Generating Distributed Programs from Event-B Models

Horatiu Cirstea
LORIA UMR 7503
Université de Lorraine
Vandœuvre-lès-Nancy, France
horatiu.cirstea@loria.fr

Alexis Grall
LORIA UMR 7503
Université de Lorraine
Vandœuvre-lès-Nancy, France
alexis.grall@loria.fr

Dominique Méry
LORIA UMR 7503
Telecom Nancy, Université de Lorraine
Vandœuvre-lès-Nancy, France
dominique.mery@loria.fr

Distributed algorithms offer challenges in checking that they meet their specifications. Verification techniques can be extended to deal with the verification of safety properties of distributed algorithms. In this paper, we present an approach for combining correct-by-construction approaches and transformations of formal models (EVENT-B) into programs (DISTALGO) to address the design of verified distributed programs. We define a subset LB (Local EVENT-B) of the EVENT-B modelling language restricted to events modelling the classical actions of distributed programs as internal or local computations, sending messages and receiving messages. We define then transformations of the various elements of the LB language into DISTALGO programs. The general methodology consists in starting from a statement of the problem to program and then progressively producing an LB model obtained after several refinement steps of the initial LB model. The derivation of the LB model is not described in the current paper and has already been addressed in other works. The transformation of LB models into DISTALGO programs is illustrated through a simple example. The refinement process and the soundness of the transformation allow one to produce correct-by-construction distributed programs.

1 Introduction

EVENT-B is a formal modelling language developed by Abrial [1] offering key features such as the use of set theory as a data modelling notation, the use of refinement to relate system models at different abstraction levels and the use of mathematical proofs to verify consistency between refinement levels. Moreover, the language is supported by the environment RODIN[2] which is extensible through the mechanism of plugin. Previous works [3, 14, 10, 11] illustrate the correct-by-construction design of distributed algorithms using EVENT-B models and refinements; those works show that at an adequate level of concretization of models, one can derive a distributed algorithm in a pseudo algorithmic notation. However, the derivation of concrete EVENT-B models requires to develop a methodology related to a given class of problems. For instance, we have produced a plugin EB2RC [12, 5] which automatically generates a recursive algorithm from an EVENT-B model derived by analysis of a problem such as Floyd's algorithm, or search algorithms, or sorting algorithms. The transformation of an EVENT-B model into a recursive algorithm was based on the definition of a class of (concrete) EVENT-B models satisfying constraints making the transformation automatic.

In the current paper, we study the systematic transformation of concrete EVENT-B models into the DISTALGO [8] programming language. In fact, the design of a distributed algorithm using the correct-by-construction approach starts by expressing the required computations in a very abstract EVENT-B model (AM) and then progressively refining the model into a final concrete model (CM) very close to an algorithmic expression of the distributed algorithm. The main advantage of such a refinement-based process is the preservation of safety properties of the different models: the refinement is checked by discharging a list of proof obligations. We do not describe the process for developing the model CM which is supposed to be a local EVENT-B model and which could be translated into an algorithmic

L. Fribourg and M. Heizmann (Eds.): VPT/HCVS 2020
EPTCS ??, 2020, pp. 1–15, doi:10.4204/EPTCS.??.??

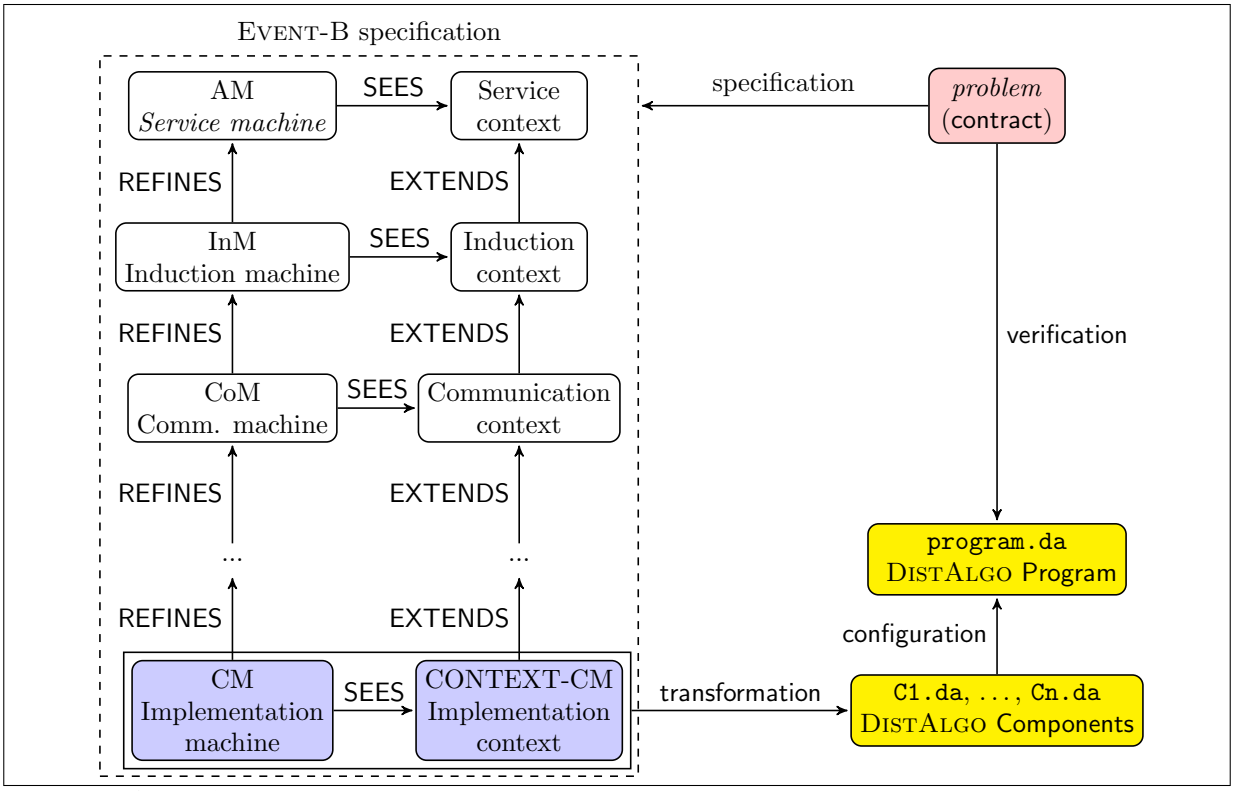


Figure 1: The global methodology for correct-by-construction distributed algorithms.

distributed notation. We focus on the transformations required for obtaining a DISTALGO program from a local EVENT-B model as indicated in Figure 1: the program `program.da` is generated from CM and CONTEXT-CM. We will not provide the proof of correctness of the translation but we will give enough details for trusting it. The proof will be given in a future work.

