TPLP **24** (1): 57–109, 2024.

doi:10.1017/S1471068423000340 First published online 31 August 2023

Compositional Verification 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 7 April 2022; revised 31 May 2023; accepted 1 August 2023

Abstract

In previous work, summarized in this paper, we proposed an operation of parallel composition for rewriting-logic theories, allowing compositional specification of systems and reusability of components. The present paper focuses on compositional verification. We show how the assume/guarantee technique can be transposed to our setting, by giving appropriate definitions of satisfaction based on transition structures and path semantics. We also show that simulation and equational abstraction can be done componentwise. Appropriate concepts of fairness and deadlock for our composition operation are discussed, as they affect satisfaction of temporal formulas. We keep in parallel a distributed and a global view of composed systems. We show that these views are equivalent and interchangeable, which may help our intuition and also has practical uses as, for example, it allows global-style verification of a modularly specified system. *Under consideration in Theory and Practice of Logic Programming (TPLP).*

KEYWORDS: rewriting logic, modularity, verification, assume/guarantee, abstraction, simulation, Maude

1 Introduction

Rewriting logic (Meseguer 1992) is a well established, logic-based formalism, useful, in particular, for the specification of concurrent and nondeterministic systems. There are ways, in this context, in which modularity can be achieved. The language Maude (Clavel *et al*. 2022), for example, strongly based on rewriting logic, includes a powerful system of modules which promotes a good organization of the code. Besides, multicomponent or distributed systems are sometimes modeled as a multiset of objects and messages. However, a truly compositional specification was not possible. By that, we mean one in which each component is an independent rewrite system and composition is specified separately, allowing, for example, reusability of components. In previous work (Martín *et al*. 2020), we proposed an operation of parallel composition of rewrite systems to

[*] Work partially supported by project S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid, and project PID2019-108528RB-C22 (ProCode-UCM) funded by Ministerio de Ciencia e Innovación (Spanish Government). The authors want to thank David de Frutos for his important remark on the correct way to define compatibility of paths, and the anonymous referees for improving this paper with their hard and very useful work.

achieve precisely that. In the present paper, we show how a compositional specification written according to our proposal can be the object of compositional verification. Note that, in this work we often use *rewrite system* as a shorthand for *system specified using rewriting logic*.

The reasons for the convenience of a compositional approach to verification are well known: to avoid the state-explosion problem; because some systems are inherently compounds and it makes all sense to specify and verify them as such; because verified systems can be safely reused as library components.

There are two alternative views on the meaning of compositional specification, which lead to different needs for compositional verification. In one, a composed specification is seen as modeling a distributed system, of which probably only one component is under our control, and the aim of verification is to ensure that our component behaves appropriately in an appropriate environment. Global states are out of the question, and the behavior we focus on is that of our component. The assume/guarantee technique (see Section 9) is designed to be helpful here.

In the other view, in contrast, the whole system is under our control, but working compositionally still makes sense for modular engineering. Then, the aim of compositional verification is to prove that each component behaves appropriately, not in a general, unknown environment, but in the particular one given by the rest of the components that we have also specified. The abstraction technique (see Section 8.2) helps here: given a component and its environment, we can abstract either or both, and perform verification on the abstracted, simplified component and/or environment. Assume/guarantee may also help. For example, we use both techniques in the mutual exclusion example introduced in Section 2.2.

We consider special kinds of atomic and composed rewrite systems which we call *egalitarian* and were introduced in our previous work (Martín *et al*. 2020). They are egalitarian in the sense that they give the same status to transitions and states. A composed egalitarian system is a set of independent but interacting atomic ones. We see them as modeling a distributed system. An egalitarian rewrite system, atomic or composed, can be translated into a standard rewrite system (called *plain* in this work) by the operation that we call the *split*. This allows to specify a system componentwise, translate the compound into a single plain system and, then, execute and verify the result monolithically using existing tools (the ones in Maude's toolset, for example). The relation between this monolithic verification and the compositional one using assume/guarantee is our Theorem 5.

We are interested in rewriting logic and, in this paper, in verifying systems specified using that logic. The underlying expectation is that a firm logical basis makes it easier to define, study, and implement modularity and composition. However, satisfaction of temporal formulas is defined on the transition structures which represent the semantics of the logical specifications. Thus, transition structures play a fundamental role in this paper, even if only as proxies for the main characters.

This is the plan of the paper. In Section 2 we show and explain the compositional specification of three simple but illustrative examples. They are revisited later in the paper, but here they are meant as an informal introduction to our previous work on composition. Section 3 contains a quick and mainly formal overview of our previous work. In Section 4, we study execution paths, needed to define satisfaction of formulas.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

We consider paths in atomic components, sets of compatible paths from different components, and global paths, representing, respectively, local, distributed, and global behaviors. The contrast and equivalence between a local view and a global one is a constant throughout the paper. In Section 5, we describe the variant of the temporal logic LTL against which we verify our systems. In Section 6, we define basic satisfaction of temporal formulas based on paths, and show the relation between the distributed and the global views of satisfaction. In Section 7, we discuss the concepts of fairness and deadlocks, and their importance for compositional verification. In Section 8, we consider the componentwise use of simulation and abstraction: simulating or abstracting a component induces the same on the whole system, with a potentially reduced effort. In addition, the abstracted system may be easier to verify. In Section 9, we consider the assume/guarantee technique, which allows the verification of isolated individual components, ensuring thus that the result holds for whatever appropriate environment the component is placed in, and we show how it can be adapted to our setting. In Section 10, we briefly present two additional examples of compositional specification and verification (fully discussed in (Martín 2021a)) which are more complex and realistic than the toy ones used throughout this paper. Finally, Section 11 discusses related and future work and contains some closing remarks.

These are the points we think may be of special interest in this paper:

- We show how to work compositionally in rewriting logic, expanding and strengthening our previous work.
- We keep in parallel, all throughout the paper, the distributed and the global (monolithic) views of satisfaction and related concepts, and show the equivalence of both views. We claim that keeping both views is worth the effort, both for our intuition and in practice.
- We show how simulation and abstraction can be performed compositionally.
- We show that the assume/guarantee technique can be transposed to our setting.
- Our definition of assume/guarantee satisfaction (inductive, but not relying on the *next* temporal operator) is new, to the best of our knowledge.
- We give path-based definitions of deadlock and fairness, discuss how they impact the verification tasks, and show how to deal with them in our setting.

2 Examples

We introduce here three examples of compositional specification. They are meant as a quick introduction to Maude and to our previous work (Martín *et al.* 2020), especially to compositional specification with the extended syntax we proposed. The formal definitions and results are in Section 3. Also, these examples set the base on which we later illustrate the techniques for compositional verification and simulation. They have been chosen to be illustrative, so they are quite simple. We are using Maude because of its availability, its toolset, and its efficient implementation. All the concepts and examples, however, are valid for rewriting logic in general.

The first example presents three buffers assembled in line. The second shows how to exert an external mutual exclusion control on two systems, provided they inform on when they are visiting their critical sections. Later, this gives us the opportunity of

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

using componentwise simulation and a very simple case of assume/guarantee. The third example concerns the well-known puzzle of a farmer and three belongings crossing a river. We compose a system implementing the mere rules of the puzzle with several other components implementing, in particular, two guidelines which prove to be enough to reach a solution. The assume/guarantee technique is later used on this system.

The complete specification for all the examples in this paper is available online (Martín 2021b). Our prototype implementation, able to deal with these examples, is also available there, though the reader is warned that, in its current state, it is not a polished tool but, rather, a proof of concept.

2.1 Chained buffers

We model a chain of three buffers. We describe the system top-down. This is the specification of the composed system:

```
sync BUFFER1 || BUFFER2 || BUFFER3
   on BUFFER1$isSending = BUFFER2$isReceiving
   /\ BUFFER2$isSending = BUFFER3$isReceiving .
```

The `sync...on...` sentence is not standard Maude, but part of our extension. That sentence expects three Maude modules to exist, called `BUFFER1`, `BUFFER2`, and `BUFFER3`, each defining the values of the so-called *properties* mentioned in the `on` part of the sentence: `isSending` and `isReceiving`. We use the `$` sign to access a property defined in a Maude module. In words, that models a composed system in which the three buffers synchronize so that when one sends the next receives. The properties are assumed to be Boolean in this example, modeling the passing of tokens. Synchronizing on more complex values is also possible, as shown in other examples.

We call the result of the composition above `3BUFFERS`. For this to be a complete model, we need to provide the specification of the internal workings of the three buffers, including the definition of the properties. There is no reason for the three buffers to be specified exactly the same. In principle, they even could be coded in different languages, as long as there is a way to access the values of the properties defined inside each of them. For the sake of simplicity, in this example the three modules are identical. This is the very simple code for each of them:

```
sort State Trans .
ops idle gotToken : -> State .
ops receiving sending : -> Trans .
rl idle      =[ receiving ]=>  gotToken .
rl gotToken  =[ sending   ]=>  idle .
```

There are two states, represented by the `State` constants `idle` and `gotToken`, and two transitions between them, represented by the `Trans` constants `receiving` and `sending`. The keyword `ops` introduces the declaration of operators with their arities. The singular `op` can be used when only one operator is being declared. In this code, we are declaring constants, so the argument sorts are absent. The keyword `rl` introduces each rewrite rule and the symbols `=[` and `]=>` separate the terms. We assume throughout the paper that the sort representing the states of the system is called `State` and the one representing transitions is called `Trans`. Also, it is convenient to have a supersort of both (not shown above), which we call `Stage`. Usually, we omit declarations of sorts and operators when they are clear from context.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Readers knowledgeable of rewriting logic and Maude would expect the rules above to be written instead as:

```
rl [receiving] : idle     => gotToken .
rl [sending]   : gotToken => idle .
```

Here, `receiving` and `sending` are rule labels. The syntax we use does not only consists of moving the label to the middle of the rule. In our case, `receiving` and `sending` are not labels, but algebraic terms of sort `Trans`, in the same way that `idle` and `gotToken` are terms of sort `State`. In general, both `States` and `Transs` can be terms of any algebraic complexity. Other examples below make this clearer.

We call these rules *egalitarian*, because transitions are represented by terms, the same as states. The rewrite systems which include them are also called *egalitarian*. More precisely, each buffer is an *atomic egalitarian rewrite system*. The result of their composition is still called *egalitarian*, but not atomic.

As illustrated above, the way we have chosen to specify composition of systems is by equality of *properties*. These are functions which take values at each state and transition of each component system. The properties of a system provide a layer of isolation between the internals of each component and the specification of the composition. This is similar to the concept of ports in other settings. It is important that properties are defined not only on states, but also on transitions, because synchronization is more often than not specified on them. That is why we have developed egalitarian systems in which transitions are promoted to first-class citizenship.

We declare and define two properties in each buffer:

```
ppt isReceiving isSending : -> Bool .
eq isReceiving @ receiving = true .
eq isReceiving @ G = false [owise] .
eq isSending @ sending = true .
eq isSending @ G = false [owise] .
```

The sentence introduced by the keyword **`ppt`** is part of our extended syntax, as is the symbol `@` representing the evaluation of a property on a state or transition. Thus, these lines declare two Boolean properties and define by means of equations (introduced by the keyword **`eq`**) their values at each state and transition. The fact that `receiving` and `sending` are algebraic terms allows their use in equations.

The attribute **`owise`** (short for *otherwise*) in two of the equations is an extralogical feature of Maude: that equation is used whenever the term being reduced matches the left-hand side and the case is not dealt with by other equations. The variable `G`, whose declaration is not shown, has sort `Stage`, so that all properties evaluate to `false` except in the two cases explicitly set to `true`.

Any property defined in a component can be used as well as a property for the resulting composed system. In this case, the properties `isReceiving` in `BUFFER1` and `isSending` in `BUFFER3` are defined but not used for synchronization. Those properties can be useful if the composed module `3BUFFERS` is used in turn as a component to be synchronized with other modules.

It is a common case that a property is defined to be true exactly at one state or transition and false everywhere else, as above. This calls for some syntactic shortcut to help the user. We do not discuss in this paper how to implement such shortcuts (of which this is not at all the only possible one), and our prototype implementation does not include them.

The execution of the composed system `3BUFFERS` consists in the independent execution of each of its three components, restricted by the need to keep the equality between properties. To that composed system, the operation we call the *split* can be applied to obtain an equivalent standard rewrite system. The resulting split system has as states triples like `< idle, gotToken, idle >`, formed from the states of the components, and has rewrite rules like

```
rl < idle, gotToken, idle > => < idle, sending, receiving > .
```

The split is named after this translation of each rule into two *halves*. The term *split* is also used later to describe related translations, though in some of those cases there is nothing split in the literal sense. The split is formally defined in Section 3.3. We usually do not care to show the internal appearance of a split system, but are only interested in the fact that it represents in a single system the global behavior of the composition.

2.2 Mutual exclusion

Consider a very simple model of a train, which goes round a closed railway in which there are three stations and a crossing with another railway. We use the three stations as the states of our model, and there are three transitions for moving between them. Using our extended syntax, we model it with the rule:

```
crl atStation N   =[ comingFrom N ]=>   atStation (N + 1)   if N < 2 .
```

The keyword **`crl`** introduces a conditional rewrite rule. We omit the needed declarations for the integer variable `N` and the constructors `atStation` and `comingFrom`.

The stations are numbered 0–2. But the transit from station 2 to 0 is different, because it passes through the crossing:

```
rl atStation 2   =[ crossing ]=>   atStation 0 .
```

Indeed, we have two trains, modeled in this example by the same specification, but as two separate components. They share the crossing, so we need safety in the access to it. To this aim, we define for each train a Boolean property `isCrossing` to be true at the transition `crossing` and false everywhere else:

```
ppt isCrossing : -> Bool .
eq isCrossing @ crossing = true .
eq isCrossing @ G = false [owise] .
```

We call the two systems thus defined `TRAIN1` and `TRAIN2`.

The mutex controller for safe access to the crossing is specified by these two rules:

```
rl idle   =[ grants 1 ]=>   idle .
rl idle   =[ grants 2 ]=>   idle .
```

We call this system `MUTEX` and define in it the parametric Boolean property `isGranting`, which is defined to be true at the respective transitions and false everywhere else:

```
ppt isGranting : Nat -> Bool .
eq isGranting(I) @ (grants I) = true .
eq isGranting(I) @ G = false [owise] .
```

The final system is the composition of the two trains and `MUTEX` so that each `isCrossing` property is synchronized with the corresponding `isGranting` one:

```
sync TRAIN1 || TRAIN2 || MUTEX
   on TRAIN1$isCrossing = MUTEX$isGranting(1)
   /\ TRAIN2$isCrossing = MUTEX$isGranting(2) .
```

In due time, in Sections 8.3 and 9.2, we will show how we can use simulation to work with even simpler models of the trains, and how we can justify that mutual exclusion holds for the composed system.

We want to insist in the value of modularity in our examples. The system MUTEX with its two properties can be used unchanged to control any two given systems, as long as they inform, by means of properties, of their being in their critical section. For general systems, the synchronization instruction would look something like

```
sync ONE-SYSTEM || ANOTHER-SYSTEM || MUTEX
   on ONE-SYSTEM$isInCS = MUTEX$isGranting(1)
   /\ ANOTHER-SYSTEM$isInCS = MUTEX$isGranting(2) .
```

Mutual exclusion between the two systems, whatever they are, is guaranteed by MUTEX satisfying the appropriate formula – see Section 9.2.

We find cases like this of particular interest. We mean a component controlling others and imposing its behavior (mutual exclusion in this case) on the compound. This is the idea behind strategies, controllers, coordination, etc. In contrast, in the example of the chained buffers in Section 2.1, the composed behavior is emergent. Our next example involves both techniques.

2.3 Crossing the river

For a quick reminder, this is the statement of the puzzle. A farmer has got a wolf, a goat and a cabbage, and needs to cross a river using a boat with capacity for the farmer and, at most, one of the belongings. The wolf and the goat should not be left alone, because the wolf would eat the goat. In the same way, the goat would eat the cabbage if left unattended. The goal is to get the farmer and the three belongings at the opposite side of the river safely.

Our specification consists of two rules: one encompasses all possible ways the farmer can cross the river; the other represents eating. This is the rule for a crossing, explained below:

```
rl   farmer B? II1  |~|     II2
  =[ II1            | B? >  II2           ]=>
     II1            |~|     farmer B? II2 .
```

Each state term contains the symbol |~| representing the river. To each side of this symbol there is a set of items, which may include the farmer and the three belongings, respectively represented by the constants farmer, wolf, goat, and cabbage. Also, there is always a special item mark which marks the side that the farmer is trying to reach with her belongings. Thus, the initial state is defined like this:

```
eq init = farmer wolf goat cabbage |~| mark .
```

The variables II1 and II2 are sets of items which, in particular, may be empty. The sort of the variable B? is MaybeBelong, that is, either one of the three belongings or the special value noBelong. Indeed, noBelong is also the identity element for sets of items. In this way, the transition term II1 | B? > II2 represents all possible crossings, with B? = noBelong interpreted as the farmer crossing alone. The symbol |~| is formally a commutative operator, so that the same rule represents movements from any side to the other. That rule is rather terse. Alternative specifications, using more than one rule, would probably be easier to grasp. That is not important for the main purpose of this paper, which has to do with composition.

The rule for eating is this one:

```
rl   goat B II1       |~|  farmer II2
  =[ eating B II1     |~|  farmer II2 ]=>
     survivor(B) II1  |~|  farmer II2 .
```

Thus, when the goat and some other belonging are at one side with the farmer at the other side, eating can take place. The function `survivor` is defined by these equations:

```
eq survivor(cabbage) = goat .
eq survivor(wolf) = wolf .
```

Thus, the goat survives if the other belonging is the cabbage, but it dies (disappears from the state term) if the other belonging is the wolf.

Our specification does not require that eating happens as soon as it is possible, but only that it *can* happen. So our aim is to avoid all danger and ensure a safe transit.

This was the specification of the rules of the game. We propose now two guidelines for the farmer to follow. The first is to avoid all movements which lead to a dangerous situation, that is, one with the goat and some other belonging left by themselves. The second is to avoid undoing the most recent crossing: for example, after crossing one way with the goat, avoid going back the other way with the goat again. These are both quite obvious guidelines to follow, and we hypothesize that they are enough to ensure that the farmer reaches the goal. As it turns out, the hypothesis is false, and we will need to strengthen the second guideline; but let us work with this for the time being.

The guidelines are enforced by avoiding certain transitions to be triggered. For that, we need to identify said transitions. First, the dangerous ones:

```
ppt danger : -> Bool .
eq danger @ (goat B II1 | B? > II2) = true .
eq danger @ G = false [owise] .
```

The variable `B` represents a belonging, while `B?`, as before, can be either a belonging or `noBelong`. In words: there is danger if the farmer is in the boat and the goat has been left alone with some other belonging.

We need to restrict the execution of `RIVER` so that `RIVER$danger = false` at all times. This is another instance where a syntactic shortcut would help, but also this requirement can be enforced by a composition with an appropriate controller.

Let us call the following system `AVOID`. It is as simple as a system can possibly be:

```
op init : -> State .
ppt avoid : -> Bool .
eq avoid @ init = false .
```

There is a single state, called `init`, no transitions and no rules, and the property `avoid` is always false. Thus, the composed system

```
sync RIVER || AVOID
   on RIVER$danger = AVOID$avoid .
```

indeed avoids all situations at which `danger` is true.

Implementing the other guideline, avoidance of the undoing of movements, requires one more step, because we need to, somehow, store the previous movement so as to be able to compare it with the potential new one. We are after a composed system like this

```
sync RIVER || PREVIOUS
   on PREVIOUS$move = RIVER$move .
```

where `RIVER` *informs* the new system `PREVIOUS` about the moves being made, and `PREVIOUS` stores at each moment the latest move. We name this composed system `RIVER-W-PREV`.

The new component PREVIOUS needs only this rule:

```
rl B?   =[ B? > B?’ ]=>   B?’ .
```

Its state sort is MaybeBelong, that is, either actually one of the three belongings or the value noBelong. In this case they are representing movements: the farmer crossing either with the specified belonging or alone. The transition term includes two such movements: the previous one and the new one. In this way, we can check them for equality when needed. To synchronize with the main system RIVER, we use this property in PREVIOUS:

```
ppt move : -> MaybeMove .
eq move @ (B? > B?’) = B?’ .
eq move @ B? = noMove .
```

Correspondingly, we need this property in RIVER:

```
ppt move : -> MaybeMove .
eq move @ (II1 | B? > II2) = B? .
eq move @ G = noMove [owise] .
```

We need to include the new constant noMove for when, indeed, no move is taking place.

Storing information about the past execution of the system is called *instrumentation* and is a common technique in system analysis. This is another instance calling for syntactic sugar. As shown with the RIVER || PREVIOUS example, it can be achieved by composition of atomic rewrite systems.

Whenever RIVER is executing a crossing, PREVIOUS is showing, in its transition term, the previous and the current moves, giving us the possibility of checking if they are equal:

```
ppt undoing : -> Bool .
eq undoing @ (B? > B?) = true .
eq undoing @ G = false [owise] .
```

Now, we need to restrict RIVER-W-PREV so as to avoid undoing movements. For that, we can use AVOID, as above. But we need two instances of that system, one to avoid danger, the other to avoid undoings, to which we refer as AVOID1 and AVOID2.

At the end, the system we are interested in is

```
sync RIVER-W-PREV || AVOID1 || AVOID2
  on RIVER$danger = AVOID1$avoid
  /\ PREVIOUS$undoing = AVOID2$avoid .
```

This completes the specification of the system. Later in the paper, in Section 9.3, we show how to verify that it leads to a solution... or, rather, that it does not. But we will also show a sufficient strengthening of the concept of undoing.

As in the previous examples, we want to draw the reader’s attention to the modularity of our specification. Some previous treatments of this problem in rewriting logic (Palomino *et al.* 2005; Rubio *et al.* 2021) used several rules to model the different ways of crossing. But this is irrelevant to us, because any specification that defines the properties move and danger will do as well.

3 Background

This section is a formal summary of our previous work on the synchronous composition of rewrite systems (Martín *et al.* 2020). Detailed explanations and proofs can be found there. This whole section is quite theoretical, consisting of many definitions and a few propositions, to complement the informal and example-based introduction in Section 2.

