Refinement-based Construction of Correct Distributed Algorithms

Dominique Méry

Refinement-based Construction of Correct Distributed Algorithms

Dominique Méry
LORIA UMR CNRS 7503

Université de Lorraine
Vandœuvre-lès-Nancy, France.
dominique.mery@loria.fr; ORCID: 0000-0001-5231-6611

***Abstract*—The verification of distributed algorithms is a challenge for formal techniques supported by tools, as model checkers and proof assistants. The difficulties, even for powerful tools, lie in the derivation of proofs of required properties, such as safety and eventuality, for distributed algorithms. Verification by construction can be achieved by using a formal framework in which models are constructed at different levels of abstraction; each level of abstraction is refined by the one below, and this refinement relationships is documented by an abstraction relation namely a gluing invariant. The highest levels of abstraction are used to express the required behavior in terms of the problem domain and the lowest level of abstraction corresponds to an implementation from which an efficient implementation can be derived automatically. We describe a methodology based on the general concept of refinement and used for developing distributed algorithms satisfying a given list of safety and liveness properties. We will show also how formal models can be used for producing distributed programs of a real programming language. The modelling methodology is defined in the Event-B modelling language using the Rodin Formal IDE.**

I. Introduction

Correctness by Construction (CbC) is a method of building software -based systems with demonstrable correctness for safety-critical applications. CbC advocates a step-wise refinement process from specification to code using tools for checking and transforming models. CbC is considering the general pattern starting with an abstract model of the problem, progressively adding details in a step-wise and checked fashion, while each step guarantees and proves the correctness of the new concrete model with respect to requirements. The CbC process is following the principle of the cleanroom model [35], which is also advocating the progressive enrichment of the model by checking at each step that no dirt has been added, when developing software-based systems especially critical systems. CbC has played an important role to develop and to verify systems progressively as for instance in the case of the B Method [1] which has demonstrated its effectiveness in industrial projects [3]. The step-wise development in the B method and its recent evolution -B [2] is implemented by the refinement relationship between formal models. The refinement is characterized by a set of proof obligations that should be discharged and the use of proof procedures may partially mechanize the checking of those proof obligations. Consequently, several important issues should be investigated as the mechanization of proof obligations checking, the sound integration of knowledge and expert domain or the refinement process. In this paper, we consider the refinement process and the use of guidelines called *proof-based patterns* to facilitate the development of distributed algorithms and systems.

The triptych approach [11] covers three main phases of the software development process: *domain description*, *requirements prescription* and *software design*. $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$ expresses a formal notation, in which $\mathcal{D}$ represents the domain concepts in form of properties, axioms, relations, functions and theories; $\mathcal{S}$ represents a system model and $\mathcal{R}$ represents the intended system requirements. This notation states that the given domain description ($\mathcal{D}$) and the system model ($\mathcal{S}$) are correct with respect to the given requirements ($\mathcal{R}$) and it relates different elements involved, when developing a solution for a given problem. The triptych does not tell us how to build its three elements but it helps to *set the scene* and to express the *what should be defined*. In a book entitled *How to Solve It*, Pólya [37] suggests the following steps Figure 1 when solving a mathematical problem: first, understanding the problem (UP); second, making a plan (MP); third, carrying out the plan (CP); finally, looking back on the work by review and extend (RE).

Understanding the problem is generally related to the formalisation of the domain of problem $\mathcal{D}$ and we are promoting the reuse of existing theories or libraries. The second step MP can be the search of a pattern; it may be also possible to sketch the system by using a diagram as Venn diagrams. However, the question is to have a list of possible so called *patterns* which can be applied and reused. Some advices of Pólya are very close to a creative point : *If you can't solve a problem, then there is an easier problem you can solve: find it.* or *If you cannot*

This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project `http://impex.loria.fr`) sketcj from the Agence Nationale de la Recherche (ANR).

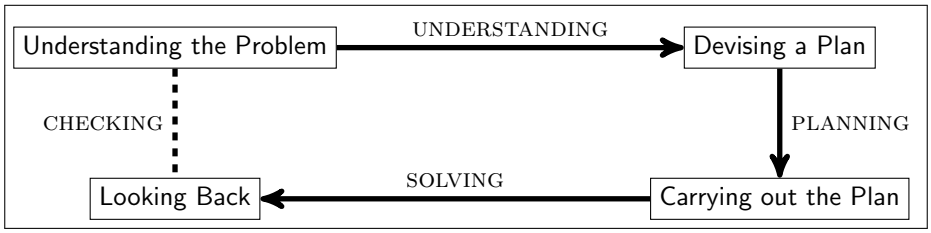

Figure 1. Schema for Pólya

solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem?. From Pólya [37] and Gamma [20], *patterns* are a key concept for solving problems in a general settlement. MoreMoreover, another key concept is the refinement of models handling the complex nature of such systems: the refinement is used for constructing models or patterns. Following Abrial et al [21] and Cansell et al [12], we revisit a list of patterns which can be used for developing distributed programs or distributed systems using the refinement and the proof as a mean to check the whole process.

