



**HAL**  
open science

# An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization

Farid Arfi, Hélène Coullon, Frédéric Loulergue, Jolan Philippe, Simon Robillard

► **To cite this version:**

Farid Arfi, Hélène Coullon, Frédéric Loulergue, Jolan Philippe, Simon Robillard. An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization. ICE 2024 - 17th Interaction and Concurrency Experience, Jun 2024, Groningen, Netherlands. pp.1-18. hal-04572043

**HAL Id: hal-04572043**

<https://inria.hal.science/hal-04572043v1>

Submitted on 2 Oct 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization

Farid Arfi                      H el ene Coullon

IMT Atlantique, Inria, LS2N, UMR 6004, F-44000 Nantes, France  
{farid.arfi, helene.coullon}@imt-atlantique.fr

Fr ed eric Loulergue

Univ. Orl eans, INSA CVL, LIFO EA 4022, France  
frederic.loulergue@univ-orleans.fr

Jolan Philippe

IMT Atlantique, Inria, LS2N, UMR 6004, F-44000 Nantes, France  
jolan.philippe@imt-atlantique.fr

Simon Robillard

LIRMM, CNRS, Universit e de Montpellier, France  
simon.robillard@umontpellier.fr

We propose an overview of the decentralized reconfiguration language CONCERTO-D through its Maude formalization. CONCERTO-D extends the already published CONCERTO language. CONCERTO-D improves on two different parameters compared with related work: the decentralized coordination of numerous local reconfiguration plans which avoid a single point of failure when considering unstable networks such as edge computing, or cyber-physical systems (CPS) for instance; and a mechanized formal semantics of the language with Maude which offers guarantees on the executability of the semantics. Throughout the paper, the CONCERTO-D language and its semantics are exemplified with a reconfiguration extracted from a real case study on a CPS. We rely on the Maude formal specification language, which is based on rewriting logic, and consequently perfectly suited for describing a concurrent model.

## 1 Introduction

Running and maintaining large-scale distributed software is now a commonplace activity, but managing the inherent complexity of this task requires dedicated tools, models, and languages. The complexity is particularly apparent when distributed software needs to be reconfigured during execution, either to satisfy changing requirements or to carry out maintenance operations.

The DevOps community (and associated tools) as well as component-base software engineering (CBSE) are the main domains focussing on the deployment and reconfiguration of distributed software systems. A reconfiguration consists of a set of actions to execute on the different pieces of software, distributed across the network, to lead the system in the new desired configuration (i.e., state). In these domains, actions are almost always orchestrated by a central coordinator [25, 9], i.e., an entity that keeps track of the actions to apply and their dependencies, as well as the global state of the distributed system.

However, a centralized model is necessarily limited in terms of resilience, as it creates a single point of failure. For instance, in the context of constrained (e.g., energy, communications) cyber-physical systems [21, 22], or edge computing where network disconnections are commonplace, as well as in the

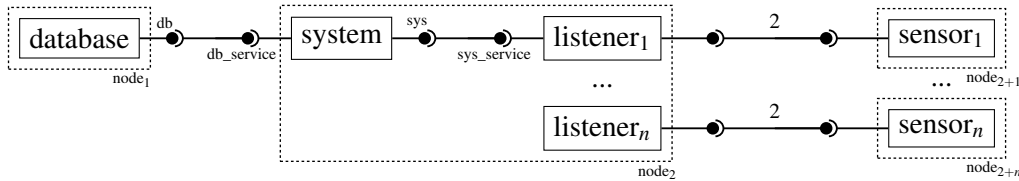


Figure 1: Full overview of the CPS use case with  $2 + n$  nodes hosting *database*, *system*,  $n$  *listeners*, and  $n$  *sensors*.

context of large-scale projects where cross-DevOps teams [16, 23, 25, 26] have to collaborate, decentralized reconfiguration models are preferred.

With this work, we extend the semantics of the reconfiguration language CONCERTO [6] and turn it into a decentralized model called CONCERTO-D, by extending the semantics to describe the specifics of communication and synchronization between components. In CONCERTO-D multiple coordinators collaborate to achieve their respective local reconfigurations (one for each node). Both CONCERTO [4, 5] and CONCERTO-D [23, 21, 22] have been the subject of experimental studies to validate the approach and compare it to related work [10, 25]. In particular, and while this is not the main subject of this paper, CONCERTO and CONCERTO-D allow better parallel execution of reconfiguration actions compared to the related work, thus leading to faster reconfigurations.

To support the development of this decentralized semantics, it appeared necessary to go beyond a pen-and-paper approach and to provide a mechanized formalization of the semantics. To this end, we used Maude [8, 20], a language based on rewriting logic, and consequently perfectly suited to describing a concurrent model. Developing and formalizing the semantics of CONCERTO-D in Maude helped us manage the complexity of the model, and clarify it where needed, but it also allowed us to generate tools such as an interpreter and a model-checker for these semantics. A Maude program describes a logical theory, while a Maude computation consists of logical deductions using the axioms specified in the theory.

Throughout the paper, we will use an example taken from a real case study from the literature: a wildlife monitoring system that utilizes sensors to capture animal sounds [19] illustrated in Figure 1. These sensors are linked to a gateway and calibrated to specific sound frequencies. Reconfiguration is regularly required in this system to adjust the listening frequency of each sensor. However, during the reconfiguration process, the sensor must temporarily cease listening. Similarly, if the gateway fails to receive data from a sensor, the sensor is required to halt its observation activities. The collected data is stored in a remote database. In such a use case, disconnections are common between listeners and sensors. A central coordination of the reconfiguration program is typically a single point of failure, which makes the system fully inactive during unavailability. Furthermore, when facing disconnections, it is more difficult to maintain the global state of the system, which slows down the process: synchronization with the central entity is required even for purely local actions. A decentralized language such as CONCERTO-D is preferable in such a context.

In this paper, we give an overview of the semantics of CONCERTO-D through its Maude formalization. The complete Maude specification with all the rules is available at <https://doi.org/10.5281/zenodo.12786127>.

First, Section 2 presents how the life cycles of components, and the creation and modification of a components assembly are formalized in CONCERTO-D. Second, Section 3 gives an overview of the main semantics rules of CONCERTO-D and how are formalized communications in CONCERTO-D. Finally,

Section 4 and Section 5 respectively give the related work and a conclusion on this contribution.

