Under consideration for publication in Formal Aspects of Computing

A mechanically proved and incremental development of IEEE 1394

Jean-Raymond Abrial[1], Dominique Cansell[2] and Dominique Méry[3]

[1]Marseille, France (jr@abrial.org)
[2]Université de Metz, LORIA, Metz, France (cansell@loria.fr)
[3]Université Henri Poincaré Nancy 1, LORIA, Vandœuvre-lès-Nancy, France (mery@loria.fr)

Abstract.
The IEEE 1394 tree identify protocol illustrates the adequacy of the *event-driven approach* used together with the B Method . This approach provides a complete framework for developing mathematical models of distributed algorithms. A specific development is made of a series of more and more refined models. Each model is made of a number of static properties (the invariant), and of a dynamic parts (the guarded events). The internal consistency of each model as well as its correctness with regards to its previous abstraction are proved with the proof engine of Atelier B, which is the tool associated with B. In the case of IEEE 1394 , the initial model is very primitive: it provides the basic properties of the graph (symmetry, acyclicity, connectivity), and its dynamic parts essentially contains a single event which *elects the leader* in one shot. Further refinements introduce more events, showing how each node of the graph non-deterministically participates to the leader election. At some stage in the development, message passing is introduced. This raises a specific potential contention problem, whose solution is given. The last stage of the refinement completely localize the events by making them taking decision based on local data only.

Keywords: B method, event-driven approach, refinement, proof-based development, proof engine, abstract model,

1. Introduction

Overview. Distributed systems are inherently complex to understand, to design and to verify. In order to master this complexity, people have developed various approach such as model-checking and theorem proving. In this paper, we illustrate the latter by applying it to the IEEE 1394 protocol [IEE95].

Correspondence and offprint requests to: Dominique Méry, Université Henri Poincaré Nancy 1, LORIA, BP239, 54506 Vandœuvre-lès-Nancy Cedex, France. e-mail: mery@loria.fr

Proof-based Development. Proof-based development methods 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. We then gradually add details to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [Bac79, Abr96a, CM88]. It is controlled by means of a number of, so-called, *proofs obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination properties. The invariant of an abstract model plays a central rôle for deriving safety properties and our methodology focuses on the incremental discovery of the invariant; the goal is to obtain a formal statement of properties through the final invariant of the last refined abstract model. When developing formal models for the IEEE 1394 protocol, we use the environment Atelier B [CLE01] for generating and proving proof obligations.

Understanding Distributed Systems. As already mentioned, a distributed system is *complex.* In this paper, the IEEE 1394 protocol is used to illustrate a method for understanding how a typical distributed system is working. Understanding a distributed system means that we are able to explain why it is working safely and how it meets its requirements. In the case of the IEEE 1394 protocol, the same piece of code is duplicated at each node of an acyclic and connected network. And the process that should be performed by these codes, each working concurrently but with a limited knowledge, is the *leader election*: in other words, at the end of the process a node should be given a special status, that of the leader, and other nodes should have a means to eventually communicate with it. It is, in fact, not clear at all that these distributed *local* computations indeed converge towards the leader election, which is a *global* result. The refinement technique we use allows us to decompose the IEEE 1394 system into four embedded models, each one providing an additional view by bringing more informations into the current invariant. For instance, the first two models contains the essence of the underlying structure of the acyclic and connected graph representing the network. They also convey the main ideas of the distributed computation. They express the way the protocol works at a very high level abstraction. The third model formalizes how the nodes communicate by means of various kinds of messages: it helps us understanding the contention problem, which is one of the critical question of the IEEE 1394 protocol. The last model deals with the *localization* of the abstract data structures used in the previous models.

Refining Formal Models. Formal models, as described in this paper, contain *events* which preserve some invariant properties; they also include aspects related to the termination. Such models are thus very close to action systems introduced by R.J. Back [Bac79] and to UNITY programs [CM88]. The refinement of formal models plays a central rôle in these frameworks and is a key concept for developing distributed systems. When one refines a formal model, the corresponding more concrete model may have new variables and new events, it may also strengthen the *guards* of more abstract events. As already mentioned, some proof obligations are generated in order to prove that a refinement is correct. Notice that, if some proof obligations remain unproved, it means that, either the formal model is not correctly refined, or that an interactive proving session is required. The prover allows us to get a complete proof of the development and hence of the final protocol. No assumption is made on the *size* of the system, for instance the number of nodes in the network. This contrasts with what should be done while using model-checking.

