Livrable 3
Proof-based design patterns

August 6, 2008:9:55 P.M.

Projet RIMEL

ANR-06-SETI-015

Equipe MOSEL, LORIA, Université Henri Poincaré Nancy 1
ClearSy
LABRI, Université de Bordeaux & CNRS
http://rimel.loria.fr

Années 2007-2008-2009

Avertissement
The following report has been written by Nazim Benaïssa and Dominique Méry and more precisely:

- The chapter THE *parametric/generic* PATTERN reuses results published in two papers written by Dominique Cansell, Paul Gibson and Dominique Méry.
- The chapter THE *call as event* PATTERN has been completely written by Dominique Méry from the paper [63].
- The chapter THE *access control* PATTERN has been written by Nazim Benaïssa and Dominique Méry from previous publications.
- The chapter THE *cryptographic* PATTERN has been wrioten by Nazim Benaïssa.
- The introduction and the conclusion are written by Dominique Méry.

Contents

List of Figures

Chapter 1

Introduction

Sommaire

1.1 Motivations

Designers can now justify design decisions (and the claims for their designs of being high quality) based on the patterns that they have (or have not) applied. In fact, much like the quality of code can easily be judged by a quick glance at whether certain coding standards have been followed, so the quality of a design can now be judged based on the use of well-understood patterns (and architectures) [43]. Good software design tools have given rise to the notion of design patterns - where expertise is wrapped up in re-usable form and support provided for the (semi-automated) application of these patterns. This now leads us on to validation and verification where we see analogous concepts:

- Validation and verification is difficult and refinement is a tool for supporting the engineering of correct systems through V & V. However, this tool - like many others - can only be effective if used in an expertly manner. Such expertise should be packaged for re-use (where possible) in the form of design patterns.
- These patterns should be targeted at formal verification and rigorous validation, to provide higher-level tools whose use will justify claims for security and trustworthiness. Of course, building such patterns will require theoretical innovations.

We think that *formal design patterns* or *proof-based design patterns* will play a vital role in any future measurements with respect to claims of security and trustworthiness based on verification. The key point is that proof-based design patterns have a very important feature: they are based on an objective way to ensure the validity of the resulting objects. The intention is to capture proof-based information about the system under development.
The concept of design pattern [43] is well known in Object Oriented Technology. The main idea is to have some sort of reproducible engineering micro-design that the software designer can use in order to develop new pieces of software. Abrial has defined (and proved) two formal patterns, where the second one happens to be a refinement of the first one and he has derived them from a study of a mechanical press. The mechanical press case study was proposed by INRS, which is a national institute on safety and security: two patterns emerge from the study and the action/reaction paradigm has been implemented in such a way. As a matter of fact, one very often encounters such patterns in the development of reactive systems, where several chains of reactions can take place between an environment and a software controller, and vice-versa.Second, the patterns are then used many times in the development of a complete reactive system where several pieces of equipment interact with a software controller. It results in a systematic construction made of six refinements. The entire system is proved completely automatically. The relationship between the formal design patterns and the formal development of the problem at hand will be shown to correspond to a certain form of refinement.
Abrial [8] and the project think that such an approach to formal developments can be generalized fruitfully to other patterns. It results in very concise and systematic developments. Let us detail the questions to be addressed in a system engineering approach using proof and refinement.
G. Polya [68] describes a set of techniques or recipes which can be used to solve problems. The different steps advocated by Polya can be summarised as follows. A first step is the understanding the problem and list the data, conditions on the data, the unknown elements and the feasibility of the requirements listed in the statement of the problem. It is clearly important to identify the redundancy and the possible inconsistencies; elicitation of requirements is clearly a very important step and it can be driven following an incremental and progressive methodology based on proof checking. The methodology can be based on drawings or graphical notations, which should be suitable and it is then clear that the role of graphical notations is central. Our expertise is on the definition of graphical notations which are *sound abstractions* of the systems under development as for instance the predicate diagrams [20, 39, 29].
Predicate diagrams constitute a framework which can be combined with textual notations for event-based systems and which can be used to help in the statement of the problem. However, the link between the problem and the first model remains to be elicitated; the notion of view is more appropriate than model and the model is the global integration of the different views. Following the thesis of Polya, it appears that the link between the data and the unknown should be defined in an appropriate way. However, Polya mentions the use of auxiliary problems or sub-problems and it leads us to discover new solutions to the given sub-problems or to reuse existing problems having already been solved: the identification of an already seen problem is something that is not so easy to carry out. The identification of a new problem with

a given existing (well-understood) problem is possible in a cognitive way but is not yet so obvious in a mathematical and logical framework. The identification of a problem is the question to be addressed; where the classification of the problem seems to be related to the solution and to the statement of the solution. For instance, the problem of searching something in a collection of data or the problem of computing a fixed-point over a structure can be formalised in a mathematical way and an algorithmic solution can be found; the problem of modelling the greedy method is more complex to solve because there are a lot of greedy algorithms which use the same principle and the question is to be able to provide a general framework for solving this problem on a special case. We should be able to produce a plan of the solution where it is made up of event-based models related by the refinement relationship and of logico-mathematical theories on data. The question is then to check if a problem is identical to another problem or if a problem is a weaker or a stronger instance. Clearly, refinement plays the role of a link between problems as long as we are able to attach a model to a problem. Moreover, the model should be as general as possible and reusable. The question of the analogy among problems is also a very important issue and it is related to an adaptation of the refinement.
The progress should be on the coding or the formalisation of development and design patterns have shown their adequacy in software engineering: the weak point is the link to the proof and verification engineering. As demonstrated by our partner from ClearSy, the B method provides a framework for expressing models smartcard security policies (EAL5+ level / Common Criteria) and it is clearly a point to integrate to our patterns. The main question is to be able to capture justifications of the adequacy to safety and security requirements.
Concepts integrating proofs, should be introduced to guarantee correctness criteria (safety and security) in a refinement-based development process. In previous works, we have already identified techniques for paramatrised development of B models; formal verification of *tamper-evident* storage for e-voting [22] and the incremental parametric development of greedy algorithms [27], are applying a technique jointly developed with J.-R. Abrial [9]. The general approach is based on the development of case studies followed by an identification of possible patterns.

1.2 Summary of the report

The current report gathers patterns identified during the last 18 months of the RIMEL project. Let us recall that we had no formal definition of what is a proof-based design pattern and we follow a general methodology including an identification phase, a definition (or statement) phase and a replay phase. One should also warn on the implementation phase of the pattern and it is related to the target platform. We have to targets platforms: the RODIN platform [69] which is provided freely by the RODIN project and the Atelier B platform [31] which will be freely available. Four patterns are presented in the document and have not the same degree of generality:

- The *parametric/generic* pattern provides a way to instantiate Event B development and we apply it on two very different case studies: the design of greedy algorithms [27] and the design of e-voting algorithms [22, 21]. The technique is very close to the instantiation of discrete models due to Abrial and Hallestede [9] and we had worked partially with J.-R. Abrial on this technique.

- The *call as event* pattern is based on the relation between an event and the call of a procedure; the relation was first introduced in a paper [28] and has been formalised in [63].

- The *access control* pattern is very close to the *parametric/generic* pattern but it relates access control models and secure systems.

- The *cryptographic* pattern is probably the most recent pattern; it provides a way to handle attacker models like Dolev-Yao [38] and to take into account the attacker model when developing cryptographic protocols.

The three phases for each pattern provides a way to validate them but it remains to implement these patterns. Bart [33] provides a set of transformations for deriving an implementation in the Aterlier B platform and in the (classical) B method. We have not yet studied carefully how Bart can be used to implement our transformations but it has already been used by Siemens to develop new applications based on previously defined applications. A set of rules is necessary and it is then clear that these current rules are proprietary and should be developed by each developer according to needs. In the next chapters, we describe each pettern and in the final chapter we will conclude on the limits of the work and the perspectives of these initial results.

Chapter 2

The *parametric/generic* pattern

Sommaire

The Event B method provides a general framework for modelling both data structures and algorithms. B models are validated by discharging proof obligations ensuring safety properties. We address the problem of development of greedy algorithms using the seminal work of S. Curtis; she has formalised greedy algorithms in a relational calculus and has provided a list of results ensuring optimality results. Our first contribution is a re-modelling of Curtis's results in the Event B framework and a mechanical checking of theorems on greedy algorithms The second contribution is the reuse of the mathematical framework for developing greedy algorithms from Event B models; since the resulting Event B models are generic, we show how to instantiate generic Event B models to derive specific greedy algorithms; generic Event B developments help in managing proofs complexity. Consequently, we contribute to the design of a library of proof-based developed algorithms.

2.1 Genericity and refinement in Event-B: a formal design pattern

The Event-B method provides a framework for developing generic models of systems, where a *problem* can be defined using parameters to be instantiated. Intuitively, this means that we are able to relate the current *problem* to be solved to an abstract *problem* solved by an already existing generic B development (theory). Following our approcah, in the existing generic solution we must find the mathematical framework that is common to both *problems*, together with some constants which need to be instantiated. Consequently, in formulating the solution to the new *problem* the main work is to check that the instantiated parameters satisfy the constraints of the generic *problem* theory.

2.1.1 Projects

The development of a fully formal generic modelling mechanism for the B event-based method is work in progress. In the following, we indicate how the current framework can be used for implementing the instantiation.
The key concept is that of a validated project: a collection of models, either machine or refinement or implementation, which are completely verified through type checking and theorem proving. For simplicity, and without loss of generality, we assume that there is only one machine in the current project, allowing us to focus on the re-usability of developed models. Hence, a project $\mathbb{G}$ is roughly speaking an acyclic directed graph of models related by the refinement relationship: $\mathbb{G} = (\{G, ..., G_n\}, \longrightarrow)$.

2.1.2 The General Model

In the following, in order to avoid confusion among names, we use different fonts for designating problems, models and projects. The creation and the development of the project $\mathbb{G}$ follows the event B methodology. We assume that $\mathbb{G}$ is an existing project corresponding to a given *generic* problem, denoted $\mathcal{G}$. The model G (see the template specification on the left of figure 1) is, in fact, the formal statement of the generic problem: it incorporates relevant aspects of the generic problem — at a high level of abstraction — in an initial model. This initial abstraction can be thought of as defining the scope of the problem and the behavioural properties that require validation.
The model G provides a general framework for the current problem; the problem is characterized by a theory defined by the clauses **sets**, **constants** and **properties**. The unique $event$ helps in solving the problem by defining the problem in an abstract form: saying what is required rather than how the solution is to be implemented. Intuitively this corresponds to the problem being viewed as pre/post-condition relation between an initial state and a final state which is arrived at after executing the single $event$. Of course, refinement permits us to move from this "magical" one-step functional view of the required system's behaviour to a richer multi-step view.
The generic model G is the starting point of the development of the project $\mathbb{G}$: it solves the problem $\mathcal{G}$; and the project is formally checked by the theorem prover. Constants in G can be instantiated, but proof obligations must be established to ensure the validity of properties in the instantiated model, which correspond to theorems in an instantiation.
Before we introduce the instantiation and refinement steps in our design pattern, we motivate the need for such a pattern. In the general *correct-by-construction* refinement-based development process, working with concrete models often leads to refinements generating large numbers of proof obligations that cannot be

model
G
sets
s
constants
c
properties
$P(s, c)$
variables
x
invariant
$J(x)$
assertions
$A(x)$
initialisation
$S(x)$
events
$event = L(x)$
end

model
H
sets
t
constants
d
properties
$Q(t, d)$
variables
y
invariant
$I(y)$
assertions
$C(y)$
initialisation
$spec\_init(y)$
events
$spec\_event = K(y)$
end

Figure 2.1: Definition of models

discharged automatically. The main goal during development is to find a *good* refinement path: a short sequence of refinement steps where a small number of proof obligations are generated at each step and which are discharged automatically. Finding such a path usually requires reformulating or restructuring of invariants, together with changes to the degree of detail (abstraction) in the models, and is very challenging for non-experts. Thus, we would like to package such expertise in a re-usable construct, which we call a formal design pattern.

In order to manage the complexity of the refinement path, one common approach (used by experts) is to find a more generic representation of their initial problem where details are hidden by constants requiring instantiation. Then, in general, the refinement path is much simpler to establish as it requires fewer interactive proofs and leads to a correct concrete generic solution. To prove that this generic solution can be re-used, through instantiation, to solve the initial concrete problem, two final steps are required. First, one must show that the initial problem model is a correct instantiation of the generic problem model; and secondly, one must show that the final solution model is a correct instantiation of the generic solution model.

In the best case, the generic refinement path has already been established and can be re-used directly. In the worst case, this path has to be developed from scratch. However, even in this worst case, developing the path at a higher level of abstraction (and proving 2 instantiations to be correct) is much easier[1] than developing the path at the lower level of abstraction.

Thus, our formal design pattern is a re-usable solution to a common design problem that can be exploited by formal developers who are not necessarily expert. This re-use requires only that the developers understand instantiation and refinement.

2.1.3 Instantiation

Consider a specific problem $\mathcal{H}$ in project $\mathbb{H}$, say. We specify the specific requirements as a new model H (see the template specification on the right of figure 1). In order to exploit our design pattern, we wish to establish that an instantiation of the generic project $\mathbb{G}$ corresponds to the given problem $\mathcal{H}$;

The instantiation of the generic project $\mathbb{G}$ for the generic problem $\mathcal{G}$ to solve the specific problem $\mathcal{H}$ consists of exhibiting a set term $\sigma(t, d)$, defined in terms of the set t and constant of d, and also a similar constant term $\gamma(t, d)$ for instantiating the constant c of G. Thus, the instantiation consists of repainting G with

[1]We have no formal metric for the complexity of a refinement path; however, intuitively a path is simpler if there are fewer proof obligations that require interactive proof as they cannot be discharged automatically.

$\sigma(t, d)$ and $\gamma(t, d)$ and to invoke it as $G\,(\sigma(t, d), \gamma(t, d))$. We must also rename each variable (resp. event) of G by a unique variable (resp. event) of H[2]. This instantiation must resolve the specific problem $\mathcal{H}$ and we propose to instantiate a development path through the refinement of H.

2.1.4 Proof Obligation of an Instantiation

Now, proof obligations of any subsequent refinement assume that the instantiated development solves the specific problem $\mathcal{H}$. The next refinement step captures the semantics of how the specific problem $\mathcal{H}$ is solved in the same way as the problem $\mathcal{G}$, after a suitable instantiation. When the instantiation is proved to be correct, we freely obtain a complete instantiated development for the new problem $\mathcal{H}$. An instantiation requires one only to prove that the properties of the system G are theorems with respect to the properties of H. We do this in two steps:

- (1) The properties of the system G, i.e. axioms defining the theory of G, are theorems in the new theory defined by the problem H:

$$\boxed{Q(t, d) \;\Rightarrow\; P(\sigma(t, d), \gamma(t, d))}$$

- (2) both models are solving the same problem and the event $spec\_event$ of H is refined by the instance of the event $event$ of G for the problem H:

$$\boxed{\begin{array}{l} Q(t, d) \;\wedge\; P(\sigma(t, d), \gamma(t, d)) \;\wedge\; I(y) \;\wedge \\ [s, c := \sigma(t, d), \gamma(t, d)]J(y1) \wedge y = y1 \wedge \\ R(L)(y1, y1') \\ \Rightarrow \\ \exists y'.(I(y') \wedge R(K)(y, y')) \end{array}}$$

Once we establish these two steps, we have a formal verification of the correctness of our concrete solution with respect to the already existing generic project:

Property 1 *When the given (previous) proof obligations are proved, the new problem $\mathcal{H}$ is solved by the development of the problem $\mathcal{G}$, up to renaming and instantiation.*

2.1.5 A formal design pattern

When the refinement is proved, the new problem $\mathcal{H}$ is solved by the development of the problem $\mathcal{G}$, upto renaming and instantiation. A new project $\mathbb{H}$ is created from the project $\mathbb{G}$ of the problem $\mathcal{G}$: events are renamed, variables are renamed, instantiations are done. Parameters are not necessarily completely instantiated or renamed: if a parameter is not instantiated then it keeps properties stated in the general model and no new proof obligation is generated. We illustrate this formal design pattern in the following diagram:

$$\begin{array}{ccc}
 & & H \\
 & & \blacktriangledown \\
G & \blacktriangleright & H.G \\
\blacktriangledown & & \triangledown \\
G_1 & \triangleright & H.G_1 \\
\blacktriangledown & & \triangledown \\
 & \triangleright & \\
\blacktriangledown & & \triangledown \\
G_n & \blacktriangleright & H.G_n \\
 & & \blacktriangledown \\
 & & H.G_{n+1}
\end{array}$$

The diagram tells us where proof obligations must be proved: filled (triangular arrow) symbols show that new proof obligations are generated and require proving; non-filled symbols show that proof obligations have been generated but their proofs are inherited from the previous project ($\mathbb{G}$). Horizontal arrows represent instantiation. Vertical arrows represent refinement. The proof that model $H.G_{n+1}$ is a correct solution to the problem H is simplified by re-use of the refinement path in project $\mathbb{G}$.

[2]We can assume that H and G have no common parameters: x is different from y and events names are different.

2.2 Application 1: Greedy algorithms

2.2.1 Introduction

Algorithms provide a class of systems on which one can apply proof-based development techniques like the event B method, especially the refinement. The main advantage is the fact that we teach data structures and algorithms to students, who should have simple explanations of why a given algorithm is effectively working or why some assertion is an invariant for the algorithm under consideration …Hence, we have a good knowledge of algorithmic problems and it is simpler for us to apply proof-based development techniques on the algorithmic problems. Greedy algorithms constitute a well defined class of algorithms (applications and properties) and we aim to provide proof-based patterns for facilitating the proof-based development (in B) of greedy algorithms.

In a previous work [7], we have developed Prim's algorithm and we have proved properties over trees: the inductive definition of trees helps in deriving intermediate lemmas asserting that the growing tree converges to the minimal spanning tree, according to the greedy strategy. The resulting algorithm was completely proved using the proof assistant [32] and we can partially reuse current developed models to obtain Dijkstra's algorithm or Kruskal's algorithm. The greedy strategy is not always optimal and the optimality of the resulting algorithm is proved by the theorem 24.1 of Cormen's book [34] in the case of the minimal spanning tree problem. The gain is clear, since we had a mechanised and verified proof of Prim's algorithm. The formalisation of greedy-oriented algorithmic structures was not so complicated but we were assuming that a general theory on greedy structures could help in designing our greedy algorithms using the event B method. Fortunately, S. Curtis [36] brings the theoretical material that was missing in our project; she has formalised in a relational framework properties required for leading to the optimality of solutions, when applying a greedy technique. However, we have not explained why we are choosing the greedy method and what for? Our quest is to propose general proof-based developments (or patterns) for a given problem or for a given paradigm. We think that the refinement provides a way to introduce generic elements in developed models. A second objective is to illustrate the adequacy of the B prover [32], when checking results over set-theoretical structures; in a sense, our work may seem to be a plagiarism of Curtis's paper, but the tool scans each detail to check and it validates each user hint, and, generally, there is no assisted significant proof without human hint (proof step or tricky lemma). Hence, our text is an exercise in checking properties over greedy structures and in proposing generic development of greedy algorithms; we do not know any other mechanized complete proof-based developments of greedy algorithms.

Greedy algorithms are used to solve optimization problems like the shortest path problem or the best order to execute a set of jobs. A greedy algorithm works in a local step to satisfy a global constraint. A greedy algorithm can be summarized by the general algorithm 1, where C is the set of candidates and S is the set containing the solution or possibly no solution. The goal is to optimize a set of candidates which is a solution to the problem; the optimization maximizes or minimizes the value of an objective function. The optimization state is checked by the Boolean function called $goodchoice$. Lectures notes of Charlier [30] provide a very complete introduction to the underlying theories of the greedy algorithms like the matroids theory for instance. S. A. Curtis [36] classifies greedy algorithms and uses a relation framework for expressing properties of the greedy algorithms; her characterization is based on the preservation of the safety properties but the termination part is missing. Our development is based on her works, it reformulates properties and proposes mechanically checked proofs in the B prover engine [6, 32]. First, we translate the mathematical notations of the models of Curtis; we check the results proved in the paper using the theorem prover of B. Then we show how to develop a greedy algorithm according to a given assumption. Finally, we show how to instantiate a specific problem that can be solved using the greedy strategy.

2.2.2 Mathematical structures for the greedy method

First, in the step-by-step development, the definition of the mathematical objects requires to ensure the existence of a solution and to identify the problem to solve. The greedy method is effective, when properties are satisfied by the underlying mathematical structures. Like S. Curtis [36], we first define operators over relations and then we prove properties related to the greedy method. In fact, it will be a checking phase of Curtis's results: we use a more conventional way to write set-theoretical objects in the B notation.

Algorithm 1: General Greedy Algorithm [34]

Pre-condition: *C is a set of possible candidates*
Post-condition: *Either a solution S, or no solution does exist*

```
BEGIN
  S:= emptyset;
  WHILE C # emptyset and not solution(S)
    DO
      x:=select(C); C:=C-{x};
      IF goodchoice(S \/ { x }) THEN S:=S \/ { x } FI;
    OD;
  IF solution(S) THEN return S ELSE return(no_solution} FI;
END
```

Mathematical definitions

We assume that a set E is given and is not empty; we use operators over binary relations over E. Those operators are defined as constants and have properties: rep, *opt*, $quotient$, DOMAIN, NOTDOMAIN, $lambda$, $greedy$.
The relation $quotient$ captures the idea of implication; in fact, it satisfies the property: $(quotient(S,T);T) \subseteq S$ which explains the choice of the name quotient. The domains of $(quotient(S,T)$ and S have the same carrier set. The range of $(quotient(S,T)$ and the domain of T have the same carrier set. The range of T and the range of S have the same carrier set.
If S is a relation between X and Y and T is a relation between Z and Y, then $quotient(S,T)$ is a relation between X and Z. The set operator does not exist in B but we can easily define it by quantification over domain and range. Hopefully, S. Curtis uses only two kinds of $quotient$: either with $X = Y = Z = E$, or $X = Y = E$ and $Z = \mathbb{P}(E)$. Two functions $quotient$ and $quotientP$ are defined and the formal definition in the B set theory is:

$$\begin{array}{l}
quotient \in (E \leftrightarrow E) \ \times \ (E \leftrightarrow E) \longrightarrow (E \leftrightarrow E) \\
\forall (R,S,T).(R \in E \leftrightarrow E \ \wedge S \in E \leftrightarrow E \ \wedge T \in E \leftrightarrow E \ \wedge R \subseteq quotient(S,T) \ \implies \ (T;R) \subseteq S) \\
\forall (R,S,T). \quad (R \in E \leftrightarrow E \ \wedge S \in E \leftrightarrow E \ \wedge T \in E \leftrightarrow E (T;R) \subseteq S \implies R \subseteq quotient(S,T))
\end{array}$$

The next lemma is useful, when using the relation $quotient$ and it has been proved by the proof tool.