## 2 Control components and reconfiguration language

This section presents the structural concepts of CONCERTO-D as well as its reconfiguration language. Some elements of the formalization in Maude are included, as well as examples based on our case study.

### 2.1 Control components

In CONCERTO-D, a distributed software system is modeled as an assembly of control component instances, linked together via *ports* (i.e., interfaces), to represent dependencies in their life cycles (e.g., data exchanges, or other interactions between components). There are two types of ports in such connections: *provide* ports and *use* ports. A provide port represents information or a resource offered by a component when the port is active. Conversely, a use port indicates that a component requires some information or resources to perform.

Each piece of software (i.e., component, service), identified by a unique identifier, is an instance of a control component type, that models a component's life cycle as a set of *places* and *transitions*. Each port is bound to a subset of the life cycle (i.e., places and transitions) that is called a group. Places represent milestones of the life cycle. A specific initial place serves as the starting point for the component's life cycle. Transitions represent concrete actions to perform between places (e.g., admin commands).

The dynamic nature of components is captured by their *behaviors*. A behavior is a subset of the component's transitions. At any given point, a control component instance executes one behavior, i.e., only the transitions in this behavior can be fired. Requests to change the active behavior for a component (see reconfiguration actions in Section 2.2) are queued and executed in the order in which they are received: a behavior remains active until no more transitions in it can be fired, at which point the behavior is changed to the next requested behavior, if any. This mechanism makes up the behavioral interface of components.

**Informal example.** Figure 1 depicts an assembly modeling our use case, where distributed sensors interact with a system through listeners. Physically, the components are distributed on several nodes. The first node serves as a host for the *database* component responsible for storing recorded data obtained from sensors. The second node comprises the components *system* and *listener*. The *system* component establishes a connection with the database through a *use* port (represented using the UML notation), enabling the usage of the database's service. Furthermore, the *system* is interconnected with  $n$  *listeners*, each corresponding to a program responsible for monitoring and reconfiguring remote sensors. These listeners are connected to *system* via their use port. Finally, each  $sensor_i$  is hosted on a node and is connected to its associated  $listener_i$ .

Figure 2 shows the internals of some components of this assembly, omitting others (i.e., *database*, *system*) for clarity. Each listener has four places: *off*, *paused*, *configured*, and *running*; and three behaviors *deploy*, *update*, and *destroy*. The sensors have five places: *off*, *provisioned*, *installed*, *configured*, and *running*; and also three behaviors: *start*, *pause*, *stop*. Each listener has two connections with its respective sensor. Through the first connection, the listener exposes the configuration to apply on the sensor (e.g., frequency of listening). Through the second connection, the listener offers a service to which data can be sent by the sensor when observing.

**Maude specification.** Let us present the definitions of sorts, subsorts, and operators used to encode the above syntactic aspects of the CONCERTO-D model, starting from elementary concepts to the construction of a net of CONCERTO-D components. In Maude, a type hierarchy can be defined using the keywords `sort` and `subsort`. An operator definition starts with the keyword `op` followed by the operator name and type signature; several operators with the same signature can be defined using `ops`.

A component type is defined by a set of places, an initial place, and a set of transitions. The definition also includes the behaviors (sets of transitions) of the component type, and its use and provide ports, each bound to a group of places. In the definition below, some details are omitted for simplicity.

We use predefined data structures and the module system of Maude to import the parameterized module `SET{Place}` and define the sort `Places` to be a supersort of `SET{Place}`. Generally, given a sort `T`, we let `Ts` stand for `SET{T}` and `QT` stand for `LIST{T}`.

```

1  sorts Place InitialPlace Transition Behavior UsePort ProvidePort GroupUse GroupProvide
2      ComponentType .
3  sorts Places Transitions Behaviors GroupUses GroupProvides .
4  subsort InitialPlace < Place .
5
6  --- [...]
7  op b(_) : Transitions -> Behavior [ctor] .
8  op g(_?) : UsePort Places -> GroupUse [ctor] .
9  op g(!_!) : ProvidePort Places -> GroupProvide [ctor] .
10
11 op { places: _,
12     initial: _,
13     --- [...]
14     transitions: _,
15     behaviors: _,
16     groupUses: _,
17     groupProvides: _
18 } :
19 Places InitialPlace
20 --- [...]
21 Transitions Behaviors GroupUses GroupProvides ->
22 ComponentType
23 [ctor] .

```

**Example in Maude.** We can now describe the component type `sensor` (an instance of which is displayed on the right of Figure 2). We first define a few constants corresponding to the element of the component type, then the type itself:

```

1  ops Running Configured Installed Provisioned Off : -> Place .
2  op Off : -> InitPlace .
3  --- [...]
4  ops RcvService ConfigService : -> UsePort .
5  ops Start11 Start12 Start13 Start2 Start3 Start4 Pause1 Stop1 : -> Transition .
6  ops Start Pause Stop : -> Behavior .
7
8  eq Deploy = b(Start11,Start12,Start13,Start2,Start3,Start4) .
9  eq Pause = b(Pause1) .
10 eq Stop = b(Stop1) .
11 op sensor : -> ComponentType .

```

```

12 eq sensor =
13   { places: Running, Configured, Installed, Provisioned, Off,
14     initial: Off,
15     --- [...]
16     transitions: (Start11,Start12,Start13,Start2,Start3,Start4,Pause1,Stop1),
17     behaviors: (Start, Pause, Stop),
18     groupUses: g(RcvService \lstinline[language=maude,basicstyle=\ttfamily]! (
19       Configured, Running)),
20       g(ConfigService \lstinline[language=maude,basicstyle=\ttfamily]! (
21         Configured, Installed, Running)),
22     groupProvides: empty } .

```

## 2.2 Reconfiguration language and assembly of components

To allow the modification of assemblies, CONCERTO-D proposes a simple imperative language for writing reconfiguration programs, that offers four commonly used topological actions to create or modify the existing assemblies:  $add(id_c, c)$ ,  $del(id_c)$  (creation and deletion of control component instances),  $con(id_{c1}, u, id_{c2}, p)$ ,  $dcon(id_{c1}, u, id_{c2}, p)$  (connection and disconnection between two control component instances), where  $id_c$  is an instance identifier,  $c$  a component type,  $u$  a use port,  $p$  a provide port. Besides these actions, two additional actions manage the execution of behaviors:  $pushB(id_c, b, id_b)$  to request the execution of a behavior  $b$  on the component instance  $id_c$ ; and  $wait(id_c, id_b)$  to synchronize onto the end of a given behavior.  $id_c$  is a component instance identifier, and  $id_b$  the identifier of a given  $pushB$ .

