O and O2W. They have to be non-deterministically chosen as subsets of W, A, and O, respectively. This is the meaning of the three first conditions. The particular meaning of $\subseteq_T$ is explained below. The Maude way to code, for instance, A2O ⊆ A would be {A2O} union AOthers := parts(A). That is, parts(A) computes the set of all subsets of A, the singled one is A2O, and all the others are included in AOthers, which is a set of sets of processes (or, rather, of their natural number identifiers).

These difference sets—W2A, A2O, and O2W—can be of any size. They can even be empty, but not all of them at the same time, or the step would be pointless. This is what the fourth condition—W2A union A2O union O2W =/= none—is telling. The sets are used in the three last but one matching conditions to compute the values for the destination state.

The first component of the state term is the turn—a natural number between 1 and the total number of processes. Remember that the algorithm must be able to give way to several processes at once—to any number of them. The subset of processes that are attended at each step is chosen according to the turn. The way to select from W a valid subset W2A is the following. Imagine the processes arranged in a circle in the order of their numbers, with process number 1 next to the last one, closing the circle. The empty set is always a valid value for W2A. Indeed, it is the only possible value when W = ∅. If we can, and decide to put some process into W2A, the first one must be T, the turn, if T ∈ W. If T ∉ W, we choose the first process after it (round the circle, if necessary). If we want a second element into W2A, and there is at least one left in W, we choose the next after the first one (round the circle). And so on. This is what the notation W2A $\subseteq_T$ W means: that W2A is a non-deterministically chosen subset of W *starting from* T.

After having put some elements from W into W2A, the turn for the next step, T', is set to 1 plus the last element put into W2A (modulo, if needed). If W2A was left empty, then, T' is set to T. This is what the expression nextTurn(T, W2A) computes.

We choose an initial state appropriate for syncing with MUTEX:

⟨Also in module DEKKER⟩:
```
eq init = (1, none, none, {1, 2}) .
```

That is, there are a total of two processes not interested as yet in being attended; it is 1's turn.

The only thing left is defining the properties through which this module commands or allows (depending on the point of view) some base systems to start their actions. The module DEKKER we are coding is not typical in that transitions in it are just arrows without information. There is a different transition for each pair of origin and destination states, as in standard rewriting logic. This is acceptable, as long as properties are defined in the correct way. We have decided that each property has the same value at a transition that it has at its origin state. This way, a property that has the same value at two consecutive states does it with no interruptions.

⟨Also in module DEKKER⟩:
```
op doIt : Nat -> BoolPpty .
eq doIt(N) @ (T, W, A, O) = N ∈ A .
eq doIt(N) @ step(T, W, A, O, W2A, A2O, O2W) = N ∈ A .
```

We can now sync `MUTEX` with `DEKKER`. We need to make them agree on which processes are granted access, and transfer the property to the composition:

```
mod MUTEX+DEKKER is
   pr MUTEX || DEKKER
   sync on MUTEX.doIt = DEKKER.doIt .
   op doIt : -> BoolPpty .
   var N : Nat .
   var M : MUTEX.Stage .
   var D : DEKKER.Stage .
   eq doIt(N) @ < M, D > = MUTEX.doIt(N) @ M .  --- or using DEKKER
endm
```

This control can be applied to any pair of base systems that need it, provided they are equipped with suitable properties.

```
mod ALL-TOGETHER is
   pr BASE1 || BASE2 || MUTEX+DEKKER
   sync on MUTEX+DEKKER.doIt(1) = BASE1.isDoingIt
        /\ MUTEX+DEKKER.doIt(2) = BASE2.isDoingIt .
endm
```

In this case, we have preferred not to show any particular examples of base systems, to remark that controllers have complete meanings by themselves.

5 A case study: cache architecture

5.1 Introduction

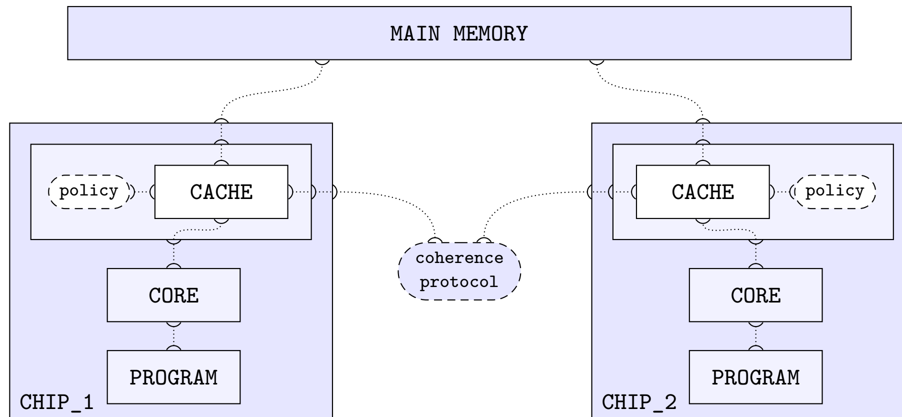
As a larger example to illustrate the use of our tools, we have chosen cache-coherence protocols. This is an example complex enough to allow displaying all the power and flexibility of the synchronous composition. The examples in the previous section were chosen for educational purposes; the present one is for showing off.

The setting is this: In a multi-core computer architecture, it is usual that each core has a small but fast memory for its private use, called cache memory, or just *cache*. Reads and writes are performed on the cache and only propagated to main memory when needed. Such decentralization has its dangers, namely, that a core writes a new value to its cache while other caches storing the same memory address are not aware of this change. It is to avoid such problems that *cache coherence protocols* are devised. A few more explanations are given below. For deeper explanations we refer the reader to [**?**].

Our model has four kinds of physical components: program storage, core, cache, and main memory. The union of a cache, a core and a program storage is called a *chip* (for the want of a better name).

The complete system consists of any number of independent chips, together with a main memory. Real systems of this kind also include a bus or similar component as communication channel between chips and main memory, but we will omit it to keep our model a little simpler. Apart from these physical components, there are two kinds of controllers: a cache policy and a coherence protocol.

This is a sketch of the whole system, showing two chips:



As in Section 4.1, systems are depicted as boxes, bump-shaped connectors represent the properties provided, and dotted-line wires represent interactions, that is, syncing. Rounded boxes are components that do not represent a physical component. (Although there won't really be a component to implement the protocol, for reasons to be seen in due time.) In this picture a single connector and a single wire may represent several properties and several syncing criteria. A component system, from the outside, is seen as a black box with properties to sync on. Some properties are transferred from a system to the composed system it belongs to, as shown in the picture.

Note that the policy only affects the cache. Then, the cache (with its policy), the core, and the program are synced to form a chip, the three at once, as a triplet, even though the cache and the program are not directly related. The protocol is used to guarantee the coherence of the data stored in the caches. Finally, main memory interacts with the caches to exchange data.

In the remainder of this section we show the use of the synchronous composition operation to model a cached architecture. At the same time, we explore some practical and methodological issues. They are explained as the need arises, in subsections with "methodology note" in their title.

At the beginning of Section 4 we described a common module that we assumed to be silently imported in all modules that need it. It is also needed in this section. It includes declarations for sorts `State`, `Trans`, `Stage`, and `Ppty`, plus all the needed property subsorts and the respective `@` operators. In addition, some other declarations and definitions are used in several modules in this example. In particular, as syncing is established by agreement on the value of some properties, it is necessary that the component modules share the data types for the values to be synced on. These are described and coded once when they first occur and assumed present whenever needed without further notice.

5.2 The program storage

The program is the list of instructions to be executed by the core (with the help of the rest of the system). It does not change during the execution, so the module representing it has no rules. It just stores an initial and unique state. Each instruction has a number. An instruction and its number form a *program line*. The program counter, that is, the number of the instruction being

currently executed by the core, is not part of this module, but of the module `CORE` shown below.

⟨In module `PROGRAM`⟩:
```
sort ProgLine .
op (_,_) : Nat Instr -> ProgLine .
op {_} : Set{ProgLine} -> State .
eq init = { (0, (r, 0))
            (1, (w, 0, 1))
            (2, (g, 0))
            (3, s) } .
```

A polymorphic implementation of sets is assumed here and several times in what follows.

We assume that the instruction numbers in the program are non-repeating, consecutive, and starting at 0. The program we have included as `init` is just an example where an instruction of each type is used. As it stands, instruction number 3 is not reachable. We consider four types of instructions: *read*, *write*, *go-to* and *stop*. This is not a Turing-complete instruction set—some kind of conditional jump would be needed, as well as arithmetic and logic calculations. But adding such instructions would make all the specification more complex without any clear advantage for us, so we omit them.

These are the four types of instruction:

⟨Assumed imported into any module that needs it⟩:
```
sorts ReadInstr WriteInstr GotoInstr StopInstr Instr .
subsorts ReadInstr WriteInstr GotoInstr StopInstr < Instr .
op (r,_) : Nat -> ReadInstr .   --- Nat is memory address
op (w,_,_) : Nat Nat -> WriteInstr .   --- memory address, data
op (g,_) : Nat -> GotoInstr .   --- Nat is instruction number
op s : -> StopInstr .
```

Memory addresses and data are natural numbers, but we consider their size limited to the number of bits represented by the constant `sizeLimit`:

⟨Assumed imported into any module that needs it⟩:
```
op sizeLimit : -> Nat .
eq sizeLimit = 4.
```

The choice of 4 as size is arbitrary. We require addresses and data to be the same maximum size, simplicity being the only reason. Sharing this declaration globally implies a good architectural design, in which values can be passed from component to component with no overflow.

The interface for syncing with the `PROGRAM` consists of one parametric property:

⟨Also in module `PROGRAM`⟩:
```
op instr : Nat -> Instr?Ppty .   --- Nat is instruction number
var N : Nat .   --- program line number
var I : Instr .
var LL : Set{ProgLine} .
eq instr(N) @ { (N, I) LL } = I .
eq instr(N) @ { LL } = error [otherwise] .
```

The `PROGRAM` is to be synced only with the `CORE`, and the only information the `CORE` needs from it is which instruction is stored in the program line with a given number. We are defining an infinite family of properties, `instr(N)`, for all values of the argument `N`. There is a property in `CORE` (called `instrFromProg` there) with which `instr` syncs. That property is also parametric, but it is defined in each case just for one value of the parameter, corresponding to the program counter

that the core stores; thus, there is only one equality to test. This is an instance of a value-passing scheme that we use several times and discuss below.

We prefer to define the value of all the rest of the properties in the family `instr` as `error`, just in case the core asks for some non-existent instruction number. If we left that undefined, the core could go on with whichever value it decided to choose.

One piece of notation needs explanation: the `?` in the middle of `Instr?Ppty`. It was already used and introduced in Section 4.2. It is frequently the case that we need to add some dummy or invalid value to an already existing sort, for cases where the actual values do not exist or do not matter. Those values are usually given names such as `none`, `void`, or `error`. Our convention is that if sort `ASort` has already been declared, we take for granted the existence of sort `ASort?`, with all the elements of `ASort` and a dummy element. We allow ourselves to use whichever name fits better in each case for the added element. Thus, the name `Instr?Ppty` refers to the sort of the properties whose values are either instructions or a dummy instruction-like value—`error` in the code above.

5.3 The core

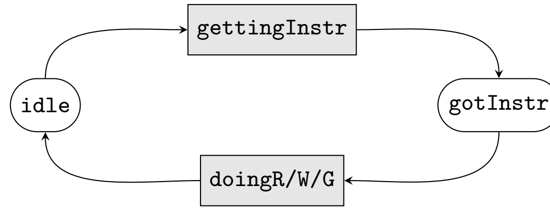
The core's job is to accept instructions and execute them, by itself, or by giving commands to the cache. Some sort declarations first:

⟨In module `CORE`⟩:
```
sorts StMode TrMode .
ops idle gotInstr -> StMode .
ops gettingInstr doingR doingW doingG -> TrMode .
op (_,_,_,_) : StMode Nat? Nat? Instr? -> State .
op (_,_,_,_) : TrMode Nat? Nat? Instr? -> Trans .
```

The first natural number, both in a state and in a transition, is the program counter, that is, the number of the program line currently being executed (or next to be). The second natural number represents the contents of the core's data register. Its maximum size is given by the constant `sizeLimit`, as defined above. (The data register is a very small memory in which the core stores the last data used, for its immediate access.)