Lemma 1 $x \mapsto y \in quotient(S,T) \Leftrightarrow \forall z \cdot (z \mapsto x \in T \implies z \mapsto y \in S)$

The equivalence is split into two implications. In the first implication ($\Rightarrow$), the second property of $quotient$'s definition is enough. For the second implication ($\Leftarrow$), we have instantiated R (the second property of $quotient$'s definition) with the following set: $\{x \mapsto y \mid x \in E \wedge y \in E \wedge \forall z \cdot (z \mapsto x \in T \implies z \mapsto y \in S)\}$

DOMAIN is the set of pairs (e,e) where e is in the domain of the binary relation used as parameter and NOTDOMAIN is the set of pairs (e,e) where e is not in the domain of the binary relation used as parameter. The B definitions are:

$$\begin{array}{l}
\text{DOMAIN} \in (E \leftrightarrow E) \longrightarrow (E \leftrightarrow E) \\
\forall R.(R \in E \leftrightarrow E \\
\qquad \implies \text{DOMAIN}(R) = id(\mathsf{dom}(R))) \\
\text{NOTDOMAIN} : (E \leftrightarrow E) \leftrightarrow (E \leftrightarrow E) \\
\forall R.(R \in E \leftrightarrow E \\
\qquad \implies \text{NOTDOMAIN}(R) = id(E) - id(\mathsf{dom}(R)))
\end{array}$$

opt assigns to each binary relation R over E a binary relation modelling the criterion of optimality. S. Curtis defines *opt* as follow: *opt*$(R) = \in \cap quotientP(R, \ni)$. The $\in$ operator is not defined in B and first, we define it as a relation In.

$$\begin{array}{l}
In \in \mathcal{P}(E) \leftrightarrow E \\
\forall (x,s).(x \in E \wedge s \in \mathbb{P}(E) \\
\qquad \Rightarrow (s \mapsto x \in In \Leftrightarrow x \in s)) \\
opt \in (E \leftrightarrow E) \longrightarrow (\mathcal{P}(E) \leftrightarrow E) \\
\forall R.(R \in E \leftrightarrow E \\
\qquad \Rightarrow \ opt(R) = In \ \cap \ quotientP(R, In^{-1}))
\end{array}$$

The next operator is called $lambda$ and it returns the image of each element in the domain of the relation; it is simply defined as the image of a singleton $\{x\}$ by the given relation:

$$\begin{array}{l} lambda \in (E \leftrightarrow E) \longrightarrow (E \longrightarrow \mathcal{P}(E)) \\ \forall(R,x).\ (R \in E \leftrightarrow E\ \wedge\ x \in \mathsf{dom}(R) \\ \qquad \Rightarrow\ lambda(R)(x) = R[\{x\}]) \end{array}$$

The greedy method is a method for computing optimal solutions in optimizations problems. The definition of the criterion for the local optimality should be stated. We assume that L defines the criterion for the local optimality and S defines the next possible step *opt* for the criterion of optimality. $greedy(L,S)$ is an operator defining pairs (x,y) built as descendants for S and optimal with respect to L.

$$\begin{array}{l} greedy \in (E \leftrightarrow E) \times (E \leftrightarrow E) \longrightarrow (E \leftrightarrow E \\ \forall(L,S).(L \in E \leftrightarrow E\ \wedge\ S \in E \leftrightarrow E \\ \qquad \Rightarrow\ greedy(L,S) = (lambda(S); opt(\end{array}$$

The operator $greedy$ satisfies a property derived with the proof tool.

Lemma 2 $\forall(L,S).(L \in E \leftrightarrow E\ \wedge\ S \in E \leftrightarrow E\ \Rightarrow\ greedy(L,S) = S \cap quotient(L, S^{-1}))$

The proof is discharged with the help of the prover; S. Curtis [36] writes that it is an useful property but she does not give any proof sketch. According to the definitions of $greedy(L,S)$ and $opt(L)$, we prove that $(lambda(S); In\ \cap\ quotientP(L, In^{-1}))\ =\ S \cap quotient(L, S^{-1})$ which is rewritten into $((lambda(S); In)\ \cap\ (lambda(S); quotientP(L, In^{-1})))\ =\ S \cap quotient(L, S^{-1})$. The proof is split into two cases:
1. $((lambda(S); In)\ =\ S$ which is easy to prove and
2. $(lambda(S); quotientP(L, In^{-1}))\ =\ quotient(L, S^{-1})$ which is split in two inclusions
2.1. $(lambda(S); quotientP(L, In^{-1}))\ \subseteq\ quotient(L, S^{-1})$ which is easy to prove with $quotient$ and $quotientP$ definitions and
2.2. $quotient(L, S^{-1})\ \subseteq\ lambda(S); quotientP(L, In^{-1}))$
For this last one we have used the first lemma: in one way ($\Rightarrow$) for $quotient(L, S^{-1})$ and the other one ($\Leftarrow$) for $quotientP(L, In^{-1})$.

This point shows clearly that the derivation of proofs of significant theorems is possible, if there is a mathematical expertise of the mathematical topics. The next operator is approaching an algorithmic idea of a process which is searching a value by repeating a step over a set as long as nothing is found. rep captures the idea of repeating a relation on a set as long as it is possible to apply the relation and the result of the application is simply a fixed-point. It behaves like a repeat-until loop and it may be operationally defined as follows: a pair (x,y) is in $rep(R)$, where R is a binary relation over E, if either $x \notin \mathsf{dom}(R)$ and $x = y$, or $x \in \mathsf{dom}(R)$ and there is a path over R leading to $y \notin \mathsf{dom}(R)$. Formally, rep is defined as follows:

$$\begin{array}{l} rep \in (E \leftrightarrow E) \longrightarrow (E \leftrightarrow E) \\ \forall R.(R\ \in\ E \leftrightarrow E \\ \qquad \Rightarrow\ rep(R)\ =\ \text{NOTDOMAIN}(R) \cup (R; rep(R))) \\ \forall(S,R).\quad (\ R \in E \leftrightarrow E \wedge S \in E \leftrightarrow E \wedge \\ \qquad\qquad \text{NOTDOMAIN}(R) \cup (R;S) \subseteq S \\ \qquad \Rightarrow\ rep(R) \subseteq S) \end{array}$$

$repn$ (standing for n steps) computes the binary relation obtained by composing a given binary relation with respect to a given natural number.

$$\begin{array}{l} repn\ \in\ (E\ \leftrightarrow\ E)\ \times\ \mathbb{N}\ \nrightarrow\ (E\ \leftrightarrow\ E) \\ \forall S.(S \in E \leftrightarrow E\ \Rightarrow\ repn(S,0) = \mathsf{id}(E)) \\ \forall(S,n).(n \in \mathbb{N}^{\star}\ \wedge\ S \in E \leftrightarrow E \\ \qquad \Rightarrow\ repn(S,n) = (repn(S,n-1); S)) \end{array}$$

The proof of theorem 3 (due to S. Curtis) requires intermediate lemmas on rep and $repn$. The first lemma is proved using the definition of rep and its minimality as a fixed-point (case $\subseteq$) and by induction (case $\supseteq$). The second lemma is a consequence of the first lemma and the two last lemmas are proved by induction.

Lemma 3 *1.* $\forall S.(S \in E \leftrightarrow E\ \Rightarrow\ rep(S) = \text{UNION}(n) \cdot (n \in \mathbb{N} | (repn(S,n);\ \text{NOTDOMAIN}(S))))$

2. $\forall(S,x,y).(S \in E \leftrightarrow E\ \wedge\ x \in E\ \wedge\ y \in E\ \wedge\ \ x \mapsto y\ \in\ rep(S)$
$\Rightarrow\ \exists n.(n \in \mathbb{N}\ \wedge\ x \mapsto y \in repn(S,n) \wedge y \mapsto y \in \text{NOTDOMAIN}(S)))$

3. $\forall(S,n).(S \in E \leftrightarrow E\ \wedge\ n \in \mathbb{N}\ \Rightarrow\ (S; repn(S,n) = (repn(S,n); S))$

4. $\forall(S,n,m).(S \in E \leftrightarrow E \ \wedge \ n \in \mathbb{N} \ \wedge \ m \in \mathbb{N} \ \wedge \ m \leq n$
   $quad \quad \Rightarrow \ repn(S,n) = (repn(S,m); repn(S,n-m))$

The definitions provide a general framework for expressing optimization problems related to the greedy method; Curtis [36] defines four classes of greedy problems by characterizing conditions over theories. She specializes the general theory and we translate her characterizations in the B set theory. Using these characterizations, properties ensuring the optimality of the solution are proved.

Properties derived from the mathematical structures

Following Curtis [36], we prove four possible cases on mathematical structures which are ensuring the optimality of the solution for the greedy method. The four cases are related by implicative properties; the stronger case is called the better-local case and the weaker (or the most general) is the best-global)(Semantical diagram of Curtis's classification [36]). It is a simple rewriting of Curtis's theorem but they are mechanically checked by our proof engine and they confirm the results of Curtis.
The terminology for greedy algorithms mentions a construction step, a local optimality criterion and a global optimality criterion. Following the terminology, we consider four cases corresponding to assumptions made on the mathematical structures and the current problem. Following Curtis [36], every greedy algorithm that finds optimal solutions to optimisation problems complies with this principle. We do not discuss this aspect and use it like a postulate. Let assume that L and C are two preorders over a set E and S is a binary relation over E. $greedy$ defines the greedy flavour of the method by combining both criteria over L for local and C for global. Curtis's theorems have the following schema:

$$\boxed{\begin{pmatrix} preorder(L) \wedge \ preorder(C) \wedge \\ S \in E \leftrightarrow E \ \wedge H(L,C,S) \end{pmatrix} \Rightarrow \ rep(greedy(L,S)) \subseteq \ (lambda(rep(S)); opt(C)))}$$

On the theorem 1 (Best-Global), $H(L,C,S)$ equals to

$$\begin{pmatrix} \text{DOMAIN}(greedy(L,S)) = \text{DOMAIN}(S) \ \wedge \\ ((rep(S))^{-1}; greedy(L,S)) \subseteq (C; rep(S)^{-1}) \end{pmatrix}$$

Our proof is quiet similar to Curtis's one.

On the theorem 2 (Better-Global), $H(L,C,S)$ equals to

$$\begin{pmatrix} \text{DOMAIN}(greedy(L,S)) = \text{DOMAIN}(S) \ \wedge \\ (S^{-1}; L; \text{DOMAIN}(S)) \subseteq (L; S^{-1}) \ \wedge \\ (S^{-1}; L; \text{NOTDOMAIN}(S)) \subseteq L \ \wedge \\ (\text{NOTDOMAIN}(S); L) \subseteq (C; rep(S)^{-1}) \end{pmatrix}$$

Our proof is similar to Curtis's one; we prove that assumptions of the theorem 2 imply assumptions of theorem 1.

On the theorem 3 (Best-Local), $H(L,C,S)$ equals to

$$\begin{pmatrix} \text{DOMAIN}(greedy(C,S)) = \ \text{DOMAIN}(S) \ \wedge \\ \forall n.(n \in \mathbb{N} \ \Rightarrow \\ \qquad repn(greedy(L,S),n) \ \subseteq \ (lambda(repn(S,n)); opt(L))) \ \wedge \\ (S^{-1}; L; \text{NOTDOMAIN}(S)) \ \subseteq \ L \ \wedge \\ (\text{NOTDOMAIN}(S); L) \ \subseteq \ (C; (rep(S))^{-1}) \end{pmatrix}$$

Our proof is quiet similar to Curtis's one. We prove that assumptions of theorem 3 imply assumptions of theorem 1.

During the mechanical proof, we discover missing obvious cases and one obvious hypothesis.
On the theorem 4 (Best-Local),$H(L,C,S)$ equals to

$$\begin{pmatrix} \text{DOMAIN}(greedy(L,S)) = \text{DOMAIN}(S) \ \wedge \\ (S^{-1}; L; \text{DOMAIN}(S)) \subseteq (L; S^{-1}) \ \wedge \\ (S^{-1}; L; \text{NOTDOMAIN}(S)) \subseteq L \ \wedge \\ (\text{NOTDOMAIN}(S); L) \subseteq (C; rep(S)^{-1}) \end{pmatrix}$$

Following Curtis's classification, we have proved the theorem in two manners: we prove that assumptions of theorem 4 imply assumptions of theorem 2 and that assumption of theorem 4 imply hypothesis of theorem 3. Our proofs are similar to Curtis's ones.
The number of interactions (clicks for choosing a function of the proof tool, choice of rules, . . .) during the proof process for lemmas and theorems is presented 742 interactions with the prover.

Greedy theories

The four theories are related according to the diagram expressing the power of a theory with respect to another one and it tells us that the best global theory is the most general one. However, the three other ones provide a way to classify the greedy algorithms and they can be simpler to derive a proved step-by-step developed greedy algorithm. Each theory must ensure the existence of an optimal solution in the set of possible solutions and it is the key property of the proof of optimality of this algorithm. We will develop the greedy algorithm from the most general theory namely the best global theory. Our intuition is that the theorem 1 is a refinement proof and that the other theorems are instantiation proof and the corresponding development can be obtain from our first development.

2.2.3 Abstract algorithmic models for the greedy method

The mathematical framework is clearly defined and we have four classes for greedy problems; we develop a sequence of refined general models for the greedy method. The development process has the following steps:

- First abstract greedy model $BG0$
- Greedy refinement model $BG1$ for the Best-Global principle
- Refinement model for getting an algorithmic expression

We assume that E and rep, opt, $quotient$, $quotientP$, DOMAIN, NOTDOMAIN, $lambda$ and rep are given; they satisfy the properties of the previous defined theory ($repn$ can be defined later). Now, we should define several new constants:

1. S is the step of the algorithm
2. C is the criterion for the global optimality and is a pre-order over E.
3. $greedy$ is the step of the greedy algorithm
4. $initial\_value$ is the initial value
5. $preorder(O) \triangleq O \in E \leftrightarrow E \ \wedge \ \mathsf{id}(E) \subseteq O \ \wedge \ (O;O) \subseteq O$

```
model
  $BG0$
sets
  $E$
constants
  $C, S, lambda, rep, opt, initial\_value$
properties
  $\ldots$
  $S \in E \leftrightarrow E$
  $preorder(C)$
  $initial\_value \in E$
variables
  $solution$
```

```
invariant
  $solution \in E$
initialisation
  $solution \ :\in \ E$
events
$compute \ \widehat{=}$ any $e$ where
    $e \in E$
    $initial\_value \mapsto e \in (lambda(rep(S)); opt(C))$
  then
    $solution := e$
  end
end
```

The first abstract model is not very surprising; it computes in one shot an optimal solution. We use a variable called $solution$. Two events are defined in the first abstract model. Initialisation is the initial event; it starts the execution of the abstract system by assigning any value to $solution$. $compute$ computes an optimal solution among the possible ones; the set of possible optimal solutions is not empty. The invariant of the system is simple: $solution \in E$; the assumptions on the constants allow us to derive the validity of the current model. There are no difficulties. This model states the initial problem and the next refinement will give some solution.

Now, we should choose one criterion and we choose to assume that the mathematical structure satisfies the Best-Global principle. We add the properties related to the principle and we introduce the function for modelling the repetition rep.

$$\boxed{\text{DOMAIN}(greedy(L,S)) = \text{DOMAIN}(S) \ \wedge ((rep(S))^{-1}; greedy(L,S)) \ \subseteq \ (C; rep(S)^{-1})}$$

The next property is derived from the assumptions over the current model.

Property 2 *Under the conditions defined by the greedy theory and the best global ones, the set* $(lambda(rep(S)); opt(C))$ *is not empty.*

The current model $BG0$ is refined by modifying the event $compute$, since the theorem 1 states that $rep(greedy(L,S)) \subseteq (lambda(rep(S)); opt(C))$. $compute$ computes an optimal solution among the possible ones; the set of possible optimal solutions is not empty.

```
refinement
  $BG1$
refines
  $BG0$
constants
  $L$
properties
  $preorder(L)$
  $\text{DOMAIN}(greedy(L,S)) = \text{DOMAIN}(S)$
  $((rep(S))^{-1}; greedy(L,S)) \subseteq (C; rep(S)^{-1})$
variables
  $solution$
```

```
invariant
  $solution \in E$
initialisation
  $solution :\in E$
events
$compute \ \widehat{=}$ any $e$ where
    $e \in E$
    $initial\_value \mapsto e \in rep(greedy(L,S))$
  then
    $solution := e$
  end
end
```

The new refinement $BG2$ introduces the effective computation step and the new invariant is much more elaborate. It includes the notion of existence of a solution while iterating. A new variable is introduced to contain the current value and it is called $current$. The new invariant states that the current value is an execution value leading to an optimal solution in finite time:
$solution \in E \ \wedge current \in E \ \wedge$
$\exists n.(n \in \mathbb{N} \ \wedge \ initial\_value \mapsto current \in \ repn(greedy(L,S),n))$

The initial event and the event $compute$ are refined and a new event is introduced to model the step of the iteration. The final event is triggered, when the guard is true.

```
$compute \ \widehat{=}$
  when
    $current \mapsto current \ \in \ \text{NOTDOMAIN}(greedy(L,S))$
  then
    $solution := current$
  end
```

The event $step$ models the step of each computation while looping. We can replace BG1 properties (assumptions of theorem 1) by assumptions of theorem 2 (resp theorem 3 or theorem 4) to obtain a refinement proof, which is similar to the proof of theorem 2 (resp. theorem 3 or theorem 4). However, these developments seem to be independent, because the classification (of Curtis) is hidden inside the proof.

```
$step \ \widehat{=}$
  any $e$ where
    $e \in E$
    $current \mapsto e \in greedy(L,S)$
  then
    $current := e$
  end
end
```

In this section, we explain how we can obtain other algorithms Best-Local, Better-Global and Better-Local by an instantiation of our previous model Best-Global. We give only an example. We omit to write the refinement model $BG3$ and we obtain a generic algorithm 2 from the refinement model $BG3$ by combining events. To obtain a complete development (similar to the previous one), we can instantiate name by name the previous development. Instantiation proof of refinement are obvious: both abstract models compute the same result. The other instantiation of proof obligation is that the properties of the Better-Global imply properties of the Best-Global system. This proof obligation is exactly the second theorem. We summarize the final statement of proofs discharged through the different refinement step.

2.2.4 Conclusion

The incremental proof-based development of greedy algorithms is illustrated from the theoretical characterization of S. Curtis [36] and we state and check properties over mathematical structures related to the greedy

Algorithm 2: General Greedy Algorithm

Pre-condition:*C is a set of possible candidates*
Post-condition:*Either a solution S, or no solution does exist*

```
BEGIN
  solution::E;
  current:=initial_value;
  WHILE current in domain(S)
    DO
      current:= ChooseOneIn(opt(L)(S[{ current}]))
     OD
  solution:=current
END
```

method. The main advantage is to obtain a complete checking of Curtis's results, since a proof tool is not accepting results as *left to the reader* and every proof step should be completely discharged. The exercise illustrates the use of generic developments based on very general mathematical structures; the genericity of proof-based development is a way to improve the proof process. This point should be developed by replaying the development of Prim's algorithm already developed using the refinement [7]; we plan to develop Kruskal's algorithm and other greedy problems. The fundamental question is also to be able to state what is the problem to solve and any development should start by the statement of mathematical structures and by the proof of properties required for the given problem (existence of solutions, for instance). Future work will develop new instantiations for the greedy-oriented developments.

2.3 Application 2: Electronic voting

The storage of votes is a critical component of any voting system. In traditional systems there is a high level of transparency in the mechanisms used to store votes, and thus a reasonable degree of trustworthiness in the security of the votes in storage. This degree of transparency is much more difficult to attain in electronic voting systems, and so the specific mechanisms put in place to ensure the security of stored votes require much stronger verification in order for them to be trusted by the public. There are many desirable properties that one could reasonably expect a vote store to exhibit. From the point of view of security, we argue that *tamper-evident* storage is one of the most important requirements: the changing, or deletion of already validated and stored votes should be detectable; as should the addition of unauthorised votes after the election is concluded. We propose the application of formal methods for guaranteeing, through construction, the correctness of a vote store with respect to the requirement for *tamper-evident* storage. We illustrate the utility of our refinement-based approach by verifying — through the application of a reusable formal design pattern — a store design that uses a specific PROM technology and applies a specific encoding mechanism.

2.3.1 Introduction

Motivation: the e-voting problem

Computer technology has the potential to modernise the voting process and to improve upon existing systems; but it also introduces new concerns with respect to secrecy, accuracy, trust and security[46]. The debate over e-voting is not a new one — recent use of such systems in actual elections has led to their analysis from a number of different viewpoints: usability[47], trustworthiness and safety criticality[57], transparency and openness[58], and risks and threats[65].
The potential advantages are generally accepted, for example: faster result tabulation, elimination of human error which occurs in manual vote tabulation, assistance to voters with "special" needs, defence against fraudulent practices (e.g. with postal votes[18]), and improving the "fairness" of count systems that incorporate "unfair" non-deterministic procedures[75].

Despite ever-increasing uncertainty over the trustworthiness of these systems — which is one of the major disadvantage associated with them — many countries (particularly in Europe[74]) have recently chosen to adopt e-voting. The main risks that have been clearly identified seem not to concern those responsible for procuring the systems. In fact, it appears that e-voting is just one, well-publicised, example of governments wishing to adopt new technologies[71] before the risks and benefits, as perceived by the public[49], have been properly analysed and debated.

This study contributes to answering two important questions: firstly, whether the public's mistrust in the security of e-voting systems is well-founded; and secondly, whether formal methods have a role to play in addressing the problem of mistrust. With respect to the first aspect, Kocher and Schneier[51] state: "The threats are real, making openness and verifiability critical to election security." As to the second aspect, expecting the public to trust the adoption of such new technology is only possible if they can be convinced that it rests on firm trustworthy foundations. The formal methods community have an important role to play in this respect: the use of formal notations provides a fundamental foundation upon which the complexities of ever-changing technologies can be managed[45]. We argue that without the adoption and promotion of formal methods as the foundations of software engineering, developing trustworthy e-voting systems will not necessarily guarantee that they will be trusted.

This study proposes that, in general, already existing formal techniques can help to alleviate many of the verification problems that the adoption of new e-voting technologies can introduce. For the specific modelling and verification in our study we use the event-B method[4], based on the B notation. We argue that it is unreasonable to expect the public to trust a system (or part of a system) to behave correctly just because it is developed using a formal method (like event-B)[3]. Instead, we propose that we must first establish a set of quality standards for reliable, re-usable, trustworthy tools and techniques that have proven themselves in the formal development of correct systems. Then, provided the public are properly informed, it is not unreasonable to expect systems built in this way to be both more trustworthy *and* more trusted. The correct-by-construction approach in this paper illustrates the type of standard process to which we are alluding.

Formal methods and vote storage

Public opinion, arising from detailed debate of the issues, would suggest that for e-voting machines to be acceptable they should be developed following best practice with regards to the engineering of critical systems. Media reports would also suggest that the secure storage of votes is one of the issues that is most mistrusted by the public.

We propose the use of formal methods as a means of ensuring that a machine securely stores votes, and we propose to demonstrate the utility of formal methods through guaranteeing simple safety properties of a voting machine store. The main property that we examine is concerned with the need for *tamper-evident* storage, which addresses the risk of unauthorised tampering of vote data after it has been correctly registered and stored. In *Analysis of an electronic voting system*[52], we see that such a security weakness already exists in one of the most widely procured voting systems:

> "... an adversary could alter election results by modifying ballot definition files, and ... it leaves no evidence that an attack was ever mounted"

Here, the "adversary" is most likely to be a single insider (election official) with access to the storage device. We argue that it is the responsibility of the storage designers to guarantee the security of the votes stored without having to make an assumption about the behaviour or intent of such officials.

In order to illustrate how a guarantee could be made, we use event-B and apply an incremental refinement approach to verifying a sequence of designs for the storage of votes, which we prove to be correct-through-construction with respect to the simple requirement that the vote storage is tamper-evident.

Refinement and genericity: a formal design pattern

¿From a technological viewpoint we know that system design has an important role in security assurance. Mercuri[60] addresses the theme of quality in the process of engineering security:

> "By encouraging artistry and applying craftsmanship to our security problems, viable solutions will emerge."

[3]This is analagous to the current common situation where the public have been asked to trust the e-voting machines because they have been independently tested by some appropriately accredited body. Experience now shows that such trust was misplaced.

This supports our view that one must start with a simple model of the vote store requirements and refine that model, during design, towards a correct implementation. For this reason we chose a simple security requirement — that only valid votes can be found in the vote store and that these cannot be tampered with without detection — and start our formal development from there. The use of formal methods to guarantee that only valid votes are passed from the machine interface to the store has already been presented[23]. The work presented in this study, which addresses the tamper-evident requirement, is complementary in nature: event-B is the common modelling language, and correctness through construction is the common formal design approach.

As one of the long-term goals of the formal methods community is to simplify the verification process for engineers[19, 50], we support the view that re-usable verification design patterns, similar in nature to the work by Mehlitz and Penix[59], as a potential solution to this problem. This study identifies a good candidate for such a re-usable pattern, combining genericity and refinement to provide a *correct-by-construction pattern* (see Section 4).

