TPLP **20** (1): 44–98, 2020. © Cambridge University Press 2019
doi:10.1017/S1471068419000425 First published online 9 October 2019

*Compositional Specification in Rewriting Logic**

ÓSCAR MARTÍN, ALBERTO VERDEJO and NARCISO MARTÍ-OLIET
Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
(*e-mails:* omartins@ucm.es, jalberto@ucm.es, narciso@ucm.es)

submitted 8 November 2017; revised 5 September 2019; accepted 5 September 2019

Abstract

Rewriting logic is naturally concurrent: several subterms of the state term can be rewritten simultaneously. But state terms are global, which makes compositionality difficult to achieve. Compositionality here means being able to decompose a complex system into its functional components and code each as an isolated and encapsulated system. Our goal is to help bringing compositionality to system specification in rewriting logic. The base of our proposal is the operation that we call synchronous composition. We discuss the motivations and implications of our proposal, formalize it for rewriting logic and also for transition structures, to be used as semantics, and show the power of our approach with some examples.

KEYWORDS: compositional specification, rewriting logic, modularity, synchronous product

1 Introduction

To anyone in the fields of computer science and engineering, the convenience of compositionality needs not be stressed. We choose the word *compositionality*, instead of the weaker *modularity*. The latter includes the mere separation of code in chunks for better organization. Compositionality hints at the existence of interfaces and encapsulation like in, say, object-oriented programming. For example, as we show in Section 2.1, compositionality allows a mutex controller to be a separate system, interacting, but not intersecting, with the controlled components. It also allows a complex system, like a computer architecture in the example in Section 2.2, to be specified as a set of independent, interacting pieces.

Rewriting logic was started by Meseguer's (1992) paper. Maude is the name of a language and a system based on rewriting logic. It is fully described in Clavel *et al.* (2007). It has proven a useful tool for, among other things, the specification of nondeterministic and concurrent systems. In a rewriting logic specification, the state of the system is represented as an algebraic term. Concurrency is made possible by several subterms of the state term being rewritten simultaneously. Compositionality is difficult, because the state of the whole system is represented by a unique term at each moment. We intend to help fixing this limitation. Our main goal in this paper is not to introduce

* This work is partially supported by MINECO Spanish project TRACES (TIN2015-67522-C3-3-R), and Comunidad de Madrid programs N-GREENS Software (S2013/ICE-2731) and BLOQUES (S2018/TCS-4339). This paper owes a good share of whatever value it may have to the very careful and useful remarks and suggestions from its anonymous referees.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

novel theoretical proposals about compositionality, but rather to adapt existing ideas and see what rewriting logic has to offer. We want to do so departing as little as possible from the setting of standard rewriting logic.

The language Maude includes a useful system of modules. When a Maude module is imported into another, the result is equivalent to copying an exact duplicate of the imported module. Thus, this kind of modularity helps organizing the code but does not provide compositionality by itself. Asynchronous message passing is often used in Maude to maintain component specifications as separate as possible, but this also has its limitations (see Section 5 for a discussion).

Different works have very different views on compositionality (see, again, Section 5). Separation of concerns is often enforced: the specification of the internal workings of a component is separated from the specification of its interaction with the environment. Although our work is only aimed at systems specified using rewriting logic, we have kept in mind the need to separate concerns. Indeed, our definition of composition does not require that components are specified in any particular formalism. We have called *synchronous composition* the operation that allows assembling systems keeping them synchronized. It works in three phases: the internal workings of each component are specified using rewriting logic in a fairly standard way; then, properties are defined on the states and transitions of the system; synchronization between components is then established by relating the values of properties from different components. Properties are functions that take values at states and transitions. They are defined extending, but not modifying the base system, so that the component needs not know whether it is going to be synchronized or used in isolation. This provides a high degree of separation of concerns. (We have drawn inspiration from the way Maude's model checker is used. See Clavel *et al*. 2007.)

Although our main goal is not to introduce novel ideas, it is inevitable that adapting old ideas to new settings inspires or requires new developments. What we have called *egalitarianism* consists in treating transitions as first-class citizens, the same as states. It is not a strict requirement for the developments described in this paper, but we find it convenient and have stuck to it. Also, we have found nowhere else our particular setting for composition, through synchronization based on simple equality of potentially complex properties.

Using just equality between properties for composition may seem too simple, but it allows a wide range of possibilities. The logic for complex interaction can be put into new components, representing channels or connectors, to be specified in rewriting logic like the others. Asynchronous communication, in particular, can be emulated through the use of components implementing buffers and delays. See also Section 3.4 for more examples.

This is what the reader can expect from the rest of the paper. In Section 2, we explain our motivations, our goals, and how they have driven us to particular choices and definitions. All the formalisms are in Section 3. Then, Section 4 contains a simple but complete example for illustrative purposes. Section 5 discusses related work. The next steps we intend to complete are discussed in Section 6, and conclusions are exposed in Section 7.

Our paper (Martín *et al*. 2018) contains additional examples, particularly a large one on the alternating bit protocol, aimed at showing the full power of our proposals. It also

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

describes techniques we have found helpful when specifying systems using our method. Additional material can be found on our website: http://maude.sip.ucm.es/syncprod.

2 Motivation, goals, and choices

We discuss the different choices we have been driven to make toward 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.

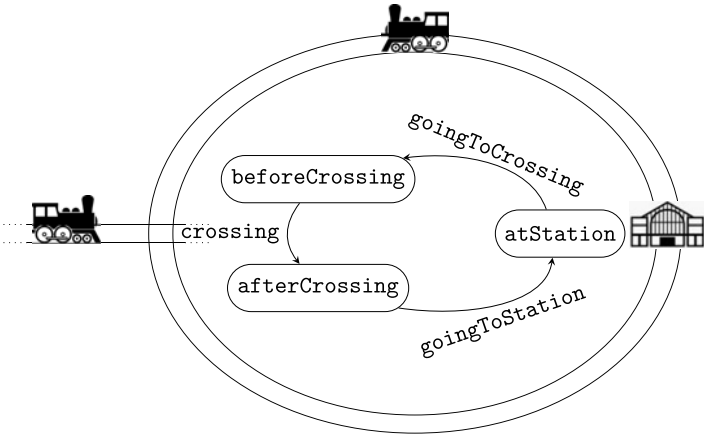
One message we want to convey in this section is that most choices were imposed on us by two goals. Our number one goal, again, is to provide flexible tools for compositionality in rewriting logic. The second is to depart as little as possible from the standard definitions of rewriting logic, so that we can benefit from all the existing machinery around it: practical tools, like model checkers, state-space explorers, and execution engines; and theoretical results about the use of abstraction, semantics, and so on.

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 modeling, specification, and analysis. This focus has also guided some of the choices described below.

As all this section is intended to be motivational, many technical details are only going to be properly explained and (hopefully) understood in later sections.

2.1 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 of 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 .
```

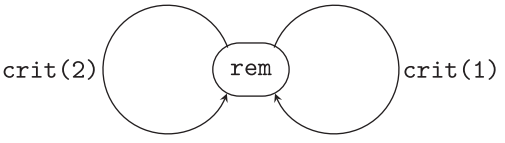
https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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 modeled the same, that share the piece of railway we have identified as `crossing`. Let us 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 modeled separately and combined afterwards. Also, the control to ensure mutual exclusion can 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)] : rem => rem .
```

This system, that we call `MUTEX`, is composed with the trains to produce a single system that we denote as `TRAIN1` $\|$ `TRAIN2` $\|$ `MUTEX`. For this composed system to work properly, we need to make sure that the transition `crossing` in system `TRAIN1` takes place synchronized with (that is, simultaneously to) transition `crit(1)` in system `MUTEX`, and `crossing` in `TRAIN2` with `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 synchronization criteria.

This opens the door to compositional verification (that only gets some passing mentions in the present paper). The system `MUTEX` satisfies mutual exclusion, represented by the linear temporal logic (LTL) formula

$$\Box(\neg \texttt{crit(1)} \vee \neg \texttt{crit(2)}),$$

where `crit(I)` is used as a proposition that holds true when the respective transition is executing, and false otherwise. Therefore, we can also assert that the composed, synchronized system satisfies mutual exclusion in the form

$$\Box(\neg \texttt{crossing}_1 \vee \neg \texttt{crossing}_2),$$

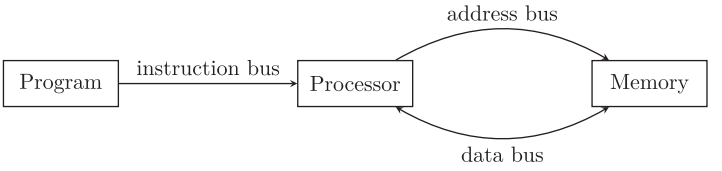
where $\texttt{crossing}_1$ represents a proposition that holds true when transition `crossing` is executing in system `TRAIN1`, and false otherwise, and similarly for $\texttt{crossing}_2$.

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

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

2.2 Second motivational example: computer architecture

We introduce now a slightly more complex example. 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.

A typical rewriting logic specification could go like this:

- the stored program is represented by 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 the 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

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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 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 behavior 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 a synchronized way. 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 .
```

Using these decoupled rules, 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. The two parts need to be synchronized. Our synchronization mechanism restricts the wild capabilities of the components. It represents the wiring or gearing in real-world systems. Such refined realism is not always needed. When it is used, each component system can be simpler and can be given independent meaning, independent specification, and, hopefully, independent analysis.

2.3 Egalitarianism

Using actions for synchronization, rather than states, is in many cases the natural choice, as illustrated in the two examples above. Thus, it is unfortunate that actions, or transitions, are often treated in a discriminatory fashion with respect to states. In the basic formulations of labeled 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.

Synchronizing states is also useful, so we do not want to adopt an action-only formalism. We need to be *egalitarian*. In Martín *et al*. (2016a), we already argued for the convenience of treating states and transitions as equals, and we proposed what we already called *egalitarian* systems. Those systems were different, though related, to the

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

ones by that adjective in the present paper. The same point had been made before, if only partially, for instance, in De Nicola and Vaandrager (1995); Kindler and Vesper (1998); Meseguer (2008). 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 Martín *et al.* (2016b), 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) 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 Meseguer (1992), 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.

2.4 What is a transition?

In our toy computer architecture example, consider 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:

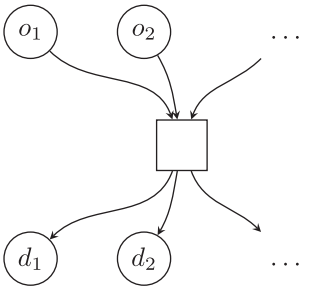
```
--- In module PROCESSOR
   rl [doingR] : ProgCounter: N
                 Instr: (r, A)
                 Data: D
              => ProgCounter: N + 1
                 Instr: void
                 Data: D' .
```

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 synchronized, 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 $\{o_1, o_2, \dots\}$ a system can reach in one step any state in a set of destination states $\{d_1, d_2, \dots\}$, it is fair to consider such steps the same transition, irrespective of the actual origin and destination states used in each actual run of the system.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press



For another example, consider 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 synchronize 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 Meseguer (1992) for rewriting logic, univocally identifies a rewriting step and contains all the information to recover the origin and destination state terms. For example, `updating(A, D'', RestOfMemory, D)` is the form of a proof term for the rule above. 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)`. It may seem odd that some of these variables are part of the transition terms, specially `RestOfMemory`, but it is the only way that the sets of possible origin and destination states can be recovered. These variables may be better interpreted as the context in which the transition is taking place, instead of being a parameter of the transition.

This feature is shared by 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 synchronization to work properly. (Interestingly, the proof terms proposed in Boudol and Castellani (1988) for CCS forget the unused part of an alternative, getting thus a little closer to our proposal.)

We said that it is *fair* to consider that some steps are instances of the same transition; we did not say it is *mandatory*. In the motivational example of Section 2.1, `crit(1)`

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

and `crit(2)` are different transitions, even though they share their unique origin and destination state, `rem`. We need them to be different transitions so that each can be synchronized to the action of a different train.

In agreement with all this, in the egalitarian transition systems we define below, transitions are represented 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*.)

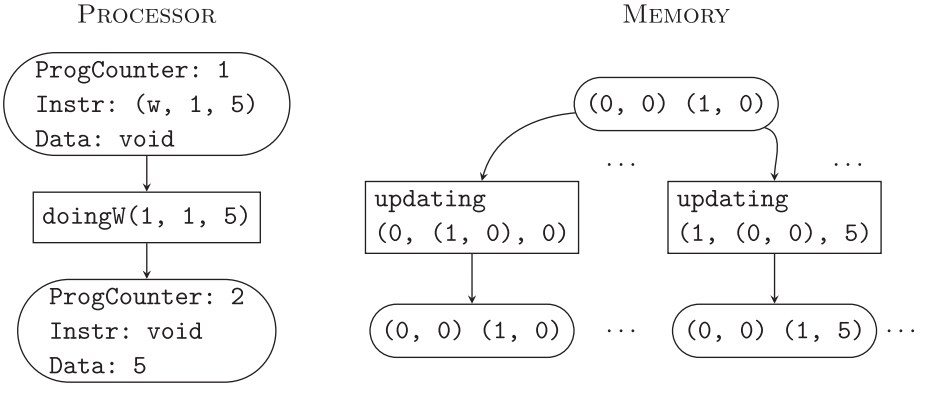
Translating all of this to rewriting logic leads us to some tweaking in the definition of the logic. It is fully described below. The idea is that, in egalitarian rewriting logic, we need to label rules, not with atomic labels, but with terms showing which parameters we have chosen to belong to the transition.

2.5 Synchronizing on properties

Consider once again the rules `doingW` and `updating` from Section 2.2, giving rise to transitions with terms of the form

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

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



In words, the processor's state is telling: that the next instruction to be executed is the one identified with number 1 in the program; that the processor has already received and stored such an instruction, which is a request to write 5 at memory address 1; and that 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 synchronized, we want the memory to execute only the transition that matches the values from the processor

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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

Thus, we use general functions on states and on transitions. We refer to these functions as *properties*. For synchronization, we need to compare the values of such properties for equality. We require their values to allow for such a comparison. No other restriction is made on the types of properties. We call *synchronization criterion* to each equality of properties required for composing two systems.

There is one more tweak: 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 synchronize (regarding that property – there may be others).

Partially defined properties are of practical interest. At states or transitions where the property is undefined, other systems can behave in any way. This can be seen as an implication: "when I say *yes*, you say *yes*; but when you say *yes*, I can say whatever, or nothing at all" (because I am allowed to be visiting a stage where the property is undefined). This is illustrated in the example in Section 4 (with the property `doMoveR`).

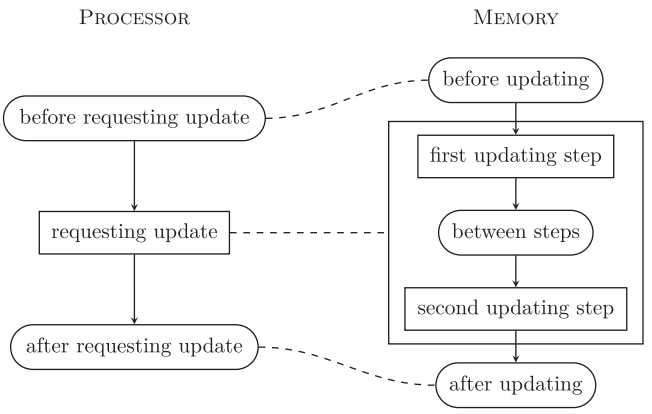
Systems with partially defined propositions have been more thoroughly studied, in a different setting, in Huth *et al.* (2001); Bruns and Godefroid (1999; 2000); Godefroid and Huth (2005), under the name of *partial structures*. These are Kripke structures on whose states propositions are defined with possible values true, false, or unknown. The *unknown* value can be used, for example, to perform model checking leaving part of the state space unexplored, or to represent a satisfiability problem as a Kripke structure in which all propositions are unknown at start.

2.6 Synchronizing states with transitions

The rule `doingW` at the processor side, as it has already been discussed, must be synchronized with whatever actions perform the real updating at the memory side. The processor cannot 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 synchronize one transition at one side with more than one at the other side, as shown in the figure, just based on the values of properties.

In our case, the *before* states on both systems must synchronize; also the *after* states must synchronize; and the unique transition at the processor side must synchronize 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 synchronization 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.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press



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. When we want to refer to either states or transitions or any composition of them, we call them *stages*.

2.7 The split

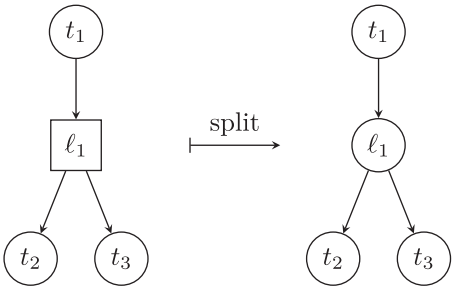
At the start of this paper, we envisioned egalitarian transition structures as represented by bipartite graphs, with states and transitions interleaved. After the discussion above, these structures are only valid as *atomic* components. Non-atomic structures are given as a set of atomic ones together with the synchronization criteria.

We have stepped into a problem. Our goal for powerful compositionality has driven us to use structures and systems (and sets of them) that dangerously depart from the standard ones. Our second 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.

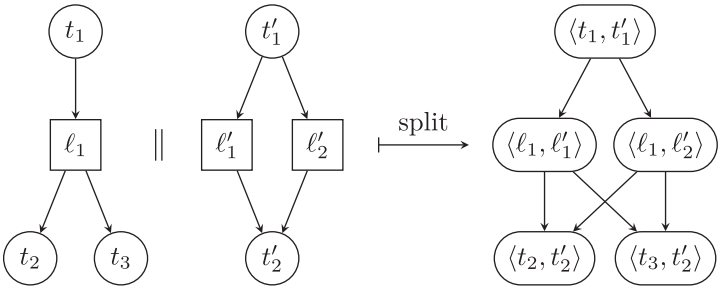
Splitting applies both to transition structures and to rewrite systems, either atomic or composed. Splitting a structure or a system produces a standard one with equivalent information and behavior. Splitting a structure or a system consists in making each state, transition, or whatever mixture of them in the original composed system into a state (of a new sort) 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 labeled rewrite rule like $\ell : t \to t'$ in an egalitarian rewrite system becomes split as $t \to \ell$ and $\ell \to t'$. What was a rule label, ℓ, becomes an intermediate state term in the split system.

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

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press



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 synchronized only with states, and transitions only with transitions:



States like $\langle t_1, \ell'_1 \rangle$ are not valid in this case, according to the properties we are assuming for synchronization. There are three more valid states in the result: $\langle t_1, t'_2 \rangle$, $\langle t_2, t'_1 \rangle$, and $\langle t_3, t'_1 \rangle$; these are not reachable from the ones shown.

Ideally, we would have at our disposal execution engines and verification tools that work in a distributed, compositional way on egalitarian systems. This is not the case at present. The intended use of the split operation is to allow us to translate problems posed on egalitarian structures and systems to standard split ones. For instance, temporal properties would be translated and Maude's model checker can be used on the split system to draw conclusions about the original egalitarian one. The detailed way to perform these formula translations will be the subject of a future paper.

2.8 On concurrency and topmostness

True concurrency is a given in rewriting logic. When two rewrite rules refer to disjoint subterms of the state term, they can be executed at the same time. A rewrite system is said to be *topmost* if rewrites can only happen on the whole state term. Topmostness prevents concurrency: if each rule uses the whole state term, there is no room left for another rule to act concurrently on a different subterm. We revisit topmostness in Section 3.

In our setting, concurrency has one additional level. In a composed system, rules from different components can be executed at the same time, irrespective of whether each

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

atomic component happen to be topmost or not. 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 is natural if we are talking about actions taking place in different components.

Indeed, we prefer that our atomic systems are topmost. 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:

