Modelling by Patterns for Correct-by-Construction Process

Dominique Méry(✉)

Université de Lorraine, LORIA UMR CNRS 7503, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy, France
dominique.mery@loria.fr

Abstract. Patterns 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 software development; it is clear that design patterns are a set of recipes that are improving the production of software. When developing models of systems, we are waiting for adequate patterns for building models and later for translating models into programs or even software. In this paper, we review several patterns that we have used and identified, when teaching and when developing case studies using the Event-B modelling language. The modelling process includes the use of formal techniques and the use of refinement, a key notion for managing abstractions and complexity of proofs. We have classified patterns in classes called paradigms and we illustrate three paradigms: the inductive paradigm, the call-as-event paradigm and the service-as-event paradigm. Several case studies are given for illustrating our methodology.

1 Introduction

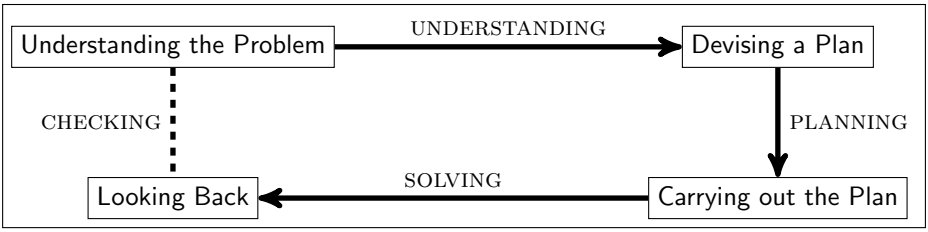
Formal methods have been used successfully for developing software-based systems especially critical systems. The *correct by construct* approach has played an important role to develop and to verify systems progressively. The triptych [9–11] approach covers three main phases of the software development process: *domain description*, *requirements prescription* and *software design*. A formal notation, namely $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$, relates three entities: $\mathcal{D}$ 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 [9–11] $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$ 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*. The modelling process includes the use of formal techniques and the use of refinement, a key

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

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 399–423, 2018.
https://doi.org/10.1007/978-3-030-03418-4_24

notion for managing abstractions and complexity of proofs. In this paper, we propose patterns organized in classes called paradigms and we illustrate three paradigms: the inductive paradigm, the call-as-event paradigm and the service-as-event paradigm. Our aim is to help users, mainly students, to lean how to use the refinement relationship when developing software-based systems.

In a book entitled *How to Solve It*, Pólya [33] suggests the following steps, 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 (UP) 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 making a plan (MP) can be the search of a pattern; it may be also possible to sketch the system to build by a diagram. 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 solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem?*. From Pólya and Gamma [21], *patterns* are a key concept for solving problems in a general settlement. Moreover, 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. [22] and Cansell et al. [13], we revisit a list of patterns which can be used for developing programs or systems using the refinement and the proof as a mean to check the whole process and which can be a mean to reuse former proofs in new developments.

Patterns [21] 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 software production. When developing (formal) system models, 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. [13] propose a way to reuse and to instantiate patterns.

Moreover, patterns intends to make the refinement-based development simpler and the tool BART [17] provides commands for automatic refinement using the AtelierB toolbox [16]. 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 [27], a distributed algorithm ... Moreover, no plugin is necessary for applying our patterns.

We are organising 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 (project) 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. We review several patterns that we have used and identified, when teaching and when developing case studies using the Event-B modelling language. The modelling process includes the use of formal techniques and the use of refinement, a key notion for managing abstractions. Moreover, we have also identified some paradigms that can be used and can facilitate the design of formal models.

The structure of the article is as follows. In Sect. 2, we review preliminary material: the modelling framework. Section 3 presents the inductive paradigm, which is illustrated in Sect. 4. In Sect. 5, we consider the call-as-event paradigm and compare it with the inductive paradigm. A paradigm is gathering patterns and Sect. 7 proposes the service-as-event paradigm which is a generalization of the call-as-event paradigm. We illustrate patterns by developing a protocol namely the *Sliding Window Protocol*. Section 8 concludes the paper and discusses future works and perspectives.

2 The Modelling Framework: Event-B for Step-Wise Development

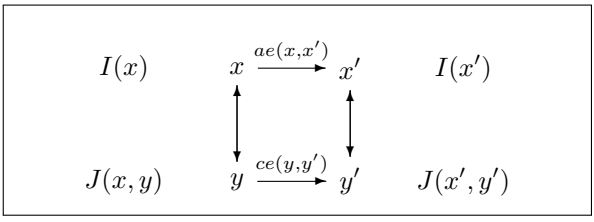
This section describes the essential components of the modelling framework. In particular, we will use the Event-B modelling language [1] for modelling systems in a progressive way. Event-B has two main components: *context* and *machine*. A *context* is a formal static structure that is composed of 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 set of invariants defines required safety properties that must be satisfied by all the defined 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) 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 Event-B and related refinement strategies [1].

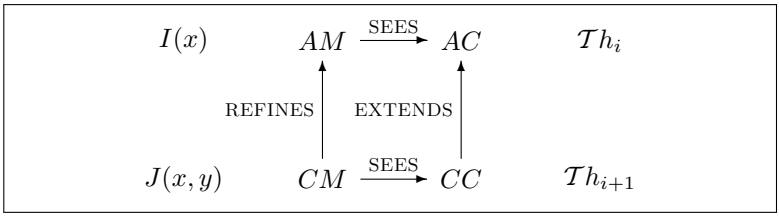
Since models may generate very *tough* proof obligations to automatically discharge, the development of proved models can be improved by the refinement process. The key idea is to combine models and elements of requirements using the refinement. The refinement [7,8] of a machine allows us to enrich a 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:



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.



The management of proof obligations is a technical task supported by the RODIN tool [2], which provides an environment for developing correct-by-construction models for software-based systems according to the diagram. Moreover, the RODIN platform integrates ProB, 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.

3 The Inductive Paradigm

First at all, we analyse the inductive paradigm using the refinement and we develop specific patterns. A computation is often characterised by the effective computing of a value of a sequence of values. The problem is to define the sequence of values and then to find a process for computing the value of a member of the sequence. The methodology is based on the case studies developed in the last decade and is the result of observations when teaching students how

to use Event-B and its refinement. Two questions are stated: how to model for getting an iterative algorithm computing a value defined by a given sequence at a given rank? and can we have a set of automatically discharged proof obligations as large as possible? The question is to define the sequence corresponding to the problem to solve and the sequence is giving the way for constructing the required value. The invariant is a very important part and is derived from analysis of the problem. The global pattern called the iterative pattern is sketched by the following diagram where machines and contexts are PREPOST, COMPUTING, PREALGO, ALGO, ALGOPC and C0. The context C0 contains the description of the problem which is the sequence (of values) defining the problem and the refinement is linking the machines. The last machine is ALGOPC which is translated into an algorithm *algorithm.*. The context C0 is enriched while the model is progressively refined.