Related works. The IEEE 1394 protocol is a distributed algorithm for electing a leader in a network. The idea of the algorithm has already been sketched by N. Lynch [Lyn96](page 501). This sketch fits our second formal model. The PVS verification [DGRV00] derives the correctness of the IEEE 1394 protocol for an I/O automaton SPEC, which corresponds to our third formal model. We notice that this I/O automaton is not detailed enough to express the *confirmation* event, which appears in our third model. Their proofs are not really helpful for understanding the rôle of the underlying structure in the convergence of the algorithmic solution. The expressiveness of their invariant is not really clear. The PVS models includes an I/O automaton TIP that corresponds to our first formal model. A specific refinement relation is used to define the link between the two I/O automata, but it is not really useful to derive safety properties. Our approach keeps a link with the documentation and tends to explain in a formal way why the current abstract model is working correctly. We are really close to the IEEE 1394 protocol in our fourth abstract model. We shall not compare our approach to that of model checking, since our modeling is completely proved and is not restricted to a given network.

Organization of the paper. Section 2 introduces our proof-based development with the B event-driven approach. It introduces definitions for event, refinement and corresponding proof obligations. Section 3 analyses the election process and describes mathematical properties of the underlying structure, namely the acyclic and connected graph; a first formal model is designed and proved to meet the requirements of the leader election. The development process starts by refining the first model and introduces a progression event in section 4; this model is then refined and introduces the contention event. Section 5 localizes events and provides an algorithmic solution close to the IEEE 1394 description. Finally, we conclude our work in the section 6.

2. Proof-based development

2.1. Event-based modeling

Our event-driven approach [Abr96b, AM98] is based on the B notation [Abr96a]. It extends the methodological scope of basic concepts such as set-theoretical notations and generalized substitutions in order to take into account the idea of *formal models.* Roughly speaking, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states some properties that must always be satisfied by the variables x and *maintained* by the activation of the events. Abstract models are close to guarded commands of Dijkstra [Dij76], action systems of Back [Bac79] and to UNITY programs [CM88]. In what follows, we briefly recall definitions and principles of formal models and explain how they can be managed by Atelier B [CLE01].

Definition 1. : Generalized Substitution

Generalized substitutions are borrowed from the B notation. They provide a way to express the transformations of the values of the state variables of a formal model. In its simple form, $x := E(x)$, a generalized substitution looks like an assignment statement. In this construct, x denotes a vector build on the set of state variables of the model, and $E(x)$ a vector of expressions of the same size as the vector x. The interpretation we shall give here to this statement is *not* however that of an assignment statement. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general form of generalized substitution. It is denoted by the construct $x : P(x_0, x)$. This is to be read: "x is modified in such a way that the predicate $P(x_0, x)$ holds", where x denotes the *new value* of the vector, whereas x_0 denotes its *old value*. It is clearly non-deterministic in general. This general form could be considered as a *normal form*, since the simplest form $x := E(x)$ is equivalent to the more general form $x : (x = E(x_0))$.

Definition 2. : Events and Before-After Predicates

An event is essentially made of two parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event can take one of the forms shown in the table below. In these constructs, *evt* is an identifier: this is the event name. The first event is not guarded: it is thus always enabled. The guard of the other events, which states the necessary condition for these events to occur, is represented by $G(x)$ in the second case, and by $\exists\, t \cdot G(t, x)$ in the third one. The latter defines a non-deterministic event where t represents a vector of distinct local variables. The, so-called, before-after predicate $BA(x, x')$ associated with each event shape, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the event "execution".

Event	Before-after Predicate $BA(x, x')$
$evt \;\widehat{=}\;$ BEGIN $x : P(x_0, x)$ END	$P(x, x')$
$evt \;\widehat{=}\;$ SELECT $G(x)$ THEN $x : Q(x_0, x)$ END	$G(x) \wedge Q(x, x')$
$evt \;\widehat{=}\;$ ANY t WHERE $G(t, x)$ THEN $x : R(x_0, x, t)$ END	$\exists t \cdot (G(t, x) \wedge R(x, x', t))$

Proof obligations are produced from events in order to state that the invariant condition $I(x)$ is preserved. We next give the general rule to be proved. It follows immediately from the very definition of the before-after predicate, $BA(x, x')$ of each event:

$$\boxed{I(x) \wedge BA(x, x') \Rightarrow I(x')}$$

Notice that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. When it is the case, the event is said to be "disabled".