As a decentralized coordination model, CONCERTO-D considers a system of  $n$  nodes, and a partition of component instances over these nodes. Each node operates its own local CONCERTO-D controller and runs its own reconfiguration program. Concrete communications allow the synchronization of actions between paired components. This is an evolution of previous work on the CONCERTO model [6], in which a single central entity executed the reconfiguration and synchronization of components, and communications were implicit. Two examples of CONCERTO-D reconfiguration programs in our case study are given in Listings 1 and 2.

Listing 1: Listeners reconfiguration on  $node_2$

```

for i in range(nb_listener):
  pushB(listener_i, update, 2+i*2)
  pushB(listener_i, deploy, 3+i*2)

```

Listing 2: One sensor reconfiguration on  $node_{2+i}$

```

pushB(sensor_i, pause, 0)
wait(listener_i, 2+i*2)
pushB(sensor_i, start, 1)

```

In CONCERTO-D reconfiguration, programs apply changes to the current assembly of components, i.e., component instances and their connection, and to the queue of requested behaviors for each component instance. A specific instance of a component (and its state at any given point of the execution) is specified by its component type, a queue of identified behaviors to be executed by the instance, and a marking that indicates the places reached and transitions fired.

```

1 sorts Instance Id PushedBehavior TransitionEnding Marking .
2
3 op (_,_) : Id Behavior -> PushedBehavior [ctor] .
4 op m(,_,,) : Places Transitions TransitionEndings -> Marking [ctor] .
5 op { type: _,

```

```

6     queue: _,
7     marking: _
8   } :
9   ComponentType List{PushedBehavior} Marking -> Instance [ctor] .

```

To illustrate this, the definition below describes an instance `sensor1` of type `sensor`, in a state where only the place `running` is marked, and a single behavior `pause` is pending in the queue of behaviors. This corresponds to the state (0) in Figure 2.

```

1 eq instanceS1 = { type: sensor,
2                 queue: (0 ; Pause),
3                 marking: m(Running, empty, empty) } .

```

In order to describe an assembly, it is also necessary to specify the connections between the ports of its component instances, through their identifiers.

```

1 sort Connection .
2 sorts Connections .
3 op (_,_)--(_,_) : Id UsePort Id ProvidePort -> Connection [ctor] .

```

Here is an example of such connections between `node2` and `node3`:

```

1 eq connectionSL1 = (sensor1, RcvService)--(listener1, Rcv) .
2 eq connectionSL2 = (sensor1, ConfigService)--(listener1, Config) .

```

We now define actions that can be executed to perform a reconfiguration on a CONCERTO-D assembly.

```

1 sorts Action Program .
2 subsort List{Action} < Program .
3
4 op add(,_): Id ComponentType -> Action [ctor] .
5 op del(,_): Id -> Action [ctor] .
6 op pushB(,_,_): Id Behavior Id -> Action [ctor] .
7 op con(,_): Connection -> Action [ctor] .
8 op dcon(,_): Connection -> Action [ctor] .
9 op wait(,_,_): Id Id -> Action [ctor] .

```

The reconfiguration program given in Listing 2 (as instantiated specifically for `node3`) can thus be specified in Maude as follows:

```

eq programNode3 =
pushB(sensor1, start, 0) wait(listener1, 4) pushB(sensor1, start, 1) .

```

### 3 Elements of operational semantics

CONCERTO-D is equipped with a small-step semantics. We first illustrate the execution of a reconfiguration on our use case to give an intuition of this semantics.

We consider deployed and running components, i.e., the places `running` are marked in all listener and sensor components. From there, we aim to trigger an update of each sensor's listening frequency. Hence, each listener has to pause to change its configuration, forcing the sensors to pause as well.

Listings 1 and 2 give the reconfiguration programs respectively for all listeners (all hosted on the same node `node2`) and one sensor `i` (corresponding to the update of the frequency of each sensor). These

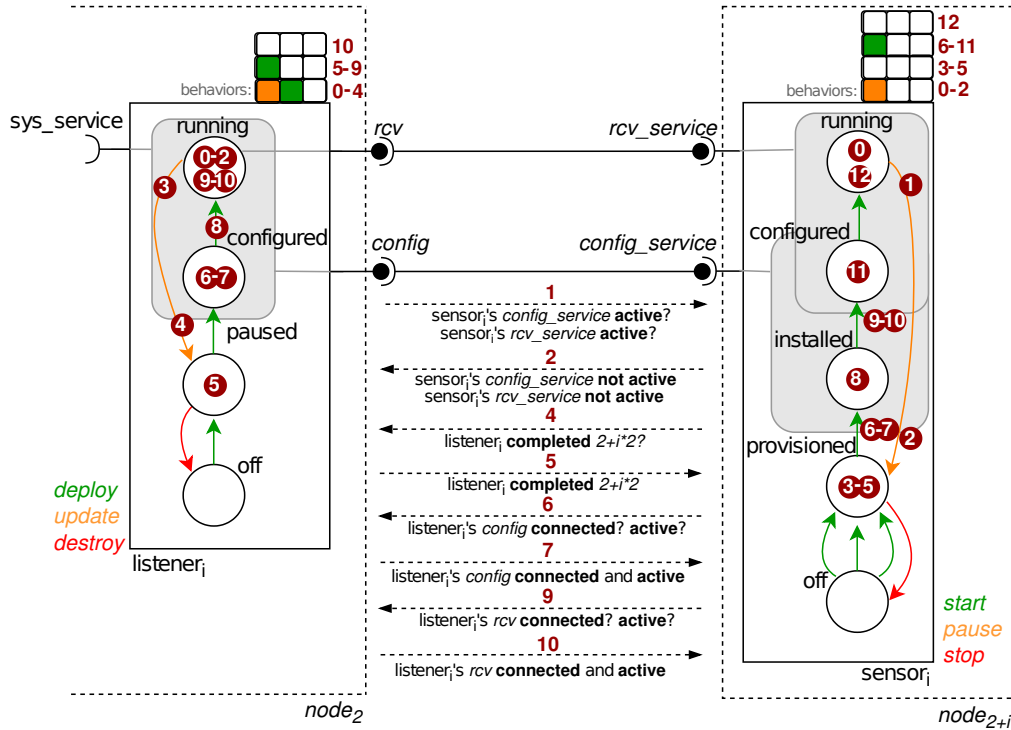


Figure 2: Control components of  $listener_i$  on  $node_2$  and  $sensor_i$  from  $node_{2+i}$ . State example when applying the reconfiguration plans of Listings 1 and 2.

programs are executed concurrently on each node. The execution of these scripts is illustrated in Figure 2 to give the reader an intuition of the semantics of CONCERTO-D. Some possible steps of the execution are represented by a number representing the current configuration of the system (i.e., a snapshot). Using these numbers, three pieces of information are given: (i) the marking, represented by the red dots on marked places and transitions; (ii) the status of the behavior queue, and (iii) the coordination information required between nodes introduced by decentralized execution of CONCERTO-D. In the following, we describe the state of each configuration according to its number. We decompose each state, and we highlight communication steps using the notation  $\Delta$ .