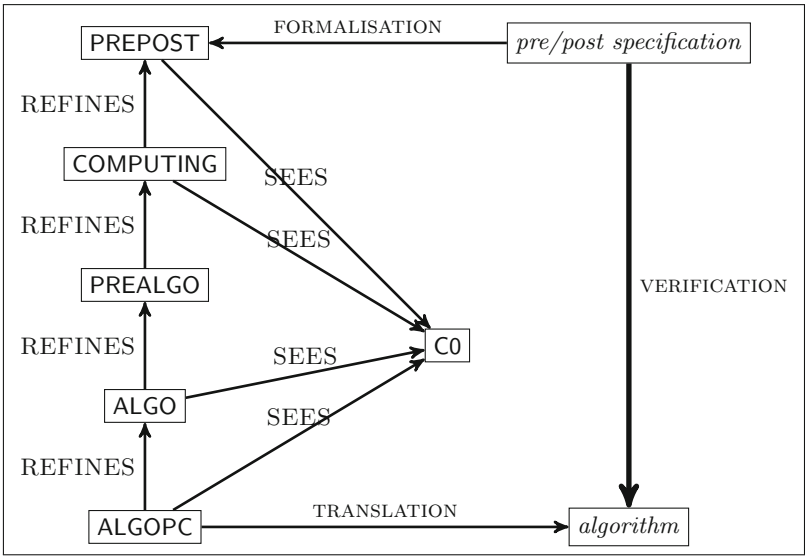


Fig. 1. The iterative pattern

CONTEXT $C0$
SETS
U
CONSTANTS
$x, v, d0, f, D$
AXIOMS
$axm1 : x \in \mathbb{N}$
$axm25 : D \subseteq U$
$axm24 : f \in D \to D$
$axm23 : d0 \in D$
$axm2 : v \in \mathbb{N} \to D$
$axm3 : v(0) = d0$
$axm4 : \forall n \cdot n \in \mathbb{N} \Rightarrow v(n+1) = f(v(n))$
$th1 : Q(d_0, d) \equiv (d = v(x))$

The context C0 is defining the sequence v which is used for expressing the post-condition $Q(d_0, d)$ with the precondition $P(d_0)$. The post-condition $Q(d_0, d)$ is equivalent to $d = v(x)$ where the sequence v is defined using d_0 as initial value of v. The theorem $th1$ should be proved in the context C0 and it states that the sequence v is soundly defining the problem. $th1$ expresses that the requested value d exists and the sequence provides an inductive process for computing it.

MACHINE *PREPOST*
SEES *C0*
VARIABLES
r
INVARIANTS
$inv1 r \in D$
EVENTS
INITIALISATION
BEGIN
$act1 : r :\in D$
END
EVENT computing
BEGIN
$act1 : r := v(x)$
END
END

The theorem $th1$ is validating the definition of the result r to compute. The event computing is expressing the *contract* of the given problem. The step from the context to the machine PREPOST is redefining the contract in a machine. The domain D is any possible domain and not only $\mathbb{N}$ but it may be a complex domain with multiple dimensions. We will illustrate it by a very simple problem that is the computation of the function n^2 using the addition operator.

Following the refinement-based approach, we introduce a refinement which is expliciting the computation process using the sequence v as a guide for reasoning. The refinement is the introduction of a very expensive variable vv recording and storing successive and necessary values of the sequence v.

The variable vv is storing the value s of v and it may appear very inappropriate. However, the goal is to structure the proof process and to introduce *modelling* variables which will be *hidden* later in the final refinement. The invariant is simply expressing the relationship between *mathematical values* of v and *modelling variable* of vv. k defines the domain of vv which is evolving during the process. The properties of variables are derived from the relationship which exists by the definition of the computation process:

(1) $vv \in \mathbb{N} \nrightarrow D$
(2) $k \in \mathbb{N}$
(3) $\forall i \cdot i \in dom(vv) \Rightarrow vv(i) = v(i)$
(4) $dom(vv) = 0 \,..\, k$
(5) $k \leq x$

Proofs obligations are discharged with a light interaction. The refinement is a progressive process and is progressing to a model close to an implementation. The refinement machine computing has a new event step updating the variable vv and the event computing is made more concrete by using the guard over the index k.

EVENT INITIALISATION
BEGIN
$act1 : r :\in D$
$act3 : vv := \{0 \mapsto d0\}$
$act5 : k := 0$
END

EVENT computing
REFINES computing,
WHEN
$grd1 : x \in dom(vv)$
THEN
$act1 : r := vv(x)$
END
END

EVENT step
WHEN
$grd1 : x \notin dom(vv)$
THEN
$act2 : vv(k+1) := f(vv(k))$
$act4 : k := k + 1$
END

Now, we have to add a new refinement which is preparing the final transformation. The idea is to make the task of the proof assistant easier and to explain how the invariant is built in a progressive way. A new variable cv is used for storing the last computed value of the sequence v: $cv \in D$ and $cv = vv(k)$.

EVENT INITIALISATION
BEGIN
$act1 : r :\in D$
$act3 : vv := \{0 \mapsto d0\}$
$act5 : k := 0$
$act8 : cv := d0$
END

EVENT computing
REFINES computing
WHEN
$grd1 : k = x$
THEN
$act1 : r := cv$
END

EVENT step
REFINES step
WHEN
$grd1 : k < x$
THEN
$act2 : vv(k+1) := f(vv(k))$
$act4 : k := k+1$
$act5 : cv := f(cv)$
END

Proof obligations are discharged without toil, thanks to the incremental refinement. The statement $vv(k+1) := f(vv(k))$ is *simulated* by the new statement $cv := f(cv)$. The new refinement is *hidden* the modelling variable vv. The variables r, cv and k are modelling the computation according to the safety properties: (1) $cv = v(k)$ (2) $k \leq x$ (3) $0 \leq k$. Events are modified by hiding the variable vv and correspond to the pattern.