```
(a, b) => (a', b') .
a => 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

```
a => a' .
a => a'' .
```

and another for the `b` part, with rule

```
b => b' .
```

We need to add the synchronization criteria to make rules `a => a'` and `b => b'` be run only simultaneously, so as to produce the same result as the initial composed rule. We have found this method useful when specifying our examples, and we expect it to allow to transform many interesting systems into synchronous compositions of topmost ones.

2.9 Explicit synchronization criteria

In automata theory and in process algebras, it is common to define the synchronous composition so that the components execute the same action at the same time. Here, "the same action" means "an action with equal name." Instead of relying on implicit understandings, we prefer to be explicit about which properties synchronize with which others in our synchronous compositions. The way we have chosen is the following. A synchronous composition is denoted as $\mathcal{S}_1 \|_Y \mathcal{S}_2$. It has a parameter in addition to the systems being composed: a set Y of *synchronization criteria* specifying properties to synchronize on.

Each criterion is given as a pair (p_1, p_2), with p_1 a property from $\mathcal{S}_1$ and p_2 from $\mathcal{S}_2$. Each criterion (p_1, p_2) is a requirement that, at every moment, the value of p_1 at the current state or transition in system $\mathcal{S}_1$ is the same as the value of p_2 at the current state or transition in system $\mathcal{S}_2$.

In the example about mutual exclusion from Section 2.1, we would define a property `grants(1)` in `MUTEX` as true in the transition `crit(1)` and false otherwise. At `TRAIN1`, we would define a property `isCrossing` to be true at its transition `crossing` and false at all other transitions and states. Then, we obtain the desired result by requiring the systems to synchronize on those properties. It is often the case that the same property seen from different sides deserves different, descriptive names. Also, it helps reusability that the `MUTEX` system does not use train-related names.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

A composition with no synchronization is acceptable, that is, with $Y = \emptyset$. It means that each component can evolve with no consideration to what the other one is doing. Also, the same property can be used several times: $Y = \{(p_1, p_2), (p_1, p_2')\}$. In Section 3, we define the synchronous composition as a binary operation, but then justify expressions in which more than two systems can be composed at once.

2.10 Properties of the composed system

A way is needed to assign properties to composed systems, so that they can be used as components in turn. Although we have mentioned encapsulation as a desirable ingredient of compositionality, we do not show in this paper the means to facilitate it. (We see in parameterized programming an ideal tool to deal with it in rewriting logic, and we have detailed this proposal in Martín *et al.* (2018).) For the purpose of the present paper, any property p_1 defined in a system $\mathcal{S}_1$ is automatically a property of any composed system of which $\mathcal{S}_1$ is a component, $\mathcal{S}_1 \|_Y \mathcal{S}_2$, even if p_1 is already used in Y.

We insist that different systems are different namespaces, so that two properties that happen to get the same name in different systems are different properties. We use the name of the system as a subscript if disambiguation is needed: $p_{\mathcal{S}}$ for property p defined in system $\mathcal{S}$.

2.11 Summing up

We have up to now described our motivation, 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*, maybe 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 in all states and transitions. Then, synchronization criteria are specified as pairs of properties whose values, if defined, must be equal for all component systems at all times. The resulting composed systems can, in their turn, be composed. Properties are inherited by the composed systems from each of its components. This is all formalized in the next section, both for transition structures and for rewrite systems.

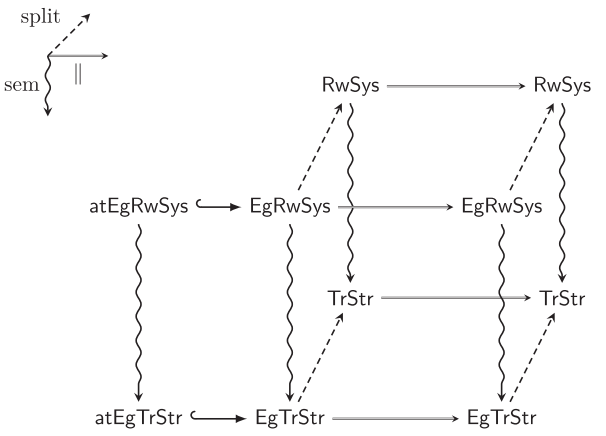
3 Formal definitions and results

We define and use below a number of structures and systems to which we give special names and abbreviations. This is a list of them:

atEgRwSys: atomic egalitarian rewrite systems
EgRwSys: egalitarian rewrite systems
RwSys: plain rewrite systems
atEgTrStr: atomic egalitarian transition structures
EgTrStr: egalitarian transition structures
TrStr: plain transition structures.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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. It is argued below that each face of that polyhedron contains a commutative diagram (namely, see Definitions 5 and 10, Proposition 5, Theorem 1, and Section 3.2.4).

3.1 Transition structures

We are interested in three types of transition structures. In atomic egalitarian transition structures, transitions have independent entity; they are, otherwise, similar to Kripke structures and other known formalisms. Then, non-atomic egalitarian transition structures are sets of atomic ones evolving in a synchronized way according to the given criteria. Finally, plain structures are useful as a non-egalitarian but quite standard equivalent to egalitarian ones.

3.1.1 Atomic egalitarian transition structures

We denote by atEgTrStr the class of *atomic egalitarian transition structures*. An element of atEgTrStr contains two kinds of nodes, called states and transitions, that can only occur interleaved.

Formally, an atomic egalitarian transition structure is a tuple $\mathcal{T} = (Q, T, \rightarrow, P, g^0)$, where:

- Q is the set of states (we reserve the letter S for later use);
- T is the set of transitions;
- $\rightarrow \subseteq (Q \times T) \cup (T \times Q)$ is the bipartite adjacency relation;
- P is the set of properties – partial functions defined on states and transitions, with possibly different codomains: $p : Q \cup T \nrightarrow C_p$ (using C_p to denote the codomain of property p);
- $g^0 \in Q \cup T$ is the initial state or transition.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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. We introduce the word *stage* to refer to either a state or a transition. We use variables typically called g, with or without subscripts or superscripts, to range over $Q \cup T$, that is, over stages. Thus, g^0 is the initial stage.

These structures can be seen as generalizations of other, well-known ones. Standard transition structures are obtained as the particular cases in which each transition has a unique origin and a unique destination, that is, in which the restricted relations $\rightarrow|_{T\times Q}$ and $\rightarrow^{-1}|_{T\times Q}$ are functions. Labeled transition structures are then obtained by adding a single property $L : T \rightarrow \Lambda$ (undefined on states), for some alphabet of actions Λ. Similarly, Kripke structures are obtained by adding instead a single property $L' : Q \rightarrow 2^{\mathrm{AP}}$ (undefined on transitions), for some set of atomic propositions AP. As discussed in Section 2.4, Petri nets are different to atomic transition structures. Even though they are both bipartite graphs, their runs are different. There is no concept of marking in our structures, no distributed state. Our means to get simultaneous firing of transitions is to compose several atomic structures, as we show in the next section.

3.1.2 Egalitarian transition structures

The class of *egalitarian transition structures*, $\mathsf{EgTrStr}$, is built up from atomic transition structures by performing pairwise synchronous compositions. Formally, we define egalitarian transition structures and, simultaneously, their sets of properties by:

- if $\mathcal{T} = (Q, T, \rightarrow, P, g^0) \in \mathsf{atEgTrStr}$, then $\mathcal{T} \in \mathsf{EgTrStr}$, with P being its set of properties;
- given $\mathcal{T}_1, \mathcal{T}_2 \in \mathsf{EgTrStr}$ (atomic or otherwise), with respective sets of properties P_1 and P_2, and given a relation $Y \subseteq P_1 \times P_2$ (the synchronization criteria), the expression $\mathcal{T}_1 \| _Y \mathcal{T}_2$ denotes a new element of $\mathsf{EgTrStr}$, called the *synchronous composition of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *with respect to* Y, whose set of properties is defined to be $P_1 \uplus P_2$.

For each synchronization criterion $(p_1, p_2) \in Y$, we require that the codomain sets of p_1 and p_2 are the same, $C_{p_1} = C_{p_2}$, so that their values can be compared for equality. The class $\mathsf{EgTrStr}$ is the smallest one obtained by applying these two rules.

All properties from $\mathcal{T}_1$ and from $\mathcal{T}_2$ are automatically properties of their synchronous composition. Ultimately, all properties are defined in atomic structures, and inherited by the composed structures in which they take part. We assume that we have access to properties defined in any atomic component of an egalitarian transition structure, at any level of nesting.

A note on namespaces: we see each egalitarian transition structure as an independent system. Thus, it may happen, as if by chance, that a property in $\mathcal{T}_1$ has the same name as another in $\mathcal{T}_2$, but they would be different properties anyway. That is why we have used the disjoint union symbol above. When disambiguation is required, we use the name of the structure as a subscript: $p_{\mathcal{T}}$, $p_{\mathcal{T}_1}$. Also the sets of states and transitions are assumed disjoint between different structures.

The behavior of each atomic component is restricted by the behavior of the other components according to the synchronization criteria. Intuitively, the atomic component

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

$\mathcal{A}$ is allowed to visit its stage g at a given moment in time only if, simultaneously, each *neighbor* $\mathcal{A}'$ of $\mathcal{A}$ is visiting a stage g' that is *compatible* with g. By a *neighbor* of $\mathcal{A}$, we mean any $\mathcal{A}'$ for which there is a criterion in Y involving a property from $\mathcal{A}$ and another from $\mathcal{A}'$; stages g and g' are *compatible* if all the criteria are satisfied when the properties are evaluated at them. By the way, that stages g and g' 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: the stage h, in the same atomic component as g, may also be compatible with g'. Each execution of the system must non-deterministically choose one of the allowed ways to go. It is always possible to define properties in a tighter way to achieve the desired behavior.

We could define concepts of global stage and global transition for the composed system, but we prefer the *distributed* view outlined in the previous paragraph. We consider those structures as models of distributed systems. In a distributed system, with components that may even reside in different locations, the concepts of global state and global transition are unimportant, and we avoid them in this work as much as possible (although the split operation defined in Section 3.1.4 provides them, indirectly but explicitly).

Thus, strictly speaking, a non-atomic egalitarian transition structure is *not* a transition structure, in the traditional sense of having states and an adjacency relation on them. The behavior of $\mathcal{T}_1 \|_Y \mathcal{T}_2$ results from the interaction, as given by Y, of the behaviors of $\mathcal{T}_1$ and $\mathcal{T}_2$ and, ultimately, of the behaviors of their atomic subcomponents.

We do not need to make formal that intuition of simultaneous stages. We do need, however, a more restricted notion of equivalence: the one given by the different ways of composing the same components. For example, $(\mathcal{T}_1 \|_Y \mathcal{T}_2) \|_{Y'} \mathcal{T}_3$ and $\mathcal{T}_1 \|_{Y''} (\mathcal{T}_2 \|_{Y'''} \mathcal{T}_3)$ are equivalent if $Y \cup Y' = Y'' \cup Y'''$. Let us make this formal.

Definition 1
The *set of atomic components* of an egalitarian transition structure is:

- $\text{atomic}(\mathcal{T}) = \{\mathcal{T}\}$ if $\mathcal{T} \in \mathsf{atEgTrStr}$,
- $\text{atomic}(\mathcal{T}_1 \|_Y \mathcal{T}_2) = \text{atomic}(\mathcal{T}_1) \cup \text{atomic}(\mathcal{T}_2)$.

Definition 2
The *total set of criteria* of an egalitarian transition structure is:

- $\text{criteria}(\mathcal{T}) = \emptyset$ if $\mathcal{T} \in \mathsf{atEgTrStr}$,
- $\text{criteria}(\mathcal{T}_1 \|_Y \mathcal{T}_2) = \widetilde{Y} \cup \text{criteria}(\mathcal{T}_1) \cup \text{criteria}(\mathcal{T}_2)$,

where $\widetilde{Y} = \{\{x, y\} \mid \langle x, y\rangle \in Y\}$.

We need to make the tuples from Y into the two-element sets in $\widetilde{Y}$ so that $\widetilde{Y} = \widetilde{Y^{-1}}$ and then $\text{criteria}(\mathcal{T}_1 \|_Y \mathcal{T}_2) = \text{criteria}(\mathcal{T}_2 \|_{Y^{-1}} \mathcal{T}_1)$.

Definition 3
Two egalitarian transition structures $\mathcal{T}_1$ and $\mathcal{T}_2$ are *equivalent*, denoted as $\mathcal{T}_1 \equiv \mathcal{T}_2$, iff $\text{atomic}(\mathcal{T}_1) = \text{atomic}(\mathcal{T}_2)$ and $\text{criteria}(\mathcal{T}_1) = \text{criteria}(\mathcal{T}_2)$.

In the particular case that $\mathcal{T}_1$ and $\mathcal{T}_2$ are atomic, we have $\mathcal{T}_1 \equiv \mathcal{T}_2 \Leftrightarrow \mathcal{T}_1 = \mathcal{T}_2$.

This equivalence relation can be seen as expressing a concept of *same behavior*. Equivalent structures are indistinguishable, and we consider them as representing the same

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

system. This allows us to group the atomic components in the most suitable way for a modular design. Consequently, we allow expressions like $\|_Y{}_{i=1}^{n}\mathcal{T}_i$, or $\|_{Y,I}\mathcal{T}_i$ for some index set I, or just $\|_Y\mathcal{T}_i$ if indexes are implied. They stand for any representative of a $\equiv$-equivalence class.

3.1.3 Plain transition structures

What we call *plain transition structures* are, in essence, traditional transition structures on whose states we define properties. These are non-egalitarian – transitions are just given by the adjacency relation between states and do not play any special role. The reason we introduce these is that, as we show in the next section, egalitarian transition structures can be translated into plain ones, with some benefits.

Formally, a plain transition structure is a tuple $\mathcal{T} = (Q, \to, P, q^0)$, where:

- Q is the set of states;
- $\to \subseteq Q \times Q$ is the adjacency relation;
- P is the set of properties – partial functions defined on states, with possibly different codomains: $p : Q \nrightarrow C_p$;
- $q^0 \in Q$ is the initial state.

The class of plain transition structures is denoted by TrStr.

The synchronous composition of two plain transition structures $\mathcal{T}_i = (Q_i, \to_i, P_i, q_i^0) \in \mathsf{TrStr}$, for $i = 1, 2$, with respect to a relation $Y \subseteq P_1 \times P_2$ (the synchronization criteria), is denoted by $\mathcal{T}_1 \|_Y \mathcal{T}_2$ and is defined to be the plain transition structure $\mathcal{T} = (Q, \to, P, q^0)$, where:

- Q is the set of pairs $\langle q_1, q_2\rangle \in Q_1 \times Q_2$ such that, for each $(p_1, p_2) \in Y$, either $p_1(q_1)$ is undefined, or $p_2(q_2)$ is undefined, or $p_1(q_1) = p_2(q_2)$;
- for $\langle q_1, q_2\rangle, \langle q_1', q_2'\rangle \in Q$, we have $\langle q_1, q_2\rangle \to \langle q_1', q_2'\rangle$ iff either $q_1 \to_1 q_1'$ and $q_2 \to_2 q_2'$, or $q_1 \to_1 q_1'$ and $q_2 = q_2'$, or $q_1 = q_1'$ and $q_2 \to_2 q_2'$;
- $P = P_1 \uplus P_2$ and, if p is a property defined in $\mathcal{T}_i$ ($i = 1$ or 2), then it is defined in $\mathcal{T}$ by $p(\langle q_1, q_2\rangle) = p(q_i)$ or, with qualification to avoid ambiguities, $p_{\mathcal{T}}(\langle q_1, q_2\rangle) = p_{\mathcal{T}_i}(q_i)$;
- $q^0 = \langle q_1^0, q_2^0\rangle$, assumed to be in Q (i.e., to satisfy the criteria in Y).