An overview of the integrated development framework Figure 1 provides an overview of our integrated development framework for refinement-based program verification of distributed algorithms. The general methodology starts by stating the *problem* to solve by listing the requirements (*i.e.* the contract) attached to the problem; the requirements can be either expressed in a formal language or in an informal textual language. One has then to specify the EVENT-B machine AM translating the main requirements for the given problem. Then a list of formal EVENT-B refined machines are produced to obtain a final EVENT-B machine and context, CM and CM-CONTEXT. Finally, the translations of these final context and machine into DISTALGO components and programs are generated in two main steps: the automatic compilation of CM and CM-CONTEXT into a DISTALGO program, and the manual tuning of the obtained DISTALGO components (if some configurations were not specified in the model).

The refinement block (with nodes AM, CONTEXT-AM, CM and CONTEXT-CM) in Figure 1, illustrates the mechanism for deriving machines via refinement. The result of the refinement is the EVENT-B machine CM, which contains the refined events and the proof obligations that must be discharged in order to prove that the refinement is correct.

Transformations of this EVENT-B machine CM into a DISTALGO program is based on the extraction of information concerning the network and the process classes from the context CONTEXT-CM, and on the analysis of the localization of the different variables. The events of CM are supposed to be local

which means that they are using only local instances of variables. For instance, *pc* will be a local variable with an instance $pc(p)$ for the process p. We will define precisely the localization process from the code of EVENT-B models. Finally, some constants whose values are not defined in the context are instantiated during the configuration phase.

Related work We described a simple extension of the *call-as-event* paradigm [9, 12] to handle the design of concurrent programs in the *coordination*-based approach but we do not target a specific programming language as DISTALGO. The EB2ALL (http://eb2all.loria.fr) framework provides a list of transformations of EVENT-B models into classical programming languages (C, C++, Java, ...) but it does not consider distributed algorithms. The current work can be considered as adding a new target programming language but with the target of a distributed program like it was proposed in Visidia (http://visidia.labri.fr) together with EVENT-B with the plugin B2VISIDIA relating the local EVENT-B model and a VISIDIA program. However, the VISIDIA approach addresses distributed programs defined as set of rewriting rules of graphs, which is less concrete and effective than DISTALGO programs. Code generation from classical B models are supported by the Atelier B (http://www.atelierb.eu) tools but those transformations do not consider distributed programming models. Atelier B supports code generation into Ada, C, C++. Moreover, it is defined over Classical B software models restricted to the B0 language which is a computable subset of the B language but without communications features. An EventB2Java [4] tool for RODIN has been developed for translating any EVENT-B specification into (sequential) JML or Java code. Finally, a Tasking EVENT-B [7] for RODIN extends the EVENT-B language to provide features for specifying concurrent multi-tasking systems. A model is decomposed into several tasking machines which schedule and perform tasks involving shared machines which correspond to protected resources accessed by tasking machines. The plugin provides a tool support for translating a tasking specification into ADA code. The generated programs are not distributed ones and consider only a subclass of the ADA language. Our work focuses on generating DISTALGO programs from local EVENT-B models and provides a way to preserve powerfull safety properties from the local models.

Overview of the paper In the next section, we briefly present the two languages EVENT-B and DISTALGO. Section 3 shows how distributed programs can be modelled in the sub-language called LB for Local EVENT-B. Finally, in Section 4 we define the transformation of LB models into DISTALGO programs. Our paper then concludes with the results and future work. A more detailed description of the translation as well as the complete definition of the LB models and of the DISTALGO program of our example are available in [6].

2 Modelling Distributed Programs

We describe briefly in this section the EVENT-B modelling language and the DISTALGO programming language. We will show later on how the corresponding specifications are implemented following the methodology described in Figure 1.

2.1 The Modelling Framework: EVENT-B

The EVENT-B language [1] contains two main components, the *context* which describes the static properties of a system using *carrier sets* s, *constants* c, *axioms* $A(s,c)$ and *theorems* $T_c(s,c)$, and the *machine* which describes behavioural properties of a system using *variables* v, *invariants* $I(s,c,v)$, *theorems* $T_m(s,c,v)$, *variants* $V(s,c,v)$ and *events* evt. A context can be extended by another context, a machine can be refined by another machine and a machine can use the *sees* relation to include other contexts.

An EVENT-B machine defines a set of *state variables Var*, taking their values in a set *Val*, and possibly modified by a set of *events Events*. A set of invariants $I_i(s,c,v)$ contains typing information and required safety properties that must be satisfied by the defined system. Each event $evt =$ ANY x WHERE $G_{evt}(s,c,v,x)$ THEN $v : |P_{evt}(s,c,v,x,v')$ END is composed of parameter(s) x, guard(s) $G_{evt}(s,c,v,x)$ and action(s) $v : |P_{evt}(s,c,v,x,v')$. Unprimed variables refer to the state variables before the event occurs and primed variables refer to the state variables after observation of the event. The *before-after* predicate $BA(evt)(s,c,v,v')$ for evt is defined by $(\exists\ x \cdot G_{evt}(s,c,v,x) \wedge P_{evt}(s,c,v,x,v'))$. A state st of a machine is an element of the set $St_{EB} = Var \rightarrow Val$. The value of a variable $u \in Var$ in the state st is $st(u)$ and is denoted $st[\![u]\!]$. The notation $[\![.]\!]$ is extended to the list of variables $v = (v_1,\ldots,v_n)$ by stating $st[\![(v_1,\ldots,v_n)]\!] = (st(v_1),\ldots,st(v_n))$. Finally, $[\![.]\!]$ is extended to handle (arithmetical, boolean) expressions by inductive definition: $st[\![exp(v)]\!] = exp(st[\![v]\!]/v)$. For two states st_1, st_2 and an expression $exp(v,v')$ on primed variables v' and unprimed variables v, $st_1[\![exp(v,v')]\!]st_2$ is defined by $exp(st_1[\![v]\!]/v, st_2[\![v]\!]/v')$ the value of expression exp where unprimed variables are evaluated in state st_1 and primed variables are evaluated in state st_2. When an event evt is observed between two states st_1 and st_2, then $st_1[\![BA(evt)(s,c,v,v')]\!]st_2$ holds. In this paper, we write deterministic *actions* of the form $v := E(s,c,v,x)$ that are equivalent to $v : |v' = E(s,c,v,x)$. Using the transition relation over the set of states, we can define state properties as safety or invariance and traces properties.