Patterns [20] have greatly improved the development of programs and software by identifying practices that could be replayed and reused in different software projects. Moreover, they help to communicate new and robust solutions for developing a software for instance; it is clear that design patterns are a set of recipes that are improving the production of software. When developing (formal) models of systems, we are waiting for adequate patterns for developing models and later for translating models into programs or even software. Abrial et al [22] have already addressed the definition of patterns in the Event-B modelling language and have proposed a plugin which is implementing the instantiation of a pattern. Cansell et al [12] propose a way to reuse and to instantiate patterns. Moreover, patterns intends to make the refinement-based development simpler and the tool BART [16] provides commands for automatic refinement using the Atelier B Formal IDE [15]. The BART process is rule-based so that the user can drive refinement. We aim to develop patterns which are following Pólya's approach in a smooth application of Event-B models corresponding to classes of problems to solve as for instance an iterative algorithm, a recursive algorithm [29], a distributed algorithm ... Moreover, no plugin is necessary for applying our patterns.

To summarize what is our notion of pattern, we simply define a pattern as a Event-B model which is solving either partially or completely a problem: for instance, the inductive pattern solves the problem of computing a value using an inductive set of values. In our previous works [31], [32] we have organized patterns with respect to paradigms identified in our refinement-based development. A paradigm is a distinct set of patterns, including theories, research methods, postulates, and standards for what constitutes legitimate contributions to designing programs. A pattern for modelling in Event-B is a set of contexts and machines that have parameters as sets, constants, variables ... The notion of pattern has been introduced progressively in the Event-B process for improving the derivation of formal models and for facilitating the task of the person who is developing a model. In our work, students are the main target for testing and using these patterns. Our definition is very general but we do not want a very precise definition since the notion of pattern should be as simple as possible and should be helpful.

In this paper, we discuss several patterns that we have used and identified, when teaching and when developing case studies using the Event-B modelling language. The structure of the article is as follows. In Section II, we review preliminary material: the modelling framework. Section III proposes the service-as-event paradigm which is a generalization of the call-as-event paradigm. We illustrate the distributed pattern by developing the mutual exclusion protocol [13], [24], [25], [38]. Section IV completes Section III by describing works that are transforming Event-B models into distributed programs of target distributed programming languages. Section V concludes the paper and discusses future works and perspectives.

II. The Modelling Framework Event-B for Step-wise Development

,

This section describes the essential components of the modelling framework. In particular, we will use the Event-B modelling language [2] for modelling systems in a progressive way. Event-B has two main components: *context* and *machine*. A *context* is a formal static structure that is composed of several other clauses, such as *carrier sets*, *constants*, *axioms* and *theorems*. A *machine* is a formal structure composed of *variables, invariants, theorems, variants* and *events*; it expresses state-related properties. A machine and a context can be connected with the *sees* relationship. Events play an important role for modelling the functional behaviour of a system and are observed. An event is a state transition that contains two main components: *guard* and *action*. A *guard* is a predicate based on the state variables that defines a necessary condition for enabling the event. An *action* is also a predicate that

allows modifying the state variables when the given guard becomes true. A list of invariants defines required safety properties and constraint the state variables. There are several proof obligations, such as invariant preservation, non-deterministic action feasibility, guard strengthening in refinements, simulation, variant, well-definiteness, that must be checked during the modelling and verification process.

EVENT-B allows us modelling a complex system gradually using *refinement*. The refinement enables us to introduce more detailed behaviour and the required safety properties by transforming an abstract model into a concrete version. At each refinement step, events can be refined by: (1) keeping the event as it is; (2) splitting an event into several events; or (3) refining by introducing another event to maintain state variables. Note that the refinement always preserves a relation between an abstract model and its corresponding concrete model. The newly generated proof obligations related to refinement ensures that the given abstract model is correctly refined by its concrete version. Note that the refined version of the model always reduces the degree of non-determinism by strengthening the guards and/or predicates. The modelling framework has a very good tool support (Rodin Formal IDE [4]) for project management, model development, conducting proofs, model checking and animation, and automatic code generation. There are numerous publications and books available for an introduction to and related refinement strategies [2].