Definition 4
Given two plain transition structures $\mathcal{T}_i = (Q_i, \to_i, P_i, q_i^0) \in \mathsf{TrStr}$, for $i = 1, 2$, we say that they are *equivalent*, and denote it by $\mathcal{T}_1 \equiv \mathcal{T}_2$, if there exist two bijections, $f : Q_1 \to Q_2$ and $F : P_1 \to P_2$, such that, for all $q_1, q_1' \in Q_1$ and all $p_1 \in P_1$:

- the adjacency relation is preserved: $q_1 \to_1 q_1' \Leftrightarrow f(q_1) \to_2 f(q_1')$;
- codomains for properties are preserved: $C_{p_1} = C_{F(p_1)}$;
- values of properties are preserved: $p_1(q_1) = F(p_1)(f(q_1))$;
- initial values are preserved: $f(q_1^0) = q_2^0$.

For example, $\mathcal{T}_1 \|_Y \mathcal{T}_2 \equiv \mathcal{T}_2 \|_{Y^{-1}} \mathcal{T}_1$. More in general, different ways to group, reorder, and compose n given structures, distributing and swapping the synchronization criteria accordingly, produce equivalent structures. Thus, as in the previous section, we allow the expressions $\|_Y{}_{i=1}^{n}\mathcal{T}_i$, or $\|_{Y,I}\mathcal{T}_i$, or $\|_Y\mathcal{T}_i$. Also, we denote by $\langle q_1, \ldots, q_n\rangle$ a typical element in $\|_Y\mathcal{T}_i$.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

The operators "$\|$" defined for TrStr and for EgTrStr are related (see next section), but are of a different nature: $\mathcal{T}_1 \|_Y \mathcal{T}_2$ is an irreducible expression in EgTrStr, but can be reduced in TrStr to a single transition structure. This is, indeed, one of the advantages of plain structures: they are always single graphs, even when they are the result of a composition.

In the next section, we show how plain transition structures are useful as a more standard equivalent to egalitarian ones. By *more standard*, we mean that they are almost Kripke structures, except that we use properties instead of simply Boolean propositions. Thus, for example, the satisfiability of temporal formulas can be defined for plain transition structures in the same way as for Kripke structures, just replacing Boolean propositions by Boolean-valued properties. Finally, if one does not care about egalitarianism, plain transition structures can be taken as a basis for discussing non-egalitarian compositional specification.

3.1.4 The split and global states

The *split* is a translation of transition structures: split : EgTrStr $\to$ TrStr. The definition is quite straightforward, but its validity relies on a lemma, proved below, to ensure that the sets of properties of $\mathcal{T}$ and of split($\mathcal{T}$) are the same.

Definition 5

- Given $\mathcal{T} = (Q, T, \to, P, g^0) \in$ atEgTrStr, its split is split($\mathcal{T}$) $= (Q \cup T, \to, P, g^0)$, that is, stages are transformed into states.
- For a non-atomic egalitarian structure $\mathcal{T}_1 \|_Y \mathcal{T}_2$, its *split* is recursively defined by split($\mathcal{T}_1 \|_Y \mathcal{T}_2$) = split($\mathcal{T}_1$)$\|_Y$ split($\mathcal{T}_2$).

Lemma 1
Denoting by ppt($\mathcal{T}$) the set of properties of either an egalitarian or plain transition structure $\mathcal{T}$, we have ppt($\mathcal{T}$) = ppt(split($\mathcal{T}$)). Thus, it is valid the use of the same Y on both sides of the equality "split($\mathcal{T}_1 \|_Y \mathcal{T}_2$) = split($\mathcal{T}_1$)$\|_Y$ split($\mathcal{T}_2$)" in the definition.

Proof
In the base case, trivially ppt($\mathcal{T}$) = ppt(split($\mathcal{T}$)) = P.

Inductively assuming that ppt($\mathcal{T}_i$) = ppt(split($\mathcal{T}_i$)), for $i = 1, 2$, we have:

$$\begin{aligned} \text{ppt}(\text{split}(\mathcal{T}_1 \|_Y \mathcal{T}_2)) &= \text{ppt}(\text{split}(\mathcal{T}_1) \|_Y \text{split}(\mathcal{T}_2)) \\ &= \text{ppt}(\text{split}(\mathcal{T}_1)) \cup \text{ppt}(\text{split}(\mathcal{T}_2)) \\ &= \text{ppt}(\mathcal{T}_1) \cup \text{ppt}(\mathcal{T}_2) \\ &= \text{ppt}(\mathcal{T}_1 \|_Y \mathcal{T}_2). \end{aligned}$$

□

Proposition 1
For $\mathcal{T}_1, \mathcal{T}_2 \in$ EgTrStr, we have $\mathcal{T}_1 \equiv \mathcal{T}_2 \Rightarrow$ split($\mathcal{T}_1$) $\equiv$ split($\mathcal{T}_2$).

Proof
Given that $\mathcal{T}_1 \equiv \mathcal{T}_2$ implies atomic($\mathcal{T}_1$) = atomic($\mathcal{T}_2$), either $\mathcal{T}_1$ and $\mathcal{T}_2$ are both atomic or none is. In the first case, we must have $\mathcal{T}_1 = \mathcal{T}_2$, and the result is immediate. In

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

the second case, the result follows easily from the definitions. For example, the states of split($\mathcal{T}_1$) and those of split($\mathcal{T}_2$) are in bijection because both are given by different ways of making pairs with one stage from each component in atomic($\mathcal{T}_1$) = atomic($\mathcal{T}_2$), restricted by the same criteria criteria($\mathcal{T}_1$) = criteria($\mathcal{T}_2$). □

The split operation can be extended in the usual way to work on the equivalence classes considered in previous sections. Thus, using the notation $[\|_{Y,I}\mathcal{T}_i]$ to represent $\equiv$-equivalence classes, we have split($[\|_{Y,I}\mathcal{T}_i]$) $\subseteq [\|_{Y,I}\,\text{split}(\mathcal{T}_i)]$, as a restatement of Proposition 1.

The states in split($\mathcal{T}$) can be seen as global states for the egalitarian structure $\mathcal{T}$. Also, the adjacency relation in split($\mathcal{T}$) can be seen as representing global transitions of $\mathcal{T}$ in the following precise sense.

Proposition 2
Take $\mathcal{T} = \|_{Y,I}\mathcal{A}_i \in \mathsf{EgTrStr}$ with each $\mathcal{A}_i$ atomic, so that split($\mathcal{T}$) = split($\|_{Y,I}\mathcal{A}_i$) = $\|_{Y,I}\,\text{split}(\mathcal{A}_i)$. Let g_i and g_i' denote stages (not necessarily different) of $\mathcal{A}_i$ for each $i \in I$. We have that:

- $\langle g_1, \ldots, g_n\rangle$ is a state in split($\mathcal{T}$) iff g_i and g_j are compatible as stages of $\mathcal{A}_i$ and $\mathcal{A}_j$ for all $i, j \in I$, that is, the properties defined at them satisfy all the criteria in Y;
- the step $\langle g_1, \ldots, g_n\rangle \to \langle g_1', \ldots, g_n'\rangle$ is valid in split($\mathcal{T}$) iff (both are states in split($\mathcal{T}$), and) for each i, either $g_i = g_i'$ or $g_i \to g_i'$ is a valid step in $\mathcal{A}_i$, with at least one instance of the latter.

Proof
Straightforward from the definitions of $\|$ and the split. □

This opens interesting possibilities for verification. Very roughly, the idea is that proving $\mathcal{A}_i \models \Phi_i$ for all i, where the Φ_i are temporal formulas, may be enough to prove split($\|_Y\mathcal{A}_i$) $\models \Phi$, for some formula Φ built based on the Φ_i and Y. These issues are left for future work. See also Section 6.3.

3.2 Rewrite systems

We are interested in three variants of rewrite systems. Atomic egalitarian rewrite systems are similar to standard ones, but rule labels are replaced by complex terms, with the same importance as state terms. Then, non-atomic egalitarian rewrite systems are sets of atomic ones interacting through given synchronization criteria. Finally, plain rewrite systems are useful as a non-egalitarian but quite standard equivalent to egalitarian ones.

The exposition in this section reflects and, in part, repeats the one in Section 3.1 concerning transition structures.

3.2.1 Atomic egalitarian rewrite systems

We slightly modify the standard definition of rewrite systems from Meseguer (1992) to accommodate in it the kind of transitions we are interested in. We call the result *atomic egalitarian rewrite systems* and denote their class by $\mathsf{atEgRwSys}$. Non-atomic systems are defined below as the composition of atomic ones.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

An atomic egalitarian rewrite system is a tuple $\mathcal{R} = (S, \leq, \Sigma, E, M, R)$, where:

- $(S, \leq, \Sigma)$ is an order-sorted signature. Concretely, $(S, \leq)$ is a partial order of sorts and subsorts, and Σ is a set of operator declarations of the form $f : s_1 \times \cdots \times s_n \to s$ for $s_1, \ldots, s_n, s$ sorts in S. The order-sorted signature $(S, \leq, \Sigma)$ can be *kind-completed* to $(\hat{S}, \hat{\leq}, \hat{\Sigma})$, by adding to it exactly the following:
  - for each weakly connected component k of $(S, \leq)$, include in $\hat{S}$ an element, that we call k as well, with $s \,\hat{\leq}\, k$ for all sorts s in k; we denote by $[s]$ the unique such an element with $s \,\hat{\leq} [s]$, called the *kind* of s;
  - for each operator $f : s_1 \times \cdots \times s_n \to s$ in Σ, include in $\hat{\Sigma}$ an overloaded operator *at the kind level* $f : [s_1] \times \cdots \times [s_n] \to [s]$.

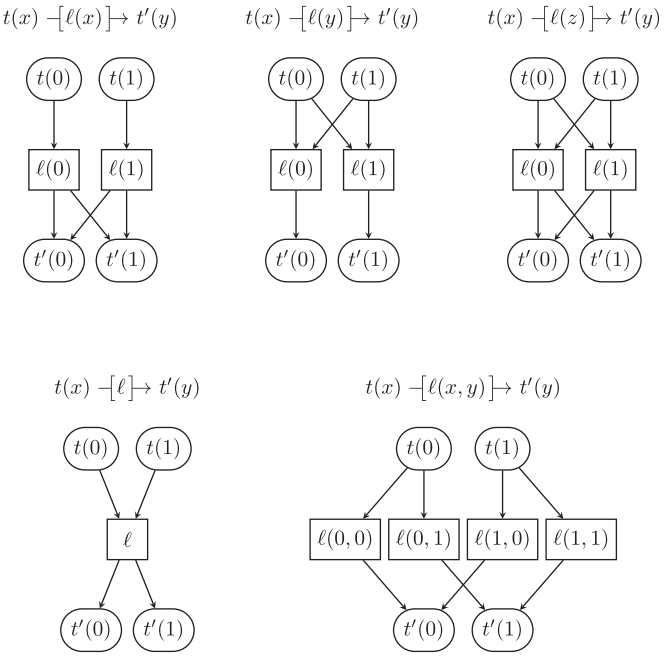
  In the rest of the paper, we assume without mention that all the order-sorted signatures we use are already the result of a kind completion. The set T_Σ of ground terms is defined in the usual way, as is the set $T_\Sigma(X)$ of terms over a given set of S-sorted variables, $X = \{X_s\}_{s \in S}$. Also, $T_{\Sigma,s}$ and $T_\Sigma(X)_s$ denote the sets of terms of sort s.
  We need atomic rewrite systems to be *topmost*, that is:
  - there are two distinguished sorts in S, that we assume called `State` and `Trans`, to represent states and transitions, respectively; for convenience, we also assume a supersort of both called `Stage` whose elements we name *stages*;
  - the operators and signatures in Σ are such that they do not allow a term of sort `Stage` to include a proper subterm of this same sort.
- E is a set of (possibly conditional) equations. Each equation has the form "$t = t'$ if C" or "$t = t'$," where $t, t' \in T_\Sigma(X)$ and t and t' are in the same kind, that is, $[\mathrm{ls}(t)] = [\mathrm{ls}(t')]$, where $\mathrm{ls}(u)$ denotes the *least sort* of the term u in $(S, \leq)$. The optional condition C is a conjunction of equational and membership conditions $C = \left(\bigwedge_i t_i = t'_i\right) \wedge \left(\bigwedge_j t_j : s_j\right)$, where, for all i and j, we have $t_i, t'_i, t_j \in T_\Sigma(X)$, $s_j \in S$, $[\mathrm{ls}(t_i)] = [\mathrm{ls}(t'_i)]$, and $[\mathrm{ls}(t_j)] = [s_j]$. A membership condition "$t_j : s_j$" represents the requirement that the term t_j has sort s_j.
- M is a set of (possibly conditional) membership axioms. Each membership axiom has the form "$t : s$ if C" or "$t : s$," where $t \in T_\Sigma(X)$, $s \in S$, $[\mathrm{ls}(t)] = [s]$, and the optional condition C is as described in the previous item. The axiom "$t : s$" states that the term t has sort s.
- R is a set of (possibly conditional) rewrite rules. Each rewrite rule has the form "$t \,-[\ell]\!\to\, t'$ if C" or "$t \,-[\ell]\!\to\, t'$," where $t, t' \in T_\Sigma(X)_{\texttt{State}}$, $\ell \in T_\Sigma(X)_{\texttt{Trans}}$, and the condition C is as above. (We do not consider rewrite conditions for the time being.)

The transition term ℓ replaces the rule labels from standard rewrite systems. We write it in the middle of the rule, both in mathematical notation, as above, and in Maude-like code in the examples from now on. The fact that ℓ is a term, and not an atomic identifier, is the main difference between standard and atomic egalitarian rewrite systems. Constructors for transition terms must be declared in Σ, the same as all other operators. Transition terms can be used in equations and in membership axioms as any other valid terms.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

In all cases, unconditional instances can be considered as shortcuts for conditional ones with the condition C being "true," which simplifies the presentation of what follows.

The choice of the transition term in a rule and, in particular, of the variables in it is a concern for the specifier, with semantic consequences. For a schematic illustration, consider the rule $t(x) \multimap\!\!\left[\,\ldots\,\right]\!\!\to t'(y)$, where x and y are the only variables in t and t', respectively. We depict below the semantics for five possible transition terms. In each transition term, 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 $t(0)$ or $t(1)$, in one step, any of $t'(0)$ or $t'(1)$ can be reached. However, the importance of transitions in modeling systems is one of the ideas put forward in this work. It is specially discussed in Sections 2.3 and 2.4.

The rewrite relation induced by an atomic egalitarian rewrite system is somewhat tricky. At first sight, the natural definition may seem to be this one, involving three terms at once:

> $u \xrightarrow{v}_{\mathcal{R}} u'$ holds for terms $u, v, u' \in T_\Sigma$ iff there are a rule "$t \multimap\!\!\left[\ell\right]\!\!\to t'$ if C" in R, and a substitution θ such that $\mathcal{R} \models C\theta$, $u =_E t\theta$, $v =_E \ell\theta$, and $u' =_E t'\theta$ (where the relation $=_E$ is the one induced by the equations in E).

This definition, however, is not appropriate for reasons introduced above: first, that would allow information to be passed from the origin to the destination states that is not present in the transition, as in the rule $x \multimap\!\!\left[a\right]\!\!\to x$ for constant a and variable x (see also Section 2.4); second, these *instantaneous* transitions would make it difficult to synchronize states with transitions, as we sometimes need (see Section 2.6); third, we lose the symmetry between the roles played by states and transitions required by egalitarianism.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Instead, we propose that each instance of a rule gives rise to two *half rewrites*. We need to devote a few paragraphs to discuss this point.

Definition 6 (*half-rewrite relation*)
Given an atomic egalitarian rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R)$ and two terms $u, v \in T_\Sigma$, we define the *half-rewrite* relation $u \to^{\text{eg}}_{\mathcal{R}} v$ to hold if either:

- there are a rule "$t \dashv\!\lfloor \ell \rfloor\!\mapsto t'$ if C" in R, and a substitution $\theta_1 : \text{vars}(\ell, t, t', C) \to T_\Sigma$ such that $\mathcal{R} \models C\theta_1$, and $u =_E t\theta_1$, and $v =_E \ell\theta_1$; or
- there are a rule "$t \dashv\!\lfloor \ell \rfloor\!\mapsto t'$ if C" in R and a substitution $\theta_2 : \text{vars}(\ell, t, t', C) \to T_\Sigma$ such that $\mathcal{R} \models C\theta_2$, and $u =_E \ell\theta_2$, and $v =_E t'\theta_2$.

The first half rewrite can be used to take the system from a state to a transition; the second, from a transition to a state. Both ornaments on the arrow, $^{\text{eg}}$ and $_{\mathcal{R}}$, are usually omitted when they are implied from the context.