The EVENT-B modelling language supports the *correct-by-construction* approach to design an abstract model and a series of refined models for developing any large and complex system. RODIN [2] is an integrated development environment for the EVENT-B modelling language based on Eclipse. It includes project management, stepwise model development, proof assistance, model checking, animation and automatic code generation.

2.2 The DISTALGO Distributed Programming Language

DISTALGO [8] is a programming language used to develop distributed algorithms by providing high level programming mechanisms such as communication primitives for the exchange of messages between a set of processes.

A DISTALGO program is composed of several process classes managed by a `main` module (see Example 4.1). A process class is made of a **`setup`** method which initializes the class attributes, a **`run`** method for carrying out the main execution flow, several **`receive`** methods for handling the reception of messages and other user defined methods that may be called by the **`run`** method. For each process class `PC`, the `main` module uses a statement of the form `pset=`**`new`**`(PC,`**`num`**`=n)` to build the set `pset` of `n` processes running the algorithm specified for `PC`. The **`setup`** method is called for the processes in each class and the **`start`** directive is eventually used to trigger the **`run`** method of all processes.

A process can send a `message` to another process `q` with a statement **`send`**`(message,`**`to`**`=q)`. When a message arrives at the receiving process, it is put in a message queue waiting to be received by the process. To receive messages, the process control flow must be at a yield point and this enables the receiving of every message in the message queue. When a message is received, the **`receive`** message handlers matching the message are executed. A yield point is a labeled statement `--l` **`if await`** `b1:s1` **`elif`** `b2:s2` **`elif`** ... **`elif`** `bn:sn` waiting for one of the conditions `bi` to hold in order to execute the corresponding branch `si`. The history of sent and received messages can be accessed in DISTALGO using the **`sent`** and **`received`** primitives. A graphical representation of the message exchanges is given in Figure 2. Since DISTALGO is implemented as a PYTHON module all the data structures and primitives of the latter can be used. In our translation we use, in particular, **`list`**s, sometimes built using the function **`range`** which creates a list interval of integers, and **`set`**s,

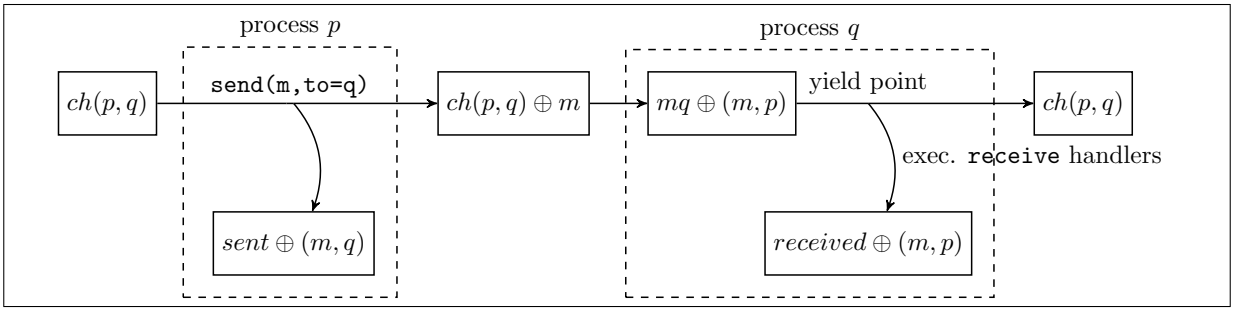


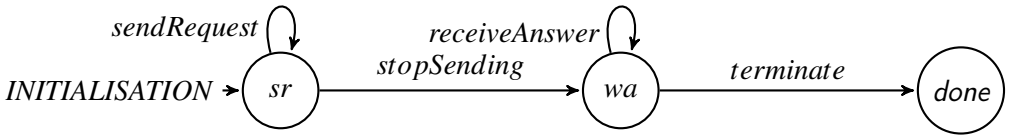
Figure 2: Comunications in DISTALGO: the communication channels *ch*, as well as the message queues *mq* cannot be accessed explicitly in DISTALGO; only the sent and received messages can be accessed using the **`sent`** and **`received`** primitives.

which can be built from a list or using the function **`setof`**$(expr(x_1,\ldots,x_n)$**`,`** x_1 **`in`** $S_1,\ldots,x_n$ **`in`** S_n**`,`** $pred(x_1,\ldots,x_n))$ which is a set comprehension with *expr*essions built out of elements in the sets $S_1,\ldots,S_n$ and satisfying a *pred*icate. PYTHON dictionaries are also used; these can be updated with the elements of another dictionary using the method **`update`** and cloned with the function **`deepcopy`** which copies an object and the objects it contains recursively. The DISTALGO boolean functions **`each`** and **`some`** acting as a universal quantifier and an existential quantifier respectively, are also used.

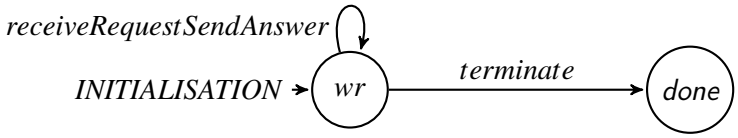
3 Modelling Distributed Algorithms in EVENT-B

We use the modelling technique of G. Tel [13] and express a distributed algorithm as a set of local algorithms, each local algorithm being able to do an internal action, or to send a message, or to receive a message. The final context CONTEXT-CM and machine CM in Figure 1 model such a distributed algorithm using a subset of the modelling language EVENT-B, denoted LB (Local EVENT-B). We use the simple distributed algorithm introduced in Example 3.1 to explain the methodology for modelling algorithms following Tel's technique and the restrictions imposed on LB.

Example 3.1. *We consider a distributed algorithm where each process q in a set of processes Q sends its stored value to a central process p who previously made the corresponding requests. The local algorithm of the requester process p has three states:*



While in state sr, p sends a request to each of the processes in Q and moves to state wa, when all requests have been sent. In the state wa, it awaits for answers from the processes in Q. When all answers are received, process p has terminated its local algorithm and moves to state done.



