Universiteit
Leiden
The Netherlands

An algebra for interaction of cyber-physical components

Lion, B.

Citation

Lion, B. (2023, June 1). *An algebra for interaction of cyber-physical components*. Retrieved from https://hdl.handle.net/1887/3619936

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3619936

Note: To cite this publication please use the final published version (if applicable).

Chapter 5

Experimental framework

The component framework introduced in Chapter 2 and its operational fragments defined in Chapter 4 lay the foundation for the verification of properties of cyber-physical systems.

In this chapter, we detail and evaluate an implementation in Maude of the cyber-physical agent framework introduced in Section 4.2 of Chapter 4. This implementation extends the operational formal model with three additional features. First, an agent is equipped with an internal strategy, similar to the one introduced in Section 4.3. Thus, the Maude implementation enables two levels to make a preference aware system more specific: by selecting a subset of best actions either at the agent level, or at the system level. Second, an agent may perform a composite action *atomically*, i.e., a sequence of actions within the same clique. As a result, the value assigned to action parameters may depend on the effects of actions earlier in the sequence. The Maude implementation enables agents to reevaluate the parameters of their actions at runtime. Third and last, we give some constraints on how a set of atomic actions, called *macrostep*, is serialized into a sequence of *microsteps*, i.e., a sequence of agent actions. More precisely, we impose that such serialization is a function given as parameter for simulation or analysis. The runtime may still be non-deterministic, as several different cliques may be enabled at the same time. The above three features are detailed in Section 5.1.

We use our implementation to simulate and analyze a series of applications. More precisely, we use the Maude runtime to verify trace properties of concurrent systems. Recall that, as defined in Chapter 2, a trace property is a set of TESs, and a component C satisfies a property P, denotes as $C \models P$, if and only if the behavior of C is a subset of the property P. In Section 2.2.2, we introduced the notion of conformance, which states that a component C is conformable to a component C' if there exists a non empty protocol P such that $C \times_\Sigma P \sqsubseteq C'$.

In the agent framework, a system is a set of interacting agents for which we gave in Section 4.2 a compositional semantics as components. Therefore, given a set of n agents $\mathcal{A}_1(s_{01}, t_0), ..., \mathcal{A}_n(s_{0n}, t_0)$ interacting under the interaction signature Σ, we say that the system $\mathcal{S} = \mathcal{A}_1(s_{01}, t_0) \times_\Sigma ... \times_\Sigma \mathcal{A}_n(s_{0n}, t_0)$ satisfies property P if the component semantics does so, i.e., if $[\![\mathcal{S}]\!] \models P$ (see Theorem 9 for soundness). In the case where $[\![S]\!] \not\models P$, we then identify two mechanisms to make $[\![S]\!]$ satisfy P.

The most obvious way is to substitute each of a subset of the agents $\mathcal{A}_i$ with a more specific agent $\mathcal{A}'_i$ such that the resulting system satisfies P. Such agent $\mathcal{A}'_i$ is more specific than agent $\mathcal{A}_i$ due to its smaller state space and behavior (where all TESs that violate property P have been removed). Finding the largest of such $\mathcal{A}'_i$ is therefore of primary importance to keep as much as possible the non-violating TESs from the behavior of agent $\mathcal{A}_i$.

An alternative way is to synthesize a coordinator agent D such that $[\![\mathcal{S} \times_\Sigma D]\!] \models P$. While similar to the first method, as it generates a system $\mathcal{C}' = \mathcal{S} \times_\Sigma D$, the coordinator D is a separate entity that can therefore be modified. In the case of a system $\mathcal{A}_1 \times_\Sigma \mathcal{A}_2$, composite of agents $\mathcal{A}_1$ and $\mathcal{A}_2$, that does not satisfy a property P, a coordinator may filter actions from $\mathcal{A}_1$ and $\mathcal{A}_2$ contextually, i.e., imposing a relation between actions occurring in $\mathcal{A}_1$ and $\mathcal{A}_2$.

We instantiate the framework to model diverse applications:

- cyber agents interacting via Reo coordination protocols. We give an implementation of a Reo nodes, primitive channels, and show how to compose protocols.

- N reservoirs, a valve, and a controller. We explore how the controller can control the valve to maintain a safety property, such as that the N reservoirs always have a water level within some thresholds.

- two robot agents, each interacting with a (shared) field and a (private) battery agent. We explore how safety, liveness, and coordination properties are enforced by a system of agents. For instance, such system is energy safe if the battery levels always stay above some thresholds. The system is alive if the agents keep patrolling between two locations on the field. Finally, coordination properties are such that agents eventually sort themselves on the grid by correctly exchanging their locations.

We run some analysis on the system using the Maude reachability search engine.

5.1 Maude framework for cyber-physical agents

The Maude framework is an instance of the general agent framework introduced in Chapter 4. We therefore introduce the Maude implementation for *actions*, *agents*, *system of agents*, and *composability relation.* The implementation is accessible at [59].

Actions An action is a pair that contains the name of the action, and the set of agent identifiers on which the action applies. An agent action is identified by the source agent identifier, and is a triple `(id, (a; ids))` where `id` is the agent doing the action with name `a` onto the set of agents `ids`, that we call resources of agent `id` for action named `a`.

```
fmod ACTION is
    inc STRING . inc BOOL . inc SET{Id} . ...
    sort AName Action AgentAction .
    op (_;_) : AName Set{Id} -> Action [ctor] .
    op (_,_) : Id Action -> AgentAction [ctor] .
    op mta : -> AgentAction .
endfm
```

Agent The `AGENT` module in Listing 5.1 defines the theories on which an agent relies, the `Agent` sort, and operations that an agent instance must implement. The module is parametrized with a `CSEMIRING` theory, that is used to rank actions of an agent. Additionally, the `AGENT` includes modules that define state and action terms. A term of sort `IdStates` is a pair of an identifier and a map of sort `MapKD`.

A term of sort `Agent` is a tuple `[id: C| state; ready?; softaction]`. The identifier `id` is unique for each agent of the same class `C`. The state `state` of an agent is a map from keys to values. For instance, the state of a robot has three keys, `position`, `energy`, and `lastAction`, with values in `Location`, `Status`, and `Bool`. The flag `ready?` is of sort `Bool` and is `True` when the agent has submitted a possibly empty list of actions, and `False` otherwise. The pending actions `softaction` is a set of actions valued in the parametrized `CSEMIRING`. The use of a constraint semiring as a structure for action valuations enables various kinds of reasoning about preferences at the agent and system levels. We use the two operations of the csemiring, sum $+$ and product $\times$, as respectively modeling the choice and the compromise of two alternatives. See [91, 84, 50] for more details.

As shown in Listing 5.1, an agent instance implements four operations: `computeActions`, `resolve`, `getOutput`, `getPostState`, and `internalUpdate`. Note that the four operations are an implementation of the abstract ϕ of Section 4.2.1. The operation `computeActions`, given a `state:MapKD` of agent `id` of class `C`, returns a set of valued actions in the parametrized `CSEMIRING`. The operation `internalUpdate`, given a `state:MapKD` of agent `id` of class `C`, returns a new state `state':MapKD`. For instance, an agent may record in its state, as an internal update, the outcome of `computeActions` that returns the set of possible actions for the agent. The `getOutput` operation, given an action name `a:Name` from agent identified by `id2` applied to an agent `id` of class `C` in a state `state`, returns a collection of outputs. The outputs generated by `getOutput` are of sort `MapKD` and therefore structured as a mapping from keys to values. For instance, the output of the action named `read` applied on a `FIELD` agent has a key `pos` that maps to the position value of the agent doing the `read` action. The operation `getPostState`, given an action name `a:AName` with inputs `input:IdStates` from agent identified by `id2` applied on an agent `id1` of class `C` in a state `state`, returns a new state. The input `input:IdStates` is a collection of key to value mappings that results from collecting the outputs, i.e., with `getOutput`, of an action `(id, an, ids)` on all its resources in `ids`. The new state returned by `getPostState` describes how the agent reacts to the input action, which could also capture, with an error state, that the action is not allowed by the agent. The operation `resolve`, given an action name `a:Name` performed by an agent with identifier `id` with class `C` in state `state`, returns a new action name `a':Name`. The `resolve` operation is called before the input action name is performed, and may instantiate some parameters of the action given the state of the agent.

Listing 5.1: Extract from the `AGENT` Maude module.

```
fmod AGENT{X :: CSEMIRING} is
 inc IDSTATE .  inc ACTION .
 sort Agent .
 op [_:_|_;_;_] : Id Class MapKD Bool X$Elt -> Agent [ctor].
 op computeActions : Id Class MapKD -> X$Elt .
 op internalUpdate : Id Class MapKD -> MapKD .
 op getPostState : Id Class Id AName IdStates MapKD -> MapKD
 op getOutput : Id Class Id AName MapKD -> MapKD .
 op resolve : AName Identifier Class MapKD -> AName .
endfm
```

The agent's dynamics are given by the rewrite rule in Listing 5.2, that updates the pending action to select one atomic action from the set of valued actions:

Listing 5.2: Conditional rewrite rule applying on agent terms.

```
crl[agent] : [sys [id : ac | state  ; false ; null]] =>
      [sys [id : ac | state' ; true  ; softaction]]
    if softaction + sactions := computeActions(id, ac, state)
       /\ state' := internalUpdate(id, ac, state) .
```

The rewrite rule in Listing 5.2 implements the abstract rule of Equation 4.2. After application of the rewrite rule, the `ready?` flag of the agent is set to `True`. The agent may, as well, perform an internal update independent of the success of the selected action.