EVENT INITIALISATION
BEGIN
$act1 : r :\in D$
$act5 : k := 0$
$act8 : cv := d0$
END

EVENT computing
REFINES computing
WHEN
$grd1 : k = x$
THEN
$act1 : r := cv$
END

EVENT step
REFINES step
WHEN
$grd1 : k < x$
THEN
$act4 : k := k+1$
$act6 : cv := f(cv)$
END

The final step is to derive an algorithm corresponding to the two events. A further refinement introduces the control variable called *pc* and we obtained the operational semantics based on relations between variables and primed variables.

Listing 1.1. Function derived from pattern for the sequence v

```
type(D)   f(int x)
{int   r,k,cv,or,ok,ocv;
  r=0;k=0;cv=0;or=0;ok=k;ocv=cv;
  while (k<x)
        {
          ok=k;ocv=cv;
          k=ok+1;
          cv=f(ocv);
        }
  r=cv;   return(r);}
```

The produced algorithm can be now checked using another proof environment as for instance Frama-C [34]. The inductive property of the invariant is clearly verified and is easily derived from the Event-B machines. The verification is not required, since the system is correct by construction but it is a checking of the process itself. Abrial [1] has addressed the question of developing sequential algorithms and has proposed a list of transformations of Event-B models; our Event-B project (or pattern) based on Abrial's case studies has added intermediate refinements and has identified model variables from programming variables. We have developed the project called ITERATIVE-PATTERN; the project is

the pattern itself and in the next section we apply it by specialising it for specific problems. The specialization leads to choose a sequence corresponding to the problem to solve and to complete the project ITERATIVE-PATTERN.

4 Applying the Iterative Pattern

The iterative pattern (see Fig. 1) can be applied by importing the previous project called ITERATIVE-PATTERN. We do not use the pattern plugin and the technique developed by Abrial and Hoang [22] and choose a solution which is simpler to apply with the RODIN platfom. The current project is then enriched by the definition of the sequence of the problem to solve. The user should find a way to express the problem by a sequence v over the domain D. The sequence v is a key point and it should be related to the required post-condition. A theorem should be derived in the context C0. We are considering two examples illustrating how the iterative pattern can be instantiated.

4.1 Example 1: x^2 and x^3 Without Toil

Computing the value x^2 for any natural number x without using the multiplication operator and using the addition operator, is a well known algorithm which is based on a simple observation. The value $(i+1)^2$ is developed into $i^2 + 2 * i + 1$ and the sequence v is defined as follows from this equality: $v(i+1) = v(i) + 2 * i + 1$ and then each term is defined as a term of another sequence $\begin{cases} v(i+1) = v(i) + w(i) + 1 \\ w(i+1) = w(i) + 2 \\ u(i+1) = u(i) + 1 \end{cases}$

In fact, the sequence v is defined with the help of two auxiliary sequences namely w and u. We can apply the iterative pattern by rewriting the previous definitions of sequence as follows:

$$\begin{pmatrix} v(i+1) \\ w(i+1) \\ u(i+1) \end{pmatrix} = \begin{pmatrix} v(i) + w(i) + 1 \\ w(i) + 2 \\ u(i) + 1 \end{pmatrix} = \begin{pmatrix} 1\ 1\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \end{pmatrix} \begin{pmatrix} v(i) \\ w(i) \\ u(i) \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

The domain D is $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and the sequence A is simply defined by:

$$\begin{cases} \forall i \in \mathbb{N} : A(i) = \begin{pmatrix} v(i) \\ w(i) \\ u(i) \end{pmatrix} \\ \forall i \in \mathbb{N} : A(i+1) = \begin{pmatrix} 1\ 1\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \end{pmatrix} A(i) + \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \end{cases}$$

The sequence A is a vector of sequences satisfying properties related to the post-condition and these properties should be proved using the proof assistant: $\forall i \in \mathbb{N} : A(i)_1 = i * i$, $\forall i \in \mathbb{N} : A(i)_2 = 2 * i$ and $\forall i \in \mathbb{N} : A(i)_3 = i$. The notation $A(i)_j$ denotes the j-th component of the vector $A(i)$. Finally, we obtain the following function which is checked using Frama-C.

Listing 1.2. Function derived from pattern power2

```
#include <limits.h>
/*@  requires  0 <= x;
     requires x*x <= INT_MAX ;
     ensures \result ==x*x;
*/
int power2(int x)
{int  r,k,cv,cw,or,ok,ocv,ocw;
  r=0;k=0;cv=0;cw=0;or=0;ok=k;ocv=cv;ocw=cw;
      /*@ loop invariant cv == k*k;
        @ loop invariant k <= x;
        @ loop invariant cw == 2*k;
        @ loop assigns k,cv,cw,or,ok,ocv,ocw; */
  while (k<x)
        {
          ok=k;ocv=cv;ocw=cw;
          k=ok+1;
          cv=ocv+ocw+1;
          cw=ocw+2;}
  r=cv;return(r);}
```

The same process can be applied for computing x^3 and we use the equality $(i+1)^3 = i^3 + 3i^2 + 3i + 1$. We introduce intermediate sequences and identify the following sequences:

- $z_0 = 0$ et $\forall n \in \mathbb{N} : z_{n+1} = z_n + v_n + w_n$
- $v_0 = 0$ et $\forall n \in \mathbb{N} : v_{n+1} = v_n + t_n$
- $t_0 = 3$ et $\forall n \in \mathbb{N} : t_{n+1} = t_n + 6$
- $w_0 = 1$ et $\forall n \in \mathbb{N} : w_{n+1} = w_n + 3$
- $u_0 = 0$ et $\forall n \in \mathbb{N} : u_{n+1} = u_n + 1$

$$\begin{pmatrix} z(i+1) \\ v(i+1) \\ t(i+1) \\ w(i+1) \\ u(i+1) \end{pmatrix} = \begin{pmatrix} z_i + v_i + w_i \\ v(i) + t(i) \\ t(i) + 6 \\ w(i) + 3 \\ u(i) + 1 \end{pmatrix} = \begin{pmatrix} 1\,1\,0\,1\,0 \\ 0\,1\,\;1\;\,0\,0 \\ 0\,0\,\;1\;\,0\,0 \\ 0\,0\,\;0\;\,1\,0 \\ 0\,0\,\;0\;\,0\,1 \end{pmatrix} \begin{pmatrix} z(i) \\ v(i \\ t(i) \\ w(i) \\ u(i) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 6 \\ 3 \\ 1 \end{pmatrix}$$