Each process in Q is initially in a state wr in which it waits for a request from p and moves to state done *after receiving the request and sending its stored value.*

CONTEXT CONTEXT-CM
EXTENDS C00
SETS
Nodes States Messages // General sets
MessagePrefixes // Algorithm specific sets
CONSTANTS
network // The topology (general)
Channels emptyChannel sent received inChannel // Communication channels (general)
send receive lose // Communication primitives (general)
P p Q // Process classes and processes (specific to the algorithm)
request answer // Algorithm specific constants
availableResources // Algorithm specific constant
sr wa wr done // Process states (specific to the algorithm except for done, general)
AXIOMS
Nodes: $partition(\mathsf{Nodes}, P, Q)$ // Partition of the set of processes
P: $partition(P, \{p\})$ // Partition of the classes of processes
network_typing: $\mathsf{network} \in \mathsf{Nodes} \to \mathbb{P}(\mathsf{Nodes})$ // Network specification
network_value: $\mathsf{network} = \{proc \cdot proc \in P | proc \mapsto Q\} \cup \{proc \cdot proc \in Q | proc \mapsto \{p\}\}$
// States of the processes
States: $partition(\mathsf{States}, \{sr\}, \{wa\}, \{wr\}, \{done\})$
// Communication channels
Channels: $\mathsf{Channels} = \mathsf{Nodes} \times \mathsf{Nodes} \to (\mathsf{Messages} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N})$
// Algorithm specific constants (types of exchanged messages, process resources)
MessagePrefixes: $partition(MessagePrefixes, \{request\}, \{answer\})$ //@P@Q
availableResources_typing: $availableResources \in Q \to \mathbb{N}$
// Communication axioms (general to all algorithms)
END

Figure 3: Sets and constants for the Example 3.1

The general architecture of the distributed algorithm (processes, topology, channels, communications) is specified in the EVENT-B context CONTEXT-CM while the list of events of the machine CM induces the specifications of the local algorithms as labelled transition systems. In the sequel, the pair CM and CONTEXT-CM defining the LB distributed model is called simply CM.

3.1 Defining the General Architecture of the Distributed Program

Sets, constants and corresponding axioms defined in the context of a distributed model are of two categories: the general ones present in the context of any algorithm and those which are specific to the modeled algorithms. The most important elements of the context corresponding to the algorithm described in Example 3.1 is given in Figure 3:

For every distributed algorithm, the set Nodes of processes is defined axiomatically as a partition into process classes, the processes of each class featuring a similar local algorithm:

$$\texttt{Nodes} : partition(\mathsf{Nodes}, PCl_1, \ldots, PCl_n)$$

For each process class PCl_i one can enumerate explicitly its processes using an axiom

$$\texttt{PCl}_i : partition(PCl_i, \{proc_1\}, \ldots, \{proc_m\})$$

These partitions depend of course on the specific algorithm modeled and, in general, the processes are not explicitly enumerated.

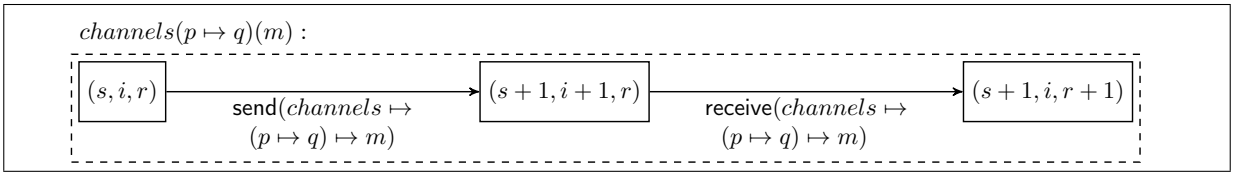


Figure 4: We suppose that m has been already sent, resp. received, s, resp. r times, and that i copies are in the channel: $channels(p \mapsto q)(m) = (s,i,r)$. When we *send* the message m we increment the sent counter and the number of messages in the channel; when we *receive* it we increment the received counter and decrement the number of messages in the channel.

The topology, denoted network, is specified by a function associating to each process its neighbours: $\mathsf{network} \in \mathsf{Nodes} \to \mathbb{P}(\mathsf{Nodes})$. The concrete definition of the topology specific to the distributed algorithm under consideration is specified using an axiom whose general form should be

$$\texttt{network\_value} : \mathsf{network} = \{proc \cdot proc \in PCl_1 | proc \mapsto expr_1\} \cup \\ \cdots \cup \{proc \cdot proc \in PCl_n | proc \mapsto expr_n\}$$

In the example, the topology is defined as a star with the process p in the center.

As we will see later on, the *control states* in States are used for structuring the observation of events in the local algorithms. The set of all possible control states of all processes is defined as a partition by an axiom `States`.

The context should also define a constant Channels modelling the set of all possible values of communication channels between processes and the set Messages of messages exchanged through these channels. The current state of a channel between two processes is defined as a multiset, corresponding to the messages that were sent, received and in transition, *i.e.* sent but not yet received or lost. For instance, sent(channel,p,q,mes) is returning how many times the message mes has been sent by p to q. Hence, for each channel we can retrieve the exchanged messages using the functions sent, received and inChannel of type $\mathsf{Channels} \times (\mathsf{Nodes} \times \mathsf{Nodes}) \times \mathsf{Messages} \to \mathbb{N}$. The functions send, receive and lose of type $\mathsf{Channels} \times (\mathsf{Nodes} \times \mathsf{Nodes}) \times \mathsf{Messages} \to \mathsf{Channels}$ describe the transformation of a channel between two processes (*i.e.* adding or removing a message) when one operation (send, receive or lose) is observed. More precisely, we consider that the channels do not preserve the order in which messages are sent, and sending a message consists in incrementing the inChannel part and the sent part of a channel between two processes. The evolution of the channel between two processes p and q concerning a message m is modeled in LB by the variable $channels(p \mapsto q)(m)$, as depicted in Figure 4. This variable models the channel *ch* and the message queues *mq* as well as the **`sent`** and **`received`** DISTALGO primitives described in Section 2.2 (Figure 2); the transfer of the message from the channel to the message queue which is builtin in DISTALGO is not explicitly modeled in LB.

Sets and constants mentioned above should be present in the context of any distributed algorithm modeled in LB. Other enumerated sets defined necessarily as the disjoint union of singletons using the *partition* construct as well as constants specific to the modeled algorithm can be defined in the context. The type of such a constant *cst* is defined by an axiom of name `cst_typing` while its value may be defined by an axiom of name `cst_value`. For instance, in our example we require that each process of Q has a non-negative integer $availableResources$. For the purpose of our example, we also define an enumerated set $MessagePrefixes$ consisting of *request* and *answer* which correspond to the two kinds of messages exchanged between the processes.