Moreover, an agent module comes with the definition of an interface. The interface for an agent contains the constructors for action names, i.e., `move: direction -> Action` and `read: sensorName -> Action` for a TROLL robot agent. An agent that interacts with another agent must therefore include the interface module of that agent, i.e., the set of actions that the agent performs.

System The `SYSTEM` module in Listing 5.3 defines the sorts and operations that apply on a set of agents. The sort `Sys` contains set of `Agent` terms, and the term `Global` designates top level terms on which the system rewrite rule applies (as shown in Listing 5.4). The `SYSTEM` module includes the `Agent` theory parametrized with a fixed semiring `ASemiring`. The theory `ASemiring` defines valued actions as pairs of an action and a semiring value. While we assume that all agents share the same valuation structure, we can also define systems in which such a preference structure differs for each agent. The `SYSTEM` module defines four operations: `linearization`, `outputFromAction`, `updateSystemFromAction`, and `updateSystem`. The operation `linearization` returns a list of `AgentAction` given a set of `AgentAction`. As there are multiple ways to generate sequences of actions from a set of actions, we assume a total order among actions and a sorted output sequence. As several total orders may exist, we leave the equational specification of the linearization operation in each scenario. The operation `outputFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, a collection of identified outputs given by the union of the results of `getOutput` produced by all agents in `ids`. The operation `updatedSystemFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, an updated system `sys'`. The updated system may raise an error if the action is not allowed by some of the resource agents in `ids` (see the battery-field-robot example in 5.4). The updated system, otherwise, updates *synchronously* all agents with identifiers in `ids` by using the `getPostState` operation. The operation updateSystem returns, given a list of agent actions agentActions and a system term sys, a new system updateSystem(sys, agentActions) that performs a sequential update of sys with every action in agentActions using updatedSystemFromAction. The list agentActions ends with a delimiter action end performed on every agent, which may trigger an error if some expected action does not occur (see PROTOCOL in Section 5.4).

Listing 5.3: Extract from the SYSTEM Maude module.

```
fmod SYS is
  inc AGENT{ASemiring} . sort Sys  Global .
  subsort Agent < Sys . op [_] : Sys -> Global [ctor] .
  op __ :  Sys Sys -> Sys [ctor assoc comm id: mt] .   ...
  op linearization : Set{AgentAction} -> List{AgentAction} .
  op outputFromAction : AgentAction Sys -> IdStates .
  op updatedSystemFromAction : AgentAction Sys -> Sys .
  op updateSystem : Sys List{AgentAction} -> Sys .
endfm
```

The rewrite rule in Listing 5.4 applies on terms of sort Global and updates each agent of the system synchronously, given that their actions are composable. The rewrite rule in Listing 5.4 implements the abstract rule of Equation 4.5. The rewrite rule is conditional on essentially two predicates: agentsReady? and kbestActions. The predicate agentsReady? is True if every agent has its ready? flag set to True, i.e., the agent rewrite rule has already been applied. The operation kbestActions returns a ranked set of cliques (i.e., composable lists of actions), each paired with the updated system. The element of the ranked set are lists of actions containing at most one action for each agent, and paired with the system resulting from the application of updateSystem. If the updated system has reached a notAllowed state, then the list of actions is not composable and is discarded. The operations getSysSoftActions and buildComposite form the set of lists of composite actions, from the agent's set of ranked actions, by composing actions and joining their preferences.

Listing 5.4: Conditional rewrite rule applying on system terms.

```
crl[transition] : [sys]  => [sys']
 if agentsReady?(sys)  /\ saAtom := getSysSoftActions(sys) /\
  saComp := buildComposite(saAtom , sizeOfSum(saAtom)) /\
  p(actseq, sys') ; actseqs := kbestActions(saComp, k, sys) .
```

Composability relation The term saComp defines a set of valued lists of actions. Each element of saComp possibly defines a clique. The operation kbestActions spec-

ifies which, from the set `saComp`, are cliques. We describe below the implementation of `kbestActions`, given the structure of action terms.
An action is a triple `(id, (an, ids))`, where `id` is the identifier of the agent performing the action `an` on resource agents `ids`. Each resource agent in `ids` reacts to the action `(id, (an, ids))` by producing an output `(id', an, O)` (i.e., the result of `getOutput`). Therefore, $\texttt{comp}((\texttt{id}, (\texttt{an}, \texttt{ids})), \texttt{a}_\texttt{i})$ holds, with $\texttt{a}_\texttt{i} : \texttt{Action}_\texttt{i}$ and $i \in \texttt{ids}$, only if $\texttt{a}_\texttt{i}$ is a list that contains an output `(i, an, O)`, i.e., an output to the action. If one of the resources outputs the value `(i,notAllowed(an))`, the set is discarded as the actions are not pairwise composable. Conceptually, there are as many action names `an` as possible outputs from the resources, and the system rule (4.2) selects the clique for which the action name and the outputs have the same value. In practice, the list of outputs from the resources get passed to the agent performing the action.

5.2 Concurrent Reo

In Chapter 3, we use our component algebra as a semantic model for Reo. More particularly, we introduce nodes as primitive components, and channels or connectors as the composition of ports under some fixed composition operators. In this section, we use our Maude implementation to provide a concurrent implementation of Reo connectors, as a set of interacting agents.

5.2.1 Reo primitives as agents

We refer to Section 3.1 for an introduction to Reo. The available implementation [64] of Reo focuses on compilation of input circuits to an executable. A remaining challenge was to construct a compositional runtime where each part of a Reo circuit can be compiled independently and run concurrently. As a consequence of such framework, one can keep the structure of a Reo circuit at runtime, mix different semantics for Reo channels (e.g., guarded commands, constraint automata, ...), while allowing for simulation and verification through reachability queries.

Our framework for concurrent Reo consists therefore of few primitive agents: `PORT`, `CHANNEL`, `CONNECTOR`.

Port as resource A port is a point of synchronization in Reo, and its typical behavior is to forward atomically data from its input connector to its output connector. We fix a port identifier to be of the form `P(i)` where `i:Float` is a rational number.

We later use the identifier of a port to define the operation of `linearization` and order actions.

A port contains a structure with two buffers that implement the atomic passing of data from the input to the output connector. One buffer collects the data that the input connector may *put*, and the other contains the request from the output connector to *take* some data. Only when the two buffers are full, as shown with the error raised by the `end` action, should the port allow both `put(d)` and `take` actions.

```
fmod PORT is
  inc AGENT{ASemiring} .
  inc PROTOCOL-INTERFACE .
  inc PORT-INTERFACE .
    ...
  eq computeActions(id, Port, M ) = null .

  ceq getPostState(r, Port, id, take, mtOutput, M) = M'
    if M[k("data")] =/= nodata /\
       M' := insert(k("sync"), bd(true), M ) .

  ceq getPostState(r, Port, id, put(data), mtOutput, M) = M'
    if M[k("data")] == nodata /\
       M' := insert(k("data"), data, M ) .

  ceq getPostState(id, Port, id, end, inputs, M) = M'
    if M[k("data")] =/= nodata /\ M[k("sync")] == bd(true) /\
       M' := insert(k("data"), nodata,
                insert(k("sync"), bd(false), M)) .
  ceq getPostState(id, Port, id, end, inputs, M) =
        notAllowed(end)
    if M[k("data")] =/= nodata or M[k("sync")] == bd(true) .

  eq getPostState(r, Port, id, a, inputs, M) = M [owise] .

  ceq getOutput(r, Port, id, take, M) =
    k("data") |-> M[k("data")] if M[k("data")] =/= nodata .

  eq getOutput(r, Port, id', a, M) = empty [owise] .

  eq internalUpdate(id, Port, M) = M [owise] .

endfm
```

Connectors as agents We give three instances of primitive Reo connectors: a SYNC, a FIFO, and a MERGER. Other primitive connectors, such as a replicator, syncdrain, etc, could be defined similarly. While we do not expand on this point here, some parametric connectors such as alternator(n) could be defined recursively as well, making use of the port naming structure.