2.2. Model Refinement

The refinement of a formal model allows us to enrich a model in a *step by step* approach. Refinement provides a way to construct stronger invariants and also to add details in a model. It is also used to transform an abstract model in a more concrete version by modifying the state description. This is essentially done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event take control for ever, and (4) relative deadlockfreeness is preserved.

Definition 3. : Refinement

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BAA(x, x')$ and $BAC(y, y')$ are respectively the abstract and concrete before-after predicates of the same event, we have to prove the following statement:

$$\boxed{I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow \exists x' \cdot (BAA(x, x') \wedge J(x', y'))}$$

This says that under the abstract invariant $I(x)$ and the concrete one $J(x, y)$, a concrete step $BAC(y, y')$ can be simulated ($\exists x'$) by an abstract one $BAA(x, x')$ in such a way that the gluing invariant $J(x', y')$ is preserved. A new event with before-after predicate $BA(y, y')$ must refine *skip* ($x' = x$). This leads to the following statement to prove:

$$\boxed{I(x) \wedge J(x, y) \wedge BA(y, y') \Rightarrow J(x, y')}$$

Moreover, we must prove that a variant $V(y)$ is decreased by each new event (this is to guarantee that an abstract step may occur). We have thus to prove the following for each new event with before-after predicate $BA(y, y')$:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; BA(y,y') \;\Rightarrow\; V(y') < V(y)}$$

Finally, we must prove that the concrete model does not introduce more deadlocks than the abstract one. This is formalized by means of the following proof obligation:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; \mathsf{grds}(AM) \;\Rightarrow\; \mathsf{grds}(CM)}$$

where $\mathsf{grds}(AM)$ stands for the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ stands for the disjunction of the guards of the events of the concrete one.

3. The Case Study: Basic Approach

The goal of the IEEE 1394 protocol is to elect in a *finite time* a specific node, called the *leader*, in a network made of various nodes linked by some communication channels. Once the leader is elected, each non-leader node in the network should have a well defined way to communicate with it. This election of the leader has to be done in a distributed and non-deterministic way.

3.1. The Basic Mathematical Structure

Before considering details of the protocol, we choose to give a very solid definition to the main topology of the network. It is essentially formalized by means of a set ND of nodes subjected to the following assumptions:

1. the network is represented by a graph g built on ND,
2. all nodes are concerned with the network,
3. the links between the nodes are *bidirectional*,
4. a node is *not directly connected to itself*.

$$\boxed{\begin{array}{l} g \subseteq ND \times ND \\ \mathsf{dom}\,(g) = ND \\ g = g^{-1} \\ \mathsf{id}(ND) \;\cap\; g \;=\; \emptyset \end{array}}$$

Items 2 and 3 above are formally represented by a *symmetric graph* whose domain (and thus co-domain too) corresponds to the entire *finite set* of nodes. The symmetry of the graph is due to the representation of the non-oriented graph by pairs of nodes and the link $x - y$ is represented by the two pairs $x \mapsto y$ and $y \mapsto x$. Item 4 is rendered by saying that the graph is *not reflexive*.

There are two other very important properties of the graph: it is *connected and acyclic*. Both these properties are formalized by claiming that the relation between each node and the spanning trees of the graph having that node as a root, that this relation is *total* and *functional*. In other words, each node in the graph can be associated with one and exactly one tree rooted at that node and spanning the graph. We can model a tree by a root r, which is a node: $r \in ND$, and a father functions t (each node has an unique father node, except the root): $t \;\in\; ND - \{r\} \;\longrightarrow\; ND$. The tree is an acyclic graph. A cycle c in a finite graph t built on a set ND is a subset of ND whose elements are members of the inverse image of c under t, formally: $c \;\subseteq\; t^{-1}[c]$. To fulfil the requirement of acyclicity, the only set c that enjoys this property is thus the empty set. This can be formalized by the left predicate that follows, which can be proved to be *equivalent* to the one situated on the right, which can be used as an induction rule:

$$\boxed{\forall c \cdot (\, c \subseteq ND \;\wedge\; c \subseteq t^{-1}[c] \;\Rightarrow\; c = \emptyset\,)} \;\Leftrightarrow\; \boxed{\forall q \cdot (\, q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q\,)}$$