Annotations of the form $@PCl_1 \ldots @PCl_n$ are used to specify that the annotated elements are *local* to the processes in the corresponding class. This is done either in the axiom `cst_typing` to indicate the

MACHINE CM
SEES CONTEXT-CM
VARIABLES
channels pc result
INVARIANTS
channels_typing: $channels \in \mathsf{Channels}$
pc_typing: $pc \in \mathsf{Nodes} \rightarrow STATES$
result_typing: $result \in P \rightarrow (\mathsf{Nodes} \nrightarrow \mathbb{N})$
EVENTS
Initialisation $\widehat{=}$
begin
act1: $channels := \mathsf{emptyChannel}$
act2: $pc := \{proc \cdot proc \in P | proc \mapsto sr\} \cup \{proc \cdot proc \in Q | proc \mapsto wr\}$
act3: $result := \{proc \cdot proc \in P | proc \mapsto \varnothing\}$
end
END

Figure 5: Variables, invariants and initialisation for the Example 3.1

process classes concerned by the constant *cst*, or in the axiom `S` specifying the *partition* of an enumerated set *S* to indicate the process classes concerned by the elements of the set; the latter applies for the axiom `MessagePrefixes` in our example.

Definition 1 (Local constants)**.** *Given a process $proc \in PCl$ and a constant cst, we say that cst is* local *to proc when it is a function whose evaluation depends on proc (i.e. of type $PCl \rightarrow cstType$ or $\mathsf{Nodes} \rightarrow cstType$) or when it is (an element of an enumerated set whose partition is) annotated by @PCl. We denote $LC(PCl)$ the set of local constants for (the processes of) PCl.*

In our example, the elements of $MessagePrefixes$ are local to both p and $q \in Q$, $\mathsf{network}(r)$ is local to any $r \in P \cup Q$ and $availableResources(q)$ is local to any $q \in Q$.

3.2 Producing Local Algorithms as State Machines

We specify now the algorithms for the set of processes. Recall that all processes in a process class run the same algorithm, the one associated to the class.

The machine CM in Figure 1 declares the types and initializes the local variables of each process class of the distributed algorithm. The variable *pc* identifying the current state of each local algorithm and the communication variable *channels* of type Channels are defined for any algorithm, the definition of other variables depends on the modeled algorithm. The variables together with their initialization in the machine CM modelling the algorithm described in Example 3.1 is given in Figure 5.

Definition 2 (Local variables)**.** *Given a process $proc \in PCl$ and a variable var, we say that var is* local *to proc when it is a function whose evaluation depends on proc (i.e. of type $PCl \rightarrow varType$ or $\mathsf{Nodes} \rightarrow varType$). We denote $LV(PCl)$ the set of local variables for (the processes of) PCl and $LV(proc) = LV(PCl)$ the set of local variables for a process $proc \in PCl$.*

Every variable is initialised as usual by a deterministic assignment which specifies the value of the variable for the processes of each concerned class using statements of the form $\{proc \cdot proc \in PCl | proc \mapsto expr\}$ with the expression *expr* using *only local constants and variables of the process proc*. For example, the algorithm specific variable *result* concerns only the process $p \in P$ with the expression $result(p)$ corresponding to the values received from the processes of Q.

Events of the machine CM correspond to state transitions of the local algorithms of the processes. Process events are observed for a specific process of a process class.

Definition 3 (LB events, states)**.** *An* event *evt in LB is such that*

- *it features one parameter proc typed by a guard* $proc \in PCl$ *with* $PCl \in \mathsf{Nodes}$*;*
- *all actions are assignments* $x(proc) := pExpr$ *or* $channels := cExpr$ *with cExpr of the form*
  - $send(channels \mapsto (proc \mapsto pExpr) \mapsto mExpr)$
  - $receive(channels \mapsto (pExpr \mapsto proc) \mapsto mExpr)$

  If the event contains an action send*, resp.* receive*, then it is called a* send event*, resp* receive event*; it is called* internal *otherwise.*
- *it features a guard* $pc(proc) = st$ *which specifies the event is enabled in state* $st \in \mathsf{States}$*;*
- *it features a typing guard* $t \in tExpr$ *for each parameter;*
- *if it is an internal or a send event, it can feature general guards gExpr or guards of the form*
  - $\mathsf{sent}(channels \mapsto (proc \mapsto pExpr) \mapsto mExpr) = nExpr$
  - $\mathsf{received}(channels \mapsto (pExpr \mapsto proc) \mapsto mExpr) = nExpr$
- *if it is a receive event, it can feature matching guards for the parameters source and message which should be always present for such an event;*

with all expressions $tExpr, gExpr, pExpr, mExpr, nExpr$ *built over local constants, local variables, parameters of the event, literal integers and booleans.*

We say that the event is observed for a process proc and moreover, that is observable *in state st. We denote by* $\mathsf{Events}(PCl)$ *and* $\mathsf{Events}(proc)$ *the set of local events for the set of processes PCl and for the process proc respectively, and by* $\mathsf{Events}(PCl, st)$ *and* $\mathsf{Events}(proc, st)$ *the events in* $\mathsf{Events}(PCl)$ *and* $\mathsf{Events}(proc)$ *respectively, that are observable in state st.*

Given a process class PCl, the set of states of processes *of PCl, denoted by* $\mathsf{StatesSet}(PCl)$*, consists of the states st such that there exists a parameter proc and a guard* $pc(proc) = st$ *for some event evt* $\in \mathsf{Events}(PCl)$.

The events of the EVENT-B machine CM corresponding to the algorithm for the process p introduced in Example 3.1 are presented in Figure 6. Note that *sendRequest* is a *send* event and does not modify $pc(p)$, $stopSending$ is an internal event with a guard verifying if p has sent a request to all its neighbours, *receiveAnswer* is a receive event for answers to the requests (the internal event $terminate$ not presented here verifies that an answer has been received from every neighbour and terminates the local algorithm of process p). The processes of Q feature similar events: we have a receive and a send event which model respectively the reception of requests from p and the dispatching of an answer. We also have an internal event for terminating the local algorithm of a process of Q once it has sent the answer.

4 Translation in DISTALGO

A pair of a machine and a context compliant with the form described in the previous section is translated towards a DISTALGO program composed of a set of process classes. The main function and the process class definitions are generated from the (axioms in the) context while the process class methods are generated from the (invariants and events in the) machine.