We define below a number of structures and systems. This is a list of them with the abbreviations we use to refer to them:

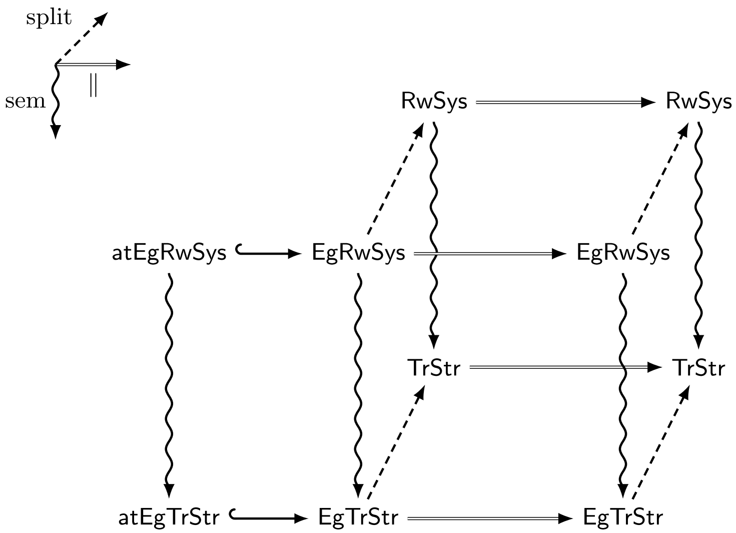


Fig. 1. The types of systems we use and their relations.

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

The polyhedron in Figure 1 shows the whole set of structures and systems with their related maps. Slanted dashed arrows represent the several concepts of split, that is, of obtaining plain transition structures or rewrite systems from egalitarian ones. Double horizontal arrows represent synchronous composition of systems or structures: composing systems or structures of the same kind produces another one of the same kind. Downward snake arrows represent semantic maps, assigning transition structures to rewrite systems. The two horizontal hooked arrows on the left represent inclusion: atomic systems and structures are particular cases of general systems and structures, respectively. All the elements in the diagram are defined below, and better explained in our previous paper (Martín *et al.* 2020).

3.1 Egalitarian structures and systems

As we mentioned above, we use transition structures (of particular types) as semantics for our rewrite systems. In due time, we define execution paths for transition structures, and satisfaction based on those paths. In this section we define atomic egalitarian transition structures, atomic egalitarian rewrite systems, the semantic relation between them, and their compositions.

Definition 1 (*atomic egalitarian transition structure*)
An *atomic egalitarian transition structure* is a tuple $\mathcal{T} = (Q, T, \rightarrow, P, g_0)$, where:

- Q is the set of states;
- T is the set of transitions;

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

- $\rightarrow \subseteq (Q \times T) \cup (T \times Q)$ is the bipartite adjacency relation;
- P is the set of properties, each one a total function p from $Q \cup T$ to some codomain C_p;
- $g_0 \in Q \cup T$ is the initial state or transition.

We refer to the elements of $Q \cup T$ as *stages*. The class of atomic egalitarian transition structures is denoted by $\mathsf{atEgTrStr}$.

The adjacency relation allows for several arrows in and out of a transition, as well as a state. The egalitarian goal also mandates that not only an initial state is possible, but also an initial transition. We use variables typically called g, with or without ornaments, to range over stages.

The definition of an atomic egalitarian transition structure is almost identical to that of a Petri net. The difference, however, is in the semantics: we are interested in a simple path semantics, instead of sets of marked places. This is better explained in Section 4.

In the definitions below, for a given signature Σ, we denote by T_Σ the set of terms on Σ, by $T_\Sigma(X)$ the terms with sorted variables from the set X, and by $T_{\Sigma,s}$ and $T_\Sigma(X)_s$ the terms of sort s from the respective sets. Finally, $\Sigma|_s = \{f : s \to s' \mid \text{ for some } s' \in S\}$ denotes the set of totally defined unary operators in Σ with domain $s \in S$.

Definition 2 (*atomic egalitarian rewrite system*)
An *atomic egalitarian rewrite system* is a tuple $\mathcal{R} = (S, \leq, \Sigma, E, R)$, where:

- $(S, \leq)$ is a poset of sorts. We assume $\texttt{State}, \texttt{Trans}, \texttt{Stage} \in S$ with $\texttt{State} \leq \texttt{Stage}$ and $\texttt{Trans} \leq \texttt{Stage}$. The terms of sort `Stage` are called *stages*.
- Σ is a signature of operators (and constants) $f : \omega \to s$ for some $\omega \in S^*$ and $s \in S$. We assume there is a constant $\texttt{init} \in \Sigma$ of sort `Stage`.
- E is a set of left-to-right oriented equations

$$t = t' \ \ \mathsf{if} \ \ C$$

  where $t, t' \in T_\Sigma(X)_s$ for some $s \in S$ and the condition C (which may be absent) is a conjunction $\bigwedge_i u_i = u_i'$ of equational conditions, for $u_i, u_i' \in T_\Sigma(X)_{s_i}$ for some $s_i \in S$.
- The set R contains egalitarian rules, that is, rules of the form

$$u \dashv\!\![t]\!\!\rightarrowtail u' \ \ \mathsf{if} \ \ C$$

  where $u, u' \in T_\Sigma(X)_{\texttt{State}}$, $t \in T_\Sigma(X)_{\texttt{Trans}}$ and C (which may be absent) is as above.

We also refer as *signature* to the triple $(S, \leq, \Sigma)$. A *property* is any element of $\Sigma|_{\texttt{Stage}}$, that is, any unary operator in Σ totally defined on `Stage` terms.

The main point in which we depart from the standard definitions of rewrite system (often called rather *rewrite theory*) (Meseguer 1992) is that our rules are egalitarian, by which we mean that they include an explicit transition term. Properties are also a nonstandard ingredient. As a passing note, we have shown (Martín 2021a, Section 6.2.4) that requiring properties to be totally defined, as we do, is not a meaningful restriction.

In Maude, and in our examples in this paper, equations are introduced by the keywords **`eq`** or **`ceq`**, and rules by **`rl`** or **`crl`**; in each case the c form is used when conditions are present. The signature is represented by sentences with keywords `sort`, `subsort`, and `op` or `ops`, though we often omit such sentences in the examples in this paper.

Some of the definitions and results that follow are very similar for transition structures and for rewrite systems. In particular, the synchronization mechanism is the same for one and the other. To minimize repetition, we deal with both of them jointly as much as possible. We refer to them in abstract as *systems* and with the letter $\mathcal{S}$.

We compose atomic systems and structures to create complex ones. In all this paper we consider each system to be its own namespace, so that the sets of properties, sorts and operators from different systems are disjoint.

Definition 3 (*suitable synchronization criteria*)
Given a set of atomic structures or systems, one for each $n = 1, \ldots, N$, either all of them in $\mathsf{atEgTrStr}$ or all of them in $\mathsf{atEgRwSys}$, each with set of properties P_n, a set of *synchronization criteria* for them is a set $Y \subseteq \bigcup_n P_n \times \bigcup_n P_n$.

We say that a set Y of synchronization criteria is *suitable* if it satisfies the following conditions. For transition structures, we require that, if $(p, p') \in Y \cap (P_m \times P_n)$, for some $m, n \in \{1, \ldots, N\}$, with $p : Q_m \cup T_m \to C$ and $p' : Q_n \cup T_n \to C'$, then the elements in C and C' can be compared for equality. Correspondingly, for rewrite systems $\mathcal{R}_n = (S_n, \leq_n, \Sigma_n, E_n, R_n)$, we require that, if $(p, p') \in Y \cap (P_m \times P_n)$, with $p : \mathtt{Stage}_m \to s$ and $p' : \mathtt{Stage}_n \to s'$, then there exists a sort s_0, common to $\mathcal{R}_m$ and $\mathcal{R}_n$, with $s_m \leq_m s_0$ and $s_n \leq_n s_0$, and an equational theory $\mathcal{E}_0$ of s_0, included as subtheory in both $\mathcal{R}_m$ and $\mathcal{R}_n$, in which the values of p and p' can be checked for equality.

To be precise, we should require that $\mathcal{E}_0$ be *embedded* (rather than *included*) by means of injective maps into the equational theories of $\mathcal{R}_m$ and $\mathcal{R}_n$. In that way, the namespaces of different systems are kept disjoint. While it is technically imprecise, we use the shorthand of saying that $\mathcal{E}_0$ is the *common equational theory* of s_0.

Definition 4 (*synchronous composition*)
The *synchronous composition* of $\mathcal{S}_n$ for $n = 1, \ldots, N$, either all of them in $\mathsf{atEgTrStr}$ or all of them in $\mathsf{atEgRwSys}$, with respect to the suitable synchronization criteria Y is denoted by $\|_Y\{\mathcal{S}_n \mid n = 1, \ldots, N\}$, or usually just $\|_Y \mathcal{S}_n$. From now on, whenever we write $\|_Y \mathcal{S}_n$, we are assuming Y is suitable. When only two components are involved, we usually write $\mathcal{S}_1 \|_Y \mathcal{S}_2$.

Definition 5 (*egalitarian structures and systems*)
We define the classes of *egalitarian transition structures*, denoted by $\mathsf{EgTrStr}$, and, respectively, of *egalitarian rewrite systems*, denoted by $\mathsf{EgRwSys}$, as the smallest ones that contain $\mathsf{atEgTrStr}$ or, respectively, $\mathsf{atEgRwSys}$, and are closed with respect to the synchronous composition operation described above.

We need to consider a notion of equivalence: the one given by the different ways of composing the same components. For example, $(\mathcal{S}_1 \|_Y \mathcal{S}_2) \|_{Y'} \mathcal{S}_3$ is equivalent to $\|_{Y \cup Y'}\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$.

Definition 6 (*equivalent structures and systems*)
The *set of atomic components* of an egalitarian transition structure or rewrite system is:

- $\mathrm{atoms}(\mathcal{S}) = \{\mathcal{S}\}$ if $\mathcal{S}$ is atomic,
- $\mathrm{atoms}(\|_Y \mathcal{S}_n) = \bigcup_n \mathrm{atoms}(\mathcal{S}_n)$.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

The *total set of criteria* of an egalitarian transition structure or rewrite system is:

- $\text{criteria}(\mathcal{S}) = \emptyset \quad$ if $\mathcal{S}$ is atomic,
- $\text{criteria}(\|_Y \mathcal{S}_n) = \widetilde{Y} \cup \bigcup_n \text{criteria}(\mathcal{S}_n)$,

where $\widetilde{Y} = \{\{p, q\} \mid (p, q) \in Y\}$ (so that (p, q) and (q, p) represent the same criterion).

Two egalitarian structures or systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be *equivalent* iff $\text{atoms}(\mathcal{S}_1) = \text{atoms}(\mathcal{S}_2)$ and $\text{criteria}(\mathcal{S}_1) = \text{criteria}(\mathcal{S}_2)$.

Proposition 1 (*equivalence to composition of atoms*)
Every egalitarian transition structure or rewrite system is equivalent to one of the form $\|_Y \mathcal{S}_n$ where each $\mathcal{S}_n$ is atomic.

Namely, $\mathcal{S} = \|_Y \mathcal{S}_n$ is equivalent to $\|_{Y'} \text{atoms}(\mathcal{S})$, where $Y' = \{(p, q) \mid \{p, q\} \in \text{criteria}(\mathcal{S})\}$.

In our previous work (Martín *et al*. 2020; Martín 2021a) we showed that equivalent systems represent the same behavior, as given by paths and satisfaction of temporal formulas. This allows us to group the atomic components in the most suitable way for a modular design. Thus, in the example in Section 2.3, we first composed `RIVER || PREVIOUS` to obtain `RIVER-W-PREV`, which was then used in the composition `RIVER-W-PREV || AVOID1 || AVOID2`.

In short, the compound $\|_Y \mathcal{S}_n$ is a set of atomic components linked by synchronization criteria. The behavior it models is that in which each component evolves according to its internal specification, with the added restriction that all synchronization criteria have to be satisfied at all times.

Definition 7 (*signature and properties of a compound*)
Let $\mathcal{R}_n = (S_n, \leq_n, \Sigma_n, E_n, R_n) \in \mathsf{atEgRwSys}$ for $n = 1, \ldots, N$. Let Y be a set of suitable synchronization criteria. The set of properties for $\mathcal{R}_n$ has already being defined as $\Sigma|_{\mathtt{Stage}_n}$. The set of properties for $\|_Y \mathcal{R}_n$ is defined to be $\biguplus_n P_n$. Also, the signature for $\|_Y \mathcal{R}_n$ is defined to be $(\bigcup_n S_n, \bigcup_n \leq_n, \bigcup_n \Sigma_n)$.

This definition, as was the case for Definition 3, is not technically precise, because we require at the same time that the namespaces be disjoint and that they share the common equational theories. A precise definition would involve pushouts. We avoid it and allow the slight informality of saying that each rewrite system is its own namespace, disjoint from the rest except for those common equational theories.

Definition 8 (*semantics in the atomic case*)
Given $\mathcal{R} = (S, \leq, \Sigma, E, R) \in \mathsf{atEgRwSys}$, we define $\text{sem}(\mathcal{R}) = (Q, T, \rightarrow, P, g_0) \in \mathsf{atEgTrStr}$ by:

- $Q = T_{\Sigma/E,\mathtt{State}}$ (that is, E-equational classes of `State` terms);
- $T = T_{\Sigma/E,\mathtt{Trans}}$ (that is, E-equational classes of `Trans` terms);
- $\rightarrow$ is the half-rewrite relation $\rightarrow^{\text{eg}}_{\mathcal{R}}$ induced by R (Martín *et al*. 2020, Definition 6);
- $P = \Sigma|_{\mathtt{Stage}}$;
- $g_0 = [\mathtt{init}]_E$ (that is, the E-equational class of `init`).

The half-rewrite relation $\rightarrow$ takes the system from a state to a transition, or vice versa, in contrast to the usual state-to-state rewrites. Roughly speaking, a rewrite rule $u \;\text{-}[t]\text{→}\; u'$

produces half rewrites from instances of u to instances of t, and from there to instances of u'.

Definition 9 (*semantics for the general egalitarian case*)
Given $\|_Y \mathcal{R}_n \in \mathsf{EgRwSys}$, we define its semantics componentwise:

$$\mathrm{sem}(\|_Y \mathcal{R}_n) = \|_Y\, \mathrm{sem}(\mathcal{R}_n) \in \mathsf{EgTrStr}.$$

A path semantics for the composition of egalitarian structures is given in Section 4.

3.2 Plain structures and systems

In addition to egalitarian structures and systems, we use standard ones which we call *plain* to avoid confusion with the egalitarian ones. An important feature of plain structures and systems is that they only have states, and not (explicit) transitions, and this allows their composition to be defined as a tuple construction. We see plain structures and systems as modeling the global behavior of composed systems, while we use egalitarian structures and systems to model local and distributed systems. The correspondence between them is given by the *split* operation defined later.

Definition 10 (*plain transition structure*)
A *plain transition structure* is a tuple $\mathcal{T} = (Q, \rightarrow, P, q_0)$, where:

- Q is the set of states;
- $\rightarrow \subseteq Q \times Q$ is the adjacency relation;
- P is the set of properties, each one a total function p from Q to some codomain C_p;
- $q_0 \in Q$ is the initial state.

The class of all plain transition structures is denoted by TrStr.

Definition 11 (*plain rewrite system*)
A *plain rewrite system* is a tuple $(S, \leq, \Sigma, E, R)$, where:

- $(S, \leq)$ is a poset of sorts which contains the element `State`.
- Σ is a signature of operators which includes the constant `init` of sort `State`.
- E is a set of equations as in Definition 2.
- R is a set of rules of the form $t \rightarrow t'$ if C, where $t, t' \in T_\Sigma(X)_s$ for some $s \in S$, and C (which may be absent) is as in Definition 2.

We also refer as *signature* to the triple $(S, \leq, \Sigma)$. We call *properties* to the elements of $\Sigma|_{\mathtt{State}}$. The class of all plain rewrite systems is denoted by RwSys.

Definition 12 (*composition for plain transition structures*)
Given plain transition structures $\mathcal{T}_n = (Q_n, \rightarrow_n, P_n, q_{n0}) \in \mathsf{TrStr}$, for $n = 1, \ldots, N$, their *synchronous composition* with respect to the synchronization criteria $Y \subseteq \bigcup_n P_n \times \bigcup_n P_n$, is denoted by $\|_Y \{\mathcal{T}_n \mid n = 1, \ldots, N\}$, or usually just $\|_Y \mathcal{T}_n$, and is defined to be $\mathcal{T} = (Q, \rightarrow, P, q_0) \in \mathsf{TrStr}$, where:

- $Q = \{\langle q_1, \ldots, q_N \rangle \in \prod_n Q_n \mid$ for each $(p, p') \in Y$ with $p \in P_m$ and $p' \in P_{m'}$ we have $p(q_m) = p'(q_{m'})\}$;

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

- for $\langle q_1, \ldots, q_N \rangle, \langle q'_1, \ldots, q'_N \rangle \in Q$, we have $\langle q_1, \ldots, q_N \rangle \to \langle q'_1, \ldots, q'_N \rangle$ iff for each n either $q_n \to_n q'_n$ or $q_n = q'_n$, with at least one occurrence of the former;
- $P = \bigcup_n P_n$ and, if p is a property originally defined in the component $\mathcal{T}_m$, then it is defined in $\mathcal{T}$ by $p(\langle q_1, \ldots, q_N \rangle) = p(q_m)$;
- $q_0 = \langle q_{10}, \ldots, q_{N0} \rangle$, assumed to be in Q (that is, to satisfy the criteria in Y).

It is an important detail that the composition of plain transition structures can be evaluated to a single, monolithic structure of the same type, while the composition of egalitarian structures is just a set of interacting but independent components.

The composition of plain rewrite systems is defined next by a tuple-like construction; in particular, rewrite rules are produced in this way. For this to work, we need the components involved to be topmost. A plain rewrite system is said to be *topmost* if its rules can only be applied on whole `State` terms, not on its subterms – see more explanations in our previous work (Martín *et al.* 2020).

Definition 13 (*composition for plain rewrite systems*)
Given plain rewrite systems $\mathcal{R}_n = (S_n, \leq_n, \Sigma_n, E_n, M_n, R_n) \in \mathsf{RwSys}$ for $n = 1, \ldots, N$, all of them topmost, their *synchronous composition* with respect to synchronization criteria Y is denoted by $\|_Y\{\mathcal{R}_n \mid n = 1, \ldots, N\}$, or usually just $\|_Y \mathcal{R}_n$, and is defined to be a new plain rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, R) \in \mathsf{RwSys}$. The elements of $\mathcal{R}$ are defined as the disjoint union of the respective elements of each $\mathcal{R}_n$ (that is, $S = \biguplus_n S_n$, and so on), except for the following:

- There is in S a new sort `State` and a constructor $\langle \_ \rangle : \texttt{State}_1 \times \cdots \times \texttt{State}_N \nrightarrow \texttt{State}$ ($\texttt{State}_n$ denotes the sort `State` from component $\mathcal{R}_n$).
- There is a constant `init` of sort `State` and an equation $\texttt{init} = \langle \texttt{init}_1, \ldots, \texttt{init}_N \rangle$ ($\texttt{init}_n$ denotes the constant `init` from component $\mathcal{R}_n$).
- For each $(p, p') \in Y$, suitability of Y (Definition 3) implies the existence of a common sort s and a common equational theory for it. These are common and, thus, included only once in the result of the composition.
- For each property p defined in the component $\mathcal{R}_m$, there is in Σ a declaration of a property with the same name and in E an equation $p(\langle q_1, \ldots, q_N \rangle) = p(q_m)$.
- We assume an equational theory of the Booleans is included, and we add the declaration of a new operator $\texttt{isValidState} : \Pi_{i=1}^{N} \texttt{State}_i \to \texttt{Boolean}$, defined by this equation:

$$\texttt{isValidState}(\langle q_1, \ldots, q_N \rangle) = \bigwedge_{(p,p') \in Y} p(\langle q_1, \ldots, q_N \rangle) = p'(\langle q_1, \ldots, q_N \rangle).$$

- The rewrite rules from the components are dropped, and the set of rules R for the composition is built in the following way. For each nonempty set $M \subseteq \{1, \ldots, N\}$, and for each set of rules $q_m \to q'_m$ if C_m, one from each R_m for $m \in M$, and setting $q'_m = q_m$ for $m \notin M$, there is the following rule in R:

$$\begin{aligned} \langle q_1, \ldots, q_N \rangle \to \langle q'_1, \ldots, q'_N \rangle \ \text{ if } \bigwedge_{m \in M} C_m \ &\wedge \ \texttt{isValidState}(\langle q_1, \ldots, q_N \rangle) \\ &\wedge \ \texttt{isValidState}(\langle q'_1, \ldots, q'_N \rangle). \end{aligned}$$

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

With these rules, only `State` terms for which synchronization criteria are satisfied are reachable from `init`.

Equations from different components are mixed together, according to this definition, but there are no conflicts, because each component is its own namespace. The resulting plain rewrite system happens to be topmost as well, so it can be used as a component in turn.

Definition 14 (*semantics for plain rewrite systems*)
Given $\mathcal{R} = (S, \leq, \Sigma, E, R) \in \mathsf{RwSys}$, we define its semantics $\mathrm{sem}(\mathcal{R}) = (Q, \to, P, q_0) \in \mathsf{TrStr}$ by:

- $Q = T_{\Sigma/E,\mathtt{State}}$;
- $\to$ is the rewrite relation $\to_{\mathcal{R}}$ induced by $\mathcal{R}$;
- $P = \Sigma|_{\mathtt{State}}$;
- $q_0 = [\mathtt{init}]_E$.