A SYNC agent has two ports on which it acts synchronously. The action of a SYNC agent consists of an atomic sequence of two actions: a take on the input port, and a put on the output port. The composite action succeeds if and only if the two parts of the action succeeds, and the value *put* on the output port corresponds to the value *taken* on the input port. Note that the value of the datum that a sync puts on its output port is initially unknown. We use the symbol ? for such unknown value, and use the operation resolve to instantiate the value at runtime.

```
fmod SYNC is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
  ...
  eq getOutput(Sync(p1, p2), Channel, id', a, M) = empty .

  eq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), take, (id, k("data
      ") |-> d), M) = insert(k("data"), d, M) .

  eq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), take, inputs, M) =
       notAllowed(take) [owise] .

  ceq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), put(d), outputs,
      M) = insert(k("data"), nodata, M)
      if M =/= notAllowed(take) .

  eq getPostState(Sync(p1, p2), Channel, id, a, outputs, M) = M [owise] .

  eq computeActions(Sync(p1, p2) , Channel, M ) = (((Sync(p1, p2) , (take
       ; p1)), (Sync(p1, p2) , (put(?) ;  p2))), 1) .

  eq internalUpdate(Sync(p1, p2), Channel, M) = M .

  ceq resolve(put(?), Sync(p1, p2), Channel, M) = put(d)
      if d := M[k("data")] .
endfm
```

A FIFO agent has two ports on which it acts in sequence. A FIFO agent has two actions, a take action that stores the data from its input port to a memory, or a put action that outputs the data on the output port. The `take` action succeeds only if the current memory cell is empty, and the `put` action succeeds only if the current memory cell is full. As a result, the `FIFO` agent alternates between taking a value from its input port, and putting that value to its output port.

```
fmod FIFO is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
  ...
  eq getOutput(id, Channel, id', a, M) = empty .

  eq getPostState(Fifo(p1, p2), Channel, Fifo(p1, p2), take, (id, k("data
      ") |-> d), M) = insert(k("data"), d, insert(k("state"), nd(1), M))
      .
  eq getPostState(Fifo(p1, p2), Channel, Fifo(p1, p2), put(d), outputs, M
      ) = insert(k("data"), nodata, insert(k("state"), nd(0), M)) .
  eq getPostState(Fifo(p1, p2), Channel, id, a, outputs, M) = M [owise] .
  ceq computeActions(Fifo(p1, p2), Channel, M) = (Fifo(p1, p2) , (take ;
      p1), 1) if M[k("state")] == nd(0) .
  ceq computeActions(Fifo(p, p'), Channel, M) = (Fifo(p, p'), (put(M[k("
      data")]); p'), 1) if M[k("state")] == nd(1) .

  eq internalUpdate(Fifo(p1, p2), Channel, M) = M .
endfm
```

A `MERGER` is a ternary connector, that acts, for each of its port input, as a synchronous channel with its output port. Moreover, the `MERGER` relates its two input ports with a relation of exclusion, i.e., the two input ports cannot fire at the same time. A `MERGER` with the list of ports `P(1), P(2), P(3)` has two actions: a forwarding action from port `P(1)` to port `P(3)`, or a forwarding action from port `P(2)` to port `P(3)`. The two actions are exclusive, as they cannot occur at the same time. Moreover, the merger always enables both actions, which raises some non-determinism at the system level (i.e., to chose which of the two actions is selected). Similarly to the `SYNC` channel, a `MERGER` agent instantiates the value for the `put` action at runtime, once the result of the `take` action is known. We use the `?` symbol to denote a symbolic value.

```
fmod MERGER is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
  eq getOutput(Merger(p1, p2, p3), Channel, id', a, M) = empty .

  ceq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), take,
       (id, k("data") |-> d), M) =
    insert(k("data"), d, M) if M[k("data")] == nodata .
  eq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), take,
     inputs, M) = notAllowed(take) [owise] .
  ceq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), put(d
     ), outputs, M) =
    insert(k("data"), nodata, M) if M =/= notAllowed(take) .
  eq getPostState(Merger(p1, p2, p3), Channel, id, a, outputs, M) = M [
     owise] .

  eq computeActions(Merger(p1, p2, p3) , Channel, M ) =
    (((Merger(p1, p2, p3), (take; p2)), (Merger(p1, p2, p3), (put(?);  p3
        ))), 1) + (((Merger(p1, p2, p3), (take; p1)), (Merger(p1, p2, p3)
        , (put(?);  p3))), 1) .

  eq internalUpdate(Merger(p1, p2, p3), Channel, M) = M .

 ceq resolve(put(?), Merger(p1, p2, p3), Channel, M) = put(d)
   if d := M[k("data")] .
endfm
```

A PROD and CONS agent respectively implement a producer and consumer. Both agents have a single port, on which they always perform, respectively, a put and a take action. The PROD agent puts natural numbers as values on its port, and increments the value if the put action succeeds. We use such canonical sequences of increasing natural numbers to verify some firing properties of a Reo circuit. When a put action succeeds for a PROD agent, its state is updated to contain the message that has been sent in the k("sent") field. Similarly, when a take action succeeds for a CONS agent, its state is updated to contain the message that has been received in the k("recv") field.

```
fmod PROD is
  inc AGENT{ASemiring} .
  inc PROD-INTERFACE .
  inc PORT-INTERFACE .

  ceq getPostState(id , Producer , id', put(d) , actionoutput, M) =
     insert(k("data"), nd(s(i)), insert(k("sent"), d, M)) if nd(i) := M[
     k("data")] .
  eq getPostState(id , Producer , id', a , actionoutput, M) = M [owise] .
  eq getOutput(id , Producer , id', a, M) = empty .

  eq computeActions(Prod(id) , Producer, M ) =
    (Prod(id), (put(M[k("data")]); id), 1) .

  eq internalUpdate(id, Producer, M) = insert(k("sent"), nodata, M) .
endfm

fmod CONS is
  inc AGENT{ASemiring} .
  inc CONS-INTERFACE .
  inc PORT-INTERFACE .

  eq getPostState(Cons(id) , Consumer , Cons(id), take, (id, k("data")
      |-> d), M) = insert(k("recv"), d, M) .
  eq getPostState(id , Consumer , id', a , actionoutput, M) = M [owise] .

  eq getOutput(id , Consumer , id', a, M) = empty .

  eq computeActions(Cons(id) , Consumer, M ) =
    (Cons(id), (take ; id), 1) .

  eq internalUpdate(id, Consumer, M) = M .

endfm
```

5.2.2 Execution and analysis

We present in `SCENARIO` two system terms for which we run some analysis. For the first scenario, the `PROD` and `CONS` are communicating through a sync channel. Therefore, the only possible composite action is a *clique* in which the `PROD` agents puts a value on port `P(1.0)`, that is forwarded to port `P(2.0)` by the `SYNC` agent, and then consumed by the `CONS` agent. The second scenario is similar, and models the `PROD` and `CONS` agents communicating through a fifo channel.

Note that the `SCENARIO` module instantiates the `linearization` operation as follows: a `take` action on a port `P(i)` happens after a `put` action a port `P(j)` if $i \geq j$; and a `take` action on a port `P(i)` happens after a `take` action on a port `P(j)` if $i > j$.

Listing 5.5: Scenarios for Producer/Consumer protocols.

```
mod SCENARIO is
  inc PORT .
  inc PROD . inc CONS .
  inc SYNC . inc FIFO . inc MERGER .
  inc RUN . inc CONVERSION .

  op init : Nat -> Global .

  eq init(1) = [
    [P(1.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [P(2.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [Prod(P(1.0)) : Producer |
        k("data") |-> nd(1) ; false ; null]
    [Cons(P(2.0)) : Consumer |
        k("data") |-> nodata ; false ; null]
    [Sync(P(1.0), P(2.0)) : Channel |
        k("data") |-> nodata ; false ; null]] .

  eq init(2) = [
    [P(1.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [P(2.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [Prod(P(1.0)) : Producer |
        k("data") |-> nd(1) ; false ; null]
    [Cons(P(2.0)) : Consumer |
        k("recv") |-> nodata ; false ; null]
    [Fifo(P(1.0), P(2.0)) : Channel | k("state") |-> nd(0),
        k("data") |-> nodata ; false ; null]] .

 ceq linearization(aSet) = a linearization(aSet')
   if a , aSet' := aSet .
 ceq (id, (take ; P(i))) (id', (put(d) ; P(j)))  aSeq'  =
   (id', (put(d); P(j))) (id, (take; P(i))) aSeq' if i >= j .
 ceq (id1, (take ; P(i))) (id2, (take ; P(j)))  aSeq' =
   (id2, (take ; P(j))) (id1, (take ; P(i))) aSeq' if i > j .
endm
```

We run the two following queries on the two initial terms of `SCENARIO`. The first query returns the first solution for which the same datum has passed from the producer to the consumer. The solution shows the synchronicity of the `SYNC` channel.

The second query returns the first solution for which the data received by the consumer has an offset of 1 with the data sent by the producer. The solution shows the asynchronicity of the `FIFO` channel.

```
search [1] init(1) =>* [sys::Sys
```

```
 [Prod(P(1.0)) : Producer | M1::MapKD,
     k("sent") |-> nd(i::Nat); false; null]
 [Cons(P(2.0)) : Consumer | M2::MapKD,
     k("recv") |-> nd(j::Nat); false; null]
 ] such that j::Nat == i::Nat .

Solution 1 (state 8) states: 9  rewrites: 1472 in 3ms cpu (0ms real)
    (443774 rewrites/second)
...
i::Nat --> 1
j::Nat --> 1
==========================================
search [1] init(2) =>* [sys::Sys
[Prod(P(1.0)) : Producer | M1::MapKD,
    k("sent") |-> nd(i::Nat); false; null]
[Cons(P(2.0)) : Consumer | M2::MapKD,
    k("recv") |-> nd(j::Nat); false; null]
] such that j::Nat =/= i::Nat .

Solution 1 (state 24) states: 25  rewrites: 6390 in 0ms cpu (2ms real) (~
     rewrites/second)
...
i::Nat --> 2
j::Nat --> 1
```

5.3 Valve-controller

5.3.1 N-reservoir problem

Consider n reservoirs, filled with water. Each reservoir connects at the top to a valve that, if switched on, inputs some water. Each reservoir has a hole at the bottom from where water goes out. Each reservoir has a sensor measuring its water level. The measures are accessible by a digital controller. A valve is placed at the top of the n reservoirs, and fills, continuously, one of the reservoir at a time. The valve moves from one reservoir to another after reception of a switch command from the controller.