Since models may generate very *tough* proof obligations to automatically discharge, the development of proved models can be improved by the refinement process. The key idea is to combine models and elements of requirements using the refinement. The refinement [9], [10] of a machine allows us to enrich a model in a *step-by-step* approach, and is the foundation of our *correct-by-construction* approach. Refinement provides a way to strengthen the invariant and to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is done by extending the list of state variables, by refining each abstract event into a corresponding concrete version, and by adding new events. The next diagram illustrates the refinement-based relationship among events and models:

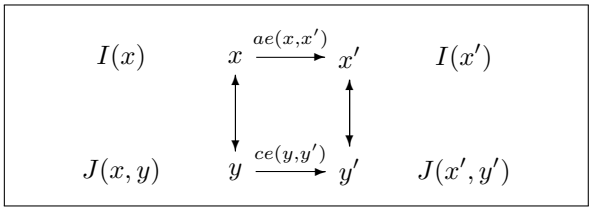

Diagram 1: Refinement of two events

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. The abstract state variables, x, and the concrete ones, y, are linked together by means of the, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event of AM is correctly refined by its corresponding concrete version of CM, (2) each new event of CM refines *skip*, which is intending to model *hidden* actions over variables appearing in the refinement model CM. More formally, if $BA(ae)(x, x')$ and $BA(ce)(y, y')$ are respectively the abstract and concrete before-after predicates of events, we say that ce in CM refines ae in AM or that ce simulates ae, if one proves the following statement corresponding to proof obligation: $I(x) \ \wedge \ J(x, y) \ \wedge \ BA(ce)(y, y') \ \Rightarrow \ \exists x' \cdot (BA(ae)(x, x') \ \wedge \ J(x', y'))$. To summarise, refinement guarantees that the set of traces of the abstract model AM contains (modulo stuttering) the traces of the concrete model CM.

The next diagram summarises links between contexts (CC extends AC); AC defines the set-theoretical logical and problem-based theory of level i called $\mathcal{T}h_i$, which is extended by the set-theoretical logical and problem-based theory of level i called $\mathcal{T}h_{i+1}$, which is defined by CC). Each machine (AM, CM) sees set-theoretical and logical objects defined from the problem statement and located in the CONTEXTS models (AC, CC). The abstract model AM of the level i is refined by CM; state variables of AM is x and satisfies the invariant $I(x)$. The refinement of AM by CM is checking the invariance of $J(x, y)$ and does need to prove the invariance of $I(x)$, since it is obtained freely from the checking of AM.

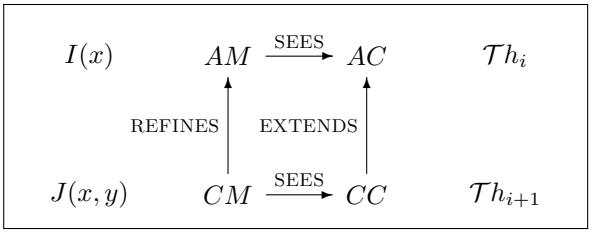

Diagram 2: Refinement of EVENT-B machines

The management of proof obligations is a technical task supported by the Rodin Formal IDE [4], which provides an environment for developing correct-by-construction models for software-based systems according to the diagram. Moreover, the Rodin Formal IDE integrates ProBn a tool for animating EVENT-B models and for model-checking finite configurations of EVENT-B models at different steps of refinement. ProB is used for checking deadlock-freedom and for helping in the discovery of invariants. Another possibility is the use of the Atelier B Formal IDE [15], which is providing functionalities for managing the modelling language called B-System; anyEVENT-B entity can be translated into a B-System entity and conversely. From the point of view of proof engineering, the Rodin Formal

IDEprovides a better interface for developing interactive proofs. However, the Atelier B Formal IDE is a mature industrial tool for assisting the design of correct-by-construction software-based systems and it has been used in successful in developing transport-related systems as the system level formal (Proof Based) verification for the CBTC of line 7 of NewYork Metro.

III. The Service-as-Event Paradigm

The next question is to handle concurrent and distributed algorithms corresponding to different programming paradigms as message-passing or shared-memory or coordination-based programming. C. Jones [23] develops the rely/guarantee concept for handling (possible and probably wanted) interferences among sequential programs. Rely/guarantee intends to make *implicit* [6] interferences as well as cooperation proofs in a proof system. In other methods as Owicki and Gries [36], the management of non-interference proofs among annotated processes leads to a important amount of extra proof obligations: checking interference freeness is explicitly expressed in the inferences rules. When considering an event as modelling a call of function or a call of a procedure, we implicitly express a computation and a sequence of state. In a joint work [33] we propose a temporal extension of Event-B to express liveness properties. The extension is a small bridge between Event-B and TLA/TLA$^+$ [26] with a refinement perspective. As C. Jones in rely/guarantee, we express implicit properties of the environment on the protocol under description by extending the call-as-event paradigm by a service-as-event paradigm. In [7], [8], the service-as-event paradigm is explored on two different classes of distributed programs/algorithms/applications: the snapshot problem and the self-healing P2P by Marquezan et al [28]. The self-healing problem is belonging to the larger class of self-$\star$ systems [17].