Concepts of equivalence can be defined for plain transition structures and for plain rewrite systems (Martín *et al.* 2020), corresponding to the equivalence in the egalitarian setting from Definition 6, to formalize the idea that the ordering and grouping of components in a composition are immaterial. For example, $(\mathcal{S}_1\|_{Y_1}\mathcal{S}_2)\|_{Y_2}\mathcal{S}_3$ is equivalent to $(\mathcal{S}_3\|_{Y_3}\mathcal{S}_1)\|_{Y_4}\mathcal{S}_2$ if $Y_1 \cup Y_2 = Y_3 \cup Y_4$, for $\mathcal{S}_n$ either plain rewrite systems or plain transition structures. (Remember that, whenever we write such composition expressions, we are assuming the synchronization criteria to be suitable.) Although we do not repeat those definitions here, when we write expressions like $\langle q_1, \ldots, q_N\rangle \in \|_Y \mathcal{S}_n$ we are assuming that some ordering and grouping of the components have been arbitrarily fixed. And when we say that two systems are equal, we rather mean they are equivalent in that sense. This is the case in the following proposition.

Proposition 2 (*semantics and composition commute*)
For plain rewrite systems $\mathcal{R}_n$, each of them topmost, and for suitable synchronization criteria Y, we have that $\mathrm{sem}(\|_Y \mathcal{R}_n) = \|_Y\, \mathrm{sem}(\mathcal{R}_n)$.

3.3 The split

Plain systems have the advantage that they are standard rewrite systems and existing theoretical and practical tools can be used on them. For that reason, it is sometimes useful to transform an egalitarian system into an equivalent plain one. This is what the operation that we call *split* does. The result of the split represents in a single system the joint evolution of the three components.

Definition 15 (*the split*)
Given $\mathcal{T} = (Q, T, \to, P, g_0) \in \mathsf{atEgTrStr}$, its *split* is $\mathrm{split}(\mathcal{T}) = (Q \cup T, \to, P, g_0) \in \mathsf{TrStr}$. That is, stages are transformed into states.

Given $\mathcal{R} = (S, \leq, \Sigma, E, R) \in \mathsf{atEgRwSys}$, its split is $\mathrm{split}(\mathcal{R}) = (S', \leq, \Sigma, E, R') \in \mathsf{RwSys}$, 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 $s \mathrel{-\!\![t]\!\!\to} s'$ if C in R to produce the two rules $s \to t$ if C and $t \to s'$ if C in R'.

For a nonatomic system $\|_Y \mathcal{S}_n$ in $\mathsf{EgTrStr}$ (resp., in $\mathsf{EgRwSys}$), its split is recursively defined by $\mathrm{split}(\|_Y \mathcal{S}_n) = \|_Y\, \mathrm{split}(\mathcal{S}_n)$, a system in TrStr (resp., in RwSys).

The composition of plain systems can always be evaluated to a single one, so the result of a split is always a single plain transition structure or rewrite system.

Proposition 3 (*semantics and split commute*)
For $\mathcal{R} \in \mathsf{EgRwSys}$ all whose atomic components are topmost, we have that $\mathrm{sem}(\mathrm{split}(\mathcal{R})) = \mathrm{split}(\mathrm{sem}(\mathcal{R}))$.

Definition 16 (*compatible stages*)
Given $\mathcal{T} = (Q, T, \to, P, g_0)$ and $\mathcal{T}' = (Q', T', \to', P', g'_0)$, the stages $g \in Q \cup T$ and $g' \in Q' \cup T'$ are said to be *compatible* (with respect to Y) iff all criteria in Y are satisfied when evaluated at them, that is, $p(g) = p'(g')$ for each $(p, p') \in Y \cap (P \times P')$. More in general, given $\mathcal{T}_n = (Q_n, T_n, \to_n, P_n, g_{n0})$ for $n = 1, \ldots, N$, we say that the stages $\{g_n\}_n$, with $g_n \in Q_n \cup T_n$, are *compatible* when they are so pairwise according to the above.

The intuitive meaning is that compatible stages can be visited simultaneously, each within its own component system. In the example of the chained buffers, Section 2.1, the states `sending` in `BUFFER1` and `receiving` in `BUFFER2` are compatible with respect to the synchronization criterion `BUFFER1$isSending = BUFFER2$isReceiving`, because `isSending` evaluates to true at `sending` and `isReceiving` evaluates also to true at `receiving`. There is a trivial bijection between compatible stages and states in the split which justifies the view that states in $\mathrm{split}(\mathcal{T})$ represent global states for the compound $\mathcal{T}$.

Proposition 4 (*distributed and global states*)
There is a bijection between the set of compatible stages in $\|_Y \mathcal{T}_n$ and the set of states in $\mathrm{split}(\|_Y \mathcal{T}_n)$.

4 Distributed and global paths

In preparation for the definition of satisfaction in following sections, we need an operational, or step, semantics for all our transition structures. They are given by paths (for atomic and plain structures) and sets of compatible paths (for compounds). They are defined in this section.

Definition 17 (*path and maximal path*)
A *path* in $\mathcal{T} = (Q, T, \to, P, g_0) \in \mathsf{atEgTrStr}$ is a finite or infinite sequence of adjacent stages $\overline{g} = g_0 \to g_1 \to \ldots$ starting at the structure's initial stage. We call such a path *maximal* if it is either infinite or it is finite and its final stage has no stages adjacent to it.

Similarly, a path in $\mathcal{T} = (Q, \to, P, q_0) \in \mathsf{TrStr}$ is a sequence of adjacent states $\overline{q} = q_0 \to q_1 \to \ldots$. We call such a path *maximal* if it is either infinite or it is finite and its final state has no states adjacent to it.

Compatibility of paths is defined by means of a relation between indices which shows a way in which all paths can be traversed together, interleaving some steps, making other simultaneous, and keeping compatibility of stages at all times. The intuitive meaning of the following definition is that, if $\langle i_1, \ldots, i_N \rangle$ is in the relation X, then the stages $g_{1i_1}, \ldots, g_{Ni_N}$ are visited at the same time, each in its structure. Thus, each relation X describes a possible execution of the composed system.

Definition 18 (*compatible paths*)
Let $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$. For each n, let $\overline{g_n} = g_{n0} \to g_{n1} \to \ldots$ be a finite or infinite path in $\mathcal{T}_n$. The paths $\{\overline{g_n} \mid n = 1, \ldots, N\}$ are said to be *compatible* (with respect to a given Y) iff there exists a relation between indices $X \subseteq \mathbb{N}^{\{1,\ldots,N\}}$ satisfying the following conditions:

1. $\langle 0, \ldots, 0 \rangle \in X$.
2. If $\langle i_1, \ldots, i_N \rangle \in X$ and g_{ni_n} is not the last stage in $\overline{g_n}$ for at least one $n \in \{1, \ldots, N\}$, then for exactly one nonempty $M \subseteq \{1, \ldots, N\}$ we have that $\langle i'_1, \ldots, i'_N \rangle \in X$, where $i'_n = i_n + 1$ if $n \in M$, and $i'_n = i_n$ otherwise.
3. All tuples in X can be obtained by means of the two previous conditions.
4. $\langle i_1, \ldots, i_N \rangle \in X$ implies the compatibility (with respect to Y) of the stages g_{ni_n} ($n = 1, \ldots, N$).
5. For each stage g_{ni} in each path $\overline{g_n}$, the index i appears as the nth component of some tuple in X.

Further, a set of paths is said to be *maximally compatible* if no path or subset of paths in it can be extended with new stages in the respective components while maintaining compatibility.

The conditions, specially Condition 2, make it possible to arrange all the tuples in X in a linear sequence, which is shown in Proposition 5 to correspond to a path in the split system. Thus, paths in the split can be seen as global paths.

Condition 5 entails that the paths are all traversed together *in their entirety*. This, however, does not mean each path is maximal in its component: a partial path can be a member of a compatible set, as long as X shows how to traverse it to its last (though maybe not terminal) stage.

For example, consider the paths for the chained buffers from Section 2.1

- in `BUFFER1`: `idle` → `receiving` → `gotToken` → `sending` → ⋯;
- in `BUFFER2`: the same as in `BUFFER1`;
- in `BUFFER3`: the single-stage path `idle`.

A set X showing how to traverse these three paths would include, among others, the following triples:

- $\langle 0, 0, 0 \rangle$, representing the three paths starting at `idle`;
- $\langle 1, 0, 0 \rangle$ and $\langle 2, 0, 0 \rangle$, representing only the first path advancing one step and two steps;
- $\langle 3, 1, 0 \rangle$, representing the first and second paths advancing to the respective stages `sending` and `receiving`, which are compatible.

As a side note, compatibility of paths cannot be defined pairwise, as we did for compatibility of stages in Definition 16. It need not be the case that three paths can be traversed simultaneously keeping compatibility of stages, even if any two of them can.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Consider $\mathrm{split}(\|_Y \mathcal{T}_n)$. Its states are tuples of components' stages. Thus, for each atomic component $\mathcal{T}_n$ of $\mathcal{T}$, a projection map π_n can be defined from the states of $\mathrm{split}(\|_Y \mathcal{T}_n)$ to the stages of $\mathcal{T}_n$. This projection can be extended to paths. However, our definitions allow for a component to advance while others stay in the same stage, so, in general, a pure projection would produce repeated stages (stuttering) that we want to remove.

Definition 19 (*projection*)
Let $q = \langle g_1, \ldots, g_N \rangle$ be a state of $\mathrm{split}(\|_Y \mathcal{T}_n)$ and $\overline{q} = q_0 \to q_1 \to \ldots$ be a path in $\mathrm{split}(\|_Y \mathcal{T}_n)$. For each $n = 1, \ldots, N$:

- we define $\pi_n(q) = g_n$;
- we define $\pi_n(\overline{q})$ as the result of removing stuttering (that is, simplifying consecutive repetitions) from $\pi_n(q_0) \to \pi_n(q_1) \to \ldots$

Proposition 5 (*distributed and global paths*)
There is a bijection between sets of compatible paths in $\{\mathcal{T}_n\}_n$ (with respect to Y) and paths in $\mathrm{split}(\|_Y \mathcal{T}_n)$. Also, there is a bijection between sets of maximally compatible paths in $\{\mathcal{T}_n\}_n$ (with respect to Y) and maximal paths in $\mathrm{split}(\|_Y \mathcal{T}_n)$.

The paths in a compatible set are not required to be maximal. Indeed, any projection of $\overline{q}$ may fail to be maximal in its component, even if $\overline{q}$ is in the split.

Proof
We prove first that, for $\overline{q}$ a path in $\mathrm{split}(\|_Y \mathcal{T}_n)$, the projections $\pi_n(\overline{q})$, for $n = 1, \ldots, N$, are compatible paths, with the relation X (required by Definition 18) being induced by $\overline{q}$ itself. Because the projections π_n remove stuttering, we need to be careful with the resulting indices. We introduce a function s which, when applied to $\pi_n(q_i)$ (the nth component of the ith state appearing in $\overline{q}$), returns the index of that stage in the path $\pi_n(\overline{q})$ (that is, after removing stuttering). Then, $X = \{\langle s(\pi_1(q_i)), \ldots, s(\pi_N(q_i))\rangle \mid q_i \text{ in } \overline{q}\}$ meets the conditions in Definition 18.

Next, we prove that, given paths $\overline{g_n}$ in $\mathcal{T}_n$, for $n = 1, \ldots, N$, which are compatible, there is a unique path $\overline{q}$ in $\mathrm{split}(\|_Y \mathcal{T}_n)$ such that $\pi_n(\overline{q}) = \overline{g_n}$. Let X be the relation whose existence is given by compatibility of paths in Definition 18. Let the initial state of $\overline{q}$ be $q_0 = \langle g_{10}, \ldots, g_{N0} \rangle$. Then, inductively, for each state q_k already in the path, let the next state q_{k+1} be the tuple whose existence is required by Condition 2 in Definition 18. Condition 5 ensures that the projections of this $\overline{q}$ produce the complete $\overline{g_n}$'s.

The *maximal* part now follows: to any hypothetical extension for a set of compatible paths would correspond an extension to the corresponding path in the split, and vice versa. □

We are not saying too much here: there is an almost trivial correspondence between tuples of paths and paths of tuples. But there are useful consequences. The split provides global, monolithic concepts of states and paths. The equivalence between those concepts and the distributed ones validates our definitions and allows us to work using the most suitable view in each case. Also, as discussed in Section 6.1, it allows us to reason about models of distributed systems, or even execute them, by performing the split and using existing techniques and tools for the corresponding global, monolithic result.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

4.1 A short diversion on locality

Even though the definition of compatibility involves all paths at once, and thus all components at once, there is room to see locality somewhat concealed in it. We have already mentioned that the contrast between local and global, or, equivalently, between a distributed view of complex systems and a monolithic one is a motivation for our work, so a short diversion is in order.

For an example, consider a system composed of a sender and a receiver, which synchronize on a Boolean property, very much in the same way as the chained buffers in Section 2.1 did:

```
sync SENDER || RECEIVER
   on SENDER$isSending = RECEIVER$isReceiving .
```

While the `SENDER` is not ready to send, its property `isSending` keeps being false, the same as `RECEIVER$isReceiving`. Meanwhile, `SENDER` can evolve in whatever way fits to its function. The system `SENDER` may even be a composed system on its own, and then its components can interact among them as they need to, with no concern about `RECEIVER`. Of course, the same is true of `RECEIVER`. This is the sense in which locality is included in our definitions. This view is more difficult to appreciate when only considering global, monolithic definitions of composition. Let us be more precise.

Proposition 6 (*compatibility and locality*)
Suppose given the egalitarian transition structure $\mathcal{T} = \|_Y \{\mathcal{T}_n \mid n = 1, \ldots, N\}$, which we rather prefer to view grouped as $\mathcal{T} = \mathcal{T}' \|_{Y_3} \mathcal{T}''$ with $\mathcal{T}' = \|_{Y_1} \{\mathcal{T}_n \mid n = 1, \ldots, N'\}$ and $\mathcal{T}'' = \|_{Y_2} \{\mathcal{T}_n \mid n = N' + 1, \ldots, N\}$ (therefore, $Y = Y_1 \uplus Y_2 \uplus Y_3$). Suppose further that the stages $\{g_n\}_n$, $n = 1, \ldots, N$, are compatible and that, for each n, there is a g'_n such that either $g_n \to_n g'_n$ or $g_n = g'_n$ (that is, either $\mathcal{T}_n$ advances one step or stays where it was). We have that the stages $\{g'_n \mid n = 1, \ldots, N\}$ are compatible if (but not only if) the three following conditions hold:

- the stages in the set $\{g'_n \mid n = 1, \ldots, N'\}$ are compatible respect to Y_1;
- the stages in the set $\{g'_n \mid n = N' + 1, \ldots, N\}$ are compatible respect to Y_2; and
- for each p used in Y_3, if $p \in P_m$, we have $p(g_m) = p(g'_m)$.

Proof
Let (p, q) be a criterion in $Y \cap (P_i \times P_j)$, that is, property p is defined in $\mathcal{T}_i$ and property q in $\mathcal{T}_j$. We need to show that $p(g'_i) = q(g'_j)$ if the three conditions hold.

If $i, j \in \{1, \ldots, N'\}$, it means that $(p, q) \in Y_1$, and then $p(g_i) = q(g_j)$ because of the first item in the proposition statement. Similarly if $i, j \in \{N' + 1, \ldots, N\}$. Finally, if $i \in \{1, \ldots, N'\}$ and $j \in \{N' + 1, \ldots, N\}$, or vice versa, then $(p, q) \in Y_3$, so that, because of the third item in the statement and the compatibility of $\{g_n\}_n$, we have $p(g'_i) = p(g_i) = q(g_j) = q(g'_j)$. □

5 Linear temporal logic

The temporal logic we use in this work is LTL (Clarke *et al.* 1999, Ch. 3) with two deviations from the standard that we discuss below. LTL is appropriate for compositional verification because its formulas are implicitly universally quantified over execution paths.

Thus, when the possible executions of a system are restricted by its interaction with the environment, the remaining ones still satisfy whatever LTL formulas were satisfied in isolation. ACTL* (Clarke *et al.* 1999, Ch. 3) is a superset of LTL that shares this universality property, but we restrict to LTL in this paper.

The first difference between our logic, which we call $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$, and standard LTL is that we avoid the use of the *next* temporal operator, usually represented by $\bigcirc$ (or, alternatively, $\mathbf{N}$ or $\mathbf{X}$). The reason for avoiding $\bigcirc$ is that its reference (the next stage) is not preserved by composition, nor by refinement. If we want to be ready for them, we should treat time as if it were dense: between the present and any *next* stage, a new stage may show up. Also, the semantics for the $\bigcirc$ operator is not clear when we have to evaluate it at both states and transitions. The resulting logic is still quite common in the literature. If we are allowed to bring in some experts to support us:

> [...] increasing the expressiveness of our temporal logic with a *next* operator would destroy the entire logical foundation for its use in hierarchical methods. (Lamport 1983)

> This definition is appropriate for reasoning about asynchronous processes since there is no notion of *next system state* in such cases. (Clarke *et al.* 1989)

The downside of quitting the *next* operator is that, well, sometimes it is useful. In particular, in our examples, we have found often the need to specify that a formula holds at each state from the current one but excluding the current one, which in LTL would be written as $\bigcirc \Box \varphi$. However, we have also found that the *next state of interest* can often be characterized by particular changes in the values of propositions (or, rather, properties). For example, for a proposition p and a temporal formula φ, the expression $p \wedge (p \mathbf{U} (\neg p \wedge \varphi))$ can be interpreted as saying that a change in the value of p identifies the next state of interest, at which point we require φ to hold.

The second difference between $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ and standard LTL is that, instead of atomic propositions, we use in our formulas *properties* and terms involving them – that is what Π is for. We usually denote by Π the set of property symbols to build formulas on, and by P the set of actual properties defined in a transition structure. We decided that properties are the interfaces of systems, and that they are all that is to be observed and known from the external world. It makes sense to use them in formulas. For instance, $\Diamond(p = 5)$ and $(p_1 + p_2 < p_3) \mathbf{U} (p_4 = \mathtt{true})$ are valid temporal formulas for us, interpretable on structures in which the respective properties, p, p_i, are defined. Using properties instead of propositions does not increase the expressive power of our formulas, because any Boolean expression involving properties can be turned into an atomic proposition (see Propositions 10 and 11), but properties fit better in our setting.

When we get to semantics below, we will need a means to evaluate expressions involving properties. For now, from a merely syntactic point of view, we need a signature on which such expressions are built. Remember from Definitions 2 and 11 that a signature in rewriting logic is a triple $(S, \leq, \Sigma)$. To such a signature we add Π, a set of S-sorted symbols to represent properties. Then, similarly to the notations $T_\Sigma(X)$ and $T_\Sigma(X)_s$ for terms with variables from X, we use the notations $T_\Sigma(\Pi)$ and $T_\Sigma(\Pi)_s$ for terms which can include sorted symbols from Π. Thus, viewing such symbols as new constants, $T_\Sigma(\Pi) = T_{\Sigma \cup \Pi}$ and $T_\Sigma(\Pi)_s = T_{\Sigma \cup \Pi, s}$.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Definition 20 (temporal formula)
Let $\Sigma = (S, \leq, \Sigma)$ be a signature. Let Π be a set of S-sorted symbols disjoint from Σ, and let $T_\Sigma(\Pi)$ and $T_\Sigma(\Pi)_s$ be as described above. A formula in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ is defined by:

- $t = u$ is an atomic formula for terms $t, u \in T_\Sigma(\Pi)_s$ for some sort $s \in S$;
- if φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, and $\varphi \mathbf{U} \psi$.

We define $\wedge$, $\rightarrow$, $\leftrightarrow$, $\diamond$, $\square$, $\mathbf{W}$, and $\mathbf{R}$ as the usual abbreviations.

In the particular case in which $t \in T_\Sigma(\Pi)_{\texttt{Bool}}$ (and assuming the sort `Bool` includes the value `true`), it is often convenient to allow the mere t as a shortcut for the formula $t = \texttt{true}$, so that we can write $p_1 + p_2 < p_3$ instead of $(p_1 + p_2 < p_3) = \texttt{true}$.

6 Basic satisfaction relations

The satisfaction relations studied in this section consider systems as closed entities, with no environment, no interaction with other systems. Sections 8 and, specially, Section 9 deal with open, interacting systems.

We need two elements to jointly provide a basis to evaluate the satisfaction of $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formulas. One is a Σ-algebra on which terms in T_Σ are evaluated. The other element we need is a transition structure on which temporal formulas make sense; for this, we use egalitarian transition structures and plain ones. Transition structures also provide interpretations for the property symbols in Π. Thus, we are dealing with satisfaction relations of the form $\mathcal{T}, \mathcal{A} \models \varphi$, where $\mathcal{T}$ is a transition structure (which, in our definition, includes its initial state or stage), $\mathcal{A}$ is a Σ-algebra, and φ is a temporal formula in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$.

The algebra $\mathcal{A}$ is a Σ-algebra in the usual sense that it is implicitly equipped with an interpretation map for all the elements in Σ. We denote the interpretations of $s \in S$ and $f \in \Sigma$ in $\mathcal{A}$, respectively, as $s_\mathcal{A}$ and $f_\mathcal{A}$. In the same way, a transition structure $\mathcal{T}$ with set of properties P is a Π-transition structure, in the sense that it is implicitly equipped with an interpretation map that assigns to each element in Π an element in P. We denote the interpretation of $p \in \Pi$ in $\mathcal{T}$ as $p_\mathcal{T}$. Also, this interpretation has to be *sort-preserving*, that is, if $p \in \Pi$ has been given sort s, then the codomain of $p_\mathcal{T}$ has to be $s_\mathcal{A}$. Often, Σ and Π are clear from context, and we omit them and say just *algebra* and *transition structure*.

Satisfaction is formalized below but, intuitively, evaluating $\mathcal{T}, \mathcal{A} \models \square(p_1 + p_2 < p_3)$ for an atomic $\mathcal{T}$ entails: (i) finding the properties in $\mathcal{T}$ that are the interpretations of p_1, p_2, and p_3; (ii) finding the values of those properties at each of $\mathcal{T}$'s stages; (iii) using $\mathcal{A}$ to evaluate $(p_1(g) + p_2(g) < p_3(g)) = \texttt{true}$ for each stage g; and (iv) using the results from the previous step and the adjacency relation in $\mathcal{T}$ to decide whether $\square((p_1 + p_2 < p_3) = \texttt{true})$ holds.

The previous discussion is equally valid for the three types of transition structures: plain or egalitarian, atomic or otherwise. The definition of satisfaction of formulas is very similar in all cases, so we present the three definitions at once, in part to avoid repetitions, but also to highlight the similarities.