2.3.2 Manchester Encoding: formalising the design of a secure vote PROM

The main design that is modelled and verified in this study is taken directly from the work by Molnar, Kohno, Sastry and Wagner[61]. Their proposed solution to providing tamper-evident storage involves the application of Manchester codes[72] and a write-once data PROM store. The encoding simply represents a 0 as a 01 and a 1 as a 10. Thus, when validating votes stored as pairs of bits there are 2 additional pair cases to be considered, where (because our memory allows only 1s to be overwritten as 0s): 11 corresponds to unwritten memory and 00 corresponds to an invalid memory that has been tampered with.

Before we formally specify and verify the proposed solution, we briefly note that there is a real pragmatic need for *tamper-evident* rather than *tamper-proof* writeable storage. The *tamper-proof* requirement can be met only by some security mechanism ensuring authorised-only update of the vote store. This security mechanism would probably be implemented as some combination of physical constraints, together with hardware and software checks. It would most likely involve some complex encryption technique and it is not clear whether one could, or should, expect voters to trust such a complex system. Contrastingly, guaranteeing the *tamper-evident* requirement is a much simpler problem that — if done well — could be both trustworthy and trusted.

Implementing storage using a write-once data store has many obvious advantages when we consider tampering: obviously, any vote that has already been written cannot be overwritten? In fact, without a more formal model of the store, this is not guaranteed to be true. For example, one form of write-once storage could allow the flipping of an initial bit state to be done once and once only. This does not necessarily guarantee that a recorded vote cannot be overwritten as individual bits of a vote will not have been flipped when a vote is recorded. In fact, as with all storage mechanisms, the (encoding) protocol used for writing information to such a store will be the deciding factor in whether the tampering requirements are met. Furthermore, there are many reasonable variations of the tampering requirement. Without a precise statement, it is not clear whether we will be able to verify whether a given system (the store properties, together with the encoding protocol) is correct.

The key property of the encoding that we shall model is that if any (sub)set of 1 bits in a stored codeword are flipped to 0s then the result is no longer a valid code word. We then wish to establish that anyone with read access to the voting store can detect an invalid memory state, where at least one codeword is invalid, and consequently any tampering after[4] the election has been completed. The verification of this safety property requires modelling of the write-once behaviour in the chosen PROM implementation (checking that 1s can be re-written as 0s but that 0s cannot be changed) in conjunction with the encoding mechanism. It also requires the use of a special *election over* bit (bit pair in PROM) to signify that the election is over, and which must be unset and untampered with for new votes to be recorded (otherwise anyone with access to the voting machine could add unauthorised votes after the election, an attack known as ballot stuffing). We chose not to include the *election-over* behaviour in the model presented in this study.

We note that this system is not tamper proof: attackers with write access to the vote store can still invalidate the election by overwriting vote data. However, this attacks would be easily identified by procedures for validating the storage state during and after the vote.

The main advantages of doing this design formally, in event-B, are development oriented:

[4]It is trivial to extend our model to dynamically detect tampering during an election but for simplicity and conciseness we do not present details of this variation of tamper-evidence in this study.

- an abstract model can be easily validated as correctly expressing the requirement,
- the actual design model can be constructed incrementally through refinement of the abstraction,
- the refinement process can continue through to modelling at very fine grain levels of detail that correspond to the chosen low-level implementation architecture,
- we can more easily reason about different variations and combinations of encodings and storage media, and
- we can analyse possible problems of integrating this requirement with other requirements of the e-voting system, in general, and the vote storage, in particular.

Thus, we are more likely to develop a trustworthy storage.
A secondary benefit arises when we consider the issue of how to build public trust in our formally developed trustworthy system. We argue that the *correct-by-construction* technique, embodied in a reusable design pattern, will become more and more trusted as it is used to develop more and more systems that prove themselves to be trustworthy. As a consequence, using such a standard technique (and associated tools) in constructing critical systems will increase confidence in the systems' correctness, from both the developers and the public users.
With tool support for automatically checking our verification proof we have another advantage: if our proof tool is trustworthy then the design is sure to be correct provided the property that we have established, in the initial abstract model, is an accurate statement of the high level requirement. To make this transparent to the users (voters) it is essential that an initial abstract model is easy to understand and validate, and that they have good reason not to mistrust our proof tool and techniques. Our design approach facilitates this type of openness and transparency.

The generic problem model $m1$

The refinement in model $m1$ introduces an abstract mechanism for encoding votes. Constants G_C (for GoodCode) and B_C (for BadCode) are two subsets of the set $CODE$. These sets are disjoint but don't neccessary cover the set $CODE$. The constant $code$ is a bijection between $VOTES$ and G_C.
The last constant chg is a relation between $CODE$s. The most important property of the relation chg is that a good or bad code can only be changed to a bad one. In B we specify these as $PROPERTIES$ of the model.

$$
\begin{array}{l}
G\_C \subseteq CODE \\
B\_C \subseteq CODE \\
G\_C \cap B\_C = \varnothing \\
code \in VOTE \rightarrowtail\!\!\!\!\twoheadrightarrow G\_C \\
chg \in CODE \leftrightarrow CODE \\
chg[G\_C \cup B\_C] \subseteq B\_C
\end{array}
$$

We refine the $corrupt$ event to ensure that any encoded vote that has been changed can be recognised as being bad. Furthermore, we refine the $skip$ event to say that if we now allow encoded votes to be changed then a bad vote is guaranteed to stay bad.

```
corrupt ≙
  any $v, c, b$ where
    $v \in vt$
    $c \in G\_C$
    $b \in CODE$
    $v \mapsto c \in Cv$
    $c \mapsto b \in chg$
  then
    $Cv(v) := b$
  end
```

```
corrupt_again ≙
  any $v, c, b$ where
    $v \in vt$
    $c \in B\_C$
    $b \in CODE$
    $v \mapsto c \in Cv$
    $c \mapsto b \in chg$
  then
    $Cv(v) := b$
  end
```

$m1$ **refines** $m0$

There is some additional work in proving that $m1$ refines $m0$ as we need to "glue together" the abstract and concrete models using a gluing invariant.

$$\boxed{\begin{array}{l} Cv \in vt \rightarrow G\_C \cup B\_C \\ good = \mathsf{dom}(Cv \rhd G\_C) \\ bad = \mathsf{dom}(Cv \rhd B\_C) \end{array}}$$

To glue together the actual set of votes (vt) with the $CODE$ we introduce Cv. Then, the abstract variables $good$ and bad can be defined in the concrete model using Cv.
We note that $m1$ generated 11 proof obligations and all but one were automatically discharged, with the single remaining obligation easily discharged through interaction with the theorem prover.

2.3.3 Project $\mathbb{H}$

In project $\mathbb{G}$ we have abstracted away from how a vote is represented. In project $\mathbb{H}$ we work with concrete representations of the votes (within the generic structure of project $\mathbb{G}$) by instantiating parameters of the models in $\mathbb{G}$.

The problem to be solved — MCH

In our pattern, the new problem to be solved is expressed by the model called MCH; which contains all specific constants, and with identical variables and identical event names as can be found in $m0$. $M0$ (resp. $M1$) is the model $m0$ (resp. $m1$) instantiated using the new constants of MCH and our new project is $\mathbb{MCH}$. MCH is the basis for our iterative refinement development process. The main point of interest is that the refinement between the model MCH and the model $M0$, an instance of $m0$, allows us to guarantee that our specific problem MCH is solved or refined by our intantiated model $M0$. For convenience (and space) we present all specific constants in two steps when presenting $M0$ and $M1$.

$M0$ **refines** MCH **and instantiates** $m0$

We start the development with an abstract model where a vote is represented by k bits. In fact, the abstract $VOTE$ set in model $m0$ is replaced (instantiated) by our more concrete $VOTES$:

$$\boxed{\begin{array}{l} k \in \mathbb{N} \\ VOTES = 1..k \rightarrow 0..1 \end{array}}$$

We have nothing to prove to verify that this instantiation is correct (there are no additional properties on abstract set $VOTE$). The proof obligation that $M0$ refines MCH is obvious (with the same abstract and concrete events) and done automatically.

$M1$ **— an intermediate design step**

For this step we enrich the representation of a vote by doubling the number of bits. Each bit in the original vote representation is paired with its inverse value. For example, a vote that was represented as 10010 will now be represented by $(10010, 01101)$

$$\boxed{\begin{array}{l} inv \in VOTES \rightarrow VOTES \\ \forall (v,i) \cdot \left(\begin{array}{l} v \in VOTES \ \ \wedge \ \ i \in 1..k \\ \Longrightarrow \\ inv(v)(i) = 1 - v(i) \end{array} \right) \\ CODE = VOTES \times VOTES \\ code \in VOTES \rightarrow CODE \\ \forall v \cdot (v \in VOTES \ \ \Longrightarrow \ \ code(v) = v \mapsto inv(v)) \end{array}}$$

For the new model, we instantiate the constants of the generic model $m0$: GC instantiates G_C and BC instantiates B_C and $chgv \cup chgi$ instantiates chg.
GC and BC are specified as follows:

$$\boxed{\begin{array}{l} GC = \{v \mapsto w | v \mapsto w \in CODE \ \wedge \ w = inv(v)\} \\ BC = \{c | c \in CODE \ \wedge \ PC(c)\} - GC \end{array}}$$

Note: in the definition of BC we use a predicate over codes, PC, that is defined to identify all "possible" codes. This predicate is true except in the case where a pair of bits in an encoded vote are both 1. This encoding is not possible because we allow only bits to change from 1 to 0 (and not from 0 to 1) and because all votes are initially coded as good codes (with pairs 01 or 10). PC is defined as follows:

$$\boxed{PC(c) = \exists (v, w) \cdot \left(\begin{array}{l} c = v \mapsto w \ \wedge \\ \forall i \cdot \left(\begin{array}{l} i \in 1..k \ \wedge \\ v(i) = 1 \\ \Longrightarrow \\ w(i) = 0 \end{array} \right) \end{array} \right)}$$

Now we wish to specify that when a change is made to a single bit of a vote's representation (in either of the pair elements) then only a bit 1 can change to the bit 0. We do this by defining $chgv$ to specify how the vote part of the pair can change, and $chgi$ to specify how (symmetrically) the inverse part of the pair can change. The specification of $chgi$ is given below:

$$\left(\begin{array}{l} (v1 \mapsto w1) \mapsto (v2 \mapsto w2) \in chgi \\ \Leftrightarrow \\ v1 = v2 \ \wedge \\ \exists i \cdot \left(\begin{array}{l} i \in 1..k \ \wedge \\ w1(i) = 1 \ \wedge \ w2(i) = 0 \ \wedge \\ \forall j \cdot \left(\begin{array}{l} j \in 1..k \ \wedge \ i \neq j \\ \Longrightarrow \\ w1(j) = w2(j) \end{array} \right) \end{array} \right) \end{array} \right)$$

The specification of $chgv$ is symmetrically defined.

2.3.4 $M1$ instantiates $m1$

We have instantiated[5] our previous generic model replacing: G_C with GC, B_C with BC and chg with $chgv \cup chgi$. Then we need to prove the instantiation to be correct by establishing the following proof obligation (from the invariant of model $m1$).

$$\boxed{\begin{array}{l} GC \subseteq CODE \\ BC \subseteq CODE \\ GC \cap BC = \varnothing \\ code \in VOTE \rightarrowtail\!\!\!\!\twoheadrightarrow GC \\ chgv \cup chgi \in CODE \leftrightarrow CODE \\ (chgv \cup chgi)[GC \cup BC] \subseteq BC \end{array}}$$

For convenience we structure the proof based on the two symmetric cases, depending on whether a change is made using $chgi$ or $chgv$. To do this, we split both events corrupt and corrupt_again into corruptv and corruptv_again, and corrupti and corrupti_again.

For brevity, we give the definition of only one half of the symmetric pair, the $chgv$ case:

```
corruptv  ≙
  any $v, c, b$ where
    $v \in vt$
    $c \in G\_C$
    $b \in CODE$
    $v \mapsto c \in Cv$
    $c \mapsto b \in chgv$
  then
    $Cv(v) := b$
  end
```

```
corruptv_again  ≙
  any $v, c, b$ where
    $v \in vt$
    $c \in B\_C$
    $b \in CODE$
    $v \mapsto c \in Cv$
    $c \mapsto b \in chgv$
  then
    $Cv(v) := b$
  end
```

We note that for this step we had 7 proof obligations (for the instantiation), 3 of which required interactive proofs.

[5]In our design pattern, $M1$ is shown as simply an instantiation of $m1$. In fact, in this study, $M1$ is constructed as a refinement of the instantiation. For the sake of brevity, we have combined a horizontal instantiation with a vertical refinement in a single development step.

2.3.5 A final implementation model — $M2$

A manchester code is a sequences of $2 \times k$ bits where the oddly ranked bits give the representation of the original k bits of an unencoded vote, and the even rank gives the bit-wise inverse. This is defined by $MNCH$, below:

$$\begin{array}{l}
MNCH = 1..2 \times k \rightarrow 0..1 \\
mcode \in VOTES \rightarrow MNCH \\
\forall (v,i) \cdot \left(\begin{array}{l} v \in VOTES \ \wedge \ \ i \in 1..k \\ \Longrightarrow \\ mcode(v)(2 \times i - 1) = v(i) \ \ \wedge \\ mcode(v)(2 \times i) = 1 - v(i) \end{array} \right)
\end{array}$$

We also define two constants, mv and mi, to extract a vote and its inverse from the manchester encoded ($2 \times k$) bits:

$$\begin{array}{l}
mv \in MNCH \rightarrow VOTES \\
mi \in MNCH \rightarrow VOTES \\
\forall (c,i) \cdot \left(\begin{array}{l} c \in MNCH \ \wedge \ \ i \in 1..k \\ \Longrightarrow \\ mv(c)(i) = c(2 \times i) - 1 \ \ \wedge \\ mi(c)(i) = c(2 \times i)) \end{array} \right)
\end{array}$$

$M2$ refines $M1$

It should be obvious that the Manchester code is a correct implementation of our requirements since it is clearly a correct implementation of $M1$. Intuitively, $M2$ refines $M1$ by changing the way in which the votes are encoded. In $M1$ they are encoded as a pair of bitsequences; in $M2$ they are single bit sequences where the original pair values have been interleaved. For example, the vote 101 is encoded as $(101, 010)$ in $M1$ but as 100110 in $M2$.
In order to formally proof this, we establish that the invariant in $M1$ is true in the refinement $M2$. In order to do this, we replace (instantiate) Cv with $Mchv$.

$$\begin{array}{l}
Mchv \in vt \rightarrow MCH \\
\forall (v,m) \cdot \left(\begin{array}{l} v \in vt \ \wedge \\ m \in MNCH \ \wedge \\ v \mapsto m \in Mchv \\ \Longrightarrow \\ v \mapsto (mv(m) \mapsto mi(m) \in Cv \end{array} \right)
\end{array}$$

Then we introduce two more constants:

```
corruptx  ≙
  any  $v, c, a$  where
    $v \in vt$
    $v \mapsto c \in Mchv$
    $GD$
    $a \in 1..2 \times k$
    $x(a)$
    $c(a) = 1$
  then
    $Mchv(v)(a) := 0$
  end
```

```
corruptx_again  ≙
  any  $v, c, a$  where
    $v \in vt$
    $v \mapsto c \in Mchv$
    $\neg GD$
    $a \in 1..2 \times k$
    $x(a)$
    $c(a) = 1$
  then
    $Mchv(v)(a) := 0$
  end
```

where:

$$GD = \forall i \cdot \left(\begin{array}{l} i \in 1..k \\ \Longrightarrow \\ c(2 \times i - 1) \neq c(2 \times i) \end{array} \right), \text{ and}$$

$v(a) = odd(a)$ and $i(a) = even(a)$.

Without going into details, we note that in this step there are 15 proof obligations, 5 of which required interactive proofs as they could not be discharged automatically by the tool.

2.3.6 Conclusions

We have argued that without the adoption and promotion of formal methods, as the foundations of software engineering, developing trustworthy e-voting systems will not necessarily guarantee that they will be trusted. We have demonstrated the application of the formal methods event-B for guaranteeing, through construction, the correctness of a vote store with respect to the requirement for *tamper-evident* storage. We illustrated the utility of our refinement-based approach

by verifying — through the application of a reusable formal design pattern — a store design that uses a specific PROM technology and applies a specific Manchester encoding mechanism. The formal design pattern is a reusable solution to a common design problem — of how genericity can help to structure the refinement proof process — that can be exploited by formal developers who are not necessarily expert.
Future work is mainly concerned with maintainability and extensibility, for example:

- **Strongly tamper-evident storage** — The design that we have presented in this study guarantees that the store is weakly tamper evident. It is said to be weak because tampering can be detected once an election is complete. In fact, with minor modifications to the design we can meet the requirement for a store that is strongly tamper evident: so that tampering can be detected during the voting process.
- **Election closed bit** — There is a separate requirement that no more votes can be recorded once an election is closed. Clearly, the implementation of this requirement will involve some extension to the storage of votes so that the store is protected against any further addition of votes after the voting process is terminated (known as vote stuffing). A proposed design is to add an election closed bit to the store and to check that this is not set as a guard for the writing of a vote to the store. Of course, with our encoding mechanism we can detect when this bit has been tampered with. However, without formal modelling it is difficult to reason about the consequences of such a design with respect to a potential denial of service attack where the bit is set before the election has really terminated.
- **History independent storage** – The requirement that the physical order of the votes recorded in the store cannot be used to deduce any information about the vote of a particular voter.

We will analyse the different structuring mechanisms in event-B and the ways in which they can be used to extend our storage requirements.

Chapter 3

The *call as event* pattern

Sommaire

Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a pattern and a methodology based on Event B and the refinement by developing Floyd's algorithm for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. The pattern provides a tool for developing many case studies of the classical algorithmics; moreover, a tool is under construction for applying the pattern.

3.1 Introduction

Overview. Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by developing algorithms for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. Floyd's algorithm is redeveloped and we add comments on the complexity of proofs and on the discovery of invariant; it should be considered as an illustration of a technique introduced in a joint paper with D. Cansell[28]. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at different levels of the university. It is also a way to introduce a pattern used for developing sequential structured programs.

Progamming methodology. The development of structured programs is carried out either using bottom-up techniques, or top-down techniques; we show how refinement and proof can be used to help in the top-down development of structured imperative programs. When a problem is stated, the incremental proof-based methodology of event B[25] starts by stating a very abstract model and further refinements lead to finer-grain event-based models which are used to derive an algorithm[5]. The main idea is to consider each *procedure call* as an *abstract event* of a model corresponding to the development of the *procedure*; generally, a procedure is specified by a pre/post specification and then the refinement process leads to a set of events, which are finally combined to obtain the *body of the procedure*. The refinement process can be considered as an *unfolding* of *calls* statements under preservation of invariants. It means that the abstraction corresponds to the procedure call and the body is derived using the refinement process. The refinement process may also use recursive procedures and supports the top-down refinement. The procedure call simulates the abstract event and the refinement guarantees the correctness of the resulting algorithm. A preliminary version[28] introduces ideas on a case study and provides an extended abstract of the current paper.

Proof-based Development. Proof-based development methods[11, 3, 62] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete events. The relationship between two successive models in this sequence is that of *refinement*[11, 3]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination. A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine[6]. At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations. The goal of a event B development is to obtain a *proved model* and to implement the correctness-by-construction[55] paradigm. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity.

3.2 The modelling framework

We do not recall oncepts of the Event B modelling language developed by J.-R. Abrial[4, 25]; we sketch the general methodology we are applying. The ingredients for describing the modelling process based on events and model refinement can be found in [4, 25]. We assume that the goal is to solve a given problem described by a semi-formal mathematical text and we assume that the problem is defined by a precondition and a postcondition[62]. The modelling process starts by identifying the domain of the problem and it is expressed using the concept of **CONTEXT**. A

CONTEXT PB

SETS

D

CONSTANTS

x, P, Q

AXIOMS

$axm1$: $x \in D$ $/*$ x belongs to a general set of the problem domain $*/$

$axm2$: $P \subseteq D$ $/*$ P is a set defining the precondition $*/$

$axm3$: $Q \subseteq D \times D$ $/*$ Q is a binary relation over S defining the postcondition $*/$

$axm4$: $x \in P$ $/*$ x is supposed to satisfy the precondition P $*/$

$axm5$: $\forall a \cdot a \in P \Rightarrow (\exists b \cdot a \mapsto b \in Q)$ $/*$ there is at least one solution for each data x satisfying the precondition P $*/$

END

Figure 3.1: Context for modelling the problem PB

Name	Syntax	Definition
Binary relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition of relations	$r_1; r_2$	$\{x, y \mid x \in a \wedge y \in b \wedge \exists z.(z \in c \wedge x, z \in r_1 \wedge z, y \in r_2)\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \wedge y \in \mathcal{P}(b) \wedge y, x \in r\}$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Range	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identity	$\mathsf{id}(s)$	$\{x, y \mid x \in s \wedge y \in s \wedge x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd s$	$r;\ \mathsf{id}(s)$
Anti-restriction	$s \mathbin{⩤} r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \mathbin{⩥} s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Overriding	$q \mathbin{⩤} r$	$(\mathsf{dom}(r) \mathbin{⩤} q) \cup r$
Partial Function	$s \rightarrowtail\!\!\!\!\!+ \ t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1}; r) \subseteq \mathsf{id}(t)\}$

Table 3.1: Set-theoretical notation for event B models

CONTEXT PB (see Figure 3.1) states the theoretical notions required to be able to express the problem statement in a formal way. The **CONTEXT** PB declares

- a domain D which is the global set of possible values of the current system.
- a list of constants x, which is specifying the input of the system under development, P, which is the set of values for x defining the precondition, and Q, which is a binary relation over D defining the postcondition of the problem.
- a list of axioms assigns types to constants and adds knowledges to the RODIN environment; for instance, the axiom 5 states that there is always a solution y, when the input value x satisfies the precondition P.

A **CONTEXT** may include a clause THEOREMS containing properties derivable in the theory defined by sets, contants and axioms; theorems are discharged using the proof assistant of the tool RODIN. The underlying language is a set-theoretical language partially given in Table 3.1. When an expression E is given, a well-definedness condition is generated by the tool; this point llows us to check that some side conditions are true. For instance, the expression $f(x)$ generates a condition as $x \in dom(f)$.
The first model provides the declaration of the procedure call. Variables y are *call-by-reference* parameters, constants x are *call-by-value* parameters and carrier sets s are used to type informations and also for defining a generic procedure:

```
MACHINE PREPOST

SEES PB

VARIABLES
        y

INVARIANTS
        inv1 : y ∈ D

EVENTS

INITIALISATION
        BEGIN
                act1 : y :∈ D
        END

EVENT call
        BEGIN
                act1 : y : |(x ∈ P ∧ x ↦ y′ ∈ Q)
        END

END
```

Figure 3.2: Machine defining the model for modelling the problem PB

procedure call(**x; var** y)
precondition $y = y_0 \wedge \ Init(y_0, x, D) \wedge \ \tilde{P}(x)$
postcondition $\tilde{Q}(x, y)$

Figure 3.2 describes the complete model for the problem PB; it is expressd by a generic procedure stating the pre/post-specification. The term *procedure* can be substituted by the term *method*. The current status of the development can be represented as follow:

$$\begin{tikzcd} \texttt{call(x,y)} \arrow[r, "call-as-event"] & \text{PREPOST} \arrow[r, "\text{SEES}"] & \text{PB} \end{tikzcd}$$

The statement of a given problem in the Event B modelling language is relatively direct, as long as we are able to express the mathematical underlying theory using the mechanism of contexts. The existence of a solution y for each value x is assumed to be an axiom; however, it would be better to derive the property as a theorem and it means that we should develop a way to validate axioms to ensure the consistency of the underlying theory.
The next section illustrates the technique used for developing new algorithms. We think that it is a good way to teach the design of algorithms. HOARE logic[48] provides a very interesting framework for dealing with specifications an development and our work shows how the ingredients of HOARE logic can be used to provide a general framework for developing sequential programs correct by construction. Event B and the RODIN plateform can be used to teach basic notions like pre and postconditions, invaraint, verification and finally design-by-contract.