State modes (`StMode`) and transition modes (`TrMode`) are ways to identify the different phases a system can go through. Depending on the mode, not all fields are used in all transition and state terms—we fill in using dummy constants when needed. In this way, all the declarations for `State` and for `Trans` are satisfyingly similar. Using tuples as rule labels may seem odd at first but, again, we intend to treat transitions and states as equals.

A state of the `CORE` can be in one of two modes: `gotInstr`, when an instruction has been read but not run, and `idle`, after an instruction has been run and the next one has not been read yet. As for transitions, there are four modes, and a rule for each of them: one for receiving instructions and three for executing, according to whether the instruction is a reading, a writing or a go-to. This loop is a scheme of the workings of the core:



This is the first rule, the one for getting an instruction:

⟨Also in module CORE⟩:
```
var PC : Nat .  --- program counter
var D : Nat .  --- data
var I : Instr .
var II : Set{Instr} .
crl [(gettingInstr, PC, D, none)] : (idle, PC, D, none)
                                 => (gotInstr, PC, D, I)
   if I II := instrSet(sizeLimit) .
```

The function `instrSet` is assumed to generate the set of all possible instructions up to the limit allowed by the given size. We use empty syntax for sets, so that union is represented by mere juxtaposition of subsets and individual elements. Thus, the matching condition `I II := someSet` means that `I` must be non-deterministically chosen among the elements of `someSet`.

The constant `none` is a dummy value of sort `Instr?`. Terms of sort `Trans` with `gettingInstr` as first component always have `none` as last component. As announced, we are using the dummy value just so that all state and transition terms have four components.

Fresh variables on the right hand side of a rule are problematic for executability; see Section 3.3.2. A remedy is finding some bound to the values of such fresh variable. In this case, `I` is restricted to the set of possible instructions for the given register size. The core is ready to accept any instruction into `I`. This is in accordance with the idea that each component models the capabilities of a system in all its wild behaviour, to be restricted later through synchronous composition. This system, if run alone, would go on with some non-deterministically chosen instruction; the synchronization with the `PROGRAM` makes the composed system deterministic.

The property needed here is:

⟨Also in module CORE⟩:
```
op instrFromProg : Nat -> Instr?Ppty .  --- Nat is program counter
eq instrFromProg(PC) @ (gotInstr, PC, D, I) = I .
```

This property poses no conditions for idle states nor for transitions. It is just this state between reading a new instruction and running it that must sync with the program, with the condition `PROGRAM.instr(PC) = CORE.instrFromProg(PC)`. This is further discussed below.

Note that it is not the *getting* transition, but the destination state, that gets the value (the instruction) and, therefore, the one that has to sync with the program. That's why `I` does not occur in the rule label term. This is an instance of a transition with multiple destination states (and also multiple origin states).

Now, the three rules for executing instructions. We need no rule for the stop instruction `s`—its absence will make the system stop after such an instruction is received. The go-to instruction is dealt with like this:

```
⟨Also in module CORE⟩:
   var N : Nat .   --- program line number to go to
   rl  [(doingG, void, D, (g, N))] : (gotInstr, PC, D, (g, N))
                                  => (idle, N, D, none) .
```

In words: the core received an instruction to go to program line `N`. Its program counter was `PC` and is changed to `N`. This is all that is needed for go-to instructions. Their execution needs no help from other parts of the computer to be complete, so no syncing is needed in this case.

For a writing instruction:

```
⟨Also in module CORE⟩:
   var A : Nat .   --- memory address
   rl  [(doingW, PC, D, (w, A, D))] : (gotInstr, PC, D, (w, A, D))
                                   => (idle, s PC, D, none) .
```

Remember that `s` is the successor operator for natural numbers. This rule does nothing meaningful by itself. It must be synced with the useful parts of the cache, as shown below. We have included in the label term the parameters `A` and `D`, that are not needed to satisfy the "from any to any" condition (as described in Section 2.2). However, it seems natural that the address and the data are attributes of a writing. Indeed, when this transition is synced with the cache module, the syncing criteria will use these values.

Finally, for a reading instruction:

```
⟨Also in module CORE⟩:
   var DD : Set{Nat} .
   crl [(doingR, PC, void, (r, A))] : (gotInstr, PC, D', (r, A))
                                   => (idle, s PC, D, none)
      if D DD := natSet(sizeLimit) .
```

The expression `natSet(sizeLimit)` is assumed to generate the set of all natural numbers from 0 to $2^{\texttt{sizeLimit}} - 1$. The core is here ready to accept any data (from the cache) within the architecture size limit. As above, this rule does nothing useful by itself—it must be synced with the parts of the cache that perform or command the actual reading from some memory storage.

In this rule, as was the case for `gettingInstr`, it is important that `D` is not part of the transition term, because that value is only known at the end of the reading, not while the reading is being performed. The other system (the cache) has the value available only after executing the actions that perform the actual reading.

The initial state for this component must be like this:

```
⟨Also in module CORE⟩:
   eq init = (idle, 0, void, none) .
```

We prefer to delay the definition of all the properties for `CORE` until presenting the `CHIP` module.

5.4 Methodology note: value passing

Some instances of syncing can be seen as implementations of value passing from one system to another. Such instances consist of a system that, at its current stage, has several options to choose from to go on, and another system that has only one possible way. Appropriate syncing, then, produces a deterministic composed system.

An example of this is the passing of the next instruction to be executed from the `PROGRAM` to the `CORE`. In the `CORE` there is a rule to the effect of

```
crl [(gettingInstr, PC, D, none)] : (idle, PC, D, none)
                                => (gotInstr, PC, D, I)
   if I is a valid instruction .
```

For given values for `PC` and `D`, there is only one origin state and only one transition, but several possible destination states. The appropriate destination state has to be chosen according to the information stored in the `PROGRAM`, which is completely determined for each given value of the program counter `PC`.

The general way to emulate value passing is to define a property at each system for the value to be passed. That is, in `CORE` we need a property `instrFromProg` with the value of the variable `I`, and in `PROGRAM` we need the property called `instr`, as coded above. Syncing those properties does the trick.

In this example, the properties use a parameter—the program counter. This is a usual case. But this example is exceptional because one of the systems has no rules. In general, obtaining the value may involve the execution of several rules at the *deterministic* side. This indeed happens in the interaction between the core and the cache described in the following sections.

5.5 The cache

The cache is specified in such a way that it needs to be controlled by an external module, the `POLICY`, that is specified independently below.

We begin by declaring a couple of sorts that this module shares with others to be seen:

```
⟨Assumed imported into any module that needs it⟩:
   sorts RWInstr Line .
   subsorts ReadInstr WriteInstr < RWInstr .
   op (_,_) : Nat Nat -> Line .  --- parameters are address and data
```

A line is the minimum amount of data the cache can deal with. The number of lines a cache can store varies among different architectures. In our architecture each cache contains exactly one line. This is not realistic, but it does not imply an important loss of generality, as the cache only deals with a line at a time anyway. The sort `Line` stores a line of data together with its address in main memory. In our implementation, the sizes of the memory address and the data are given by `sizeLimit`, already declared.

Other needed sort declarations, just for this module:

```
⟨In module CACHE⟩:
   op (_,_) : Line? RWInstr? -> State .
   op gettingInstr : Line? -> Trans .
   op doingR : Nat Nat -> Trans .  --- address and data
   op gettingLineFromMM : Nat -> Trans .  --- address
   op doingW : Nat Nat -> Trans .  --- address and data
   op copying2MM : Line RWInstr? -> Trans .
   op invalidatingLine : RWInstr -> Trans .
```

In this module, rule label expressions are not tuples as above. This provides examples of different, equally valid alternatives.

The action of getting a new instruction is very similar to the one for the core. Note that only *read* or *write* requests arrive to the cache.

```
⟨Also in module CACHE⟩:
   var L : Line? .
   var I : RWInstr .
   var II : Set{RWInstr} .
   crl [gettingInstr(L)] : (L, none) => (L, I)
```

```
      if I II := rwInstrSet(sizeLimit) .
```

In this rule, as was already the case for some others above, only a part of the state changes. The part that does not change has to occur in the transition term so that it satisfies the “from any to any” condition.

The simplest instruction is a reading one in which the address requested is the one already stored in the cache line:

```
⟨Also in module CACHE⟩:
   vars A D : Nat .
   rl [doingR(A, D)] : ((A, D), (r, A)) => ((A, D), none) .
```

As in previous cases, this rule may seem to do nothing, but it completely describes the changes that occur in the state of the cache. What is missing is the passing of the data `D` to the core, that is, syncing.

When the address requested is not the one currently in the cache, the following rule must be executed first, with the purpose of getting the needed line from main memory:

```
⟨Also in module CACHE⟩:
   vars A' D' : Nat .
   var DD : Set{Nat} .
   crl [gettingLineFromMM(A')] : ((A, D), (r, A'))
                               => ((A', D'), (r, A'))
      if A =/= A'
      /\ D' DD := natSet(sizeLimit) .
```

This is yet another instance of value passing. The variable `D` must *receive*, through property syncing, a value from the module `MEMORY` (specified below). This rule produces a state on which `doingR` can and must be applied to complete a reading.

A writing request is dealt with in one simple step:

```
⟨Also in module CACHE⟩:
   rl [doingW(A, D)] : (L, (w, A, D)) => ((A, D), none) .
```

That is, the cache stores the address and the data in its line. Copying the contents of the cache line to main memory is certainly needed at times, to allow public access to it. The exact moment at which this happens depends on the policy being applied, that we discuss in Section 5.9. This module does not care about when the copy to main memory is performed, but provides a rule to that aim:

```
⟨Also in module CACHE⟩:
   rl [copying2MM(L, I)] : (L, I) => (L, I) .
```

This rule can only be executed when the line `L` is not void. However, an explicit condition for this is not needed, as we have declared the operator `copying2MM` as accepting only a `Line` as first argument, not a `Line?`. The state does not change when applying this rule. Thus, the same transition can be executed several consecutive times, and some kind of control is needed... somewhere else.

The remaining rule is to *invalidate* the line stored in the cache, that is, to take note that the stored value for the stored address is not valid any more This must be done when some other cache has written to its line a new value for the same address. This cache does not need the new value right now, so it is enough to note that the old one is invalid. In real-world caches, the line of data is not deleted, just marked as not valid; we make the line void.

```
⟨Also in module CACHE⟩:
   rl [invalidatingLine(A, I)] : ((A, D), I) => (void, I) .
```

We include the parameter `A` in the transition term. It seems natural that the action of invalidating depends on the address stored in this cache and overwritten in some other. As it stands, this rule can be executed at any time (though just one time in a row). If we do not want this to happen (which we surely don't), this needs syncing—namely, with writing instructions from other caches. More on this in Section 5.15.

5.6 Methodology note: logical implication

As explained in Section 2.4, syncing on criteria that are not equalities is not necessary (nor appropriate). It is not necessary because other kinds of criteria can be emulated by means of just equalities. In this case, we are concerned with logical implication, because we use it several times below. That is, given Boolean properties b_1 in system $\mathcal{S}_1$ and b_2 in system $\mathcal{S}_2$, we want to allow stages g_1 and g_2 to happen at the same time only when

$$b_1 \mathbin{@} g_1 \Rightarrow b_2 \mathbin{@} g_2.$$

This is implication in the classical logical sense of $\neg A \vee B$.