Remember from Definition 4 that the set of properties of a composed transition structure is the disjoint union of the properties of its components. So, if each $\mathcal{T}_n$ is a

Π_n-transition structure, then $\mathcal{T} = \|_Y \mathcal{T}_n$ is a $(\biguplus_n \Pi_n)$-transition structure. The interpretation of p in $\mathcal{T}$, $p_{\mathcal{T}}$, is also $p_{\mathcal{T}_n}$ for some n.

Definition 21 (*evaluation of terms*)
Consider a Π-transition structure $\mathcal{T}$, with set of properties P, implicitly equipped with a sort-preserving interpretation for properties $p \mapsto p_{\mathcal{T}}$.

- For $\mathcal{T} \in \mathsf{atEgTrStr}$, consider the mapping $v : p \mapsto p_{\mathcal{T}}(g_0)$, that is, the evaluation of the property p at $\mathcal{T}$'s initial stage.
- Respectively, for $\mathcal{T} \in \mathsf{EgTrStr}$, consider the mapping $v : p \mapsto p_{\mathcal{T}}(g_{m0})$ if $p \in P_m$, that is, the evaluation of the property p at the initial stage of the component it is defined on.
- Respectively, for $\mathcal{T} \in \mathsf{TrStr}$, consider the mapping $v : p \mapsto p_{\mathcal{T}}(q_0)$, that is, the evaluation of the property p at $\mathcal{T}$'s initial state.

The mapping v can be extended to $T_\Sigma(\Pi)$ homomorphically in the standard way: $\overline{v}(p) = v(p)$, and $\overline{v}(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}(\overline{v}(t_1), \ldots, \overline{v}(t_n))$. We denote as $t_{\mathcal{T},\mathcal{A}}$ the image of t under $\overline{v}$, that is, the evaluation of the term t in $\mathcal{T}$ and $\mathcal{A}$.

The type of $v(p)$ is what we called C_p in Definition 1, so it is dependent on p.

Syntactically speaking, the role of Π in $T_\Sigma(\Pi)$ is analogous to the role of a set of variables in $T_\Sigma(X)$. In this sense, the valuation v for properties is analogous to the classical valuation maps that assign to each variable in X an element in the algebra.

There is a technical point regarding interpretations and the split that we need to take care of: both $\mathcal{T}$ and $\mathrm{split}(\mathcal{T})$ are Π-transition structures, both with the same set of properties, say P, so that they are both equipped with an interpretation from Π to P, respectively, $p \mapsto p_{\mathcal{T}} \in P$ and $p \mapsto p_{\mathrm{split}(\mathcal{T})} \in P$. In principle, the interpretations need not be the same, but that is the natural and convenient way to proceed.

Definition 22 (*the split, revisited*)
Given $\mathcal{T} \in \mathsf{EgTrStr}$, considered as a Π-transition structure and equipped with an interpretation $p \mapsto p_{\mathcal{T}}$, we define $\mathrm{split}(\mathcal{T}) \in \mathsf{TrStr}$ as in Definition 15 and equipped with the interpretation $p \mapsto p_{\mathrm{split}(\mathcal{T})} = p_{\mathcal{T}}$.

Definition 23 (π^i *and* $\mathcal{T}(g)$)

- For a path $\pi = \overline{g}$ in $\mathcal{T} \in \mathsf{atEgTrStr}$, we denote as π^i the result of removing from π its first i stages. Also, $\mathcal{T}(g)$ is the result of replacing in $\mathcal{T}$ its initial stage by g, that is, $\mathcal{T}(g) = (Q, T, \rightarrow, P, g)$.
- The definition is a little more involved for $\mathsf{EgTrStr}$. Let $\pi = \{\overline{g_n}\}_n$ be a set of compatible paths, and let X be the relation from Definition 18 which shows how to traverse them all simultaneously. As observed there, the tuples in X can be ordered linearly. Let $\langle r_1, \ldots, r_N\rangle$ be the ith tuple in that linear sequence. We denote as π^i the result of removing from each component path g_n its first r_n stages. Also, $\mathcal{T}(\{g_{nk_n}\}_n)$ is the result of replacing in each $\mathcal{T}_n$ its initial stage by g_{nk_n}.
- Finally, for a path $\pi = \overline{q}$ in $\mathcal{T} \in \mathsf{TrStr}$, we denote as π^i the result of removing from π its first i states. Also, $\mathcal{T}(q)$ is the result of replacing in $\mathcal{T}$ its initial state by q, that is, $\mathcal{T}(q) = (Q, \rightarrow, P, q)$.

Now, we can define the satisfaction relation for each of our three classes of transition structures.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Definition 24 (*satisfaction for transition structures*)
Let $\Sigma = (S, \leq, \Sigma)$ be a signature, $\mathcal{A}$ be a Σ-algebra, and Π be a set of S-sorted symbols, disjoint from Σ. Also,

1. let $\mathcal{T} = (Q, T, \rightarrow, P, g_0) \in \mathsf{atEgTrStr}$ be an atomic Π-structure;
2. respectively, let $\mathcal{T} = \|_Y \mathcal{T}_n \in \mathsf{EgTrStr}$ be a nonatomic Π-structure;
3. respectively, let $\mathcal{T} = (Q, \rightarrow, P, q_0) \in \mathsf{TrStr}$ be a plain Π-structure.

Finally, let $t, u \in T_\Sigma(\Pi)_s$ for some $s \in S$, and let φ, ψ be formulas in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$. The satisfaction relation $\mathcal{T}, \mathcal{A} \models \varphi$ is defined by:

- $\mathcal{T}, \mathcal{A} \models t = u$ iff $t_{\mathcal{T},\mathcal{A}} = u_{\mathcal{T},\mathcal{A}}$;
- otherwise, $\mathcal{T}, \mathcal{A} \models \varphi$ iff
  1. for each maximal path π in $\mathcal{T}$,
  2. respectively, for each maximally compatible set of paths π in $\mathcal{T}$,
  3. respectively, for each maximal path π in $\mathcal{T}$,

  we have $\mathcal{T}, \mathcal{A}, \pi \models \varphi$.

Satisfaction of a formula by a path is defined by:

- $\mathcal{T}, \mathcal{A}, \pi \models t = u$ iff $\mathcal{T}, \mathcal{A} \models t = u$;
- $\mathcal{T}, \mathcal{A}, \pi \models \neg\varphi$ iff not $\mathcal{T}, \mathcal{A}, \pi \models \varphi$;
- $\mathcal{T}, \mathcal{A}, \pi \models \varphi \vee \psi$ iff $\mathcal{T}, \mathcal{A}, \pi \models \varphi$ or $\mathcal{T}, \mathcal{A}, \pi \models \psi$;
- $\mathcal{T}, \mathcal{A}, \pi \models \varphi\, \mathbf{U}\, \psi$ iff there is some $i \geq 0$ such that $\mathcal{T}(\pi_i), \mathcal{A}, \pi^i \models \psi$, and for all $j < i$, we have $\mathcal{T}(\pi_j), \mathcal{A}, \pi^j \models \varphi$.

These definitions are not only formally similar, but also equivalent in a sense made precise in the two propositions that follow. This is again an instance of the equivalence of the distributed and the monolithic views achieved through the split.

Proposition 7 (*split and terms*)
For any egalitarian Π-transition structure $\mathcal{T} \in \mathsf{EgTrStr}$, atomic or otherwise, and Σ-algebra $\mathcal{A}$, we have that $t_{\mathcal{T},\mathcal{A}} = t_{\mathrm{split}(\mathcal{T}),\mathcal{A}}$ for every term $t \in T_\Sigma(\Pi)$.

Proof
The role of the structure, $\mathcal{T}$ or $\mathrm{split}(\mathcal{T})$, is providing values for properties. By Definition 22, the properties of $\mathcal{T}$ and those of $\mathrm{split}(\mathcal{T})$ are the same, and the interpretations are also the same. □

Proposition 8 (*split and satisfaction for transition structures*)
For any egalitarian Π-transition structure $\mathcal{T} \in \mathsf{EgTrStr}$, atomic or otherwise, and Σ-algebra $\mathcal{A}$, we have that $\mathcal{T}, \mathcal{A} \models \varphi$ iff $\mathrm{split}(\mathcal{T}), \mathcal{A} \models \varphi$ for every formula φ in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$.

Proof
We proceed by structural induction on the shape of the formula. First:

$$\begin{aligned}
\mathcal{T}, \mathcal{A} \models t = u &\iff t_{\mathcal{T},\mathcal{A}} = u_{\mathcal{T},\mathcal{A}} \\
&\iff t_{\mathrm{split}(\mathcal{T}),\mathcal{A}} = u_{\mathrm{split}(\mathcal{T}),\mathcal{A}} \\
&\iff \mathrm{split}(\mathcal{T}), \mathcal{A} \models t = u.
\end{aligned}$$

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

The second equivalence is because of Proposition 7; the other two are by the definition of satisfaction.

For the inductive case, satisfaction is defined in terms of paths. We need to use the bijection between compatible sets of paths in (the atomic components of) $\mathcal{T}$ and paths in its split, and between their maximal versions, from Proposition 5. We did not give a name to that bijection in the proposition, but it will be useful to have one now. For consistency, we denote by $\operatorname{split}(\pi)$ the path in $\operatorname{split}(\|_Y \mathcal{T}_n)$ that corresponds to the set of paths π. (There is nothing being actually split here in the literal sense of the word, so we take it just as a convenient name.)

Then, we want to prove these equivalences:

$$\begin{aligned}\mathcal{T}, \mathcal{A} \models \varphi &\iff \text{for each } \pi \text{ max. compat. set of paths in } \mathcal{T}, \text{ we have } \mathcal{T}, \mathcal{A}, \pi \models \varphi \\ &\iff \text{for each } \pi \text{ max. path in } \operatorname{split}(\mathcal{T}), \text{ we have } \operatorname{split}(\mathcal{T}), \mathcal{A}, \pi \models \varphi \\ &\iff \operatorname{split}(\mathcal{T}), \mathcal{A} \models \varphi.\end{aligned}$$

The middle equivalence is the one that still needs a proof. More concretely, we are going to prove something a little stronger: for each π which is a maximally compatible set of paths in (the atomic components of) $\mathcal{T}$ we have $\mathcal{T}, \mathcal{A}, \pi \models \varphi$ iff $\operatorname{split}(\mathcal{T}), \mathcal{A}, \operatorname{split}(\pi) \models \varphi$. Because each path in $\operatorname{split}(\mathcal{T})$ is the split of a set of compatible paths in $\mathcal{T}$, the result follows.

We proceed again by induction on the structure of the formula. The case $t = u$ is now dealt with easily, as are negation and disjunction. Thus, what remains to be proved is: for each π which is a maximally compatible set of paths in (the atomic components of) $\mathcal{T}$, we have

$$\mathcal{T}, \mathcal{A}, \pi \models \varphi \mathbf{U} \psi \quad \text{iff} \quad \operatorname{split}(\mathcal{T}), \mathcal{A}, \operatorname{split}(\pi) \models \varphi \mathbf{U} \psi.$$

This chain of equivalences is quite trivial except maybe for the third one:

$$\begin{aligned}\mathcal{T}, \mathcal{A}, \pi \models \varphi \mathbf{U} \psi &\iff \exists i \geq 0 \text{ such that } \mathcal{T}(\pi_i), \mathcal{A}, \pi^i \models \psi \\ &\qquad \text{and } \forall j < i,\ \mathcal{T}(\pi_j), \mathcal{A}, \pi^j \models \varphi \\ &\iff \exists i \geq 0 \text{ such that } \operatorname{split}(\mathcal{T}(\pi_i)), \mathcal{A}, \operatorname{split}(\pi^i) \models \psi \\ &\qquad \text{and } \forall j < i,\ \operatorname{split}(\mathcal{T}(\pi_j)), \mathcal{A}, \operatorname{split}(\pi^j) \models \varphi \\ &\iff \exists i \geq 0 \text{ such that } \operatorname{split}(\mathcal{T})(\operatorname{split}(\pi)_i), \mathcal{A}, \operatorname{split}(\pi)^i \models \psi \\ &\qquad \text{and } \forall j < i,\ \operatorname{split}(\mathcal{T})(\operatorname{split}(\pi)_j), \mathcal{A}, \operatorname{split}(\pi)^j \models \varphi \\ &\iff \operatorname{split}(\mathcal{T}), \mathcal{A}, \operatorname{split}(\pi) \models \varphi \mathbf{U} \psi.\end{aligned}$$

For the third equivalence to hold, we need $\operatorname{split}(\mathcal{T}(\pi_i)) = \operatorname{split}(\mathcal{T})(\operatorname{split}(\pi)_i)$, and $\operatorname{split}(\pi^i) = \operatorname{split}(\pi)^i$. Both are easy to justify, and will not be proved here. □

Definition 25 (*satisfaction for rewrite systems*)
Let $\mathcal{R}$ be an egalitarian rewrite system, atomic or otherwise, or a plain one. Let $\Sigma = (S, \leq, \Sigma)$ be its signature and P be its set of properties. Let Π be a set of S-sorted symbols and assume there is an interpretation from Π to P. Note that $\operatorname{sem}(\mathcal{R})$ is a Π-transition structure. Also, let $\mathcal{A}(\mathcal{R})$ be the initial algebra for the equational theory in $\mathcal{R}$ (defined as the union of the equational theories of the components, if $\mathcal{R}$ is not atomic). Note that $\mathcal{A}(\mathcal{R})$ is a Σ-algebra. Finally, let φ be an $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formula. We define $\mathcal{R} \models \varphi$ by $\operatorname{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}) \models \varphi$.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Proposition 9 (*split and satisfaction for rewrite systems*)
In the conditions of the previous definition, let $\mathcal{R}$ be an egalitarian rewrite system, and let φ be a formula in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$. We have $\mathcal{R} \models \varphi$ iff $\mathrm{split}(\mathcal{R}) \models \varphi$.

Proof

$$\begin{aligned}
\mathcal{R} \models \varphi &\iff \mathrm{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}) \models \varphi \\
&\iff \mathrm{split}(\mathrm{sem}(\mathcal{R})), \mathcal{A}(\mathcal{R}) \models \varphi \\
&\iff \mathrm{sem}(\mathrm{split}(\mathcal{R})), \mathcal{A}(\mathcal{R}) \models \varphi \\
&\iff \mathrm{sem}(\mathrm{split}(\mathcal{R})), \mathcal{A}(\mathrm{split}(\mathcal{R})) \models \varphi \\
&\iff \mathrm{split}(\mathcal{R}) \models \varphi.
\end{aligned}$$

The third equivalence is because of Proposition 3. The fourth is because $\mathcal{A}(\mathcal{R}) = \mathcal{A}(\mathrm{split}(\mathcal{R}))$, given that the equational theories from $\mathcal{R}$ are copied as such into $\mathrm{split}(\mathcal{R})$. □

Sometimes, we say that a $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formula is a formula *in the language of* $\mathcal{T}$ or *in the language of* $\mathcal{R}$, meaning that Σ and Π are the signature and property symbols associated with the transition structure $\mathcal{T}$ or the rewrite system $\mathcal{R}$, but we do not care to make Σ and Π explicit.

6.1 Back to the standards

Plain transition structures are very much like standard Kripke structures; also Boolean properties and atomic propositions are equivalent. So, in the particular case in which all properties in a plain transition structure are Boolean and all atomic formulas have the shape $p = \texttt{true}$, our definitions agree with the standard ones for Kripke structures and LTL. Even out of this particular case, everything expressible in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ using properties is also expressible in LTL with Boolean propositions. And it may be worth doing so, because it would allow the use of existing tools on our nonstandard specifications. We make it formal in this section.

Definition 26 (*translation into LTL*)
We define $[\varphi]$ for any formula $\varphi \in \mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ inductively on the structure of φ.

- From each atomic formula $t = u$ in $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$, we create an atomic proposition, which we denote as $[t = u]$.
- For each nonatomic $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formula φ, the LTL formula $[\varphi]$ is the result of replacing each atomic subformula of φ by its corresponding atomic proposition. That is:
  - $[\neg\xi] = \neg[\xi]$;
  - $[\xi \vee \xi'] = [\xi] \vee [\xi']$;
  - $[\xi \mathbf{\,U\,} \xi'] = [\xi] \mathbf{\,U\,} [\xi']$.

Definition 27 (*standardization of structures*)
Consider given Σ, Π, and $\mathcal{A}$ as usual, and an $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formula φ. Let $\mathcal{T} = (Q, \rightarrow, P, q_0) \in \mathsf{TrStr}$. We generate a Kripke structure $\mathcal{K} = \mathcal{K}(\mathcal{T}, \mathcal{A}, \varphi)$ as $\mathcal{K} = (Q, \rightarrow, \mathrm{AP}, \mathcal{L}, q_0)$, where:

- Q, $\to$, and q_0 are in $\mathcal{K}$ the same as in $\mathcal{T}$;
- $\mathrm{AP} = \{[\xi] \mid \xi$ is an atomic subformula of $\varphi\}$;
- $\mathcal{L}(q) = \{[\xi] \in \mathrm{AP} \mid \mathcal{T}(q), \mathcal{A} \models \xi\}$, for $q \in Q$ and $\mathcal{T}(q)$ being the transition structure that results by replacing $\mathcal{T}$'s initial stage by q.

Proposition 10 (standardization of satisfaction)
Let Σ, Π, $\mathcal{A}$, φ, $\mathcal{T}$, $[\varphi]$, and $\mathcal{K}$ as in the previous paragraphs. We have $\mathcal{T}, \mathcal{A} \models \varphi$ iff $\mathcal{K} \models [\varphi]$.

The satisfaction relation for Kripke structures and LTL formulas is the standard one (Clarke *et al.* 1999).

Proof
The proof is an easy induction on the structure of formulas. We also need to prove the equivalence for paths: $\mathcal{T}, \mathcal{A}, \overline{q} \models \varphi$ iff $\mathcal{K}, \overline{q} \models [\varphi]$. We illustrate it with just two cases:

$$\begin{aligned} \mathcal{T}, \mathcal{A} \models t = u &\iff [t = u] \in \mathcal{L}(q^0) \\ &\iff \mathcal{K} \models [t = u]. \end{aligned}$$

$$\begin{aligned} \mathcal{T}, \mathcal{A} \models \varphi \,\mathbf{U}\, \psi &\iff \text{for each maximal path } \overline{q} \text{ in } \mathcal{T}, \text{ we have } \mathcal{T}, \mathcal{A}, \overline{q} \models \varphi \,\mathbf{U}\, \psi \\ &\iff \text{for each maximal path } \overline{q} \text{ in } \mathcal{T}, \exists i \geq 0 \text{ such that} \\ &\qquad \mathcal{T}(q_i), \mathcal{A}, \overline{q}^i \models \psi \text{ and } \forall j < i, \mathcal{T}(q_j), \mathcal{A}, \overline{q}^j \models \varphi \\ &\iff \text{for each maximal path } \overline{q} \text{ in } \mathcal{K}, \exists i \geq 0 \text{ such that} \\ &\qquad \mathcal{K}(q_i), \overline{q}^i \models [\psi] \text{ and } \forall j < i, \mathcal{K}(q_j), \overline{q}^j \models [\varphi] \\ &\iff \text{for each maximal path } \overline{q} \text{ in } \mathcal{K}, \text{ we have } \mathcal{K}, \overline{q} \models [\varphi] \,\mathbf{U}\, [\psi] \\ &\iff \mathcal{K} \models [\varphi] \,\mathbf{U}\, [\psi] \\ &\iff \mathcal{K} \models [\varphi \,\mathbf{U}\, \psi]. \end{aligned}$$

We are using here the fact that, for $\mathcal{K} = \mathcal{K}(\mathcal{T}, \mathcal{A}, \varphi)$, the Kripke structure $\mathcal{K}(\pi_i)$, that is, $\mathcal{K}$ with its initial state replaced by π_i, can be obtained as $\mathcal{K}(\mathcal{T}(\pi_i), \mathcal{A}, \varphi)$, which is easy to prove. □

An immediate consequence is that, for $\mathcal{T} \in \mathsf{EgTrStr}$, we have $\mathcal{T}, \mathcal{A} \models \varphi$ iff $\mathrm{split}(\mathcal{T}), \mathcal{A} \models \varphi$ iff $\mathcal{K} \models [\varphi]$. This allows verifying the satisfaction of formulas in egalitarian structures by using standard tools.

It is worth noting that this procedure does not work componentwise. Suppose, for an example, that for the structure $\mathcal{T}_1$ we are interested in the formula $\Box(p_1 \leq 5)$ for some numerical property p_1. In the same way, in the structure $\mathcal{T}_2$ we have the formula $\Box(p_2 \geq 5)$ for a numerical property p_2, which is to be synchronized with p_1. After performing the standardization procedure above on both structures, we get two Boolean propositions $[p_1 \leq 5]$ and $[p_2 \geq 5]$ which are unrelated and the relation between p_1 and p_2 cannot be preserved. The implicit message is that using properties instead of Boolean propositions does not increase the expressive power of our formulas, but does increase the possibilities for synchronization.

Plain rewrite systems are close relatives of standard ones. As we have just done for transition structures, we take the final step into the standard setting.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Definition 28 (*standardization of rewrite systems*)
From a plain rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, R) \in \mathsf{RwSys}$ and an $\text{LTL}_{\otimes}(\Sigma, \Pi)$ formula φ (for some set of property symbols Π), we define AP and $[\varphi]$ as in Definitions 26 and 27. We generate a standard rewrite system $\mathcal{R}(\varphi) = (S^+, \leq, \Sigma^+, E^+, R)$ in the following way. The new set of sorts S^+ is defined to be S plus new sorts `Bool` and `Prop` (for atomic propositions). To obtain Σ^+ we add to Σ, for each $[\xi] \in \text{AP}$, the declaration of a Boolean proposition, which we also denote by $[\xi]$, and also an operator $\models$: `State` × `Prop` → `Bool`. Finally, we obtain E^+ by adding to E equations to define $\models$ for each new proposition in AP:

- $\big(g \models [t(\overline{p}) = u(\overline{p})]\big) = \big(t[\overline{p}(g)/\overline{p}] = u[\overline{p}(g)/\overline{p}]\big)$, where $\overline{p}$ is the sequence of properties in t (and in u) and $t[\overline{p}(g)/\overline{p}]$ is the result of replacing in t each p by its evaluation at g, that is, by $p(g)$.