Methodological note: *The challenge of the teacher is to relate the Event B notations to the notations of the programming language. We have used the Event B notations in lectures on fixed-point theory and on the explanation of sequential algorithms. It is then clear that we should provide more systematic rules for deriving algorithms. The management of definitions using a tool, like RODIN, helps students to understand why a function call like $f(x)$ generates conditions like $x \in dom(f)$. Nobody can cheat with the tool. Moreover, when a tool is available for a free download, it is really a teachermate.*

The illustration of the methodology is given by developing algorithms for solving the shortstes path problem using a dynamic programming paradigm. We will summarize the pattern in the last section of this chapter. We annotate our text by messages containing references to a teacher.

3.3 The Shortest Path Problem

3.3.1 Summary of the problem

Floyd's algorithm[42] computes the shortest distances of a graph and is based on an algorithmic design technique called dynamic programming: simpler subproblems are first solved before the full problem is solved. It computes a distance matrix from a cost matrix: the costs of the shortest path between each pair of vertices are in $O(|V|^3)$ time.

Methodological note: *In the case of Floyd's algorithm, there is a mathematical definition of the matrix we have to compute from a starting state defining the initial basic link between nodes with cost. The function is called* d *and should be first defined in a context of the problem.*

The set of nodes N is $1..n$, where n is a constant value and the graph is simply represented by the distance function d ($d \in N \times N \times N \rightarrowtail \mathbb{N}$) and when the function is not defined, it means that there is no vertex between the two nodes. The relation of the graph is defined as the domain of the function d. n is clearly greater than 1 and it means that the set of nodes is not empty.

The distance function d is defined inductively from bottom to top a ccording to the dynamic programming principle and the next axioms define this function:

- $\boxed{axm1 : d \in N \times N \times N \rightarrowtail \mathbb{N}}$
- $\boxed{axm5 : \forall i \cdot i \in N \Rightarrow 0 \mapsto i \mapsto i \in dom(d) \wedge\ d(0 \mapsto i \mapsto i) = 0}$
- $\boxed{axm6 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge (k-1 \mapsto i \mapsto k \notin dom(d) \vee k-1 \mapsto k \mapsto j \notin dom(d)) \end{array} \right) \\ \Rightarrow \\ (\ k \mapsto i \mapsto j \in dom(d) \wedge\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j)\ ) \end{array} \right)}$
- $\boxed{axm7 : \forall i, j, k \cdot \begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge\, k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\, k-1 \mapsto k \mapsto j \in dom(d) \\ \wedge\, d(k-1 \mapsto i \mapsto j) \leq d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \wedge\, d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right) \end{array}}$
- $\boxed{axm8 : \forall i, j, k \cdot \begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge\, k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\, k-1 \mapsto k \mapsto j \in dom(d) \\ \wedge\, d(k-1 \mapsto i \mapsto j) > d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \wedge\, d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \end{array}}$
- $\boxed{axm9 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \notin dom(d) \\ \wedge\, k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\, k-1 \mapsto k \mapsto j \in dom(d) \end{array} \right) \\ \Rightarrow \\ k \mapsto i \mapsto j \in dom(d) \end{array} \right)}$

The optimality property is derived from the definition of d itself, since it starts by defining bottom elements and applies an optimal principle summarized as follows: $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1)+D_i(i+1,b))$ and means that the distances in D_i represent paths with intermediate vertices smaller than i; D_{i+1} is defined by comparing new paths including $i+1$. D_i is defined by a partial function over $N \times N \times N$. The partiality of d leads to some possible problems for computing the minimum and when at least one term is not defined, we should define a specific definition for the resulting term. Floyd's algorithm provides an algoithmic process for obtaining a matrix of all shortest possible paths with respect to a given initial matrix representing links between nodes together with their distance. Our first attempt was based on the computation of a shortest path between to given nodes a and b. The resulting matrix is called R and a boolean variable FD tells us if the shortest path exists. By the way, this first attempt is not the strict Floyd's algorithm but it will use the same principle of computation for the resulting matrix R.

The first step defines the *context* of the problem and the context is validated by the RODIN platform[69]. We decide to design an algorithm which is computing the value of the shortest path between two given nodes but using the same principle than Floyd's algorithm.

Methodological note: *The validation of the context SHORTESTPATH0 helps us to define carefully the function d. The translation of mathematocal properties is made easier by the notion of partial function. The expression* $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1) + D_i(i+1,b))$ *hides possible underfinedness and generally the non-existence of an edge between two nodes is defined by an extra value like* ∞*. We have to compute the following value* $\lambda i, j \in N.d(l \mapsto i \mapsto j)$ *but the* λ *notation is not directly usable in the B notations. However, we are computing in fact the value of d for the triple* $l \mapsto i \mapsto j$ *because it seems to be simpler to state.*

3.3.2 Writing the function call

The first model provides the declaration of the procedure shortestpath. Variables D and FD are *call-by-reference* parameters, constants l, a, b, D are *call-by-value* parameters:

procedure shortestpath$(l, a, b, G;$ **var** $D, FD)$
precondition $G = d_0 \wedge\ FD = FALSE \wedge\ l > 0 \wedge\ a \in N \wedge\ b \in N$
postcondition $(FD = true \Rightarrow D = d(l, a, b))$

We apply the *Call as Event principle* and we have to define a new model called SHORTESTPATH1 , which is defining an event corresponding to the action of calling the procedure.

$$\begin{tikzcd} \texttt{shortestpath(l,a,b,g,D,FD)} \arrow[r, "call-as-event"] & \mathsf{SHORTESTPATH1} \arrow[r, "\text{SEES}"] & \mathsf{SHORTESTPATH0} \end{tikzcd}$$

Methodological note: *The event is considered as a function call; we can explain at this time that the event is triggered because the guard is true. It is not a precondition.*

The new model SHORTESTPATH1 is using definitions of the context SHORTESTPATH0 . The event FLOYDKO models the fact that the call of `floyd` is returning a value FALSE for FD: there is no path between a and b. The event FLOYDOK returns the value TRUE for FDand the value of the minimal path from a to b. The two events are also interpreted by a procedure which is called with respect to the existence of a path.

```
MACHINE SHORTESTPATH1
SEES SHORTESTPATH0

VARIABLES
        $D$
        $FD$

INVARIANTS
        $inv1 : D \in N \times N \twoheadrightarrow \mathbb{N}$
        $inv2 : FD \in BOOL$

EVENTS

INITIALISATION
        BEGIN
                $act1 : D :| \left( \begin{array}{l} D' \in N \times N \twoheadrightarrow \mathbb{N} \\ \wedge (\forall i, j \cdot 0 \mapsto i \mapsto j \in dom(d) \Rightarrow i \mapsto j \in dom(D') \wedge\ D'(i \mapsto j) = d(0 \mapsto i \mapsto j))) \end{array} \right)$
                $act2 : FD := FALSE$
        END

EVENT shortestpathOK
        WHEN
                $grd1 : l \mapsto a \mapsto b \in dom(d)$
        THEN
                $act1 : D(a \mapsto b) := d(l \mapsto a \mapsto b)$
                $act2 : FD := TRUE$
        END

EVENT shortestpathKO
        WHEN
                $grd1 : l \mapsto a \mapsto b \notin dom(d)$
        THEN
                $act1 : FD := FALSE$
        END

END
```

Now, we have two events really non-deterministic, since they are defined using the constant d which should be computed in fact!. The solution is to refine the model SHORTESTPATH1 into a new model SHORTESTPATH2 which reduces non-determinism.

Methodological note: *It is very important to explain the difference between a flexible [53] variable and a rigid variable. Rigid variable like* d *denotes values which are defined as mathematical static objects and flexible variables denotes a name which is assigned to a value depending on the current state.*

3.3.3 Refining the procedure call

The main idea is to unfold the calls or to refine the events to get a model which is closer to an algorithm. We introduce several new variables:

- D and FD are both variables of the models SHORTESTPATH1 and SHORTESTPATH2 .
- c ($inv1 : c \in C$) expresses the control flow and the possible values of c are in the set C ($axm15 : C = \{start, end, step1, step2, step3, finalstep\}$).
- $D1$, $D2$ and $D3$ are three variables storing the values required for computing the next value of D at a given step; the values may be undefined and the undefinedness is controlled by the three variables $FD1$, $FD2$ and $FD3$. Variables are typed according to the following part of the invariant:

– $\boxed{inv2 : D1 \in \mathbb{Z}}$

– $\boxed{inv3 : D2 \in \mathbb{Z}}$

– $\boxed{inv4 : D3 \in \mathbb{Z}}$

– $\boxed{inv5 : FD1 \in BOOL}$

– $\boxed{inv6 : FD2 \in BOOL}$

– $\boxed{inv7 : FD3 \in BOOL}$

We do not give more details for the invariant and we will give later the details of the invariant of the current model. First we give the different events of the model SHORTESTPATH2 .
The event
INITIALISATION
is simply setting the variables as follows: $act1 : D := D0$, $act2 : FD := FALSE$, $act3 : FD1 := FALSE$, $act4 : FD2 := FALSE$, $act5 : FD3 := FALSE$, $act6 : D1 :\in \mathbb{Z}$, $act7 : D2 :\in \mathbb{Z}$, $act8 : D3 :\in \mathbb{Z}$, $act10 : c := start$.
Since $d(0 \mapsto i \mapsto j)$ models the existence of an elementary path from i to j, $D0$ is defined by the following axioms:

- $\boxed{axm12 : D0 \in N \times N \nrightarrow \mathbb{N}}$

- $\boxed{axm13 : dom(D0) = \{i \mapsto j | 0 \mapsto i \mapsto j \in dom(d)\}}$

- $\boxed{axm14 : \forall i, j \cdot i \mapsto j \in dom(D0) \Rightarrow D0(i \mapsto j) = d(0 \mapsto i \mapsto j)}$

Now, we can introduce refinement of existing events of SHORTESTPATH1 and new events which are not in the abstraction.

Refining events of SHORTESTPATH1

First, we give elements for competing the invariant; the typing informations can be completed as follows and they correspond to an analysis of the definition of d. We introduce a new variable c which is expressing the control state and whose possible values are given by the set C: $C = \{start, end, step1, step2, step3, finalstep\}$. We summarize the different steps for computing D.

Methodological note: *Using a graphical notation helps to communicate the meaning of control assertions. The steps of the algorithm appear. Moreover, steps provide a guide for defining the invariant which is based on the construction of d.*

$$\begin{tikzcd} & c = start \arrow[dl, "l\neq 0\ floyd\ call"'] \arrow[dr, "l=0"] & \\ c \in \{step1, step2, step3\} \arrow[r, "floyd\ call\ return"] & c = finalstep \arrow[r, "analysis"] & c = end \end{tikzcd}$$

Methodological note: *The invariant is based on the decomposition into steps and each step analyses the definition of values required for computing the minimum of $D1$ and $D2 + D3$. The invariant should take into account th definedness of these values and the tool helps us to complete the invariant.*

The *analysis* step provides a decision depending on the values of $D1$, $D2$ and $D3$, if they are defined. The boolean expression $FD1 \wedge (FD2 \vee FD3)$ is the key for updating $D(a \mapsto b)$ and it is triggered , when the control is finalstep.

Methodological note: *The expression $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1)+D_i(i+1,b))$ should be carefully analysed and it allows us to derive specific conditions for structuring the algorithm.*

When the control is at start:

- when l is initially equal to 0, D and d are equal too; D is defined when d is defined and reciprocally:
  - $\boxed{inv20 : c = start \land\ a \mapsto b \notin dom(D) \land\ l = 0 \Rightarrow 0 \mapsto a \mapsto b \notin dom(d)}$
  - $\boxed{inv8 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \land\, a \mapsto b \in dom(D) \\ \land\, l = 0 \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} 0 \mapsto a \mapsto b \in dom(d) \\ \land\, D(a \mapsto b) = d(0 \mapsto a \mapsto b) \end{array} \right) \end{array} \right)}$

- when l is not equal to 0 and there is no path from a to b with intermadiate nodes whose numbers is smaller than $l-1$, $a \mapsto b$ is not in D.
  - $\boxed{inv34 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \land\, l \neq 0 \\ \land\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \land\, l-1 \mapsto l \mapsto b \notin dom(d) \end{array} \right) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array} \right)}$
  - $\boxed{inv37 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \land\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \land\, l-1 \mapsto a \mapsto l \notin dom(d) \end{array} \right) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array} \right)}$

When the control is at end:
If the control is at end, the invariant enumerates the different cases for the resulting computation. The variable D should contain the values correspondin to l.

- $\boxed{inv12 : \begin{array}{l} \left(\begin{array}{l} c = end \\ \land\, FD = TRUE \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} a \mapsto b \in dom(D) \\ \land\, l \mapsto a \mapsto b \in dom(d) \\ \land\, D(a \mapsto b) = d(l \mapsto a \mapsto b) \end{array} \right) \end{array}}$
- $\boxed{inv14 : c = end \land\ FD = FALSE \Rightarrow a \mapsto b \notin dom(D) \land\ l \mapsto a \mapsto b \notin dom(d)}$
- $\boxed{inv18 : c = end \land\ l \mapsto a \mapsto b \notin dom(d) \Rightarrow FD = FALSE}$
- $\boxed{inv19 : c = end \land\ a \mapsto b \notin dom(D) \Rightarrow FD = FALSE}$
- $\boxed{inv21 : c = end \land\ a \mapsto b \in dom(D) \Rightarrow FD = TRUE}$

When the control is at finalstep:
The invariant states that the variables $FD1$, $FD2$ and $FD3$ are related to the definition of the expression $Min(D(a,b), D(a,l)+ D(l,b))$. $Min(D(a,b), D(a,l) + D(l,b))$. is defined,if, end only, if $FD1 \land\ (FD2 \lor FD3)$. The invariant explores the different cases for the definition of D for the given pairs. Moreover, the values are stired in the variables $D1$, $D2$ and $D3$ when defined.

- $\boxed{inv11 : \begin{array}{l} c = finalstep \land\ FD3 = TRUE \\ \Rightarrow \\ l-1 \mapsto l \mapsto b \in dom(d) \land\ D3 = d(l-1 \mapsto l \mapsto b) \end{array}}$

- $\boxed{\begin{array}{ll} & c = finalstep \wedge \ FD1 = TRUE \\ inv15: & \Rightarrow \\ & l-1 \mapsto a \mapsto b \in dom(d) \wedge \ D1 = d(l-1 \mapsto a \mapsto b) \end{array}}$
- $\boxed{\begin{array}{ll} & c = finalstep \wedge \ FD2 = TRUE \\ inv16: & \Rightarrow \\ & l-1 \mapsto a \mapsto l \in dom(d) \wedge \ D2 = d(l-1 \mapsto a \mapsto l) \end{array}}$
- $\boxed{inv13: \left(\begin{array}{l} \left(\begin{array}{l} c = finalstep \\ \wedge\, FD1 = FALSE \\ \wedge\, (FD2 = FALSE \vee FD3 = FALSE) \end{array}\right) \\ \Rightarrow \\ \left(\begin{array}{l} l \mapsto a \mapsto b \notin dom(d) \\ \wedge\, a \mapsto b \notin dom(D) \end{array}\right) \end{array}\right)}$
- $\boxed{inv24 : c = finalstep \wedge \ FD3 = FALSE \Rightarrow l-1 \mapsto l \mapsto b \notin dom(d)}$
- $\boxed{inv27 : c = finalstep \wedge \ FD2 = FALSE \Rightarrow l-1 \mapsto a \mapsto l \notin dom(d)}$
- $\boxed{inv29 : c = finalstep \wedge \ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$
- $\boxed{inv38: \left(\begin{array}{l} \left(\begin{array}{l} c = finalstep \\ \wedge\, FD1 = TRUE \\ \wedge\, (FD2 = FALSE \vee FD3 = FALSE) \\ \end{array}\right) \\ \Rightarrow \\ \left(\begin{array}{l} l \mapsto a \mapsto b \in dom(d) \\ \wedge\, d(l \mapsto a \mapsto b) = d(l-1 \mapsto a \mapsto b) \end{array}\right) \end{array}\right)}$

The diagram shows that `shortestpath` is made up of three steps.

$$\begin{tikzcd}
c = step1 \arrow[d, "evaluation\ FD1{,}D1"'] & & \\
c = step2 \arrow[r, "evaluation\ FD2{,}D2"] & c = step3 \arrow[r, "evaluation\ FD3{,}D3"] & c = finalstep
\end{tikzcd}$$

When the conrol is in $\{step1, step2, step3\}$:

- When the control is in $\{step1, step2, step3\}$, since $\boxed{inv28 : c \neq start \wedge \ c \neq end \Rightarrow l \neq 0}$, l is not equal to 0.
- When the control is at $step1$, l is not equal to 0. There are two conditions for the undefinedness of D in relationship to d.
  - $\boxed{\begin{array}{ll} & c = step1 \wedge \ l-1 \mapsto a \mapsto b \notin dom(d) \wedge \ l-1 \mapsto l \mapsto b \notin dom(d) \\ inv33: & \Rightarrow \\ & a \mapsto b \notin dom(D) \end{array}}$
  - $\boxed{\begin{array}{ll} & c = step1 \wedge \ l-1 \mapsto a \mapsto b \notin dom(d) \wedge \ l-1 \mapsto a \mapsto l \notin dom(d) \\ inv36: & \Rightarrow \\ & a \mapsto b \notin dom(D) \end{array}}$
- When the control is in $step2$, either the evaluation of $D1$ is successful or not.

- $\boxed{inv9 : c = step2 \land\ FD1 = TRUE \Rightarrow l-1 \mapsto a \mapsto b \in dom(d) \land\ D1 = d(l-1 \mapsto a \mapsto b)}$
- $\boxed{inv22 : c = step2 \land\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$
- $\boxed{inv32 : \begin{array}{l} c = step2 \land\ FD1 = FALSE \land\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}}$
- $\boxed{inv35 : \begin{array}{l} c = step2 \land\ l-1 \mapsto a \mapsto l \notin dom(d) \land\ FD1 = FALSE \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}}$

- When the control is in $step3$, either the evaluation of $D2$ is successful or not.
  - $\boxed{inv10 : \begin{array}{l} c = step3 \land\ FD2 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto l \in dom(d) \land\ D2 = d(l-1 \mapsto a \mapsto l) \end{array}}$
  - $\boxed{inv17 : \begin{array}{l} c = step3 \land\ FD1 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto b \in dom(d) \land\ D1 = d(l-1 \mapsto a \mapsto b) \end{array}}$
  - $\boxed{inv23 : c = step3 \land\ FD2 = FALSE \Rightarrow l-1 \mapsto a \mapsto l \notin dom(d)}$
  - $\boxed{inv25 : c = step3 \land\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$
  - $\boxed{inv26 : c \neq finalstep \land\ c \neq end \land\ 0 \mapsto a \mapsto b \notin dom(d) \Rightarrow a \mapsto b \notin dom(D)}$
  - $\boxed{inv30 : c = step3 \land\ FD1 = FALSE \land\ FD2 = FALSE \Rightarrow a \mapsto b \notin dom(D)}$
  - $\boxed{inv31 : \begin{array}{l} c = step3 \land\ FD1 = FALSE \land\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}}$

Refining shortestpathOK Now, we define each transition between the different steps according to the invariant. We consider severall possible cases depending on l and other conditions. When the value of l is 0 and when D is defined for the pair $a \mapsto b$, it means that there is a path between a and b without any intermediate node. It is the basic case and one returns the value TRUE for FD. The control is set to end, since the procedure is completed:

$$\begin{tikzcd}
 & l = 0 \land c = start \arrow[dl, "elementarypath"'] \\
l = 0 \land c = start \land a \mapsto b \in dom(D) \arrow[d, "floydOK"'] & \\
l = 0 \land c = end \land Fpath = TRUE &
\end{tikzcd}$$

```
EVENT shortestpathOK
REFINES shortestpathOK
    WHEN
        $grd2 : l = 0$
        $grd1 : a \mapsto b \in dom(D)$
        $grd3 : c = start$
    THEN
        $act2 : FD := TRUE$
        $act3 : c := end$
    END
```

When the control expresses the accessibility of the last control point ($c = finalstep$) and when the three values $D1$, $D2$ and $D3$ are defined and satisfy the condition $D1 \leq D2 + D3$, we can update D in $a \mapsto b$ by $D1$. In fact, the value is not modified. The control is set to the final control point called end. There is a path and FD is set to TRUE.

```
EVENT shortestpathcallOKmin
REFINES shortestpathOK
    WHEN
        $grd1 : FD1 = TRUE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
        $grd2 : D1 \leq D2 + D3$
        $grd3 : c = finalstep$
    THEN
        $act1 : D(a \mapsto b) := D1$
        $act2 : FD := TRUE$
        $act3 : c := end$
    END
```

The next case is stating that there is a new path from a to b, which is shortest than the current one ($grd3 : D1 > D2 + D3$) and we should update D by the new value $D2 + D3$.

```
EVENT shortestpathcallOKmax
REFINES shortestpathOK
    WHEN
        $grd1 : FD1 = TRUE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
        $grd2 : c = finalstep$
        $grd3 : D1 > D2 + D3$
    THEN
        $act1 : D(a \mapsto b) := D2 + D3$
        $act2 : c := end$
        $act3 : FD := TRUE$
    END
```

The next possible case is that the value $D1$ is not defined; it means that there is not yet a path from a to b and we have discovered that there is a node which can be reached from a and which can reach b. Hence, the variable D is defined in $a \mapsto b$ by the value $D2 + D2$.

```
EVENT shortestpathFD2FD3
REFINES shortestpathOK
    WHEN
        $grd1 : c = finalstep$
        $grd2 : FD1 = FALSE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
    THEN
        $act1 : D(a \mapsto b) := D2 + D3$
        $act2 : FD := TRUE$
        $act3 : c := end$
    END
```

Finally, when either $D2$ or $D3$ is not defined, the value of D is not modified and remains equal to $D1$.

EVENT shortestpathFD1
REFINES shortestpathOK
    WHEN
        $grd1 : c = finalstep$
        $grd2 : FD1 = TRUE \wedge \ (FD1 = FALSE \vee FD2 = FALSE)$
    THEN
        $act1 : D(a \mapsto b) := D1$
        $act2 : c := end$
        $act3 : FD := TRUE$
    END

The refinement of abstract events should be completed by events which compute the values $D1$, $D2$ and $D3$.

Refining shortestpathKO We consider severall possible cases depending on l and other conditions.
When the value of l is 0 and when D is not defined for the pair $a \mapsto b$, it means that there is no elementary path between a and b. It is the basic case and one returns the value FALSE for FD. The control is set to end, since the procedure is completed:

$$\begin{tikzcd}
l = 0 \wedge c = start \arrow[dr, "noelementarypath"] & \\
& l = 0 \wedge c = start \wedge a \mapsto b \notin dom(D) \arrow[d, "floydKO"'] \\
& l = 0 \wedge c = end \wedge Fpath = FALSE
\end{tikzcd}$$

EVENT shortestpathKO
REFINES shortestpathKO
    WHEN
        $grd2 : l = 0$
        $grd1 : a \mapsto b \notin dom(D)$
        $grd3 : c = start$
    THEN
        $act1 : FD := FALSE$
        $act2 : c := end$
    END

When the value of l is not 0 and when $D1$ is not defined and either $D2$ is not defined, or $D3$ is not defined, for the pair $a \mapsto b$, it means that there is no path between a and b. One returns the value FALSE for FD. The control is set to end, since the procedure is completed:

```
EVENT shortestpathKOelse
REFINES shortestpathKO
    WHEN
        $grd1 : c = finalstep$
        $grd2 : FD1 = FALSE \land \ (FD2 = FALSE \lor FD2 = FALSE)$
    THEN
        $act1 : c := end$
        $act2 : FD := FALSE$
    END
```

Introducing new events in SHORTESTPATH2

The first new event models the calling step of the procedure `floyd` and it transfers the control to the control point $step1$.

```
EVENT shortestpathcallone
    WHEN
        $grd1 : l > 0$
        $grd2 : c = start$
    THEN
        $act1 : c := step1$
    END
```

Now, we consider the three steps for computing $D1$, $D2$ and $D3$.