0. Both  $listener_i$  and  $sensor_i$  services are running. Thus, the running places are marked. The behaviors to trigger the update are pushed in the queue, i.e., update and deploy for  $listener_i$ , and pause for  $sensor_i$ . The behavior start for  $sensor_i$  is not pushed due to wait action of a behavior of  $listener_i$ .
1. By triggering pause,  $sensor_i$  leaves the place running.
  - $\Delta$   $listener_i$  needs to know if any component is using its ports before executing update. Then,  $listener_i$  requests information on  $sensor_i$ 's use ports  $rcv\_service$  and  $config\_service$ , to know if they are active or not.
2.  $sensor_i$  ends its previously fired transition.
  - $\Delta$   $sensor_i$  answers  $listener_i$  about its use ports  $rcv\_service$  and  $config\_service$ , indicating that both are inactive.



3.  $listener_i$  can begin the `update` behavior, therefore deactivating its `provide` ports.  $sensor_i$  removes the `pause` behavior from its queue, as no more transitions in this behavior can be fired.
  - Δ The information sent in 2 is received by  $listener_i$ , allowing it to start its `update`.
4.  $listener_i$  finishes its transition related to its `update` behavior.
  - Δ  $sensor_i$  is now waiting the behavior identified by  $2+i*2$ , and runs on  $listener_i$ , to be ended. Then it sends a request to  $listener_i$  to know if the behavior  $2+i*2$  (`update`) is completed.
5. The behavior `update` of  $listener_i$  is retrieved from its queue.
  - Δ  $sensor_i$  is informed that this behavior is completed, which allows it to continue its execution, i.e., to push and execute the transitions of the `start` behavior.
6. After pushing the behavior `start` and completing its first transition.
  - Δ  $sensor_i$  sends a request to determine whether the port `config` of  $listener_i$  is connected and active, before entering in `config_service` group.
7. Δ  $sensor_i$  receives information on  $listener_i$ 's port `config`. It is now active and connected.
8.  $sensor_i$  enters the place `installed`.  $listener_i$  begins the activation of its service.
9.  $listener_i$  activates its service and provides the port `rcv`.
  - Δ  $sensor_i$  sends a request asking whether the port `rcv` is active and connected before entering the place `configured`.
10.  $listener_i$  removes the `deploy` behavior from its queue.
  - Δ  $sensor_i$  get the information that  $listener_i$  provides the port `rcv`.
11.  $sensor_i$  enters the place `configured`.
12.  $sensor_i$  reactivates its service and removes the `start` behavior from its queue.

The rest of this section gives some formal elements of the small-step semantics of CONCERTO-D. Because of space reasons, we cannot fully detail all rules of the semantics. Instead, the section first illustrates one execution rule at a reconfiguration program level, one execution rule at a component level, and then details communications, i.e., the main formalization challenge in a decentralized model as CONCERTO-D.

### 3.1 Execution and synchronization of reconfiguration actions

Let us first describe the semantics rules that govern the execution of actions in reconfiguration programs. Recall that each node executes its own reconfiguration program, affecting components hosted on that node. Actions to add and delete component instances are straightforward. Actions for connections and disconnections of components are also fairly standard. They only modify the view that one node has of the component's topology. When components to (dis)connect are located on different nodes, both nodes should eventually execute their own similar (dis)connection actions, however, the node-local views of the topology do not have to be kept continuously synchronized. Finally, we turn to the `pushB` action, which is specific to the CONCERTO-D model. The rule in Listing 3 gives the semantic of that action, describing the modification of a node (as defined in Section 3.2). The action adds a behavior request to the queue of behaviors of one component. The target component is indicated by its identifier, and the behavior type is indicated, as well as an identifier. The latter is used for synchronization points, in order to distinguish between multiple executions of the same behavior type.

The behavior requests determine the active behavior of a component instance at a given point, which plays a role in the evolution of this component. We now describe the rules of this evolution. As described

in Section 2, the life cycles of components are modeled by places (milestones in the reconfiguration) and transitions between places (reconfiguration actions). A marking on places and transitions indicates the current state of the reconfiguration. When a place is marked, its outgoing transitions can be fired. The place is then unmarked, and the fired transitions are marked. Conversely, when all the incoming transitions toward a place are finished, those transitions are unmarked and the place is entered. This model is well suited to represent concurrent execution, in particular, multiple transitions can be marked simultaneously, corresponding to parallel execution of reconfiguration actions.

Listing 3: The rule to execute the pushB action. The rule applies to a node (as defined in Section 3.2) but rewrites a single component instance in that node.

```

1  cr1 [PushingBehavior] :
2  < --- [...]
3    instances: (Id1 ↦ { type: Ct, queue: Qb, marking: M }) , Isx,
4    --- [...]
5    program: pushB(Id1, B, IdB) RPx,
6    --- [...] >
7  =>
8  < --- [...]
9    instances: ( Id1 ↦ { type: Ct, queue: append(Qb,(IdB ; B)), marking: M } ), Isx,
10   --- [...]
11   program: RPx,
12   --- [...]
13  >
14  if ( B in (Ct).behaviors) .

```

This evolution of markings is described by four rules: *FiringTransitions*, *EndingTransition*, *EnteringPlace*, and *FinishingBehavior*. For space reasons, we only detail one of these rules: *Firing Transitions* given in Listing 4. It modifies the marking in one component on one node, namely unmarking a place P and marking the outgoing transitions of Ts (only those transitions that belong to the active behavior of the component). Note that this is a conditional rule, as indicated by the keyword `cr1` and the boolean conditions after the rule. The first condition merely ensures that there are transitions to fire. The second condition relies on the predicate `isSafeToFire` to check dependencies towards other components, modeled by ports, are not violated by the firing of the transitions. In particular, this predicate is true only if the firing of the transitions does not lead to deactivating a provide port that is being used, i.e., connected to an active use port. The rule to end a transition (not given) is somewhat similar but instead requires that use ports attached to the reached place are connected to an active provide port. Thus, ports impose inter-component synchronization conditions on the execution.