Thus, for example, $\big(g \models [p = 3]\big) = \big(p(g) = 3)\big)$, assuming $p = 3$ is an atomic formula in φ for a numeric property p.

Proposition 11 (*standardization of satisfaction*)
Let Σ, Π, $\mathcal{A}$, φ, $\mathcal{R}$, $[\varphi]$, and $\mathcal{R}(\varphi)$ as in Definition 28. We have $\mathcal{R} \models \varphi$ iff $\mathcal{R}(\varphi) \models [\varphi]$.

Proof
We have that $\mathcal{R} \models \varphi$ iff $\text{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}) \models \varphi$ iff $\mathcal{K}(\text{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}), \varphi) \models [\varphi]$ (the first equivalence by Definition 25; the second by Proposition 10). We assert that the Kripke structure $\mathcal{K}(\text{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}), \varphi)$ is isomorphic to the standard semantics for rewrite systems associated to $\mathcal{R}(\varphi)$. From that, the proposition follows. The assertion is not difficult to check. For example, the terms of sort `State` are the same in $\mathcal{R}$ and in $\mathcal{R}(\varphi)$, and they produce state nodes in the transition structure $\text{sem}(\mathcal{R})$, which correspond to state nodes in the Kripke structure $\mathcal{K}(\text{sem}(\mathcal{R}), \mathcal{A}(\mathcal{R}), \varphi)$. Similarly, the adjacency relation and the values of propositions can be checked to correspond. □

7 On fairness and deadlocks

When a system $\mathcal{S}$ interacts with an environment $\mathcal{E}$, its repertoire of execution paths is restricted to a subset of the ones that are possible when $\mathcal{S}$ is run in isolation. For an LTL formula φ in the language of $\mathcal{S}$, the statement $\mathcal{S} \models \varphi$ means that all maximal paths in $\mathcal{S}$ satisfy φ. In particular, all maximal paths in $\mathcal{S}$ that remain after the environment restriction still satisfy φ. And because φ only speaks about $\mathcal{S}$ (and not about $\mathcal{E}$), we may be willing to assert that $\mathcal{S} \models \varphi$ implies $\mathcal{S} \|_Y \mathcal{E} \models \varphi$ for any environment $\mathcal{E}$ and criteria Y. Except this does not hold when the interaction with the environment prevents $\mathcal{S}$ from executing long enough to satisfy φ. This may be the case when paths in $\mathcal{S}$ that are not maximal become maximal in $\mathcal{S} \|_Y \mathcal{E}$, which can happen because of lack of fairness between components or because of emerging deadlocks.

Some models of interaction avoid these issues by establishing that only fair executions are part of the semantics (Pnueli 1985; Grumberg and Long 1994), or that all interactions consist of message passing and the receiver is at all times ready to receive (Lynch and Tuttle 1989), thus preventing emerging deadlocks. We are taking a more permissive approach, which both requires and allows a discussion of the details. This section

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

introduces such a discussion. We consider only transition structures, but the results are equally valid for rewrite systems by means of their semantics.

First, consider deadlocks. More precisely, emerging deadlocks, that is, the ones which result from the failure of the component systems to agree on a next action to perform.

Definition 29 (*deadlock*)
Let $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$. A set of maximally compatible paths $\{\overline{g_n}\}_n$, each $\overline{g_i}$ being a finite or infinite path in $\mathcal{T}_i$, is said to be *deadlocked* iff no component path $\overline{g_i}$ is maximal in its component $\mathcal{T}_i$.

According to this definition, there is no deadlock as long as some component system keeps running, even if only one does. Moreover, if one path in the set is finite in its system and reaches its final state in the compatible set, there is no deadlock, even though the composed system may come to a halt. This is our working definition, certainly not the only possible one. This capability to accommodate different definitions within the same framework is made possible by a permissive concept of composition like ours.

Deadlock can be prevented with some extra work on the part of the specifier. Being aware that the system is meant to work inside a largely unknown environment, the specifier should be able to anticipate unfriendly behaviors and be ready to deal with them. That is, the specification should include reactions to wrong environment behaviors, even if only with the aim of raising exceptions or performing error recovery. The following proposition shows a particular case in which this is achieved.

Proposition 12 (*a case of deadlock freeness*)
Let $\mathcal{T}_1, \mathcal{T}_2 \in \mathsf{atEgTrStr}$ and let $Y = \{(p, q)\}$ be the singleton set of synchronization criteria to compose $\mathcal{T}_1$ and $\mathcal{T}_2$. Suppose that in $\mathcal{T}_1$ we have that, for each pair of stages g, g' with $g \rightarrow_1 g'$, and each possible value v in the range of p, there exists a stage g_v, still in $\mathcal{T}_1$, such that $p(g_v) = v$ and $g \rightarrow_1 g_v$. Then no set of compatible paths in $\{\mathcal{T}_1, \mathcal{T}_2\}$ can be deadlocked.

Proof
According to Definition 29, we need to prove that if $\overline{g_1}$ and $\overline{g_2}$ are compatible paths, then at least one of them is maximal in its component, $\mathcal{T}_1$ or $\mathcal{T}_2$. Let us consider a particular stage g_1^1 in $\overline{g_1}$, which is not a final stage in $\mathcal{T}_1$ (if taken as an isolated system), that is, there exist g_1^2 with $g_1^1 \rightarrow_1 g_1^2$ in $\mathcal{T}_1$. We show that g_1^1 cannot be the last stage in $\overline{g_1}$.

Because of the statement of the proposition, each possible value v of p is realized in a g_v in $\mathcal{T}_1$ such that $g_1^1 \rightarrow_1 g_v$. Now, suppose the relation X from Definition 18 (the one which shows how the two paths can be traversed) pairs g_1^1 with g_2^1, and then consider the particular value $v = q(g_2^1)$. Then, the path in $\mathcal{T}_1$ that has taken us to g_1^1 can be extended to g_v while $\mathcal{T}_2$ stays at g_2^1, so that $\overline{g_1}$ can run indefinitely, or as long as $\mathcal{T}_1$ allows it to. □

The conditions in Proposition 12 can be paraphrased as one component acting as a receiver which is ready to receive any value at any time. Less demanding conditions would be enough to guarantee absence of deadlocks.

This technique may seem too convoluted, but something similar is implicitly used in some models of composition. Typically, those models divide the possible interactions of a

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

component with its environment into inputs and outputs. Inputs represent the reception of a value from the environment. The input value is controlled only by the environment, and the component is assumed to be ready to receive it, at any time, whatever it is. In the same way, the environment is assumed to be ready to receive any value the component outputs. For example, input/output automata (Lynch and Tuttle 1989), explicitly state that input events are not in control of the automata that receives them. The technique proposed in the previous paragraph is no more than an explicit implementation of this.

Consider now fairness between components. Fairness is difficult to characterize in the presence of deadlocks, so we only define it for non-deadlocked sets of paths.

Definition 30 (fairness)
Let $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$. Consider the set of maximally compatible paths $\{\overline{g_n}\}_n$, each $\overline{g_i}$ being a finite or infinite path in $\mathcal{T}_i$. Assuming there is no deadlock in $\{\overline{g_n}\}_n$, the set of compatible paths $\{\overline{g_n}\}_n$ is said to be *fair* iff each component path $\overline{g_i}$ is maximal in its transition structure $\mathcal{T}_i$.

Thus, fairness entails that, if a partial path can be extended in its component alone, then it gets eventually extended in the composition as well. This is different from intra-component fairness: for our purposes here, we do not care about fairness inside each individual component, but only in their interactions.

Our definition of synchronous composition does not require fairness, so it is possible that a component starves. An extreme case is that in which no synchronization criteria are specified, so that the different systems are just put together, but allowed to execute independently. In this case, $\mathcal{S} \models \varphi$ does not entail $\mathcal{S} \|_{\emptyset} \mathcal{E} \models \varphi$, because a possible evolution of the composed system is that $\mathcal{E}$ executes but $\mathcal{S}$ does not perform a single step.

A way in which fairness is ensured is by requiring synchronization infinitely often. For example, as in the following proposition.

Proposition 13 (a case of fairness)
Let $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$. Suppose that for each $i, j \in \{1, \ldots, N\}$, $i \neq j$, there is a pair of Boolean properties $(p, q) \in Y \cap (\mathcal{T}_i \times \mathcal{T}_j)$, such that

$$\mathcal{T}_i \models \Box \Diamond (p = \mathtt{true}) \land \Box \Diamond (p = \mathtt{false})$$

and

$$\mathcal{T}_j \models \Box \Diamond (q = \mathtt{true}) \land \Box \Diamond (q = \mathtt{false}).$$

Then any set of compatible (and not deadlocked) paths in $\|_Y \mathcal{T}_n$ is fair.

Proof
First, we note that, for any $i = 1, \ldots, N$, the fact that $\mathcal{T}_i \models \Box \Diamond (p = \mathtt{true}) \land \Box \Diamond (p = \mathtt{false})$ implies that any path in $\mathcal{T}_i$ is infinite; thus, no finite maximal paths exist. According to Definition 30, we need to prove that in any set of compatible non-deadlocked paths $\{\overline{g_n}\}_n$, each component $\overline{g_i}$ is infinite. As we are supposing they are not deadlocked, we know that at least one path is infinite. Without loss of generality, suppose $\overline{g_1}$ is infinite, let $\overline{g_i}$ be any other path, and let q be the property in $\mathcal{T}_i$ which synchronizes with p, according to the proposition's statement.

Let X be the relation from Definition 18 which shows how to traverse the compatible paths. Let us say g_1^1 and g_i^1 are compatible stages in $\mathcal{T}_1$ and $\mathcal{T}_i$, resp., appearing in $\overline{g_1}$

and $\overline{g_i}$, resp., which are paired by X. Further, suppose, again without loss of generality, that $p(g_1^1) = q(g_i^1) = \texttt{true}$. Because $\overline{g_1} \models \Box\Diamond(p = \texttt{false})$, there is a stage g_1^2 in $\overline{g_1}$ which does not satisfy p. Therefore, so that the criterion (p, q) is kept, $\overline{g_i}$ must contain a stage g_i^2 which does not satisfy q and that is accessible from g_i^1, that is, $g_i^1 \to \cdots \to g_i^2$. Thus, $\overline{g_i}$ is infinite. □

The condition in Proposition 13 is an example. It is nice in that it can be expressed as a temporal logic formula and, thus, checked by the usual means. More general and easy to meet conditions may be found which are sufficient to ensure fairness. As mentioned above, in some models of computation and composition, fairness is included from the start, that is, the path semantics of a specification includes, by definition, only fair executions, even though the specification, textually taken, would allow unfair ones. This is different from our view, in which we require the specification to be fair as given. The two views, however, are not completely disjoint. In Section 9, we consider the assume/guarantee technique and mention that temporal formulas expressing fairness requirements can be added to the *assume* part of a specification. This, in a sense, makes fair semantics a particular case of our model. Also, sometimes a system can be externally controlled to allow only fair executions in it. Maybe, even, such a control can be exerted via a synchronous composition with a suitable system. But many different concepts of fairness are possible, and it is not to be expected that all of them can be dealt with in this way.

Proposition 14 (*deadlock freeness and fairness are enough*)
Given $\|_Y \mathcal{T}_n$, with $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$, if all sets of maximally compatible paths are non-deadlocked and fair, then $\mathcal{T}_i \models \varphi$ implies $\|_Y \mathcal{T}_n \models \varphi$ for each $i \in \{1, \ldots, N\}$ and each formula φ in the language of $\mathcal{T}_i$.

Proof
The assertion $\mathcal{T}_i \models \varphi$ means that all paths in $\mathcal{T}_i$ satisfy φ. Each compatible set of paths in $\|_Y \mathcal{T}_n$ contains as ith component a path in $\mathcal{T}_i$ which, because of fairness and absence of deadlock, is guaranteed to be maximal in its component system $\mathcal{T}_i$. And because φ is expressed in the language of $\mathcal{T}_i$, its satisfaction does not depend on other component paths. Therefore, $\|_Y \mathcal{T}_n \models \varphi$. □

Besides, deadlocks and fairness become unimportant when φ is a safety formula: by definition, a safety formula is satisfied by a path iff it is satisfied by every initial segment of that path, even the empty one. Thus, the proof of the following proposition is immediate.

Proposition 15 (*safety formulas are enough*)
Given $\|_Y \mathcal{T}_n$, with $\mathcal{T}_n \in \mathsf{atEgTrStr}$ for $n = 1, \ldots, N$, if φ is a safety formula in the language of $\mathcal{T}_i$ for some $i \in \{1, \ldots, N\}$, then $\mathcal{T}_i \models \varphi$ implies $\|_Y \mathcal{T}_n \models \varphi$.

A component from which we only require to satisfy safety formulas can be seen as imposing its behavior on the compound. It acts as a controller or a strategy. This is the case for the mutual exclusion controller example from Section 2.2 (revisited in Sections 8.3 and 9.2).

The next proposition is a simple remark that a kind of converse implication always holds.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Proposition 16 (*satisfaction in any environment*)
With the usual notation, we have that if $(\mathcal{T}\|_Y\mathcal{E}), \mathcal{A} \models \varphi$ for every environment $\mathcal{E}$ and every suitable Y, then $\mathcal{T}, \mathcal{A} \models \varphi$. Likewise, if $\mathcal{R}\|_Y\mathcal{E} \models \varphi$ for every environment $\mathcal{E}$ and every suitable Y, then $\mathcal{R} \models \varphi$.

Proof
We can define an environment $\mathcal{E}_0$ that preserves all the behaviors of $\mathcal{T}$ in the following way: let $\mathcal{E}_0$ consist of a single system with a unique state, and let $Y = \emptyset$, that is, no requirements for synchronization. The behaviors of $\mathcal{T}$ in this environment are the same as the ones of $\mathcal{T}$ alone.

We are assuming $(\mathcal{T}\|_Y\mathcal{E}), \mathcal{A} \models \varphi$ for all $\mathcal{E}$ and Y, so, in particular, $(\mathcal{T}\|_\emptyset\mathcal{E}_0), \mathcal{A} \models \varphi$ and, because of the previous paragraph, also $\mathcal{T}, \mathcal{A} \models \varphi$. The proof for rewrite systems follows easily. □

Concerning deadlocks and fairness, our framework sets the responsibility in the hands of the user. This is also the case in (Glabbeek and Höfner 2019, Chapter 16), where some actions may need to be explicitly declared as *non-blocking*. And in (Klai *et al.* 2005), where an algorithm is provided to check that an interaction between Petri nets is *non-constraining*, which is a similar concept. This is another point where an actual implementation may include tools to help. We do not go deeper into this issue here.

8 Componentwise simulation and abstraction

A way to analyze a system is to find another one that in some sense behaves the same and is simpler. This is formalized with the concept of *simulation* (Clarke *et al.* 1999). A particular kind of simulation is *abstraction*, in which the simpler system is obtained by forgetting some features from the original. In rewriting logic, a well studied kind of abstraction is *equational abstraction* (Meseguer *et al.* 2008). In this section, we show that componentwise simulation and equational abstraction translate into global ones. That is, roughly speaking, if there is a simulation (resp., equational abstraction) between systems $\mathcal{S}_1$ and $\mathcal{S}_1'$, then there is also a simulation (resp. equational abstraction) between $\mathcal{S}_1\|_Y\mathcal{S}_2$ and $\mathcal{S}_1'\|_Y\mathcal{S}_2$. This is sometimes phrased as simulation and equational abstraction being congruences.

Another kind of abstraction that has been studied in relation to rewriting logic is *predicate abstraction* (Bae and Meseguer 2014). According to it, states of the original system which coincide in the values assigned to all atomic propositions are identified in the abstract system. Predicate abstraction in one component does not need to map to predicate abstraction in the composition. However, predicate abstraction induces a simulation in the abstract component, which does map to a simulation at the global level. We have not much else to say about predicate abstraction in this paper, though we use it in the example in Section 8.3.

There are several ways in which abstraction can be useful for compositional verification. First, instead of verifying $\mathcal{S}$ in the environment $\mathcal{E}$ (that is, $\mathcal{S}\|_Y\mathcal{E}$), we can verify an abstraction of $\mathcal{S}$ in the same environment. Second, if we verify $\mathcal{S}$ in $\mathcal{E}$, the result will also hold for any environment of which $\mathcal{E}$ is an abstraction. Often, we model intuitively our systems from scratch as abstractions. This is certainly the case for the example on

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

chained buffers in Section 2.1. The results which follow in this section show that, if we later need to refine our initial specification, verification may not need to be redone.

8.1 Simulation

Up to now, we have been taking care of defining each concept in both the distributed and the monolithic view. For example, we defined a compatible set of paths and showed it equivalent to a path in the split; and we then defined satisfaction of formulas based on both and, again, showed equivalence. Definition 31 just below, however, defines simulation for composed egalitarian structures as simulations for their splits. We proceed in this way from now on, because it makes definitions and results easier. Still, when we want to enforce one or the other view, distributed or monolithic, we use one or the other of the two equivalent notations, like, for example, either $\|_Y \mathcal{S}_n \models \varphi$ or $\mathrm{split}(\|_Y \mathcal{S}_n) \models \varphi$.

Definition 31 (*simulation*)
Given a set Π of property symbols and two atomic egalitarian Π-transition structures $\mathcal{T} = (Q, T, \rightarrow, P, g_0)$ and $\mathcal{T}' = (Q', T', \rightarrow', P', g_0')$, a *simulation* $\mathsf{S} : \mathcal{T} \rightarrow \mathcal{T}'$ is a relation $\mathsf{S} \subseteq (Q \cup T) \times (Q' \cup T')$ such that:

- $g_0 \; \mathsf{S} \; g_0'$;
- if $g \; \mathsf{S} \; g'$ then $p_{\mathcal{T}}(g) = p_{\mathcal{T}'}(g')$ for each $p \in \Pi$;
- if $g_1 \; \mathsf{S} \; g_1'$ and $g_1 \rightarrow g_2$ in $\mathcal{T}$, then there exists a finite path in $\mathcal{T}'$, $g_1' \rightarrow' \ldots \rightarrow' g_k'$, with $k \geq 1$, such that $g_1 \; \mathsf{S} \; g_i'$ for $i = 1, \ldots, k-1$ and $g_2 \; \mathsf{S} \; g_k'$.

If both S and S^{-1} are simulations, we say that S is a bisimulation.

The definition for plain transition structures is a straightforward adaptation of the above. Finally, a simulation between nonatomic egalitarian transition structures $\|_Y \mathcal{T}_n$ and $\|_{Y'} \mathcal{T}_n'$ is, by definition, a simulation between their splits: $\mathsf{S} : \mathrm{split}(\|_Y \mathcal{T}_n) \rightarrow \mathrm{split}(\|_{Y'} \mathcal{T}_n')$.

A (bi)simulation is with respect to the symbols in Π. When we need to make this explicit, we say it is a Π-(bi)simulation.

The third item in the definition allows, in particular, $k = 1$, so that the requirement becomes $g_1 \; \mathsf{S} \; g_1'$ and $g_2 \; \mathsf{S} \; g_1'$ – so to speak, $\mathcal{T}$ advances while $\mathcal{T}'$ waits.

The concept defined above is analogous to the ones called *stuttering (bi)simulation* and *weak (bi)simulation* in the literature. However, we decided to avoid the use of the *next* operator in our temporal logic, and also decided that only the values of properties are important, not paying attention to possible internal steps. Thus, we are always working in a way that pretty much corresponds to stuttering or weakness. So, we drop adjectives and call our concept just *(bi)simulation*.

Theorem 1 (*simulation and satisfaction*)
Consider Σ, Π, and $\mathcal{A}$ as usual, and $\mathcal{T}, \mathcal{T}' \in \mathsf{EgTrStr} \cup \mathsf{TrStr}$. If there exists a simulation $\mathsf{S} : \mathcal{T} \rightarrow \mathcal{T}'$, then for every $\mathrm{LTL}_{\otimes}(\Sigma, \Pi)$ formula φ we have that $\mathcal{T}', \mathcal{A} \models \varphi$ implies $\mathcal{T}, \mathcal{A} \models \varphi$. If S is a bisimulation, then $\mathcal{T}, \mathcal{A} \models \varphi$ iff $\mathcal{T}', \mathcal{A} \models \varphi$.

Proof
It is an easy adaptation of the proof for more traditional settings (Clarke *et al.* 1999). It proceeds by induction on the structure of φ. It relies on two lemmas that hold whenever

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

there is a simulation $\mathsf{S} : \mathcal{T} \to \mathcal{T}'$ (they are easy, and we do not prove them here): first, that $t_{\mathcal{T},\mathcal{A}} = t_{\mathcal{T}',\mathcal{A}}$ for any term $t \in T_\Sigma(\Pi)$; second, that for each path in $\mathcal{T}$ there is a (stuttering, weak) corresponding path in $\mathcal{T}'$. Let us sketch just one base case and one inductive case:

$$\begin{aligned}\mathcal{T}',\mathcal{A} \models t = u &\iff t_{\mathcal{T}',\mathcal{A}} = u_{\mathcal{T}',\mathcal{A}}\\ &\iff t_{\mathcal{T},\mathcal{A}} = u_{\mathcal{T},\mathcal{A}}\\ &\iff \mathcal{T},\mathcal{A} \models t = u.\end{aligned}$$

The second equivalence is justified by the first lemma mentioned above.

$$\begin{aligned}\mathcal{T}',\mathcal{A} \models \varphi\, \mathbf{U}\, \psi &\iff \text{for each path } \overline{g}' \text{ in } \mathcal{T}' \text{ we have } \mathcal{T}',\mathcal{A},\overline{g}' \models \varphi\, \mathbf{U}\, \psi\\ &\iff \text{for each path } \overline{g}' \text{ in } \mathcal{T}' \text{ there exists } i' \geq 0 \text{ such that}\\ &\qquad \mathcal{T}'(g'_{i'}),\mathcal{A},\overline{g}'^{i'} \models \psi \text{ and, for all } j' < i', \mathcal{T}'(g'_{j'}),\mathcal{A},\overline{g}'^{j'} \models \varphi\\ &\implies \text{for each path } \overline{g} \text{ in } \mathcal{T} \text{ there exists } i \geq 0 \text{ such that}\\ &\qquad \mathcal{T}(g_i),\mathcal{A},\overline{g}^{i} \models \psi \text{ and, for all } j < i, \mathcal{T}(g_j),\mathcal{A},\overline{g}^{j} \models \varphi\\ &\iff \text{for each path } \overline{g} \text{ in } \mathcal{T} \text{ we have } \mathcal{T},\mathcal{A},\overline{g} \models \varphi\, \mathbf{U}\, \psi\\ &\iff \mathcal{T},\mathcal{A} \models \varphi\, \mathbf{U}\, \psi.\end{aligned}$$

The " $\implies$ " step in the middle is justified by the second lemma mentioned above. □

The next theorem is our main result about simulations, stating that componentwise simulations induce global ones. It can be seen as an adaptation of (Clarke *et al.* 1999, Ch. 12).