Different substitutions can be used for each half rewrite. The substitutions θ_1 and θ_2, when used for two consecutive half rewrites, necessarily agree on $\text{vars}(\ell)$, but other variables can get different instantiations. This formalizes the fact that, as soon as a transition starts executing, it forgets which particular state it came from. Thus, no data can be passed from the origin state to the destination state, unless it is present in the transition term as well. The scope of variables is limited to half rewrites. A consequence is that the rule "$t(x) \dashv\!\lfloor \ell \rfloor\!\mapsto t'(x)$," where $x \notin \text{vars}(\ell)$, is valid, but its semantics may not be the one expected: the two occurrences of x are interpreted as different variables, and it may well be that $\theta_1(x) \neq \theta_2(x)$. Indeed, that rule is equivalent to "$t(x) \dashv\!\lfloor \ell \rfloor\!\mapsto t'(y)$" (whose semantics was depicted above). This is counterintuitive, so we prefer that our specifications do not contain such rules. We say that a rule is *readable* when all occurrences of a variable can be interpreted as the same variable.

Definition 7 (*readability*)
Consider an atomic egalitarian rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R)$ and a rule in it "$t \dashv\!\lfloor \ell \rfloor\!\mapsto t'$ if C." We say that the rule is *readable* iff for every two consecutive half rewrites $u \to v$ and $v \to w$ performed using that rule with substitutions θ_1 and θ_2, respectively, the same two half rewrites can be obtained using the same substitution for both. An atomic egalitarian rewrite system is *readable* if all its rules are.

According to this definition, the rule "$x \dashv\!\lfloor a \rfloor\!\mapsto x$," for constant a and integer variable x, is not readable, because it allows the two half rewrites $3 \to a \to 4$ that cannot be obtained with a single substitution for x. In contrast, the rule "$x \dashv\!\lfloor a \rfloor\!\mapsto z$" is readable, as is "$x \dashv\!\lfloor x \rfloor\!\mapsto x$." The rule "$x \dashv\!\lfloor y \rfloor\!\mapsto z$ if $x = z$," for variables x, y, and z, is not readable. The similar "$x \dashv\!\lfloor y \rfloor\!\mapsto z$ if $x = y \wedge x = z$" is indeed readable, though we would prefer the equivalent "$x \dashv\!\lfloor y \rfloor\!\mapsto z$ if $x = y \wedge y = z$," that does not directly relate x and z.

Fortunately, every atomic egalitarian system can be transformed into an equivalent readable one, as shown below. In practice, we prefer readable systems, but our theory treats them all the same, either readable or not. We do not require that a readable rule is always used with the same substitution for two consecutive half rewrites, but just that it *could* be used that way. The point is that some variables in a readable rule are only important for a half rewrite, and can be given any values for the other. For example, the

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

readable rule "$x \multimap\!\!\left[a\right]\!\!\rightarrow z$" allows rewriting $4 \to a \to 5$ with $\theta_1(x) = 4$ and $\theta_2(y) = 5$, choosing whatever values for $\theta_1(y)$ and $\theta_2(x)$, in particular, it is valid to choose 5 and 4, to make $\theta_1 = \theta_2$.

Proposition 3
Every atomic egalitarian rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R)$ can be transformed into a readable one $\mathcal{R}' = (S, \leq, \Sigma, E, M, R')$ (where only the set of rules has changed) that induces the same half-rewrite relation.

Proof
We will show that each rule "$t \multimap\!\!\left[\ell\right]\!\!\rightarrow t'$ if C" in R can be transformed into a readable one. The first step in the transformation is the following: for each variable, say x, in $\mathrm{vars}(t) \cap \mathrm{vars}(\ell)$, we choose two fresh variables, say x_t and x_ℓ, and replace t by $t[x_t/x]$ and ℓ by $\ell[x_\ell/x]$, and add a new condition $x_\ell = x_t$ to the set of conditions C. We perform the same renaming for variables shared by ℓ and t', or by t and t', or by all the three of them. The result is that the sets of variables $\mathrm{vars}(t)$, $\mathrm{vars}(\ell)$, and $\mathrm{vars}(t')$ are pairwise disjoint, and all relations among them are part of the rule conditions. For example, the rule "$x \multimap\!\!\left[a\right]\!\!\rightarrow x$" is transformed into "$x_t \multimap\!\!\left[a\right]\!\!\rightarrow x_{t'}$ if $x_t = x_{t'}$" (which is not readable yet). The resulting rule is easily seen to induce the same half rewrites as the original one.

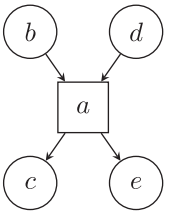
Assume, then, that $\mathrm{vars}(t)$, $\mathrm{vars}(\ell)$, and $\mathrm{vars}(t')$ are pairwise disjoint in our rule. Next, we duplicate each condition in C. We denote the result by "$t \multimap\!\!\left[\ell\right]\!\!\rightarrow t'$ if $C \wedge C'$," where C and C' are identical lists of conditions. The intuitive idea is that C' (after some additional work on it) will be only important for the second half rewrite, and C only for the first. For each variable, say x, that is in $\mathrm{vars}(C) \setminus \mathrm{vars}(t, \ell)$, we choose a fresh variable, say x', and replace C by $C[x'/x]$. In the same way, for each variable, say y, that is in $\mathrm{vars}(C') \setminus \mathrm{vars}(\ell, t')$, we choose a fresh variable, say y', and replace C' by $C'[y'/y]$. We keep the names C and C' for the results of performing all these renamings. Some conditions may still be present in its exact same form in both resulting C and C', but this is not a problem for readability.

The new rule is semantically equivalent to the original one. Each half rewrite that was possible using the original rule and a substitution θ is still possible using the new rule and the extended substitution θ', that is defined on the new variables as they were in the original ones, that is, $\theta'(x') = \theta(x)$, if x' is the new variable chosen to rename x. Also, if a half rewrite is possible according to the new rule with a substitution θ, the same half rewrite was possible with the original rule and the corresponding substitution. Our example rule has become "$x_t \multimap\!\!\left[a\right]\!\!\rightarrow x_{t'}$ if $x_t = x'_{t'} \wedge x'_t = x_{t'}$."

The resulting rule is readable. The only variables that can be in $\mathrm{vars}(C) \cap \mathrm{vars}(C')$ are the ones in $\mathrm{vars}(\ell)$. But, as noted before, θ_1 and θ_2, used for consecutive half rewrites, must coincide in $\mathrm{vars}(\ell)$. Thus, we can define the common substitution $\theta = \theta_1|_{\mathrm{vars}(C)} \cup \theta_2|_{\mathrm{vars}(C')}$. □

Another noteworthy consequence of our definition of the half-rewrite relation is that a transition term can be rewritten with a rule that is not the one that produced it. Consider an atomic egalitarian rewrite system with these two rules: $b \multimap\!\!\left[a\right]\!\!\rightarrow c$ and $d \multimap\!\!\left[a\right]\!\!\rightarrow e$, for constants a, b, c, d, and e of appropriate sorts. They specify a single transition identified by the term a. The joint semantics is:

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press



Once the transition a is executing, it forgets where it came from. Half rewrites $b \to a \to e$ must be allowed. This may be acceptable, but one should think twice before using it in the practice of system specification.

Still one more consequence worth mentioning is that the initial term that we identify as `init` can be a state term or a transition term. All this is in agreement with our egalitarian view of rewrite systems.

In a rule "$t \dashv\!\lfloor \ell \rfloor\!\to t'$ if C" such that $\mathrm{vars}(t) \cup \mathrm{vars}(t') \cup \mathrm{vars}(C) \subseteq \mathrm{vars}(\ell)$, a transition term univocally identifies one origin and one destination state terms. Its rewrite semantics is equivalent to that of the standard-setting rewrite rule "$t \dashv\!\lfloor \ell \rfloor\!\to t'$ if C" considering now ℓ as an unstructured atomic label. Transition terms, in this case, are proof terms. Thus, if we care to transform a standard rewrite system into a semantically equivalent atomic egalitarian one, all we need to do is to transform each rule label ℓ into an operator that takes as parameters all the variables in $\mathrm{vars}(t) \cup \mathrm{vars}(t') \cup \mathrm{vars}(C)$ and produces a term of sort `Trans`.

A *property* in an atomic system is any unary operator in Σ equationally defined, partially or totally, on `Stage` or some of its subsorts. Properties are used below to establish synchronization criteria for compositions (see Section 3.2.2). An important ingredient of our view of compositionality is that properties are defined only at the stages where they are meaningful, and undefined properties pose no requirements for synchronization. That is why we allow partially defined properties.

Rewriting logic is built on a base equational logic. As is usual since (Meseguer 1992), we have chosen *membership equational logic*. This is apparent in our definition of atomic rewrite system at the beginning of this section that includes membership axioms and membership conditions.

Membership equational logic allows partially defined properties (or operators in general) by declaring them *at the kind level*. All our properties have to be defined like this: $p : \texttt{Stage} \to [s]$. Sometimes, $p(t)$ has sort s, and we say it is defined at t; sometimes, $p(t)$ has just kind $[s]$ but no sort, which we interpret as it being undefined at this particular t. It is not necessary to make this explicit in the declaration of properties, however, because in a kind-completed signature each operator is automatically overloaded at the kind level, as already described.

(A different, standard way to represent partial functions is by adding a *bottom* value to its codomain. Whether to use this approach or ours (representing undefined properties by kinded but not sorted expressions) is, to some extent, a matter of personal taste. But we find our approach more appealing both for theory and for implementation, for technical reasons that need not be discussed here.)

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

3.2.2 Egalitarian rewrite systems

The class of *egalitarian rewrite systems*, EgRwSys, is built up from atomic ones by performing synchronous compositions. Formally, we define egalitarian rewrite systems and, simultaneously, their sets of properties by:

- if $\mathcal{R} \in$ atEgRwSys, then $\mathcal{R} \in$ EgRwSys, with its properties being its operators partially or totally defined on `Stage` or some subsort;
- given $\mathcal{R}_1, \mathcal{R}_2 \in$ EgRwSys, with respective sets of properties P_1 and P_2, and given a relation $Y \subseteq P_1 \times P_2$ (the synchronization criteria), the expression $\mathcal{R}_1 \|_Y \mathcal{R}_2$ denotes a new element of EgRwSys, called the *synchronous composition of* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *with respect to* Y, whose set of properties is defined to be $P_1 \uplus P_2$.

The class EgRwSys is the smallest one obtained by applying these two rules.

All properties from $\mathcal{R}_1$ and from $\mathcal{R}_2$ are automatically properties of their synchronous composition. Ultimately, all properties are defined in atomic systems and inherited by the composed systems in which they take part. When disambiguation is needed, we refer to a property by $p_{\mathcal{R}_1}$, or by $p_{\mathcal{A}}$ if $\mathcal{A}$ is the atomic system on which the property was initially defined.

Each synchronization criterion $(p_1, p_2) \in Y$ is interpreted (see below) as a required equational equivalence between the values of $p_1(g_1)$ and $p_2(g_2)$ (if defined) at the respective stages g_1 and g_2. This implies that some common equational infrastructure in the systems involved is needed. Suppose that, for $i = 1, 2$, the system $\mathcal{A}_i$ is the atomic component of $\mathcal{R}_i$ in which p_i was originally defined, and declared as $p_i : \mathtt{Stage}_{\mathcal{A}_i} \to s_i$, for $s_i \in S_{\mathcal{A}_i}$. We require and assume the existence of a membership equational theory $\mathcal{E} = (S_{\mathcal{E}}, \leq_{\mathcal{E}}, \Sigma_{\mathcal{E}}, E_{\mathcal{E}}, M_{\mathcal{E}})$, with $s_1, s_2 \in S_{\mathcal{E}}$ and $[s_1]_{\mathcal{E}} = [s_2]_{\mathcal{E}}$, which is a common subtheory of the membership equational theories of $\mathcal{A}_1$ and of $\mathcal{A}_2$. That is, $\mathcal{E}$ is component-wise included and protected in both membership equational theories of $\mathcal{A}_1$ and of $\mathcal{A}_2$. The word *protected* has here the same meaning as in Maude: the set of ground terms is preserved, that is, for each $s \in S_{\mathcal{E}}$, we have $T_{\Sigma_{\mathcal{E}},s} = T_{\Sigma_{\mathcal{A}_1},s} = T_{\Sigma_{\mathcal{A}_2},s}$. If $s = \mathrm{lub}(s_1, s_2)$ is the least common supersort of s_1 and s_2 in $\mathcal{E}$, we refer to $\mathcal{E}$ as *the common theory of* s. It is in this theory that the equality of $p_1(g_1)$ and $p_2(g_2)$ is to be checked for synchronization, namely in the form $p_1(g_1)\downarrow_{\mathcal{A}_1} =_{\mathcal{E}} p_2(g_2)\downarrow_{\mathcal{A}_2}$ (down arrows representing reduction to normal forms, as usual). This is only needed when $p_1(g_1)$ and $p_2(g_2)$ are both defined, that is, they have sorts s_1 and s_2, respectively, both in $S_{\mathcal{E}}$, so that the equational equivalence check makes sense. All this is used a few lines below.

Strictly speaking, a non-atomic rewrite system is *not* a rewrite system, in the traditional sense of inducing a rewrite relation between terms. A non-atomic rewrite system models a distributed system and, thus, its behavior can only be described based on the interaction between the behaviors of, ultimately, its atomic subcomponents. The behavior of an atomic component of an egalitarian rewrite system is restricted by the other components according to the synchronization criteria.

The definition of the set of atomic components and the total set of criteria for an egalitarian rewrite system are almost identical to the ones given for transition structures in Section 3.1.2 (Definitions 1 and 2) and need not be repeated here. Like there, the relation $\mathcal{R}_1 \equiv \mathcal{R}_2$ is defined by $\mathrm{atomic}(\mathcal{R}_1) = \mathrm{atomic}(\mathcal{R}_2)$ and $\mathrm{criteria}(\mathcal{R}_1) = \mathrm{criteria}(\mathcal{R}_2)$, and is interpreted as $\mathcal{R}_1$ and $\mathcal{R}_2$ representing the same behavior, that is, being equivalent models

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

of the same system. We allow expressions like $\|_Y{}_{i=1}^{n}\mathcal{R}_i$, or $\|_{Y,I}\mathcal{R}_i$ for some index set I, or just $\|_Y\mathcal{R}_i$ if indexes are implied, to stand for any representative of a $\equiv$-equivalence class.

3.2.3 Plain rewrite systems

What we call *plain rewrite systems* are, in essence, traditional rewrite systems whose rules have no labels. These are non-egalitarian – transitions do not play any special role. The reason we introduce these is that, as we show in the next section, egalitarian rewrite systems can be translated into plain ones, with some benefits.

In the following definition, we avoid repeating what has already been said in Section 3.2.1 when defining atomic egalitarian systems. A plain rewrite system is a tuple $(S, \leq, \Sigma, E, M, R)$, where:

- $(S, \leq, \Sigma)$ is an order-sorted signature, assumed kind-complete.
- E is a set of (possibly conditional) equations, each of the form "$t = t'$ if C," with $C = \left(\bigwedge_i t_i = t'_i\right) \wedge \left(\bigwedge_j t_j : s_j\right)$.
- M is a set of (possibly conditional) membership axioms, each of the form "$t : s$ if C," with C as above.
- R is a set of (possibly conditional) unlabeled rewrite rules. Each rewrite rule has the form "$t \to t'$ if C," where $t, t' \in T_\Sigma(X)$, and $[\mathrm{ls}(t)] = [\mathrm{ls}(t')]$, and the condition C is as above.

The class of plain rewrite systems is denoted by RwSys.

The mandatory absence of labels in rules is the main difference between standard rewrite systems and our plain ones. It has the effect of making transitions identifiable just by their origin and destination states, thus merging all edges between two given states into one.

The (one-step) rewrite relation $u \to_{\mathcal{R}} u'$ in a plain rewrite system is as in the standard case, that is, $u \to_{\mathcal{R}} u'$ holds for terms $u, u' \in T_\Sigma$ iff there are a rule "$t \to t'$ if C" in R, a term $w \in T_\Sigma$, a position p in w, and a substitution θ such that $\mathcal{R} \models C\theta$, $u =_E w = w[t\theta]_p$, and $u' =_E w[t'\theta]_p$.

We assume that the distinguished sort of the terms we are interested in rewriting is called `State`. The initial state is assumed to be called `init`. A *property*, in this context, is any operator in Σ partially or totally defined on `State` or some subsort of it. Properties are used to establish synchronization criteria for compositions.

The operation of synchronous composition for plain rewrite systems is only defined for operands that satisfy certain requirements. Given two plain rewrite systems, $\mathcal{R}_1$ and $\mathcal{R}_2$, with respective sets of properties P_1 and P_2, their synchronous composition with respect to the synchronization criteria $Y \subseteq P_1 \times P_2$ is only defined if (i) $\mathcal{R}_1$ and $\mathcal{R}_2$ are topmost and (ii) the properties used in Y are totally defined, each in its respective system. We justify these points.

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. As detailed below, the synchronous composition uses the components' rules to produce composed ones. If a system includes the rule $a \to a'$ and another system includes $b \to b'$, then their composition includes the rule $\langle a, b\rangle \to \langle a', b'\rangle$ (with some conditions), to be used on

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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 the systems to be topmost, that is, that rewrites are possible only at the top of state terms.