Following the steps described in Section 2.4 for this particular case produces a result that is not the simplest possible. Our tailor-made proposal, instead, is the following. Define in $\mathcal{S}_2$ a new property b_2', a weakened version of b_2, by

$$b_2' \mathbin{@} g_2 := \text{false} \;\text{ if } b_2 \mathbin{@} g_2 = \text{false}.$$

That is, falsehood is preserved but truth of b_2 is turned into undefinedness of b_2'. The syncing criterion we propose is simply

$$b_1 = b_2'.$$

As an undefined value poses no restrictions on syncing, the requirement contained in this criterion is that if $b_2' = b_2 = \text{false}$, then $b_1 = \text{false}$. Or, equivalently, that $b_1 \Rightarrow b_2$, as we needed.

In later sections we use this trick, and express it by using in the **sync on** section of a synchronous composition a syntactic sugar like

```
MOD1.B1 implies MOD2.B2
```

5.7 Methodology note: patterns as properties

In many occasions we use Boolean properties whose definition depends only on the shape of the state or transition term at which it is evaluated. As an example, a property that informs of whether the `CORE` is running a reading instruction can be defined as true at any tuple with `doingR` as its first component, and false otherwise:

```
op isReading : -> BoolPpty .
eq isReading @ (doingR, PC, D, I) = true .
eq isReading @ G = false [otherwise] .
```

We propose the use of patterns of terms as shortcuts in those cases. Thus, our example property `isReading` is represented by `(doingR, _, _, _)` or even `(doingR, ...)`. As shown, we use underscores as placeholders for unimportant

parameters, and ellipsis for a sequence of several of them. We can even use the expression `_:State` to represent a property to tell states from transitions.

A very similar idea was proposed in [?] for proof terms under the name of *spatial actions*. Only Boolean properties are amenable to be expressed like this. And they are completely defined: true for transitions and states whose terms match the pattern, false otherwise.

To mark clearly that such expressions denote properties, and are not standard state or transition terms, we use them enclosed in double square brackets: `[[(doingR, ...)]]`, `[[_:State]]`, and so on.

The use of patterns in syncing criteria compromises the modularity of the specification. Syncing criteria must not involve internal details of the implementation, like the operators present in patterns. Even if this is a concern (and it should), we can still use patterns to ease the definition of properties. For instance:

```
eq isReading = [[(doingR, ...)]] .
eq isState = [[_:State]] .
```

5.8 Methodology note: until

The cache policy, as explained in the next section, mandates that, immediately after writing to the cache line, the change be propagated to main memory. That is, no significant action in the cache is allowed between the writing and the propagation. This is an instance of a general scheme that we can name "after p_1 satisfy p_2 until p_3", for Boolean properties p_1, p_2, p_3 defined in a base system module $\mathcal{M}$. More verbosely: after a sequence of stages (maybe just one) satisfying p_1, the next stages (at least one) must satisfy p_2 until eventually reaching a stage at which p_3 is satisfied. This kind of control can certainly be exerted through synchronous composition. The following is our proposal.

Consider this module `UNTIL`:

```
⟨In module UNTIL⟩:
   ops inBetween somewhereElse : -> State .
   ops trigger target : -> Trans .
   rl [trigger] : somewhereElse => inBetween .
   rl [target] : inBetween => somewhereElse .
   eq init = somewhereElse .
```

There are two states and two transitions arranged in a loop. An action is considered the trigger (the one corresponding to p_1), the other the target (p_3). The non-significant parts, the ones satisfying p_2, are represented by the state `inBetween`. Let's define appropriate properties:

```
⟨Also in module UNTIL⟩:
   ops isTrigger isAllowedInBetween isTarget : -> BoolPpty .
   eq isTrigger = [[trigger]] .
   eq isInBetween = [[inBetween]] .
   eq isTarget = [[target]] .
```

The syncing between these properties and p_1, p_2, p_3 cannot be by mere equality. Stages of $\mathcal{M}$ satisfying p_3 can be visited when triggered by p_1... or in some other situation. Also, stages allowed in between can be also allowed elsewhere. What we need here are implications, as explained in Section 5.6. This is the correct way to force "after p_1 satisfy p_2 until p_3":

```
pr M || UNTIL
sync on UNTIL.isTrigger = M.p1
```

```
      /\ UNTIL.isInBetween implies M.p2
      /\ UNTIL.isTarget implies M.p3 .
```

5.9 The policy

The cache policy determines when a change in the cache line is propagated to main memory. A typical choice is to defer propagation as much as possible, until the value is going to be lost because the line is going to be overwritten. Or until another cache needs the updated value. We have chosen an easier policy, called *write-through*: each time new data is written to the cache, the change is immediately propagated to main memory. That is, the writing to the cache line triggers propagation to main memory and no significant action is allowed in between. We know how to do this from the previous section.

The properties needed in the cache are:

```
⟨Also in module CACHE⟩:
  var T : Trans .
  var S : State .
  ops isWriting isCopying2MM isNotSignificant : -> BoolPpty .
  eq isWriting = [[doingW(...)]] .
  eq isCopying2MM = [[copying2MM(...)]] .
  eq isNotSignificant @ T = false .
  eq isNotSignificant @ S = true .
```

That is, in our implementation of caches all transitions are significant and banned between writing and propagating, and all states are allowed (although, indeed, only states `(L, none)` are possible at that point).

Finally:

```
⟨In module CACHE-UNDER-POLICY⟩:
  pr CACHE || UNTIL
  sync on UNTIL.isTrigger = CACHE.isWriting
       /\ UNTIL.isInBetween implies CACHE.isNotSignificant
       /\ UNTIL.isTarget implies CACHE.isCopying2MM .
```

5.10 Methodology note: synchrony as in Greek

Another scheme that frequently arises is that of synchrony in its etymological sense, that is, establishing that some sequence of stages in one component has to happen at the same time that some other sequence in the other component. For example, the cache performs actions for reading the value in its line and the core performs actions reflecting the reading, and both sequences of actions must be simultaneous. It is not important the particular interleaving of individual actions and states in the sequences, but only that the sequences as a whole begin and end at the same time.

This can certainly be implemented by synchronous composition. A simple way is to delimit *sections* in the code, assigning to each section a value (of a property) with equal values for sections that must be simultaneous. The values of those properties can be of any sort—Booleans often work fine.

In our examples, it is often the case that a single transition in one system syncs with several in the other, like when the core *commands* the cache to read. However, as we have already reminded more than once, whether a certain action is represented in a system by just one transition or several of them is something circumstantial, that depends on the degree of refinement of the specification.

In some cases, value passing and sectioning can be accomplished using only one property in each system. Take, for example, the core requesting a reading from the cache and the cache processing the request. We can define a property (in each system) with the value of the address being accessed, giving to it a dummy value `none` when the system (either one) is not reading. Syncing on those properties passes the value and also delimits two sections: one where the value of the property is `none`, and the rest.

All this will be used soon.

5.11 Methodology note: after

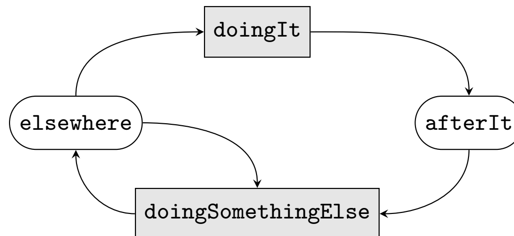
In the module `CORE`, as specified above, a state of the form `(idle, PC, D, none)`, for some values of `PC` and `D`, is reached after executing any of the rules `doingR`, `doingW`, or `doingG`. Only when reached after `doingR` must such a state be synced, so that the value of `D` is *received* from the `CACHE`. The sync is neither necessary nor acceptable when the same state is reached through the other rules. This is the problem: how can we get a same state to sync or not based on the transition that got the system to it?

An easy but imperfect solution for our example consists in replacing the mode `idle` with two: one called, say, `idleAfterR`; the other, `idleOtherwise`. The rule for `doingR` would produce a state with mode `idleAfterR`. Syncing with the cache for receiving the value of `D` would be specified only for states in this mode. This solution is imperfect, because the difference between the two idle modes has no importance regarding the workings of the `CORE`, but only for the particular way we need to sync it in this case.

Our solution is based on the synchronous composition operation, as was to be expected. Consider this module:

```
mod AFTER is
   op afterIt elsewhere : -> State .
   op doingIt doingSomethingElse : -> Trans .
   var S : State .
   rl [doingIt] : elsewhere => afterIt .
   rl [doingSomethingElse] : S => elsewhere .
endm
```

Graphically:



Now, we can compose `CORE` with `AFTER`, and establish criteria such that the transition `doingIt` syncs with transitions with mode `doingR`. All other transitions have to sync with `doingSomethingElse`. And states have to sync with states. In the composed system `CORE || AFTER`, the states of the form

```
< (idle, PC, D, I), afterIt >
```

correspond to the ones with mode `idleAfterR` in the imperfect solution above. This solves our problem in a truly modular way. Although requiring states to

sync with states is usually not appropriate, because it is not stable through refinement, we have chosen to use this solution in this quite particular case. More general solutions are, of course, possible.

We use this scheme several times below, and we think it deserves a shortcut. For the general case, suppose we have a Maude module $\mathcal{M}$ in which we have completely defined a Boolean property b and there is also another property p of sort, say, `XPpty`. We use the name "p `after` b" to denote a property that has the value of p if b was true at the stage previous to the current one; it is undefined otherwise. Using this shortcut we write in the next section a criterion similar to

```
CORE.(dataFromCache after isDoingR) = CACHE.dataStored
```

The exact equivalence of this shortcut is the following. Whenever a property $\mathcal{M}$`.(`p `after` b`)` is used, it is to be understood that we are working not really in $\mathcal{M}$ but in this system $\mathcal{M}^\bullet$:

```
⟨In module M•⟩:
   pr M || AFTER
   sync on M.b = AFTER.[[doingIt]]
        /\ M.[[_:State]] = AFTER.[[_:State]] .
   var G : M.Stage .
   op p after b : -> XPpty .
   eq (p after b) @ < G, afterIt > = M.p @ G .
```

Thus, the property is only defined when `AFTER` is in state `afterIt`; otherwise, it poses no requirements for syncing.

5.12 Chips

Our next step is assembling the (controlled) cache, the core, and the program to produce what we call a chip. Our final computer will include more than one chip, but the specification is the same for each of them. Leaving the syncing criteria to be made explicit later, this is the chip:

```
⟨In module CHIP⟩:
   pr PROGRAM || CORE || CACHE-UNDER-POLICY
   sync on ⟨Criterion for passing the instruction from the program to the core⟩
        /\ ⟨Criterion for passing the instruction from the core to the cache⟩
        /\ ⟨Criterion for syncing the core and the cache for writing⟩
        /\ ⟨Criteria for the core to read data from the cache⟩ .
```

We have already discussed the first criterion:

```
⟨Criterion for passing the instruction from the program to the core⟩:
   PROGRAM.instr(PC) = CORE.instrFromProg(PC)
```

This needs

```
⟨Also in module CHIP⟩:
   var PC : Nat .
```

The two properties, `instr` and `instrFromProg`, have already been defined in Sections 5.2 and 5.3.

Similar to the former is the following, using three properties still to be defined:

```
⟨Criterion for passing the instruction from the core to the cache⟩:
   CORE.instrToCache
   = CACHE-UNDER-POLICY.(instrFromCore after isGettingInstr)
```

The shortcut with `after` is used as discussed in the previous section.