Calling the procedure `floyd` for evaluating $D1$ and $FD1$ The event $shortestpathcalltwook$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is successfully computed, since $FD1$ is TRUE. The event $shortestpathcalltwoko$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is unsuccessfully computed, since $FD1$ is FALSE.

```
EVENT
floydcalltwook
    WHEN
        $grd1 : c = step1$
        $grd2 : l - 1 \mapsto a \mapsto b \in dom(d)$
    THEN
        $act1 : D1 := d(l - 1 \mapsto a \mapsto b)$
        $act2 : FD1 := TRUE$
        $act3 : c := step2$
    END
```

```
EVENT shortestpathcalltwoko
    WHEN
        $grd1 : l - 1 \mapsto a \mapsto b \notin dom(d)$
        $grd2 : c = step1$
    THEN
        $act1 : FD1 := FALSE$
        $act2 : c := step2$
    END
```

Calling the procedure `floyd` for evaluating $D2$ and $FD2$ The event $shortestpathcallthreeok$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is successfully computed, since $FD2$ is TRUE. The event $shortestpathcallthreeko$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is unsuccessfully computed, since $FD2$ is FALSE.

```
EVENT shortestpathcallthreeok
    WHEN
        $grd1 : c = step2$
        $grd2 : l - 1 \mapsto a \mapsto l \in dom(d)$
    THEN
        $act1 : D2 := d(l - 1 \mapsto a \mapsto l)$
        $act2 : FD2 := TRUE$
        $act3 : c := step3$
    END
```

```
EVENT shortestpathcallthreeko
    WHEN
        $grd1 : c = step2$
        $grd2 : l - 1 \mapsto a \mapsto l \notin dom(d)$
    THEN
        $act1 : c := step3$
        $act2 : FD2 := FALSE$
    END
```

Calling the procedure `floyd` for evaluating $D3$ and $FD3$ The event $shortestpathcallfourok$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is successfully computed, since $FD3$ is TRUE. The event $shortestpathcallfourko$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is unsuccessfully computed, since $FD3$ is FALSE.

```
EVENT shortestpathcallfourok
    WHEN
        $grd1 : c = step3$
        $grd2 : l-1 \mapsto l \mapsto b \in dom(d)$
    THEN
        $act1 : c := finalstep$
        $act2 : D3 := d(l-1 \mapsto l \mapsto b)$
        $act3 : FD3 := TRUE$
    END
```

```
EVENT shortestpathcallfourko
    WHEN
        $grd1 : c = step3$
        $grd2 : l-1 \mapsto l \mapsto b \notin dom(d)$
    THEN
        $act1 : FD3 := FALSE$
        $act2 : c := finalstep$
    END
```

3.3.4 Producing the shortestpath procedure

The shortestpath procedure can be derived from the list of events of the model SHORTESTPATH2 and we structure events into conventional programming structures like `while` or `if` statements. J.-R. Abrial[5] has proposed several rules for producing algorithmic statements. The next diagram gives the complete description of the process we have followed:

$$\begin{tikzcd}
\texttt{shortestpath(l,a,b,G,D,FD)} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \mathsf{SHORTESTPATH1} \arrow[r, "\text{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \mathsf{SHORTESTPATH0} \\
\texttt{floyd} & \mathsf{SHORTESTPATH2} \arrow[l, "mapping"'] \arrow[ur, "\text{SEES}"] &
\end{tikzcd}$$

The procedure header is `shortestpath(l,a,b,G,D,FD)` and the text of the procedure is given by the algorithms 1 and 2.

Algorithm 3: Algorithm Version 1

```
precondition  : $l \in 1..n \wedge$
postcondition : $D, FD$
local variables: $FD1, FD2, FD3 \in BOOL$

$FD := FALSE$;
$FD1 := FALSE$;
$FD2 := FALSE$;
$FD3 := FALSE$;
if $l = 0$ then
    if $(a, b) \in dom(D)$ then
        $FD := TRUE$;
        $R := D[a, b]$;
    else
        $FD := FALSE$;
else
    $floyd(l-1, a, b, D1, FD1); floyd(l-1, a, l, D2, FD2); floyd(l-1, l, b, D3, FD3);$
    case $FD1 \wedge FD2 \wedge FD3$
        if $D1 < D2 + D3$ then
            $R := D1$;
        else
            $R := D2 + D3$;
        ;
        $FD := TRUE$;
    ;
    case $FD1 \wedge (\neg FD2 \vee \neg FD3)$
        $R := D1$;
        $FD := TRUE$;
    ;
    case $\neg FD1 \wedge (FD2 \wedge FD3)$
        $R := D2 + D3$;
        $FD := TRUE$;
    ;
    case $\neg FD1 \wedge (\neg FD2 \vee \neg FD3)$
        $FD := FALSE$;
    ;
;
```

The two next frames are containing C codes produced for the two algorithms 3.3.4 and 3.3.4; we have produced the C codes by hand and we have forgotten that C arrays starts by 0 and it means that our initial calls were wrongly written. It is clear that we need a way to produce codes in a mechanized way. Moreover, there are some conditions to check and some interactions to manage with the user to help in choices.

Methodological note: *It is the time to recall that we are planning to use a real programming language and that we should represent abstract objects by concrete objects. It would be better to add informations on the integers of computer scientists and it is easy to add the constraint.*

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;
  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1);  shortestpath(l-1,a,l,g,&D2,&FD2);
       shortestpath(l-1,l,b,g,&D3,&FD3);
      if (FD1 == 1 && (FD2==1 &&  FD3==1))
        {  if (D1 < D2+D3)
            {*D= D1;}
           else
             {*D=D2+D3;};
            *FD = 1;
        }
      else if (FD1==1 && ( FD2==0 ||  FD3==0))
        {*D= D1;*FD = 1;}
      else if (FD1==0 &&  ( FD2 == 1 && FD3==1)) {*D=D2+D3; *FD=1;}
      else /* (FD1==0 && ( FD2==0 || FD3==0)) */ { *FD = 0;}
    }
}
```

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;

  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1);
       if (FD1 == 1) {
          shortestpath(l-1,a,l,g,&D2,&FD2);
          if (FD2==1) {
            shortestpath(l-1,l,b,g,&D3,&FD3);
            if (FD3==1) {
            if (D1 < D2+D3)
              {*D= D1;}
            else
              {*D=D2+D3;};
            *FD = 1;}
            else
              {*D=D1;*FD=1;}}
          else
            {*D=D1;*FD=1;}}
       else
         {
           if  ( FD2 == 1 && FD3==1) {*D=D2+D3; *FD=1;}
       else
         {*FD=0;};}
    }}
```

The complete development has a cost related to proof obligations. The refinement generates 493 proof obligations and 328 proof obligations were automaically discharged. 165 proof obligations were manually discharged with minor interactions.

model	Total	Auto	Manual	Reviewed	Undischarged
SHORTESTPATH0	8	8	0	0	0
SHORTESTPATH1	5	4	1	0	0
SHORTESTPATH2	493	328	165	0	0
Global	506	340	166	0	0

Methodological note: *Proof obligations are not very difficult to discharge;there were based on the properties of* d *and it was boring to click the tool for discharging mechanichally them. Efforts were made on the definition of* d*.*

Now, it turns that our goal was to get Floyd's algorithm and we have an algorithm for computing the existence or the non existence of a shortest path between two nodes. The next section address the question.

3.4 Floyd's algorithm

We can use the developed algorithm to produce a result equivalent to Floyd's execution. In fact, we apply our algorithm on each pair of possible nodes and we store it in a matrix. The algorithm 3.4 describes the real algorithm which can be found in any lecture notes.

Now, we are considering the problem of derivation of this solution. In fact, the development starts from the same context. Two new constants are defined namely DF and Daf. Df is the final value of the matrix D correponding to d

Algorithm 4: Algorithm Version 2

```
precondition  : $l \in 1..n \wedge a, b \in \mathbb{N} \wedge G \in N \times N \twoheadrightarrow N$
postcondition : $D, FD$
local variables: $FD1, FD2, FD3 \in BOOL$

$FD := FALSE$;
$FD1 := FALSE$;
$FD2 := FALSE$;
$FD3 := FALSE$;
if $l = 0$ then
    if $(a, b) \in dom(D)$ then
        $FD := TRUE$;
        $R := D[a, b]$;
    else
        $FD := FALSE$;
else
    $shortestpath(l - 1, a, b, D1, FD1)$;
    if FD1 then
        $shortestpath(l - 1, a, l, D2, FD2)$;
        if FD2 then
            $shortestpath(l - 1, l, b, D3, FD3)$;
            if FD3 then
                if $D1 < D2 + D3$ then
                    $R := D1$;
                else
                    $R := D2 + D3$;
                ;
                $FD := TRUE$;
            else
                $R := D1$;
                $FD := TRUE$;
            ;
        else
            $R := D1$;
            $FD := TRUE$;
        ;
    else
        if $FD2 \wedge FD3$ then
            $R := D2 + D3$;
            $FD := TRUE$;
        else
            $FD := FALSE$;
        ;
;
```

Algorithm 5: Floyd's Algorithm Wikipedia

```
precondition  : $l \in 1..n \land matrix \in N \times N \rightarrow N$
postcondition : $matrix \in N \times N \rightarrow N \land$
local variables: $FD1, FD2, FD3 \in BOOL$

foreach $k = 1; k <= n; k++$ do
    foreach $i = 1; i <= n; i++$ do
        foreach $j = 1; j <= n; j++$ do
            if $matrix[i][j] > (matrix[i][k] + matrix[k][j])$ then
                $matrix[i, j] = matrix[i][k] + matrix[k][j]$
```

for the value l. C is simpler and is defined as follows: $\boxed{axm15 : C = \{start, end, call, finalstep\}}$.
New axioms define new constants:

- $\boxed{axm39 : Df \in N \times N \rightarrow N}$
- $\boxed{axm40 : dom(Df) = \{u \mapsto v | l \mapsto u \mapsto v \in dom(d)\}}$
- $\boxed{axm41 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)}$
- $\boxed{axm42 : Daf \in N \times N \rightarrow N}$
- $\boxed{axm43 : l \neq 0 \Rightarrow dom(Daf) = \{u \mapsto v | l-1 \mapsto u \mapsto v \in dom(d)\}}$
- $\boxed{axm44 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l-1 \mapsto u \mapsto v))}$
- $\boxed{axm22 : l = 0 \Rightarrow Df = D0}$
- $\boxed{axm23 : l \neq 0 \Rightarrow D0 \subseteq Daf}$
- $\boxed{axm24 : l \neq 0 \Rightarrow Daf \subseteq Df}$
- $\boxed{axm25 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l-1 \mapsto u \mapsto v))}$
- $\boxed{axm26 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)}$
- $\boxed{axm27 : \forall u, v \cdot u \mapsto v \in dom(D0) \Rightarrow D0(u \mapsto v) = d(0 \mapsto u \mapsto v)}$
- $$\boxed{axm28 : \begin{array}{l} (\ l \neq 0\ ) \\ \Rightarrow \\ \forall u, v, w \cdot \left(\begin{array}{l} w \mapsto v \in dom(Df) \\ \land\ w \mapsto u \in dom(Daf) \\ \land\ u \mapsto v \in dom(Daf) \\ \land\ Daf(w \mapsto v) > Daf(w \mapsto u) + Daf(u \mapsto v) \end{array} \right) \end{array}}$$

The new model FLOYD1 assigns the value Df to D. The new relationship between models and call is given by the next diagram:

$$\boxed{\begin{tikzcd} \texttt{floyd} \arrow[r, "call-as-event"] & \mathsf{FLOYD1} \arrow[r, "\text{SEES}"] & \mathsf{FLOYD0} \end{tikzcd}}$$

The problem is to refine the model FLOYD1 to get a list of events which lead to an algorithm. The two constants Df and Daf are used to state the final step and the intermediate step:

- Daf is the result of the call of the under construction algorithm for $l-1$
- Df is the final value which is computed from Daf.

We obtain the following diagram for expressing events corresponding to Floyd's algorithm:

$$\begin{tikzcd}
 & c = start \wedge l \in \mathbb{N} \wedge D = D0 \arrow[dl, "l{\neq}0\ floyd\ call"'] \arrow[dr, "l{=}0"] & \\
c = start \wedge l \neq 0 \wedge D = D0 \arrow[d, "tofloydcall"'] & & c = start \wedge l = 0 \wedge D = D0 \arrow[ddd, "floyd0"] \\
c = call \wedge l \neq 0 \wedge D = D0 \arrow[d, "floycall"'] & & \\
\begin{array}{l} c = finalcall \wedge l \neq 0 \wedge \\ TD = Daf \wedge D = D0 \end{array} \arrow[d, "floyd{-}ok"'] & & \\
\begin{array}{l} c = end \wedge l \neq 0 \wedge \\ TD = Daf \wedge D = Df \end{array} \arrow[dr, "l{\neq}0"] & & c = end \wedge l = 0 \wedge D = Df \arrow[dl, "l{=}0"'] \\
 & c = end \wedge D = Df &
\end{tikzcd}$$

The new model has three variables: c, D, TD.

- $\boxed{inv6 : TD \in N \times N \nrightarrow N}$
- $\boxed{inv1 : c \in C}$
- $\boxed{inv2 : c = start \Rightarrow TD = D0 \wedge\ D = D0}$
- $\boxed{inv3 : c = end \Rightarrow D = Df}$
- $\boxed{inv4 : c = call \Rightarrow TD = D0 \wedge\ l \neq 0}$
- $\boxed{inv5 : c = finalstep \Rightarrow TD = Daf \wedge\ l \neq 0}$

Initial conditions over variables are defined by $act1 : D := D0$, $act2 : c := start$, $act3 : TD := D0$. Events are very simple to write from the diagram:

EVENT floyd-ok
REFINES floyd
    WHEN
        $grd1 : c = finalstep$
    THEN

$$act1 : D, c : | \left(\begin{array}{l}
\left(D' \in N \times N \mapsto N \wedge c' = end \right) \\
\wedge \left(\begin{array}{ll} \forall w, v \cdot & \left(\begin{array}{l} \left(\begin{array}{l} w \in N \wedge v \in N \\ \wedge w \mapsto v \in dom(TD) \\ \wedge w \mapsto l \in dom(TD) \\ \wedge l \mapsto v \in dom(TD) \\ \wedge TD(w \mapsto v) > TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \\ \Rightarrow \\ D'(w \mapsto v) = TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{ll} \forall w, v \cdot & \left(\begin{array}{l} \left(\begin{array}{l} w \mapsto v \in dom(TD) \\ \wedge w \mapsto l \in dom(TD) \\ \wedge l \mapsto v \in dom(TD) \\ \wedge TD(w \mapsto v) \leq TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} w \mapsto v \in dom(D') \\ \wedge D'(w \mapsto v) = TD(w \mapsto v) \end{array} \right) \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{ll} \forall u, v \cdot & \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \in dom(TD) \\ \wedge (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD)) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} u \mapsto v \in dom(D') \\ \wedge D'(u \mapsto v) = TD(u \mapsto v) \end{array} \right) \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{ll} \forall u, v \cdot & \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \notin dom(TD) \\ \wedge (u \mapsto l \in dom(TD) \\ \wedge l \mapsto v \in dom(TD)) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} u \mapsto v \in dom(D') \\ \wedge D'(u \mapsto v) = TD(u \mapsto l) + TD(l \mapsto v) \end{array} \right) \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{ll} \forall u, v \cdot & \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \notin dom(TD) \\ \wedge (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD)) \end{array} \right) \\ \Rightarrow \\ u \mapsto v \notin dom(D') \end{array} \right) \end{array} \right)
\end{array} \right)$$

    END

The event
EVENT floyd-ok
uses a structure of Event B, which is assigning a value to variables and values are in a set. The set can be either empty, a singleton or a general set. In our case, the statement defines only one possible singleton and then the statement is clearly deterministic. However, we can subtitute the event by a call of a new procedure and we should starts a new development in another development using the same principle. We get the nestesd loops.

The two algorithms 3.4 and 3.4 are produced from the set of events. The recursive version is simply derived using the control points. The second algorithm is the iterative version which is produced by applying the classical transformations over recursive algorithms. The function nld is derived from an independant development by applying the same pattern.

EVENT floyd0
REFINES floyd
    WHEN
        $grd1 : c = start \wedge \ l = 0$
    THEN
        $act2 : c := end$
    END

```
EVENT tofloydcall
    WHEN
        $grd1 : c = start \wedge\ l > 0$
    THEN
        $act1 : c := call$
    END
```

```
EVENT floydcall
    WHEN
        $grd1 : c = call$
    THEN
        $act1 : c := finalstep$
        $act2 : TD := Daf$
    END
```

Algorithm 6: Recursive algorithm floyd

```
    precondition  : $l \in 1..n \wedge G$
    postcondition : $D$
    local variables: $TD$
    $TD := D0$;
    if $l \neq 0$ then
        $floyd(l-1, G, TD)$;
        $D := nld(TD)$;
    else
        $D := TD$;
    ;
```

Algorithm 7: Non-recursive algorithm floyd

```
    precondition  : $l \in 1..n \wedge G$
    postcondition : $D$
    local variables: $TD$
    $TD := D0$;
    $l := 0$;
    while $l \neq 0$ do
        $TD := d(TD)$;
    ;
    $D := TD$;
```

What we have learnt from the case study is summarized as follows:

1. Developing a first abstract one-shot model using pre/post-condition. It provides the declarations part of the procedure (method) related to the one-shot model. The basic structure to express is the function d which the key of the problem. Constants of the model are defined as *call-by-value* parameters and variable of the model are *call-by-reference* parameters, The context SHORTESTPATH0 is clearly reusable and we have reused it for the effective algorithm of Floyd.

2. Refining the abstract model to obtain the body of procedure. New variables are defined as *local variables*. The refinement introduces control states which provide a way to structure the body of the procedure. We have clearly the first control point namely *start* and the last control point namely *end*. The diagram helps to decompose the procedure into steps of the call and a special control point called *call* is introduced. The main question is to obtain a deterministic transition system in the new refinement model.

3. If there are still remaining non-deterministic events, we can eliminate the non-deterministic events by developing each non-deterministic event in a specific B development starting by the statement of a new problem expressed by the non-deterministic event itself. In fact, it is what is done with the last version of Floyd's algorithm and the event computing D' from TD is clearly refined to get two nested loops.

4. Proof obligations are relatively easy to check because the invariant is written by a list of properties of d according to d. Evene if the number of *manual* proof obligations is high, it was very easy to discharge them using the prover and to reuse former intercative ones.

5. The translation of Event B model into a C program was carried out by hand and we did a mistake. We forgot that C arrays are starting the index by 0 and it leads to a bad call. We should mechanize this step to avoid this mistake.

3.5 Summary of the pattern

The pattern is simply defined by the following diagram:

$$\begin{tikzcd}
\texttt{call} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \mathrm{M1} \arrow[r, "\text{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \mathrm{M1} \\
\texttt{procedure} & \mathrm{M2} \arrow[l, "mapping"'] \arrow[ur, "\text{SEES}"] &
\end{tikzcd}$$

- `call` is the call statement which is defining the pre and post conditions over formal parameters.
- M0 is a context defining the mathematical framework of the problem to solve.
- M1 is a machine which is defining the `call` as an event instance.
- M2 is a refinement of M1; if an event of the refinement is non-deterministic, we are defining a new call of new procedure and the pattern should be reapplied in this case. The refinement allows us to introduce control points, which can be usd to generate an algorithm from the M2 machine.
- The *mapping* transformation models the generation of an algorithm from a list of events.
- The *call-as-event* transformation is simply a translation or an embedding of the problem in an Evenet B model.

We have developed the following case studies using the pattern:

- Binomials coefficients
- Sorting by selection [28]
- Sorting by insertion

A tool is under development for supporting the application of the pattern and the generation of the algorithm by a mechanical process. The first transformation is very technical, since it requires to state the problem and the mathematical expression of the mathematical environment The main benefit is thye fact the resulting algorithm is correct by construction according to the pre and post conditions.
The technique of developmment is a top/down approach, which is clearly well known in earlier works of Dijkstra[37, 62], and to use the refinement for controlling the correctness of the resulting algorithm. It relies on a more fundamental question related to the notion of *problem to solve*.

Chapter 4

The *access control* pattern

Sommaire

We address the proof-based development of (system) models satisfying a security policy. The security policy is expressed in OrBAC or RBAC models, which allows one to state permissions and prohibitions on actions and activities and belongs to the family of role-based access control formalisms. The main question is to validate the link between the security policy expressed in OrBAC and the resulting system; a first abstract B model is derived from the OrBAC specification of the security policy and then the model is refined to introduce properties that can be expressed in OrBAC. The refinement guarantees that the resulting B (system) model satisfies the security policy. We present a development pattern of a system with resepct to a security policy and it can be instnatiated later for a given security policy.

4.1 Introduction

One of the most challenging problems in managing large networks is the complexity of security administration. Role-based access control has become the predominant model for advanced access control because it reduces the complexity and cost of security administration in large networked applications. Others models, like OrBAC [1], have been introduced by providing a structure based on the application domain and by introducing the concept of organisation. Networks or software systems can be abstracted by action systems or event B models; however, security requirements should be integrated into the proof-based design of such systems and we address the integration of security policy - expressed in a security model OrBAC - in the final systems. This leads us to deal with security properties like permissions and prohibitions. We leave obligations as out of the scope of the current work. This study consists to elaborate a pattern for modelling systems which control the how a set of subjects perform actions on a set of objects defined in the system. J.-R. Abrial [2] contributes to the access control problem: the study consists to elaborate a system which controls the access to a building at differents persons. He does not refer to security model but influences our current work.

4.1.1 Integration of security policies in system development

When a system is under development, it is necessary to consider requirements documentation. The document is either written in a natural language, or in a semi-formal language, or in a formal language and it may include different aspects or views of the target system. Security policy is a possible part of this document and it may be expressed in a specific modelling language designed for expressing permissions, prohibitions, recommendations, obligations, . . . related to the target system. Now, a key question is to ensure that the resulting system conforms to the security policy and it appears to us that in the existing systems the link between the system and its security policy is not clearly established and formally validated, as stated by the figure 4.1: the satisfaction relation should be established in a formal way. We illustrate the problem to be solved by considering two modelling languages:

- the OrBAC modelling language for security policy
- the event B modelling language for systems

Another important point is that we focus on the access control problem and in the figure 4.2, we describe several steps to obtain an implementation of the system from the statement of the security policy:

1. Generating a B model OM from the security policy O: the translation relation is explained in the current paper and can be mechanized.
2. Generating a B model RM by refining OM and by adding progressively details of the document which are not yet integrated into the current model: the refinement of B models is the key concept ensuring the validation of the satisfaction relation..
3. Writing a system model $\mathcal{SYS}$ from the last B model: the implementation of a refined B model into a system language can be directed by transformations over events.

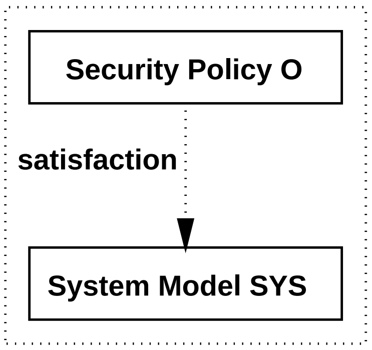


Figure 4.1: The satisfaction relation

The goal of this work is to elaborate a proof-based developement pattern of models satisfying a security policy. The security policy can be expressed in a formal language and it is possible to analyse the security policy, especially the consistency of the policy. The refinement ensures the correctness of the satisfaction relation: the system satisfies the security policy.