In previous patterns, we identify one event which *simulates* the execution of an algorithm either as an iterative version or as a recursive version. We are now introducing and illustrating the distributed pattern which is a representative of the service-as-event paradigm.

A. The distributed pattern

An event can be observed in a complex environment. The environment may be active and should be expressed by a set of events which are simulating the environment. Since the systems under consideration are reactive, it means that we should be able to model a service that a system should ensure. For instance, a communication protocol is a service which allows to transfer a file of a process A into a file of a process B.

Fig. 2 sketches the distributed pattern. The machine SERVICE is modelling services of the protocol; the machine PROCESS is refining each service considered as an event and makes the (computing) process explicit. The machine COMMUNICATION is defining the communications among the different agents of the possible network. Finally the machine LOCALALGO is localizing events of the protocol. The distributed pattern is used for expressing *phases* of the target distributed algorithm (for instance, requesting mutual exclusion) and to have a separate refinement of each phase. We sketch the service-as-event paradigm as follows. We consider one service. The target algorithm $\mathcal{A}$ is first described by a machine M0 with variables x satisfying the invariant $I(x)$.

The first step is to list the services e $S \ \widehat{=}\ \{s_0, s_1, \ldots, s_m\}$ provided by the algorithm $\mathcal{A}$ and to state for each service s_i a liveness property $P_i \rightsquigarrow Q_i$. We characterise by $\Phi_0 \ \widehat{=}\ \{P_0 \rightsquigarrow Q_0, P_1 \rightsquigarrow Q_1, \ldots, P_m \rightsquigarrow Q_m\}$. We add a list of safety properties defined by $\Sigma_0 = \{Safety_0, Safety_1, \ldots, Safety_n\}$. An event is defined for each liveness property and standing for <mutualthe eventuality of e by a fairness assumption which is supposed on e. Liveness properties can be visualised by assertions diagrams helping to understand the relationship among phases.

The second step is its refinement M1 with variables y glued properties in by $J(x, y)$ using the Event-B refinement and using the REF refinement which is defined using the temporal proof rules for expanding liveness properties. $P \rightsquigarrow Q$ in Φ_0 is proved from a list of Φ_1 using temporal rules. For instance, $P \rightsquigarrow Q$ in Φ_0 is then refined by $P \rightsquigarrow R, R \rightsquigarrow Q$, if $P \rightsquigarrow R, R \rightsquigarrow Q \vdash P \rightsquigarrow Q$. If we consider C as the context and M as the machine, C, M satisfies $P \rightsquigarrow Q$ and C, M satisfies $\Box Safety$. We use a temporal semantics relating contexts, machines and properties [33]. The link called LIVE expresses the satisfaction relationship. The diagram in table MITEX3 is summarising the relationship among models.

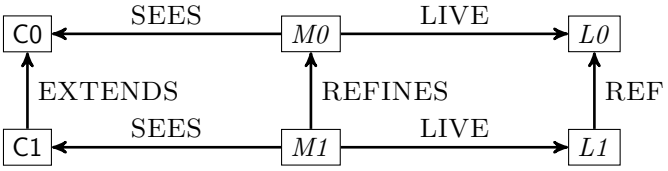

Diagram 3: The REF Refinement

The liveness property $P \xrightarrow{e} Q$ means that

- $\forall x, x' \cdot P(x) \wedge I(x) \wedge BA(e)(x, x') \Rightarrow Q(x')$
- $\forall x \cdot P(x) \wedge I(x) \Rightarrow (\exists x' \cdot BA(e)(x, x'))$
- $\forall f \neq e \cdot \forall x, x' \cdot P(x) \wedge I(x) \wedge BA(f)(x, x') \Rightarrow (P(x') \vee Q(x'))$

$P \xrightarrow{e} Q$ expresses implicitly that tyhe event e is under weak fairness. Each liveness property $P_i \rightsquigarrow Q_i$ in Φ_0 is modelled by an event:

$$\textbf{EVENT } \mathsf{e}_i \ \widehat{=}\ \textbf{WHEN } P_i(x) \textbf{ THEN } x : |Q_i(x') \textbf{ END}$$

We can add some fairness assumption over the event:

- $P_i \xrightarrow{\mathsf{e_i}} Q_i$ with weak fairness on e ($WF_x(\mathsf{e_i})$),
- $P_i \overset{\mathsf{e_i}}{\Rightarrow} Q_i$, with strong fairness on e ($SF_x(\mathsf{e_i})$).