We define both instruction-valued properties to be `none` when no actual value passing is taking place. This is needed so that the particular points where the value passing has to happen are always visited simultaneously. Thus, each of those two properties provide a sectioning of its system, in the style described in Section 5.10.

```
⟨Also in module CORE⟩:
   var G : Stage .
   op instrToCache : -> Instr?Ppty .
   eq instrToCache @ (doingR, PC, D, I) = I .
   eq instrToCache @ (doingW, PC, D, I) = I .
   eq instrToCache @ G = none [otherwise] .
```

In the cache side, the properties do not depend on the policy:

```
⟨Also in module CACHE-UNDER-POLICY⟩:
   op instrFromCore : -> Instr?Ppty .
   op isGettingInstr : -> BoolPpty .
   var Ca : CACHE.Stage .
   var Un : UNTIL.Stage .
   eq instrFromCore @ < Ca, Un > = CACHE.instrFromCore @ Ca .
   eq isGettingInstr @ < Ca, Un > = CACHE.isGettingInstr @ Ca .
```

with

```
⟨Also in module CACHE⟩:
   op instrFromCore : -> Instr?Ppty .
   op isGettingInstr : -> BoolPpty .
   var G : Stage .
   eq instrFromCore @ (L, I) = I .
   eq instrFromCore @ G = none [otherwise] .
   eq isGettingInstr = [[gettingInstr(_)]] .
```

This is the next criterion:

```
⟨Criterion for syncing the core and the cache for writing⟩:
   CORE.sectionW = CACHE-UNDER-POLICY.sectionW
```

The two properties are Booleans, in the spirit of what was discussed in Section 5.10:

```
⟨Also in module CORE⟩:
   op sectionW : -> BoolPpty .
   eq sectionW = [[(doingW, ...)]] .
```

At the cache side, again, the property does not depend on the policy:

```
⟨Also in module CACHE-UNDER-POLICY⟩:
   op sectionW : -> BoolPpty .
   eq sectionW @ < Ca, Un > = CACHE.sectionW @ Ca .
```

```
⟨Also in module CACHE⟩:
   ops isStateAfterDoingW sectionW : -> BoolPpty .
   eq isStateAfterDoingW @ G
    = ([[_:State]] after [[doingW(...)]]) @ G == true .
   eq sectionW @ G = [[doingW(...)]] @ G
                  or isStateAfterDoingW @ G
                  or [[copying2MM(...)]] @ G .
```

These last equations need explanation. In short, the section for writing comprises the transactions for writing to the cache line (`doingW`) and for copying to main memory (`copy2MM`), and also the state in between. The first and third lines of the last equation deal with `doingW` and `copying2MM` using patterns. Thus, the property `[[_:State]] after [[doingW(...)]]` is true at any state reached after writing to the cache line. However, this property is undefined at other states,

and this is not appropriate in our case: we need it to be false. This is why we compare its value to `true` and save the result with the name `isStateAfterDoingW` to be used in the second line of the next equation.

The last criteria are these:

⟨Criteria for the core to read data from the cache⟩:
```
      CORE.sectionR = CACHE-UNDER-POLICY.sectionR
   /\ CORE.(dataFromCache after isReading)
      = CACHE-UNDER-POLICY.dataStored
```

The last criterion's purpose is value passing; the first delimits simultaneity of actions by using sections. Let's consider sectioning first. This closely follows the definition for `sectionW` above:

⟨Also in module `CORE`⟩:
```
   op sectionR : -> BoolPpty .
   eq sectionR = [[(doingR, ...)]] .
```

⟨Also in module `CACHE-UNDER-POLICY`⟩:
```
   op sectionR : -> BoolPpty .
   eq sectionR @ < Ca, Un > = CACHE.sectionR @ Ca .
```

⟨Also in module `CACHE`⟩:
```
   ops isStateAfterGettingLineFromMM sectionR : -> BoolPpty .
   eq isStateAfterGettingLineFromMM @ G
      = ([[_:State]] after [[gettingLineFromMM(...)]]) @ G == true .
   eq sectionR @ G = [[gettingLineFromMM(...)]] @ G
                  or isStateAfterGettingLineFromMM @ G
                  or [[doingR(...)]] @ G .
```

Finally, about value passing:

⟨Also in module `CORE`⟩:
```
   op dataFromCache : -> Nat?Ppty .
   eq dataFromCache @ (idle, PC, D, none) = D .
   eq dataFromCache @ G = none [otherwise] .
   op isReading : -> BoolPpty .
   eq isReading = [[(doingR, ...)]] .
```

This is only different from `none` for idle states reached from a read. Again, the policy does not matter:

⟨Also in module `CACHE-UNDER-POLICY`⟩:
```
   op dataStored : -> Nat?Ppty .
   eq dataStored @ < Ca, Un > = CACHE.dataStored @ Ca .
```

⟨Also in module `CACHE`⟩:
```
   op dataStored : -> Nat?Ppty .
   eq dataStored @ ((A, D), I) = D .
   eq dataStored @ G = none [otherwise] .
```

The last one is valid both if the data was already in the cache or if it has been read from main memory. With property `dataStored` we are being somewhat liberal, and return the data whenever it exists. This seems consequent with the name of the property, and entails no problem, as that value is needed only at one point in `CORE`.

5.13 The main memory

This is the last physical component remaining to be modelled

⟨In module `MEMORY`⟩:
```
   op {_} : Set{Line} -> State .
   op updating : Nat Nat Nat Set{Line} -> Trans .
   --- parameters: chip id, address, data, and rest of memory contents
```

Remember that the `Line` was defined in Section 5.5 as a pair (address, data), both represented by natural numbers. Lookup is not represented by a rule in `MEMORY` as it does not involve any change in its state. Updating is the only transition this system can perform. The update operation is requested by a chip, and the request includes the chip id, the address and the value for the data to be written. We also need to keep the rest of the memory during the transition.

For simplicity, addresses whose data are not explicitly stored in the memory are assumed to store a 0. Indeed, we add this equation to remove lines that contain a 0, to help keep a short state term:

⟨Also in module `MEMORY`⟩:
```
var LL : Set{Line} .
eq { (A, 0) LL } = { LL } .
```

The action of updating is coded in two rules (which are, by the way, the last two rules in our whole example):

⟨Also in module `MEMORY`⟩:
```
vars CI A D D' : Nat .  --- CI is for chip id
vars DD CII AA : Set{Nat} .
crl [updating(CI, A, D, LL)] : { (A, D') LL } => { (A, D) LL }
   if D DD := natSet(sizeLimit)
   /\ CI CII := chipIdSet .

crl [updating(CI, A, D, LL)] : { LL } => { (A, D) LL }
   if A AA := natSet(sizeLimit)
   /\ D DD := natSet(sizeLimit)
   /\ not (A in addrSet(LL))
   /\ CI CII := chipIdSet .
```

Both rules update the contents of the memory in answer to a request from a chip. One of them adding a value for an address that was not explicit in the state (because it was storing a 0), the other when the address was explicit. The memory is ready to update any address to any data—the chip provides the correct ones with each request.

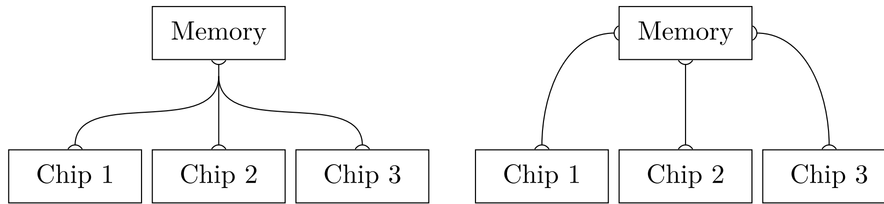
Note that we have chosen the same label expression for both rules. It seems appropriate, because whether one or the other is executed depends only on the internal representation we have chosen for the memory, namely, that zeros are not explicitly stored. This choice satisfies the "from any to any" condition. Note also that the id of the chip that is being served is not part of the state, but it is part of the transition.

Finally, an example of memory's initial contents:

⟨Also in module `MEMORY`⟩:
```
eq init = { ( 0, 4)
            ( 5, 2)
            ( 6, 2)
            (12, 5) } .
```

5.14 The computer

We decided not to include a bus in our specification, so it is the main memory that has to provide means to communicate with the chips. Which wiring is appropriate?



The left one is based on the idea that the memory sends, with each piece of information, the addressee's chip id, using its unique port. But this would require some kind of routing mechanism at the fork—a new component, that we intent to avoid. Discarding ring and other exotic topologies that would complicate the design of each component, we stick to the wiring depicted on the right.

These architectural considerations have their counterpart in the implementation. Ports and wires correspond to syncs by means of one or more properties. Our choice of topology means that the memory needs to provide different although equivalent ports (properties) for each of the chips.

This is the definition of a computer with three chips:

```
⟨In module COMPUTER⟩:
  pr CHIP_1 || CHIP_2 || CHIP_3 || MEMORY
  sync on ⟨Criteria for line updating⟩
      /\ ⟨Criteria for looking up data⟩ .
```

Each of the three chips is a renamed instance of the module `CHIP` described in Section 5.12, all exactly the same except for the module name. For that reason, all the syncing criteria and all the property definitions are the same for the three chips. The memory provides three sets of properties, with equivalent definitions.

Let's start with line updating:

```
⟨Criteria for line updating⟩:
     CHIP_1.lineBeingCopied = MEMORY.lineBeingUpdated(1)
  /\ CHIP_2.lineBeingCopied = MEMORY.lineBeingUpdated(2)
  /\ CHIP_3.lineBeingCopied = MEMORY.lineBeingUpdated(3)
```

The property `lineBeingUpdated(`i`)` returns the line that is being written to memory in answer to a request from chip i, or the constant `none` when not writing, or not by request of chip i. In the same way, `lineBeingCopied` is `none` if the chip is not in the process of copying to main memory. Assigning `none` to those properties, instead of leaving them undefined, provides sectioning, as described in Section 5.10.

Let's put all this in code.

```
⟨Also in module MEMORY⟩:
  ops lineBeingUpdated : Nat -> Line?Ppty .  --- Nat is chip id
  var G : Stage .
  var CI : Nat .
  eq lineBeingUpdated(CI) @ updating(CI, A, D, LL) = (A, D) .
  eq lineBeingUpdated(CI) @ G = none [otherwise] .
```

```
⟨Also in module CHIP⟩:
  op lineBeingCopied : -> Line?Ppty .
  eq lineBeingCopied @ < Pr, Co, CaPo >
   = CACHE-UNDER-POLICY.lineBeingCopied @ CaPo .
```

```
⟨Also in module CACHE-UNDER-POLICY⟩:
  op lineBeingCopied : -> Line?Ppty .
  eq lineBeingCopied @ < Ca, Un > = CACHE.lineBeingCopied @ Ca .
```

⟨ Also in module CACHE ⟩:

```
op lineBeingCopied : -> Line?Ppty .
eq lineBeingCopied @ copying2MM(L, I) = L .
eq lineBeingCopied @ G = none [otherwise] .
```

Only the properties about looking up data are left.

⟨ Criteria for looking up data ⟩

```
   CHIP_1.(data(A) after isGettingLineFromMM) = MEMORY.data(1, A)
/\ CHIP_2.(data(A) after isGettingLineFromMM) = MEMORY.data(2, A)
/\ CHIP_3.(data(A) after isGettingLineFromMM) = MEMORY.data(3, A)
```

It is only in the particular state after getting the line that syncing is needed. We single out that state using the `after` trick.

For each chip, `data` is a parametric collection of properties that has an address as argument, but only one of them is defined at each time: the one for the address of the line stored in the cache. In the memory, the three properties `data(i)` are defined for any address.

⟨ Also in module MEMORY ⟩:

```
ops data : Nat Nat -> Nat?Ppty .
   --- 1st Nat is chip id, 2nd is address, result is data
var T : Trans .
eq data(CI, A) @ { (A, D) LL } = D .
eq data(CI, A) @ LL = 0 [otherwise] .
eq data(CI, A) @ T = error .
```

We prefer to avoid looking up data in the memory while an updating is taking place, even if the address is not the one being overwritten. We return an `error` value in that case. This value does not exist at the chip side, which prevents such a case from actually happening.