We prove the equivalence using the tool Atelier B. We can now define a spanning tree (with root r and father function t) of a graph g as one whose father function is included in g, formally:

$$\mathsf{spanning}\,(r,t,g) \;\hat{=}\; \left(\begin{array}{l} r \in ND \quad \wedge \\ t \in ND - \{r\} \longrightarrow ND \quad \wedge \\ \forall q \cdot (q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q) \quad \wedge \\ t \subseteq g \end{array}\right)$$

As mentioned above, each node in the graph can be associated with exactly one tree rooted at that node and which spans the graph. For this, we define the following total function f connecting each node r of the graph with its spanning tree $f(r)$:

$$f \in ND \rightarrow (ND \twoheadrightarrow ND)$$

$$\forall (r,t) \cdot \left(\begin{array}{l} r \in ND \;\wedge \\ t \in ND \twoheadrightarrow ND \\ \Rightarrow \\ t = f(r) \;\Leftrightarrow\; \mathsf{spanning}\,(r,t,g) \end{array}\right)$$

The graph g and the function f are thus *two global constants of the problem*.

3.2. The First Model

From the basic mathematical structure developed in previous section, the essence of the abstract algorithm implemented by the protocol is very simple: it consists in building gradually (and non-deterministically) *one of the spanning trees* of the graph. Once this is done, then the *root* of that tree is *the elected leader* and the communication structure between the other nodes and the leader is obviously the *spanning tree itself*. The protocol, considered globally, has thus *two variables*: (1) the future spanning tree, *sp*, and (2) the future leader, *ld*.

The *first formal model* of the development contains the definitions and properties of the two global constants (the above graph g and function f together with their properties), and the definition of the two mentioned global variables *sp* and *ld* typed in a very loose way: *sp* is a binary relation built on ND and *ld* is a node. The dynamic aspect of the protocol is essentially made of one *event*, called elect, which claims *what the result of the protocol is, when it is completed*. In other words, at this level, there is no protocol, just the formal definition of its intended result, namely a spanning tree *sp* and its root *ld*.

```
elect  ≙
  BEGIN
    $ld, sp : \mathsf{spanning}\,(ld, sp, g)$
  END
```

As can be seen, the election is done in one step. In other words, the spanning tree appears at once. The analogy of someone closing and opening eyes can be used here to “explain” the process of election at this very abstract level.

4. Refining the First Model

In this section, we present two successive refinements of the previous initial model. In the first one, we give the essence of the *distributed* algorithm. In the second refinement, we introduce some *communication mechanisms* between the nodes.

4.1. First Refinement: Gradual Construction of a Spanning Tree

In the first model, the construction of the spanning tree was performed in “one shot”. Of course, in a more realistic (concrete) formalization, this is not the case any more. In fact, the tree is constructed on a step by step basis. For this, a new variable, called tr, and a new event, called progress, are introduced. The variable tr represents a sub-graph of g, it is made of several trees (it is thus a *forest*) which will *gradually converge* to the final tree, which we intend to build eventually. This convergence is performed by the event progress. This event involves two nodes x and y, which are neighbours in the graph g. Moreover, x and y are supposed to be both outside the domain of tr. In other words, each of them has no "father" yet in tr. However, the node x is the father of all its *other neighbours* (if any) in g. This last condition can be formalized by means of the predicate $g[\{x\}] \;=\; tr^{-1}[\{x\}] \;\cup\; \{y\}$ since the set of neighbours of x in g is $g[\{x\}]$ while the set of sons of x in tr is $tr^{-1}[\{x\}]$. When these conditions are fulfiled, then the event progress can be enabled and its action has the effect of making the node y the father of x in tr. The abstract event elect is now refined. Its new version is concerned with a node x which happens to be the father of all its neighbours in g. This condition is formalized by the predicate $g[\{x\}] \;=\; tr^{-1}[\{x\}]$. When this condition is fulfiled the action of elect makes x the leader ld and tr the spanning tree sp. Next are the formal representations of these events

```
progress  ≙
  ANY $x, y$ WHERE
    $x, y \in g \;\wedge\; x \notin \mathsf{dom}(tr) \;\wedge\; y \notin \mathsf{dom}(tr) \;\wedge$
    $g[\{x\}] \;=\; tr^{-1}[\{x\}] \;\cup\; \{y\}$
  THEN
    $tr := tr \;\cup\; \{x \mapsto y\}$
  END
```

```
elect  ≙
  ANY $x$ WHERE
    $x \in ND \;\wedge$
    $g[\{x\}] \;=\; tr^{-1}[\{x\}]$
  THEN
    $ld, sp := x, tr$
  END
```