Syntactic conditions equivalent to topmostness are given, for example, in Meseguer and Thati (2007). They are similar to the ones we required for atomic egalitarian rewrite systems at the beginning of Section 3.2.1. That same 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. As explained in Section 2.8, instead of completing rules, we prefer to decompose systems until each atomic system is topmost.

The reason to require that all properties used in Y be totally defined is the following. Properties undefined on a particular state pose no conditions for synchronization. If, for example, the synchronization criteria is $Y = \{(p_1, p_2)\}$, and $p_1(q_1)$ is undefined at a given $q_1 \in \mathtt{State}_{\mathcal{R}_1}$, then q_1 is compatible with any $q_2 \in \mathtt{State}_{\mathcal{R}_2}$, and $\langle q_1, q_2\rangle$ has sort $\mathtt{State}_{\mathcal{R}_1 \|_Y \mathcal{R}_2}$. We run into problems when we need to express this as a valid membership condition. Undefinedness of a term, in membership equational logic, is represented by terms having kind but not sort. While positive sort or kind assertions like $q : [\mathtt{State}]$ are valid, negative assertions like $\neg(q : \mathtt{State})$ are not valid as conditions in membership equational logic (because every membership axiom is positive, only positive conclusions can be reached).

There are some ways to overcome or mitigate this limitation. The language Maude includes built-in, extra-logical features to check whether *or not* a term has a sort, and that solves the problem in practice. Without leaving the strictness of rewriting logic, it is not uncommon that the cases where a property must be left undefined are identifiable and expressible as valid conditions. This opens some possibilities. One of them is to create for each state q for which the property $p : \mathtt{State} \to [s]$ is undefined as many states as values are in the universe of the sort s. For example, if s is $\mathtt{Bool}$, we would replace q by $q_{\mathtt{true}}$ and $q_{\mathtt{false}}$. We have to duplicate (or, in general, replicate) all rules and equations needed so that each of the new states has the same behavior as the old one had. Then, we define $p(q_b) = b$, making p total. States at the other system can always synchronize with the appropriate q_b. This procedure can result in very complicated specifications if applied literally for each partially defined property. In our experience, undefinedness is not often needed, so we hope this is not such a serious constraint.

We define the synchronous composition of two topmost plain rewrite systems, $\mathcal{R}_1$ and $\mathcal{R}_2$, with respect to a synchronization criteria Y that uses only totally defined properties. It is denoted by $\mathcal{R}_1 \|_Y \mathcal{R}_2$ and is defined to be a new plain rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R)$ that happens to be topmost as well. The set of rules R is described below. The rest of the elements of $\mathcal{R}$ are defined as the almost disjoint (see below) union of the respective components of $\mathcal{R}_1$ and $\mathcal{R}_2$ (that is, $S = S_{\mathcal{R}_1} \cup S_{\mathcal{R}_2}$, and so on), except for the following:

- There is in S a new sort $\mathtt{State}$ $(= \mathtt{State}_{\mathcal{R}})$ and a constructor $\langle _, _\rangle : \mathtt{State}_{\mathcal{R}_1} \times \mathtt{State}_{\mathcal{R}_2} \to [\mathtt{State}]$.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

- There is a new constant `init` of sort `State` and an equation $\texttt{init} = \langle \texttt{init}_{\mathcal{R}_1}, \texttt{init}_{\mathcal{R}_2} \rangle$.
- For each $(p_1, p_2) \in Y$, with $p_i : \texttt{State}_{\mathcal{R}_i} \to [s_i]$, we assume, as we did in Section 3.2.2, and for the same reasons, the existence of a common equational theory of $\mathrm{lub}(s_1, s_2)$, subtheory of both $\mathcal{R}_1$ and $\mathcal{R}_2$. (This is where the unions are not disjoint.)
- We add the following membership axiom, formalizing compatibility of states:

$$\langle q_1, q_2 \rangle : \texttt{State} \text{ if } q_1 : \texttt{State}_{\mathcal{R}_1} \wedge q_2 : \texttt{State}_{\mathcal{R}_2} \wedge \bigwedge_{(p_1,p_2)\in Y} p_1(q_1) = p_2(q_2).$$

- For each property $p : \texttt{State}_{\mathcal{R}_i} \to [s_i]$ defined in $\mathcal{R}_i$, for $i = 1$ or 2, there is in Σ a declaration $p : \texttt{State} \to [s_i]$ and in E an equation *exporting* the property: "$p(\langle q_1, q_2 \rangle) = p(q_i)$ if $\langle q_1, q_2 \rangle : \texttt{State}$."

The set of rules R is built in the following way. For each pair of rules "$q_1 \to q_1'$ if C_1" from R_1 and "$q_2 \to q_2'$ if C_2" from R_2, there are three rules in R, namely:

- $\langle q_1, q_2 \rangle \to \langle q_1', q_2' \rangle$ if $C_1 \wedge C_2 \wedge \langle q_1, q_2 \rangle : \texttt{State} \wedge \langle q_1', q_2' \rangle : \texttt{State}$,
- $\langle q_1, q_2 \rangle \to \langle q_1', q_2 \rangle$ if $C_1 \wedge \langle q_1, q_2 \rangle : \texttt{State} \wedge \langle q_1', q_2 \rangle : \texttt{State}$,
- $\langle q_1, q_2 \rangle \to \langle q_1, q_2' \rangle$ if $C_2 \wedge \langle q_1, q_2 \rangle : \texttt{State} \wedge \langle q_1, q_2' \rangle : \texttt{State}$.

We remind the reader that we assume each system to be its own namespace (with the exception of the common equational subtheories mentioned in Section 3.2.2). It is in this sense that we speak about (almost) disjoint unions in previous lines. In particular, when we join together equations coming from $\mathcal{R}_1$ and from $\mathcal{R}_2$, their effects do not overlap.

As in previous sections, some different ways to compose plain rewrite systems are intuitively seen to be equivalent. We denote by $\Sigma|_{\texttt{State}}$ the subset of the operators in Σ that take a single argument of sort `State`, and by $T_{\Sigma/E,\texttt{State}}$, as usual, the set of terms of sort `State` modulo equations.

Definition 8
Given two plain rewrite systems $\mathcal{R}_i \in \mathsf{RwSys}$, for $i = 1, 2$, we say that they are *equivalent*, and denote it by $\mathcal{R}_1 \equiv \mathcal{R}_2$, if there exist two bijections, $f : T_{\Sigma_1,\texttt{State}} \to T_{\Sigma_2,\texttt{State}}$ and $F : \Sigma_1|_{\texttt{State}} \to \Sigma_2|_{\texttt{State}}$, such that, for all $q_1, q_1' \in T_{\Sigma_1,\texttt{State}}$ and all $p_1 \in \Sigma_i|_{\texttt{State}}$:

- the rewrite relation is preserved: $q_1 \to_{\mathcal{R}_1} q_1' \Leftrightarrow f(q_1) \to_{\mathcal{R}_2} f(q_1')$;
- result sorts for properties are the same: if p_1 is declared in $\mathcal{R}_1$ as $p_1 : \texttt{State} \to s$, for a sort $s \in S_1$, then also $s \in S_2$ and $F(p_1)$ is declared in $\mathcal{R}_2$ as $F(p_1) : \texttt{State} \to s$, and $T_{\Sigma_1/E_1,s} = T_{\Sigma_2/E_2,s}$;
- values of properties are preserved: $p_1(q_1) = F(p_1)(f(q_1))$;
- initial states are preserved: $f(\texttt{init}) = \texttt{init}$.

This definition closely resembles Definition 4. As in previous sections, we allow expressions like $\|_Y^n{}_{i=1}\mathcal{R}_i$, or $\|_{Y,I}\mathcal{R}_i$ for some index set I, or just $\|_Y\mathcal{R}_i$ if indexes are implied. Also, we denote by $\langle q_1, \ldots, q_n \rangle$ a typical element of $\|_Y\mathcal{R}_i$.

Although the operator "$\|$" for RwSys reflects the one for EgRwSys, they are of a different nature: $\mathcal{R}_1\|_Y\mathcal{R}_2$ is an irreducible expression in EgRwSys, but can be reduced in RwSys to a single rewrite system as already described.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

In the next section, we show how plain rewrite systems are useful as a more standard equivalent to egalitarian ones. Plain systems also have the advantage of being always single systems, even when they are the result of a composition. If one does not care about egalitarianism, plain rewrite systems can be taken as a basis for discussing non-egalitarian compositional specification.

3.2.4 The split

The *split* is a translation of rewrite systems: split : $\mathsf{EgRwSys} \to \mathsf{RwSys}$. Translating an atomic system into a plain one is straightforward: given $\mathcal{R} = (S, \leq, \Sigma, E, M, R) \in \mathsf{atEgRwSys}$, its split is $\text{split}(\mathcal{R}) = (S', \leq, \Sigma, E, M, R') \in \mathsf{RwSys}$, where S' is the result of renaming in S the sort `State` to `State'`, and `Stage` to `State` (with the only aim of getting the top sort still being called `State`), and R' is the result of splitting each rule "$t \dashv\!\lfloor \ell \rfloor\!\to t'$ if C" in R to produce the two rules "$t \to \ell$ if C" and "$\ell \to t'$ if C" in R'. (This rule splitting is, by the way, the reason for choosing the name *split* for all similar operations in this paper.)

For a non-atomic system $\mathcal{R}_1 \|_Y \mathcal{R}_2$, its *split* is recursively defined by $\text{split}(\mathcal{R}_1 \|_Y \mathcal{R}_2) = \text{split}(\mathcal{R}_1) \|_Y \text{split}(\mathcal{R}_2)$. With this definition, it can be proved inductively (similar to Section 3.1.4) that the sets of properties for $\mathcal{R}$ and for $\text{split}(\mathcal{R})$ are the same for any $\mathcal{R} \in \mathsf{EgRwSys}$ and, thus, that the same relation Y can be used in both sides of the definition. But the composition of plain systems in the right-hand side of this equation has to satisfy the two provisos: that the properties in Y are totally defined and that the operands are topmost. Topmostness, in particular, deserves a few lines. We have not defined when an atomic egalitarian rewrite system is topmost, but the definition for plain systems can be readily adapted: when rewrites are only possible for complete state terms and only through complete transition terms. With this definition, it is easily seen that $\mathcal{R} \in \mathsf{atEgRwSys}$ is topmost iff $\text{split}(\mathcal{R}) \in \mathsf{RwSys}$ is. Also easy is that the composition of topmost plain rewrite systems is topmost. Thus, repeated application of "$\|$" is allowed, and we are entitled to write $\text{split}(\|_{Y,I} \mathcal{R}_i) = \|_{Y,I} \text{split}(\mathcal{R}_i)$.

This definition of the split operation is theoretically sound, but can produce a large number of rules in the resulting system, so it is not suited for an eventual implementation. This is illustrated with the example at the end of Section 4.

As we observed in Section 3.1.4, the states and rewrites in $\text{split}(\mathcal{R})$ can be taken as global states and global rewrites for the egalitarian system $\mathcal{R}$.

Proposition 4
For $\mathcal{R}_1, \mathcal{R}_2 \in \mathsf{EgRwSys}$, we have $\mathcal{R}_1 \equiv \mathcal{R}_2 \Rightarrow \text{split}(\mathcal{R}_1) \equiv \text{split}(\mathcal{R}_2)$.

Proof
Given that $\mathcal{R}_1 \equiv \mathcal{R}_2$ implies $\text{atomic}(\mathcal{R}_1) = \text{atomic}(\mathcal{R}_2)$, either $\mathcal{R}_1$ and $\mathcal{R}_2$ are both atomic or none is. In the first case, indeed, $\mathcal{R}_1 = \mathcal{R}_2$, and the result is immediate. In the second case, the result follows easily from the definitions. For example, the terms of sort `State` in $\text{split}(\mathcal{R}_1)$ and those in $\text{split}(\mathcal{R}_2)$ are in bijection because both are built by applying tuple operators to the same set of `State` sorts from atomic components in different ways, and restricted by the same membership axioms, stemming from $\text{criteria}(\mathcal{R}_1) = \text{criteria}(\mathcal{R}_2)$. □

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

3.3 Semantics

Definition 9
Given an atomic egalitarian rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R) \in \mathsf{atEgRwSys}$, we define its associated atomic egalitarian transition structure $\mathrm{sem}(\mathcal{R}) = (Q, T, \rightarrow, P, g^0) \in \mathsf{atEgTrStr}$ by:

- Q is the set of ground terms of sort `State` in $\mathcal{R}$ modulo E, that is, $T_{\Sigma/E,\mathtt{State}}$;
- T is the set of ground terms of sort `Trans` in $\mathcal{R}$ modulo E, that is, $T_{\Sigma/E,\mathtt{Trans}}$;
- $\rightarrow$ is the half-rewrite relation $\rightarrow_{\mathcal{R}}^{eg}$ induced by R according to Definition 6;
- P is $\Sigma|_{\mathtt{State}}$ (the set of operators in Σ whose domain is `Stage` or one of its subsorts);
- $g^0 = [\mathtt{init}]_E$.

Definition 10
Given a non-atomic egalitarian rewrite system $\mathcal{R}_1 \|_Y \mathcal{R}_2 \in \mathsf{EgRwSys}$, we define its associated non-atomic egalitarian transition structure by $\mathrm{sem}(\mathcal{R}_1 \|_Y \mathcal{R}_2) = \mathrm{sem}(\mathcal{R}_1) \|_Y \mathrm{sem}(\mathcal{R}_2) \in \mathsf{EgTrStr}$.

Note that the same Y can be used on both sides.

Definition 11
Given a plain rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, M, R) \in \mathsf{RwSys}$, we define its associated plain transition structure $\mathrm{sem}(\mathcal{R}) = (Q, \rightarrow, P, g^0) \in \mathsf{TrStr}$ by:

- Q is the set of ground terms of sort `State` in $\mathcal{R}$ modulo E, that is, $T_{\Sigma/E,\mathtt{State}}$;
- $\rightarrow$ is the rewrite relation $\rightarrow_{\mathcal{R}}$ induced by R according to Section 3.2.3;
- P is $\Sigma|_{\mathtt{State}}$;
- $g^0 = [\mathtt{init}]_E$.

Proposition 5
For plain rewrite systems $\mathcal{R}_1, \mathcal{R}_2 \in \mathsf{RwSys}$, each of them topmost with respective sets of totally defined properties P_1 and P_2, and for synchronization criteria $Y \subseteq P_1 \times P_2$, we have $\mathrm{sem}(\mathcal{R}_1 \|_Y \mathcal{R}_2) \equiv \mathrm{sem}(\mathcal{R}_1) \|_Y \mathrm{sem}(\mathcal{R}_2)$ (for the equivalence relation on TrStr from Definition 4).

Proof
Easy application of the definitions. For example, the set of states for $\mathrm{sem}(\mathcal{R}_1) \|_Y \mathrm{sem}(\mathcal{R}_2)$ is the product $T_{\Sigma_1/E_1,\mathtt{State}} \times T_{\Sigma_2/E_2,\mathtt{State}}$ (that is, pairs of terms of sort `State`, one from each system) restricted by the conditions from Y. Meanwhile, the set of states for $\mathrm{sem}(\mathcal{R}_1 \|_Y \mathcal{R}_2)$ is $T_{\Sigma/E,\mathtt{State}}$, where `State` is defined by the constructor $\langle _, _ \rangle : \mathtt{State}_{\mathcal{R}_1} \times \mathtt{State}_{\mathcal{R}_2} \rightarrow [\mathtt{State}]$, and subject to the membership axioms drawn from Y. These sets can be easily put in bijection as required by Definition 4. □

As a way to validate our definitions of semantics, we state the following results.

Proposition 6
For $\mathcal{R}_1, \mathcal{R}_2 \in \mathsf{EgRwSys}$, we have $\mathcal{R}_1 \equiv \mathcal{R}_2 \Rightarrow \mathrm{sem}(\mathcal{R}_1) \equiv \mathrm{sem}(\mathcal{R}_2)$.

Proof
Trivial for both the atomic case and the composed, inductive case. □

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Proposition 7
For $\mathcal{R}_1, \mathcal{R}_2 \in \mathsf{RwSys}$, we have $\mathcal{R}_1 \equiv \mathcal{R}_2 \Rightarrow \mathrm{sem}(\mathcal{R}_1) \equiv \mathrm{sem}(\mathcal{R}_2)$.

Proof
Easy thanks to the very similar-looking definitions of the two equivalence relations. □

Theorem 1
For every egalitarian rewrite system $\mathcal{R} \in \mathsf{EgRwSys}$ with totally defined properties and all whose components are topmost we have $\mathrm{sem}(\mathrm{split}(\mathcal{R})) \equiv \mathrm{split}(\mathrm{sem}(\mathcal{R}))$ (for the equivalence relation on TrStr from Definition 4).

This states the commutativity of one of the faces of the polyhedron in the introduction to Section 3. Indeed, this was the only face whose commutativity remained to be addressed.