The property for the chip depends only on the cache:

⟨ Also in module CHIP ⟩:

```
op data : Nat -> Nat?Ppty .  --- 1st Nat is address, 2nd is data
eq data(A) @ < Pr, Co, CaPo> = CACHE-UNDER-POLICY.data(A) @ CaPo .
```

⟨ Also in module CACHE-UNDER-POLICY ⟩:

```
op data : Nat -> Nat?Ppty .  --- 1st Nat is address, 2nd is data
eq data(A) @ < Ca, Un > = CACHE.data(A) @ Ca .
```

⟨ Also in module CACHE ⟩:

```
op data : Nat -> Nat?Ppty .  --- 1st Nat is address, 2nd is data
eq data(A) @ ((A, D), I) = D .
```

Finally, the remaining property:

⟨ Also in module CHIP ⟩:

```
op isGettingLineFromMM : -> BoolPpty .
eq isGettingLineFromMM @ < Pr, Co, CaPo>
   = CACHE-UNDER-POLICY.isGettingLineFromMM @ CaPo .
```

⟨ Also in module CACHE-UNDER-POLICY ⟩:

```
op isGettingLineFromMM : -> BoolPpty .
eq isGettingLineFromMM @ < Ca, Un >
 = CACHE.isGettingLineFromMM @ Ca .
```

⟨ Also in module CACHE ⟩:

```
op isGettingLineFromMM : -> BoolPpty .
eq isGettingLineFromMM = [[gettingLineFromMM(...)]] .
```

5.15 The cache-coherence protocol

The COMPUTER as specified above will not work. Consider this situation: two caches store lines with the same address and the same correct data—the same

one also stored in main memory. Then, one of the caches performs a writing to the same address, that is immediately propagated to main memory. But the other chip does not know about the change. If now this second chip reads from its cache, it will get data that is not valid any more To avoid this situation, we included in the specification of the caches an *invalidate* operation. Each cache must invalidate its line when some other writes to the same address. What we need is to control the whole system so that the writing in a cache syncs with the invalidating of the lines of all other caches that store the same address. This is called a *cache-coherence* protocol. (The possibility of the cache executing invalidation at any point is another source of danger. Invalidation must be executed only precisely when needed.)

We could add a new component to implement the protocol, as indeed shown in the picture in Section 5.1, but it turns out not to be needed. A new sync inside the `COMPUTER` module is enough:

```
⟨In module COHERENT-COMPUTER⟩:
   var A : Nat .  --- address
   pr COMPUTER
   sync on COMPUTER.amWriting(1, A)
             implies COMPUTER.amInvalidating(2, A)
        /\ COMPUTER.amWriting(1, A)
             implies COMPUTER.amInvalidating(3, A)
        /\ COMPUTER.amWriting(2, A)
             implies COMPUTER.amInvalidating(1, A)
        /\ COMPUTER.amWriting(2, A)
             implies COMPUTER.amInvalidating(3, A)
        /\ COMPUTER.amWriting(3, A)
             implies COMPUTER.amInvalidating(1, A)
        /\ COMPUTER.amWriting(3, A)
             implies COMPUTER.amInvalidating(2, A) .
```

All the properties are parametric on (the chip id and) the address. The criteria specify that, when one chip writes, all others have to invalidate if they are storing the same address. All the criteria are implications, not equalities. The reason is that chip 1, for example, may need to invalidate either because chip 2 is writing or because chip 3 is. That is, `amInvalidating(1, _)` must not imply neither `amWriting(2, _)` nor `amWriting(3, _)`, but must be implied by both.

These two properties of the computer depend only on the caches:

```
⟨Also in module COMPUTER⟩:
   ops amWriting amInvalidating : Nat Nat -> BoolPpty .
   --- 1st Nat is chip id, 2nd is address
   var Ch_1 : CHIP_1.Stage .
   var Ch_2 : CHIP_2.Stage .
   var Ch_3 : CHIP_3.Stage .
   var M : MEMORY.Stage .
   eq amWriting(1, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_1.amWriting(A) @ Ch_1 .
   eq amWriting(2, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_2.amWriting(A) @ Ch_2 .
   eq amWriting(3, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_3.amWriting(A) @ Ch_3 .
   eq amInvalidating(1, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_1.amInvalidating(A) @ Ch_1 .
   eq amInvalidating(2, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_2.amInvalidating(A) @ Ch_2 .
   eq amInvalidating(3, A) @ < Ch_1, Ch_2, Ch_3, M >
    = CHIP_3.amInvalidating(A) @ Ch_3 .
```

⟨ Also in module CHIP ⟩:
```
ops amWriting amInvalidating : Nat -> BoolPpty .   --- Nat is address
eq amWriting(A) @ < Pr, Co, CaPo >
 = CACHE-UNDER-POLICY.amWriting(A) @ CaPo .
eq amInvalidating(A) @ < Pr, Co, CaPo >
 = CACHE-UNDER-POLICY.amInvalidating(A) @ CaPo .
```

⟨ Also in module CACHE-UNDER-POLICY ⟩:
```
ops amWriting amInvalidating : -> BoolPpty .
eq amWriting(A) @ < Ca, Un > = CACHE.amWriting(A) @ Ca .
eq amInvalidating(A) @ < Ca, Un > = CACHE.amInvalidating(A) @ Ca .
```

⟨ Also in module CACHE ⟩:
```
ops amWriting amInvalidating : -> BoolPpty .
eq amWriting(A) = [[doingW(A, _)]] .
eq amInvalidating(A) = [[invalidatingLine(A, _)]] .
```

This completes the design and the specification of our computer.

6 Related work

Modularity, by itself, is not related to our work unless it aims at synchronous composition in some flavour or other. Thus, plain object oriented design and programming do not belong to this section. The division of a task into concurrent threads, as for multiprocessor architectures, is not related to our approach either, because it is usually not aimed at supporting compositional design. The list of works that are indeed related to ours includes attribute-oriented programming, agents, actors, TLA+, NuSMV, discrete event systems... From this unbounded-size list we next highlight and discuss a few items.

6.1 Process algebras

The parallel composition operator in process algebras (like CCS and CSP) is a notable precedent of our synchronous composition. See, for instance, [?, ?]. A well-known example in CCS involves the specification of a coffee machine CM:

$$\mathtt{CM} := \mathtt{coin} \,.\, \overline{\mathtt{coffee}} \,.\, \mathtt{CM}.$$

This must be read as: a coffee machine CM is ready to accept a coin, then is able to produce a coffee, and then behaves again as defined by CM. Thus, coin and coffee are action identifiers. The line over coffee means it is an output action, while coin, with no line, is an expected input. Then, a computer scientist CS using such a machine is specified as:

$$\mathtt{CS} := \overline{\mathtt{coin}} \,.\, \mathtt{coffee} \,.\, \overline{\mathtt{paper}} \,.\, \mathtt{CS}.$$

The composed system is written as

$$(\mathtt{CM} \,|\, \mathtt{CS}) \setminus \mathtt{coin} \setminus \mathtt{coffee}.$$

The *restrictions* at the end of that expression have the effect of requiring that actions coin and coffee be performed each simultaneous to its overlined counterpart. The composed system, therefore, is one able to produce papers with no end.

The specification in CSP is very similar, except that it does not distinguish input and output actions, and just requires that actions with equal identifiers in different processes happen simultaneously.

This can be translated to rewriting logic by coding two modules, `CM` and `CS`, with simple rules representing the actions and then:

```
mod ComposedSystem is
   CM || CS
   sync on CM.isAcceptingCoin = CS.isInsertingCoin
        /\ CM.isProducingCoffee = CS.isTakingCoffee .
endm
```

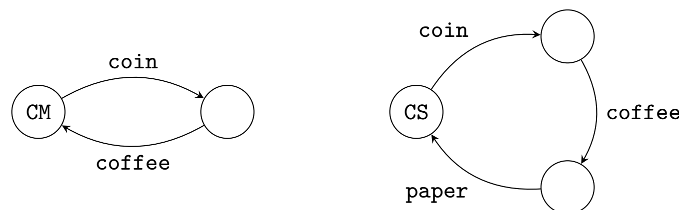
Each of the properties, in this case, is associated with one rule.

Process algebras were designed to be of theoretical interest, not to be used in the specification of real-world systems. Typically, a real-world system would need to perform some internal computations, and to store and handle complex data. Process algebras do not address these needs. Even for the purpose of syncing, process algebras restrict to actions (not states), and then, only by coincidence of atomic identifiers. Rewriting logic is more flexible and powerful, and the synchronous composition gets more involved, but also more interesting.

6.2 Automata and labelled transition structures

Both automata and labelled transition structures are formally digraphs, whose nodes represent states and whose edges represent actions. Automata are discussed in [?], and labelled transition structures, for example, in [?], although they are rather called *Kripke structures* there. In automata, edges are labelled with atomic identifiers from some alphabet. In labelled transition structures, each node is assigned a set of propositions that are said to hold at that state; edge labels can be used as well. A labelled transition structure is thought of as modelling a system; an automaton is rather thought of as accepting a language: the one whose words are the sequences of edge labels the automata can go through.

These two graphs represent the coffee machine and the scientist from the previous section:



In this example, propositions on states are not needed. Thus, the graphs can be equally seen as automata accepting infinite words or as labelled transition structures running non-stop.

Different definitions for synchronous products have been proposed; see references cited above. For automata, they are usually defined so that the language accepted by the product automata is the intersection of the languages accepted by the components. To that aim, an action is allowed in the product only when all the components can perform it simultaneously. This would not be very useful in our example, where only `coin` followed by `coffee` would be accepted by the

product system, that would then abruptly stop working. For the modelling of systems we need to accept individual actions as well as synced ones.

In any case, the drawbacks mentioned in the previous section about process algebras can be repeated here: there is no support for in-process computations, and the support for syncing is very restricted compared to our approach. Again, these formalisms cannot address real-world systems—they were not designed to that aim.

6.3 Our own previous work

Our paper [?] was a first step towards defining a synchronous composition of rewrite systems. There we proposed to sync the execution of rules in different rewrite systems based on the coincidence of rule labels. This reflects the syncing of actions that has been described above for process algebras, automata, and labelled transition structures. We also proposed to sync states by agreement on the Boolean values of propositions defined on them. This, again, reflects the way labelled transition structures are composed. We implemented that concept of synchronization on Maude and included several examples to show its usefulness.

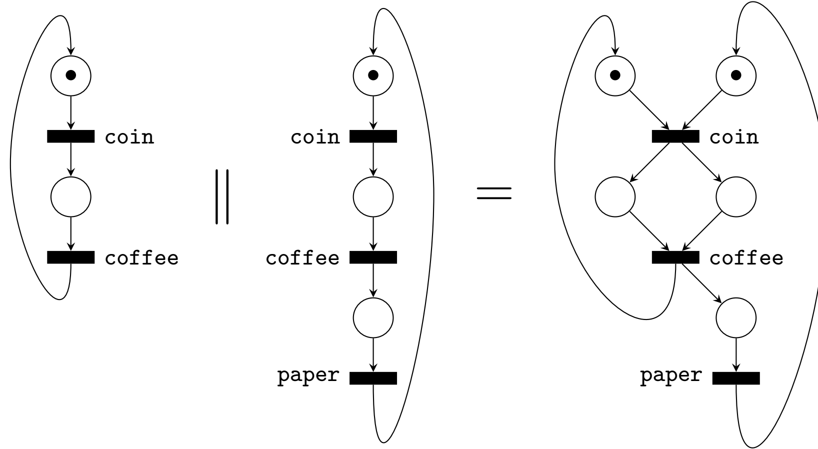
That proposal had the advantage that it used standard machinery already existing in Maude: rule labels are basic elements of Maude's syntax, and propositions are customarily defined and used to build LTL formulas for model checking. Also, working in Maude we have at our disposal all the power of functional programming and algebraically defined data types. So what is missing? Why is the present, much more involved paper needed?