Listing 4: The rule to fire a transition. As before, this rule modifies a component instance in a node.

```

1  cr1 [FiringTransitions] :
2  < nodeInventory: Idsx,
3    instances: (Id1 ↦ { type: Ct,
4                      queue: (IdB ; b(TsB)) Qb,
5                      marking: m((P, Ps),Ts,Tes)
6                      }) , Isx,
7    connections: Csx,
8    --- [...] >
9  =>
10 < nodeInventory: Idsx,

```

```

11     instances: (Id1 ↦ { type: Ct,
12                       queue: (IdB ; b(TsB)) Qb,
13                       marking: m(Ps,union(Ts, getTransitionsofPlace(TsB,P) ),
14                                     Tes)
15                       }), Isx ,
16     connections: Csx,
17     --- [...] >
18     if ( getTransitionsofPlace(TsB,P) /= empty and
19         isSafeToFire( Idsx,
20                       m(Ps,union(Ts, getTransitionsofPlace(TsB,P) ),Tes) ,
21                       (Id1 ↦ { type: Ct,
22                               queue: (IdB ; b(TsB)) Qb,
23                               marking: m(P, Ps,Ts,Tes)
24                               } ,Isx),
25                       eStatex,
26                       getPConnectionsofCInst(Id1, Csx))) .

```

### 3.2 Communications

Firing or ending a transition on one component may require checking the activity status of a port on another component. In previous work [6], the coordination was assumed to be carried out by a central entity that kept track of the status of every port. Instead, CONCERTO-D is meant to represent a decentralized process, it is therefore necessary to explicitly model communications between the nodes that host components.

The following boolean information may be needed for synchronization, and can be called on remote nodes: (1) the completion of a *dcon* action; (2) the completion of a behavior (i.e., *wait* action); (3) the activity of a use or provide port.

When evaluating the status of one of the above queries (such as the status of a port in the predicate *safeToFire*), we distinguish the case where the element of interest (*dcon* action, behavior, port etc.) belongs to the local node, from the case where messages must be exchanged between nodes:

```

1   eq question(R1,Ids,RcvAns, Is,Cs,RP ) = if (isLocal(R1,Ids)) then
2       localQuestion(R1,Is,Cs,RP)
3       else externQuestion(R1,RcvAns) fi .

```

Essentially, each CONCERTO-D node maintains a localized perspective of its components and communicates with neighboring nodes (i.e., other CONCERTO-D controllers hosting connected control components). In CONCERTO-D, an asynchronous message-based communication model, increasingly favored in distributed systems, manages the communication [21]. To achieve asynchronous communications each node is equipped with an incoming queue of messages. Messages must transit through this queue before being effectively received (this assumes that the order of messages is preserved).

A message can either be a *Question* or an *Answer*. The former is aimed at a specified component instance and contains one of several possible queries while an *Answer* gives the boolean value corresponding to a question. It can take the values *true*, *false*.

Listing 5: Constructors for Question, Answer and Message

```

1   sorts Query .
2   op isActive(_) : Port -> Query [ctor] .

```

```

3 op isRefusing(_) : Port -> Query [ctor] .
4 op isConnected(_) : Connection -> Query [ctor] .
5 op isCompleted(_) : Id -> Query [ctor] .
6 op onDisconnect(_) : Connection -> Query [ctor] .
7 op [ dst: _ , query: _ ] : Id Query -> Question [ctor] .
8 op [ question: _ , value: _ ] : Question Bool -> Answer [ctor] .
9 op mkMsg : Question -> Message [ctor] .
10 op mkMsg : Answer -> Message [ctor] .

```

For example, the question [ dst: listener1, query: isActive(Rcv)] can be sent by node3 (which hosts sensor1) to node2 (which hosts listener1) to check whether the port Rcv of listener1 is active. This corresponds to the state 9 in Figure 2. The answer [ req: [ dst: listener1, query : isActive(Rcv)], value: true ] is the answer which will be returned at state (10) to indicate that the port is indeed active. For more details about the remaining queries, the reader can refer to [6].

A node is specified by the set of identifiers of all the component instances involved in the reconfiguration programs, a mapping (called an inventory) that associates these identifiers to their associated nodes, the connections between local component instances and external instances, and the local reconfiguration program to be executed on the node. A node is additionally specified by five parameters used for communication: (1) a local vision of the state of external components `externState`; (2) a queue of pending questions waiting for an answer `pendingQuestions`; (3) a queue of outgoing questions asked to other nodes `outgoingQuestions`; (4) a queue of incoming messages `incomingMsgs` including incoming questions asked by other nodes and answers received from other nodes to previously sent questions; and finally (5) a history of previously sent questions (to avoid redundant messages for already pending requests). Received answers are stored in `externState` as a mapping that associates requests to the value of the answer. This represents the local node's vision of the status of other components. This information is updated each time external information is needed in a semantics rule.

#### Listing 6: Local configuration

```

1 op < nodeInventory: _,
2     instances: _,
3     connections: _,
4     program: _,
5     externState: _,
6     pendingQuestions: _,
7     outgoingQuestions: _,
8     history: _,
9     incomingMsgs: _
10 > :
11 Ids Instances Connections Program Map{Question, Bool} List{Question} List{
12     Question} Set{Question} List{Message}
13 -> LocalConfiguration

```

For example, the description of the local configuration of node3 in state 0 is as follows:

```

1 eq ConfNode3 =
2   < nodeInventory: sensor1,
3     instances: { sensor1 ↦ instanceS1 },
4     connections: { connectionSL1, connectionSL2 },
5     program: programNode3,
6     externState: eStateNode3,

```

```

7     pendingQuestions: nil,
8     outgoingQuestions: nil,
9     history: empty,
10    incomingMsgs: nil >

```

where `instanceS1`, `connectionSL1`, `connectionSL2` and `programNode3` are the elements previously described for our use case in Sections 2 and 3.2. `sensor1` is the identifier of `instanceS1` and `eStateNode3` is the external state (i.e., mapping of questions and received answers sent by node3). Following the deployment steps that preceded the reconfiguration state 0, `eStateNode3` of node3 is described as follows:

```

1 eq eStateNode3 =
2   [ dst: listener1, query: isRefusing(Rcv) ] ↪ false ,
3   [ dst: listener1, query: isConnected(connectionSL1) ] ↪ true ,
4   [ dst: listener1, query: isRefusing(Config) ] ↪ false ,
5   [ dst: listener1, query: isConnected(connectionSL2) ] ↪ true .