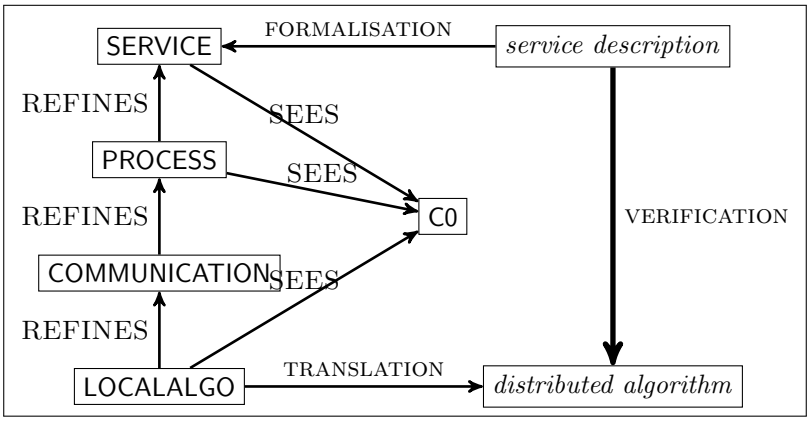

Figure 2. The distributed pattern

If we consider the leader election protocol [5], we have the following elements:

- **Sets**: ND (set of nodes).
- **Constants :** g is acyclic and connected ($acyclic(g) \wedge connected(g)$).
- **Variables :** $x = (sp, rt)$ (sp is a spanning tree of g).
- **Precondition) :** $P(x) \ \widehat{=} \ sp = \varnothing \wedge rt \in ND$
- **Postcondition :** $Q(x) \ \widehat{=} \ spanning(sp, rt, g)$

We can express the main liveness property: $(sp = \varnothing \wedge rt \in ND) \rightsquigarrow spanning(sp, rt, g)$ and we define the machine $\mathsf{Leader_0}$ satisfying the liveness property:

EVENT $\mathsf{election_0} \ \widehat{=}$
BEGIN
$sp, rt : |spanning(sp', rt', g)$
END

$$\mathsf{C_0} \xleftarrow{\text{SEES}} \mathsf{Leader_0} \xrightarrow{\text{LIVE}} (WF_x(\mathsf{election_0}), \{P \rightsquigarrow Q\}$$

We have introduced the service specification which should be refined separately from events of the machine *M0*. The next refinement should first introduce details of a computing process and then introduce communications in a very abstract way. The last refinement intends to localise the events. The model LOCALALGO is in fact an expression of a distributed algorithm. A current work explores the DistAlgo programming language as a possible solution for translating the local model into a distributed algorithm. Y.A. Liu et al [27] have proposed a language for distributed algorithms, DistAlgo, which is providing features for expressing distributed algorithms at an abstract level of abstractions. The DistAlgo approach includes an environment based on Python and managing links between the DistAlgo algorithmic expression and the target architecture. The language allows programmers to reason at an abstract level and frees her/him from architecture-based details. According to experiments of authors with students, DistAlgo improves the development of distributed applications. From our point of view, it is an application of the coordination paradigm based on a given level of abstraction separating the concerns.

B. Analysing the mutual exclusion problem

The mutual exclusion problem (MUTEX) [13], [24], [25], [38] is an important and interesting problem to address using the refinement-based methodology. The problem is to find a way to ensure that a finite set of distributed processes are sharing a resource R in an exclusive way under fairness assumption.

The general idea of MUTEX is to manage a *fifo* queue of processes *waiting for* the agreement of the other processes as indicated in the diagram 4.

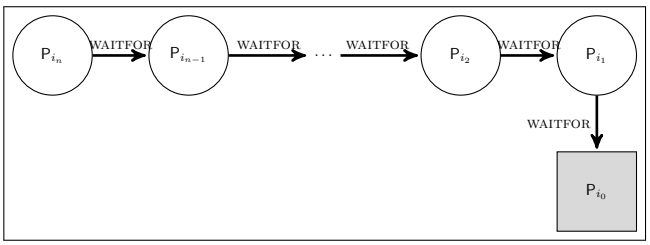

Diagram 4: General idea of the MUTEX problem

The diagram is expressing that the set of *waiting* processes is $\{P_{i_1}, \ldots, P_{i_n}\}$ and the *waitfor* relation over processes is defining a queue which is satisfying the fifo policy. The diagram is indicating that the *square* process P_{i_0} is using the critical section (cs). The *waitfor* relation means that when P *waitfpr* Q is true, P is waiting for the agreement of Q and the agreement of processes related to Q by the *waitfor* relation. In our diagram, the process P_{i_j} waits for the processes $P_{i_{j-1}}, \ldots P_{i_1}, P_{i_0}$. The protocol is then simply described as follows:

- When the process P_0 exits the critical section, it informs each process that it agrees to give an agreement for entering the critical section.
- When the process P_1 gets the agreement from P_0, it can enter the critical section.