The new event progress clearly refines *skip* since it only updates the variable tr which is a *new variable* of this refinement with no existence in the abstraction. Also notice that progress clearly decreases the quantity $\mathsf{card}(g) - \mathsf{card}(tr)$. The situation is far less clear concerning the refinement of event elect. We have to prove that when its guard is true then tr is indeed a spanning tree of the graph g whose root is precisely x. Formally, this leads to proving the following

$$\boxed{\forall x \cdot (\, x \in ND \;\wedge\; g[\{x\}] = tr^{-1}[\{x\}] \;\;\Rightarrow\;\; \mathsf{spanning}\,(x, tr, g)\,)}$$

According to the definition of the constant function f, the previous property is clearly equivalent to

$$\boxed{\forall x \cdot (\, x \in ND \;\wedge\; g[\{x\}] = tr^{-1}[\{x\}] \;\;\Rightarrow\;\; tr = f(x)\,)}$$

This means that tr and $f(x)$ should have the same domain, namely $ND - \{x\}$, and that for all n in $ND - \{x\}$, $tr(n)$ is equal to $f(x)(n)$. This amounts to proving the following:

$$\boxed{ND \;=\; \{x\} \;\cup\; \{\, n \,|\, n \in ND - \{x\} \;\wedge\; f(x)(n) = tr(n)\,\}}$$

This is done using the *inductive property* associated with each spanning tree $f(x)$. Notice that we also need the following invariants:

$$\boxed{\begin{array}{l} tr \in ND \rightarrowtail\!\!\!\!\!+ \; ND \\ \mathsf{dom}\,(tr) \;\lhd\; (tr \;\cup\; tr^{-1}) \;=\; \mathsf{dom}\,(tr) \;\lhd\; g \\ tr \;\cap\; tr^{-1} \;=\; \emptyset \end{array}}$$

This new model, although more concrete than the previous one, is nevertheless still an abstraction of the "real" protocol: it just explains how the leader can be eventually elected by the gradual transformation of the forest tr into a unique tree spanning the graph g.

4.2. Second Refinement: Introducing Communication Channels

In the previous refinement, the event progress was still very abstract: as soon as two nodes x and y with the required properties were detected, the corresponding action took place immediately: in other words, y became the father of x "in one shot". In the "real" protocol things are not so "magic": once a node x has detected that it is the father of all its neighbours except one y, it sends a *request* to y in order to ask it to become its father. Node y then *acknowledges* this request and finally node x establishes the "father" connection with node y. This connection, which is thus established in *three distributed steps*, is clearly closer to what happens in the real protocol. We shall see however in the next refinement that what we have just described is not yet the final word. But let us formalized this for the moment. In order to do so, we need to define at least two new variables: *req*, to handle the requests, and *ack*, to handle the acknowledgements. *req* is a partial function from ND to itself. When a pair $x \mapsto y$ belongs to *req* it means that node x has send a request to node y asking it to become its father: the functionality of *req* is due to the fact that x has only one father. Clearly, *req* is also included in the graph g. When node y sends an acknowledgement to x this is because y has *already* received a request from x: *ack* is thus a partial function included in *req*.

$$\boxed{\begin{array}{l} req \in ND \rightarrowtail\!\!\!\!+ ND \\ req \subseteq g \\ ack \subseteq req \\ tr \subseteq ack \\ ack \cap ack^{-1} = \emptyset \end{array}}$$

Notice that when a pair $x \mapsto y$ belongs to *ack*, it means that y has sent an acknowledgment to x (clearly y can send several acknowledgements since it might be the father of several nodes). It is also clear that it is not possible in this case for the pair $y \mapsto x$ to belong to *ack*. The final connection between x and y is still represented by the function *tr*. Thus *tr* is included in *ack*. All this can be formalized as shown.