```

The rest of this section gives the execution semantics of communication in CONCERTO-D using Maude rewrite rules, which operate on a system of local configurations of nodes. The associated rules for communication are presented in listings 7, 8, 9 and 10. The rules consider two nodes  $x$  and  $y$ .

First, Listing 7 describes the rule [SendQuestion] sending of a request [ `dst: Dst, query: Q` ] from node  $x$  to node  $y$ . The destination node  $y$  is determined since the identifier of the instance of the request `Dst` appears in the inventory of node  $y$ . Sending the request implies adding to the incoming messages of  $y$  the message `mkMsg([ dst: Dst, query: Q ])`. The sent request will also be inserted in the history `Hx` of node  $x$  to avoid the request being sent multiple times while waiting for an answer.

Listing 7: The rule to send a question, that rewrites two nodes (unaffected variables are omitted).

```

1 rl [SendQuestion] :
2   < --- [...]
3   outgoingQuestions: [ dst: Dst, query: QY ] OutQx,
4   history: Hx,
5   --- [...]
6   > ,
7   < nodeInventory: (Dst, Idsy),
8   ---[...]
9   incomingMsgs: iMsgsy >
10  =>
11  < --- [...]
12  outgoingQuestions: OutQx,
13  history: ([ dst: Dst, query: Q ], Hx),
14  --- [...]
15  > ,
16  < nodeInventory: (Dst, Ids),
17  --- [...]
18  incomingMsgs: append(iMsgsy, mkMsg([ dst: Dst, query: QY ])) >

```

Second, Listing 8 describes the sending of an answer from node  $x$  to node  $y$  concerning a question `Q` previously sent by  $y$ . The destination node  $y$  is chosen when the concerned request `Q` is in its history of sent requests, and has not yet received in an answer to this request (!not occurs(`mkMsg(Q)`, `iMsgsy`)!). The value of the answer is computed on node  $x$  by the function `localQuestion` and sent to node  $y$  by placing the answer in the incoming messages of node  $y$ .

It is important to maintain the consistency of shared information, so if a node  $x$  sends information that could allow an external component  $y$  to change the status of one of its ports  $p$ , the previously recorded information about  $p$  on  $x$  is deleted. For example, when a component on node  $y$  asks  $x$  if one of its provide port  $p$  is active, it means that the associated use port  $u$  on  $y$  may be activated soon after the answer is sent by  $x$ . Consequently, the node  $x$  that sends the answer on  $p$  will simultaneously reset the information in its external state regarding  $u$  on  $y$  (of course, it should not yet assume that the use port is active). In Listing 8, this update is carried out by the function `resetState`.

Listing 8: The (conditional) rule to send an answer

```

1  cr1 [SendAnswer] :
2    < --- [...]
3      instances: Isx,
4      connections: Csx,
5      program: RPx,
6      externState: eStatex,
7      pendingQuestions: Q pendingQx,
8      --- [...]
9    > ,
10   < --- [...]
11     history: (Q, Hy),
12     incomingMsgs: iMsgsy >
13   =>
14   < --- [...]
15     instances: Isx,
16     connections: Csx,
17     program: RPx,
18     externState: resetState(eStatex, getIdsForReset(Csx, Q, localQuestion(Q, Isx, Csx, RPx))),
19     pendingQuestions: pendingQx,
20     --- [...]
21   > ,
22   < --- [...]
23     history: Hy,
24     incomingMsgs: append(iMsgsy, mkMsg([ question: Q,
25                                           value: localQuestion(Q, Isx, Csx, RPx) ]))
26   >
27   if (not occurs(mkMsg(Q), iMsgsy)) .

```

Third, Listing 9 describes the rule for receiving a question on a component local to  $x$ . This is started when the identifier `Dst` of the request [`dst: Dst, query: Q`] is in the set of identifiers of the components of node  $x$ . The request is evaluated and the answer is placed in the queue of pending questions.

Listing 9: The rule for receiving a question

```

1  rl [ReceiveQuestion] :
2    < nodeInventory: (Dst, Idsx),
3      --- [...]
4      pendingQuestions: pendingQx,
5      --- [...]
6      incomingMsgs: mkMsg([dst: Dst, query: QY]) iMsgsx >
7    =>

```

```

8 < nodeInventory: (Dst, Ids),
9   --- [...]
10 pendingQuestions: append(pendingQx,[ dst: Dst, query: QY ]),
11 incomingMsgs: iMsgsx >

```

Finally, Listing 10 describes the rule for receiving an answer to a question  $Q$  previously sent by node  $x$  (as indicated by its presence in the history). The answered value  $val$  is recorded ( $!insert(R, val, RcvA)$ ) and it replaces previous information about the result of request  $Q$ , if any. The request  $Q$  is removed from the history of node  $x$ , so that a similar request may be sent again in the future.

Listing 10: The rule for receiving an answer

```

1 rl [ReceiveAnswer] :
2 < --- [...]
3   externState: eStatex,
4   --- [...]
5   history: (Q,Hx),
6   incomingMsgs: mkMsg([ question: Q, value: Val ]) iMsgsx >
7   =>
8 < --- [...]
9   externState: insert(Q,Val,eStatex),
10  --- [...]
11  history: Hx,
12  incomingMsgs: iMsgsx >

```

**Example.** To illustrate the communication protocol of the above semantics rules, a subpart of the example of Figure 2 is used with one listener and one sensor. In step 4, the sensor sends a question to the listener regarding the completion of its behavior  $idb4$ . The question is created following the  $wait(listener, idb4)$  action in the local reconfiguration program of the sensor, and is placed in  $outgoingQuestion$ . The question is  $[dst: listener1, query: isCompleted(idb4)]$ . The following communication rules are applied:

1. [SendQuestion] is applied: the question in  $outgoingQuestion$  on sensor is popped and placed into history of sensor. The  $incomingMsgs$  of listener appends the question received from sensor.
2. [ReceiveQuestion] is applied: The node listener appends the incoming question to its list  $pendingQuestions$ . The question is removed from  $incomingMsgs$ .
3. [SendAnswer] is applied: The question in  $pendingQuestions$  on listener is popped. The answer is created and pushed to the  $incomingMsgs$  of sensor. The answer message to the question is  $[question: [dst: listener1, query: isCompleted(idb4)], value: true]$ . The original question is removed from history on sensor.
4. [ReceiveAnswer] is applied: The answer received on sensor in  $incomingMsgs$  is removed and inserted (added or updated) to the  $externState$  of sensor as a mapping  $[dst: listener, query: isCompleted(idb4)] \mapsto true$ .