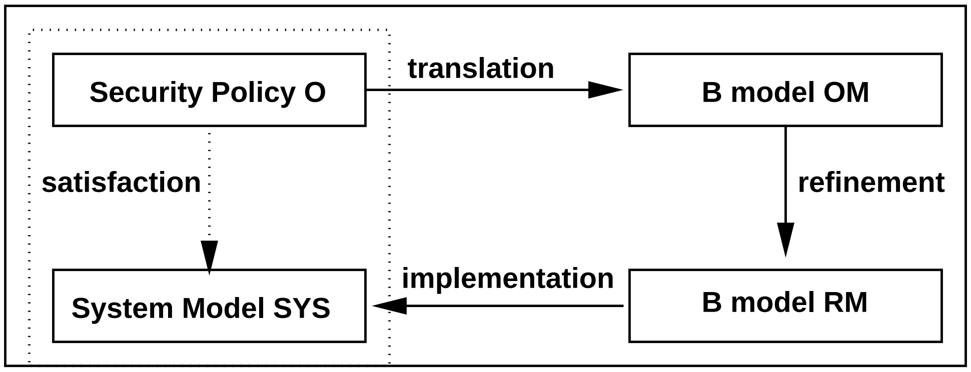


Figure 4.2: Global process description

4.2 Models for Security Policy

The interaction of people with IT systems generate various security needs to guarantee that each system user benefits of its advantages without trespassing on someonelse's rights. These needs vary according to the activity field required. It could be regarding: Confidentiality (Non disclosure of sensitive information to non authorised persons), Integrity (Non alteration of sensitive information), Availability (Supply of information to users according to their rights to access these information), Auditability (The ability to trace and determine the actions carried out in the system).
Such requirements usually result in setting up an access control model that expresses security policies, defining for each user his permissions, prohibitions and obligations. Users (or subjects) are active entities operating on objects (passive entities) of the system.
Several access control models have been proposed: DAC [54], MAC [13, 14], RBAC [40, 70, 44] or OrBAC [1]. In the Role- Based access control model, the RBAC model, security policy does not directly grant permissions to users but to roles [40]. A role is an abstraction for users. Each user is assigned to one or several roles, and will inherit permissions or prohibitions associated with these roles. Such a security model states security properties on the target system and on a hidden state of the current system. The hidden state is clearly stating dynamic properties related to permissions and prohibitions. The classical role-based models have no explicit state variable; the context information might be used to express the state changes but we think that a state-based approach like B provides a simpler framework for integrating security policy specification in the design of a system. Moreover, the refinement may help us in introducing security properties in a proof-based step.

4.2.1 Organization-Based Access Control model: OrBAC

The OrBAC (Organization-Based Access Control model) for modelling the security policies is an extension of the RBAC model. OrBAC is based on the concept of organization. The specification of the security policy is completely parametrized by the organization such that it is possible to handle simultaneously several security policies associated with different organizations [1].
Another advantage of the OrBAC model compared to other models is that it makes it possible to express contextual permissions or prohibitions.
OrBAC takes again the concept of role such as it was defined in RBAC. Users are assigned to roles and inherit their privileges. The concept of $view$ (or object's groups) is also introduced as an abstraction of the objects of the system. The construction of these groups of objects must be semantically well founded, this construction is related to the way in which the various roles carry out various actions on these objects. It should be noted that there are similarities with the concept of $view$ in relational databases where it is a question of gathering objects which have similar properties. Just as for the objects, the actions are also gathered in activities, this implies that there are two levels of abstraction in OrBAC:

- Abstract level: roles (doctor, nurse), activities (management) and views (patient files, administration files) of the system on which various permissions and prohibitions are expressed.
- Concrete level: subjects (Paul, Peter, John), actions (create, delete) and objects (patient_file1, patient_file2) of the system.

Subjects, actions and objects are respectively assigned to roles, activities and views by relations defined over these entities(see figure 4.3). We detail relations in the next sub-section.

Empower, Use and Consider

Assignment of subjects to roles: subjects are assigned to one or more roles in order to define their privileges. Contrary to RBAC, subjects play their roles in organizations, which implies that subjects are assigned to roles through a ternary relation including the organization:

$empower(org, s, r)$: means that the subject s plays the role r in the organization org.

Assignment of actions to activities: As for roles and subjects, activities are an abstraction of various actions authorized in the system. The relation binding actions to activities is also a ternary relation including the organizations:

$consider(org, a, act)$: means that the action a is considered as an activity act in the organization org.

Assignment of objects to views: As in relational databases, a view in OrBAC corresponds to a set of objects having a common property. The relation binding the objects to the views to which they belong is also a ternary relation including the organization:

$use(org, o, v)$: means that the organization org uses the object o in the view v.

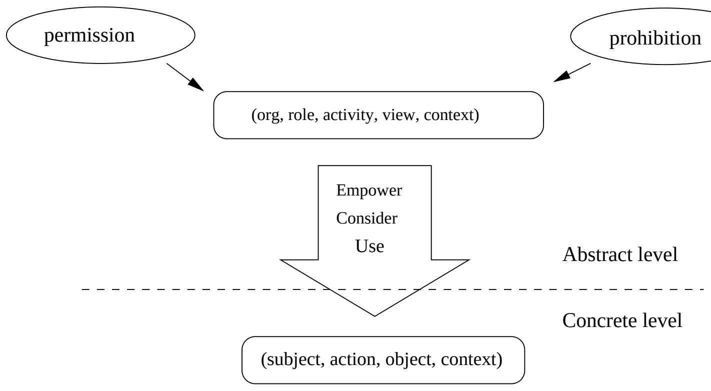

permission
prohibition
(org, role, activity, view, context)
Empower
Consider
Use
Abstract level
Concrete level
(subject, action, object, context)

Figure 4.3: Abstract and Concrete level of OrBAC

Modeling a security policy with OrBAC

When subjects, actions, and objects are respectively assigned to roles, activities and views, it is now possible to describe the security policy. It consists of defining different permissions and prohibitions:

- $permission(org, r, act, v, c)$: means that the organization org grants to the role r the permission to carry out the activity act on the view v in context c.
- $prohibition(org, r, act, v, c)$: means that the organization org prohibits the role r to carry out the activity act on the view v in the context c.

The concept of context, which did not exist in RBAC, is important in OrBAC, since it makes it possible to express contextual permissions (or prohibitions). Let us consider the example of a security policy in a medical environment. If one wants to restrict the access to patients records or files to their attending practitioner, the following permission should be added to the security policy:

$$permission(hospital, physician, consult, patient\_file, attending\_practitionar)$$

If there is no context:

$$permission(hospital, physician, consult, patient\_file)$$

A physician could therefore access the file of any patient, which needs to be avoided. To be able to use this concept of context, a new relation $define$ should be introduced:

Define(org, s, a, o, c): means that within organization org, the context c is true between subject s, the object o and action a.

Hierarchy in OrBAC

The OrBAC model makes it possible to define role hierarchies (as in RBAC) but also with respect to the organization hierarchies. The hierarchies allow the inheritance of the privileges (permissions or prohibitions), if for example $r2$ is a sub-role of $r1$, for an organization org, an activity av and a view v in the context ctx:

$$permission(org, r1, av, v, ctx) \rightarrow permission(o, r2, av, v, ctx)$$

and

$$prohibition(org, r1, av, v, ctx) \rightarrow prohibition(org, r2, av, v, ctx)$$

In the same way for the organizations, if $org2$ is a sub-organization of $org1$ then, for a role r an activity av and a view v in the context ctx:

$$permission(org1, r, a, v, ctx) \rightarrow permission(org2, r, av, v, ctx)$$

and

$$prohibition(org1, r, a, v, ctx) \rightarrow prohibition(org2, r, av, v, ctx)$$

The concept of inheritance is a key concept in OrBAC, since it allows gradual building of the security policy. Indeed, it is necessary to start by establishing a flow chart of the organizations (and roles) and defining the privileges on the basic organizations, it will then be enough to add gradually the privileges of the sub-organisations(sub-roles).

4.3 Event B models from OrBAC

A complete introduction of B can be found in [24]. The question is to integrate the event B method and the OrBAC method; we have shortly introduced the event B concepts and the OrBAC concepts. In a B model, we should define the mathematical structures on which is based the development and the system under development; this information can be used to derive further properties that will be used in the validation of models. The B models have a static part and a dynamic part and in the specification of a security policy in OrBAC one has to state dynamic properties and to check the consistency of the resulting theory. The MOTOrBAC tool [35] provides a framework for defining a security policy and for checking the consistency of the set of facts and rules in a PROLOG-like style; this approach is clearly based on a fixed-point definition of permissions. The question of expressing administration model in OrBAC is also very crucial and it is very simple to express the administration of security policy in B, since one can model the permissions as a variable satisfying the security policy expressed in an invariant. These points will be recalled when we present the effective translation of OrBAC models into event B models.
The current status of the work is as follows:

- We assume to have an OrBAC description of the security policy.
- The security policy is supposed to be stable and consistent; the consistency is checked using tools like, for instance MOTOrBAC.
- The security policy states permissions and prohibitions.

The problem is to translate OrBAC statements into the event B modelling language.
The translation of the security policy into event B includes several successive stages. A first B model is built and then other successive refinements are made as shown by figure 4.4. The refinement validates the link between the abstract level (role, ...) and the concrete level (subject, ...).

4.3.1 Abstract model with permissions and prohibitions

Such as presented in the paragraph 4.2.1, the OrBAC specification has two levels of abstraction (see figure 4.3).
The first step consists in a event B model modelling the abstract part of the security policy, i.e. initially, only concepts of organization, role, view, activity and context are considered. In the first model permissions and prohibitions of the OrBAC model should be described.

- The clause $SETS$ in the event B model contains basic sets such as organisations, roles, activities, views and contexts: *ORGS*, *ROLES*, *ACTIVITIES*, *VIEWS*, *CONTEXTS*.

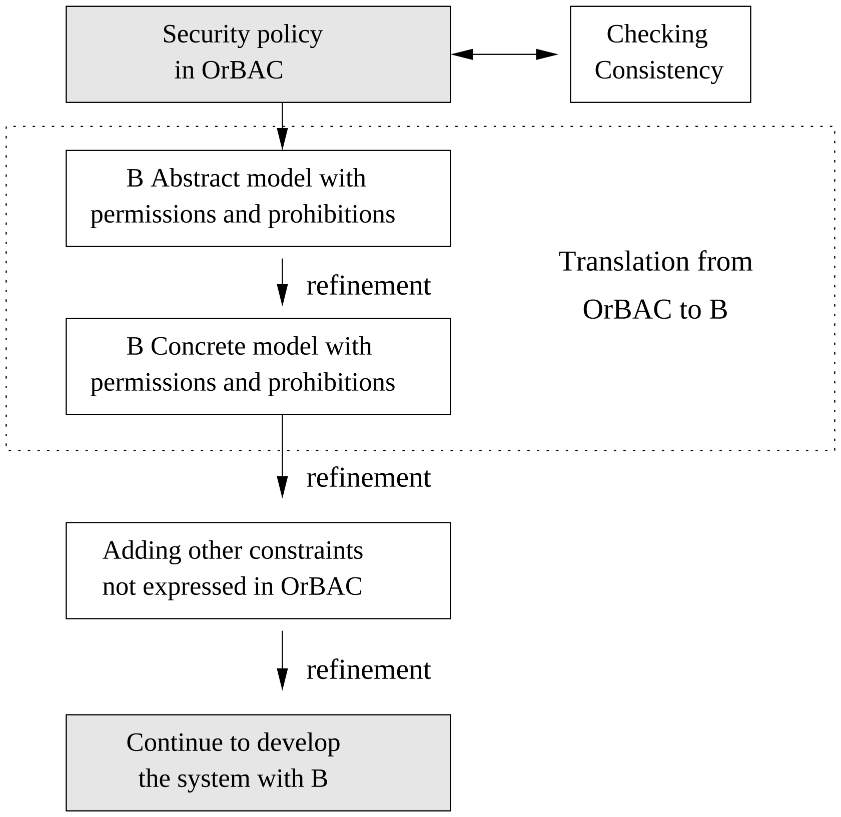

Security policy in OrBAC
Checking Consistency
B Abstract model with permissions and prohibitions
refinement
B Concrete model with permissions and prohibitions
Translation from OrBAC to B
refinement
Adding other constraints not expressed in OrBAC
refinement
Continue to develop the system with B

Figure 4.4: Steps of the passage from OrBAC to a B event-based model

- The clauses $CONSTANTS$ et $PROPERTIES$ contain the constants like $permission$ and $prohibitions$ that will contain privileges of the OrBAC description. One of the most important concepts contained in OrBAC is the concept of hierarchy whether it is organization hierarchy or role hierarchy. Two new constants sub_role and sub_org are introduced to take into account respectively the role and organization hierarchy. It is enough to specify which roles and which organizations are concerned with inheritances, permissions and prohibitions corresponding to inheritances are deductively generated.

```
SETS
  ORGS;
  ROLES;
  ACTIVITIES;
  VIEWS;
  CONTEXTS
```

```
CONSTANTS
  permission,
  prohibition,

  sub_org,
  sub_role,

  default /* default context value */
```

PROPERTIES
$permission \subseteq ORGS \times ROLES \times ACTIVITIES \times VIEWS \times CONTEXTS$
$prohibition \subseteq ORGS \times ROLES \times ACTIVITIES \times VIEWS \times CONTEXTS$

$sub\_org \subseteq ORGS \times ORGS$
$sub\_role \subseteq ROLES \times ROLES$

$default \in CONTEXTS$

```
/* Organization hierarchies */
∀(org1, org2, r, av, v, ctx).(
   (org1 ∈ ORGS ∧ org2 ∈ ORGS∧
   r ∈ ROLES ∧ av ∈ ACTIVITIES∧
   v ∈ VIEWS ∧ ctx ∈ CONTEXTS∧
   (org1 ↦ org2) ∈ sub_org∧
   (org2 ↦ r ↦ av ↦ v ↦ ctx) ∈ permission)
⇒
   (org1 ↦ r ↦ av ↦ v ↦ ctx) ∈ permission)

/* Role hierarchies */
∀(org, r1, r2, av, v, ctx).(
   (r1 ∈ ROLES ∧ r2 ∈ ROLES∧
   org ∈ ORGS ∧ av ∈ ACTIVITIES∧
   v ∈ VIEWS ∧ ctx ∈ CONTEXTS∧
   (r1 ↦ r2) ∈ sub_role∧
   (org ↦ r2 ↦ av ↦ v ↦ ctx) ∈ permission)
⇒
   (org ↦ r1 ↦ av ↦ v ↦ ctx) ∈ permission)

/* Properties for prohibitions */
```

To apply the pattern to a given particular case, it is enough to initialize sets in the clause SETS by entities, organizations, roles, views, activities, contexts. Properties of constants, like $permission$, $interdiction$, sub_role and sub_org, should also be set in the clause $PROPERTIES$. Consequently, permissions and prohibitions can not be modified, since they are defined as constants; the OrBaC definitions are expressing properties satisfied by a consistent theory of permissions and prohibitions. We will address later the administration of OrBaC.

Introducing state variables

An event B model expresses properties over state and state variables; the main problem is effectively that OrBAC has no explicit variables. In fact, OrBAC users are using some kind of state modifications but no explicit state exists in OrBAC, even if contexts might be used to model it. Variables are used to model the status of the system with respect to permissions and authorizations:

- The clause $VARIABLES$ contains two variables, the state variable $hist\_abst$ that contains the history of system activities; the variable $context$ determines the running context of the system.

Variables satisfy the following properties added to the invariant:

INVARIANT
$context \in CONTEXTS$
$hist\_abst \subseteq ORGS \times ROLES \times ACTIVITIES \times VIEWS \times CONTEXTS$
$hist\_abst \subseteq permission$

The initial values of the two variables are set as follows:

$$context := default \; || hist\_abst := \varnothing \quad (4.1)$$

As the security policy is supposed to be consistent, we should be able to prove in the clause $ASSERTIONS$:

```
ASSERTIONS
    $pemission \cap prohibition = \varnothing$
    $hist\_abst \cap prohibition = \varnothing$
```

- The clause $EVENTS$ contains the following events :
  - The event $action$ models, when an authorization request for the access of a subject to an object of the system occurs.
  - The two events $set\_default$ and $set\_context\_value$ are attached to the change of the system context.

```
action ≙
  any  $org, r, v, av$  where
    $org \in ORGS$
    $r \in ROLES$
    $v \in VIEWS$
    $av \in ACTIVITIES$
    $(org \mapsto r \mapsto av \mapsto v \mapsto context) \in permission$
  then
    $hist\_abst := hist\_abst \cup \{(org \mapsto r \mapsto av \mapsto v \mapsto context)\}$
  end
```

```
set_context_default ≙
  begin
    $context := default$
  end
```

```
set_context_value ≙
  begin
    $context :\in CONTEXTS - \{default\}$
  end
```

The invariant should be preserved and it means that any activity in the system is controled by the security policy through the variable $hist\_abst$.

4.3.2 First refinement: Concrete model with permissions and prohibitions

One of the goals is to use the refinement to validate the relation between security models; OrBaC defines two levels of abstractions and the current model is refined into a concrete model. The refinement introduces subjects, actions and objects: sets $SUBJECTS$, $ACTIONS$ and $OBJECTS$ contain respectively subjects, actions and objects of the system under development. The clause $CONSTANTS$ includes the following constants:

- $empower$: assignment of $subjects$ to $roles$.
- use: assignment of subjects to $views$.
- $consider$: assignment of $actions$ to $activities$.

and properties of constants are stated as follows:

```
PROPERTIES
    $empower \subseteq ORGS \times ROLES \times SUBJECTS$
    $use \subseteq ORGS \times VIEWS \times OBJECTS$
    $consider \subseteq ORGS \times ACTIVITIES \times ACTIONS$
```

As for the abstract model, to apply the pattern to a given particular case, it is enough to initialize sets in the clause SETS by entities, subjects, objects and activities. Properties of constants, like *empower*, *use*, *consider*, should also be set in the clause $PROPERTIES$.

Concrete variables

A new variable $hist\_conc$ models the control of the system according to the security policy; it contains the history of actions occurences performed by a subject on a given object. The context in which the action occurred is also stored in this variable.

$$hist\_conc \subseteq SUBJECTS \times ACTIONS \times OBJECTS \times CONTEXTS$$

The relation between $hist\_conc$ and the variable $hist\_abst$ of the abstract model is expressed in the gluing invariant: The first part of the invariant states properties satisfied by variables with respect to permissions:

```
INVARIANT
$\forall(s, a, o, ctx).($
    $(s \in SUBJECTS \wedge a \in ACTIONS \wedge$
    $o \in OBJECTS \wedge ctx \in CONTEXTS \wedge$
    $(s \mapsto a \mapsto o \mapsto ctx) \in hist\_conc) \wedge$
$\Rightarrow$
    $(\exists(org, r, av, v).($
    $org \in ORGS \wedge r \in ROLES \wedge$
    $av \in ACTIVITIES \wedge v \in VIEWS \wedge$
    $(r \mapsto s) \in empower \wedge$
    $(v \mapsto o) \in use \wedge$
    $(av \mapsto a) \in consider \wedge$
    $(org \mapsto r \mapsto av \mapsto v \mapsto ctx) \in hist\_abst)))$
```

The invariant states that each action performed by the the system satisfies the security policy.

For the prohibitions, when a subject s wants to carry out an action a on an object o in an organization org, it is necessary to check it does not exist a prohibition :

$$prohibition(org, r, act, v, ctx) \wedge empower(org, r, s) \wedge use(org, v, o) \wedge consider(org, act, a)$$

for any organization org, activity av, view v and ctx as current context.
The second part of the invariant states properties satisfied by variables with respect to prohibitions:

```
INVARIANT
$\forall(s, a, o, ctx).($
    $(s \in SUBJECTS \wedge a \in ACTIONS \wedge$
    $o \in OBJECTS \wedge ctx \in CONTEXTS \wedge$
    $(s \mapsto a \mapsto o \mapsto ctx) \in hist\_conc) \wedge$
$\Rightarrow$
    $(\forall(org, r, av, v).($
    $org \in ORGS \wedge r \in ROLES \wedge$
    $av \in ACTIVITIES \wedge v \in VIEWS \wedge$
    $(r \mapsto s) \in empower \wedge$
    $(v \mapsto o) \in use \wedge$
    $(av \mapsto a) \in consider) \wedge$
    $\Rightarrow$
        $(org \mapsto r \mapsto av \mapsto v \mapsto ctx) \notin prohibition)))$
```

The events

The refinement of the event $action$ from the abstract model should take in consideration permissions, prohibitions for a subject s that ask to perform an action a on an object o.

```
action $\widehat{=}$
  **any** $s, a, o, org, r, v, av$ **where**
    $s \in SUBJECTS \wedge a \in ACTIONS \wedge o \in OBJECTS \wedge$
    $org \in ORGS \wedge r \in ROLES \wedge$
    $av \in ACTIVITIES \wedge v \in VIEWS \wedge$
    $(r \mapsto s) \in empower \wedge$
    $(v \mapsto o) \in use \wedge$
    $(av \mapsto a) \in consider \wedge$
  $/*\ permission\ */$
    $(org \mapsto r \mapsto av \mapsto v \mapsto ctx) \in permission \wedge$
  $/*\ prohibition\ */$
    $(\forall(orgi, ri, avi, vi).($
    $(orgi \in ORGS \wedge ri \in ROLES \wedge$
    $avi \in ACTIVITIES \wedge vi \in VIEWS \wedge$
    $(ri \mapsto s) \in empower \wedge$
    $(vi \mapsto o) \in use \wedge$
    $(avi \mapsto a) \in consider) \wedge$
    $\Rightarrow$
    $((orgi \mapsto ri \mapsto avi \mapsto vi \mapsto ctx) \notin prohibition))$
  **then**
    $hist\_conc := hist\_conc \cup \{(s \mapsto a \mapsto o \mapsto context)\}$
  **end**
```

Discussion on contextual security policies

In the different cases we studied, It appeared that the context notion has two different aspects. The first aspect concerns the contexts that are global to the system. An example of this global contexts is a system managing accesses to a building in a company. We may have a permission (or a prohibition) :

$$permission(company, agent, access, building, opening\_hours)$$

In this permission, the context $opening\_hours$ is global to the system, i.e. the whole system is, at a given moment, in the context $default$ or $opening\_hour$. A state variable $context$ indicating the running context of the system is used in this case. On the other hand, in the case, for example, of a system managing the access to the patient files in a hospital, we may have permissions of the form:

$$permission(hospital, physician, consult, patient\_file, attending\_practitionner)$$

In this permission, the context $attending\_practitionner$ (that means that the permission is valid only if the physician is the attending practitionar of the patient) isn't global to the system but links subjects to the objects. In this case a new constant $define$ (as in OrBAC) is used. This constant defines links between objects and subjects, it has the following form :

$define \subseteq ORGS \times SUBJECTS \times ACTIONS \times OBJECTS \times CONTEXTS$

In order to give the system designer the possibility of expressing contextual permissions of each type, modifications must be made to the B model. If $context\_value$ is a value of a global context, invariant should be modified as follows :

INVARIANT
$\forall(s, a, o, ctx).($
$\quad (s \in SUBJECTS \wedge a \in ACTIONS \wedge$
$\quad o \in OBJECTS \wedge ctx \in CONTEXTS \wedge$
$\quad (s \mapsto a \mapsto o \mapsto ctx) \in hist\_conc) \wedge$
$\Rightarrow$
$\quad (\exists(org, r, av, v).($
$\quad org \in ORGS \wedge r \in ROLES \wedge$
$\quad av \in ACTIVITIES \wedge v \in VIEWS \wedge$
$\quad (r \mapsto s) \in empower \wedge$
$\quad (v \mapsto o) \in use \wedge$
$\quad (av \mapsto a) \in consider \wedge$
$\quad (((org \mapsto s \mapsto a \mapsto o \mapsto ctx \mapsto) \in define) \vee (ctx = context\_value)) \wedge$
$\quad (org \mapsto r \mapsto av \mapsto v \mapsto ctx) \in hist\_abst)))$

4.3.3 Second refinement: Adding other constraints not expressed in OrBAC

The state variables of the event B model give additional information on the system which was not available with OrBAC. It was impossible to know at a given moment the state of the system and, for example, which member of a company staff consulted or modified which file. This point is important since in practice the security policies are increasingly complex and new types of constraints appears. The passage towards B allows us to implement the security policy such as it was established in OrBAC, and enrich it with the possibility of introducing new constraints. We aim to include constraints in our pattern such as workflow constraints or the duty separation. The pattern user can then choose the constraints he wants to include in his model depending from the case study.