Definition 32 (*$\sim$ for synchronization criteria*)
For $n = 1, \ldots, N$, let

$$A_n = (Q_{A_n}, T_{A_n}, \to_{A_n}, P_{A_n}, g^0_{A_n}) \quad \text{and} \quad B_n = (Q_{B_n}, T_{B_n}, \to_{B_n}, P_{B_n}, g^0_{B_n})$$

be atomic egalitarian Π_n-transition structures such that there are Π_n-simulations $\mathsf{S}_n : A_n \to B_n$. Consider the composed systems $\|_Y A_n$ and $\|_Z B_n$. We denote by $Y \sim Z$ the fact that, for $p, q \in \bigcup_n \Pi_n$, we have $(p_{A_n}, q_{A_m}) \in Y \cap (P_{A_n} \times P_{A_m})$ iff $(p_{B_n}, q_{B_m}) \in Z \cap (P_{B_n} \times P_{B_m})$.

Theorem 2 (*simulation and composition*)
Let A_n and B_n be atomic egalitarian Π_n-transition structures such that there are Π_n-simulations $\mathsf{S}_n : A_n \to B_n$ for $n = 1, \ldots, N$. (The identity is a bisimulation, so this includes the case that $A_n = B_n$ for some or all n.) Consider $A = \mathrm{split}(\|_Y A_n)$ and $B = \mathrm{split}(\|_Z B_n)$ for some Y and Z with $Y \sim Z$. Then, there is a simulation $\mathsf{S} : A \to B$ (as plain transition structures). In addition, if all S_n are bisimulations, then S can be taken to be a bisimulation as well.

Proof
The simulation S is defined by

$$\langle g_{A_1}, \ldots, g_{A_N} \rangle \;\mathsf{S}\; \langle g_{B_1}, \ldots, g_{B_N} \rangle \iff g_{A_n} \;\mathsf{S}_n\; g_{B_n} \text{ for all } n.$$

We must show that this is indeed a simulation (if each S_n is), that is, that the three items in Definition 31 hold.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

The first item in the definition, that $\langle g_{A_1 0}, \dots, g_{A_N 0}\rangle \mathrel{\mathsf{S}} \langle g_{B_1 0}, \dots, g_{B_N 0}\rangle$, follows immediately from $g_{A_n 0} \mathrel{\mathsf{S}_n} g_{B_n 0}$ holding for each n.

For the second item in Definition 31, we must prove that, for arbitrary g_{A_n} and g_{B_n}, if $\langle g_{A_1}, \dots, g_{A_N}\rangle \mathrel{\mathsf{S}} \langle g_{B_1}, \dots, g_{B_N}\rangle$, we have $p_A(\langle g_{A_1}, \dots, g_{A_N}\rangle) = p_B(\langle g_{B_1}, \dots, g_{B_N}\rangle)$. So, take a particular $p_A \in \bigcup_n P_{A_n}$. Suppose $p_A = p_{A_k} \in P_{A_k}$ and, therefore, because $Y \sim Z$, $p_B = p_{B_k} \in P_{B_k}$. Then, $p_A(\langle g_{A_1}, \dots, g_{A_N}\rangle) = p_{A_k}(g_{A_k}) = p_{B_k}(g_{B_k}) = p_B(\langle g_{B_1}, \dots, g_{B_N}\rangle)$.

For the third item in Definition 31, we consider the simpler case with only two components, that is, $N = 2$. This simplifies the presentation. The case for a general N follows the same lines.

Thus, from $\langle g_{A_1}, g_{A_2}\rangle \mathrel{\mathsf{S}} \langle g_{B_1}, g_{B_2}\rangle$ and $\langle g_{A_1}, g_{A_2}\rangle \to_A \langle g'_{A_1}, g'_{A_2}\rangle$ we must be able to produce a path in B with the needed properties. From $\langle g_{A_1}, g_{A_2}\rangle \mathrel{\mathsf{S}} \langle g_{B_1}, g_{B_2}\rangle$ we get $g_{A_1} \mathrel{\mathsf{S}_1} g_{B_1}$ and $g_{A_2} \mathrel{\mathsf{S}_2} g_{B_2}$. And from $\langle g_{A_1}, g_{A_2}\rangle \to_A \langle g'_{A_1}, g'_{A_2}\rangle$ we deduce both $(g_{A_1} \to_{A_1} g'_{A_1}$ or $g_{A_1} = g'_{A_1})$ and $(g_{A_2} \to_{A_2} g'_{A_2}$ or $g_{A_2} = g'_{A_2})$.

For A_1, if $g_{A_1} \to_{A_1} g'_{A_1}$, because S_1 is a simulation, we have that there exist a finite path $g_{B_1} = g^1_{B_1} \to_{B_1} \dots \to_{B_1} g^{i_1}_{B_1}$, $i_1 \geq 1$, such that $g_{A_1} \mathrel{\mathsf{S}_1} g^1_{B_1}, \dots, g_{A_1} \mathrel{\mathsf{S}_1} g^{i_1-1}_{B_1}$ and $g'_{A_1} \mathrel{\mathsf{S}_1} g^{i_1}_{B_1}$. If instead $g_{A_1} = g'_{A_1}$, we choose $g_{B_1} = g'_{B_1}$, which can be seen as a path of length 1. The same can be done for A_2, after which we end with a path in B_1 and another in B_2.

From these paths in B_1 and in B_2 we build now one in $B = B_1 \|_Z B_2$. The idea is to interleave in whichever way the paths $g^1_{B_1} \to^*_{B_1} g^{i_1-1}_{B_1}$ and $g^2_{B_2} \to^*_{B_2} g^{i_2-2}_{B_2}$, and then take a last joint step $\langle g^{i_1-1}_{B_1}, g^{i_2-1}_{B_2}\rangle \to_B \langle g^{i_1}_{B_1}, g^{i_2}_{B_2}\rangle$. For example: $\langle g_{B_1}, g_{B_2}\rangle = \langle g^1_{B_1}, g^1_{B_2}\rangle \to^*_B \langle g^{i_1-1}_{B_1}, g^1_{B_2}\rangle \to^*_B \langle g^{i_1-1}_{B_1}, g^{i_2-1}_{B_2}\rangle \to_B \langle g^{i_1}_{B_1}, g^{i_2}_{B_2}\rangle$

Two points remain to be proved. First, that $\langle g_{A_1}, g_{A_2}\rangle \mathrel{\mathsf{S}} g$ for all stages g in the path, except the last one, and that $\langle g'_{A_1}, g'_{A_2}\rangle \mathrel{\mathsf{S}} \langle g^{i_1}_{B_1}, g^{i_2}_{B_2}\rangle$. This is immediate, because it holds componentwise. Second, that the exhibited path is indeed a path in B, that is, that all stages in it satisfy the synchronization criteria in Z. The key here is that stages related by the simulation assign equal values to corresponding properties. For example, for the final stage, we know that $g'_{A_1} \mathrel{\mathsf{S}_i} g^{i_1}_{B_1}$ and $g'_{A_2} \mathrel{\mathsf{S}_i} g^{i_2}_{B_2}$ and, therefore, for each property p we have $p_{A_1}(g'_{A_1}) = p_{B_1}(g^{i_1}_{B_1})$ and $p_{A_2}(g'_{A_2}) = p_{B_2}(g^{i_2}_{B_2})$. But $\langle g'_{A_1}, g'_{A_2}\rangle$ is a stage in A, and, thus, satisfies all criteria in Y. Finally, because $Y \sim Z$, the criteria in Z are satisfied by $\langle g^{i_1}_{B_1}, g^{i_2}_{B_2}\rangle$. □

On the other hand, similarly behaved systems can be specified from quite different components, so it is not to be expected that any (bi)simulation $\mathsf{S} : \mathrm{split}(T_1) \to \mathrm{split}(T_2)$, for T_1, T_2 egalitarian transition structures, can be factored as a set of (bi)simulations on the components.

Given the importance of deadlocks and fairness in our setting, as discussed in Section 7, it is necessary to explore how they relate to simulation. It is not difficult to see that a mere simulation does not even preserve maximal compatibility of paths, which is needed to make sense of the definitions. The situation is different with bisimulation.

Proposition 17 (*bisimulations preserve fairness and deadlock freeness*)
Let $\mathcal{T}_n, \mathcal{T}'_n \in \mathsf{atEgTrStr}$ for $n = 1, \dots, N$, with each $\mathcal{T}_n$ and $\mathcal{T}'_n$ being a Π_n-transition structure. Let Y be any suitable set of synchronization criteria. For convenience, we say that $\|_Y \mathcal{T}_n$ (resp., $\|_Y \mathcal{T}'_n$) is deadlock-free iff no set of maximally compatible paths in it

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

is deadlocked (as defined in Definition 29). Similarly, we say that $\|_Y \mathcal{T}_n$ (resp., $\|_Y \mathcal{T}'_n$) is fair iff all non-deadlocked and maximally compatible sets of paths are fair (as defined in Definition 30). Suppose there are bisimulations S_n: $\mathcal{T}_n \to \mathcal{T}'_n$ for each n. Then, $\|_Y \mathcal{T}_n$ is deadlock-free iff $\|_Y \mathcal{T}'_n$ is. Also, $\|_Y \mathcal{T}_n$ is fair iff $\|_Y \mathcal{T}'_n$ is.

Proof
Given a path $\overline{g'_n}$ in $\mathcal{T}'_n$, the bisimulation S_n allows us to find a corresponding path in $\mathcal{T}_n$ which we denote as S_n^{-1} $(\overline{g'_n})$. It is easily justified that S_n^{-1} $(\overline{g'_n})$ is maximal in $\mathcal{T}_n$ iff $\overline{g'_n}$ is in $\mathcal{T}'_n$. Also, the set of paths $\{\overline{g'_n}\}_n$ is (maximally) compatible iff the set of paths $\{\mathsf{S}_n^{-1}$ $(\overline{g'_n})\}_n$ is. The definitions of fairness and deadlock depend only on the concepts of maximal path and of maximally compatible set of paths, hence the proposition. □

8.2 Equational abstraction

A well-known way to implement simulations is by equational abstraction in a rewrite system (Meseguer *et al.* 2008). In short, on an atomic egalitarian or plain rewrite system $\mathcal{R} = (S, \leq, \Sigma, E, R)$, we can perform equational abstraction by adding equations E' to obtain the new system $\mathcal{R}' = (S, \leq, \Sigma, E \cup E', R)$, so that states satisfying certain conditions are now equated and considered the same. The usual questions about computability apply here, that is, we must ensure that the new set of equations (oriented left to right) is ground Church-Rosser and terminating, and that the rules are still ground coherent with respect to the new set of equations. In (Martín *et al.* 2020, Section 3.5), we justified that checking for computability can be made componentwise. Therefore, checking whether a global abstraction is executable can also be done componentwise.

Proposition 18 (*equational abstraction induces bisimulation*)
Let $\mathcal{R}' = (S, \leq, \Sigma, E \cup E', R) \in \mathsf{atEgRwSys}$ (resp., RwSys) be an equational abstraction of $\mathcal{R} = (S, \leq, \Sigma, E, R)$. The relation $\{([t]_E, [t]_{E\cup E'}) \mid t \in T_{\Sigma,\mathtt{Stage}}\}$ (resp., $t \in T_{\Sigma,\mathtt{State}}$) is a bisimulation.

Proof
We have to check that the three conditions in Definition 31 hold in both directions.

- Each stage (resp., state) is trivially related to its abstraction. In particular, initial ones are, which ensures the first condition is met.
- The set E of equations includes the ones that define the values of properties. Thus, if the extended set of equations $E \cup E'$ is Church-Rosser, we infer that properties are *preserved*, that is, $t \equiv_{E\cup E'} u \implies p(t) = p(u)$, or, in words, that all stages (resp., states) that have been fused into the same abstract stage (resp., state) assign the same values to properties. This ensures the second condition is met.
- All transitions are kept through equational abstraction. Even if two stages (resp., states) t and u for which $[t]_E \to [u]_E$ get abstracted into the same, that is, $[t]_{E\cup E'} = [u]_{E\cup E'}$, we will still have $[t]_{E\cup E'} \to [u]_{E\cup E'}$. And every transition in the abstracted system derives from one in the original one. This ensures the third condition is met. □

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Theorem 3 (*equational abstraction and composition*)
Let $\mathcal{R}_n, \mathcal{R}'_n \in \mathsf{atEgRwSys}$ be such that each $\mathcal{R}'_n$ is an equational abstraction of the corresponding $\mathcal{R}_n$ for $n = 1, \ldots, N$. Consider $\|_Y \mathcal{R}_n$ and $\|_Y \mathcal{R}'_n$ for some set of synchronization criteria Y. Then, $\mathrm{split}(\|_Y \mathcal{R}'_n)$ can be obtained as an equational abstraction of $\mathrm{split}(\|_Y \mathcal{R}_n)$.

Proof
The difference between the contribution of each $\mathcal{R}_n$ to $\mathrm{split}(\|_Y \mathcal{R}_n)$ and the contribution of $\mathcal{R}'_n$ to $\mathrm{split}(\|_Y \mathcal{R}'_n)$ are some equations. So, $\mathrm{split}(\|_Y \mathcal{R}'_n)$ is $\mathrm{split}(\|_Y \mathcal{R}_n)$ plus some equations, that is, an equational abstraction. □

8.3 Example: mutual exclusion (continued)

We apply now simulation to our mutual exclusion example from Section 2.2. For a reminder, these were the instructions we used in the specification of each of the trains:

```
crl atStation N =[ comingFrom N ]=> atStation (N + 1) if N < 2 .
rl atStation 2 =[ crossing ]=> atStation 0 .
eq isCrossing @ crossing = true .
eq isCrossing @ G = false [owise] .
```

The property `isCrossing` embodies all our model cares about in each train system, and it makes sense to perform abstraction based on it, so that all stages with the same value for that property get equated. In this case, equational abstraction would result in all stages except `crossing` being equationally reduced to one of them. Equivalently, we can perform predicate abstraction to get one state for the truth of `isCrossing` and another for its falsehood, producing the following:

```
rl false =[ true ]=> false .
eq isCrossing @ B = B .
```

The state `true` represents the former `crossing`, while `false` is the abstraction for all the other states. We call the two systems with this specification `S-TRAIN1` and `S-TRAIN2`.

It is quite straightforward to see that the conditions in Definition 31 are met and these are indeed simulations. Because of Theorem 3, we can use this specification instead of the original one in composed systems and draw conclusions based on it.

Now we perform a three-way synchronous composition to build a new system that we call `SAFE-TRAINS`:

```
sync S-TRAIN1 || S-TRAIN2 || MUTEX
  on S-TRAIN1$isCrossing = MUTEX$isGranting(1)
  /\ S-TRAIN2$isCrossing = MUTEX$isGranting(2) .
```

We want to show that mutual exclusion holds for the crossings, that is:

$$\texttt{SAFE-TRAINS} \models \Box\, \neg(\texttt{S-TRAIN1\$isCrossing} \wedge \texttt{S-TRAIN2\$isCrossing}) \tag{1}$$

from which we can readily deduce the same formula holds for `TRAIN1` and `TRAIN2` and the same `MUTEX`. One way to prove (1) is to use our prototype implementation to perform the split on `SAFE-TRAINS` and then use Maude's model checker. A more compositional way is also possible, which is shown later, in Section 9.2.

9 The assume/guarantee technique

The classical satisfaction relation between a system $\mathcal{S}$ and a temporal formula φ, which we write $\mathcal{S} \models \varphi$, considers the system as if run in isolation – as a non-interacting, closed

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

system. For open systems, techniques have been devised to verify that a component satisfies a given specification in a suitable environment. Well-known among such techniques is assume/guarantee (Pnueli 1985), A/G from now on. This section is devoted to discussing this technique and its adaptation to our setting for verifying rewrite systems.

Satisfaction, according to the A/G technique, involves two formulas: one stating what can be assumed from the environment; the other stating what one particular component is ready to guarantee based on the assumption and on its own internal behavior. The notation we are using is $\mathcal{S} \models \alpha \triangleright \gamma$ (Elkader *et al.* 2018) (or $\mathcal{T}, \mathcal{A} \models \alpha \triangleright \gamma$ for transition structures, or $\mathcal{R} \models \alpha \triangleright \gamma$ for rewrite systems), where α is the assumption and γ the guarantee. Both are $\mathrm{LTL}_{\oslash}(\Sigma, \Pi)$ formulas expressed in the language of $\mathcal{S}$. Thus, α speaks about the environment by means of the properties of $\mathcal{S}$, some of which are to be synchronized with the ones of the environment.

The naïve reading of $\mathcal{S} \models \alpha \triangleright \gamma$ as "$\mathcal{S}$ guarantees the satisfaction of γ if placed in an environment that satisfies α" is misleading. It is not really necessary that the environment satisfies α – it is the interaction that matters. For example, an environment that behaves according to the CCS expression $a.P \mid b.Q$ does not ensure the execution of a in general, but for some processes, like $a.P' \mid c.Q'$, it does: the environments $a.P \mid b.Q$ and $a.P$ are equivalent for that process, they induce the same restrictions. More in general, it is not necessary that the environment satisfies α, but only that the interaction of the environment with the process does.

Moreover, the assumption α can sometimes be intuitively thought of as reflecting other convenient laws or facts, not always expected to be realized by an environment, like fairness assumptions, or the fact that time is strictly increasing in the case of timed systems (Abadi and Lamport 1995). In those cases, the assumption would only involve properties not used for synchronization, so that it restricts the component and not the environment.

A definition of A/G satisfaction based on the intuitions in the previous paragraphs may consider execution paths in their fullness, that is, they may be ultimately based on assertions like "if some full execution path satisfies α, then it also satisfies γ." This is unsuitable, however, because it allows that first a system fails to satisfy γ and only later the environment fails to satisfy α. This is probably not what we have in mind when we think about A/G. Instead, we can choose an inductive definition (Misra and Chandy 1981; Jonsson and Yih-Kuen 1996), which could be stated in this way:

$$\text{if } \mathcal{S} \|_Y \mathcal{E} \models_i \alpha, \text{ then } \mathcal{S} \models_{i+1} \gamma,$$

where $\models_i$ represents the satisfaction up to i steps away from the current state.

A posteriori, the two concepts, the full-path one and the inductive one, turn out to be equivalent, which reflects the fact that a system could only take advantage of the difference if it knew that the environment was going to fail to satisfy α in the future, which it cannot. Similar results are known in other settings (Kupferman and Vardi 2000, Theorem 5.1) (Abadi and Lamport 1995, Section 5.1). We prove it now in our own setting. It is Theorem 4 below, but we need some considerations first.

For the same reasons that we avoid the *next* temporal operator, we prefer to avoid explicit references to steps. For, if we later refine that *next* step into a sequence of them, the reference to state $i + 1$ turns out to be a moving one. The important concept here is that the partial path up to the present time is *compatible* with the satisfaction of the

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

formula, that is, that some maximal path that extends the partial one satisfies α. We make this formal. The definition of when a path is a prefix (or initial segment) of another is the usual one, denoted by the symbol $<$, with reflexive closure $\leq$.

Definition 33 (*path compatible with formula*)
Given $\mathcal{T} \in \mathsf{atEgTrStr}$, a finite path $\overline{g}$ in it, and a formula φ in its language, we say that $\overline{g}$ is *compatible* with φ, and denote it as $\overline{g} \mathrel{|\!\!\approx} \varphi$, iff there is some maximal path $\overline{g}'$ in $\mathcal{T}$ such that $\overline{g} \leq \overline{g}'$ and $\overline{g}' \models \varphi$.

Given that $\mathcal{T}$ and $\mathcal{A}$ are going to be fixed, we spare them when writing $\overline{g} \mathrel{|\!\!\approx} \varphi$. Also, sometimes we write just $\overline{g} \models \varphi$ instead of the full $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi$.

According to this definition, if $\overline{g}$ is maximal, then $\overline{g} \mathrel{|\!\!\approx} \varphi$ iff $\overline{g} \models \varphi$.

For a simple example, consider a finite path $g_0 \dots g_k$ such that at each g_i the value of a certain Boolean property p is `true`. And suppose there are two possible ways that path can be maximally extended: $g_0 \dots g_k g_{k+1} \dots$ and $g_0 \dots g_k g'_{k+1} \dots$, with all unprimed stages still assigning `true` to p, but g'_{k+1} assigning `false`. Then

$$g_0 \dots g_k g_{k+1} \dots \models \Box(p = \mathtt{true}),$$

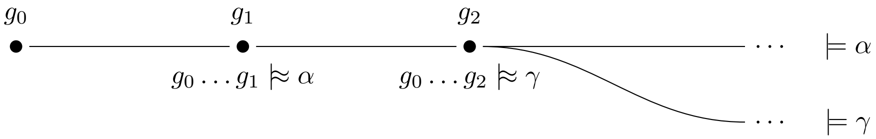
and, therefore,

$$g_0 \dots g_k \mathrel{|\!\!\approx} \Box(p = \mathtt{true}),$$

even though

$$g_0 \dots g_k g'_{k+1} \cdots \not\models \Box(p = \mathtt{true}).$$

Our definition of A/G satisfaction is somewhat involved, so we first give an intuitive explanation. Very informally, $\mathcal{T}, \mathcal{A} \models \alpha \triangleright \gamma$ is a promise from $\mathcal{T}$ of not being the first to fail to perform its duties – if we see γ as its duties and α as the environment's. Consider this diagram, showing two paths running from left to right, starting at the initial stage g_0.


g_0
g_1
g_2
$\cdots \quad \models \alpha$
$g_0 \cdots g_1 \mathrel{|\!\!\approx} \alpha$
$g_0 \cdots g_2 \mathrel{|\!\!\approx} \gamma$
$\cdots \quad \models \gamma$

At present, $\mathcal{T}$ and its environment have traversed together the path from g_0 to g_1, and have done so in a way compatible with the satisfaction of α. The component $\mathcal{T}$ cannot know what the environment is going to do in the future; it may choose to go along the upper path, that is, to keep on being compatible with α. To ensure $\mathcal{T}, \mathcal{A} \models \alpha \triangleright \gamma$, the component $\mathcal{T}$ has to keep on being compatible with γ at least a little longer than the environment, for instance, until g_2.