The domain D is $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and the sequence B is simply defined by:

$$\begin{cases} \forall i \in \mathbb{N} : B(i) = \begin{pmatrix} z(i) \\ v(i \\ t(i) \\ w(i) \\ u(i) \end{pmatrix} \\ \forall i \in \mathbb{N} : B(i+1) = \begin{pmatrix} 1\,1\,0\,1\,0 \\ 0\,1\,\;1\;\,0\,0 \\ 0\,0\,\;1\;\,0\,0 \\ 0\,0\,\;0\;\,1\,0 \\ 0\,0\,\;0\;\,0\,1 \end{pmatrix} B(i) + \begin{pmatrix} 0 \\ 0 \\ 6 \\ 3 \\ 1 \end{pmatrix} \end{cases}.$$

The sequence B is a vector of sequences satisfying properties related to the postcondition and these properties should be proved using the proof assistant: $\forall i \in \mathbb{N} : B(i)_1 = i * i * i$. The notation $B(i)_j$ denotes the j-th component of the vector $B(i)$. Finally, we obtain the following function which is checked using Frama-C.

Listing 1.3. Function derived from pattern power3

```
#include <limits.h>
/*@  requires  0 <= x;
     requires x*x*x <= INT_MAX ;
     ensures \result ==x*x*x;
*/
int power3(int x)
{int  r,ocz,cz,cv,cu,ocv,cw,ocw,ct,oct,ocu,k,ok;
  cz=0;cv=0;cw=1;ct=3;cu=0; ocw=cw;ocz=cz;
  oct=ct;ocv=cv;ocu=cu;k=0;ok=k;
      /*@ loop invariant cz == k*k*k;
        @ loop invariant cu  == k;
        @ loop invariant cv+ct==3*(cu+1)*(cu+1);
        @ loop invariant cz+cv+cw==3*(cu+1)*(cu+1)*(cu+1);
        @ loop invariant cv== 3*cu*cu;
        @ loop invariant cw == 3*cu+1;
        @ loop invariant k <= x;
        @ loop assigns ct,oct,cu,ocu,cz,ocz,k,cv,cw,r,ok;
        @ loop assigns ocv,ocw;*/
  while (k<x)
       {
         ocz=cz;ok=k;ocv=cv;ocw=cw;oct=ct;ocu=cu;
         cz=ocz+ocv+ocw;
         cv=ocv+oct;
         ct=oct+6;
         cw=ocw+3;
         cu=ocu+1;
         k=ok+1;}
  r=cz;return(r);}
```

In this case, the loop invariant is inductive but Frama-C does not prove it completely. This is not the case with the RODIN platform which is able to discharge the whole set of proof obligations. However, the Event-B model is using auxiliary knowledge over sequences used for defining the computing process. The most difficult theorem is to prove that $\forall n \in \mathbb{N} : z_n = n * n * n$. The second example is a new algorithm for computing n^3 with only addition operator and it is based on sequences which are defined from the equality simplifying $(i+1)^3$. The technique can be applied for the computation of i^k for any k.

4.2 Example 2: The Fibonacci Family

We consider a function $f \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ defined the complete set of natural numbers and an infinite sequence of natural values defined using a inductive definition as follow:

- $u_0 \in \mathbb{N}$
- $u_1 \in \mathbb{N}$
- $\forall n \in \mathbb{N} : u_{n+2} = f(u_n, u_{n+1})$

The inductive definition is considering not only the last previous element of the sequence but two last previous terms. We use an expression for reformulating the problem to solve and introduce a sequence f defined as follows:

- $F_1 \in \mathbb{N} \times \mathbb{N}$ where $F_1 = (u_0, u_1)$
- $\forall n \in \mathbb{N} : n \neq 0 :$
  $\begin{pmatrix} F_{n+1} = g(F_n) \\ g(F_n) = (f((F_{n-1})_1, (F_{n-1})_2), f((F_n)_1, (F_n)_2)) = ((F_n)_2, f((F_n)_1, (F_n)_2)) \end{pmatrix}$

The reformulation leads to the general format of the iterative pattern and it indicates also the necessity to have a specific variable for keeping the two previous values: *cv* is containing a pair.

Listing 1.4. Function derived from pattern fibo

```
type(D)   fibo(int x)
{int   r,k,cv,or,ok,ocv;
  k=0;cv=(u0,u1);ok=k;ocv=cv;
  while (k<x)
        { ok=k;ocv=cv;
          k=ok+1;
          /* cv=g(ocv);
          cv= (ocv_2,f(ocv_1,ocv_2));}
   r=cv;   return(r); }
```

In the algorithm, the variable *cv* is a pair keeping the two last values and we denote *ocv_i* the i-th component of *ocv*.

4.3 On Proofs Summary

Applying the iterative pattern requires to replay the proofs by instantiating the constants of the problem. The new model is solving a specific problem and we should prove extra theorems to derive the final post-condition as for instance $\forall n \in \mathbb{N} : z_n = n * n * n$. The progressive design of models facilitates the proofs as we have noticed for the function `power3`: Frama-C was not able to discharge any proof of the loop invariant. In the case of Fibonacci, the development of the solution using Event-B and the refinement show that the resulting function is correct by construction and the use of Frama-C is not possible since we need to prove that r contains a value of the sequence: Frama-C necessitates the definition of a theory with definition of Fibonacci sequence.

5 The Call-as-Event Paradigm

The inductive paradigm and the iterative pattern are using a sequence of values in a domain $\mathcal{D}$ and the computation process is based on the recording of the values of the sequence. In the case of the call-as-event paradigm, the pattern is based on the link between the occurrence of an event and a call of a function or procedure or method satisfying the pre-condition and post-condition respectively at the call point and the return point. The context C0 defines the sequence of values and the definition of the sequence is used as a guide for the shape of events. The definitions of sequence are reformulated by a diagram which is simulating the different cases when the procedure under development is called namely $P(x, r)$.