Once completed a pushed behavior cannot be executed again as being uniquely identified. For this reason, no consistency issue can happen in this small example. The  $resetState$  of the rule  $SendAnswer$  has no consequences on  $externState$  in this case.

## 4 Related work

CONCERTO and CONCERTO-D can be considered as component models, such as defined in Component-Based Software Engineering (CBSE). As explained in [6, 9] CONCERTO, and by extension CONCERTO-D, differ from usual component models by modeling the life cycle of components instead of modeling the functional code of components. For this reason, both CONCERTO and CONCERTO-D are more comparable to DevOps approaches. Only one other component model can be compared to CONCERTO: Aeolus [10]. However, Aeolus is more limited than CONCERTO in terms of parallelism and concurrency. As for CONCERTO, and unlike CONCERTO-D, Aeolus is a centralized model, and it has been formalized manually.

Regarding DevOps approaches for deployments, a few contributions have studied a decentralized approach. In [16, 26] each component of the application expresses its dependencies with the other components in a central plan, distributed to the corresponding nodes, deploying their part of the application. Deployment executions are then coordinated between the nodes according to their dependencies. Here, the execution is decentralized as in CONCERTO-D. In [25], an extension of the DevOps tool Pulumi is presented to handle both the computation and execution of deployment and update programs in a decentralized manner. However, the three approaches above, and almost all DevOps tools, lack formal specifications: their language is defined by a unique implementation and informal documentation.

To our knowledge, there are only two DevOps contributions that offer formal semantics of their system configuration languages: SMARTFROG [2] a tool no longer maintained, and  $\mu$ PUPPET [14] a subset of PUPPET. The semantics of SMARTFROG is a denotational semantics of a *compiler* for a core SMARTFROG fragment. From the high-level language SF, which handles features such as inheritance, composition, references, etc., the compilation process produces a store, basically a tree of attribute-value pairs. These trees are also the abstractions for system states in SMARTFROG's semantics. The authors wrote three implementations (in Scala, Haskell, and OCaml) of the compiler guided by the semantics but not proved correct w.r.t. to the formal semantics (which is not mechanized). These implementations were randomly tested and a few implementation errors were found. They were also tested against the production compiler: it allowed them to find both a misunderstanding of the semantics of SMARTFROG and bugs in the production compiler. The semantics of  $\mu$ PUPPET is a small-step operational semantics. An implementation of a  $\mu$ PUPPET compiler exists, guided by the semantics but not proved correct w.r.t. the formal semantics, which is also not mechanized. The output of the compiler is a catalog, i.e., a structure close to the stores' output by SMARTFROG's compiler. A catalog is also an abstraction for a system state. To help debug  $\mu$ PUPPET manifests, Fu [13] also proposed an analysis of provenance [7] based on the  $\mu$ PUPPET operational semantics. In both cases, the formal semantics is not an executable artifact as is our proposal.

Khebbeb et al. also use Maude to formalize adaptation in the Cloud [18] and at the edge [17]. The motivation and approach are however very different from our work. There is no reconfiguration language: the goal is to automatically adapt resources depending on the load of the Cloud system. On the one hand, this work considers the provisioning and de-provisioning of virtual machines, an aspect we do not consider. On the other hand, their concept of service is very simplistic: a service is something that processes requests (and only the number of requests is formalized) and there are no connections between services. Their rewrite rules implement pre-defined elasticity strategies. While they perform model-checking on a small example (3 services, 1 VM, and 2 Fog nodes) [17], they neither provide any information about the number of states, nor the time required for the verification. In its complexity, our proposal is closer to work that models APIs for e.g., [27]. As Yu et al., to analyze interesting enough case studies, we will need to optimize the model-checking by implementing partial order reduction [12].



## 5 Conclusion and Future Work

In this paper, we proposed a formalization in Maude of the decentralized reconfiguration language CONCERTO-D, an extension of the centralized reconfiguration language CONCERTO [6]. In CONCERTO-D, the assembly of components is distributed among a set of nodes. Each node is responsible for a local reconfiguration program that may require some coordination with the nodes that host connected component instances. CONCERTO-D automatically manages the necessary communications between nodes via asynchronous communications, which involve nodes communicating by exchanging messages through buffers. The advantage of having a formal semantics with Maude is that it is executable and hence we can check by running examples that the semantics we designed behaves as we expect it to do.

This is however a first step. We ambition to verify properties of specific assemblies and their reconfiguration programs as well as to verify properties of the semantics itself.

To verify specific assemblies and their programs, Maude offers on-the-fly model-checking of LTL formulas. Preliminary experiments show that in its current form the semantics does not allows to scale up to the model-checking of realistic applications. We will first explore the application of partial order reduction techniques [12] to limit the state-space explosion problem. Another way to avoid such an explosion is to use symbolic model-checking techniques in particular verification by over-approximation of the set of descendants [15, 3]. In these approaches, the goal is to check that a set of bad states does not intersect the over-approximations of the sets of descendants. There are however constraints on how the (approximations of) sets of terms are described (for e.g. regular tree automata) and constraints on the rewriting rules describing the dynamic of the system.

Verifying properties of the CONCERTO-D model itself may require the use of interactive theorem proving (ITP). There are ongoing efforts to develop a dedicated interactive theorem prover for Maude [11] as well as to translate Maude specifications [24] into existing ITP specification languages. The main properties we plan to prove interactively are that the semantics preserves well-formed instances and well-formed systems, as well as that the consistency of the communicated information between nodes is ensured.

Finally, thanks to CONCERTO-D, CONCERTO could be transformed into a choreographic language (as defined in [1]). In a choreographic approach CONCERTO would be the choreography language. A compilation process would then automatically generate the CONCERTO-D programs (i.e., local projections) on the nodes, and guarantee that required communications between nodes will be performed when required. The formalization may be used to prove that the set of CONCERTO-D programs yielded by that process has a behavior that corresponds to the behavior original centralized CONCERTO program.