Workflow constraints

Workflow constraints express properties on the task scheduling in a system. For instance, a rule for a given workflow states that an action act should be executed, only if a set of actions $act1$, $act2$..., $actn$ is already executed (see figure 4.5). Those constraints can not be expressed in OrBAC, because, when a subject is assigned to a given role, it obtains its complete privileges. A permission is systematically delivered to execute the action act, if one of the roles to which a subject is assigned, has the appropriate privilege, even if one of the actions $act1$, $act2$,..., $actn$ has not yet been executed. The implementation of these constraints in a B model leads to the following invariant:

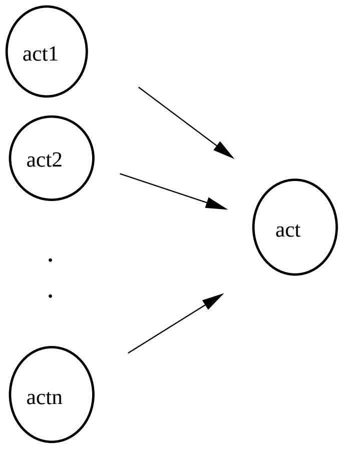


Figure 4.5: Workflow constraints

INVARIANT
$\forall(s, o, ctx).($
$\quad(s \in SUBJECTS \wedge$
$\quad o \in OBJECTS \wedge$
$\quad ctx \in CONTEXTS \wedge$
$\quad(s \mapsto act \mapsto o \mapsto ctx) \in hist\_conc) \wedge$
$\Rightarrow$
$\quad(\exists(sw, cw).(sw \in SUBJECTS \wedge cw \in CONTEXTS \wedge$
$\qquad(sw \mapsto act1 \mapsto o \mapsto cw) \in hist\_conc) \wedge$
$\quad\exists(sw, cw).(sw \in SUBJECTS \wedge cw \in CONTEXTS \wedge$
$\qquad(sw \mapsto act2 \mapsto o \mapsto cw) \in hist\_conc) \wedge$
$\quad\exists(sw, cw).(sw \in SUBJECTS \wedge cw \in CONTEXTS \wedge$
$\qquad(sw \mapsto actn \mapsto o \mapsto cw) \in hist\_conc)))$

The refinement provides a way to add such a constraint in model and proof obligations ensure the correctness of the transformation. Another refinement can be done to introduce specific rules like for instance duty separation.

Duty separation

Duty separation aims to prevent fraud and errors by disseminating action's execution privileges among different subjects. To implement a system satisfying this type of constraints, it is necessary that when a subject asks for the authorization to execute an action on an object to be able to check, if it did not already act throughout the process, which is impossible to do with OrBAC in a simple way.
However, there is a form of duty separation known as static duty separation (implemented with RBAC [40]). This one consists in preventing a subject to cumulate several important functions, it can be achieved when subjects are assigned to roles. In the B model, the following assertion should be proved to guarantee that no subject cumulates two critical given roles $r1$, $r2$. In the clause $ASSERTIONS$:

$$\forall ss.\ ((ss \in SUBJECTS\ \wedge (\ org \mapsto r1 \mapsto ss) \in\ empower) \Rightarrow\ (org \mapsto r2 \mapsto ss) \notin\ empower)$$

Proceeding this way may be too rigid in some cases. A subject s can cumulate several functions if it does not intervene many times in the management of the same object o. To prevent that a subject s executes two critical actions $act1$, $act2$ on an object o with $act1 \neq act2$, the following invariant has been proved:

INVARIANT
$\forall(s1, s2, o, ctx1, ctx2).($
$\quad(s1 \in SUBJECTS \wedge s2 \in SUBJECTS \wedge$
$\quad o \in OBJECTS \wedge$
$\quad ctx1 \in CONTEXTS \wedge ctx2 \in CONTEXTS \wedge$
$\quad(s1 \mapsto act1 \mapsto o \mapsto ctx1) \in hist\_conc) \wedge$
$\quad(s2 \mapsto act2 \mapsto o \mapsto ctx2) \in hist\_conc) \wedge$
$\Rightarrow$
$\quad(s1 \neq s2))$

The duty separation and workflow constraints are only particular cases of constraints where instant system state must be known in order to express them.

4.3.4 Refinement for distributing control

The main idea is to refine the current model with workflow constraints into a model which splits the abstract event $action$ into two concrete events attached to the control of permissions and prohibitions and the workflow control. The computation of the guard of the abstract event $action$ is decomposed into the computation of two guards. In further refinement, the two events will be localized to a given part of the real system.
We want to implement a simplified WfMS (Workflow Management System) [41]. Figure 4.6 shows different components of the WfMS. Workflow Database contains workflow steps' definitions and constraints, it also contains informations about current workflow progress. Scheduling is performed by the workflow engine which refers to the workflow database to have information about workflow progress.
There are two events in this refinement:

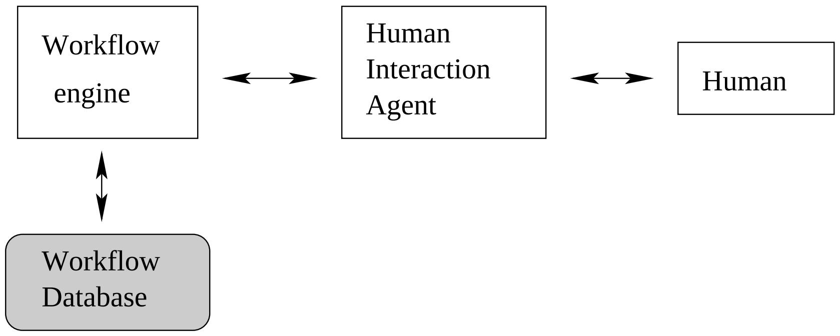


Figure 4.6: A simplified WfMS

- Event $action\_aut$: that will be executed on the Human Interaction Agent, this event checks only for the permissions and prohibitions.
- Event $action$: that is the refinement of the event $action$ of the previous model. It checks for the workflow and duty separation constraints. It will be executed on the workflow engine.

Variables

We keep the variable $hist\_conc$, this variable will be implemented in the workflow database. Two new variables tmp_aut and $bool\_workflow$ are added for the synchronization of the two events $action\_aut$ and $action$:

- tmp_aut: contains an action that has the necessary privileges to be executed and for which workflow constraints are not already checked.
- $bool\_workflow$: a boolean variable used to synchronize the two events $action\_aut$ and $action$.

```
action_aut ≙
  **any** $s, a, o, org, r, v, av$ **where**
    $s \in SUBJECTS \wedge a \in ACTIONS \wedge$
    $o \in OBJECTS \wedge$
    $org \in ORGS \wedge r \in ROLES \wedge$
    $av \in ACTIVITIES \wedge v \in VIEWS \wedge$
    $(r \mapsto s) \in empower \wedge$
    $(v \mapsto o) \in use \wedge$
    $(av \mapsto a) \in consider \wedge$
  $/*\ permission\ */$
    $(org \mapsto r \mapsto av \mapsto v \mapsto ctx) \in permission \wedge$
  $/*\ prohibition\ */$
    $(\forall(orgi, ri, avi, vi).($
    $(orgi \in ORGS \wedge ri \in ROLES \wedge$
    $avi \in ACTIVITIES \wedge vi \in VIEWS \wedge$
    $(ri \mapsto s) \in empower \wedge$
    $(vi \mapsto o) \in use \wedge$
    $(avi \mapsto a) \in consider) \wedge$
    $\Rightarrow$
    $((orgi \mapsto ri \mapsto avi \mapsto vi \mapsto ctx) \notin prohibition))$
  $/*\ synchronization\ */$
    $bool\_workflow = FALSE$
  **then**
    $tmp\_aut := (s \mapsto a \mapsto o \mapsto context) ||$
    $bool\_workflow = TRUE$
  **end**
```

```
action $\widehat{=}$
  any $s, a, o, ctx$ where
    $s \in SUBJECTS \wedge a \in ACTIONS \wedge$
    $o \in OBJECTS \wedge ctx \in CONTEXTS \wedge$
    $tmp\_aut = (s \mapsto a \mapsto o \mapsto ctx) \wedge$
  $/*\ Workflow\ constraints\ */$
    $(a = act$
  $\Rightarrow$
    $(\exists(sw, cw).(sw \in SUBJECTS \wedge$
    $cw \in CONTEXTS \wedge$
    $(sw \mapsto act1 \mapsto o \mapsto cw) \in hist\_conc) \wedge$
    $\exists(sw, cw).(sw \in SUBJECTS \wedge$
      $cw \in CONTEXTS \wedge$
      $(sw \mapsto act2 \mapsto o \mapsto cw) \in hist\_conc) \wedge$
    $\exists(sw, cw).(sw \in SUBJECTS \wedge$
      $cw \in CONTEXTS \wedge$
      $(sw \mapsto actn \mapsto o \mapsto cw) \in hist\_conc))) \wedge$
  $/*\ Duty\ separation\ */$
    $(a = act1$
  $\Rightarrow$
    $\forall(sp, cp).($
    $(sp \in SUBJECTS \wedge cp \in CONTEXTS \wedge$
    $(sw \mapsto act2 \mapsto o \mapsto cp) \in hist\_conc)$
    $\Rightarrow$
      $sp \neq s)) \wedge$
    $(a = act2$
  $\Rightarrow$
    $\forall(sp, cp).($
    $(sp \in SUBJECTS \wedge cp \in CONTEXTS \wedge$
    $(sw \mapsto act1 \mapsto o \mapsto cp) \in hist\_conc)$
    $\Rightarrow$
      $sp \neq s)) \wedge$
  $/*\ synchronization\ */$
    $bool\_workflow = TRUE$
  then
    $hist\_conc := hist\_conc \cup \{(s \mapsto a \mapsto o \mapsto ctx)\} ||$
    $bool\_workflow = FALSE$
  end
```

4.4 Conclusion and open issues

The development of software systems satisfying a given security policy should be based on techniques for validating the link between the security policy and the resulting system. The link between the security policy and the system is called satisfaction and we have used the event B method, especially the refinement, for relating the security policy expressed in OrBAC and the final system. Our work is greatly influenced by the case study developed by J.-R. Abrial [2]; he shows how a system for controling access to buildings, can be derived by refinement and he starts by expressing the essence of the access control. In our case, we use an elaborate formalism OrBAC for expressing the security policy and for checking its consistency; we derive a mathematical theory from OrBAC specification and we define an explicit state of a system which is not explicit in OrBAC. The refinement provides us a way to develop a list of models which progressively integrate details that seems to be not possible to express in OrBAC: workflow constraints, for instance. Our models are generic with respect to the security policy and can be reused to develop a real system. We have also mentionned the fact that one can distribute the control of guard and its evaluation. A crucial question would be to use our models for developing an infrastructure for controling an existing system with respect to a security policy. Moreover, security policy is expressing permissions and prohibitions but it remains to consider obligations which are very difficult to refine because they are close to liveness properties and should be expressed on traces. Finally, case studies should be developed using this pattern.

Chapter 5

The *cryptographic* pattern

Sommaire

We consider the proof-based development of cryptographic protocols satisfying security properties. The main motivation is that cryptographic protocols are very complex systems to prove and to design, since they are based on specific assumptions. For instance, the model of Dolev-Yao provides a way to integrate a description of possible attacks when designing a protocol. We use existing protocols and want to provide a systematic way to prove but also to design cryptographic protocols; moreover, we would like to provide patterns for integrating cryptographic elements in an existing protocol. For instance, the MONDEX case study integrates cryptographic elements and we have exprimented the way to take into account the model of Dolev-Yao. Communication channels are supposed to be unsafe. Analysing cryptographic protocols requires precise modelling of the attacker's knowledge. In this chapter, we present a pattern for modelling and proving key establishment protocols. The attacker's behaviour conforms to the Dolev-Yao model. In the Dolev-Yao model, the attacker has full control of the communication channel, and the cryptographic primitives are supposed to be perfect.

5.1 Introduction

To provide a secure communication between two agents over an unsecure communication channel, these agents should establish a *fresh key* to use in their subsequent communications. The chosen *session key* must be known only by the two agents involved in the communication, it also needs to be a fresh key to avoid using a previously established key in a previous session. There is several cryptographic protocols dedicated to key establishment that aim to provide such properties. To be able to prove such properties on a protocol, we must be able to model the knowledge of the attacker. A *pet* model of attacker's behaviour is the Dolev-Yao model [38]; this model is an informal description of all possible behaviours of the attacker as described in section 5.3.4. In this chapter, we present a pattern to model and prove key establishment protocols using event B [3, 15, 26] as a modelling language. We give a presentation of the pattern and an application of the pattern on two protocols: the well known Needham-Schroeder public key protocol [64] and Blake-Wilson-Menezes key transport protocol [16].
Proving properties on cryptographic protocols such as *secrecy* is known to be undecidable. However, works involving formal methods for the analysis of security protocols have been carried out. Theorem provers or model checkers are usually used for proving properties. For model checking, one famous example is Lowe's approach [56] using the process calculus CSP and the model checker FDR. Lowe discovered the famous bug in Needham-Schroeder's protocol. Model checking is efficient for discovering an attack if there is one, but it can not guarantee that a protocol is reliable. We should be carefull on the question of stating properties of a given protocol and it is clear that the modelling language should be able to state a given property and then to check the property either using model checking or theorem proving. Other works are based on theorem proving: Paulson [67] used an inductive approach to prove safety properties on protocols. He defined protocols as sets of traces and used the theorem prover Isabelle [66]. Other approaches, like Bolignano [17], combines theorem proving and model checking taking general formal method based techniques as a framework. Let us remember that we focus on a correct-by-construction approach and we are not proposing new cryptographic protocols.

5.2 Introducing the protocols

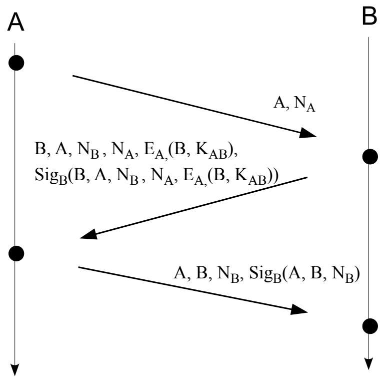
Two protocols illustrate the usefullness of our pattern. We have already developed the MONDEX case study [73] and we have identified a structure for this kind of protocol. Protocols are summarizd by diagrams showing the information flows and the interactions among agents.

5.2.1 Blake-Wilson-Menezes key transport protocol

This protocol is a key transport protocol. Agent B creates a fresh session key K_{BA} and sends it to the agent A. The protocol is based in signed messages using public cryptographic keys in order to provide mutual authentication.

5.2.2 Needham-Schroeder public key protocol

This protocol provides mutual authentication using exchanged shared nonces N_a, N_b (see figure 5.2). These nonces can be used as shared secret for key establishment, this is why the last message that contains N_b remains encrypted even if it is not necessary for authentication. We consider in this paper the Lowe's fixed version of the protocol. Lowe discovered an attack on this protocol using FDR model checker and proposed a variant protocol where the identifier B was added in the second message of the protocol run.



A
B
A, N_A
B, A, N_B, N_A, E_A(B, K_AB),
Sig_B(B, A, N_B, N_A, E_A(B, K_AB))
A, B, N_B, Sig_B(A, B, N_B)

Figure 5.1: Blake-Wilson-Menezes key transport protocol.

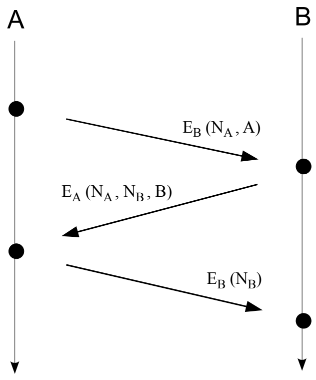


Figure 5.2: Needham-Schroeder public key protocol.

Both protocols can be modelled using interaction diagrams. The proposed pattern is based on this observation and on integration of elements of attack.

5.3 Pattern for modelling the protocols

The pattern is defined by a proof-based development of Event B models which are modelling protocols in a very abstract and general way. A protocol is a system which is controling the traffic of messages between agents. The three models defining the development are mentionned in the next figure:

- Abstract model: In this first model, different steps of the the protocol run are modelled using the notion of *abstract transactions*. A transaction has different attributes such as it's source and destination. These attributes are used to express safety properties we want to prove on our protocol.
- First refinement: In this first refinement, we add the remaining parts of the protocol that were not modelled in the abstract model. Attacker event keep the invariant preserved.
- Second refinement: In the second refinement, attacker event models the behaviour of a Dolev-Yao style attacker. The attacker knowledge is modelled and used to prove that the safety properties of the model are maintained though attacker's behaviour.
- Third refinement: The last refinement is a data refinement where the abstract transactions are replaced by concrete nonces.

We should recall that the goal is to help the developer in discharging proof obligations and the table of proof obligations for the pattern applied to the two protocols is given by the next table:
Now, we give first a description of the pattern and then we show how it is applied to model both protocols.

Table 5.1: Proof obligation of the Needham-Schroeder public key protocol

Model	Total number of PO	Automatics	Interactives
Abstract model	60	60 (100%)	0 (0%)
First refinement	71	71 (100%)	0 (0%)
Second refinement	44	32 (73%)	12 (27%)
Total	175	163 (94%)	12 (6%)

Table 5.2: Proof obligation of the Blake-Wilson-Menezes key transport protocol

Model	Total number of PO	Automatics	Interactives
Abstract model	50	50 (100%)	0 (0%)
First refinement	50	50 (100%)	0 (0%)
Second refinement	10	7 (70%)	3 (30%)
Total	110	107 (97%)	3 (3%)

5.3.1 Abstract model

The pattern is based on the notion of *abstract transactions*. Safety properties will first be expressed over these abstract transactions. In cryptographic protocols, nonces are used to identify each session or protocol run. Intuitively, each transaction of the abstract model corresponds to a fresh nonce in the concrete model. A transaction has several attributes and, before giving these attributes, we need to introduce the basic sets we will use in our model.

SETS
T
$Agent$
MSG

AXIOMS
$axm1 : I \in Agent$

- T is the set of abstract transactions
- $Agent$ is the set of agents
- MSG is the set of possible messages among agents.
- $axm1 : I \in Agent$ expresses the existence of a special agent called the intruder.

Note that for most protocols, even if there is more than one dishonnest agent in the system, it suffices to consider only one attacker that will combine the abilities and knowledge of all the other dishonnest agents.

Variables

In public key protocols, we often have an agent that initiates the protocol run by sending a message to a given agent and then waits for the corresponding answer. This answer is usually encrypted with the source agent public key or signed by the destination agent private key. After receiving this answer, the source agent trusts the authenticity of the destination agent identity. Our pattern captures this idea thus a transaction has a source (t_src) and a destination (t_dst). A running transaction is contained in a set $trans$. When a transaction terminates it is added to a set end.

INVARIANTS
$inv1 : trans \subseteq T$
$inv2 : end \subseteq trans$
$inv3 : t\_src \in trans \rightarrow Agent$
$inv4 : t\_dst \in trans \rightarrow Agent$

The answer from the destination agent is transmitted via a channel ($channel$). A message from this channel has a source and a destination (msg_src, msg_dst) but also a variable (msg_t) that binds the message to a transaction.

$inv5 : channel \subseteq MSG$
$inv6 : msg\_src \in channel \rightarrow Agent$
$inv7 : msg\_dst \in channel \rightarrow Agent$
$inv8 : msg\_t \in channel \rightarrow T$

To complete the authentication of the destination agent we need an additional variable t_bld_dst that contains the believed destination agent by the source agent where the variable t_dst contains the real destination agent.

$inv9 : t\_bld\_dst \in trans \rightarrow Agent$

In this case, *to prove authentication, we need to prove that both variables are equal when a transaction terminates*.

INVARIANTS
$inv10 : \forall t \cdot t \in end \Rightarrow t\_dst(t) = t\_bld\_dst(t)$

Events

The figure 5.3 contains a description of the model events. In the beginning of a transaction, the agent A sets the value of the variable t_bld_dst with some agent B and adds the transaction t to the set $trans$. An agent B answers by sending a message to A, the variable msg_src is set to the value B. Events of the model are enumerated as follows:

- **EVENT Init**: The transaction source agent initiates the transaction by adding this transaction to the set $trans$ and sets the values of the variables t_src to himself and t_bld_dst to the agent he wants to communicate with.
- **EVENT V**: The transaction destination agent answers the source agent by sending a message on the variable $channel$ and sets the variable msg_src to himself for the sent message.
- **EVENT End**: The transaction source agent receives a message corresponding to this transaction. The variable t_dst is set to the received message source agent contained in the variable msg_src.
- **EVENT Attk**: The attacker sends messages to randomly chosen agents to try to mislead them about his identity. The variable msg_src is set to the attacker's identity.

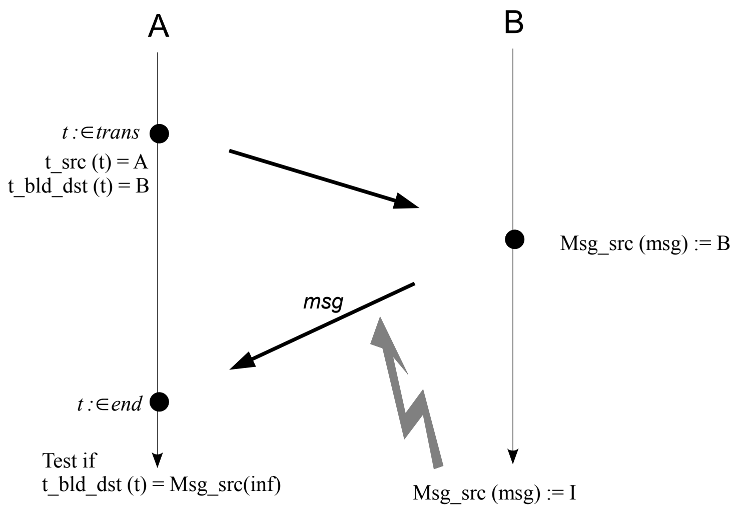


Figure 5.3: Pattern for modelling public key protocols

```
EVENT Init
    ANY
        $t$
        $A$
        $B$
    WHERE
        $grd1 : t \in T \setminus trans$
        $grd2 : A \in Agent \wedge$
                $B \in Agent \wedge$
                $A \neq B$
    THEN
        $act1 : trans := trans \cup \{t\}$
        $act2 : t\_src(t) := A$
        $act3 : t\_bld\_dst(t) := B$
    END
```

```
EVENT V
    ANY
        $A$
        $B$
        $t$
        $msg$
    WHERE
        $grd1 : A \in Agent \wedge$
                $B \in Agent \wedge$
                $A \neq B$
        $grd2 : t \in T$
        $grd3 : msg \in MSG \setminus channel$
    THEN
        $act1 : channel := channel$
                        $\cup \{msg\}$
        $act2 : msg\_src(msg) := B$
        $act3 : msg\_dst(msg) := A$
        $act4 : msg\_t(msg) := t$
    END
```

When an agent A receives a message that corresponds to a transaction, he initiated from an agent B, he sets the variable t_dst to the value B. Thus, the variable t_dst contains the real transaction destination. The value of this variable is not set in the V event, when the agent B sends the message because many agents may answer to agent A's request and the real transaction destination is known only once A receives one of the messages. Since a message may contain additional informations like a shared session key that B may send to A, when A receives the key, t_dst will contain the identity of the transaction key issuer.