We have devoted Section 2 to answer those questions. In short: Boolean-valued propositions are not enough to allow flexible synchronization and value-passing; we need to give more substance to transitions; we want to be able to sync an action at one system with several consecutive ones at the other system; etcetera. A complex realistic example like the one on cached computer architecture in Section 5 would be impossible in our previous setting.

6.4 Petri nets

Petri nets do not include compositionality as a built-in ingredient. Several works have proposed means to introduce compositionality in them, in the same spirit that the present paper proposes ideas to introduce compositionality in rewriting logic specifications. According to [?], the first to address that problem was Mazurkiewicz in [?]. Notably, while compositionality deserves no mention in [?] (one of the standard references on Petri nets), it is the subject of several chapters in the much more modern [?] (one of the standard references on coloured Petri nets).

Typically, there are two ways to compose Petri nets. One is given by hierarchical nets, that is, nets in which a transition can represent a complete separate net, that is described independently. The second way is to identify, or *fuse*, either places or transitions from two different nets. For example, the coffee machine and the scientist can be modelled and then composed by fusing transitions like this:



The problem with this naive approach is that compositional verification is difficult, because the result is computed using internal details of the specification of the components. To circumvent this problem, some approaches propose the introduction of some kind of interfaces. That is the case, for example, for the recent work described in [?]. In our simple example, the coffee machine's interface should contain some indication of the existence of the transitions `coin` and `coffee`, even though no transitions with those literal names need to exist in the specification, or even though several actual transitions in the specification may correspond to one in the interface. Components can be seen as black boxes, and reasoning about the composed system can be based on the component's interfaces, not on actual implementations.

Petri nets seem to have been extended in all possible directions. There are extensions in which complex data types are handled, and others that can include pieces of executable code within a net. Still, for the proposals about composition of nets that we are aware of, many of the drawbacks discussed previously can be repeated here: synchronization is performed only on basic terms; value-passing is not addressed; one-to-many transition syncing is not directly possible.

6.5 Maude modules and asynchronous messages

The language Maude includes a useful system of modules. A module can be imported into another, with the possibility of renaming sorts and operators on the fly if needed to avoid clashes. A typical example is the importation of the predefined module `NAT`, implementing the theory of the natural numbers.

In Maude, there is also the possibility that a module takes another as a parameter, in a way that can be seen as generalizing polymorphic types and functions. See [?, Sect. 8.3] for details. For functional modules (that is, modules that do not include rewrite rules and just define sorts and functions) this can be very useful, but for system modules (modules with rules that describe the

evolution of systems) its value is dubious, and we are not aware of any case where system modules have been used as parameters.

Back to importation, when a Maude module is imported into another, the result is like copying an exact duplicate of the imported module. Thus, this kind of modularity helps organizing the code, but does not provide compositionality by itself. Think of a system made of two components. We can separate its rules into three groups: two to describe the internal workings of each system, and a third to describe their interactions. But interactions change the internal state of both components. For example, a rule for the dispensing/drinking of a coffee would need to change the states of both the machine and the user. So, the three groups of rules cannot be disentangled in search for compositionality.

A usual construct to overcome those problems is asynchronous message passing. Rules that involve more than one component (to implement interactions) can be replaced by two: one in which a component sends a message, and another in which the other component receives and processes the message. The components must agree on the structure of the messages they exchange, but this can be seen as the specification of the interface. This certainly allows for some kind of modular design.

In rewriting logic specifications *the soup* is sometimes used. The soup is a data structure built with a commutative and associative operator. In algebraic terms, it is a multiset. It is usually written with empty syntax, which allows to write the different elements one after the other in a composed state term. This is often more convenient than using a tuple: it has the advantage that new elements, like messages, can be made to appear in the soup on the fly. In Maude, there is even an object-based notation that eases soup coding and message passing.

Asynchronous messages have been widely used and studied as a paradigm of concurrency. However, message passing has its limitations. Each component has to be prepared to send messages, as part of its workings. In our own proposal, when we need to define properties for syncing, we can do so extending the base system's specification, not modifying it. But, for message passing, the base system needs not know from the start that it is going to be synced, and be ready for it. Also, sending a message usually modifies the internal state of the sender, so that message passing has to be implemented by rewrite rules inside the code of the base system. This hinders reusability.

Implementing a controller by asynchronous message-passing is clumsy. Imagine, for instance, that a controller has sent a message with a grant for a component to perform some action. Circumstances change and the controller needs to revoke the grant. But it happens that the grantee gets ready to perform its action exactly at the same time. Some mechanism has to be implemented to give priority to the controller. Certainly, protocols can be devised to deal with this and other problematic situations, based on the sending of acknowledgements and other bureaucratic messages. But, then, each component must be internally ready, not only to send messages, but to follow protocols.

Another example where message passing is not appropriate is when physical components are physically assembled. Think of a scooter modelled by just two pieces: the handle and the front wheel. They must turn right or left at exactly the same time. If the wheel cannot turn, because a rock on the ground is blocking it, the handle cannot either. Again, protocols can probably be used, but it is not satisfactory, nor realistic, that something that must be immediate

is implemented by a multiple exchange of messages.

We are confident that all those issues can easily be addressed with our synchronous composition operation.

6.6 Aspect-oriented programming

Aspects are concerns of a system that *cross-cut* the system's base functionality. Typical examples of aspects are tracing, error handling, and monitoring (as for runtime verification). Using other programming paradigms, code related to those concerns is scattered among all modules that need it. This makes the code difficult to understand and maintain. Aspect-oriented programming, as explained in [**?**], proposes, first, grouping in one module all code related to a given aspect, and, second, establishing at which points in the base code the aspect code must be executed. The language must provide some means for establishing those *join points*. For instance, some monitoring method can be needed each time an object is created, or each time a method named `imdangerous` is executed, or each time a particular *annotation* is found in the basic code. This frees the programmer of the base code from worrying about such concerns.

The motivations for aspect-oriented programming are not too different from ours. Typical examples of use, like the three mentioned in the previous paragraph, turn out to be not only cross-cutting, but also independent from the system's base functionality. A system can be run with or without tracing. This is akin to our request that each component be meaningful by itself. The definition of properties is the mechanism we use to establish when other (aspect) code must be run.

There are differences, however. Aspects are seen as an ingredient *added* to an existing programming paradigm, like object orientation. The most notable implementation of aspects is AspectJ, built on Java. Objects provide a first tool for modularity. In our case, all modularity is based on the synchronous composition. Another difference is our insisting in simultaneity. Aspect code is usually executed right after or right before the need is found in the base code. Our components, on the contrary, execute simultaneously. Thus, a controller (or monitor) can prevent the system from executing a dangerous action. For aspects, it is more difficult to prevent, and easier to react.

A potential source of problems with aspects is that programmers who are coding base functionality do not know what is really happening in the whole system—aspect code executes out of their control. In principle, aspects can modify variables, create new objects, and so on. A monitor, for example, can be designed, not just to detect unsafe states, but to take control of the system and bring it back to safety. Such powerful aspects can be seen as breaking modularity. Our own view on this is that a monitor or controller does not introduce new behaviours on the system. If the system can be taken to a safe state, it is because it was already able to get there. Left to itself, the system would probably not choose that path, so the task of the controller is to enforce the path to safety and ban the rest. Also, our properties are defined not mixed with the base code (as annotations and function calls are), but as an extension to it, and they can be used as an interface for syncing that enforces modularity.

Our implementation of the synchronous composition operation is still in the making. But even when it finally comes to live, AspectJ with its Eclipse plug-in

will be unbeatable. On the other hand, we offer the strong theoretical basis of rewriting logic and the existing tool set around Maude.

6.7 Behavioural programming

Use-cases and scenarios are techniques used in the requirement-collection phase of software development. Behavioural programming, as described for example in [**?**], proposes that it is possible to keep these up to the coding phase, transformed into synced threads, so called *behavioural threads*, or just *behaviours*, each one corresponding to a use-case. The concept of behaviour here is really elementary. For instance, when coding the rules for the game of tic-tac-toe, there is a behavioural thread that prevents marking cell $(1, 1)$ if it is already marked; other eight behaviours do the same for the other eight cells; yet another enforces alternative turns; and so on.

The syncing mechanism for behavioural threads is based on the *request-wait-block* paradigm. Each thread, at each sync point, *request* some events (needs some of them to go on), *waits* for some others (wants to be informed if they happen), and *blocks* some others. The set of events is global and shared. When all threads reach a sync point, somehow an event is chosen that is requested by some thread and blocked by none, and so the system goes on to the next simultaneous sync point. Our property-based syncing mechanism is different, mainly because we do not require all transitions that can be run to be run indeed, but only a subset of them; in behavioural programming, all threads that request an event must go on once that event is chosen to happen.

Behavioural programming is supposed to allow incremental development, because new behaviours can be added to an already running system with no need to modify existing ones (as long as new events are not needed). For example, once the rules for tic-tac-toe are in place, behaviours implementing game strategies can be added, say to detect two marks and a free cell in a row and only allow to play the third. Probably the same can be said about our setting. One point in our favour is that each component is meaningful by itself, and properties can be added for syncing if needed. Meanwhile, for behavioural threads events are not an addition, but an indispensable part of their logic.

Our views on programming methodology also differ. In the case of the implementation of tic-tac-toe, we would design a system composed of a board, two players and, maybe, a referee. We would code each of them separately, either monolithically or, why not, in a behavioural fashion. Then, the components would be synced. Interestingly, the paper [**?**] hints at the possibility that, for very complex systems, behavioural threads may need to be grouped into *nodes*, each thread synced only with others in its node, and nodes synced by external events.

Several implementations of behavioural programming exist that, though not as impressive as AspectJ, are certainly more impressive that our non-existing one. At the risk of annoying the reader, we need to mention again the strong theoretical basis of rewriting logic and the tool set around Maude.

7 Future work

Bringing compositionality to rewriting logic can open it to new fields like coordination models and component-based software development. Also, hardware specification can systematically benefit from this new possibility. But all this is quite speculative right now. In this section we describe the more feasible tasks we intend to apply ourselves to in the near future.

In addition to those tasks, the ultimate test for our tool would be its use to model and analyse a real system. We do not mean just a realistic example, but a real-world system—either one in which rewriting logic or other formal methods have already been used with success, and that we can approach with our tools and compare results, or a new system that can particularly benefit from our methods. This will only be possible once the implementation is working and the other two lines of work described below have been at least partially explored.

7.1 Implementation

We need a usable prototype of the synchronous composition operation that supports running all the examples in this paper. It will be developed by extending Full Maude. This is a reimplementation of the Maude interpreter in Maude itself. It is described, for instance, in [**?**]. Full Maude has the advantage of being easily extensible. It is the natural choice to implement our tool.

At the syntactic level, our implementation must be ready to accept, first, complex terms in rule labels, and, second, the synchronous composition operator `||` with its **`sync on`** clauses. Also, the split of the synchronous composition must be computed when needed so as to allow an easy use of Maude's execution and search engines and the LTL model checker.

That plain implementation will need to be improved in two aspects. First, the number of rewrite rules of the resulting split system is twice the product of the number of rules of each component system—that can easily get very large and inefficient. Depending on the case, some or many of the conditions of these rules are trivially false or trivially true, and a straightforward static analysis can remove them. Even whole rules can be statically removed sometimes. This can have a large impact on performance.

Second, we have found that coding the specifications for composed systems in the way we propose is not always easy nor intuitive; but we have also found that some tricks and shortcuts can be used. See, for example, the *methodology notes* in Section 5. We must consider including some of these tricks in the implementation to increase our tool's usability.