## References

- [1] Davide Ancona et al. (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3, doi:10.1561/25000000031.
- [2] Paul Anderson & Herry Herry (2016): *A Formal Semantics for the SmartFrog Configuration Language*. *J. Netw. Syst. Manag.*, doi:10.1007/s10922-015-9351-y.
- [3] Yohan Boichut, Jacques Chabin & Pierre Réty (2019): *Towards more precise rewriting approximations*. *J. Comput. Syst. Sci.* 104, pp. 131–148, doi:10.1016/J.JCSS.2017.01.006.
- [4] Maverick Chardet, Hélène Coullon & Christian Pérez (2020): *Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto*. In: *20th IEEE/ACM International Symposium*

- on Cluster, Cloud and Internet Computing (CCGrid), IEEE, doi:10.1109/CCGrid49817.2020.00-59.
- [5] Maverick Chardet, Hélène Coullon, Christian Pérez, Dimitri Pertin, Charène Servantie & Simon Robillard (2020): *Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Maude*. hal: hal-02737859.
- [6] Maverick Chardet, Hélène Coullon & Simon Robillard (2021): *Toward safe and efficient reconfiguration with Concerto*. *Sci. Comput. Program.*, doi:10.1016/j.scico.2020.102582. hal: hal-03103714.
- [7] James Cheney, Laura Chiticariu & Wang Chiew Tan (2009): *Provenance in Databases: Why, How, and Where*. *Found. Trends Databases*, doi:10.1561/1900000006.
- [8] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer, Rubén Rubio & Carolyn Talcott (2024): *Maude manual (version 3.4)*. Available at <https://maude.lcc.uma.es/maude-manual/>.
- [9] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue & Simon Robillard (2023): *Component-Based Distributed Software Reconfiguration: A Verification-Oriented Survey*. *ACM Comput. Surv.*, doi:10.1145/3595376.
- [10] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli & Gianluigi Zavattaro (2014): *Aeolus: a Component Model for the Cloud*. *Information and Computation*, doi:10.1016/j.ic.2014.11.002.
- [11] F Durán, S Escobar, J Meseguer & J Sapina (2024): *An Inductive Theorem Prover for Maude Equational Theories*. Available at <https://nuitp.webs.upv.es/download/NuITP.pdf>.
- [12] Azadeh Farzan & José Meseguer (2006): *Partial Order Reduction for Rewriting Semantics of Programming Languages*. In: *Workshop on Rewriting Logic and its Applications (WRLA), ENTCS 176*, Elsevier, doi:10.1016/J.ENTCS.2007.06.008.
- [13] Weili Fu (2019): *Semantics and provenance of configuration programming language  $\mu$ Puppet*. Ph.D. thesis, University of Edinburgh, UK. Available at <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.798916>.
- [14] Weili Fu, Roly Perera, Paul Anderson & James Cheney (2017): *muPuppet: A Declarative Subset of the Puppet Configuration Language*. In: *31st European Conference on Object-Oriented Programming (ECOOP), LIPIcs*, doi:10.4230/LIPIcs.ECOOP.2017.12.
- [15] Thomas Genet (1998): *Decidable Approximations of Sets of Descendants and Sets of Normal Forms*. In: *Rewriting Techniques and Applications (RTA), LNCS 1379*, Springer, pp. 151–165, doi:10.1007/BFB0052368.
- [16] Herry Herry, Paul Anderson & Michael Rovatsos (2013): *Choreographing configuration changes*. In: *9th International Conference on Network and Service Management (CNSM 2013)*, doi:10.1109/CNSM.2013.6727828.
- [17] Khaled Khebbab, Nabil Hameurlain & Faiza Belala (2020): *A Maude-Based rewriting approach to model and verify Cloud/Fog self-adaptation and orchestration*. *J. Syst. Archit.*, doi:10.1016/J.SYSARC.2020.101821.
- [18] Khaled Khebbab, Nabil Hameurlain, Faiza Belala & Hamza Sahli (2019): *Formal modelling and verifying elasticity strategies in cloud systems*. *IET Softw.* 13(1), doi:10.1049/IET-SEN.2018.5030.

- [19] Vincent Lostanlen, Antoine Bernabeu, Jean-Luc Béchenec, Mikaël Briday, Sébastien Faucou & Mathieu Lagrange (2021): *Energy Efficiency is Not Enough: Towards a Batteryless Internet of Sounds*. In: *16th International Audio Mostly Conference*, doi:10.1145/3478384.3478408.
- [20] Peter Csaba Ölveczky (2017): *Designing Reliable Distributed Systems*. Springer, doi:10.1007/978-1-4471-6687-0.
- [21] Antoine Omond, H el ene Coullon, Issam Ra is & Otto Anshus (2023): *Leveraging Relay Nodes to Deploy and Update Services in a CPS with Sleeping Nodes*. In: *16th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*, IEEE, doi:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00102. hal: hal-04372320.
- [22] Antoine Omond, Issam Ra is & H el ene Coullon (2023): *Evaluating the energy consumption of adaptation tasks for a CPS in the Arctic Tundra*. In: *19th IEEE International Conference on Green Computing and Communications (GreenCom)*, IEEE, doi:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00122. hal: hal-04372340.
- [23] Jolan Philippe, Antoine Omond, H el ene Coullon, Charles Prud'Homme & Issam Ra is (2024): *Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study*. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE. hal: hal-04457484.
- [24] Rub en Rubio & Adri an Riesco (2022): *Theorem proving for Maude specifications using Lean*. In: *International Conference on Formal Engineering Methods*, Springer, pp. 263–280, doi:10.1007/978-3-031-17244-1\_16.
- [25] Daniel Sokolowski, Pascal Weisenburger & Guido Salvaneschi (2021): *Automating Serverless Deployments for DevOps Organizations*. In: *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, doi:10.1145/3468264.3468575.
- [26] Karoline Wild, Uwe Breitenb ucher, K alman K epes, Frank Leymann & Benjamin Weder (2020): *Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models*. In: *Advanced Information Systems Engineering (CAiSE)*, Springer, doi:10.1007/978-3-030-49435-3\_2.
- [27] Geunyeol Yu, Seunghyun Chae, Kyungmin Bae & Sungkun Moon (2024): *Formal Specification of Trusted Execution Environment APIs*. In: *Fundamental Approaches to Software Engineering*, Springer Nature Switzerland, doi:10.1007/978-3-031-57259-3\_5.