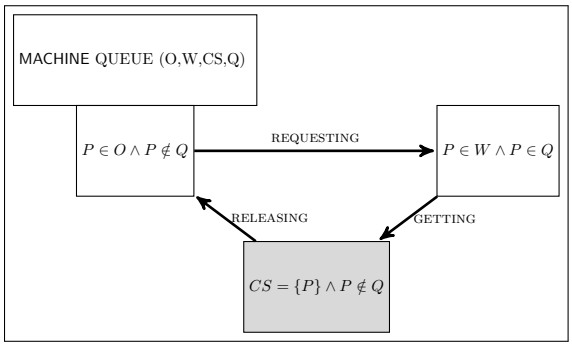

Diagram 5: General steps for machines QUEUE

Figure 3 is applying the distributed pattern (see Figure 2) and the machine SERVICE is describing the protocol by events requesting, getting and releasing. The machine SERVICE is only describing the three main steps of the process for requesting the mutual exclusion. The second machine QUEUE is explicitly using the *fifo* queue and the queue is expressing the required priority for the service MUTEX. However, the model is remaining *centralized* and we have now to introduce the possibility of using messages for requesting and releasing the ressource. The new machine COMQUEUE is introducing new variables as *req* ($p \mapsto q \in req$ means that p is sending a request to q for obtaining the ressource) and *rel* ($p \mapsto q \in rel$ means that p is sending a release to q for releasing the ressource). New events are introduced and are modelling the management of messages among the processes. Finally, the last refinement is implementing the queue by a pair of numbers namely the number for the process and the *osn* variable which is used for defining a total ordering among processes requesting the critical section but in a distributed way. The queue is simulated by these pairs of numbers. Proofs are not easy and models have been checked first using the ProB model-checker of Rodin Formal IDE [4]. We have cited solutions proposed in the literature [13], [24], [25], [38] . The last machine is very close to the two solutions and we have in fact a distributed solution for the MUTEX problem. When teaching the list of algorithms on the MUTEX problem, we use the current solution for showing the *story* of these algorithms.

IV. Translation of Event-B Models into Distributed Programs

The goal of our patterns is to produce *correct-by-construction* distributed algorithms and programs using proof techniques as well as proof engines. When considering the leader election protocol [5] , we have shown that we can list rules for localizing Event-B models. In [14], we have applied the general process of translation to a specific distributed programming language DistAlgo following the schema of the figure 4. The translation can be automated and will extend the production of *correct-by-construction* distributed programs.

Figure 4 identifies a last step labelled by `transformation` and the step is indicating that it is possible to generate an effective distributed program from an Event-B model satisfying constraints for making the transformation as automatic as possible. Let us recall that the Atelier B Formal IDE [15]provides functionalities for deriving C/C++/ADA codes from B machines called implementations and written in a subset of the B modelling language namely B0.

Following the same idea, the RIMEL (http://rimel.loria.fr) project has developed transformations of Event-B models into the local computation model of Visidia. The local computation model of Visidia [39], [40] has first considered static graph of processes and classical distributed algorithms as the leader election, the coloring, the naming or the enumeration, have been used for illustrating the transformation of local computation model into Visidia program. Recently, Mosbah [18] extends the scope of the transformation of local computation model in the case of dynamic networks.

V. Conclusion and Perspectives

The main goal in the current approach is to provide in the same process both modelling entities and proofs of properties required for these entities. The main idea is to develop a pattern-based approach which is ensuring both modelling and proving in the same process, as we have already illustrated in past works [6], [8], [29]–[34], [40]. Abrial et al [21] have proposed a plugin for automating the pattern application. Their contribution is to assist anyone who wants to obtain a completely checked Event-B project for a given problem with less toil. The tool is a real and useful proof companion but it requires a specific skill in proof development. Archives of Event-B projects are available at the following link `http://eb2all.loria.fr` and are used by students of the MsC programme at Université de Lorraine and Telecom Nancy. We use these patterns for teaching the art of using refinement when solving a problem and it turn out that students improved the projects as well as the exams. Proofs are carried using the proof assistant of Rodin Formal IDE [4]and these patterns help in organizing the different steps of refinement and in discharging proof obligations. The integration of Event-B models and the local computation model used in Visidia has been also in a very important issue in the creation of new patterns. In fact, we were able to address the proof of very technical algorithms as the naming algorithm of Mazurkiewicz (see http://rimel.loria.fr for the archives). The mechanization of the liveness part should be done and is part of the perspectives. The distributed pattern can be adapted for given computation models as we have done for the local computation model [19]. Finally, the translation from Event-B models into a distributed algorithm should