7.2 Strategies

As important as the possibility of assembling *physical* components (like a processor and a memory) is that of using *abstract* components to control others. By *abstract* we mean that they do not represent physical entities. Several examples in this paper illustrate the use of components as controllers or—as they are usually called in rewriting logic, with roughly the same meaning—strategies. The basic idea is that we make mandatory that some actions in the base system are synced with some actions in the controller. Thus, if the controller refuses

to execute a particular action, the corresponding one on the base system is prevented.

This allows us to specify a base system with all its non-deterministic capabilities as one component, and use it only under the control of another component; even in different ways under different controls. This is the idea used in [**?**, **?**, **?**] to implement Knuth-Bendix-like completion as a basic set of correct rules on which different strategies are applied to get different actual procedures. Again, the same idea is used in [**?**] for congruence closure. And in [**?**, **?**] insertion sort is implemented as a base system with a single rule for swapping cell contents and, then, a control on how to use that rule. We have adapted this last example to our setting in Section 4.2. Also, in the large example of Section 5, physical components like caches, cores, and memories are mixed with two abstract components that implement a policy and a cache-coherence protocol.

The language Maude includes the nice possibility of working at the metalevel. That is, a Maude module can be *metarepresented*, stored, handled, and used as an object within another (metalevel) Maude module. Rewriting can be performed in a controlled way at the metalevel. This is a direct way to implement strategies in Maude. However, working at the metalevel is sometimes cumbersome, and requires a deeper understanding of Maude. And the results obtained are difficult to export to other formalisms, even to other rewriting-based ones. That's why object-level strategy languages are desirable and have indeed been developed, in Maude and in other formalisms (for Maude, see [**?**]).

There are two tasks to be addressed. The first, theoretical one is putting our proposal in the context of the existing work on strategies: studying which kinds of strategies are implementable using synchronous composition, with which advantages and drawbacks, seeing if we can help clarify the philosophical question on the nature of strategies. The second task is implementing translations that render strategies written in some strategy language as components in rewriting logic and syncing criteria to compose them.

7.3 Modular verification

Modular specification is very nice, but it gets even better if one can go on working modularly, particularly for verification purposes. This observation has certainly been made many times in the past and there is abundant work on modular, or component-wise, verification. A well-known paradigm is *assume-guarantee*. It proposes that each component assumes some nice behaviour from the rest of the system and, under such assumption, guarantees its own nice behaviour. If each component can be proven to satisfy some condition of this type, conclusions can be drawn on the whole composed system.

We do not foresee substantial theoretical developments on our part here. We just intend to adapt existing relevant work to our framework. This can include the implementation of new commands or facilities for modular model checking.

8 Conclusion

We are confident that the sexiest parts of our work are still to come. Strategies, modular verification, runtime verification, trace model checking, coordination models, component-based software development—compositionality can

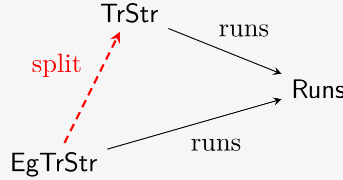
turn rewriting logic suitable for some of these or other fields. These explorations will be enjoyed, hopefully, at some future time. But all the necessary theoretical basis for them has been laid down in this paper.

We have explained why transitions and states must be treated as equals, and how this is possible in so-called *egalitarian* rewrite systems, which allow for complex rule labels instead of the usual atomic ones. We have proposed a flexible means for specifying how several rewrite systems are synced, based on agreement on the values of *properties*. We have shown the power of properties in several realistic examples, and we have also justified why we need all that power, even though it entails more complexity. We have developed the theory for labelled transition structures as well, so that our rewrite systems get a semantic ground. We have described the *split* operations, that translate sets of synced egalitarian rewrite systems into standard ones, and we have shown in a series of theorems how this allows the use of existing tools like Maude's model checker to be used to analyse our non-standard systems.

We can now begin our further explorations walking on firm ground.

A All the proofs

Theorem 1. *The following diagram is commutative:*

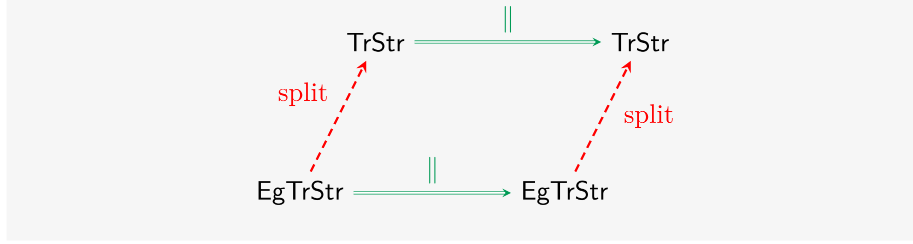


(Originally stated in page 21.)

Proof. We need separate proofs for $\mathsf{atEgTrStr}$ and for $\mathsf{EgTrStr} \setminus \mathsf{atEgTrStr}$. So, let's take first $\mathcal{T} = (S, T, \rightarrow, P, g^0) \in \mathsf{atEgTrStr}$. Remember that $\mathrm{split}(\mathcal{T}) = (S \cup T, \rightarrow, P, g^0)$. Although S and T are mixed together, the adjacency relation $\rightarrow$ is kept unchanged, so it is still bipartite and still the same. The initial g^0 is also the same. Thus, each run for $\mathcal{T}$ is also a run for $\mathrm{split}(\mathcal{T})$ and vice versa, and $\mathrm{runs}(\mathcal{T}) = \mathrm{runs}(\mathrm{split}(\mathcal{T}))$.

Let's consider now $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z} \in \mathsf{EgTrStr} \setminus \mathsf{atEgTrStr}$. By definition, the stages, the adjacency relation, and the initial stage for the composed structure and for its split are the same. That is all that is needed. □

Theorem 2. *The following diagram is commutative:*



(Originally stated in page 22.)

Proof. First of all, we need a more precise statement. In addition to the structures being composed, the synchronous composition operation takes two parameters, the ones we called Y and Z, that are not explicit in the diagram. What we precisely mean is:

$$\mathrm{split}((\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z}) = (\mathrm{split}(\mathcal{T}_1) \| \cdots \| \mathrm{split}(\mathcal{T}_n))_{Y,Z}.$$

This makes sense, because Y and Z are specified based on the properties of the component structures, and the properties of each $\mathrm{split}(\mathcal{T}_i)$ are the same of the original $\mathcal{T}_i$, according to the definition of the split. Thus, each pair $p_i = p_j \in Y$ that is interpreted as $\mathcal{T}_i.p_i = \mathcal{T}_j.p_j$ on the left-hand side of the equality, is interpreted as $\mathrm{split}(\mathcal{T}_i).p_i = \mathrm{split}(\mathcal{T}_j).p_j$ on the right-hand side. And correspondingly for the assignments in Z.

So, consider n structures $\mathcal{T}_i \in \mathsf{EgTrStr}$, either atomic or not. Let's call G_i their sets of stages, $\to_i$ their adjacency relations, P_i their sets of propositions, and g_i^0 their initial stages, so that, immediately, $\mathrm{split}(\mathcal{T}_i) = (G_i, \to_i, P_i, g_i^0)$. The stages of $(\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z}$ are tuples compatible wrt Y. These are also the states of $\mathrm{split}((\mathcal{T}_1 \| \cdots \| \mathcal{T}_n)_{Y,Z})$. And also the states of $(\mathrm{split}(\mathcal{T}_1) \| \cdots \| \mathrm{split}(\mathcal{T}_n))_{Y,Z}$.

Similar, straightforward reasoning show that the adjacency relation, the set of properties (as defined by Y), and the initial state are the same for both structures. Thus, they are indeed the same structure. □

Theorem 3. *1. The standard-setting rule*

$$l : s \to s' \ \text{ if } \ \bigwedge_{j=1}^{m} C_j$$

has the same transition semantics as the atEgRwSys *rule*

$$l(\overline{v}) : s \to s' \ \text{ if } \ \bigwedge_{j=1}^{m} C_j,$$

turning l from an atomic label into an operator, and where $\overline{v}$ is the tuple with all the variables in s, in s', and in all the conditions C_j. The term $l(\overline{v})$ is called a proof term *in standard rewriting logic.*

2. *For that case, the admissibility conditions for rules in* atEgRwSys *are equivalent to the standard ones.*

(Originally stated in page 26.)

Proof. For the first part, note that the rule "$l : s \to s'$ if $\bigwedge_{j=1}^{m} C_j$" generates a different transition, with a different proof term, for each instantiation of the variables in s, in s', and in the conditions. But that is exactly what the transition term $l(\overline{v})$ achieves.

For the second part, remember that the only admissibility condition that changes from the standard setting to atEgRwSys is the first one. In the standard setting it says

$$\text{vars}(s') \subseteq \text{vars}(s) \cup \bigcup_{j=1}^{m} C_j, \tag{1}$$

while for atEgTrStr it is

$$\text{vars}(t) \subseteq \text{vars}^*(s) \qquad \text{and} \qquad \text{vars}(s') \subseteq \text{vars}^*(t). \tag{2}$$

In this case, $t \equiv l(\overline{v})$. We have that (2) implies (1), even with no further assumptions: taking into account that $\text{vars}(t) \subseteq \text{vars}^*(s)$ implies $\text{vars}^*(t) \subseteq \text{vars}^*(s)$, and that $\text{vars}^*(s) \subseteq \text{vars}(s) \cup \bigcup_{j=1}^{m} C_j$, we have

$$\text{vars}(s') \subseteq \text{vars}^*(t) \subseteq \text{vars}^*(s) \subseteq \text{vars}(s) \cup \bigcup_{j=1}^{m} C_j.$$

Now, (1) implies (2) with the help of the assumption that t contains all variables. Thus, the statement $\text{vars}(s') \subseteq \text{vars}^*(t)$ is trivial. About the last statement left, $\text{vars}(t) \subseteq \text{vars}^*(s)$, it is not necessarily true for general rules in atEgRwSys, because t may contain variables that are bound to constants (for example, to non-deterministically choose an element from a given constant set). But when the rule label is $l(\overline{v})$ no such new variables are possible. Thus, all variables in all the C_j get their values ultimately drawn from s. Assuming the standard condition $\text{vars}(s') \subseteq \text{vars}(s) \cup \bigcup_{j=1}^{m} C_j$, we have that $\text{vars}^*(s) = \text{vars}(s) \cup \text{vars}(s') \cup \bigcup_{j=1}^{m} C_j$, which is equal to $\text{vars}(t)$ according to the theorem statement. □

Theorem 4. *The "from any to any" condition for rule labels is implied by the admissibility conditions for* atEgRwSys*.*

(Originally stated in page 27.)

Proof. The important point here is $\text{vars}(s') \subseteq \text{vars}^*(t)$. Thus, as far as s' is concerned, all its information comes through t; all information that allows to tell apart some origin states from others is lost when t starts executing. That's all we need. □

Theorem 5. *A rewrite system $\mathcal{R}$ in* atEgRwSys *satisfies the admissibility conditions for* atEgRwSys *iff* $\mathrm{split}(\mathcal{R})$ *satisfies the admissibility conditions for* RwSys.

(Originally stated in page 31.)