In principle, the path that satisfies γ needs not be the same one that satisfies α, as shown in the diagram above. However, compatibility has to be preserved up to stages g_1 arbitrarily far in the future. The result is that the two branches get zipped into one.

Definition 34 (*path allowed in a compound*)
A path $\overline{g}$ in $\mathcal{T} \in \mathsf{atEgTrStr}$ is said to be *allowed* in $\mathcal{T} \|_Y \mathcal{E}$, for $\mathcal{E} \in \mathsf{EgTrStr}$, if $\overline{g}$ is an element of some set of compatible paths in $\mathcal{T} \|_Y \mathcal{E}$.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

In a similar way, a path $\overline{q}$ in $\mathcal{T} \in \mathsf{TrStr}$ is said to be allowed in $\mathcal{T}\|_Y\mathcal{E}$, for $\mathcal{E} \in \mathsf{TrStr}$, if there is a path in $\mathcal{T}\|_Y\mathcal{E}$ whose projection on $\mathcal{T}$ is $\overline{q}$.

Definition 35 (A/G satisfaction)

- For $\mathcal{T} \in \mathsf{atEgTrStr}$, an algebra $\mathcal{A}$, and two formulas α, γ, we define $\mathcal{T}, \mathcal{A} \models \alpha \rhd \gamma$ by: for each egalitarian transition structure $\mathcal{E}$ (the environment) and suitable Y, and for each finite path $\overline{g}$ in $\mathcal{T}$ allowed in $\mathcal{T}\|_Y\mathcal{E}$ such that $\overline{g} \models\!\!\approx \alpha$, we have that:
  - either $\overline{g}$ is maximal (hence $\overline{g} \models \alpha$) and then $\overline{g} \models \gamma$;
  - or $\overline{g}$ is not maximal and, then, for each maximal $\overline{g}'$ with $\overline{g} < \overline{g}'$ and $\overline{g}' \models \alpha$, there is $\overline{g}''$ with $\overline{g} < \overline{g}'' \leq \overline{g}'$ and $\overline{g}'' \models\!\!\approx \gamma$, that is, along each maximal extension that satisfies α (of which there must be some, because $\overline{g} \models\!\!\approx \alpha$) there is an intermediate path compatible with γ.
- For $\mathcal{T} \in \mathsf{TrStr}$, the definition is, as usual, very similar to the above.
- For $\mathcal{T} \in \mathsf{EgTrStr}$, we define $\mathcal{T}, \mathcal{A} \models \varphi$ as equivalent to $\mathrm{split}(\mathcal{T}), \mathcal{A} \models \varphi$.
- For rewrite systems of any kind, the definition is based on the transition structures which are their semantics, as usual.

Possibly the simplest alternative concept of A/G satisfaction is $\mathcal{T}, \mathcal{A} \models \alpha \to \gamma$ (being $\to$ classical implication); that is, each path that satisfies the assumption also satisfies the guarantee. Though our definition of satisfaction is much more complex than that, *a posteriori* both concepts turn out to be equivalent.

Theorem 4 (equivalence of $\rhd$ *and* $\to$*)*
In the conditions of Definition 35,

$$\mathcal{T}, \mathcal{A} \models \alpha \rhd \gamma \quad \text{iff} \quad \mathcal{T}, \mathcal{A} \models \alpha \to \gamma$$

and

$$\mathcal{R} \models \alpha \rhd \gamma \quad \text{iff} \quad \mathcal{R} \models \alpha \to \gamma.$$

Proof
First, we prove the theorem for $\mathcal{T} \in \mathsf{atEgTrStr}$, that is, for an atomic $\mathcal{T}$. Assume $\mathcal{T}, \mathcal{A} \models \alpha \rhd \gamma$, and let us prove that $\mathcal{T}, \mathcal{A} \models \alpha \to \gamma$. We have to show that each path $\overline{g}$ in $\mathcal{T}$ which is maximal in $\mathcal{T}$ and satisfies α also satisfies γ. We place $\mathcal{T}$ in an arbitrary environment $\mathcal{E}$ with empty synchronization criteria: $\mathcal{T}\|_\emptyset\mathcal{E}$. Certainly, $\overline{g}$ is allowed in $\mathcal{T}\|_\emptyset\mathcal{E}$ and is maximal in $\mathcal{T}\|_\emptyset\mathcal{E}$, because it is in $\mathcal{T}$. Then, because $\mathcal{T}, \mathcal{A} \models \alpha \rhd \gamma$ and the definition of A/G satisfaction, we have $\overline{g} \models \gamma$, as we wanted.

Now, assume $\mathcal{T}, \mathcal{A} \models \alpha \to \gamma$. Fix $\mathcal{E}$ and Z. Fix also a path $\overline{g}$ in $\mathcal{T}$ which is allowed in $\mathcal{T}\|_Z\mathcal{E}$ and is compatible with α: $\overline{g} \models\!\!\approx \alpha$. If $\overline{g}$ happens to be maximal in $\mathcal{T}$, then $\overline{g} \models \alpha$ and, because of the assumption, $\overline{g} \models \gamma$. Otherwise, if $\overline{g}$ if not maximal in $\mathcal{T}$, fix a path $\overline{g}'$ which is maximal in $\mathcal{T}$, extends $\overline{g}$ and satisfies α. Because of the assumption, $\overline{g}' \models \gamma$. We can take $\overline{g}'' = \overline{g}'$, and this completes the proof for atomic transition structures.

The same proof is almost verbatim valid for plain ones. And because satisfaction for composed structures is equivalent to the one for their splits, the result also holds for TrStr. Finally, because satisfaction for rewrite systems is defined based on their semantics, the result also holds for the three types of rewrite systems. □

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

This is a most welcome result, because it means that we can use standard verification tools to perform compositional verification.

The particular case when $\alpha \equiv \texttt{true}$ is worth stating.

Corollary 1 (*true assumption*)
In the conditions of Definition 35,

$$\mathcal{T}, \mathcal{A} \models \texttt{true} \rhd \gamma \quad \text{iff} \quad \mathcal{T}, \mathcal{A} \models \gamma$$

and

$$\mathcal{R} \models \texttt{true} \rhd \gamma \quad \text{iff} \quad \mathcal{R} \models \gamma.$$

After having Theorem 4, in view of our convoluted definition for A/G satisfaction, and considering also how trivial some of our examples are (maybe specially the one on chained buffers in Section 9.1 below), it is legitimate to ask if it would not have been better to use just implication to characterize A/G to begin with. The answer, in our opinion, is *no*. The assertion $\mathcal{S} \models \alpha \to \gamma$ is about the internal behavior of $\mathcal{S}$; in contrast, $\mathcal{S} \models \alpha \rhd \gamma$ is an assertion about $\mathcal{S}$'s interaction with other systems. Their equivalence (in appropriate conditions) is a fortunate, *a posteriori* fact. Perhaps it could be likened to the equivalence between $\vdash \varphi \to \psi$ and $\varphi \vdash \psi$ in classical first-order logic.

We finish this section with the theorem which justifies the soundness of A/G.

Theorem 5 (*soundness of A/G*)
With the notational conventions used so far, let $\mathcal{R}_n$ $(n = 1, \ldots, N)$ be rewrite systems of any of the kinds discussed in this work, and let $\mathcal{R}$ be their composition with respect to the synchronization criteria Y, $\mathcal{R} = \|_Y \mathcal{R}_n$. If all the following hold:

1. for each $n = 1, \ldots, N$ and each $i = 1, \ldots, \ell_n$, we have that
   (a) $\mathcal{R}_n \models \alpha_{ni} \rhd \gamma_{ni}$,
   (b) $\mathcal{R}_n \models \alpha_{ni} \to \gamma_{ni}$ implies $\mathcal{R} \models \alpha_{ni} \to \gamma_{ni}$,
2. $\left(\bigwedge_{n=1}^{N} \bigwedge_{i=1}^{\ell_n} \alpha_{ni} \to \gamma_{ni}\right.$ and $\left.\bigwedge_{(p,p') \in Y} p = p'\right)$ imply $\alpha \to \gamma$,

then $\mathcal{R} \models \alpha \rhd \gamma$.

Proof
Because of Theorem 4, for each i, we have $\mathcal{R}_1 \models \alpha_{1i} \rhd \gamma_{1i}$ iff $\mathcal{R}_1 \models \alpha_{1i} \to \gamma_{1i}$ and, because of Condition 1b, this implies $\|_Y \mathcal{R}_n \models \alpha_{1i} \to \gamma_{1i}$. The same reasoning holds for the other components $\mathcal{R}_n$ and its A/G statements in view of Condition 1a. Additionally, $\|_Y \mathcal{R}_n \models p_1 = p_2$ for each $(p_1, p_2) \in Y$, because of the very definition of the synchronous composition and of satisfaction. Thus, $\|_Y \mathcal{R}_n$ satisfies all conjuncts in the left-hand side of Condition 2. And, thus, it satisfies the right-hand side, that is, $\|_Y \mathcal{R}_n \models \alpha \to \gamma$ which, again because of Theorem 4, is equivalent to $\|_Y \mathcal{R}_n \models \alpha \rhd \gamma$. □

Each of the conditions included in Condition 1a asks for an A/G statement to hold in a component. Often, a single A/G statement is asked from each component, that is, $\ell_n = 1$ for some or all n. In particular, the statement that $\mathcal{R}_1 \models \varphi$ implies $\mathcal{R}_1 \|_Y \mathcal{E} \models \varphi$, can be seen as the particular case where $n = 2$, $\ell_1 = 1$, $\ell_2 = 0$, $\mathcal{R}_2 = \mathcal{E}$, and $\alpha_{11} = \alpha = \texttt{true}$.

This theorem allows to reduce the proof of an A/G statement on a composed system to similar proofs on smaller systems, plus checking the validity of an LTL formula. The word

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

reduce in the previous sentence is questionable, because the number of tasks seems to have increased. In addition, obtaining the formulas needed in the premises is not always easy. (We will have something more to say about this in Section 11.3.) The positive side is that each statement has to be proved now in a smaller model. And that, once proved, each component can be reused with no need for new proofs.

We want to remark that Maude provides the tools needed for compositional verification using Theorem 5: the model checker can be used to verify $\mathcal{R}_1 \models \alpha_{11} \to \gamma_{11}$ and the other similar results, and the tautology checker can be used to check the validity of the final formula. Maude is not able to handle our properties, so the formulas must be transformed to use only Boolean propositions as discussed in Section 6.1.

Condition 1b holds for safety formulas, as shown in Proposition 15, or in the presence of fairness between components and absence of emerging deadlocks, as shown in Proposition 14. A case of interest is when the assumption α implies such fairness and deadlock freeness requirements. An instance of this is the following. In each $\mathcal{R}_n$, there is a property t_n defined so that it holds true at each transition and false at each state of $\mathcal{R}_n$. In addition, the formula α includes as a conjunct (or implies otherwise) the formula $\varphi_n = \bigwedge_n (\Box \Diamond t_n = \mathtt{true} \wedge \Box \Diamond t_n = \mathtt{false})$, which means that component $\mathcal{R}_n$ advances infinitely often, which implies by itself fairness between components and absence of emerging deadlocks. Those formulas φ_n need not be appropriate for every case. With our definitions, terminating systems may be fair and still not satisfy φ_n. In each case, more refined formulas may be better suited.

It may be worth noting, to avoid confusion, that $\mathcal{R}_n \models \varphi_n$ for all n does not entail fairness or deadlock freeness. The reason is that it may happen that $\mathcal{R}_n \models \varphi_n$ but $\mathcal{R} \not\models \varphi_n$ because, well, deadlocks or lack of fairness. Things are different when we use the φ_n, not as guarantees, but as assumptions, which we did in the previous paragraph.

There is another way to verify a compositional specification, which is to split it into a monolithic one and use standard verification techniques on it. This works thanks to the following proposition.

Proposition 19
With the notational conventions used so far, for $\mathcal{R}_n$ egalitarian rewrite systems for $n = 1, \ldots, N$, we have that

$$\|_Y \mathcal{R}_n \models \alpha \rhd \gamma \quad \text{iff} \quad \mathrm{split}(\|_Y \mathcal{R}_n) \models \alpha \rhd \gamma.$$

Proof
This is an easy corollary of Theorem 4 and Proposition 9. □

9.1 Example: chained buffers (continued)

This continues the example from Section 2.1. It is immediate to prove that each of the buffers satisfies $\mathtt{BUFFER}n \models \Diamond\,\mathtt{isReceiving} \to \Diamond\,\mathtt{isSending}$. Therefore, by Theorem 4, $\mathtt{BUFFER}n \models \Diamond\,\mathtt{isReceiving} \rhd \Diamond\,\mathtt{isSending}$. We expect to be able to prove a similar behavior for the whole chain of buffers, `3BUFFERS`, using Theorem 5. Concretely, in this case:

- $N = 3$,
- $\mathcal{R}_n = \mathtt{BUFFER}n$,
- $\ell_n = 1$,

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

- $\alpha_{n1} = \Diamond\,\texttt{BUFFER}n\texttt{\$isReceiving}$, $\gamma_{n1} = \Diamond\,\texttt{BUFFER}n\texttt{\$isSending}$,
- $\alpha = \Diamond\,\texttt{BUFFER1\$isReceiving}$, $\gamma = \Diamond\,\texttt{BUFFER3\$isSending}$.

Regarding the conditions in Theorem 5: Condition 1a (that is, $\mathcal{R}_n \models \alpha_{n1} \rhd \gamma_{n1}$ for each n) has already been justified, and Condition 2 (implication for temporal formulas) is easily seen to hold. Fairness is ensured by the way the buffers synchronize, and we hope it is clear that no deadlocks can emerge, so also Condition 1b holds. From which we can deduce

$$\texttt{3BUFFERS} \models \Diamond\,\texttt{BUFFER1\$isReceiving} \rhd \Diamond\,\texttt{BUFFER3\$isSending}.$$

The properties `BUFFER1$isReceiving` and `BUFFER3$isSending` can be better seen here as properties of `3BUFFERS`, and our extension to Maude's syntax allows to define synonyms, so that the above can also be written as

$$\texttt{3BUFFERS} \models \Diamond\,\texttt{3BUFFERS\$isReceiving} \rhd \Diamond\,\texttt{3BUFFERS\$isSending}.$$

9.2 Example: mutual exclusion (continued)

We finish now our discussion of the example from Sections 2.2 and 8.3. It is easy to prove, either by model checking or by mere inspection, that

$$\texttt{MUTEX} \models \Box\,\neg(\texttt{isGranting(1)} \wedge \texttt{isGranting(2)}).$$

Reasoning intuitively, we know that the same formula holds when `MUTEX` is made a component of `SAFE-TRAINS`. And, because the composition requires each `isGranting` property to be synchronized with the corresponding `isCrossing`, we deduce

$$\texttt{SAFE-TRAINS} \models \Box\,\neg(\texttt{S-TRAIN1\$isCrossing} \wedge \texttt{S-TRAIN2\$isCrossing}).$$

Formally, we have used Theorem 5 with

- $N = 3$,
- $\mathcal{R}_1 = \texttt{S-TRAIN1}$, $\mathcal{R}_2 = \texttt{S-TRAIN2}$, $\mathcal{R}_3 = \texttt{MUTEX}$,
- $\ell_1 = \ell_2 = 0$, $\ell_3 = 1$,
- $\alpha_{11} = \texttt{true}$, $\gamma_{11} = \Box\,\neg(\texttt{isGranting(1)} \wedge \texttt{isGranting(2)})$,
- $\alpha = \texttt{true}$, $\gamma = \Box\,\neg(\texttt{S-TRAIN1\$isCrossing} \wedge \texttt{S-TRAIN2\$isCrossing})$.

Component fairness does not always hold, but the formulas involved are safety ones, which is enough according to Proposition 15.

It was observed before that the system `MUTEX` can be seen as a controller or strategy. The verification task is, in this case, of a different nature from the one in the previous example on chained buffers, in which the behavior of the compound necessarily results from the interactions between the components.

9.3 Example: crossing the river (continued)

We now verify the example whose compositional specification was given in Section 2.3.

We want to prove that the composed system satisfies the formula $\Diamond\,\texttt{success}$, where the property is defined in `RIVER` like this:

```
ppt success : -> Bool .
eq success @ (noBelong |~| mark farmer wolf goat cabbage) = true .
eq success @ G = false [owise] .
```

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

We consider a `success` that all items, including the `mark`, are on the same side of the river. So, our hypothesis is that each possible sequence of crossings executed according to our two guidelines leads to a valid solution.

Our implementation allows to transform the four-component system into a single standard one (that is, to apply the split), and use Maude's model checker in the resulting system. This split approach has the advantage that we do not need to find the A/G statements for each component. However, in this case the needed A/G statements for `RIVER-W-PREV`, `AVOID1`, and `AVOID2` are quite clear. So, we work compositionally on those three, although we use the split below to verify the two-component system `RIVER-W-PREV`. Namely, we need `RIVER-W-PREV` to satisfy the A/G statement

$$\texttt{RIVER-W-PREV} \models \Box\neg\texttt{danger} \wedge \Box\neg\texttt{undoing} \ \triangleright\ \Diamond\,\texttt{success}.$$

In words: success is eventually reached assuming the environment allows neither dangerous situations nor undoings. If we are able to prove this, and having into account that, quite obviously, `AVOID1` and `AVOID2` satisfy $\Box\neg\texttt{avoid}$, we can use Theorem 5 to deduce that the four-component system satisfies $\Diamond\,\texttt{success}$. Thus, we perform the split on `RIVER-W-PREV` and model check it for $\Box\neg\texttt{danger} \wedge \Box\neg\texttt{undoing} \ \rightarrow\ \Diamond\,\texttt{success}$, as allowed by Theorem 4.

The result is that the formula *does not hold*, and the model checker hands us a counterexample: an infinite execution that never gets to the desired state. On inspection, we find out that the problem with our solution stems from a symmetry between the roles of the wolf and the cabbage. For example, suppose we are in this situation (in which we omit the `mark`, because its location does not make any difference):

```
farmer wolf goat |~| cabbage
```

Then, the farmer crosses with the wolf, to get

```
goat |~| farmer wolf cabbage
```

and, then, the farmer crosses back with the cabbage to get

```
farmer cabbage goat |~| wolf
```

The new situation is symmetric to the first one, because the roles of cabbage and wolf are similar: eating can take place whenever the goat is left unattended with any of them. As critical as the difference may be for the goat itself, it is irrelevant for us who eats whom. Indeed, if a solution is obtained for a specific situation, the corresponding symmetric solution can be applied to the symmetric situation.

At the end, what we need is to strengthen the concept of undoing to avoid also symmetric movements, which we get by adding two equations to the definition of the property `undoing`:

```
eq undoing @ (wolf > cabbage) = true .
eq undoing @ (cabbage > wolf) = true .
```

Now, the A/G satisfaction holds, showing that the strengthened guidelines are sufficient. Indeed, only two solutions are left, one symmetric to the other, both optimal in their number of moves. And, fixed this part of the composed system, we already know the whole compound works.

10 Additional examples

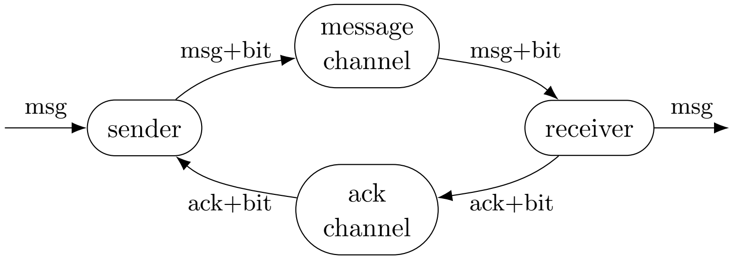
The examples used in this paper up to now have been chosen to be illustrative, so they are rather simple on purpose. Because of such simplicity, we have been able to omit many details of our implementation, which are unimportant for the theoretical work here presented. In this section, we offer a cursory overview of two more complex examples which were presented and discussed at length in (Martín 2021a, Chapter 7). Our presentation in this section is necessarily incomplete. The source code for the examples is available online (Martín 2021b). Both examples have been run through our prototype implementation and the results have been verified using Maude's toolset with the techniques described in this paper.

It has been mentioned that there are two ways to verify a compositional rewriting-logic specification. The first is to perform a compositional verification, according to Theorem 5, using A/G assertions for each component. This is the approach taken in the examples shown so far in this paper, where the A/G assertions for the components were easily found. The second way, justified by Proposition 19, consists in splitting the compositional specification to obtain an equivalent monolithic one which is then monolithically verified. This is the technique illustrated in the new examples we are presenting next. In them, finding the A/G assertions for the component systems turns out to be a difficult task (mainly because there is no general controller, but rather an emergent behavior). So we have used our prototype implementation of the split to obtain a standard Maude module, which is the one we have actually verified using Maude's toolset.

Questions about performance are analyzed in (Martín 2021a, Chapter 7). We quote: "our implementation needs more than two minutes to process the ABP specification and produces more than 18,000 rules, while the Needham-Schroeder example is processed in only two seconds and produces less than 600 rules." These numbers are largely dependent on the implementation we are using. Again, more details are in (Martín 2021a).

10.1 Alternating bit protocol

The first example is a specification of the alternating bit protocol, ABP from now on, to send messages reliably on a channel which may lose some of the packets it receives. We consider an ABP system as consisting of four interacting components:



The sender and the receiver are the components which implement the protocol. There are two channels: one for transmitting messages it gets from the sender; the other for transmitting acknowledgments back. The internal workings of the two channels are the same. Missing, at the two ends of the diagram, are a producer and a consumer. Thus, the result of our four-component specification is meant to be used in turn as a component in a larger system.