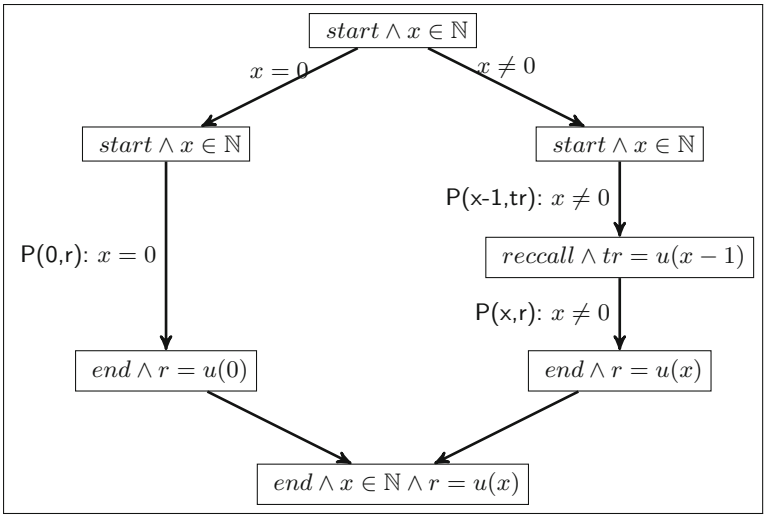


Fig. 2. Organisation of the computation in a recursive solution using assertion diagram

The diagram is derived from the Event-B model called ALGOREC and is a finite state diagram. It includes a liveness proof very close to the proof lattices of Owicki and Lamport [32]. We use special names for events in the diagram: P(0,r): $x = 0$ stands for the event observed when the procedure P(x,r) is called with x=0; P(x-1,tr): $x \neq 0$ models the observation of the recursive call of P; P(x,r): $x \neq 0$ stands for the event observed when the procedure P(x,r) is called with $x \neq 0$. P(0,r): $x = 0$ and P(x,r): $x \neq 0$ are refining the event computing which is observed when the procedure P is called.

MACHINE *ALGOREC*
REFINES *PREPOST*
SEES *C0*
VARIABLES
r, pc, tr
INVARIANTS
$art : pc \in L$
$inv1 : tr \in D$
$inv2 : pc = callrec \Rightarrow tr = v(x-1)$
$inv3 : pc = end \Rightarrow r = v(x)$

The refinement is an organisation of the inductive definition using a control variable *pc*. The control variable *pc* is organising the different steps of the computations simulated by the events. The invariant is derived directly from the definitions of the intermediate values. Proof obligations are simple to prove. It remains to prove that the values of the sequence v correspond to the required value in the post-condition.

EVENT P(x,r):x=0
REFINES computing
WHEN
$grd1 : x = 0$
$grd2 : pc = start$
THEN
$act1 : r := d0$
$act2 : pc := end$
END

EVENT P(x-1,tr):x/=0
WHEN
$grd1 : pc = start$
$grd2 : x \neq 0$
THEN
$act1 : tr := v(x-1)$
$act2 : pc := callrec$
END

EVENT P(x,r):x/=0
REFINES computing
WHEN
$grd1 : pc = callrec$
THEN
$act1 : r := f(tr)$
$act2 : pc := end$
END

The machine is simulating the organisation of the computations following two cases according to the Fig. 2. The first case is the path on the left part of the diagram and is when x is 0 and the second case if when x is not 0.

The first path is a three steps path and is labelled by the condition $x = 0$ an d the event P(0,r):x=0. The event P(x,r):x=0 is assigning the value $d0$ to r according to the definition of $u(0)$. It refines the event computing in the abstraction. The third step is an implication leading to the postcondition.

The second path is a four steps path and is labelled by the condition $x \neq 0$, then the event P(x-1,r):$x \neq 0$ is modelling the recursive call of the same procedure. Finally the event P(,r):$x \neq 0$ is refining the event computing. The call as event paradigm is applied when one considers that one event is defining the specification of the recursive call and the user is giving the name of the call to indicate that the event should be translated into a call. The EB2RC plugin [15] generates automatically a C-like program.

The model ALGOREC is simple to checked. Proof obligations are simple, because the recursive call is hiding the previous values stored in the variable vv of the iterative paradigm. The prover is much more efficient.

The recursive pattern is linked to a diagram which is helping to structure the solution. We have labelled arrows by guards or by events. The diagram helps to structure the analysis based on the inductive definitions. Following this pattern, we have developed the ERB2RC plugin based on the identification of three possible events. When a pre/post specification is stated, the program to build can be expressed by a simple event expressing the relationship between input and output and it provides a way to express pre/post specification as events. The first model is a very abstract model containing the pre/post events.

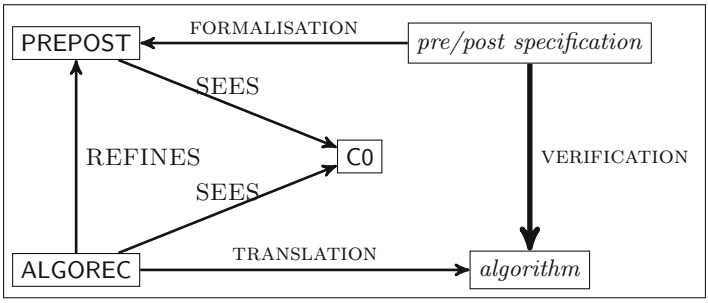


Fig. 3. The recursive pattern

Since the refinement-based process requires an idea for introducing more concrete events. A very simple and powerful way to refine is to introduce a more concrete model which is based on an inductive definition of outputs with respect to the input.

A first consequence is that the concrete model is containing events which are computing the same function but corresponding to a recursive call expressed as events (**EVENT** rec%PROC(h(x),y)%P(y)). The event **EVENT** rec%PROC(h(x),y)%P(y) is simply simulating the recursive call of the same function and this expression makes the proofs easier. The invariant is defined in a simpler way by analysing the inductive structure and a control variable is introduced for structuring the inductive computation. We have identified three possible events to use in the concrete model:

EVENT
e
WHERE
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x)$
END

EVENT
rec%PROC(h(x),y)%P(y)
ANY y
WHERE
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END

EVENT
call%APROC(h(x),y)%P(y)
ANY y
WHERE
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END

6 Applying the Recursive Pattern

Applying the recursive pattern is made easier by the first steps of the iterative pattern. In fact, the context C0 and the machine PREPOST are the starting points of the iterative pattern as well as the recursive pattern. We use the computation of the function x^2 and we obtained the following refinement of PREPOST. Figure 2 is the diagram analysing the way to solve the computation of the value of $u(x)$ following the call-as-event paradigm.