The problem is to design a controller such that none of the reservoirs is observed with its water level outside of given bounds, noted l_1^k for the lower and l_2^k for the higher bounds, for the k-th reservoir. We first introduce some intuitive physical explanations about the dynamics of the reservoirs, then specify the digital controller that interfaces the physical components, and finally analyse one instance of the composite cyber-physical system.

Physical part In the n-reservoirs problem, the physical part is described by the collection of n reservoirs, and a valve. The water level of a reservoir follows a continuous evolution: at each time $t \in \mathbb{R}_+$, there exists a value $x(t)$ that corresponds to the water level of the reservoir at t.

Along this part, we use $t \in \mathbb{R}_+$ for continuous time, $n \in \mathbb{N}$ for the number of reservoirs, and $k \in \{1, ..., n\}$ to denote the k-th reservoir.

Reservoir The reservoir of radius r captures, as a component, the evolution of the water level over time. The height of the water at time t in the k-th reservoir, written $x_k(t)$, follows the law $x_k(t) = x_k^{t_0} + (\int_{t_0}^{t} i_k(u) - o_k(u)\, du)/(\pi r^2)$, where $x_k^{t_0} \in \mathbb{R}$ is the initial level of water at $t_0 \in \mathbb{R}_+$, $i_k(t)$ is the rate of incoming water and $o_k(t)$ is the rate of outgoing water for $t \geq t_0$. Note that we use the relation $V = \pi r^2 L$ where V is the volume of a cylindrical reservoir of radius r and height L. The functions x_k, o_k and i_k are functions from $\mathbb{R}_+$ to $\mathbb{R}_+$.

The definition of $x_k(t)$ involves elements internal to the physical description of the reservoir (e.g. $o_k(t)$, the rate of water going out), and external elements (e.g. $i_k(t)$, the rate of water coming in). In particular, the function $i_k(t)$ depends on the valve component's specification.

The component for the reservoir of radius r is the pair $Res(r, i) = (E, L)$ where

$$E = \{read(l), inFlow(k) \mid l, k \in [0, 20]\}$$

and $L \subseteq TES(E)$ with $\sigma \in L$ implies there exists a function $f : \mathbb{R}_+ \to \mathbb{R}_+$ with

- $f(0) = 15$ and $f' = -outFlow$, where *outFlow* is a constant that models how fast water goes out of the reservoir;
- $\sigma(t_i) = \{read(l)\}$ implies $l = \max(f(t_i), 0)$;
- $\sigma(t_i) = \{inFlow(k)\}$ implies $f(t) = f(t_i) + (t - t_i) \times (k - outFlow)/(r^2\pi)$ for all $t \in]t_i, t_{i+1}]$, where k is the rate of water going in the reservoir in the interval $]t_i, t_{i+1}]$.

Valve The valve fixes the rate at which the water flows into the reservoir, and moves its arm from one reservoir to another. The state of the valve is described by a value $s \in \{0, ..., n\}$ where $s = i$ encodes the fact that the valve is placed above reservoir i only. Under an input signal *switch(k)*, the valve changes state from $s = i$ to $s = k$.

The component for the valve is the pair $\mathit{Valve}(n) = (E(n), L)$ where

$$E(n) = \{\mathit{switch}(k), \mathit{outFlow}(k, j) \mid k \in \{1, ..., n\}, j \in [0, 20]\}$$

and $L \subseteq \mathit{TES}(E(n))$ with $\sigma \in L$ such that every switch event is simultaneous with 0 outFlow in the reservoir that are not recipient of the valve, i.e., for all t, if $\mathit{switch}(k) \in \sigma(t)$ then $\mathit{outFlow}(i, 0) \in \sigma(t)$ for $i \in \{1, ..., n\} \setminus \{k\}$ and $\mathit{outFlow}(i, j) \in \sigma(t)$ with $j \neq 0$.

Continuous formulation of the problem The problem can be formulated in the continuous setting as finding a sequence of states for the valve (or timed input signals $\delta(t)$, equivalently) such that

$$\forall k \in \{1, .., n\}.\ l_1^k \leq x_k(t) \leq l_2^k$$

where l_1^k and l_2^k are respectively lower and upper bounds for the water level x_k of reservoir k.

The function δ is not built as a physical component, but results from the measures and the actions of a digital component on the physical system. We show in the next subsection an instance of a controller, and later provide an instance of the physical components (reservoir and valve) in order to analyse the resulting interaction with a controller.

Cyber part The cyber component consists of a controller, reading from the reservoirs' sensors, and acting on the valve. In sequence, this subsection details the measurement device that interface the reservoir and the controller, the action that the controller performs on the valve, and the controller's state transition system.

The problem of decision inherent in the n-reservoirs system is dependent on the water level measured by the controller. If the agent can read precisely the level of water in each of the reservoir, at a high frequency, the reactivity of the system will be higher than the case where the agent measures the reservoir's state at low frequency, with less accuracy. From a formal point of view, increasing the sensor precision also increases the possible states in which the system can be observed, and therefore the complexity of the representation. The trade off is to find a reading frequency that discriminates the minimal amount of states, while ensuring the agent to be reactive to achieve its goal.

As a result of an action, the dynamic of the physical system may change, and lead to different sequences of measures from the controller.

Controller The controller has two types of events: *read* and *switch*. The *read(i,l)* event is parametrized by the reservoir i that the controller reads, and displays the value l that is read at the sensor. The *switch(i)* event synchronizes with the same event of the valve and takes as parameter the value of the reservoir to which the valve should switch.

A controller is a component defined as the pair $C(T) = (E, L(T))$ with

$$E = \{read(i,l), switch(i) \mid i \in \{1,...,n\}, l \in [0,20]\}$$

and $L(T) \subseteq TES(E)$ is such that $\sigma \in L(T)$ implies there exists a function $f : \mathbb{R}_+ \to \{1,...,n\} \to [0,20]$ with

- observations occur at multiple of T, i.e, $\sigma(t) \neq \emptyset$ implies $t = 0 \mod T$;
- $read(i,l) \in \sigma(t)$ implies $f(t)(i) = l$

The controller may change state after performing some measurements or moving the valve, such as recording the sensor values or the reservoir to which the valve has switched. In the next subsection, we give an executable specification for the controller, the valve, and the reservoir. We analyse the strategy of the controller to keep the water level of the reservoirs within a margin.

5.3.2 Execution and analysis

We present the three main Maude modules, namely the `CONTROLLER`, the `VALVE`, and the `RESERVOIR`, to model and analyse properties of the N-reservoirs problem.

Controller The `CONTROLLER` agent has two main actions: `read` and `switch`. The `read` action returns the value of the water level in all reservoirs. The `CONTROLLER` agent stores the value in its state, and implements the function `lowestLevel` to return the reservoir with the lowest water level (ordered with the reservoir identifier, in case of two reservoirs having the smallest water level). The `switch` action takes a reservoir's identifier as argument and acts on the `VALVE` agent by switching the valve to that reservoir.

Listing 5.6: A module for the `CONTROLLER` agent.

```
fmod CONTROLLER is
    inc SET{Nat} .
    inc TRACE-INTERFACE .
    inc RESERVOIR-INTERFACE .
    inc CONTROLLER-INTERFACE .
    inc AGENT{ASemiring} .
    ...
    eq getPostState(id, Controller , id, read, output, M) =
        getSensorValues(getResources(id, read), output, M) .
    eq getPostState(id, Controller, id', switch(r), actionoutput, M) =
        insert(k("state"), d(r), M) .
    eq getPostState(id, Controller, id', a, actionoutput, M) = M [owise]
        .

    eq computeActions(id , Controller , M) =  null [owise] .

    *** Upper and lower bound for the reservoir level
    ceq computeActions(id, Controller, M) = (id, (switch(r)  ;
        getResources(id , switch(r)) ) , 1) if
         M =/= empty /\ r := lowestLevel(M) /\
         d(r) =/= M[k("state")] /\ fd(j) := M[k("lev", r)] /\
         j <= lowbound(r) .
    eq computeActions(id, Controller, M) = (id, (read ; getResources(id,
        read)),  1) [owise] .

    eq internalUpdate(id, Controller, M) = M .
endfm
```

Valve The `VALVE` agent has a `fill` action for each reservoir. The `fill` action takes a flow value as argument, and changes the inflow rate in the `RESERVOIR` agent. As a consequence, the state of the `RESERVOIR` will change its dynamic and the `read` action from the `CONTROLLER` on the `RESERVOIR` will get updates accordingly. The `fill` action of the `VALVE` is composite of two other actions that turn `off` the inflow of the other `RESERVOIR` agents. In Listing 5.7, we show one of such composite action that fills reservoir 1 and turns off the two other reservoirs.