Event *sendRequest* $\widehat{=}$
  any *proc q*
  where
    grd1: $proc \in P$
    grd2: $q \in \mathsf{network}(proc)$
    grd3: $pc(proc) = sr$
    grd4: $\mathsf{sent}(channels \mapsto (proc \mapsto q) \mapsto request) = 0$
  then
    act1: $channels := \mathsf{send}(channels \mapsto (proc \mapsto q) \mapsto request)$
  end

Event *stopSending* $\widehat{=}$
  any *proc*
  where
    grd1: $proc \in P$
    grd2: $pc(proc) = sr$
    grd3: $\forall q \cdot (q \in \mathsf{network}(proc) \Rightarrow \mathsf{sent}(channels \mapsto (proc \mapsto q) \mapsto request) > 0)$
  then
    act1: $pc(proc) := wa$
  end

Event *receiveAnswer* $\widehat{=}$
  any *proc source message r*
  where
    grd1: $proc \in P$
    grd2: $source \in \mathsf{Nodes}$
    grd3: $message \in \mathsf{Messages}$
    grd4: $r \in \mathbb{Z}$
    grd5: $pc(proc) = wa$
    grd7: $message = answer \mapsto r$
  then
    act1: $result(proc) := result(proc) \lhd \{source \mapsto r\}$
    act2: $channels := \mathsf{receive}(channels \mapsto (source \mapsto proc) \mapsto message)$
  end

Figure 6: Events for the Example 3.1

4.1 Translation of Expressions

We first define a translation function, denoted $\mathscr{T}_{\overrightarrow{x}}()$, which transforms a well-formed EVENT-B expression (or predicate) *expr* into the corresponding DISTALGO code $\mathscr{T}_{\overrightarrow{x}}(expr)$ *w.r.t.* a set $\overrightarrow{x}$ of bound variables.

Arithmetic expressions are translated in an obvious way. Set expressions are also translated straightforwardly with sets built using the PYTHON primitives **`set`** and **`setof`**, and the set operations encoded by corresponding PYTHON operations. Finite functions are translated using PYTHON dictionaries. Predicates are translated into boolean expressions with the quantifiers encoded using the **`each`** and **`some`** DISTALGO functions.

The action for the sending of a message is translated using the DISTALGO function **`send`**:

$\mathscr{T}_{\overrightarrow{x}}(channels := \mathsf{send}(channels \mapsto (proc \mapsto dest) \mapsto msg)) \triangleq$ **`send`**`(`$\mathscr{T}_{\overrightarrow{x}}(msg)$`, to=`$\mathscr{T}_{\overrightarrow{x}}(dest)$`)`

Note that *channels* and *proc* are not present in the resulting code since *channels* is implicit in DISTALGO and *proc* corresponds to the process executing the **`send`** statement.

The sent and received events defined in EVENT-B are translated as DISTALGO queries on message history. DISTALGO allows patterns inside queries on messages and any plain variable `x` in such a query

is considered free and is potentially instantiated by a value following a successful matching. To indicate that a variable is bound in a query it should be of the form _x. We consider thus the translation function $\mathscr{T}^b_{\overrightarrow{x}}()$ which is defined exactly as the function $\mathscr{T}_{\overrightarrow{x}}()$ except for variables for which we have

$$\mathscr{T}^b_{\overrightarrow{x}}(x) \triangleq \texttt{\_x} \qquad \text{when } x \in \overrightarrow{x}$$
$$\mathscr{T}^b_{\overrightarrow{x}}(x) \triangleq \mathscr{T}_{\overrightarrow{x}}(x) \qquad \text{when } x \notin \overrightarrow{x}$$

The two expressions involving *sent* or *received* events supported by our approach are translated using

$\mathscr{T}_{\overrightarrow{x}}(sent(channels \mapsto (proc \mapsto dest) \mapsto msg) > 0) \triangleq$ **some** (**sent** ($\mathscr{T}^b_{\overrightarrow{x}}(msg)$, **to**=$\mathscr{T}^b_{\overrightarrow{x}}(dest)$))

$\mathscr{T}_{\overrightarrow{x}}(received(channels \mapsto (source \mapsto proc) \mapsto msg) > 0) \triangleq$ **some** (**received** ($\mathscr{T}^b_{\overrightarrow{x}}(msg)$, **from_**=$\mathscr{T}^b_{\overrightarrow{x}}(source)$))

4.2 Generation of the Main Function

The main function of the generated DISTALGO program defines different local constants as well as the different processes to execute, and starts the local algorithms of all the processes. This function is generated using exclusively the context CONTEXT-CM and more precisely, only the axioms of the context. The (identifiers of these) axioms should thus respect the rules given in Section 3.1 and the names of the variables and constants are inferred correspondingly.

The code of the main function contains a fixed part independent of the algorithm and specifying, for example, the behaviour of the communication channels. We omit here the fixed part and the various imports that might be needed and focus on the part generated from the EVENT-B model.

The axiom `Nodes` allows us to infer the set $\{PCl_1, \ldots, PCl_n\}$ of process classes and to generate, for each process class, a fresh variable $\texttt{PClSet}_i$ corresponding to the set of processes in PCl_i. We can thus initialize each variable $\texttt{PClSet}_i$ as a set of $\texttt{NPCl}_i$ processes of class $\texttt{PCl}_i$ (generated later on) and then, the variable `Nodes` corresponding to the set of all processes:

$\texttt{PClSet}_1$ = **new**($\texttt{PCl}_1$, **num**=$\texttt{NPCl}_1$)
...
$\texttt{PClSet}_n$ = **new**($\texttt{PCl}_n$, **num**=$\texttt{NPCl}_n$)
Nodes = **set**.union($\texttt{PClSet}_1$,...,$\texttt{PClSet}_n$)

We use the axioms $\texttt{PCl}_i$ to initialize the variables for each set and $\texttt{NPCl}_i$ to the cardinal of the corresponding set ($\texttt{NPCl}_i$ should be configured manually if the axiom is not present):

(proc1,...,procm) = **list**($\texttt{PClSet}_i$)
$\texttt{NPCl}_i = |\{proc_1, \ldots, proc_m\}|$

Starting from the axiom `network_value` we generate the map `network` for the topology

network = {proc:$\mathscr{T}_{\varnothing}(expr_1)$ **for** proc **in** $\texttt{PClSet}_1$}
network.**update**({proc:$\mathscr{T}_{\varnothing}(expr_2)$ **for** proc **in** $\texttt{PClSet}_2$})
...
network.**update**({proc:$\mathscr{T}_{\varnothing}(expr_n)$ **for** proc **in** $\texttt{PClSet}_n$})

In fact, for each (local) constant *cst* in the context which is a function ($cst \in PCl \rightarrow cstType$) and features an axiom `cst_value`: $cst = \{proc \cdot proc \in PCl | proc \mapsto expr\}$ for some *PCl* we generate an initialization:

cst = {proc:$\mathscr{T}_{\varnothing}(expr)$ **for** proc **in** PClSet}

For each process class PCl_i the following code is generated for the initialisation:

for proc **in** $\texttt{PClSet}_i$:
  setup({proc}, (cst_1[proc],...,cst_n[proc])

with $\{cst_1, \ldots, cst_n\} = \mathsf{LC}(PCl_i)$.