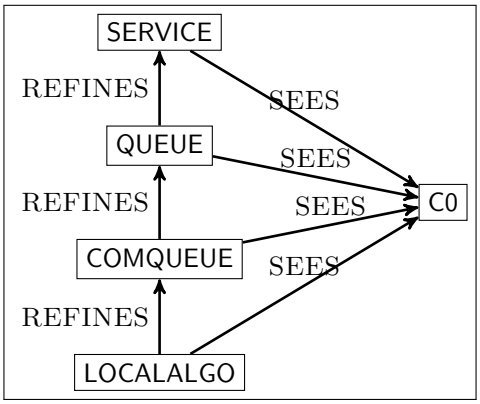

Figure 3. The mutual exclusion problem

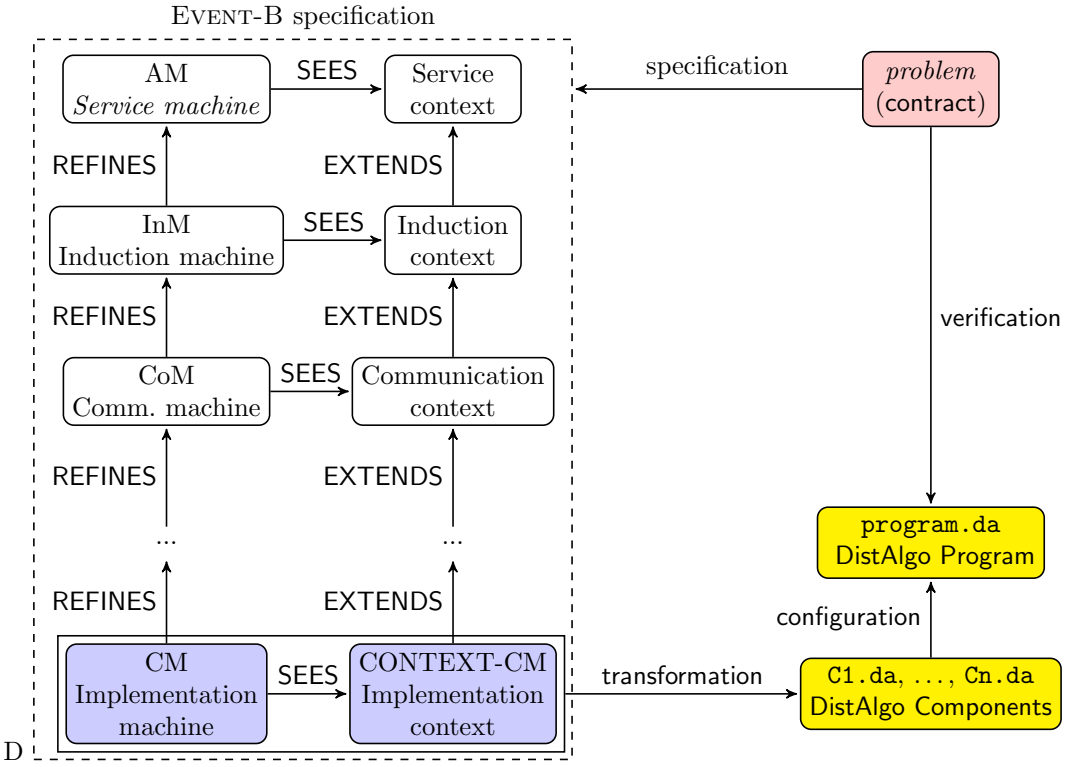

Figure 4. The global methodology for correct-by-construction distributed algorithms ([14]).

be improved. The previous works on translation into the DistAlgo distributed programming language [27] should be extended to other programming languages as MPI or even GO. One key-point is the preservation of the semantics while producing real distributed programs and the link with refinement.

Acknowledgment

The author thanks the organiser of the conference for the invitation to give a talk and to present ideas developed in previous projects mainly supported by the French National Agency for Research ANR as RIMEL (http://rimel.loria.fr) and IMPEX (http://impex.loria.fr) and the Ulysses programme for R& D projects mainly with the Maynooth University. He thanks the referees for the comments and the feedbacks. A special thank to Professor Dines Bjørner who has encouraged this research while he -was visiting the author in Nancy. Partners of the RIMEL project and the IMPEX project (Mohammed Mosbah, Yves Métivier, Pierre Castéran) help me in the different related publications. PhD students have also been very efficient persons especially Neeraj Kumar Singh and Manamiary Bruno Andriamiarina. Jean-Raymond Abrial and Dominique Cansell did the first steps in this domain with me and help me to clarify ideas on proofs and development. Finally, I warmly thank Professor Yamine

Ait-Ameur for his support, encouragement and help in my research.