Proof
We use structural induction on the shape of $\mathcal{R}$. The base case is the atomic one. Let $\mathcal{R} = (S, \leq, \Sigma, E, M, R) \in \mathsf{atEgRwSys}$. From the definitions, it is not difficult to see that both $\mathrm{split}(\mathrm{sem}(\mathcal{R}))$ and $\mathrm{sem}(\mathrm{split}(\mathcal{R}))$ result in the plain transition structure

$$(T_{\Sigma/E,\mathtt{Stage}}, \rightarrow^{\mathrm{eg}}_{\mathcal{R}}, \Sigma|_{\mathtt{Stage}}, [\mathtt{init}]_E).$$

For the composed case, $\mathcal{R} = \mathcal{R}_1 \|_Y \mathcal{R}_2$:

$$\begin{aligned}
& \mathrm{sem}(\mathrm{split}(\mathcal{R}_1 \|_Y \mathcal{R}_2)) \\
= \quad & \text{(definition in Section 3.2.4, split for } \mathsf{EgRwSys}) \\
& \mathrm{sem}(\mathrm{split}(\mathcal{R}_1) \|_Y \mathrm{split}(\mathcal{R}_2)) \\
\equiv \quad & \text{(Proposition 5)} \\
& \mathrm{sem}(\mathrm{split}(\mathcal{R}_1)) \|_Y \mathrm{sem}(\mathrm{split}(\mathcal{R}_2)) \\
= \quad & \text{(induction hypothesis)} \\
& \mathrm{split}(\mathrm{sem}(\mathcal{R}_1)) \|_Y \mathrm{split}(\mathrm{sem}(\mathcal{R}_2)) \\
= \quad & \text{(Definition 5, split for } \mathsf{EgTrStr}) \\
& \mathrm{split}(\mathrm{sem}(\mathcal{R}_1) \|_Y \mathrm{sem}(\mathcal{R}_2)) \\
= \quad & \text{(Definition 10, sem for } \mathsf{EgRwSys}) \\
& \mathrm{split}(\mathrm{sem}(\mathcal{R}_1 \|_Y \mathcal{R}_2)).
\end{aligned}$$

□

3.4 The power of simple synchronization

Our mechanism for handling properties and synchronization is quite simple: synchronization is given by mere equality of properties, which are inherited untouched from a component to its composition. In contrast, other formalisms (some of which are mentioned in Section 5) have complex channels for communication. And, in object-oriented technologies, for example, the interface for a container object may be quite different from the interfaces of the objects it contains or uses. Our concept of connector is that of a simple wire. A complex *connector*, with some logic in it, has to be modeled as a new, intermediate component.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

We expect that this choice for utmost simple interaction will make the compositional analysis of systems easier. The implicit danger is that such simplicity restricts the variety of systems that can be modeled. We show in this section two examples of how our simplicity is not a restriction.

Complex properties. In this example, there are two components, $\mathcal{S}_1$ and $\mathcal{S}_2$, each providing a natural number by means of a property, n_1 and n_2, respectively. A different system $\mathcal{S}'$ has a property n', and we want something to the effect of $(\mathcal{S}_1 \|_\emptyset \mathcal{S}_2) \|_{\{(n_1+n_2,n')\}} \mathcal{S}'$. In words: at each moment, only stages g_1, g_2, and g' satisfying $n_1(g_1) + n_2(g_2) = n'(g')$ can be visited simultaneously.

This can be achieved by adding a third component able to perform addition. This component $\mathcal{S}_+$ has to present three properties: m_1, m_2, and s, and has to guarantee that $s(g) = m_1(g) + m_2(g)$ at each stage g. For example, the system $\mathcal{S}_+$ can be implemented with pairs of numbers as states and a single rule "$(p, q) \; -\!\!\left[\ell\right]\!\!\rightarrow \; (p', q')$." The properties are defined as $m_1((p, q)) = p$, $m_2((p, q)) = q$, and $s((p, q)) = p + q$. The system S_+ has no restrictions about which state to visit next, but appropriate synchronization makes the state of S_+ reflect the current values of n_1 and n_2 from the components. The composition we were looking for is $((\mathcal{S}_1 \|_\emptyset \mathcal{S}_2) \|_{\{(m_1,n_1),(m_2,n_2)\}} \mathcal{S}_+) \|_{\{(s,n')\}} \mathcal{S}'$. Or, if we prefer, $\|_{\{(m_1,n_1),(m_2,n_2),(s,n')\}} \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_+, \mathcal{S}'\}$.

Synchronizing on relations other than equality. Suppose that we need to synchronize two systems not just by equality of the values of some properties, but on some other relation between them. For example, both properties are numbers and the relation is $<$. Or a property is a number, the other a set, and the relation is $\in$. This can be obtained by introducing an intermediate component.

More formally, suppose we have a system $\mathcal{S}_1$ that defines a property p_1 with codomain C_1, and another $\mathcal{S}_2$ that defines p_2 with codomain C_2, and we need to synchronize so that the values of the properties are related by a given relation $R \subseteq C_1 \times C_2$. That is, something to the effect of $\mathcal{S}_1 \|_{R(p_1,p_2)} \mathcal{S}_2$. The intermediate component $\mathcal{S}_R$ has a single rule "$(p, q) \; -\!\!\left[\ell\right]\!\!\rightarrow \; (p', q')$ if $R(p', q')$," and properties $\pi_1((p, q)) = p$ and $\pi_2((p, q)) = q$. The composed system is $\|_{\{(p_1,\pi_1),(p_2,\pi_2)\}} \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_R\}$.

3.5 Executability

Executability is one big benefit of rewriting logic as a system specification formalism. A specification that satisfies some requirements (see below) can be executed and model checked. This is in addition to being formal and, thus, amenable to formal analysis. If our proposal for synchronous composition is to provide some value, executability of composed systems is essential.

In general, executability of each atomic component in isolation may be neither necessary nor sufficient to ensure executability of the composed system: not necessary, because a specification that is not executable by itself may become so when restricted or enriched by interaction with the appropriate environment, and not sufficient, because also synchronization criteria have to be evaluated at runtime.

The rewriting relation between terms (i.e., whether a term t can be rewritten to t' in a rewrite system) is undecidable for arbitrary sets of equations and rules. In the setting of

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

standard rewriting logic, in Meseguer (2008), 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 or associativity of certain operators) is decidable;
- the set of equations is ground Church–Rosser and ground terminating (modulo the axioms A);
- rules are ground coherent with respect to the equations (modulo the axioms A).

These conditions are easy to meet. Usually, the rewrite systems a sensible programmer would code are computable.

We can adapt these conditions to atomic egalitarian rewrite systems. Only the third one deals with rules and, thus, only it needs to be adapted. In the standard setting, coherence means that if a rewrite is possible from a state term t to another t', then from any term in the equational class of t a rewrite is possible to a term in the equational class of t'. This allows an execution engine to work by, first, reducing the current state term to its normal form with respect to the equations and, then, rewriting from the normal form; see (Clavel *et al*. 2007, Section 6.3) for complete explanations. In our setting, transition terms can be reduced by equations, in the same way as state terms can. Thus, we need coherence in the two phases:

- if a transition with term ℓ can be fired from a state with term t, then from any state term in the equational class of t a transition can be fired whose term is in the equational class of ℓ;
- if a transition with term ℓ can reach a state with term t', then from any transition term in the equational class of ℓ a state can be reached whose term is in the equational class of t'.

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

For a non-atomic egalitarian rewrite system, we also need that the synchronization criteria be effectively checkable. But these are equalities in what we have called in Section 3.2.2 the common equational theory for the sort of the values involved in the synchronization. As this common theory is included in each atomic component that needs it, its decidability is already implied by the conditions above. Thus, we can call a non-atomic rewrite system *computable* if all its atomic components are. With this definition, a system is computable iff its split is.

That a system is computable in the sense above is a basic requirement toward executability. The practical implementation of an execution engine or a model checker is to be expected to need additional requirements. In the case of Maude's standard engine and model checker, there are a series of conditions to ensure that any variable in the right-hand side or the conditions of a rule can be instantiated at runtime (maybe in a non-deterministic way). These are called *admissibility conditions* in Clavel *et al*. (2007).

More recently, narrowing-based procedures have been developed for symbolic execution and model checking of Maude specifications; see Meseguer and Thati (2007) for details.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

These procedures do not require the same admissibility conditions, but have their own limitations. For example, they cannot handle conditional rules.

In Section 3.4, we proposed using the rule "$(p, q) \; -[\ell]\rightarrow \; (p', q')$ if $R(p', q')$." This involves fresh, unbound variables that Maude's execution engine is not able to handle (i.e., even when split into two standard rules). And it is conditional, so the existing narrowing engine cannot be used either. We discuss some possible ways out in the rest of this section.

The language Maude allows *matching conditions* in equations and in rules. They take the form `if pattern := term`. Semantically, this is equivalent to the equational condition `if pattern = term`, but it allows Maude's engine to instantiate variables appearing in the `pattern` by matching against the reduced `term`; see Clavel *et al.* (2007) for precise descriptions of what a pattern is and related explanations. In particular, matching conditions can be used to make Maude's engine choose a value non-deterministically from a set:

```
op FiniteSetOfAdmissibleValues : -> Set{Value} .
eq FiniteSetOfAdmissibleValues = ...
vars V V' : Value .
var Rest : Set{Value} .
crl V => V' if V' Rest := FiniteSetOfAdmissibleValues .
```

Assuming the empty-syntax set constructor used in `V' Rest` is commutative and associative, the value of `V'` is chosen from `FiniteSetOfAdmissibleValues` non-deterministically.

A model of a memory cell, for one example, must be ready to store any value *within a range*, corresponding to the computer's word size. It is to be expected that models of real-world systems are often restricted in a similar way to a finite set of different states. Incorporating this knowledge into the model using matching conditions may turn the system into an executable one.

This idea is not always helpful, however. The system $\mathcal{S}_+$ from Section 3.4, for example, must be able to provide a *sum property* for whatever values it receives, and it must be usable in any environment. So, an *a priori* finite range should not be used. But systems like this are not meant to be executed in isolation. In the composition $\|_{\ldots}\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_+, \mathcal{S}'\}$, if we know that the values coming from $\mathcal{S}_1$ and $\mathcal{S}_2$ are within some finite range, that will give us a handle to the executability of the whole, composed system. Syntactically, it seems that a good way to represent this in code is extending the syntax for synchronization criteria to allow *matching criteria*. Instead of writing $\|_{\{(m_1,n_1),(m_2,n_2),(s,n')\}}\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_+, \mathcal{S}'\}$, as we did above, we would write something like $\|_{\{m_1:=n_1,\, m_2:=n_2,\, n':=s\}}\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_+, \mathcal{S}'\}$. We can even expect this criteria to be translated into matching conditions by the split operation. All this, however, concerns the implementation that is not available as yet.

In addition to making the composed system executable, this idea of injecting knowledge from one component into a neighboring one is also related to the assume-guarantee technique for compositional verification (also mentioned in Section 6.3). That is, the assumption that *input* values satisfy some restrictions, either in the form of temporal formulas or otherwise, can make a system amenable to both execution and verification.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

4 A simple but complete example

We show the specification of an example system, small but completely developed, with the aim of showing the way to think and use our tools for specification. We use the language Maude extended with some syntactic constructs for synchronous composition that we intend to implement in the near future.

A few declarations of sorts and operators are used often. So as to avoid repeating them, we assume they are included in a common module that is implicitly and silently imported whenever needed.

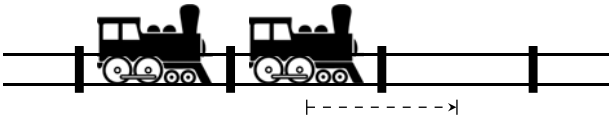
```
--- In the common module
   sorts State Trans Stage .
   subsorts State Trans < Stage .
   op init : -> Stage .
```

Some changes are convenient in the implementation with respect to the theoretical description. Properties will not be functions in our Maude extension. Property sorts are created with this parameterized module:

```
fmod PPTY{X :: TRIV} is
   sort Ppty{X} .
   op _@_ : Ppty{X} Stage -> [X$Elt] .
endfm
```

In this module, `X` is the name of the formal parameter, and `TRIV` is the name of a *trivial* standard theory in Maude that just requires the existence of a sort `Elt` in the actual parameter. If we need, for example, to declare a Boolean-valued property `B`, we import `PPTY{Bool}` and declare `op B : -> Ppty{Bool}`. The instantiated module `PPTY{Bool}` makes available the sort `Ppty{Bool}` and also the infix evaluation operator `_@_ : Ppty{Bool} Stage -> [Bool] .` Thus, to get the value of property `B` at the stage `G`, we write the expression `B @ G` (instead of `B(G)`), which has kind `[Bool]`, in this case.

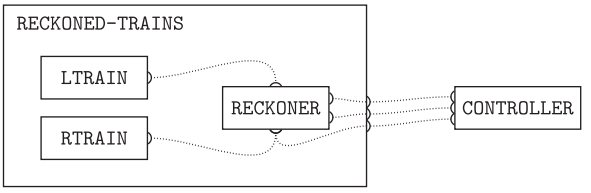
This example is inspired in one from De Nicola *et al.* (1993). Two trains move independently on the same linear railway divided into track sections. They both move rightwards, one track section at a time. The train on the left is not allowed to reach the one on the right, so that, when the trains are in consecutive track sections, the right one has to move.



The two trains are modeled 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 behavior 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*.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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



Each system is represented as a box. Rounded connectors on the edges represent the properties each system provides. Dotted-line wires represent synchronization. The reckoner, for example, needs to synchronize with the trains to know when each is moving. The controller sees the system `RECKONED-TRAINS` as a black box that provides three properties to synchronize on. Inside the system `RECKONED-TRAINS`, we see that its three properties are originally defined as properties of the reckoner. This is another difference between the implementation and the theoretical description: although the internals of the `RECKONED-TRAINS` module are not hidden, we prefer to pretend they are, and the `CONTROLLER` behaves as if they were.

We proceed in a top-down fashion to translate the diagram into Maude. This is the top-level system:

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

The keyword `pr` is short for *protecting*, a way to import modules in Maude. Synchronization 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 names of the properties are prefixed with the names of their modules, so as to avoid ambiguities. The six properties used in the synchronization criteria (three from each system) are defined below.

This is the three-way composition that produces the system `RECKONED-TRAINS`, again assuming some properties to be defined below:

```
--- In module RECKONED-TRAINS
   pr LTRAIN || RTRAIN || RECKONER
   sync on LTRAIN.isMoving = RECKONER.isLMoving
        /\ RTRAIN.isMoving = RECKONER.isRMoving .
```

This has to be complemented by the declaration and definition of the properties that have been used in the criteria for `CONTROLLED-TRAINS`:

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

```
--- Also in module RECKONED-TRAINS
   pr PPTY{Bool} .
   ops areConsec isSomeMoving isRMoving : -> Ppty{Bool} .
   var G : Stage .
   eq areConsec @ G = areConsec @ RECKONER(G) .
   eq isSomeMoving @ G = isSomeMoving @ RECKONER(G) .
   eq isRMoving @ G = isRMoving @ RECKONER(G) .
```

These properties are defined for `RECKONED-TRAINS` based on the properties of the `RECKONER` component, following the design in the figure above. Instead of using the properties automatically inherited from the components, for clarity, we prefer to define explicitly new properties for the composed system, even if they are called the same. The design of our extension for the language Maude includes the existence of projection operators for the stage term with the same name as the components. Thus, `areConsec @ RECKONER(G)` is the value of the property `areConsec` at the `RECKONER` part of the stage `G`. (We have argued before that the concept of global state is better avoided for distributed systems, so we would better interpret the use of the variable `G` as convenient syntax.)

We have arrived at the level of the atomic components. The systems that model the two trains are identical, with a single state called `stopped`, a single transition called `moving`, and a single property called `isMoving`, used to *inform* the reckoner. The egalitarian rule is written with the transition term in the middle.

```
--- In modules LTRAIN and RTRAIN
   op stopped : -> State .
   op moving : -> Trans .
   rl stopped =[ moving ]=> stopped .

   pr PPTY{Bool} .
   op isMoving : -> Ppty{Bool} .
   eq isMoving @ moving = true .
   eq isMoving @ stopped = false .
```

The property is defined as true at transitions and false at states. This may 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:

```
--- In module RECKONER
   subsort Int < State .
   ops lmoving|_ rmoving|_ 2moving|_ : Int -> Trans .
   var D : Int .
   rl D =[ lmoving | D ]=> D - 1 .
   rl D =[ rmoving | D ]=> D + 1 .
   rl D =[ 2moving | D ]=> D .
```

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

The declaration `subsort Int < State` is a way of saying that an integer is a state term by itself. Transitions with the first rule allow the distance to become zero, and even negative. This is just a reckoner, so it has nothing to say about it. The controller has to take care.

This is one instance in which the transitions have to *remember* some value from the state, `D` in this case; otherwise, each `D` could be interpreted as a different variable (see Section 3.2.1). This is sometimes counterintuitive. It is probably more intuitive to view `D` as the *context* in which the movement takes place, instead of being a parameter of the movement. We have chosen a syntax for transition terms that enforces this view: `lmoving | D`.

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

```
--- Also in module RECKONER
   pr PPTY{Bool} .
   ops areConsec isSomeMoving isLMoving isRMoving : -> Ppty{Bool} .
   var T : Trans .
   eq areConsec @ 1 = true .
   eq areConsec @ D = false [otherwise] .
   eq isSomeMoving @ D = false .
   eq isSomeMoving @ T = true .
   eq isLMoving @ (lmoving | D) = true .
   eq isLMoving @ (rmoving | D) = false .
   eq isLMoving @ (2moving | D) = true .
   eq isLMoving @ D = false .
   eq isRMoving @ (lmoving | D) = false .
   eq isRMoving @ (rmoving | D) = true .
   eq isRMoving @ (2moving | D) = true .
   eq isRMoving @ D = false .
```

We do not define the value of `areConsec` while the trains are moving. We assume that the controller (the ultimate user of this property) only needs to know its value when the trains are stopped. However, this is part of the interface (of `RECKONER` and, then, of `RECKONED-TRAINS`) and we must be careful and consistent with our assumptions throughout the specification.