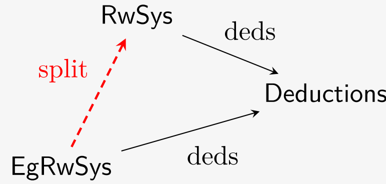
Proof. In standard rewrite systems, we have $\mathrm{vars}^*(s) = \mathrm{vars}(s) \cup \bigcup_{j=1}^{m} \mathrm{vars}(C_j)$. This is because all fresh variables in conditions need to be ultimately given value from s. (Matching conditions $u_i := u'_i$ where $\mathrm{vars}(u'_i) = \emptyset$, like $\{x, y\} := \{1, 2\}$, can be seen as a particular case, because all subset relations hold. The same for rewrite conditions.) Thus, the first admissibility condition, $\mathrm{vars}(s') \subseteq \mathrm{vars}(s) \cup \bigcup_{j=1}^{m} \mathrm{vars}(C_j)$, can equivalently be stated as $\mathrm{vars}(s') \subseteq \mathrm{vars}^*(s)$. Standard admissibility for each of the rules resulting from the split, $s \to t$ if C' and $t \to s'$ if C'', is, respectively, $\mathrm{vars}(t) \subseteq \mathrm{vars}^*(s)$ and $\mathrm{vars}(s') \subseteq \mathrm{vars}^*(t)$, the very same inclusions that define admissibility for the original rule $t : s \to s'$ if C in atEgRwSys. □

Theorem 6. *Given $\mathcal{R}$ in* atEgRwSys*, its set of rules is coherent with respect to its set of equations (according to the definition at the end of Section 3.3.2) iff* $\mathrm{split}(\mathcal{R})$ *is coherent with respect to the same equations (according to the standard definition).*

(Originally stated in page 31.)

Proof. Quite trivial. The two items that define coherence for atEgRwSys exactly correspond to standard coherence for the rules resulting from a split. □

Theorem 7. *The following diagram is commutative:*



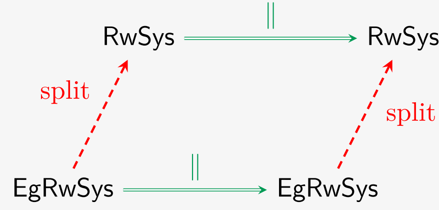
(Originally stated in page 35.)

Proof. We need separate proofs for atEgRwSys and for EgRwSys \ atEgRwSys. Out of all the deduction rules, only *rewriting* is different between RwSys and atEgRwSys. But their difference clearly parallels the split operation.

For EgRwSys \ atEgRwSys, we need to prove the following: for each given $\mathcal{R} = (\mathcal{R}_1 \| \ldots \| \mathcal{R}_n)_{Y,Z}$, a valid deduction in $\mathcal{R}$ is also valid in $\mathrm{split}(\mathcal{R})$ and vice versa. So let $\langle g_1^0, \ldots, g_n^0 \rangle$, $\langle g_1^1, \ldots, g_n^1 \rangle, \ldots$ be a valid deduction in $\mathcal{R}$. Then, each

tuple is a compatible stage in $\mathcal{R}$ and, thus, a stage in split($\mathcal{R}$), and the first tuple is the initial stage. Now, if $\langle g_1^j, \ldots, g_n^j \rangle$ is followed by $\langle g_1^{j+1}, \ldots, g_n^{j+1} \rangle$ in such a deduction, it means that for each $i = 1, \ldots, n$, either $g_i^{j+1} = g_i^j$ (but not for all i) or g_i^j is followed by g_i^{j+1} in a component's deduction. Then, there is a rule in $\mathcal{R}_i$ that allows deducing g_i^{j+1} from g_i^j. The composition of all those rules is a rule in split($\mathcal{R}$) that allows deducing $\langle g_1^{j+1}, \ldots, g_n^{j+1} \rangle$ from $\langle g_1^j, \ldots, g_n^j \rangle$. This reasoning is reversible, which gives equivalence. □

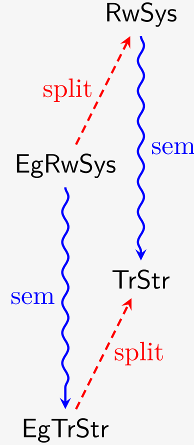
Theorem 8. *The following diagram is commutative:*



(Originally stated in page 35.)

Proof. We noted in Section 3.3.4 that this proof is trivial because the definitions for the split and for the synchronous composition in RwSys are formally almost identical. □

Theorem 9. *The following diagram is commutative:*



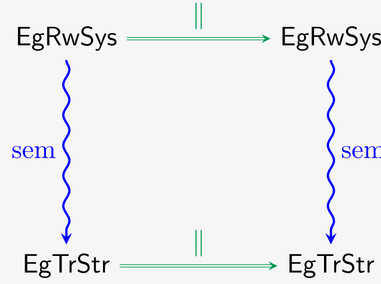
(Originally stated in page 36.)

Proof. We need separate reasoning for atEgRwSys and for EgRwSys \ atEgRwSys. So, let's take first $\mathcal{R} = (\Sigma, \leq, \Omega, E \cup A, M, R) \in$ atEgRwSys, and prove for it

sem(split($\mathcal{R}$)) = split(sem($\mathcal{R}$)). Remember that from $\mathcal{R}$ to split($\mathcal{R}$) no sort and no constructor changes, except the addition of split($\mathcal{R}$).`State` as a synonym for $\mathcal{R}$.`Stage`. Also, the sort $\mathcal{R}$.`Stage` includes $\mathcal{R}$.`State` and $\mathcal{R}$.`Trans`. Thus, the set of states in sem(split($\mathcal{R}$)) is the set of (equational classes of) ground terms of this inclusive, renamed sort. On the other hand, sem($\mathcal{R}$) has as set of states (resp., transitions) the set of (equational classes of) ground terms of sort `State` (resp., `Trans`), and the split operation joins these two sorts into the new set of states. In sum, the set of states in the same in both cases: equational classes of ground terms of sort $\mathcal{R}$.`State` and $\mathcal{R}$.`Trans` in $\mathcal{R}$.

Similar, straightforward considerations work for the adjacency relation, the set of properties, and the initial state. □

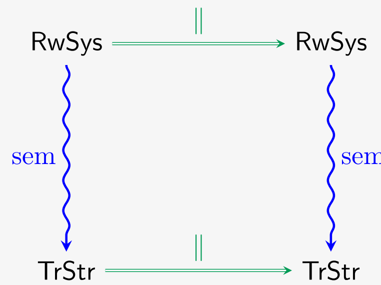
Theorem 10. *The following diagram is commutative:*



(Originally stated in page 37.)

Proof. The precise statement is: given $\mathcal{R} = (\mathcal{R}_1 \| \cdots \| \mathcal{R}_n)_{Y,Z} \in \mathsf{EgRwSys}$, we have $\text{sem}(\mathcal{R}) = (\text{sem}(\mathcal{R}_1) \| \cdots \| \text{sem}(\mathcal{R}_n))_{Y,\text{sem}(Z)}$, where sem($Z$) is the set of property definitions that result from the usual term-rewriting semantics for equational systems. But this is exactly how sem was defined for EgRwSys in Definition 2. □

Theorem 11. *The following diagram is commutative:*



(Originally stated in page 37.)

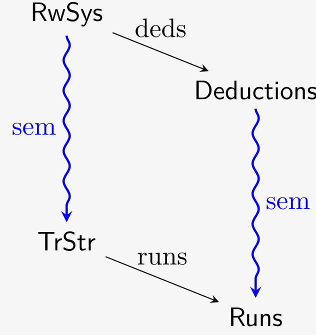
Proof. More precisely, the statement is this: given $\mathcal{R}_1, \ldots \mathcal{R}_n \in \mathsf{RwSys}$ and appropriate sets Y and Z, we have

$$\mathrm{sem}((\mathcal{R}_1 \| \ldots \| \mathcal{R}_n)_{Y,Z}) = (\mathrm{sem}(\mathcal{R}_1) \| \ldots \| \mathrm{sem}(\mathcal{R}_n))_{Y,\mathrm{sem}(Z)} \in \mathsf{TrStr}.$$

First, the set of states of $\mathrm{sem}((\mathcal{R}_1 \| \ldots \| \mathcal{R}_n)_{Y,Z})$ is, according to the definition of sem, the set of (equational classes of) ground terms of sort `State` on the symbols in $\bigcup_i \mathcal{R}_i.\Omega \cup \Omega' \cup \{\langle_\rangle, @, \texttt{init}\}$. From all those symbols, the only constructor for new `State`s is the tuple operator, $\langle_\rangle$. Thus, states are given as tuples of compatible (wrt Y) component states. Through the other path, the states of each $\mathrm{sem}(\mathcal{R}_i)$ are the equational classes of ground terms of sort $\mathcal{R}_i$.`State`. Then, the states of $(\mathrm{sem}(\mathcal{R}_1) \| \ldots \| \mathrm{sem}(\mathcal{R}_n))_{Y,\mathrm{sem}(Z)}$ are tuples of compatible (wrt Y) component states.

Similar, straightforward considerations work for the adjacency relation, the set of properties, and the initial state. □

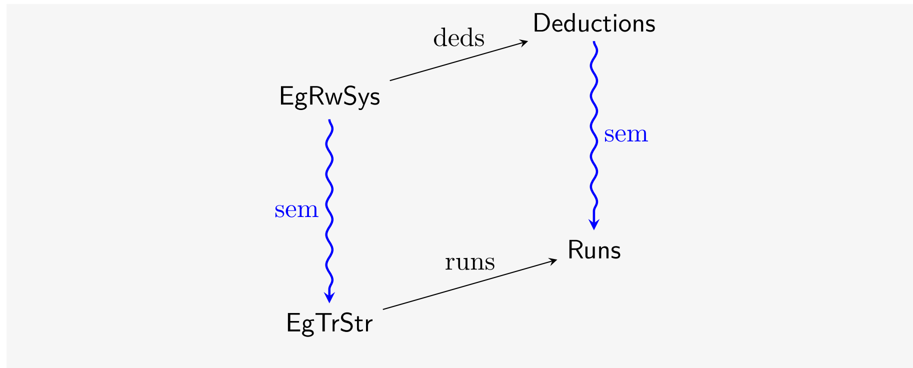
Theorem 12. *The following diagram is commutative:*



(Originally stated in page 37.)

Proof. For a given $\mathcal{R} \in \mathsf{RwSys}$, its set of deductions, $\mathrm{deds}(\mathcal{R})$, contains sequences of the shape $s^0, s^1, s^2, \ldots,$ where $s^0 = \texttt{init}$, and the step from each s^j to the next is allowed by the rules of deduction. The semantics of those are the runs $[s^0]_{E\cup A}, [s^1]_{E\cup A}, [s^2]_{E\cup A}, \ldots$ On the other hand, the runs of $\mathrm{sem}(\mathcal{R})$ have the same shape, but here each step is allowed by the rewriting relation derived from $\mathcal{R}$. It remains to be proved that the steps allowed by both means are exactly the same. A quick inspection of the definitions of the functions involved is enough. Indeed, the deduction rule we called *rewriting* in Section 3.3.7 and the definition of the adjacency relation for sem on RwSys from Section 3.4 are almost the same thing. □

Theorem 13. *The following diagram is commutative:*



(Originally stated in page 37.)

Proof. This one can be deduced from the commutativity (proved in previous theorems) of the four faces that surround it in the picture at the beginning of Section 3.1. We have:

$$\begin{aligned}
&\mathsf{EgRwSys} \xrightarrow{\text{deds}} \mathsf{Deductions} \xrightarrow{\text{sem}} \mathsf{Runs} = \\
&\mathsf{EgRwSys} \xrightarrow{\text{split}} \mathsf{RwSys} \xrightarrow{\text{deds}} \mathsf{Deductions} \xrightarrow{\text{sem}} \mathsf{Runs} = \\
&\mathsf{EgRwSys} \xrightarrow{\text{split}} \mathsf{RwSys} \xrightarrow{\text{sem}} \mathsf{TrStr} \xrightarrow{\text{runs}} \mathsf{Runs} = \\
&\mathsf{EgRwSys} \xrightarrow{\text{sem}} \mathsf{EgTrStr} \xrightarrow{\text{split}} \mathsf{TrStr} \xrightarrow{\text{runs}} \mathsf{Runs} = \\
&\mathsf{EgRwSys} \xrightarrow{\text{sem}} \mathsf{EgTrStr} \xrightarrow{\text{runs}} \mathsf{Runs}.
\end{aligned}$$

□