References

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[3] Jean-Raymond Abrial. From Z to B and then Event-B: Assigning proofs to meaningful programs. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[4] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[5] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[6] Yamine Aït Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Sci. Comput. Program.*, 121:100–127, 2016.

[7] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Analysis of self-⋆ and P2P systems using refinement. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[8] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Revisiting snapshot algorithms by refinement-based techniques. *Comput. Sci. Inf. Syst.*, 11(1):251–270, 2014.

[9] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[10] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25, 1988. 10.1007/BF00291051.

[11] Dines Bjorner. *Software Engineering 1 Abstraction and Modelling; Software Engineering 2 Specification of Systems and Languages ; Software Engineering 3 Domains, Requirements, and Software Design.* Texts in Theoretical Computer Science. An EATCS Series. Sringer, 2006.

[12] Dominique Cansell, J. Paul Gibson, and Dominique Méry. Formal verification of tamper-evident storage for e-voting. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 329–338. IEEE Computer Society, 2007.

[13] Osvaldo Carvalho and Gérard Roucairol. On the distribution of an assertion. In Robert L. Probert, Michael J. Fischer, and Nicola Santoro, editors, *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, CanadaAugust 18-20, 1982*, pages 121–131. ACM, 1982.

[14] Horatiu Cirstea, Alexis Grall, and Dominique Méry. Generating distributed programs from Event-B models. In Laurent Fribourg and Matthias Heizmann, editors, *Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020 2020, and 7th Workshop on Horn Clauses for Verification and SynthesisDublin, Ireland, 25-26th April 2020*, volume 320 of *EPTCS*, pages 110–124, 2020.

[15] Clearsy System Engineering. Atelier B. http://www.atelierb.eu/, 2002.

[16] Clearsy System Engineering. BART, 2010. http://tools.clearsy.com/tools/bart/.

[17] Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.

[18] Faten Fakhfakh, Mohamed Tounsi, and Mohamed Mosbah. Modeling and proving distributed algorithms for dynamic graphs. *Future Gener. Comput. Syst.*, 108:751–761, 2020.

[19] Faten Fakhfakh, Mohamed Tounsi, Mohamed Mosbah, Dominique Méry, and Ahmed Hadj Kacem. Proving distributed coloring of forests in dynamic networks. *Computación y Sistemas*, 21(4), 2017.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[21] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. *Software and System Modeling*, 12(2):229–244, 2013.

[22] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. *Software and System Modeling*, 12(2):229–244, 2013.

[23] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[24] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[26] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[27] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Trans. Program. Lang. Syst.*, 39(3):12:1–12:41, 2017.

[28] Clarissa Cassales Marquezan and Lisandro Zambenedetti Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies.* Springer Briefs in Computer Science. Springer, 2012.

[29] Dominique Méry. Refinement-based guidelines for algorithmic systems. *Int. J. Software and Informatics*, 3(2-3):197–239, 2009.

[30] Dominique Méry. Playing with state-based models for designing better algorithms. *Future Gener. Comput. Syst.*, 68:445–455, 2017.

[31] Dominique Méry. Modelling by patterns for correct-by-construction process. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*, volume 11244 of *Lecture Notes in Computer Science*, pages 399–423. Springer, 2018.

[32] Dominique Méry. Verification by construction of distributed algorithms. In Robert M. Hierons and Mohamed Mosbah, editors, *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings*, volume 11884 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2019.

[33] Dominique Méry and Michael Poppleton. Towards an integrated formal method for verification of liveness properties in distributed systems: with application to population protocols. *Softw. Syst. Model.*, 16(4):1083–1115, 2017.

[34] Dominique Méry and Monahan Rosemary. Transforming Event-B Models into Verified C# Implementations. In Alexei Lisitsa and Andrei Nemytykh, editors, *VPT 2013 - First International Workshop on Verification and Program Transformation*, volume 16 of *EPIC*, pages 57–73. Alexei Lisitsa and Andrei Nemytykh, July 2013.

[35] H. D. Mills, M. Dyver, and R. Linger. Cleanroom software-engineering. *IEEE Software*, 4(5):19–25, 1987.

[36] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[37] George Pólya. *How to Solve It.* Garden City, NY: Doubleday, 1957.

[38] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[39] Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. Proving distributed algorithms by combining refinement and local computations. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 35, 2010.

[40] Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. From Event-B specifications to programs for distributed algorithms. *Int. J. Auton. Adapt. Commun. Syst.*, 9(3/4):223–242, 2016.