MACHINE *square*
REFINES *specquare*
*Doub*SEES *control0*
VARIABLES
r, c, tr
INVARIANTS
$inv1 : r \in \mathbb{N}$
$inv2 : c = end \Rightarrow r = n * n$
$inv3 : c = callrec \Rightarrow n \neq 0$
$inv4 : c = callrec \Rightarrow tr = (n-1) * (n-1)$
$inv5 : c \in C$
$inv6 : tr \in \mathbb{N}$
$inv7 : c = end \Rightarrow r = n * n$
$inv8 : c = end \wedge n \neq 0$
$\Rightarrow tr = (n-1) * (n-1) \wedge r = tr + 2 * (n-1) + 1$
$inv9 : c = callrec \Rightarrow n * n = tr + 2 * (n-1) + 1$

EVENT INITIALISATION
BEGIN
$act1 : r := 0$
$act2 : c := start$
$act3 : tr :\in \mathbb{N}$
END
EVENT square0
REFINES square(n;r)
WHEN
$grd1 : c = start$
$grd2 : n = 0$
THEN
$act1 : c := end$
$act2 : r := 0$
END

EVENT squaren
REFINES square(n;r)
WHEN
$grd1 : c = callrec$
THEN
$act1 : r := tr + 2 * (n-1) + 1$
$act2 : c := end$
END
EVENT rec%square(n-1;tr)
WHEN
$grd1 : c = start$
$grd2 : n \neq 0$
THEN
$act1 : c := callrec$
$act2 : tr := (n-1) * (n-1)$
END

The variable c is modelling the control in the diagram. We introduce control points corresponding to assertions in the labels of the diagram as $C = \{start, end, callrec\}$. Three events are defined and the invariant is written very easily and proofs are derived automatically. The event rec%square(n-1;tr) is the key event modelling the recursive call. In the current example, we have modified the machine by using directly the fact that $v(n) = n * n$ and normally we had to use the sequence following the recursive pattern and then we had to derive the theorem $v(n) = n * n$.

Proofs are simpler and invariants are easier to extract from the inductive definitions. The use of Frama-C shows that the proofs are also very simple in the case of a recursive algorithm. Missing expertise in using Frama-C leads to the introduction of auxiliary lemmas as $((x-1)+1)^2 = (x-1)^2 + 2x + 1$. In this example, we do not use the event like call%APROC(h(x),y)%P(y) but the event is clearly a call for another procedure or function. For instance, when a sorting algorithm is developed, you may need an auxiliary operation for scanning a list of values to get the index of the minimum. It means that we have a way to define a library of models and to use correct-by-construction procedures or functions. In [15], we detail the tool and the way to define a library of *correct-by-construction programs*.

7 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. Jones [23] develops the rely/guarantee concept for handling (possible and probably wanted) interferences among sequential programs. Rely/guarantee intends to make *implicit* [4,12] interferences as well as cooperation proofs in a proof system. In other methods as Owicki and Gries [31], the management of non-interference proofs among annotated processes leads to

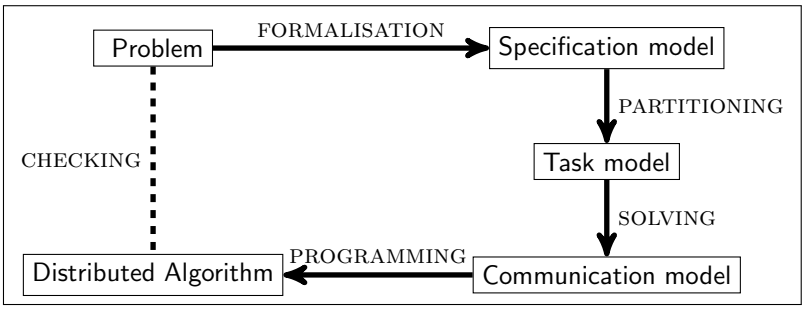
a important amount of extra proof obligations: checking interference freeness is explicitly expressed in the inferences rules. When considering an event as modelling a call of function or a call of a procedure, we implicitly express a computation and a sequence of state. We [30] propose a temporal extension of Event-B to express liveness properties. The extension is a small bridge between Event-B and TLA/TLA$^+$ [24] 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 [5,6], 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. [26]. The self-healing problem is belonging to the larger class of self-$\star$ systems [18].

In previous patterns, we identify one event which *simulates* the execution of an algorithm either as an iterative version or as a recursive version. Figures 1 and 3 separate the problem to solve into three problem domains: the domain for expressing pre/post specifications, the domain of Event-B models and the domain of programs/algorithms. The translation function generates effective algorithms producing the same traces of states. We are now introducing patterns which are representatives of the service-as-event paradigm.

7.1 The PCAM Pattern

Coordination [14] is a paradigm that allows programmers to develop distributed systems; web services are using this paradigm for organising interactions among services and processes. In parallel programming, coordination plays also a central role and Foster [20] has proposed the PCAM methodology for designing concurrent programs from a problem statement: PCAM emphasizes a decomposition into four steps corresponding to analysis of the problem and leading to a machine-independant solution. Clearly, the goal of I. Foster is to make concurrent programming based on abstractions, which are progressively adding details leading to specific concurrent programming notation as, for instance MPI (http://www.open-mpi.org/). The PCAM methodology identifies four distinct stages corresponding to a Partition of identified tasks from the problem statement and which are concurrently executed. A problem is possibly an existing complex C or Fortran code for a computing process requiring processors and concurrent executions. Communication is introduced by an appropriate coordination among tasks and then two final steps, Agglomeration and Mapping complete the methodology steps. The PCAM methodology includes features related to the functional requirements in the two first stages and to the implementation in the two last stages. I. Foster has developed the PCAM methodology together with tools for supporting the implementation of programs on different architectures. The success of the design is mainly due to the coordination paradigm which allows us to freely organise the stages of the development.

The PCAM methodology includes features related to the functional requirements in the two first stages and to the implementation in the two last stages. The general approach is completely described in [28].



We consider the two first stages (Partitioning, Communication) for producing state-based models satisfying functional requirements and which will be a starting point for generating a concurrent program following the AM last suffix. We have described a general methodology for developing correct by construction concurrent algorithms and we have developed a solution specified by a unique event.

7.2 The Distributed Pattern

Section 5 introduces the call-as-event paradigm which is based on an implicit relationship between a procedure/function call and an event. The main idea is to analyse a problem as a pre/post specification which is then refined by a machine corresponding to the simulation of a recursive function or procedure. The class of algorithms is the class of sequential algorithms and there is no concurrent or distributed interpretation of an event. However, 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.