Only the controller is left. 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. As shown above, we synchronize on three criteria: the first for the controller to be aware of consecutive-train 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.

The word *command* must be correctly understood. Synchronization works in a symmetrical way: 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. In an

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

abstract way, the controller's task is to detect when the base system is in a particular situation and command that only a particular action be allowed: 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.

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

```
--- In module CONTROLLER
   ops consec nonConsec : -> State .
   ops fromConsec fromNonConsec : -> Trans .
   rl consec =[ fromConsec ]=> consec .
   rl consec =[ fromConsec ]=> nonConsec .
   rl nonConsec =[ fromNonConsec ]=> consec .
   rl nonConsec =[ fromNonConsec ]=> nonConsec .
```

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 synchronize with `RECKONED-TRAINS`:

```
--- Also in module CONTROLLER
   pr PPTY{Bool} .
   ops areConsec doMove doMoveR : -> Ppty{Bool} .
   var S : State .
   var T : Trans .
   eq areConsec @ consec = true .
   eq areConsec @ nonConsec = false .
   eq doMove @ S = false .
   eq doMove @ T = true .
   eq doMoveR @ fromConsec = true .
```

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

This finishes the specification of the whole system. Let us describe informally how it works at a low level, that is, looking at the internals of the specifications. Requiring that `doMove` equals `isSomeMoving` (as we do in the synchronization criteria for `CONTROLLED-TRAINS`) has the effect that states in `RECKONER` can only be simultaneous with states in `CONTROLLER`, and the same for transitions. The equality of the two properties called `areConsec` ensures, then, that the state `consec` in `CONTROLLER` is always visited simultaneously with the state `1` in `RECKONER`. The only possible transition from `consec` is called `fromConsec` and sets the property `doMoveR` to true. The synchronization criterion `doMoveR = isRMoving`, then, mandates that a transition at which `isRMoving` is true be

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

executed. There are two such transitions in RECKONER: `lmoving` and `2moving`. Each of them makes RTRAIN execute its `moving` transition, as can be inferred by continuing this chain of implications inside the definition of RECKONED-TRAINS and its components.

Let us use this example to illustrate how compositional verification would work, even if only informally. The temporal property we are interested in proving is the absence of crashes between the trains: $\Box \neg \texttt{crash}$, where the proposition `crash` originates in the RECKONER and can be defined as equivalent to $\texttt{D} < 1$. However, the RECKONER by itself cannot guarantee the formula. Its satisfaction depends on the CONTROLLER, who, in turn, does not know about distances among trains.

The system RECKONED-TRAINS can guarantee $\Box \neg \texttt{crash}$ if it assumes something from the environment, namely the formula:

$$\varphi = \texttt{areConsec} \rightarrow (\neg \texttt{isSomeMoving}\, \mathbf{U}\, \texttt{isRMoving}).$$

(We are using our Boolean-valued properties as Boolean propositions.) This is something that the CONTROLLER is ready to guarantee if we replace the names of the Boolean-valued properties by the equivalent ones at its side:

$$\varphi = \texttt{areConsec} \rightarrow (\neg \texttt{doMove}\, \mathbf{U}\, \texttt{doMoveR}).$$

The synchronization criteria allow us to see these two formulas as the same one. In conclusion, we are saying that the CONTROLLER guarantees φ, the RECKONED-TRAINS guarantee $\Box \neg \texttt{crash}$ under the assumption of φ, and from that we can assert $\texttt{CONTROLLED-TRAINS} \models \Box \neg \texttt{crash}$. Adapting assume-guarantee techniques to formally justify this is left for future work.

As a last service from this example, we use it to illustrate the split operation. It allows to transform all the above into an equivalent, monolithic specification that can be fed to Maude's tools, in case it would be needed. We illustrate how it works for the smaller, but still composed system RECKONED-TRAINS. The states of the split system are triples of component stages:

```
op <_,_,_> : LTRAIN.Stage RTRAIN.Stage RECKONER.Stage -> [State] .
```

The membership axiom asserting when a triple can be considered a true state is this:

```
var L : LTRAIN.Stage .
var R : RTRAIN.Stage .
var K : RECKONER.Stage .
cmb < L, R, K > : State if isMoving @ L = isLMoving @ K
                        /\ isMoving @ R = isRMoving @ K .
```

The condition is a direct translation of the synchronization criteria.

For atomic components, the split of a rule produces two. Thus, the split of each train system produces two rules:

```
rl stopped => moving .
rl moving => stopped .
```

The split of the reckoner produces six. This makes a total of $2 \times 2 \times 6 = 24$ combinations of rules. For each of these 24 triplets of rules, one has to consider that all the three rules can execute in one step, or just one or two of them. Each of these possibilities produces a rule

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

in the resulting split system. This gives a large number of rules for a quite simple system. This is theoretically sound, but certainly not the way to go for an implementation. In some cases, like the present one, some membership conditions turn out to be trivially false, and the corresponding rules can be removed. Consider, for instance, this composed rule:

```
crl < moving, stopped, lmoving | D > => < moving, moving, lmoving | D >
   if < moving, stopped, lmoving | D > : State /\
   /\ < moving, moving, lmoving | D > : State .
```

The first membership condition is always true and the second is always false, no matter what the value of `D` is. A straightforward static analysis should be able to detect these cases and to avoid generating the rules.

In contrast, in the rule

```
crl < moving, stopped, lmoving | D > => < stopped, stopped, D - 1 >
   if < moving, stopped, lmoving | D > : State /\
   /\ < stopped, stopped, D - 1 > : State .
```

both conditions are always true, so that it must be generated, but a static analysis should be able to remove the condition from it. The implementation and all the issues related to it will be the subject of future work.

5 Related work

The literature on compositionality is vast. From it we highlight and discuss below a few items: theoretical foundations, general frameworks, language and system implementations. Although this section is big, inevitably, many interesting models, languages, and systems are left out. We hope that the items that have made it into our discussion provide a useful overview of the available landscape.

Before delving into the main topic of compositionality, we review the precedents of what we have called *egalitarianism*.

Egalitarianism. Several temporal logics have been proposed that make joint use of actions and propositions on states: ACTL* (De Nicola and Vaandrager 1990), RLTL (Sánchez and Samborski-Forlese 2012), SE-LTL (Chaki *et al*. 2004), TLR* (Meseguer 2008). There are also transition structures with mixed ingredients: LKS (Chaki *et al*. 2004), L2TS (De Nicola and Vaandrager 1995). Although all of them bring actions (or transitions) to the focus, none tries to be utterly egalitarian, as we do.

The best move toward egalitarianism we know of is the temporal logic of rewriting, TLR* (which was an inspiration for the present work). The explanations and examples in Meseguer (2008) are good arguments for an egalitarian view. Consider this formula to express fairness in the execution of a rule with label ℓ: $\Box\Diamond\,\texttt{enabled}.\ell \rightarrow \Box\Diamond\,\texttt{taking}.\ell$. The proposition $\texttt{enabled}.\ell$ is on states: it means that the current state of the system admits the rule ℓ to be applied to it. But $\texttt{taking}.\ell$ is on transitions: it means that a transition is being executed according to rule ℓ. The simplicity of the formula is only possible by being egalitarian.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Plain TLR*, as described in Meseguer (2008), stays a step away from our goal, because transitions are given by proof terms that univocally determine one origin state and one destination state for each transition. TLR* uses proof-term patterns (called *spatial actions*) that are used literally in temporal formulas. The problem is that, in this way, a TLR* formula is tied to a particular algebraic specification (one in which the pattern makes sense). In contrast, an LTL or CTL formula is meaningful by itself and can be used on any system specification using atomic proposition definitions as interfaces. Notably, propositions on transitions have been added to plain TLR*, in some way or another, in all the implementations of model checkers for (the linear-time subset of) TLR* that we are aware of Bae and Meseguer (2010; 2012; 2015) and Martín *et al.* (2014). None of them, however, tries to allow a same proposition to be defined both in states and in transitions, which we need for flexible synchronization.

Our previous work. Our paper (Martín *et al.* 2016b) contains a first definition of the synchronous composition of rewrite systems. There, we proposed to synchronize the execution of rules from different systems based on the coincidence of (atomic) rule labels. This reflects the synchronization of actions in process algebras and in automata, for example. We also proposed to synchronize states by agreement on the Boolean values of propositions defined on them. We implemented that concept of synchronization on Maude. 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 to be used with Maude's model checker. Why is the present, much more involved paper needed? We refer the reader to Section 2. 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 synchronize an action at one system with several consecutive ones at the other system. A complex realistic example like the one on the alternating bit protocol in Martín *et al.* (2018) would not be possible in our previous setting.

In a different topic, the paper Martín *et al.* (2018) also describes the use of parameterized programming to add encapsulation to our setting. We have already mentioned it in Section 2.10. We outline it roughly referring to the example from Section 4 on two controlled trains. First, a so-called theory is used to state that a train is any system that defines a Boolean-valued property called `isMoving`. Requirements for reckoners and controllers are similarly stated. These are our interface specifications. The composition is specified in a parameterized module, whose formal parameters are the theories (i.e., the interfaces). Thus, the composition can only be specified using the formal names and the properties in the interfaces. The particular implementations of trains and the other components are written and the needed properties are defined. Finally, the parameters of the composition module are instantiated with the component implementations, producing an implementation of the complete system.

Process algebras. The parallel composition operator in process algebras (like CCS and CSP) is a notable precedent of our synchronous composition. Classic references are Hoare (1978) and Milner (1980). A well-known example in CCS involves the specification of a coffee machine `CM` and a computer scientist `CS`:

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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

They are composed requiring actions `coin` and `coffee` to execute simultaneously:

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

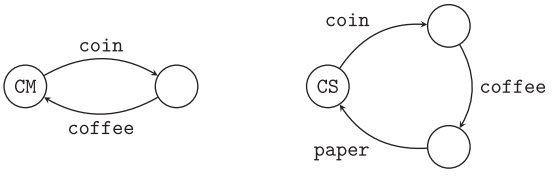
This can be translated to rewriting logic by coding two modules, `CM` and `CS`, with simple rules representing the actions, trivially defining properties, and then:

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

Process algebras were initially designed as theoretical tools. They focus on actions and synchronization, and do not provide any means to specify internal computations or to handle complex data types. However, later developments have taken process algebras as a basis for practical modeling and verification tools. Examples are occam (Welch and Barnes 2004), SCEL (De Nicola *et al.* 2015), FSP+LTSA (Magee and Kramer 2006), CSP||B (Butler and Leuschel 2005), and LOTOS and the CADP tool (Garavel *et al.* 2017).

Automata and labeled transition structures. Both automata and labeled transition structures are formally digraphs, whose nodes represent states and whose edges represent actions. Classic references are Hopcroft *et al.* (2006) for automata, and Clarke *et al.* (1999) for labeled transition structures (although they are rather called *Kripke structures* there). In automata, edges are labeled with atomic identifiers from some alphabet. In labeled 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 labeled transition structure is thought of as modeling a system; an automaton is rather thought of as accepting a language.

These two graphs represent the coffee machine and the scientist from the previous section, and can be seen either as automata or as labeled transition structures:

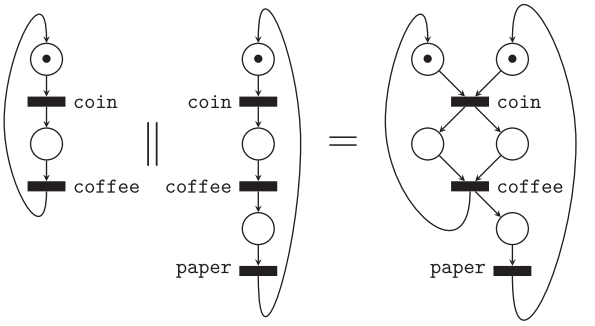


We have used labeled transition structures as semantics in this paper. However, they can be used as modeling tools by themselves. Their limitation that they share with process algebras is that they do not provide means for specifying internal computations or handling complex data types. But variations and extensions abound, and different definitions for synchronous products have been proposed; see references cited above. Worth mentioning are I/O automata (Lynch and Tuttle 1989), explicitly designed with compositionality and distributed systems in mind.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Petri nets. Petri nets do not include compositionality as a built-in ingredient. According to Sobociński (2016), the first to address this problem was Mazurkiewicz (1988). Notably, while compositionality deserves no mention in Reisig (1985) (one of the standard references on Petri nets), it is the subject of several chapters in the much more modern (Jensen and Kristensen 2009) (one of the standard references on colored Petri nets), and is also introduced in Reisig (2013).

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 modeled and then composed by fusing transitions like this:



Some approaches propose the introduction of interfaces that allow to see each component net as a black box. That is the case of the recent work described in Bruni *et al.* (2013).

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. However, for the proposals about compositionality that we are aware of, synchronization is performed only on basic terms, and value-passing is not addressed.

Tile logic. Tile logic was introduced in Gadducci and Montanari (2000), and is closely related to rewriting logic. In short, tile logic is rewriting logic with side effects for composition. A tile is written as $t \xrightarrow[b]{a} t'$ with a being the condition for, and b the effect of, rewriting t to t'. The intuitive meaning of that tile is: "the term t is rewritten to the term t', producing an effect b, but the rewrite can only happen if the variables of t (that represent as yet unspecified subcomponents) are rewritten with a cumulative effect a." Effects are given by terms of any complexity.

Tiles can be composed in three ways: vertically, so that t is first rewritten to t' and then t' to t'' according to another tile, if conditions are met; horizontally, so that several subterms of t are rewritten according to some tiles, their effects being composed; and in parallel, representing the simultaneous but independent execution of several rewrites. Conditions and effects are the boundaries (interfaces) of the tiles, and they are the only thing important for composition.

Connections between tile logic and rewriting logic have been drawn in Meseguer and Montanari (1997) and Bruni *et al.* (2002), mainly in the language of category theory.

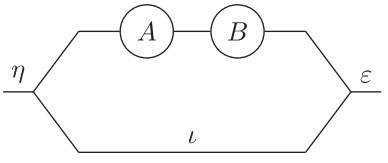
https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Span(*Graph*). The definition of Span(Graph) is based on category theory, although this is largely irrelevant for its use. It is described, for example, in Gianola *et al.* (2017). Each component is given by a graph (a kind of automaton), and each has left and right interfaces. Each action that takes place in the graph is reflected (or not) in the left and/or right interfaces.

There are three forms of composition available: parallelism without communication, parallelism with communication (where the right interface of a component gets attached to the left interface of another), and sequentiality.

The split of the interface in two parts, left and right, is an artifact only needed so that each component can be linearly assembled to other two, one in each side. This eases formalization. There is no difference for an interface to be left or right. In particular, it is not the case that left is input and right is output. Indeed, there are no in and out interfaces, it depends on how they are used (it is the same in our setting).

Complex compositions can be written as algebraic expressions. For example, for processes (graphs) A and B, a circuit with a feedback connection is written like this: $\eta * ((A * B) \otimes \iota) * \varepsilon$, and drawn like this:



That is, η is producing a fork with two copies of its input, ε is doing the opposite at the other end, and ι is identity (a simple piece of wire). The operator $*$ represents sequence, and $\otimes$ is parallelism. The circuit has to be described from left to right, even though the bottom wire is intended to work the other way.

Asynchronous messages. Asynchronous message passing can be used to achieve some extent of modular design. Actions that involve more than one component (because they implement interactions) are replaced by two or more: one or several actions in which a component produces and sends a message, and one or several actions in which the other component receives and deals with the message. The components must agree on the structure of the messages they exchange, which can be seen as the specification of the interface. The components and the messages are made to be elements of a set, which allows new elements, like messages, to be added or removed on the fly. This is a usual technique in Maude, where there is even an object-based notation, described in Clavel *et al.* (2007, Ch. 11), to ease the coding of components and messages and sets of them.

Asynchronous message passing has its limitations, though. In each component, the logic for message passing has to be mixed with the specification of its proper workings. This hinders reusability. In addition, some systems are better seen as synchronous, and are difficult to emulate with asynchronous messages. A controller, for example, would have a hard time trying to efficiently manage some component systems just by placing asynchronous messages. Also, asynchrony is not appropriate when physical components are physically geared: it is not satisfactory, nor realistic, that something that must be immediate is implemented by a multiple exchange of messages.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Behavioral programming. Use cases and scenarios are techniques used in the requirement-collection phase of software development. Behavioral programming, as described for example in Harel *et al.* (2012), proposes that it is possible to keep these up to the coding phase, transformed into synchronized threads, so-called *behavioral threads*, or just *behaviors*, each one corresponding to a use case. The concept of behavior here is really elementary. For instance, when coding the rules for the game of tic-tac-toe, there is a behavioral thread that prevents marking cell $(1, 1)$ if it is already marked; other eight behaviors do the same for the other eight cells; yet another enforces alternating turns; and so on. The paper (Harel *et al.* 2012) hints at the possibility that, for complex systems, behavioral threads may need to be grouped into *nodes*, each thread synchronized only with others in its node, and nodes synchronized by external events.

The synchronization mechanism for behavioral threads is based on the *request-wait-block* paradigm. Each thread, at each synchronization point, *requests* 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 synchronization 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 synchronization point. Separation of concerns (computation from interaction) is not perfect, as requesting, blocking, and waiting are an indispensable part of the logic of each thread.