Listing 5.7: A module for the `VALVE` agent.

```
fmod VALVE is
  inc AGENT{ASemiring} .
  inc VALVE-INTERFACE .
  inc RESERVOIR-INTERFACE .
  inc TIME-INTERFACE .
  inc CONTROLLER-INTERFACE .
  ...
  *** Passive agent:
  ceq computeActions(id, Valve, M ) = (((valve, (fill(k) ; res(1))), (
      valve, (off ; res(2))), (valve, (off ; res(3)))), 1) if nd(1) := M[
      k("state")] /\ fd(k) := M[k("inFlow")] .
  ...
  ceq getPostState(r, Valve, id, switch(res(i)), output, M) =  M' if M'
      :=  insert( k("state") , nd(i) , M) .
  eq getPostState(r, Valve, id, switch(r), output, M) = notAllowed(switch
      (r)) [owise] .
  eq getPostState(r, Valve, id, an, output, M) =  M [owise] .

  eq internalUpdate(id, Valve, M) = M .
endfm
```

Reservoir For simplicity, we set the radius of the `RESERVOIR` to be equal to $\sqrt{\pi}$, and the inflow and outflow corresponds to the change of level in the reservoir. The `RESERVOIR` agent has no actions, but reacts to the `fill` and `off` actions of the `VALVE` agent, and the `read` action of the `CONTROLLER` agent. The `read` action generates an output that contains the current level of the reservoir. The `fill` action changes the `vin` state variable to take the value given by the `VALVE` agent. The `off` action sets the `vin` state variable to 0.0. A `RESERVOIR` agent defines a function `f` that computes the water level given the current level, the inflow and outflow models by `vin` and `vout` respectively. At the end of every round, the `RESERVOIR` updates its state with the value computed by `f`.

Listing 5.8: A module for the `RESERVOIR` agent.

```
fmod RESERVOIR is
  inc AGENT{ASemiring} .
  inc RESERVOIR-INTERFACE .
  inc TIME-INTERFACE .
  inc CONTROLLER-INTERFACE .
  inc VALVE-INTERFACE .
  ...
  *** Passive agent:
  eq computeActions(id , Reservoir, M ) = null .
  *** Compute the level of the reservoir
  op f : Float Float Float -> Float .
  eq f(cl, vin, vout) = max(min(cl + (vin - vout) , 20.0), 0.0) .

  eq getOutput(id, Reservoir, id', read, M) = k("lev") |-> M[k("lev")] .
  eq getPostState(id, Reservoir, id', read, output, M) = M .
  ceq getPostState(id, Reservoir, id', end, output, M) = insert(k("lev"),
       fd(f(cl, vin, vout)), M)
      if  k("inFlow") |-> fd(vin)  , k("outFlow") |-> fd(vout), k("lev")
          |-> fd(cl), M' := M .
  eq getPostState(res(j), Reservoir, id', fill(vin), output, M) = insert(
      k("inFlow"), fd(vin), M)  .
  eq getPostState(id, Reservoir, id', off, output, M) =  insert(k("inFlow
      "), fd(0.0), M)  .
  eq getPostState( r, Reservoir, id , an , output, M) = M [owise] .

  eq internalUpdate(id, Reservoir, M) = M .
endfm
```

Scenarios We present in Listing 5.9 a scenario for which a controller periodically reads the value of the water level in the reservoirs, and switches the valve accordingly. We set the maximum capacity for each reservoir to be 20.0, the outflow rate to be 2.0 and the inflow rate from the valve to be 5.0. Initially, the valve is above `res(2)`.

We also define the `linearization` operation to order actions as follows. The `read` action occurs first in the sequence, followed by the `switch` action, and the other actions occur freely. The value taken by `read` action therefore takes the value of the water level at the end of the previous round, i.e., after update of each `RESERVOIR` agent.

Listing 5.9: A module for the `SCENARIO` agent.

```
mod SCENARIO is
  inc RESERVOIR . inc VALVE .
  inc CONTROLLER . inc RUN .
  ...
  *** Resources for controller agent.
  eq getResources(id , read) = res(1), res(2), res(3) .
  eq getResources(id , switch(id')) = valve .
  eq getResources(id , off) = res(1), res(2), res(3) .

  eq lowbound(res(i)) = 15.0 .  eq upbound(res(i)) = 18.0 .

  eq init = [[res(1) : Reservoir | k("lev") |-> fd(20.0),  k("state") |->
       nd(0), k("inFlow") |-> fd(0.0) , k("outFlow") |-> fd(2.0) ; false
      ; null ]
   [res(2) : Reservoir | k("lev") |-> fd(20.0),  k("state") |-> nd(1), k(
       "inFlow") |-> fd(5.0) , k("outFlow") |-> fd(2.0) ; false ; null ]
   [res(3) : Reservoir | k("lev") |-> fd(20.0),  k("state") |-> nd(0), k(
       "inFlow") |-> fd(0.0) , k("outFlow") |-> fd(2.0) ; false ; null ]
   [valve : Valve | k("state") |-> nd(2), k("inFlow") |-> fd(5.0)  ;
       false ; null ]
   [controller : Controller | k("state") |-> d(nil) ; false ; null ]] .
  ...
  ceq linearization(aSet) = a linearization(aSet')
   if a , aSet' := aSet .
  eq a (id, (read ; res )) = (id, (read ; res)) a  .
  ceq a (id, (switch(id'); res)) = (id, (switch(id'); res)) a
   if (id', (an ; res' )) := a  /\ an =/= read .
endm
```

Search queries We run two queries on the scenario of Listing 5.9. The first query searches for a state for which all the controller has read a value for the water level which is below 8.0. We can see that one of such solution exists as the result of the query.

The second query searches for a state for which the controller read one of the reservoir's water level to be 0.0. As shown in the output, a solution exists, and `res(2)` may reach a water level of 0.0.

Listing 5.10: Two search queries for some safety properties.

```
search [1] in SCENARIO : init =>* [sys:Sys
[controller : Controller | M::MapKD, k("lev",res(1)) |-> fd(j::Float), k(
    "lev",res(2)) |-> fd(i::Float), k("lev",res(3)) |-> fd(k::Float) ;
    false ; null]
] such that (j::Float < 8.0 and k::Float < 8.0) and i::Float < 8.0 = true
    .

Solution 1 (state 180)
states: 181  rewrites: 27201 in 13ms cpu (12ms real) (2041963 rewrites/
    second)
...
j::Float --> 3.0 i::Float --> 7.0 k::Float --> 7.0


search [1] init =>*
[ sys:Sys
    [controller : Controller |  k("lev", res(1)) |-> fd(j::Float), k("lev
        ", res(2)) |-> fd(i::Float), k("lev", res(3)) |-> fd(k::Float), M
        ::MapKD ; false ; null]
]  such that i::Float == 0.0 or j::Float == 0.0 or k::Float == 0.0  .

Solution 1 (state 96)
```

```
states: 97  rewrites: 14927 in 6ms cpu (6ms real) (2240618 rewrites/
    second)
...
j::Float --> 1.4e+1 i::Float --> 0.0 k::Float --> 9.0
```

However, after changing the rate of the valve from 5.0 to 7.0, the same queries as in Listing 5.10 return no solutions. As a consequence, the strategy implemented in the `CONTROLLER` agent therefore successfully keeps the water level above 8.0 for all reservoirs, and prevent any reservoir to reach 0.0.

5.4 Robot-Battery-Field system

We propose to study three properties:

1. Safety property: In the first scenario, we model two `TROLL` agents, moving on a shared `FIELD`, with private `BATTERY` agents; we study the cases for which the two robots can exchange their position without running out of energy.

2. Liveness property: In the second scenario, the field is equipped with a station, where the `TROLL` agent can recharge its battery. We want to prevent the agent from running out of energy while oscillating between the two locations, i.e., if the station can always supply energy, we want a sequence of actions such that the agent never runs out of energy.

3. Self-sorting property: In the third scenario, we place three `TROLL` agents on a grid, each with a unique natural number identifier. We study some self-sorting property of the system by global coordination (e.g., use of an external protocol) or local strategies (e.g., ranking of each agent's actions).

We consider the battery, robot, and grid components introduced in Section 1.3 and formalized in Section 2.1.6 and Section 2.2.3 as the expression

$$Sys(n, T_1, ..., T_n) = \bowtie_{i \in \{1,...,n\}} (R(i, T_i) \times_{\Sigma_{R_i B_i}} B_i) \times_{\Sigma_{RF}} G_\mu(\{1, ..., n\}, n, 2)$$

made of n robots $R(i, T_i)$, each interacting with a private battery B_i under the interaction signatures $\Sigma_{R_i B_i}$, and in product with a grid G under the interaction signature Σ_{RG}. We use $\bowtie$ for the product with the free interaction signature (i.e., every pair of TESs is composable), and the notation $\bowtie_{i \in \{1,...,n\}} \{C_i\}$ for $C_1 \bowtie ... \bowtie C_n$ as $\bowtie$ is commutative and associative.

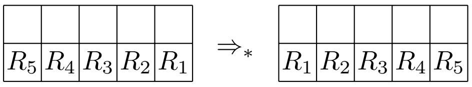

Figure 5.1: Initial state of the unsorted robots (left), and final state of the sorted robots (right).

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	-
t_2	W	-	-	-	-
t_3	$(3;1)$	-	-	-	-
...	...	...	...	...	...