Figure 4 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 the eventuality of e by a

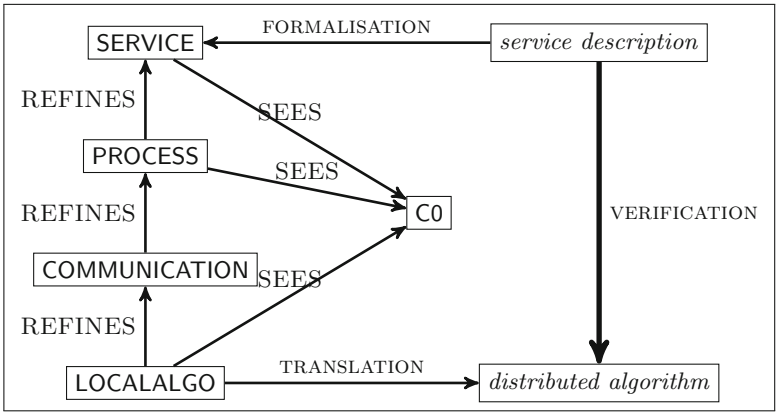
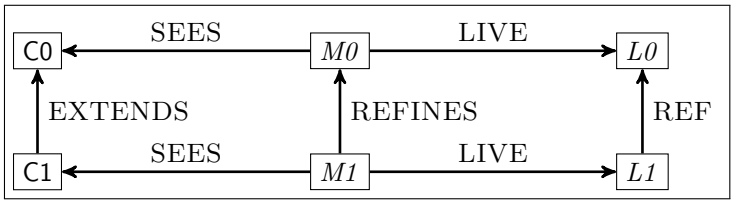


Fig. 4. The distributed pattern

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 [30]. The link called LIVE expresses the satisfaction relationship. The next diagram is summarising the relationship among models.



Liveness properties can be gathered in *assertions diagrams* which are already used for the recursive pattern in Fig. 2. For instance, $P \xrightarrow{e} Q$ 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:

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

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 ($SF_x(\mathsf{e_i})$).

If we consider the leader election protocol [3], 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
$sp, rt :\mid spanning(sp', rt', g)$
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. Liu et al. [25] 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.

7.3 Applying the Distributed Pattern

The distributed pattern (Fig. 4) is applied for the famous *sliding window protocol*. The *service description* is expressing that a process P is sending a file IN to a process Q and the received file is stored in a variable OUT. The service is simply expressed by the liveness property $(at(P, s) \wedge IN \in 0..n \rightarrow D) \rightsquigarrow (at(Q, r) \wedge OUT = IN)$ and the event **EVENT** communication $\widehat{=}$ **WHEN** $at(P, s) \wedge IN \in 0..n \rightarrow D$ **THEN** $OUT := IN$ **END** is defining the service. $at(P, s)$ means that P is at the sending statement called s and $at(Q, r)$ means that Q is at the receiving statement r. The context C0 and the machine SERVICE are defined in Fig. 4. The next step is to decompose the liveness property using one of the possible inference rules of the temporal framework as transitivity, induction, confluence of the *leadsto* operatior. In this configuration, we have to introduce the computation process which is simulating the protocol. Obviously, we use and induction rule to express that te file IN is sent item per item and we introduce sending and receiving events and the sliding events. In the new machine PROTOCOL, variables are OUT, i, chan, ack, got and satisfied the following invariant:

INVARIANTS

$inv1 : OUT \in \mathbb{N} \nrightarrow D$
$inv2 : i \in 0 \,..\, n + 1$
$inv3 : 0 \,..\, i - 1 \subseteq dom(OUT) \wedge dom(OUT) \subseteq 0 \,..\, n$
$inv7 : chan \in \mathbb{N} \nrightarrow D$
$inv8 : ack \subseteq \mathbb{N}$
$inv9 : got \subseteq i \,..\, i + l \cap 0 \,..\, n$
$inv10 : got \subseteq \mathbb{N}$
$inv12 : dom(chan) \subseteq 0 \,..\, i + l \cap 0 \,..\, n$
$inv13 : got \subseteq dom(OUT)$
$inv14 : ack \subseteq dom(OUT)$
$inv16 : 0 \,..\, i - 1 \lhd OUT = 0 \,..\, i - 1 \lhd IN$
$inv17 : chan \subseteq IN$
$inv18 : OUT \subseteq IN$
$inv19 : ack \subseteq 0 \,..\, i + l \cap 0 \,..\, n$

Name	Total	Auto	Inter
isola-swp	124	101	23
C0	1	1	0
SERVICE	4	2	20
PROCESS	63	51	12
WINDOW	19	13	6
BUFFER	21	18	3
LOCAL	16	16	0

The variable *got* is simulating a window identified by the values between i and i+l in the variables chan, got and ack. The sliding window is in fact defined by the variable i which is sliding or incrementing, when the value OUT(i) is received or equivalently when *iinack*. The events are send, receive, receiveack, sliding together with events which are modelling possible loss of messages. The machine PROCESS is simulating the basic mechanism of the sliding window protocol and is expressing the environment of the protocol. The next refinement WINDOW is introducing an explicit *window* variable satisfying the invariant $w \in \mathbb{N} \nrightarrow D \wedge w \subseteq chan \wedge dom(w) \subseteq i..i+l$. The events are enriched by guards and actions over the variable *window*. The *window* variable is still an abstract view of the window which is contained in a buffer b. The buffer b is introduced in the refinement called BUFFER. The new variable b is preparing the localisation and introduced the explicit communications: $b \in 0 \,..\, l \nrightarrow D \wedge \forall k \cdot k \in dom(b) \Rightarrow i + k \in dom(w) \wedge b(k) = w(i + k) \wedge \forall h \cdot h \in dom(w) \Rightarrow h - i \in dom(b) \wedge w(h) = b(h - i)$. The visible variables of the machine are OUT, i, chan, ack, got, w and b and in the next refinement, we obtain a local modle called LOCAL with OUT, i, chan,

ack, got and b: the window is not part of the implementation of the protocol. The events are localised by hiding the variable w and the final model can now be transformed into the Sliding Window Protocol. The proof obligations summary shows that proof obligations for the machine PROCESS correspond to the main effort of proof, when the induction is introduced. However, we have not checked the liveness properties using the temporal proof system namely TLA and it remains to be effectively supported by the toolbox for TLA/TLA$^+$. We use the temporal proof rules to as guidelines for decomposing liveness properties while we are refining events in Event-B. The technique has been already used for developing population protocols [30].

8 Conclusion and Perspectives

The refinement-based modelling technique combines modelling and proving through discharging proof obligations. Our contribution is to assist anyone who wants to obtain a completely checked Event-B project for a given problem with less toil. The toil is related to the use of the RODIN platform. The tool is a real and useful proof companion but it requires a specific skill in proof development.