When specifying systems of some complexity, we like to enforce modularity further by using the syntax for parametric specification in Maude (Martín *et al.* 2018). In our case, the final module, which represents the whole ABP system, is specified as

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

```
emod ABP-BP{Sndr    :: SENDER-IF,
            MsgChnl :: CHANNEL-IF,
            AckChnl :: CHANNEL-IF,
            Rcvr    :: RECEIVER-IF} is
   sync Sndr || MsgChnl || AckChnl || Rcvr
      on ...
endem
```

We have hidden the synchronization criteria. The important point here is that the module ABP-BP is parametric in the four modules it receives, each modeling one of the components: a sender, two channels, and a receiver. The suffix -BP stands for *blueprint*, and -IF stands for *interface*. For example, Sndr is the name given to the first formal parameter, which is to be instantiated with a module that implements the interface specified in SENDER-IF. Namely, SENDER-IF (which is called a *theory* in Maude jargon) includes the declaration of the following five properties:

```
ppt msgIn : -> Msg? .
ppts msgPckOut ackPckIn : -> Packet? .
ppts canAckChnlPass canMsgChnlGet : -> Bool .
```

The interfaces contain declarations, and no implementation. Then, whichever module defines these properties is valid as the first argument to build an ABP. In the end, after we have specified the needed modules which fit the interfaces, with respective module names Sender, MsgChannel, AckChannel, Receiver, we obtain the final result with

```
emod ABP is
   inc ABP-BP{Sender, MsgChannel, AckChannel, Receiver} .
endem
```

For the two channels, their specification includes the possibility that messages are lost. As a consequence, a fairness constraint is required for each channel, to ensure that at least some messages get through. Namely, we use the assumption $\square\lozenge\,\texttt{isPassing}$, where isPassing is a Boolean property defined to be true exactly when a message is going out of the channel. We also need fairness assumptions on the sender and the receiver, which we do not care to show here.

For verification, given those assumptions, we want to prove the following formula:

$$\gamma \;=\; \square(\texttt{isAccepting} \rightarrow (\texttt{isAccepting}\ \mathbf{U}\ (\neg\,\texttt{isAccepting}\ \mathbf{U}\ \texttt{isDelivering}))).$$

The property isAccepting is meant to be true whenever the sender gets a message from some producer process (that is, when the sender is executing a transition to the purpose of getting such a message); similarly, isDelivering is true whenever the receiver gives the message to some consumer process. In words, this says that to each input to the ABP system follows an output, with no other input in between. This can also be interpreted as saying that "at the next stage of interest" isDelivering holds (see discussion in Section 5).

If we call α the conjunction of the four fairness constraints mentioned above, we want to prove $\texttt{ABP} \models \alpha \rhd \gamma$. To prove it, we first use our prototype implementation to obtain a monolithic standard Maude module equivalent to the original compositional specification of ABP, then we verify on the resulting module the formula $\alpha \rightarrow \gamma$ using Maude's model checker.

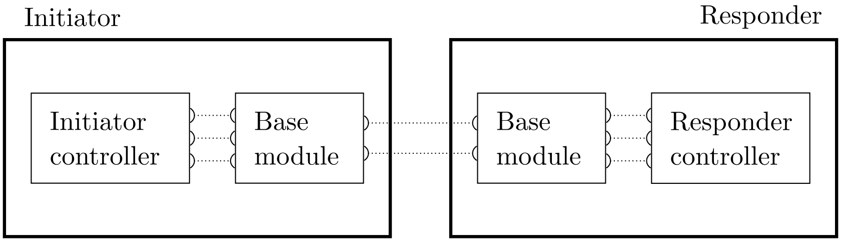
10.2 Needham-Schroeder public-key protocol

Needham-Schroeder is a public-key protocol for safe communication between two actors: an initiator and a responder. It is known to be unsafe in the presence of an attacker, but we are here interested in the simple case where there is no attacker.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

This example illustrates the convenience of compositionality in system specification in two ways. First, the two actors (initiator and responder, Alice and Bob) are specified as independent systems, and only later made to interact. Second, and more to the point of the purpose of this example, each actor is specified as a base module describing all its nondeterministic capabilities (sending, encrypting…), which is then controlled by another module making it behave actually as initiator or responder. The technique we use to make possible this control is, roughly speaking, the use of a language by which the controller sends commands to the base module. This is achieved, of course, through the use of properties and synchronous composition.

This diagram shows the components we model in Maude:



Each of the small arcs represents a property, and the dotted lines represent synchronization. The initiator and the responder both implement the interface theory called `ACTOR-IF`. The base modules, which we call `INITIATOR-BASE` and `RESPONDER-BASE`, are exactly the same except for their initial states. The blueprint for the total system is this:

```
emod NSPKP-BP{I :: ACTOR-IF, R :: ACTOR-IF} is
   sync I || R
      on R$msgRcv := I$msgSnd
      /\ I$msgRcv := R$msgSnd .
   ...
   ag True |> [] <> isCommEstablishedInI /\ [] <> isCommEstablishedInR .
endem
```

This is again a parameterized module, which has to be fed with implementations for the initiator `I` and the responder `R`. The sentence with the **`ag`** keyword is an A/G assertion saying that, with no assumption (`True`), the module has to guarantee that communication is established arbitrarily often for both actors. (The ABP example also had A/G assertions, which we preferred to omit to simplify the presentation.) The symbol `:=` in the synchronization criteria is, for our purposes here, equivalent to the `=` we have used all the time.

To implement control, we have used a very simple language of commands that the controller issues and the base module executes. For example, the `INITIATOR` is specified as the composition of `INITIATOR-PROTOCOL` and `INITIATOR-BASE`:

```
emod INITIATOR is
   sync INITIATOR-PROTOCOL || INITIATOR-BASE
      on INITIATOR-BASE$action := INITIATOR-PROTOCOL$action
      /\ INITIATOR-BASE$arg := INITIATOR-PROTOCOL$arg
      /\ INITIATOR-PROTOCOL$isErrorState := INITIATOR-BASE$isErrorState .
   ...
endem
```

Thus, the base module `INITIATOR-BASE` receives from the controller (by synchronizing properties) the `action` to be performed and the `arg`uments on which to perform them. The feedback to the controller is whether there has been some error (namely, an unsuccessful

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

decryption). The states of the base system are given by a set of pairs key-value. For example, this is the initial state for the `INITIATOR`:

```
eq init = ('myid : alice) ('xid : bob)
          ('myprivkey : priv(alice)) ('xpubkey : pub(bob))
          ('mynonce : nonce(alice)) .
```

Thus, the key `'myid` is storing the value `alice`, and so on. There are rules in the base module to specify the different actions it is able to perform: send, receive, decrypt, encrypt, and check whether the values stored under two given indices are equal. For example:

```
rl D =[ sending | D ]=> D .
rl D =[ receiving(M) | D ]=> D <+ ('msg : M) .
```

So, sending leads to no change in the values stored, but receiving adds or overwrites a pair with the key `'msg` and the value of whatever it received. Slightly more complex rules implement the rest of the capabilities.

Then, these are the rules for the `INITIATOR-PROTOCOL` that makes `INITIATOR-BASE` behave as an actual initiator for the protocol:

```
rl 1 =[ 1 : encrypt('mynonce 'myid)         ]=> 2 .
rl 2 =[ 2 : send                            ]=> 3 .
rl 3 =[ 3 : receive                         ]=> 4 .
rl 4 =[ 4 : decrypt('recmynonce 'recxnonce) ]=> 5 .
rl 4 =[ 4 : decrypt('recmynonce 'recxnonce) ]=> error .
rl 5 =[ 5 : check('recmynonce, 'mynonce)    ]=> 6 .
rl 5 =[ 5 : check('recmynonce, 'mynonce)    ]=> error .
rl 6 =[ 6 : encrypt('recxnonce)             ]=> 7 .
rl 7 =[ 7 : send                            ]=> 8 .
rl 8 =[ 8 : reset                           ]=> 1 .
```

The states are represented by mere numbers. But the interesting part is that the initiator part of the protocol can be read line by line in the transition terms: first, encrypt the values stored under the keys `'mynonce` and `'myid`, then send the result of the encryption, and so on. The way the properties are defined and the way the synchronization is specified ensures that the `send` in the controller is executed synchronized with the `sending` in the base module. The wildly nondeterministic behavior of the base system is transformed into an almost fully deterministic one once synchronized with the controller.

For verification, as in the previous example, we transform the compositional specification into a monolithic, standard one and, then, use Maude's model checker to verify the formula given in the **`ag`** statement. Again, this method is fast and simple and avoids the costly search for the components' A/G assertions.

11 Closing material

11.1 Related work

We are not aware of any other work dealing precisely with compositional verification and rewriting logic, but certainly our work on compositionality, both for specification and for verification, is inspired by others, including process algebras, coordination models, and many more. Our results on A/G are also strongly based on existing work for other settings (Elkader *et al*. 2018; Jonsson and Yih-Kuen 1996; Abadi and Lamport 1995).

Besides A/G, many other verification techniques are discussed in the literature which are compositional in nature. Often they consist in simplifying the isolated components before composing them into a single global system. This is related to our work on simulation and abstraction in Section 8. Simplification is performed either taking into account the behavior of the environment, or the temporal formula to be proved, or both. Sometimes, the global system is not even produced in full, but instead the global state-space is created on the fly traversing in parallel the components. The paper (Garavel *et al.* 2015) describes these and other techniques and their implementation in the toolset CADP (INRIA 2023). On this same matter of simplifying a component before using it, (André *et al.* 2012) proposes the use of Symbolic Observation Graphs, similar to the predicate abstractions we mentioned also in Section 8. All these works use LTSs to model processes.

In the field of Petri nets, (Klai *et al.* 2005) deals with decomposing a Petri net into smaller ones. Isolated component nets, detached from the rest, are enriched with *abstraction places* representing, in a sense, the environment. It also discusses *non-constraining* interactions between components, a concept similar to our requirement of fairness and deadlock freeness. Their conclusion is worth quoting: "experimental results show that this technique is efficient for some models, but for others the combinatorial explosion is not really attacked." Similar thoughts are expressed by (Garavel *et al.* 2015) and endorsed by our own experience.

11.2 Future work

The most substantial path we would like to explore in the future is the possibility of implementing strategies by synchronous composition. We see strategies in a broad sense, encompassing controllers, protocols, monitors, coordinators… Strategies are applied to nondeterministic systems to guide them, reducing or removing their nondeterminism. The rules of chess allow for many movements from each position. On that, a good strategy reduces the possibilities to probably just one at each point in a match. In the same way, when specifying the behavior of systems, we can specify a base system with all its nondeterministic capabilities and, then, use it under the control of a strategy; even in different ways under different strategies. This idea has been used with Maude and its strategy language to implement Knuth-Bendix-like completion as a basic set of correct rules on which different strategies are applied (Lescanne 1989; Clavel and Meseguer 1997; Verdejo and Martí-Oliet 2012), also for congruence closure (Bachmair *et al.* 2003), and for specifying insertion sort as a base system with a single rule for swapping cell contents which is then conveniently controlled (Martí-Oliet *et al.* 2004; Eker *et al.* 2007). Our examples in this paper and in previous work (Martín 2021a; Martín *et al.* 2020) can be also viewed in this way.

Besides that, and being more concrete, we lack a proof that the procedure in Theorem 5 is complete. That is, it is not proved whether, given α and γ, appropriate formulas α_{ni}, γ_{ni} can always be found. Based on similar results in similar contexts, we conjecture it is complete, but a proof is currently missing.

Adding other similar rules would also enrich our work. In particular, circular deduction rules (Elkader *et al.* 2018) are different enough to deserve our attention. We mean, from $\mathcal{R}_1 \models \varphi \triangleright \varphi'$ and $\mathcal{R}_2 \models \varphi' \triangleright \varphi$, deduce $\mathcal{R}_1 \| \mathcal{R}_2 \models \Phi$ for some formula $\Phi = \Phi(\varphi, \varphi')$.

Some works (Cobleigh *et al.* 2003; Bobaru *et al.* 2008) have shown how A/G reasoning can be automated. And also abstraction can be automated, for example, with the technique known as *counterexample-guided abstraction refinement* (CEGAR) (Clarke *et al.*

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

2000; Chaki *et al.* 2004). This can even be applied to a compositional system specification. Adding any such automation to our implementation would increase its usefulness. Specially the generation of intermediate formulas, because that would mean we have a new completely automated way to verify systems.

Our prototype implementation can be advanced in several ways to make it more complete, efficient, reliable, and easy to use. Also, translating compositional specifications in extended Maude to, for example, CADP syntax would allow the use of the rich CADP toolset for compositional verification.

11.3 Conclusion

There are reasons to be skeptic about the value of compositional verification, and in particular about the A/G technique. The main reason is the difficulty of finding the needed *intermediate* formulas: in a simple case, a compositional proof of $\mathcal{R}_1\|_Y\mathcal{R}_2 \models \varphi$ requires finding a formula γ such that $\mathcal{R}_1 \models \gamma$ and $\mathcal{R}_2 \models \gamma \rhd \varphi$. Finding such a formula γ, or whatever is needed in more complex cases, is in general a difficult task. We have chosen the examples in this paper so that those intermediate formulas are easily found. Examples in previous work, mentioned in Section 10, showed other examples for which the intermediate formulas were not obvious at all, which made us prefer to use the split, finishing with a monolithic verification for a compositional specification.

Techniques have been devised for automatically generating such intermediate formulas (Elkader *et al.* 2018; Cobleigh *et al.* 2003; Bobaru *et al.* 2008). However, an experimental study (Cobleigh *et al.* 2006) on the efficiency of these techniques with two actual tools draws this conclusion: "This discouraging result, although preliminary, raises doubts about the usefulness of assume-guarantee reasoning." In a different style, a computation of the theoretical complexity of A/G (Kupferman and Vardi 2000) finds it to be quite large: "The results of this paper indicate that modular model checking [...] is rather intractable." Additionally, not many of the well-known tools for verification include the possibility of compositional verification, notable exceptions being BIP (Basu *et al.* 2008) and TLA$^+$ (Lamport 2002). Whether this is because their practitioners have not found the need for it or for some other reason, we cannot say. To this, we can add our own, limited experience trying to perform compositional verification within our proposed framework, from which we have learned that the generation of temporal formulas for components is a laborious task. Moreover, the use of Theorem 5, in its Condition 2, requires checking that a certain LTL formula, potentially large, is a tautology. We have used Maude's tautology checker, which in some cases takes very long to reach an answer. In all, if we want to verify a compositionally specified system, the cheaper way, both in human time and in computer time, may well be transforming it into a monolithic one (through the split operation), and performing monolithic model checking on the result. There is ongoing work in this area, so improvements can be expected. It may seem, however, that we need to justify our work on compositional verification. We devote a few lines to it.

First, we have already mentioned at the end of Section 11.1 that both (Klai *et al.* 2005) and (Garavel *et al.* 2015) have found their compositional techniques (which do not include A/G) to be more effective than monolithic ones in some, though not all, cases.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

One of our goals in this work was to show that compositional reasoning in rewriting logic is possible based on our framework for compositional specification. Componentwise abstraction and simulation and the A/G technique, in addition to whatever value they may have by themselves, were chosen by us as case studies to put our framework to the test. After having written this paper, we feel confident that new developments could be adapted as well.

The discouraging studies mentioned above miss a key ingredient of modularity, namely, reuse. They consider compositional verification as if it has to be completely redone from scratch every time. But, once the design of a system has been carried out modularly, the temporal formulas needed from each component have been determined, and the proof that some global formula follows from those of the components has been completed, all that is valid forever. If one of the components has to be modified, refined, or replaced, only the new component needs verification against the formulas already known to be needed from it.

A library of ready-to-use components is another instance of the convenience of modular design and verification. We have already remarked that we are specially interested in studying how strategies can be implemented as components that exert their control by means of synchronous composition. In these cases, the component implementing a strategy has to be independent of the rest, and has to perform its task whatever system it happens to be attached to. Thus, because the mutual exclusion controller in our example satisfies the mutual exclusion property, so does any composed system that relies on it. No need to find intermediate formulas or check complex provisos. In contrast, it is in cases when the global behavior is emergent, as in the ABP example in Section 10.1, that finding the intermediate formulas is a difficult task.

The above discussion has to do with verification. In specification (or design, or modeling) the value of compositionality is less controversial.

For a final summary, our goal was to develop a framework for compositional specification in rewriting logic and Maude and, in the present paper, to show the way for compositional reasoning on such specifications. This much we are confident to have achieved.

We like to think that, through compositionality, rewriting logic can become easier or more suitable to apply to some domains, like runtime verification, coordination models, component-based software development, and hardware specification. All of it is quite speculative at present, which means we have some appealing lines of work ahead of us.

Competing interests

The authors declare none.

References

Abadi, M. and Lamport, L. 1995. Conjoining specifications. *ACM Transactions on Programming Languages and Systems 17,* 3, 507–534.

André, É., Klai, K., Ochi, H. and Petrucci, L. 2012. A counterexample-based incremental and modular verification approach. In *Large-Scale Complex IT Systems. Development, Operation and Management*, R. Calinescu and D. Garlan, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 283–302.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

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

Bae, K. and Meseguer, J. 2014. Predicate abstraction of rewrite theories. In *Rewriting and Typed Lambda Calculi—Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, 14–17 July 2014. Proceedings*, G. Dowek, Ed. Lecture Notes in Computer Science, vol. 8560. Springer, 61–76.

Basu, A., Bozga, M. and Sifakis, J. 2008. Modeling heterogeneous real-time components in BIP. In *Perspectives Workshop: Model Engineering of Complex Systems (MECS 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.

Bobaru, M. G., Păsăreanu, C. S. and Giannakopoulou, D. 2008. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Lecture Notes in Computer Science, vol. 5123. Springer Berlin Heidelberg, 135–148.

Chaki, S., Clarke, E. M., Ouaknine, J., Sharygina, N. and Sinha, N. 2004. State/event-based software model checking. In *Integrated Formal Methods*. Lecture Notes in Computer Science, vol. 2999. Springer Berlin Heidelberg, 128–147.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1855. Springer Berlin Heidelberg, 154–169.

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

Clarke, E. M., Long, D. E. and McMillan, K. L. 1989. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989)*. IEEE Computer Society, 353–362.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R. and Talcott, C. 2022. *Maude Manual (Version 3.2.1)*.

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 Berlin Heidelberg, 1–12.

Cobleigh, J. M., Avrunin, G. S. and Clarke, L. A. 2006. Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning. In *ISSTA'06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. Association for Computing Machinery (ACM).

Cobleigh, J. M., Giannakopoulou, D. and Pasareanu, C. S. 2003. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. Springer Berlin Heidelberg, 331–346.

Eker, S., Martí-Oliet, N., Meseguer, J. and Verdejo, A. 2007. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science 174,* 11, 3–25.

Elkader, K. A., Grumberg, O., Păsăreanu, C. S. and Shoham, S. 2018. Automated circular assume-guarantee reasoning. *Formal Aspects of Computing 30,* 5, 571–595.

Garavel, H., Lang, F. and Mateescu, R. 2015. Compositional verification of asynchronous concurrent systems using cadp. *Acta Informatica 52*, 337 – 392.

Glabbeek, R. V. and Höfner, P. 2019. Progress, justness, and fairness. *ACM Computing Surveys 52,* 4, 1–38.

Grumberg, O. and Long, D. E. 1994. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems 16,* 3, 843–871.

INRIA. 2023. CADP – construction and analysis of distributed processes (website). http://maude.ucm.es/syncprod.

Jonsson, B. and Yih-Kuen, T. 1996. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science 167,* 1-2, 47–72.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press

Klai, K., Haddad, S. and Ilié, J.-M. 2005. Modular verification of Petri nets properties: A structure-based approach. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, F. Wang, Ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 189–203.

Kupferman, O. and Vardi, M. Y. 2000. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems 22,* 1, 87–128.

Lamport, L. 1983. What good is temporal logic? *Information Processing 83*, 657–668.

Lamport, L. 2002. *Specifying Systems*. Pearson Education (US).

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, Barcelona, Spain, 13–17 March 1989* (1989), 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.

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)* (2004-01), N. Martí-Oliet, Ed. Electronic Notes in Theoretical Computer Science, vol. 117. Elsevier BV, 417–441.

Martín, Ó. 2021a. *Composition in Rewriting Logic*. Ph.D. thesis, Universidad Complutense de Madrid - Facultad de Informática. URL: http://eprints.ucm.es/id/eprint/68887.

Martín, Ó. 2021b. Composition in rewriting logic (website). URL: http://maude.ucm.es/syncprod.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2018. Parameterized programming for compositional system specification. In *Rewriting Logic and Its Applications*, V. Rusu, Ed. Lecture Notes in Computer Science, vol. 11152. Springer International Publishing, 59–75.

Martín, Ó., Verdejo, A. and Martí-Oliet, N. 2020. Compositional specification in rewriting logic. *Theory and Practice of Logic Programming 20,* 1, 44–98.

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

Meseguer, J., Palomino, M. and Martí-Oliet, N. 2008. Equational abstractions. *Theoretical Computer Science 403,* 2-3, 239–264.

Misra, J. and Chandy, K. M. 1981. Proofs of networks of processes. *IEEE Transactions on Software Engineering SE-7,* 4, 417–426.

Palomino, M., Martí-Oliet, N. and Verdejo, A. 2005. Playing with Maude. *Electronic Notes in Theoretical Computer Science 124,* 1, 3–23. Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004).

Pnueli, A. 1985. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. NATO ASI Series, vol. 13. Springer Berlin Heidelberg, 123–144.

Rubio, R., Martí-Oliet, N., Pita, I. and Verdejo, A. 2021. Strategies, model checking and branching-time properties in Maude. *Journal of Logical and Algebraic Methods in Programming 123*, 100700.

Verdejo, A. and Martí-Oliet, N. 2012. Basic completion strategies as another application of the Maude strategy language. In *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming* (2012-04), S. Escobar, Ed. Electronic Proceedings in Theoretical Computer Science, vol. 82. Open Publishing Association, 17–36.

https://doi.org/10.1017/S1471068423000340 Published online by Cambridge University Press