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	-
t_2	W	E	N	-	-
t_3	S	$(4;0)$	E	-	-
...	...	...	...	...	...

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	N
t_2	W	E	-	W	E
t_3	S	-	-	-	S
...	...	...	...	...	...

Table 5.1: Each table displays the three first observables at times t_1, t_2, and t_3 for three TESs in the behavior of the product of components R_1, R_2, R_3, R_4, and R_5 on the grid of Figure 5.1. We omit the subscript and use the column to identify the events. The symbol - represents the absence of observation in the TES.

We fix $n = 5$ and the same period T for each robots. We write E for the set of events of the composite system $Sys(2, T)$. We also reuse the grid component introduced in Section 2.2.3 where μ is the initial position of the robots on the grid, and the parameters n and 2 a refers to the x and y length of the grid. For simplicity, use R_i to denote the composite component $(R(i, T) \times_{\Sigma_{R_i B_i}} B_i)$ with fixed period T.

Self-sorting robots Figure 5.1 shows five robot instances, each of which has a unique and distinct natural number assigned, positioned at an initial location on a grid. The goal of the robots in this example is to move around on the grid such that they end up in a final state where they line-up in the sorted order according to their assigned numbers.

Three first observations for three behaviors are displayed in Table 5.1. Each behavior exposes different degrees of concurrency, where in the left behavior, only robot R_1 moves, while in the middle behavior, robots R_1 and R_2 swap their positions, and in the right behavior both R_1 and R_4 swap their positions with R_2 and R_5, respectively.

We consider the following property: *eventually, the position of each robot R_i is* $(i, 0)_{R_i}$, i.e., every robot successfully reaches its place. This property is a trace property, which we call P_{sorted} and consists of every behavior $\sigma \in TES(E)$ such that there

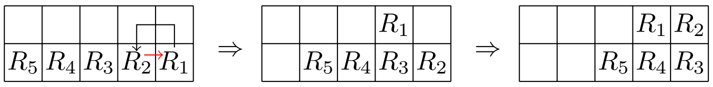

Figure 5.2: Initial state of the unsorted robot (left) leading to a possible deadlock (right) if each robot follows its strategy.

exists an $n \in \mathbb{N}$ with $\sigma(n) = (O_n, t_n)$ and $(i, 0)_{R_i} \in O_n$ for all robots R_i. As shown in Table 5.1, the set of behaviors for the product of robots is large, and the property P_{sorted} does not (necessarily) hold *a priori*: there exists a composite behavior τ for the component $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie F(\{1, 2, 3, 4, 5\})$ such that $\tau \notin P_{sorted}$.

Robots may beforehand decide on some strategies to swap and move on the grid such that their composition satisfies the property P_{sorted}. For instance, consider the following strategy for each robot R_n:

- *swapping*: if the last read (x, y) of its location is such that $x < n$, then move North, then West, then South.
- *pursuing*: otherwise, move East.

Remember that the grid prevents two robots from moving to the same cell, which is therefore removed from the observable behavior. We emphasize that some sequences of moves for each robot may deadlock, and therefore are not part of the component behavior of the system of robots, but may occur operationally by taking a composable action step by step (see Section 4.1.2). Consider Figure 5.2, for which each robot follows its internal strategy. Because of non-determinism introduced by the timing of each observations, one may consider the following sequence of observations: first, R_1 move North, then West; in the meantime, R_2 moves West, followed by R_3, R_4, and R_5. By a similar sequence of moves, the set of robots ends in the configuration on the right of Figure 5.2. In this position and for each robot, the next move dictated by its internal strategy is disallowed, which corresponds to a *deadlock*. While behaviors do not contain finite sequences of observations, which makes the scenario of Figure 5.2 not expressible as a TES, such scenario may occur in practice. We give in next Section some analysis to prevent such behavior to happen.

Alternatively, the collection of robots may be coordinated by an external protocol that guides their moves. Besides considering the robot and the grid components, we add a third kind of component that acts as a coordinator. In other words, we make the protocol used by robots to interact explicit and external to them and the grid; i.e., we assume exogenous coordination. Exogenous coordination allows robots to decide a priori on some strategies to swap and move on the grid, in which case their external coordinator component merely unconditionally facilitates their interactions. Alternatively, the external coordinator component may implement a protocol that guides the moves of a set of clueless robots into their destined final locations. The most intuitive of such coordinator is the property itself as a component. Indeed, let $C_{sorted} = (E, L)$ be such that $E = \bigcup_{i \in I} E_{R_i}$ with $I = \{1, 2, 3, 4, 5\}$ and $L = P_{sorted}$. Then, and as shown in [62], the coordinated component $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie G(\{1, 2, 3, 4, 5\}, 5, 2) \bowtie C_{sorted}$ trivially satisfies the property P_{sorted}. While easily specified, such coordination component is non-deterministic and not easily implementable. We provide an example of a deterministic coordinators.

As discussed, we want to implement the property P_{sorted} as a collection of small coordinators that swap the position of unsorted robots. Intuitively, this protocol mimics the behavior of bubble sort, but for physical devices. Given two robot identifiers R_1 and R_2, we introduce the swap component $S(R_1, R_2)$ that coordinates the two robots R_1 and R_2 to swap their positions. Its interface $E_S(R_1, R_2)$ contains the following events:

- start(S(R_1,R_2)) and end(S(R_1, R_2)) that respectively notify the beginning and the end of an interaction with R_1 and R_2. Those events are observed when the swap protocol is starting or ending an interaction with either R_1 or with R_2.
- $(x, y)_{R_1}$ and $(x, y)_{R_2}$ that occur when the protocol reads, respectively, the position of robot R_1 and robot R_2,
- d_{R_1} and d_{R_2} for all $d \in \{N, W, E, S\}$ that occur when the robots R_1 and R_2 move;
- locked(S(R_1,R_2)) and unlocked(S(R_1,R_2)) that occur, respectively, when another protocol begin and end an interaction with either R_1 and R_2.

The behavior of a swapping protocol $S(R_1, R_2)$ is such that, it starts its protocol sequence by an observable start(S(R_1, R_2)), then it moves R_1 North, then R_2 East, then R_1 West and South. The protocol starts the sequence only if it reads a position for R_1 and R_2 such that R_1 is on the cell next to R_2 on the x-axis. Once the sequence of moves is complete, the protocol outputs the observable end(S(R_1, R_2)). If the protocol is not swapping two robots, or is not locked, then robots can freely read their positions.

Swapping protocols interact with each others by locking other protocols that share the same robot identifiers. Therefore, if S(R_1,R_2) starts its protocol sequence, then

S(R_2, R_i) synchronizes with a locked event locked(S(R_2,R_i)), for $2 < i$. Then, R_2 cannot swap with other robots unless S(R_1,R_2) completes its sequence, in which case end(S(R_1, R_2)) synchronizes with unlocked(S(R_2,R_i)) for $2 < i$. We extend the underlying composability relation κ on observations such that, for $i < j$, simultaneous observations (O_1, t) and (O_2, t) are composable, i.e., $((O_1, t), (O_2, t)) \in \kappa$, if:

$$\begin{aligned}\text{start}(\text{S}(R_i,R_j)) \in O_1 \implies & \exists k.k < i.\text{locked}(\text{S}(R_k,\ R_i)) \in O_2 \vee \\ & \exists k.j < k.\text{locked}(\text{S}(R_j,\ R_k)) \in O_2\end{aligned}$$

and

$$\begin{aligned}\text{end}(\text{S}(R_i,R_j)) \in O_1 \implies & \exists k < i.\text{unlocked}(\text{S}(R_k,\ R_i)) \in O_2 \vee \\ & \exists j < k.\text{unlocked}(\text{S}(R_j,\ R_k)) \in O_2\end{aligned}$$

For each pair of robots R_i, R_j such that $i < j$, we introduce a swapping protocol S(R_i, R_j). As a result, the coordinated system is given by the following composition:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie G(\{1,2,3,4,5\},5,2) \bowtie_{i<j} S(R_i, R_j)$$

Note that the definition of $\bowtie$ imposes that, if one protocol starts its sequence, then all protocols that share some robot identifiers synchronize with a lock event. Similar behavior occurs at the end of the sequence.

5.4.1 Execution and analysis

Main agents To illustrate the use of our framework to simulate and verify cyber-physical systems, we present an agent specification for three components: a `FIELD`, a `TROLL`, and a `BATTERY`. A `FIELD` component interacts with the `TROLL` component by reacting to its move action, and its sensor reading. As shown in Listing 5.11 the `FIELD` agent has no actions, but reacts to the move action of the `TROLL` agent by updating its state and changing the agent's location. Currently, the update is discrete, but more sophisticated updates can be defined (e.g., changing the mode of a function recording the trajectory of the `TROLL` agent). In the case where the state of the `FIELD` agent forbids the `TROLL` agent's move, the `FIELD` agent enters in a disallowed state marked as `notAllowed(an)`, with `an` as the action name. The `FIELD` responds to the read sensor action by returning the current location of the `TROLL` agent as an output.