Depending of the protocol structure, the agent A should know, if the source of the message he receives is the trusted destination of the transaction to guarantee the authentication of the protocol. But in this abstract model, we add the *guard 8* that guarantees this property.

```
EVENT End
    ANY
        $t$
        $A$
        $B$
        $msg$
    WHERE
        $grd1 : t \in trans \setminus end$
        $grd2 : A \in Agent \wedge B \in Agent \wedge A \neq B$
        $grd3 : t\_src(t) = A$
        $grd4 : msg \in channel$
        $grd5 : msg\_t(msg) = t$
        $grd6 : msg\_src(msg) = B$
        $grd7 : msg\_dst(msg) = A$
        $grd8 : msg\_src(msg) = t\_bld\_dst(t)$
    THEN
        $act1 : end := end \cup \{t\}$
        $act2 : t\_dst(t) := B$
    END
```

We also add in this model the attacker event. In this event the attacker can add a message with randomly chosen attributes to the *channel*.

```
EVENT Attk
    ANY
        $t$
        $A$
        $msg$
    WHERE
        $grd1 : t \in T$
        $grd2 : A \in Agent$
        $grd3 : msg \in MSG \setminus channel$
        $grd4 : A \neq I$
    THEN
        $act1 : channel := channel \cup \{msg\}$
        $act2 : msg\_src(msg) := I$
        $act3 : msg\_dst(msg) := A$
        $act4 : msg\_t(msg) := t$
    END
```

Another event modelling the loss of messages is added. Messages are removed from the *channel* randomly. This loss can be caused by a malicious attacker action or by an error in the communication channel.

Invariant

To guarantee authentication, the following invariant was added and proved. It states that *for completed transaction, the trusted destination is the real transaction destinations*.

$$inv11 : \forall t \cdot t \in end \Rightarrow t\_dst(t) = t\_bld\_dst(t)$$

This invariant is easy to prove even for the event **Attk**, because of the *guard 8* of the event **End**.

5.3.2 Applying the pattern

In the Needham Schroeder public key protocol (see figure 5.2) and the Blake-Wilson-Menezes key transport protocol (see figure 5.1), the agent A first initiates a transaction and waits for the answer from agent B. The agent B does the same, and waits for A's answer. The figure 5.4 shows how the pattern is applied two times to model each protocol.

When the pattern is applied, the corresponding variables, events and invariants are generated. For both protocols, the following variables are generated : $transA$, $transB$, A_t_src, B_t_src, A_t_dst, B_t_dst, $A\_t\_bld\_dst$, $B\_t\_bld\_dst$. *Invariant10* is generated two times :

```
INVARIANTS
    $\forall t \cdot t \in end \Rightarrow A\_t\_dst(t) = A\_t\_bld\_dst(t) \land$
    $\forall t \cdot t \in end \Rightarrow B\_t\_dst(t) = B\_t\_bld\_dst(t)$
```

5.3.3 First refinement

In this refinement, the remaining details of the modelled protocol are added. For instance, in the last step of the Blake-Wilson-Menezes protocol, agent A sends to agent B a signed message containing A, B, N_B. In this message, B is contained in the variable msg_dst and N_B is in the variable msg_t, an additional variable is needed for modelling the attribute A in the message. In the second step of the Needham Schroeder public key protocol, agent B sends to agent A a message containing $(B, N_B, N_A)_{KA}$, an additional variable is also needed for modelling the attribute B in the message. This is done exhaustively with all the modelled protocol steps.
Let MSG_VAR the set of additional variables. In the abstract model we use the *guard 8* in the **EVENT End** to prove authentication, with this guard agent A could know if the message authentic or not. In cryptographic protocols it is not possible to perform such tests except for the signed messages. Thus for the Blake-Wilson-Menezes protocol, agent B signs his message with his private key, when agent A receives the message, he can apply B's public key on the

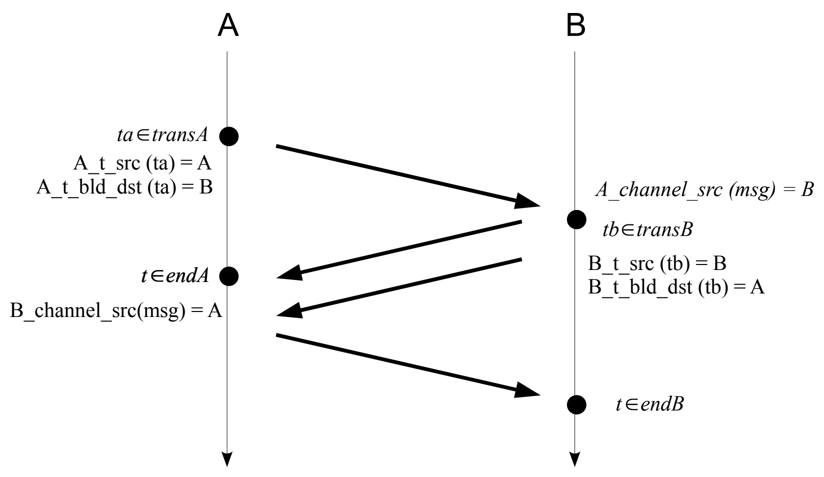


Figure 5.4: Pattern for modelling public key protocols

message and check if the source of the message is really B. Note that this is not true for the Needham Schroeder public key protocol, since agent B doesn't sign his message. In this case, the structure of the message itself guarantees authentication. Accordingly, when messages are not signed, *guard 8* in the **EVENT End** have to be substituted by a condition over the received message content. This condition is a predicate over the set MSG_VAR, we call it $Protocol\_Cond(MSG\_VAR, t)$. The predicate is directly derived from the protocol itself. The **EVENT End** of the pattern becomes[1]:

```
EVENT End
    ANY
        $t$
        $A$
        $B$
        $msg$
    WHERE
        $\oplus\ grd8 : Protocol\_Cond(MSG\_VAR, t)$
        $\ominus\ grd8 : msg\_src(msg) = B$
    THEN
        $act1 : end := end \cup \{t\}$
        $act2 : t\_dst(t) := B$
    END
```

To prove that the concrete **EVENT End** refines the abstract event, the following invariant have to be added:

INVARIANTS

$inv12 : \forall t, msg, A, B \cdot$
$t \in trans \wedge A \in Agent \wedge B \in Agent \wedge msg \in channel \wedge$
$t\_src(t) = A \wedge t\_bld\_dst(t) = B \ \wedge msg\_t(msg) = t$
$protocol\_Cond(MSG\_VAR, t)$
$\Rightarrow$
$msg\_src(msg) = t\_bld\_dst(t)$

[1] $\oplus$ and $\ominus$ are respectively the added and removed guards compared to the refined event.

The attacker

The next refinement models attackers' knowledge and , in this refinement, the **EVENT Attk** keeps the $invariant12$. To achieve this we call $Attk\_Cond(t)$ the weakest condition that maintains the $invariant12$. Note that this predicate is obtained automatically from the invariant. We use $Attk\_Cond$ to refine the *event Attk* as follows:

```
EVENT Attk
    ANY
        $t$
        $A$
        $msg$
    WHERE
        $\oplus\ \ grd7 : Attk\_Cond(t)$
    THEN
        $act1 : \oplus MSG\_VAR$
    END
```

Applying the pattern

To apply the pattern, we first need to define every additional variable. In the Needham Schroeder public key protocol, agent B answers agent A's request by resending A's nonce and his own identity B. Thus, we add the variable msg_Agt.

$$inv13 : msg\_Agt \in channel \to Agent$$

In this protocol, when agent A receives B's answer, he checks if the identity of the agent in the message he received (msg_Agt), is the same than the identity of the agent, he sends the message to (t_bld_dst). Thus, in this case, the predicate $Protocol\_Cond(MSG\_VAR, t)$ is:

$$msg\_Agt(msg) = t\_bld\_dst(t)$$

This condition is then introduced in the **EVENT End** and also in the invariant as explained in the pattern. The invariant becomes:

```
INVARIANTS
    $inv14 : \forall t, msg, A, B \cdot$
    $t \in trans \wedge A \in Agent \wedge B \in Agent \wedge msg \in channel\ \wedge$
    $t\_src(t) = A \wedge t\_bld\_dst(t) = B\ \ \wedge msg\_t(msg) = t$
    $msg\_Agt(msg) = t\_bld\_dst(t)$
    $\Rightarrow$
    $msg\_src(msg) = t\_bld\_dst(t)$
```

In the **EVENT Attk**, the attacker tries to mislead an agent A claiming himself to be agent B. A is the source of a transaction t. The attacker adds a new message msg to the $channel$ with $msg\_src(msg) = I$ and $msg\_Agt(msg) = B$. To keep the invariant, $Attk\_Cond(t)$ is defined as follows:

$$t \in trans \wedge t\_src(t) = A \wedge t\_bld\_dst(t) = B \Rightarrow I = B$$

5.3.4 Second refinement: attacker's knowledge

To be able to prove properties such as secrecy and authentication on a protocol, we have to be able to model the knowledge of the attacker. To model the knowledge of the attacker, it is necessary to know exactly what the attacker is able to do. One popular model for attacker's behavior is the Dolev-Yao model. Regardless of the chosen attacker's model we will use a variable Mem. This variable is the memory of all agents because the attacker himself may act like a honest agent. The memory of each agent is a set of transactions.

$$inv15 : Mem \in Agent \rightarrow \mathbb{P}(T)$$

We need to answer two issues :

- What is in the variable *Mem* ?
- How do the intruder uses the knowledge contained in this variable ?

The answer of the second issue is immediate, the *event Attk* is refined by changing the *guard 7*. Now the attacker uses only transaction that are in his memory. The *event Attk* become:

```
EVENT Attk
    ANY
        $t$
        $A$
        $msg$
    WHERE
        $\ominus$ $grd7 : t \in trans \wedge t\_src(t) = A \wedge t\_bld\_dst(t) = B$
                $\Rightarrow$ $I = B$
        $\oplus$ $grd7 : t \in Mem(I)$
    THEN
        $act1 : channel := channel \cup \{msg\}$
        $act2 : msg\_src(msg) := I$
        $act3 : msg\_dst(msg) := A$
        $act4 : msg\_t(msg) := t$
    END
```

And the following invariant is added:

INVARIANTS
$inv16 : \forall t.t \in Mem(I) \Rightarrow Attak\_Cond(t)$

To prove this invariant we need to answer the first issue: what is in the attacker memory ? This will depend from the chosen attacker model. In the Dolev-Yao one, attacker has full control of communication channel :

- He can intercept and remove any message of the channel.
- He can also generate infinite number of messages.
- He can decrypt parts of the message if he has the appropriate key.
- He can split unencrypted messages.

In our model we have already added event where messages are lost no matters if it is done by the attacker or not. And we didn't limit the number of messages the attacker can send. To model the fact that an attacker decrypts parts of the message, if he has the appropriate key, we add systematically the following substitution, when, in an event, any agent A sends to another agent B a message where a nonce is encrypted with agent B's key, in this substitution, the transaction t is added to agents A and B memory.

$$Mem := Mem \mathbin{\lhd\!\!-} \{A \mapsto Mem(A) \cup \{t\}, B \mapsto Mem(B) \cup \{t\}\}$$

And the following substitution when any agent A sends to another agent B a message, where a nonce is not encrypted:

$$Mem := Mem \lhd\!\!- \{a \mapsto b | a \in Agent \land b = Mem(a) \cup t\}$$

In the previous substitution, the transaction t is added to each agent's memory. To prove the invariant $invariant12$ we need an additional invariant that gives a characterization of the attacker's knowledge. This invariant is different from a protocol to another. In the case of the Needham Schroeder public key protocol where the pattern was applied two times (see figure 5.4) we had proven that the attackers's memory contains only transactions that are not in the set $trans$ or transactions where the attacker is the source or the trusted destination.

INVARIANTS

$inv17 : \forall t \cdot t \in Mem(I) \Rightarrow$
$($
$t \notin transA \cup transB \lor$
$(t \in transA \land (I = A\_t\_bld\_dst(t) \lor I = A\_t\_src(t))) \lor$
$(t \in transB \land (I = B\_t\_bld\_dst(t) \lor I = B\_t\_src(t)))$
$)$

5.3.5 Attacker's knowledge for proving secrecy

Until now, we used attacker's knowledge to prove authentication property. We may need to prove secrecy property on specific parts of the protocol. For example, in the Blake-Wilson-Menezes key transport protocol, we need to prove that the exchanged key remains a shared secret between the agent B that issued the key and the trusted destination of the transaction. We introduce a new basic set P_KEY and a new variable t_key that gives the secret key of each running transaction. The variable t_key is an injective function; two different transactions have different keys associated to them. This is for proving freshness property over session keys.

SETS
P_KEY

$$inv18 : t\_key \in trans \rightarrowtail P\_KEY$$

We introduce a new variable K_Mem. This variable gives for each agent the key he can obtain :

$$inv19 : K\_Mem \in Agent \rightarrow \mathbb{P}(P\_KEY)$$

As we did for the variable Mem, we add substitutions in all the events where an agent A sends the key to another agent B. Once again two cases have to be considered, when the key is encrypted or not. Finally we need to add an invariant that proves that a key is known only by the issuer of the key and the trusted destination. Unlike the $invariant17$ that may be different from a protocol to another, this invariant is common to all key transport protocols.

INVARIANTS

$inv20 : \forall t, K \cdot t \in trans \land K = t\_key(t) \land K \in K\_Mem(I) \Rightarrow$
$(I = t\_bld\_dst(t) \lor I = t\_src(t))$

The last refinement consists in replacing the abstract transactions by the nonces. This refinement is a simple data refinement, since for each session of the protocols a fresh nonce is created and a single source and single trusted destination are assoicated to this nonce. Accordingly, variables t_src, t_bld_dst are replaced by variables $nonce\_src$, $nonce\_bld\_dst$. The invariant $invariant12$ that states secrecy property for the abstract transactions holds for the nonces introduced in the last refinement.
We emphasize that in the Needham Schroeder public key protocol since secrecy property for the nonces has been proven, nonces can be used as shared secrets for key establishment. This is not true for the Blake-Wilson-Menezes key transport protocol.

5.4 Conclusion

We have introduced an event B based design pattern for key establishment protocols and we have applied it on two different protocols. Two properties were proved on these protocols, key authentication and key freshness. Less than 5% of the proofs of the models were interactive. As a perspective of our work, it is necessary to add new properties that are desired in some situation such as key confirmation where an agent receives an evidence that the other agent involved in the protocol run received the session key. It is also necessary to address questions on extensions of Dolev-Yao models.

Chapter 6

Conclusion and perspectives

Sommaire

6.1 Summary of the patterns

Four patterns have been presented:

- The *parametric/generic* pattern provides a way to instantiate Event B development and we apply it on two very different case studies: the design of greedy algorithms [27] and the design of e-voting algorithms [22, 21]. The technique is very close to the instantiation of discrete models due to Abrial and Hallestede [9] and we had worked partially with J.-R. Abrial on this technique. The main idea is to solve a general problem and then to understand that another problem is a specialized problem of the general problem. The refinement plays a central role to check that the new problem is a special case of the general problem up to instantiation. The implementation of the pattern is based on the possibility to instnatiate contexts and models.
- The *call as event* pattern is based on the relation between an event and the call of a procedure; the relation was first introduced in a paper [28] and has been formalised in [63]. Moreover, a plugin is under development and will assist the developer when he/she ants to apply it.
- The *access control* pattern is very close to the *parametric/generic* pattern but it relates security models and secure systems. The pattern was developed from case studies of a previous project namely DESIRS, supported by ANR; the pattern provides a way to separate security policy and secure system using the refinement.
- The *cryptographic* pattern is probably the most recent pattern; it provides a way to handle attack models like Dolev-Yao [38] and to take into account attack model when developing cryptographic protocols.

The four patterns have now yet reched the maturity necessary for developing a plugin for each pattern; it remains to study the possibilities provided by Bart which requires to define a list of rules for helping the developer. When the free version will be available, we will make experiments.

6.2 The BART experience

The BART (B Automatic Refinement Tool) project aims at developing an automatic refinement tool for B machines. This tool will allow for a B0 implementation for a machine or a sufficiently detailed B refinement to be automatically generated. BART operates on the basis of refinement rules. Additional refinement rules may be added in order to allow for the customization of the refinement of some components. The BART automatic refinement tool has been developed in the context of the RIMEL project and will be integrated into the next version of Atelier B. Specifications are currently being finalized.

6.3 Perspectives

In the two first delivrables, we have focused on case studies related to time-sensitive problems and to distributed algorithms; the deliverable 2 [10] introduces patterns for developing models of systems handling time constraints. These patterns can be added to the four patterns of this report. We have a basis of patterns that we should now try to implement. However, whereas the implementation seems to be feasible for the *call as event* pattern, for the *parametric/generic* pattern and for the *access control* pattern, the *cryptographic* pattern requires yet attention. It is also the case for developing distributed algorithms and further studies and case studies are necessary for delivering additional patterns. The formalisation of patterns is also a second possible direction and is clearly related to the implementation. The relationship with Bart and how Bart can be used for the implementation is another point to address.

Bibliography

[1] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[2] J.-R. Abrial. Etude systï¿½me: mï¿½thode et exemple. http://www.atelierb.societe.com/documents.html.

[3] J.-R. Abrial. *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] J.-R. Abrial. $B^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer, June 2003.

[5] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003*, pages 51–74, 2003.

[6] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOLs [12]*, pages 1–24, 2003.

[7] J.-R. Abrial, D. Cansell, and D. Méry. Formal derivation of spanning trees algorithms. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer Verlag, June 2003.

[8] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 761–768. ACM, 2006.

[9] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[10] Projet ANR-RIMEL. Intégration de contraintes temps-réel au sein d'un processus de développement incrémental basé sur la preuve. Livrable rimel, LORIA, Juillet 2008.

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

[12] D. A. Basin and B. Wolff, editors. *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003.

[13] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. MTR-2997, (ESD-TR-75-306), available as NTIS AD-A023 588, MITRE Corporation, 1976.

[14] K. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, 1975.

[15] Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.

[16] Simon Blake-Wilson and Alfred Menezes. Entity authentication and authenticated key transport protocols employing asymmetric techniques. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 137–158, London, UK, 1998. Springer-Verlag.

[17] Dominique Bolignano. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 77–87. Springer, 1998.

[18] Ian Brown. Who is enfranchised by remote voting? In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, page 500, Washington, DC, USA, 2005. IEEE Computer Society.

[19] James L. Caldwell. Formal methods technology-transfer: a view from NASA. In S. Gnesi and D. Latella, editors, *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems*, Oxford England, March 1996.

[20] D. Cansell, D. Méry, and S. Merz. Predicate diagrams
for the verification of reactive systems. In Bill Stoddart, editor, *IFM 2000 Conference*, SAARBRÜCKEN, August 2000. Springer.

[21] Dominique Cansell, J. Paul Gibson, and Dominique Méry. Refinement: A constructive approach to formal software design for a secure e-voting interface. *Electronic Notes in Theoretical Computer Science*, Volume 183:39–55, July 2007. Proceedings of the First International Workshop on Formal Methods for Interactive Systems (FMIS 2006).

[22] Dominique Cansell, Paul Gibson, and Dominique Méry. Formal verification of *tamper-evident* storage for e-voting. In *SEFM 2007 5th IEEE International Conference on Software Engineering and Formal Methods*, 2007.

[23] Dominique Cansell, J. Paul Gibson, and Dominique Méry. Refinement: a constructive approach to formal software design for a secure e-voting interface. In *Proceedings, International Workshop on Formal Methods for Interactive Systems (FMIS), Macao SAR China*, 2006.

[24] Dominique Cansell and Dominique Méry. Logical foundations of the B method. *Computers and Informatics*, 22, 2003.

[25] Dominique Cansell and Dominique Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [15].

[26] Dominique Cansell and Dominique Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [15].

[27] Dominique Cansell and Dominique Méry. Incremental parametric development of greedy algorithms. *Electr. Notes Theor. Comput. Sci.*, 185:47–62, 2007.

[28] Dominique Cansell and Dominique Méry. Proved-patterns-based development for structured programs. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *Lecture Notes in Computer Science*, pages 104–114. Springer, 2007.

[29] Dominique Cansell, Dominique Méry, and Stephan Merz. Diagrams Refinement for the Design of Reactive Systems. *Journal of Universal Computer Science*, 7(2):159–174, 2000.

[30] B. Charlier. The Greedy Algorithms Class: Formalization, Synthesis and Generalization. Lectures Notes, UCL, Belgium, September 1995.

[31] ClearSy. *Atelier B, Manuel Utilisateur*, 2001.

[32] ClearSy. *Web site B4free set of tools for development of B models*, 2004.

[33] ClearSy. Bart: B automatic refinement tool. Internal Document, Jannuary 2008.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[35] F. Cuppens. Orbac web page. http://www.orbac.org.

[36] S. A. Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49:125–157, 2003.

[37] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall International, 1976.

[38] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[39] L. Fejoz, D. Méry, and S. Merz. DIXIT an editor for predicate diagrams. In *AVIS'05*, 2005.

[40] D.F. Ferraiolo, R.Sandhu, S.Gavrila, D.R. Kuhn, and R.Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):222–274, 2001.

[41] L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.

[42] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[43] E. Gamma, R. Helm, R. Johnson, R. Vlissides, and P. Gamma. *Design Patterns : Elements of Reusable Object-Oriented Software design Patterns*. Addison-Wesley Professional Computing, 1997.

[44] Serban I. Gavrila and John F. Barkley. Formal specification for role based access control user/role and role/role relationship management. In *ACM Workshop on Role-Based Access Control*, pages 81–90, 1998.

[45] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[46] Dimitris Gritzalis, editor. *Secure Electronic Voting*, volume 7 of *Advances in Information Security*. Springer, 2003.

[47] Paul S. Herrnson, Benjamin B. Bederson, Bongshin Lee, Peter L. Francia, Robert M. Sherman, Frederick G. Conrad, Michael Traugott, and Richard G. Niemi. Early appraisals of electronic voting. *Soc. Sci. Comput. Rev.*, 23(3):274–292, 2005.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12:576–580, 1969.

[49] Mark Horst, Marg&#244;t Kuttschreuter, and Jan M. Gutteling. Perceived usefulness, personal experiences, risk perception and trust as determinants of adoption of e-government services in The Netherlands. *Comput. Hum. Behav.*, 23(4):1838–1852, 2007.

[50] J. Paul Gibson. Formal requirements engineering: Learning from the students. In *Australian Software Engineering Conference*, pages 171–180, 2000.

[51] Paul Kocher and Bruce Schneier. Insider risks in elections. *Commun. ACM*, 47(7):104, 2004.

[52] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–40. IEEE, 2004.

[53] L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, 1994.

[54] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[55] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[56] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAS*, pages 147–166, 1996.

[57] Margaret McGaley and J. Paul Gibson. E-Voting: A Safety Critical System. Technical Report NUIM-CS-TR-2003-02, NUI Maynooth, Comp. Sci. Dept., 2003.

[58] Margaret McGaley and Joe McCarthy. Transparency and e-Voting: Democratic vs. Commercial Interests. In *Electronic Voting in Europe - Technology, Law, Politics and Society*, pages 153 – 163. European Science Foundation, July 2004.

[59] P. Mehlitz and J. Penix. Design for verification - using design patterns to build reliable systems. In *Proceedings of 6th ICSE workshop on component based software engineering*, Portland, Oregan, May 2003.

[60] Rebecca T. Mercuri. Computer security: quality rather than quantity. *Commun. ACM*, 45(10):11–14, 2002.

[61] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage —or— how to store ballots on a voting machine. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 365–370, Washington, DC, USA, 2006. IEEE Computer Society.

[62] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[63] Dominique Méry. Teaching programming methodology using event b. In H. Habrias, editor, *The B Method: from Research to Teaching*, June 2008.

[64] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[65] Peter G. Neumann. Inside risks: risks in computerized elections. *Commun. ACM*, 33(11):170, 1990.

[66] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[67] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[68] G. Polya. *How to Solve It*. Princeton University Press, 2nd edition, 1957. ISBN 0-691-08097-6.

[69] Project RODIN. The rodin project: Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/.

[70] R.Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[71] William L. Scherlis and Jon Eisenberg. IT research, innovation, and e-government. *Commun. ACM*, 46(1):67–68, 2003.

[72] William Stallings. *Data and computer communications*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1985.

[73] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[74] Jörgen Svensson and Ronald Leenes. E-voting in europe: Divergent democratic practice. *Information Polity*, 8(1):3–15, 2003.

[75] Nicolaus Tideman and Daniel Richardson. Better voting methods through technology: The refinement-manageability trade-off in the single transferable vote. *Public Choice*, 103(1):13–34, April 2000.