Following the ideas of Pólya, we enrich the library of patterns for providing guidelines for defining fully proved Event-B models, when considering problems to solve defined by explicit inductive definitions. The iterative pattern (Fig. 1) is not defining the solution of the problem and it requires to prove that the computed term of the sequence is satisfying the postcondition. In our example, we have to prove the property $v(n) = n^2$ which is the key of the verification process with the definition of an invariant. The iterative pattern gives a very general invariant which should be improved for the specific problem. The summary of proof obligations shows that the refinement helps in the proof process. The proof of the property $v(n) = n^2$ is probably the most complicated task and the user is focusing on this main question. The invariants of the Event-B models can be reused in the verification using Frama-C, for instance, and the verification of the resulting algorithm is a confirmation of the translation.

The recursive pattern (Fig. 3) gives also a different way to discover the invariant and to discharge generated proof obligations. It improves the proof process as well as the definition of the invariant which is a reformulation of the inductive definition. The relationship between the iterative invariant and the recursive invariant is to explicit but a perspective is to have an effective process for deriving the iterative invariant from the recursive invariant. The motivation is to help in the definition of iterative solution. Another choice was to prove that the translation of a recursive solution into an iterative solution is correct and we used this argumentation in our paper in the call-as-event technique [29].

The call-as-event paradigm (Fig. 3) is generalised by the service-as-event paradigm which is based on a correspondence between temporal *leadsto* operator and events. The idea is to generalise the notion of call by the notion of service which is more appropriate for modelling distributed applications. The idea is based on the use of a graph of assertions (see Fig. 2 for the recursive pattern). Such a graph is close to the proof lattice of Owicki and Lamport [32] and

we have developed self-⋆ systems using these diagrams and the service-as-event paradigm [5]. The Event-B models contain both events corresponding to the system under development. and the environment. When considering the service-as-event paradigm, liveness properties play the role of guidelines for refining machines.

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. 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 be improved and we plan to explore distributed programming languages with a high level of abstraction as for instance DistAlgo [25].

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Abrial, J.-R., Cansell, D., Méry, D.: A mechanically proved and incremental development of IEEE 1394 tree identify protocol. Formal Asp. Comput. **14**(3), 215–227 (2003)
4. Ameur, Y.A., Méry, D.: Making explicit domain knowledge in formal system development. Sci. Comput. Program. **121**, 100–127 (2016)
5. Andriamiarina, M.B., Méry, D., Singh, N.K.: Analysis of self-* and P2P systems using refinement. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 117–123. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_9
6. Andriamiarina, M.B., Méry, D., Singh, N.K.: Revisiting snapshot algorithms by refinement-based techniques. Comput. Sci. Inf. Syst. **11**(1), 251–270 (2014)
7. Back, R.J.R.: On correct refinement of programs. J. Comput. Syst. Sci. **23**(1), 49–68 (1979)
8. Back, R.J.R.: A calculus of refinements for program derivations. Acta Inform. **25**, 539–624 (1988). https://doi.org/10.1007/BF00291051
9. Bjorner, D.: 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. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31288-9
10. Bjorner, D.: Software Engineering 2 Specification of Systems and Languages. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). ISBN 978-3-540-21150-1
11. Bjorner, D.: Software Engineering 3 Domains, Requirements, and Software Design. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-33653-2. ISBN 978-3-540-21151-8

12. Bjørner, D.: Domain analysis & description - the implicit and explicit semantics problem. In: Laleau, R., Méry, D., Nakajima, S., Troubitsyna, E. (eds.) Proceedings Joint Workshop on Handling IMPlicit and EXplicit Knowledge in Formal System Development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD). Electronic Proceedings in Theoretical Computer Science, Xi'an, China, 16 November 2017, vol. 271, pp. 1–23. Open Publishing Association (2018)
13. Cansell, D., Paul Gibson, J., Méry, D.: 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, pp. 329–338. IEEE Computer Society (2007)
14. Carriero, N., Gelernter, D.: A computational model of everything. Commun. ACM **44**(11), 77–81 (2001)
15. Cheng, Z., Méry, D., Monahan, R.: On two friends for getting correct programs - automatically translating event B specifications to recursive algorithms in rodin. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I. LNCS, vol. 9952, pp. 821–838. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_57
16. Clearsy System Engineering. Atelier B (2002). http://www.atelierb.eu/
17. Clearsy System Engineering. BART (2010). http://tools.clearsy.com/tools/bart/
18. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
19. Fakhfakh, F., Tounsi, M., Mosbah, M., Méry, D., Kacem, A.H.: Proving distributed coloring of forests in dynamic networks. Comput. Sist. **21**(4), 863–881 (2017)
20. Foster, I.T.: Designing and Building Parallel Programs - Concepts and Tools for Parallel Software Engineering. Addison-Wesley, Reading (1995)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
22. Hoang, T.S., Fürst, A., Abrial, J.-R.: Event-B patterns and their tool support. Softw. Syst. Model. **12**(2), 229–244 (2013)
23. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
24. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
25. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. ACM Trans. Program. Lang. Syst. **39**(3), 12:1–12:41 (2017)
26. Marquezan, C.C., Granville, L.Z.: Self-* and P2P for Network Management - Design Principles and Case Studies. Springer Briefs in Computer Science. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4471-4201-0
27. Méry, D.: Refinement-based guidelines for algorithmic systems. Int. J. Softw. Inform. **3**(2–3), 197–239 (2009)
28. Méry, D.: Playing with state-based models for designing better algorithms. Future Gener. Comp. Syst. **68**, 445–455 (2017)
29. Méry, D., Monahan, R.: Transforming event B models into verified c# implementations. In: Lisitsa, A., Nemytykh, A.P. (eds.) First International Workshop on Verification and Program Transformation, VPT 2013. EPiC Series in Computing, Saint Petersburg, Russia, 12–13 July 2013, vol. 16, pp. 57–73. EasyChair (2013)
30. Méry, D., Poppleton, M.: Towards an integrated formal method for verification of liveness properties in distributed systems: with application to population protocols. Softw. Syst. Model., 1–33 (2015)
31. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inform. **6**, 319–340 (1976)

32. Owicki, S.S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. **4**(3), 455–495 (1982)
33. Pólya, G.: How to Solve It. Doubleday, Garden City (1957)
34. The Frama-C Development Team. Frama-C. CEA. https://frama-c.com/