Listing 5.11: Extract from the `FIELD` Maude module.

```
fmod FIELD is
  inc TROLL-INTERFACE .
  inc FIELD-INTERFACE .
  inc PROTOCOL-INTERFACE .
  inc AGENT{ASemiring} .
  ...
  *** Passive agent:
  eq computeActions(id , Field, M) = null .

  eq internalUpdate(id , Field, M) = M .

  ceq getPostState(r, Field, id, a, mtOutput, M) = M'
      if isMove?(a) /\
         k(loc) |-> d(id) , M1' := M /\
         loc' := next(loc, a) /\
         loc' =/= loc /\
         M[k(loc')] == undefined /\
         M' := k(loc') |-> d(id), M1' .

  ceq getPostState(r, Field, id, a, mtOutput, M) = notAllowed(a)
      if isMove?(a) /\ k(loc) |-> d(id) , M1' := M /\
         loc' := next( loc , a ) /\ ((loc' =/= loc and  M[k(loc')] =/=
             undefined) or loc' == loc) .

  ceq getOutput(r, Field, id, readSensors(position sn), M)
                = ( k("pos") |-> loc , M'  )
      if  k(loc) |-> d(id) , M1' := M /\
          M' := ( k("obstacles") |-> obstacle( 1, id, loc, M))   .
endfm
```

A TROLL agent reacts to no other agent actions, and therefore does not include any agent interface. A TROLL agent uses the function `getSoftActions` to return a ranked set of actions given its state, and implements the `computeActions` operation by returning the ranked set of actions. The operation `getSoftAction` implements a strategy for the agent which, for instance, ranks higher the move action that moves the robot closer to its target location. The expression may contain more than one action, with different weights. The weights of the action may depend on the internal goal that the agent set to itself, as for instance reaching a location on the field. The TROLL agent specifies how it reacts to, e.g., the sensor value input from the field, by updating the corresponding key in its state with `getSensorValues`.

Listing 5.12: Extract from the TROLL Maude module.

```
fmod TROLL is
```

```
  inc AGENT{ASemiring} .
  inc LOCATION .
  inc TROLL-INTERFACE .

  eq computeActions(id , Troll , M ) = getSoftActions(id, M ,
      trollActions(id, M)) .

  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(1), M) if M[k("
      read")] == nd(0) .
  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(0), M) if M[k("
      read")] == nd(1) .

  ceq getPostState(id, Troll, id, readSensors(sn), sensorvalues, M) = M'
      if M' := getSensorValues(getResources(id, readSensors(sn)) ,
          sensorvalues), k("goal") |-> M[k("goal")], k("read") |-> nd(1)
           .
endfm
```

A BATTERY agent does not act on any other agent, as the FIELD, but reacts to the TROLL agent actions. Each move action triggers in the BATTERY agent a change of state that decreases its energy level. As well, each charge action changes the BATTERY agent state to increase its energy level. Similarly to the field, in the case where the state of the battery agent has 0 energy, the battery enters a disallowed state marked as notAllowed(an), with an as the action name. A sensor reading by the TROLL agent triggers an output from the BATTERY agent with the current energy level.

Listing 5.13: Extract from the BATTERY Maude module.

```
fmod BATTERY is
 inc AGENT{ASemiring} .
 inc BATTERY-INTERFACE .
 inc TROLL-INTERFACE .


 *** Passive agent:
 eq computeActions(id, Battery, M ) = null .
 eq internalUpdate(id, Battery, M ) = M .


 ceq getOutput(r, Battery, id, readSensors(energy sn), M)
         =  k("bat") |-> M[k("bat")]
     if r := getBattery(id) .

 *** Next state.
 ceq getPostState(r, Battery, id, an, mtOutput, M) =  M'
```

```
     if isMove?(an) /\
        k("bat") |-> nd(s i) , M1' := M /\
        M' := insert( k("bat") , nd(i) , M) .

 ceq getPostState(r, Battery, id, charge(j), mtOutput, M)  = M1
     if nd(i) := M[k("bat")] /\
         i  < capacity  /\
         M1 := insert( k("bat") , nd(min ( i + j, capacity)) , M ) .

 ceq getPostState(r, Battery, id, an, mtOutput, M) = notAllowed(an)
      if isMove?(an) /\ M[k("bat")] == nd(0) .

endfm
```

A `PROTOCOL` agent `swap(id1,id2)` acts on the `TROLL` agents `id1` and `id2`, and is used as a resource by the two `TROLL` agent move actions. A `PROTOCOL` internally has a finite state machine `T(id):Fsa` that accepts or rejects a sequence of actions. Each `move` action of a `TROLL` is accepted only if there is a transition in the `PROTOCOL` agent state transition system. A `PROTOCOL` agent `swap(id1, id2)` always tries to swap agents with ids `id1` and `id2`. Thus, if `id2` is on the direct East position of `id1` on the field, then action `start` succeeds, and the protocol enters in the sequence `move(N)` for `id2`, `move(W)` for `id2`, `move(E)` for `id1`, and then `move(S)` for `id2`. Eventually the sequence ends with `finish` action. The `PROTOCOL` agent may also have some transitions labeled with a set of actions, one for each of the agent `id1` and `id2`. In which case, the transition succeeds if the clique contains, for each agent involved in the protocol, an action that is composable with the action labeling the protocol transition. We use the `end` action to mark the end of the sequence of actions forming a clique. The `PROTOCOL` may reject such `end` action if the clique does not cover the set of actions labeling the transition, which therefore discard the set of actions as not composable.

Listing 5.14: Extract from the `SWAP` protocol Maude module.

```
fmod SWAP is
    inc AGENT{ASemiring} .
    inc TROLL-INTERFACE .
    inc PROCESS-INTERFACE .
    inc FIELD-INTERFACE .
    inc PROTOCOL-INTERFACE .

    op T : Identifier -> Fsa .
    *** Update of state from external move or its own swapping actions
    ceq getPostState(id, Protocol, id', move(d), sysState, M ) = M'
        if  {q(i)} := getState(M) /\
            M' :=  insert( k("recv") , recv(union(getLabel(M), {l(id',
                move(d))})) , M) .

    *** Ending transition correctly
    ceq getPostState(id , Protocol , id, end , sysState, M) = M'
        if  state := getState(M) /\
            label := getLabel(M) /\
            tr := getTransitions(T(id)) /\
            (state, label, state'), tr' := tr /\
            M' :=  insert( k("recv") , recv({}), insert( k("state") , ds(
                state') , M)) .

    *** Not allowed states
    eq getPostState(id, Protocol, id', end, sysState, M ) = notAllowed(
        end) [owise] .
    eq getPostState(id, Protocol, id', a, sysState, M) = M [owise] .

    eq getOutput(id, Protocol, id', a, M) = empty .
    ceq computeActions(swap(id, id') , Protocol, M ) = ((swap(id, id') ,
        ( start ; getResources(swap(id, id'), start))), 5)
        if {q(0)} := getState(M) .
    eq computeActions(swap(id, id'), Protocol, M) = null [owise] .
    eq internalUpdate(swap(id, id'), Protocol, M) = M .

endfm
```

Composability relation The `TROLL`, `FIELD`, and `BATTERY` modules specify the state space and transition functions for, respectively, a `TROLL`, `FIELD`, and `BATTERY` agent. A system consisting of a set of instances of such agents would need a composability relation to relate actions from each agent.
More precisely, we give some possible *cliques* of a system consisting of two `TROLL` agents with identifiers `id(0), id(1):TROLL`, one `field:FIELD` agent, and two `BATTERY` agents `bat(0), bat(1):BATTERY`.
The actions of agent `id(0)` compose with outputs of its corresponding battery `bat(0)` and of the shared `field` agent.
For instance, a move action of the `id(0)` agent is of the form `(id(0), (move(d), {bat(0), field}))`, where `d` is a direction for the move, and composes with outputs of the battery and field, both notifying that the move is possible.
Alternatively, a read action of the `id(0)` agent is of the form `(id(0), (read, {bat(0), field}))` and composes with outputs of the battery and field, each giving the battery level and the location of agent `id(0)`.

Safety property: not running of energy We consider a system containing two TROLL agents, with identifiers id(0) and id(1), paired with two BATTERY agents with identifier bat(0) and bat(1), and sharing the same FIELD resource. The goal for each agent is to reach the initial location of the other agent. If both agents follow the shortest path to their goal location, there is an instant for which the two agents need to swap their positions. The crossing can lead to a livelock, where agents move symmetrically until the energy of the batteries runs out.

The initial system term, without the protocol, is given by:

```
eq init = [[id(0): Troll | k("goal") |-> (5 ; 5) ; false ; null]
[bat(0) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[id(1): Troll | k("goal") |-> (0 ; 5) ; false ; null]
[bat(1) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[field : Field | (k(( 0 ; 5 )) |-> d(id(0)) , k(( 5 ; 5 )) |-> d(id(1)))
    ; false ; null]] .
```

The initial system term with the protocol is given by:

```
eq init = [[id(0): Troll | k("goal") |-> (5 ; 5) ; false ; null]
[bat(0) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[id(1): Troll | k("goal") |-> (0 ; 5) ; false ; null]
[bat(1) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[swap(id(0),id(1)) : Protocol | k("state") |-> ds({q(0)}), k("recv") |->
    recv({}) ; false ; null]
[field : Field | (k(( 0 ; 5 )) |-> d(id(0)) , k(( 5 ; 5 )) |-> d(id(1)))
    ; false ; null]] .
```

We analyze in Maude two scenarios. In one, each robot has as strategy to take the shortest path to reach its goal. As a consequence, a robot reads its position, computes the shortest path, and submits a set of optimal actions. A robot can sense an obstacle on its direct next location, which then allows for sub-optimal lateral moves (e.g., if the obstacle is in the direct next position in the West direction, the robot may go either North or South). In the other scenario, we add a protocol that swaps the two robots if robot id(0) is on the direct next location on the west of robot id(1). The swapping is a sequence of moves that ends in an exchange of positions of the two robots.