Finally, the processes are executed with the DISTALGO command **start**(Nodes).

Example 4.1. *Given the context in Section 3.1 the following main function is generated.*

```
def main():
    NP = 1
    NQ = #NQ - to be configured

    PSet = new(P, num=NP)
    (p,) = list(PSet)
    QSet = new(Q, num=NQ)

    Nodes = set.union(PSet, QSet)
    network = {proc:QSet for proc in PSet}
    network.update({q:{p} for q in QSet})
    availableResources = #availableResources - to be configured

    for proc in PSet:
        setup({proc}, (network[proc],))
    for proc in QSet:
        setup({proc}, (network[proc], availableResources[proc]))
    start(Nodes)
```

In the same time with the main class we generate the code corresponding to the enumerated sets defined in the context using an axiom $\mathtt{S} : partition(S, \{el_1\}, \{el_2\}, \ldots)$ like, *e.g.*, $MessagePrefixes$. For all these sets we generate a separate file (imported when needed) containing the corresponding code:

```
class S(Enum):
    el1 = "el1"
    el2 = "el2"
    ...
```

The access to the elements of the respective set is done as expected: $\mathscr{T}^{l}_{\vec{x}}(el_i) \triangleq \mathtt{S.}el_i$, for any member el_i of the enumerated set.

4.3 Generation of the Process Classes

For each process class *PCl* we generate a DISTALGO process class `PCl` featuring the necessary methods.

For the purpose of the translations presented in this section we consider the function $\mathscr{T}^{l}_{\vec{x}}()$ which behaves exactly like $\mathscr{T}_{\vec{x}}()$ except for one case: $\mathscr{T}^{l}_{\vec{x}}(f(proc)) \triangleq \mathtt{self.f}$ when $f \in \mathsf{LV}(PCl) \cup \mathsf{LC}(PCl)$, $proc \in PCl$.

The **setup** method gets the values of the local constants as parameters and initializes the local variables. We have thus for each process class *PCl* in the context a DISTALGO class:

```
class PCl( process ) :
    def setup(cst1, ..., cstn) :
        self.var1 = T^l_∅(expr1)
        ⋮
        self.varm = T^l_∅(exprm)
```

with $\{cst_1,\ldots,cst_n\} = \mathsf{LC}(PCl)$, $\{var_1,\ldots,var_m\} = \mathsf{LV}(PCl)$, and $\{expr_1,\ldots,expr_m\}$ the corresponding expressions $var_i := \{proc \cdot proc \in PCl | proc \mapsto expr_i\}$ in the **Initialisation** section of the machine. For a variable *var* (resp. constant *cst*), the translation of $var(proc)$ (resp. $cst(proc)$) is then `self.var` (resp. `self.cst`).

For each state $st \in \mathsf{StatesSet}(PCl)$ a method `st` describing the behavior on reception of an event observable in state *st* is generated as explained below. The **run** method defining the control flow of the program for the respective process consists of a loop which calls at each iteration the method `st` corresponding to the current value of `self.pc` and terminates when `self.pc` reaches the termination state done. When $\mathsf{StatesSet}(PCl) = \{st_1,\ldots,st_n\}$ the following code is generated:

```
def run():
    stateFunctions = {"st1":st1,...,"stn":stn}
    while(self.pc!=done):
    stateFunctions[self.pc]()
```

Given an event $evt \in \mathsf{Events}(PCl)$ we denote by $\mathsf{Guards}(evt)$ the set of its guards, by $\mathsf{Actions}(evt)$ the set of its actions and by $\mathsf{Params}(evt)$ the set of its parameters. The translation $\mathscr{G}^i()$ of a set of guards of an *internal* or a *send* event is as follows:

$$\mathscr{G}^i(\{proc \in PCl, t_1 \in S_1,\ldots,t_l \in S_l,\ pc(proc) = st, g_1,\ldots,g_n\}) \triangleq \texttt{self.pc=="st"}\ \mathbf{and\ some}\ (t_1\ \mathbf{in}\ \mathscr{T}^l_{\varnothing}(S_1),\ldots,t_l\ \mathbf{in}\ \mathscr{T}^l_{\varnothing}(S_l),\ \mathbf{has}{=}\mathscr{T}^l_{\mathsf{Params}(evt)}(g_1)\ \mathbf{and}\ldots\mathbf{and}\ \mathscr{T}^l_{\mathsf{Params}(evt)}(g_n))$$

where $\mathsf{Params}(evt) = \{t_1,\ldots,t_l\}$ and $S_1,\ldots,S_l$ are finite sets. The translation $\mathscr{A}_{\overrightarrow{x}}(\mathsf{Actions}(evt))$ of a set of actions of an *internal* or *send* event *evt* is defined as the juxtaposition of the translations $\mathscr{T}^l_{\overrightarrow{x}}(a_j)$ of each action in the set $\mathsf{Actions}(evt)$. Since the actions of $\mathsf{Actions}(evt)$ are observed concurrently but translated as a sequence of assignments, fresh temporary variables are defined as copies of the local variables prior to the event and are used to access the old values of the local variables. However, for simplicity, we omit these temporary fresh variables in our example.

For each state $st \in \mathsf{StatesSet}(PCl)$ we use the set $\{evt_1,\ldots,evt_m\} \subseteq \mathsf{Events}(PCl, st)$ of all *internal* and *send* events observable in state *st* to generate the method `st`:

```
def st():
    --st
    if await (Gi(Guards(evt1))):
        A_{Params(evt1)}(Actions(evt1))
    ⋮
    elif(Gi(Guards(evtm))):
        A_{Params(evtm)}(Actions(evtm))
    elif(self.pc != "st"):
        pass
```

with the label `--st` and the keyword **await** added only if there is a *receive* event in $\mathsf{Events}(PCl, st)$; this statement is used to enable the reception of messages. When an **await** statement is reached every message that has arrived to destination but has not been processed yet, *i.e.* messages in the message queue of this process, is handled (using the **receive** methods) before the **if** conditions are evaluated. Messages are received until the message queue is empty and one of the guard conditions is satisfied.