Behavioral programming is claimed to allow incremental development, because new behaviors can be added to an already running system with no need to modify existing ones (as long as new events are not needed). Several implementations of behavioral programming are available – see at `www.wisdom.weizmann.ac.il/~bprogram`.

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. Aspect-oriented programming, as explained in Kiczales *et al.* (1997), 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. 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. Our properties can be seen as the mechanism to establish when other (aspect) code must be run. A difference is our insisting in simultaneity. Aspect code is usually executed right after or right before the need is found in the base code. For aspects, it is more difficult to prevent, and easier to react.

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 (and with a nice plug-in for Eclipse). Objects provide a first tool for modularity. In our case, all modularity is based on the synchronous composition operation.

A frequent criticism to 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

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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 behaviors 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.

Assembly theories and interface theories. Assembly theories and interface theories share the goal of establishing requirements for something to be called an assembly or interface (which are similar concepts). They are described, respectively, in Hennicker *et al.* (2014) and de Alfaro and Henzinger (2005). Assemblies and interfaces, as characterized in those papers, are rich structures that exert a distributed control on the components by prescribing how and when they can or must communicate or collaborate.

Assembly theories are more abstract than interface theories. Where interface theories consider ports and connections, assembly theories just suppose that some form of communication between components is provided. Where interface theories use automata to specify requirements to a component's interface, assembly theories assume some requirements can be made. Each at its level of abstraction, both formalisms discuss refinement, encapsulation of several components into one, composition of assemblies/interfaces. And both state needed properties such as compositionality of refinement.

A singular feature of interface theories is that they see the environment in an optimistic way, that is: a component is declared valid if there is *some* environment in which it can be used. The component restricts the environment, which must know the correct way to *use* the component. An interface specification needs not be ready for any inputs it could receive, but just for the correct ones, which makes the specification simpler. The authors claim this eases incremental design, although incremental verification requires the pessimistic view.

Coordination. The goal of coordination is to make different components work together. The components may have been coded in different languages, reside in different servers, with different architectures. The paper (Papadopoulos and Arbab 1998) is a comprehensive reference, though old.

There is a large variety of languages and systems for coordination. Some enforce complete separation of concerns between coordination and computation; others require each component to include its part of the coordination logic. Communication may be port to port, with well-defined interfaces; or there may be a central repository of data shared by all components; or anything in between. Channels may be active components, able to hold data and follow some logic, or they may be just lines joining two ports. Some models allow dynamic reconfiguration of the component network; others are more static. Communication can be synchronous or asynchronous.

Coordination is a very general term, and some of the proposals we have discussed above can be seen as belonging to it. Let us name a few additional examples. Linda, with all its variants and implementations, is one of the best known coordination languages. See Wells (2005) for a relatively recent take. It is based on the idea of a shared repository of data and relies on each component using coordination primitives in appropriate ways. REO is at the other extreme. It enforces separation of concerns and is based on composing basic

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

channels to provide port to port communication between components. Basic channels can be of any imaginable sort (synchronous or not, lossy...) and the means to compose them are very flexible. REO is described in Arbab (2004).

BIP stands for *behavior*, *interaction*, *priority* – the three *layers* of a composed specification, as proposed by the authors. The behavior of atomic components is specified by automata (of a special kind) some of whose actions are also taken as port names for communication. These automata are a specification of requirements on the component, whose real implementation can be made using any language or tool. Interaction is performed through connectors linking ports in potentially complex ways. Among the interactions that are allowed at any given time, the one with the highest priority is chosen and performed. Interaction and priority together implement control. The paper (Basu *et al*. 2008) has a good overview. Several implementations exist that allow to use the BIP framework within programming languages like Java and C++.

6 Future work

Bringing compositionality to rewriting logic in the sense discussed in this paper can open it to new fields of application like coordination models and component-based software development. Also, hardware specification could benefit. 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 analyze a real system, preferably not just a realistic one, but one drawn from the real world, 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 lines of work described below have been at least partially explored.

6.1 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 Clavel *et al*. (2007). 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, transition terms in rules, 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 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.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

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, in particular, Martín *et al.* (2018). We must consider including some of these tricks in the implementation to increase our tool's usability.

6.2 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 synchronized 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.

Using this technique, a base system, specified with all its non-deterministic capabilities, can be put under the control of different strategy-components to achieve different results. This is the idea used in Lescanne (1989); Clavel and Meseguer (1997); Verdejo and Martí-Oliet (2012) 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 Bachmair *et al.* (2003) for congruence closure. And in Martí-Oliet *et al.* (2004); Eker *et al.* (2007) 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.

The language Maude includes the nice possibility of working at the meta-level. That is, a Maude module can be *meta-represented*, stored, handled, and used as an object within another (meta-level) Maude module. Rewriting can be performed in a controlled way at the meta-level. This is a direct way to implement strategies in Maude. However, working at the meta-level 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 is why object-level strategy languages are desirable and have indeed been developed in Maude and in other formalisms (for Maude, see Martí-Oliet *et al.* 2009).

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 synchronization criteria to compose them.

6.3 Compositional verification

Compositional specification is very nice, but it gets even better if one can go on working compositionally, 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*, initially proposed in Pnueli (1985). According to it, each component assumes some nice behavior from the rest of the system and, under such an assumption, guarantees its own nice behavior.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

If each component can be proven to satisfy some condition of this type, conclusions can be drawn on the whole composed system. We have already mentioned this at the end of Section 3.1.4.

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 compositional model checking.

7 Conclusion

We are confident that the most appealing parts of our work are still to come. Strategies, compositional 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 transition terms instead of the usual atomic labels. We have proposed a flexible means for specifying how several rewrite systems are synchronized, 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 transition structures as well, so that our rewrite systems get a semantic ground. We have described the *split* operations that translate sets of synchronized egalitarian rewrite systems into standard ones.

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

References

Arbab, F. 2004. REO: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14,* 3, 329–366.

Bachmair, L., Tiwari, A. and Vigneron, L. 2003. Abstract congruence closure. *Journal of Automated Reasoning 31,* 2, 129–168.

Bae, K. and Meseguer, J. 2010. The linear temporal logic of rewriting Maude model checker. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20–21, 2010, Revised Selected Papers*, P. C. Ölveczky, Ed. Lecture Notes in Computer Science, vol. 6381. Springer, 208–225.

Bae, K. and Meseguer, J. 2012. A rewriting-based model checker for the linear temporal logic of rewriting. *Electronic Notes in Theoretical Computer Science 290*, 19–36.

Bae, K. and Meseguer, J. 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming 99*, 193–234.

Basu, A., Bozga, M. and Sifakis, J. 2008. Modeling heterogeneous real-time components in BIP. In *Perspectives Workshop: Model Engineering of Complex Systems (MECS), August 10–13, 2008*, U. Aßmann, J. Bézivin, R. F. Paige, B. Rumpe and D. C. Schmidt, Eds. Dagstuhl Seminar Proceedings, vol. 08331. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

Boudol, G. and Castellani, I. 1988. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae 11*, 433–452.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Bruni, R., Melgratti, H. C., Montanari, U. and Sobociński, P. 2013. Connector algebras for C/E and P/T nets' interactions. *Logical Methods in Computer Science 9,* 3.

Bruni, R., Meseguer, J. and Montanari, U. 2002. Tiling transactions in rewriting logic. *Electronic Notes on Theoretical Computer Science 71*, 90–109.

Bruns, G. and Godefroid, P. 1999. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification: 11th International Conference, CAV '99*, N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, Trento, Italy, 274–287.

Bruns, G. and Godefroid, P. 2000. Generalized model checking: Reasoning about partial state spaces. In *CONCUR 2000—Concurrency Theory: 11th International Conference*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1877. Springer, University Park, PA, USA, 168–182.

Butler, M. and Leuschel, M. 2005. Combining CSP and B for specification and property verification. In *FM 2005: Formal Methods*, J. Fitzgerald, I. J. Hayes and A. Tarlecki, Eds. Springer, Berlin, Heidelberg, 221–236.

Chaki, S., Clarke, E. M., Ouaknine, J., Sharygina, N. and Sinha, N. 2004. State/event-based software model checking. In *IFM*, E. A. Boiten, J. Derrick, G. Smith, Eds. Lecture Notes in Computer Science, vol. 2999. Springer, 128–147.

Clarke, E. M. J., Grumberg, O. and Peled, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. L. 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, Heidelberg.

Clavel, M. and Meseguer, J. 1997. Internal strategies in a reflective logic. In *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, B. Gramlich and H. Kirchner, Eds. Springer, Townsville, Australia, 1–12.

de Alfaro, L. and Henzinger, T. A. 2005. Interface-based design. In *Engineering Theories of Software Intensive Systems*, M. Broy, J. Grünbauer, D. Harel and T. Hoare, Eds. Springer Netherlands, Dordrecht, 83–104.

De Nicola, R., Fantechi, A., Gnesi, S. and Ristori, G. 1993. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems 25,* 7, 761–778.

De Nicola, R., Latella, D., Lafuente, A. L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F. and Vandin, A. 2015. The SCEL language: Design, implementation, verification. In *Software Engineering for Collective Autonomic Systems: The ASCENS Approach*, M. Wirsing, M. Hölzl, N. Koch and P. Mayer, Eds. Springer International Publishing, Cham, 3–71.

De Nicola, R. and Vaandrager, F. W. 1990. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, I. Guessarian, Eds. Lecture Notes in Computer Science, vol. 469. Springer, 407–419.

De Nicola, R. and Vaandrager, F. 1995. Three logics for branching bisimulation. *Journal of the Association for Computing Machinery 42,* 2, 458–487.

Eker, S., Martí-Oliet, N., Meseguer, J. and Verdejo, A. 2007. Deduction, strategies, and rewriting. In *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, M. Archer, T. B. de la Tour and C. Muñoz, Eds. Electronic Notes in Theoretical Computer Science, vol. 174. Elsevier, Seattle, WA, USA, 3–25.

Gadducci, F. and Montanari, U. 2000. The tile model. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, G. D. Plotkin, C. Stirling and M. Tofte, Eds. The MIT Press, 133–166.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Garavel, H., Lang, F. and Serwe, W. 2017. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, J.-P. Katoen and R. Langerak, Eds. Lecture Notes in Computer Science, vol. 10500. Springer, 3–26. DOI: 10.1007/978-3-319-68270-9_1.

Gianola, A., Kasangian, S. and Sabadini, N. 2017. Cospan/span(graph): An algebra for open, reconfigurable automata networks. In *7th Conference on Algebra and Coalgebra in Computer Science, CALCO 2017, June 12–16, 2017, Ljubljana, Slovenia*, F. Bonchi and B. König, Eds. LIPIcs, vol. 72. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2:1–2:17.

Godefroid, P. and Huth, M. 2005. Model checking vs. generalized model checking: Semantic minimizations for temporal logics. In *Proceedings of 20th Annual IEEE Symposium on Logic in Computer Science (LICS '05)*. IEEE, Chicago, IL, USA, 158–167.

Harel, D., Marron, A. and Weiss, G. 2012. Behavioural Programming. *Communications of the Association for Computing Machinery 55,* 7, 90–100.

Hennicker, R., Knapp, A. and Wirsing, M. 2014. Assembly theories for communication-safe component systems. In *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, S. Bensalem, Y. Lakhnech and A. Legay, Eds. Lecture Notes in Computer Science, vol. 8415. Springer, 145–160.

Hoare, C. A. R. 1978. Communicating sequential processes. *Communications of the Association for Computing Machinery 21,* 8, 666–677.

Hopcroft, J. E., Motwani, R. and Ullman, J. D. 2006. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Huth, M., Jagadeesan, R. and Schmidt, D. 2001. Modal transition systems: A foundation for three-valued program analysis. In *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001*, D. Sands, Ed. Lecture Notes in Computer Science, vol. 2028. Springer, Genova, Italy, 155–169.

Jensen, K. and Kristensen, L. M. 2009. *Coloured Petri Nets*. Springer, Berlin, Heidelberg.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J. and Lopes, C. 1997. Aspect-oriented programming. In *ECOOP '97—Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, Jyväskylä, Finland, 220–242.

Kindler, E. and Vesper, T. 1998. ESTL: A temporal logic for events and states. In *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN '98*, J. Desel and M. Silva, Eds. Lecture Notes in Computer Science, vol. 1420. Springer, Lisbon, Portugal, 365–384.

Lescanne, P. 1989. Completion Procedures as Transition Rules + Control. In *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, J. Díaz and F. Orejas, Eds. Lecture Notes in Computer Science, vol. 351. Springer, Berlin, Heidelberg, 28–41.

Lynch, N. A. and Tuttle, M. R. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 219–246.

Magee, J. and Kramer, J. 2006. *Concurrency - State Models and Java Programs* (2nd ed.). Wiley.

Martí-Oliet, N., Meseguer, J. and Verdejo, A. 2004. Towards a strategy language for Maude. In *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*, N. Martí-Oliet, Ed. Electronic Notes in Theoretical Computer Science, vol. 117. Elsevier, Barcelona, Spain, 417–441.

Martí-Oliet, N., Meseguer, J. and Verdejo, A. 2009. A rewriting semantics for Maude strategies. In *Proceedings of the Seventh International Workshop on Rewriting Logic and its*

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Applications (WRLA 2008), G. Rosu, Ed. Electronic Notes in Theoretical Computer Science, vol. 238. Elsevier, Budapest, Hungary, 227–247.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2014. Model checking TLR* guarantee formulas on infinite systems. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, S. Iida, J. Meseguer and K. Ogata, Eds. Lecture Notes in Computer Science, vol. 8373. Springer, 129–150.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2016a. Egalitarian state-transition systems. In *Rewriting Logic and Its Applications: WRLA 2016*, D. Lucanu, Ed. Lecture Notes in Computer Science, vol. 9942. Springer, Eindhoven, The Netherlands, 98–117.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2016b. Synchronous products of rewrite systems. In *Automated Technology for Verification and Analysis: ATVA 2016*, C. Artho, A. Legay and D. A. Peled, Eds. Lecture Notes in Computer Science, vol. 9938. Springer, Cham, 141–156.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2018. Parameterized programming for compositional system specification. In *Rewriting Logic and Its Applications: WRLA 2018*, V. Rusu, Ed. Lecture Notes in Computer Science, vol. 11152. Springer.

Mazurkiewicz, A. 1988. Compositional semantics of pure place/transition systems. In *Advances in Petri nets: APN 1987*, G. Rozenberg, Ed. Lecture Notes in Computer Science, vol. 340. Springer, Oxford, UK, 307–330.

Meseguer, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96,* 1, 73–155.

Meseguer, J. 2008. The temporal logic of rewriting: A gentle introduction. In *Concurrency, Graphs and Models*, P. Degano, R. D. Nicola and J. Meseguer, Eds. Lecture Notes in Computer Science, vol. 5065. Springer, Berlin, Heidelberg, 354–382.

Meseguer, J. and Montanari, U. 1997. Mapping tile logic into rewriting logic. In *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT '97, Tarquinia, Italy, June 1997, Selected Papers*. Lecture Notes in Computer Science, vol. 1376. Springer, 62–91.

Meseguer, J. and Thati, P. 2007. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation 20*, 123–160.

Milner, R. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer-Verlag, Berlin, Heidelberg.

Papadopoulos, G. A. and Arbab, F. 1998. Coordination models and languages. *Advances in Computers 46*, 329–400.

Pnueli, A. 1985. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Springer, Berlin, Heidelberg, 123–144.

Reisig, W. 1985. *Petri Nets: An Introduction*. EATCS Monographs in Theoretical Computer Science. Springer, Berlin, Heidelberg.

Reisig, W. 2013. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, Berlin, Heidelberg.

Sánchez, C. and Samborski-Forlese, J. 2012. Efficient regular linear temporal logic using dualization and stratification. In *Proceedings - 2012 19th International Symposium on Temporal Representation and Reasoning, TIME 2012*, 13–20.

Sobociński, P. 2016. Compositional model checking of concurrent systems, with Petri nets. In *Developments in Computational Models: DCM 2015 Proceedings*, C. A. Muñoz and J. A. Pérez, Eds. Electronics Proceedings in Theoretical Computer Science, vol. 204. Open Publishing Association, Cali, Colombia, 19–30.

Verdejo, A. and Martí-Oliet, N. 2012. Basic completion strategies as another application of the Maude strategy language. In *Workshop on Reduction Strategies in Rewriting and Programming (WRS2011)*, S. Escobar, Ed. Electronic Proceedings in Theoretical Computer Science, vol. 82. Open Publishing Association, Novi Sad, Serbia, 17–36.

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press

Welch, P. H. and Barnes, F. R. M. 2004. Communicating mobile processes. In *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7–8, 2004, Revised Invited Papers*, A. E. Abdallah, C. B. Jones and J. W. Sanders, Eds. Lecture Notes in Computer Science, vol. 3525. Springer, 175–210.

Wells, G. 2005. New issues on coordination and adaptation techniques. In *Proceedings of the Second International Workshop on Coordination and Adaptation Techniques for Software Entities WCAT'05*, 25 Jul. 2005, S. Becker, C. Canal, J. M. Murillo, P. Poizat and M. Tivoli, Eds. Glasgow, Scotland, 87–89. Held in conjunction with ECOOP 2005, Technical Report TR ITI-05-07, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga. URL: https://www.cs.cmu.edu/∼jcmoreno/files/WCAT05Proceedings.pdf. [Accessed on September 26, 2019].

https://doi.org/10.1017/S1471068419000425 Published online by Cambridge University Press