In the two scenarios, we analyze the behavior of the resulting system with two queries. The first query asks if the system can reach a state in which the energy level of the two batteries is 0, which means that its robot can no longer move:

```
search [1] init =>* [sys::Sys
       [ bat(1) : Battery | k(level) |-> 0 ; true ; null],
       [ bat(2) : Battery | k(level) |-> 0 ; true ; null]] .
```

The second query asks if the system can reach a state in which the two robots successfully reached their goals, and end in the expected locations:

```
search [1] init =>* [sys::Sys  [ field : Field | k(( 5 ; 5 ))
  |-> d(id(0)), k(( 0 ; 5 )) |-> d(id(1)) ; true ; null]] .
```

As a result, when the protocol is absent, the two robots can enter in a livelock behavior and eventually fail with an empty battery:

```
Solution 1 (state 80)
states: 81  rw: 223566 in 73ms cpu (74ms real) (3053554 rw/s)
```

Alternatively, when the protocol is used, the livelock is removed using exogenous coordination. The two robots therefore successfully reach their end locations, and stop before running out of battery:

```
No solution. states: 102
rewrites: 720235 in 146ms cpu (145ms real) (4920041 rw/s)
```

In both cases, the second query succeeds, as there exists a path for both scenarios where the two robots reach their end goal locations. The results can be reproduced by downloading the archive at [1].

Liveness property: patrolling trolls A strategy ranks the action of an agent with respect to some internal measure. For instance, a TROLL agent prefers an action that moves it closer to its goal.

```
fmod STRATEGY is
  inc TROLL-INTERFACE .
  inc AGENT{ASemiring} .
  ...
  *** Valuation of an action based on the state of the agent M, and some
      additional measures (current goal,
  ***  distance to its goal, obstacles on the next cells).
  ceq getValue(id, M, a) = (id, (a; getResources(id, a)), 2)
      if isMove?(a)  /\
         M[k("read")] == nd(1) /\
         M[k("pos")] =/= undefined /\
         closest(a, M) /\
         enabled?(a , M) /\
         M[k("pos")] =/= M[k("goal")] /\
         M[k("charging")] =/= nd(1) .
  ceq getValue(id, M, a) = (id, (a; getResources(id, a)), 1)
      if isMove?(a) /\
         M[k("read")] == nd(1) /\
         M[k("pos")] =/= undefined /\
         not closest(a, M) /\
         a a' := neighbors(a, M) /\
         enabled?(a, M) /\
         M[k("pos")] =/= M[k("goal")] /\
         M[k("charging")] =/= nd(1) .
  ...
endfm
```

We verify the property of liveness for the system, i.e., that the two robots can always eventually swap their positions. Using the Maude search engine, we look for a state for which the two robots have an empty battery level. We find at least one solution for such state:

```
search [1] in SCENARIO : init =>* [sys::Sys
[bat(0) : Battery | k("bat") |-> nd(0) ; true ; null]
[bat(1) : Battery | k("bat") |-> nd(0) ; true ; null]] .

Solution 1 (state 389)
states: 390  rewrites: 1739739 in 299ms cpu (301ms real) (5803637
    rewrites/second)
sys::Sys -->
[field : Field | k(2 ; 3) |-> d(id(0)), k(3 ; 3) |-> d(id(1)) ; true ;
    null]
[id(0) : Troll | k("bat") |-> nd(1), k("goal") |-> 5 ; 5, k("next") |-> 0
     ; 5, k("pos") |-> 2 ; 2, k("read") |-> nd(0)
    ; false ; null]
[id(1) : Troll | k("bat") |-> nd(1), k("goal") |-> 0 ; 5, k("next") |-> 5
     ; 5, k("pos") |-> 3 ; 2, k("read") |-> nd(0)
    ; false ; null]
```

For the liveness property, the `TROLL` changes its goal while it reaches its first objective. Additionally, the `TROLL` agent has an additional `charge` action that, when located on the station, charges its corresponding battery.

```
fmod TROLL-ALT is
  inc TROLL .
  inc STRATEGY .

  var id : Identifier .
  var M M' upds : MapKD .
  var a : AName .
  var names : ANames .
  var actionoutput : IdStates .
  var sn : SensorNames .
  var sensorvalues : IdStates .
  vars d1 d2 : Data .
  op swapGoal? : MapKD -> MapKD .
  ceq swapGoal?(M) = insert( k("next"), d1, insert(k("goal"), d2, M))
      if d1 := M[k("goal")] /\
         d2 := M[k("next")] /\
         M[k("pos")] == M[k("goal")] .
  eq swapGoal?(M) = M [owise] .

  *** Alternates between read an write actions
  ceq getPostState(id, Troll, id, a, actionoutput, M) = insert(k("read"),
       nd(0), M) if isMove?(a) .

  ceq getPostState(id, Troll, id, readSensors(sn), sensorvalues, M) = M'
      if upds := getSensorValues(getResources(id, readSensors(sn)) ,
          sensorvalues) /\
         M' := insert(k("read"),  nd(1),
               insert(k("bat"), upds[k("bat")],
               insert(k("station"), upds[k("station")],
               insert(k("pos"), upds[k("pos")], M))))   .

  ceq getPostState(id, Troll, id, lock, actionoutput, M) =  M'
      if M' := insert(k("charging"), nd(1), M) .
  ceq getPostState(id, Troll, id, unlock, actionoutput, M) = M'
      if M' := insert(k("charging"), nd(0), M) .

  eq getPostState(id, Troll, id, end, actionoutput, M) = swapGoal?(M) .

  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(1), M) if M[k("
      read")] == nd(0) .
  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(0), M) if M[k("
      read")] == nd(1) .

  eq computeActions(id , Troll , M ) = getSoftActions(id, M ,
      trollActions(id, M)) .
  ceq getSoftActions( id , M , a names ) = getValue(id, M, a) +
      getSoftActions( id , M ,names ) if a =/= void .
  eq getSoftActions( id , M , void) = null .


endfm
```

We change `SCENARIO` to now use the new `TROLL-ALT` module. We search if both trolls can run out of energy:

```
search [1] in SCENARIO-STATION : init =>* [sys::Sys
[bat(0) : Battery | k("bat") |-> nd(0) ; true ; null]
[bat(1) : Battery | k("bat") |-> nd(0) ; true ; null]] .

No solution.
states: 280  rewrites: 1155509 in 189ms cpu (189ms real) (6091179
    rewrites/second)
```

The new strategy for the robot changes the overall behavior of the composition, and makes the liveness property true.

Self-sorting property The property P_{sorted} is a reachability property on the state of the grid, that states that *eventually, all robots are in the sorted position.* In Maude, given a system of 3 robots, we express such reachability property with the following search command:

```
 search [1] init =>*
    [sys::Sys  [ field : Field |
                 k((0;0)) |-> d(id(0)),
                 k((1;0)) |-> d(id(1)),
                 k((2;0)) |-> d(id(2)) ; true ; null]] .
```

The initial configuration of the grid is such that robot 0 is on location $(2;0)$, robot 1 on $(1;0)$, and robot 2 on $(0;0)$. Since the grid is of size 3 by 2, robots need to coordinate to reach the desired sorted configuration.

Table 5.2 features three variations on the sorting problem. The first system is composed of robots whose move are free on the grid. The second adds one battery for each component, whose energy level decreases for each robot move. The third system adds a swap protocol for every pair of two robots. The last system adds protocol and batteries to compose with the robots.

We record, for each of those systems, whether the sorted configuration is reachable (P_{sorted}), and if all three robots can run out of energy (P_{bat}). Observe that the reachability query returns a solution for both system: the one with and without protocols. However, the time to reach the first solution increases as the number of states and transition increases (adding the protocol components). We leave as future work some optimizations to improve on our results.

Table 5.2: Evaluation of different systems for the P_{sorted} and P_{bat} behavioral properties, where *st.* stands for states, *rw* for rewrites. Note that the P_{bat} property is not evaluated when the system does not contain battery components.

System	P_{sorted}	P_{bat}
$\underset{0\leq i\leq 2}{\bowtie} R_i \bowtie G$	12.10^3 st., 25s, 31.10^6 rw	
$\underset{0\leq i\leq 2}{\bowtie} (R_i \bowtie B_i) \bowtie G$	12.10^3 st., 25s, 31.10^6 rw	true
$\underset{0\leq i\leq 2}{\bowtie} R_i \bowtie G \underset{0\leq i<j\leq 2}{\bowtie} S(R_i, R_j)$	8250 st., 44s, 80.10^6 rw	
$\underset{0\leq i\leq 2}{\bowtie} (R_i \bowtie B_i) \bowtie G \underset{0\leq i<j\leq 2}{\bowtie} S(R_i, R_j)$	8250 st., 71s, 83.10^6 rw	false