Example 4.2. *In our example, we have* $\mathsf{Events}(P, sr) = \{sendRequest, stopSending\}$ *and thus the following code is generated for the method* `sr`.

```
def sr():
    # event sendRequest
    if(self.pc == "sr" and
       some(q in self.network,
          has=not(some(sent((MessagePrefixes.request,), to=_q))))):
       send((MessagePrefixes.request,), to=q)
    # event stopSending
    elif(self.pc == "sr" and
         each(q in self.network,
            has=some(sent((MessagePrefixes.request,), to=_q)))):
       self.pc = "wa"
    elif(self.pc != "sr"):
       pass
```

For each *receive* event *evt* in $\mathsf{Events}(PCl, st)$ we generate a **receive** method in the class `PCl`:

def receive$(\mathscr{G}^r(\mathsf{Guards}(evt))):$
$\quad \mathscr{A}_{\mathsf{Params}(evt)}(\mathsf{Actions}(evt))$

where the translation $\mathscr{G}^r(\mathsf{Guards}(evt))$ of a set of guards of a *receive* event *evt* is as follows:

$$\mathscr{G}^r(\{proc \in PCl, msg \in \mathsf{Messages}, source \in \mathsf{Nodes}, t_1 \in S_1, \ldots, t_l \in S_l, \\ pc(proc) = st, msg = msgExpr, source = procExpr\}) \triangleq$$

msg=$(\mathscr{T}_\varnothing(msgExpr))$**,**
from_=$\mathscr{T}_\varnothing(procExpr)$**,**
at=`(st,)`

If *procExpr* is empty, *i.e.* not specified in the model then a free variable it is used in the translation (to indicate the source of the message is not specified). We proceed similarly when *msgExpr* is empty. The actions of a *receive* event are translated in the same way as the actions of an *internal* or *send* event.

Example 4.3. *The following code corresponds to the* receive *event receiveAnswer.*

```
def receive(msg=(MessagePrefixes.answer, r), from_=source,
            at=(wa,)):
    self.result[source] = r
```

The translation has been implemented in Java as a RODIN plugin and the source code together with the installation instructions are available at `https://gitlab.inria.fr/agrall/eb2da`.

5 Concluding Remarks and Future Work

The localization of EVENT-B has been used when a distributed algorithm [3, 1] has been developed using the correct-by-construction paradigm and especially the refinement relationship among levels of abstractions. The translation of local EVENT-B models was a manual process and the current work provides a systematic way to produce a DISTALGO program from a local EVENT-B model.

We claim the LB modelling language is sufficiently powerful to model a large variety of distributed algorithms and abstract enough to be considered as the basis for the translation towards different target distributed programming languages. A couple of algorithms have been modelled and the programs obtained by translation allowed the simulation of the algorithms for different numbers of nodes. We continue to develop more and more elaborated case studies.

In the short term we plan of course to produce the proof of soundness of the translation. The communication model used for the algorithms implemented so far although reliable does not guarantee the

order of messages; we intend to provide the model for other communications models together with the corresponding translation. At the implementation level, we should first provide an automatic packaging and facilitate the installation as a RODIN plugin. The definition of transformations for other target distributed programming languages is a more long term objective.

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, doi:10.1017/S0956796812000081.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. International Journal on Software Tools for Technology Transfer 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] Jean-Raymond Abrial, Dominique Cansell & Dominique Méry (2003): *A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol.* Formal Aspects of Computing 14(3), pp. 215–227, doi:10.1007/s001650300002.

[4] Néstor Cataño & Víctor Rivera (2016): *EventB2Java: A Code Generator for Event-B*. In Sanjai Rayadurgam & Oksana Tkachuk, editors: *NASA Formal Methods*, Springer International Publishing, Cham, pp. 166–171, doi:10.1007/978-3-319-40648-0_13.

[5] Zheng Cheng, Dominique Méry & Rosemary Monahan (2016): *On Two Friends for Getting Correct Programs - Automatically Translating Event B Specifications to Recursive Algorithms in Rodin*. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, pp. 821–838, doi:10.1007/978-3-319-47166-2_57.

[6] Horatiu Cirstea, Alexis Grall & Dominique Méry (2020): *Generating Distributed Programs from Event-B Models*. Research Report, LORIA UMR 7503 CNRS, INRIA, Université de LORRAINE. Available at `https://hal.inria.fr/hal-02572971`.

[7] Andrew Edmunds & Michael Butler (2011): *Tasking Event-B: An extension to Event-B for generating concurrent code*. In: *PLACES 2011*. Available at `http://eprints.soton.ac.uk/id/eprint/272006`.

[8] Yanhong A Liu, Scott D Stoller, Bo Lin & Michael Gorbovitski (2012): *From clarity to efficiency for distributed algorithms*. In: *ACM SIGPLAN Notices*, 47, ACM, pp. 395–410, doi:10.1145/2384616.2384645.

[9] Dominique Méry (2009): *Refinement-based guidelines for algorithmic systems*. International Journal of Software and Informatics 3(2-3), pp. 197–239.

[10] Dominique Méry (2018): *Modelling by Patterns for Correct-by-Construction Process*. In: Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I, pp. 399–423, doi:10.1007/978-3-030-03418-4_24.

[11] Dominique Méry (2019): *Verification by Construction of Distributed Algorithms*. In: Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings, pp. 22–38, doi:10.1007/978-3-030-32505-3_2.

[12] Dominique Méry & Rosemary Monahan (2013): *Transforming Event B Models into Verified C# Implementations*. In Alexei Lisitsa & Andrei P. Nemytykh, editors: First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, July 12-13, 2013, EPiC Series in Computing 16, EasyChair, pp. 57–73.

[13] Gerard Tel (2000): *Introduction to distributed algorithms*. Cambridge University Press, doi:10.1017/CBO9781139168724.

[14] Mohamed Tounsi, Mohamed Mosbah & Dominique Méry (2016): *From Event-B specifications to programs for distributed algorithms*. IJAACS 9(3/4), pp. 223–242, doi:10.1504/IJAACS.2016.079623.