Two new events are defined in order to manage requests and acknowledgements: send_req, and send_ack. As we shall see, event progress is modified, whereas event elect is left unchanged. Here are the new events and the refined version of progress:

```
send_req  ≙
  ANY $x, y$ WHERE
    $x, y \in g \;\wedge\; y, x \notin ack \;\wedge$
    $x \notin \mathrm{dom}(req) \;\wedge$
    $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
  THEN
    $req := req \cup \{x \mapsto y\}$
  END
```

```
send_ack  ≙
  ANY $x, y$ WHERE
    $x, y \in req \;\wedge$
    $x, y \notin ack \;\wedge$
    $y \notin \mathrm{dom}(req)$
  THEN
    $ack := ack \cup \{x \mapsto y\}$
  END
```

```
progress  ≙
  ANY $x, y$ WHERE
    $x, y \in ack \;\wedge$
    $x \notin \mathrm{dom}(tr)$
  THEN
    $tr := tr \cup \{x \mapsto y\}$
  END
```

Event send_req is enabled when a node x discovers that it is the father of all its neighbours except one y: $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$. Notice that, as expected, this condition is exactly the one that allowed event progress in the previous model to be enabled. Moreover x must not have sent already a request to any node: $x \notin \mathrm{dom}(req)$. Finally x must not have already sent an acknowledgement to node y: $y, x \notin ack$. When these conditions are fulfiled then the pair $x \mapsto y$ is added to *req*. Event send_ack is enabled when a node y receives a request from node x, moreover y must not have already sent an acknowledgement to node x: $x, y \in req$ and $x, y \notin ack$. Finally node y must not have sent a request to any node: $y \notin \mathrm{dom}(req)$ (we shall see very soon what happens when this condition does not hold). When these conditions are fulfiled, node y sends an acknowledgement to node x: the pair $x \mapsto y$ is thus added to *ack*. Event progress is enabled when a node x receives an acknowledgement from node y: $x, y \in ack$. Moreover node x has not yet established any father connection: $x \notin \mathrm{dom}(tr)$. When these conditions are fulfiled the connection is established: the pair $x \mapsto y$ is added to *tr*.

Events send_req and send_ack clearly refine *skip*. Moreover their actions increment the cardinal of *req* and *ack* respectively (these cardinals are bounded by that d g). It remains for us to prove that the new version of event progress is a correct refinement of its abstraction. The actions being the same, it just remains for us to prove that the concrete guard implies the abstract one. This amounts to proving the following left predicate, which is added as an invariant:

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y \in ack \quad \wedge \\ x \notin \mathsf{dom}\,(tr) \\ \Rightarrow \\ x,y \in g \quad \wedge \\ x \notin \mathsf{dom}(tr) \quad \wedge \\ y \notin \mathsf{dom}(tr) \quad \wedge \\ g[\{x\}] \;=\; tr^{-1}[\{x\}] \;\cup\; \{y\} \end{array}\right)$$

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y \in req \quad \wedge \\ x,y \notin ack \\ \Rightarrow \\ x,y \in g \quad \wedge \\ x \notin \mathsf{dom}(tr) \quad \wedge \\ y \notin \mathsf{dom}(tr) \quad \wedge \\ g[\{x\}] \;=\; tr^{-1}[\{x\}] \;\cup\; \{y\} \end{array}\right)$$

When trying to prove that the left predicate is maintained by event send_ack, we find that the right predicate above must also be proved. It is thus added as a new invariant, which is, this time, easily proved to be maintained by all events.

The problem of contention. The guard of the event send_ack above contains the condition $y \notin \mathsf{dom}\,(req)$. If this condition does not hold while the other two guarding conditions hold, that is $x,y \in req$ and $x,y \notin ack$ hold, then clearly x has sent a request to y and y has sent a request to x: each one of them wants the other to be its father! This problem is called the *contention* problem. In this case, no acknowledgements should be sent since then each node x and y would be the father of the other. In the "real" protocol the problem is "solved" by means of timers. As soon as a node y discovers a contention with node x, it waits for very a short delay in order to be certain that the other node x has also discovered the problem. The very short delay in question is at least equal to the message transfer time between nodes (such a time is supposed to be *bounded*). After this, each node randomly chooses (with probability 1/2) to wait for either a "short" or a "large" delay (the difference between the two is at least twice the message transfer time). After the chosen delay has passed each node sends a new request to the other *if it is in the situation to do so.* Clearly, if both nodes choose the same delay, the contention situation will reappear. However if they do not choose the same delay, then the one with the largest delay becomes the father of the other: when it wakes up, it discovers the request from the other while it has not itself already sent its own request, it can therefore send an acknowledgement and thus become the father. According to the *law of large numbers*, the probability for both nodes to indefinitely choose the same delay is null. Thus, at some point, they will (in probability) choose different delays and one of them will thus become the father of the other. We shall only present here a partial formalization of the contention problem. The idea is to introduce a *virtual channel* called *cnt*.

$$\begin{array}{l} cnt \;\subseteq\; req \\ ack \,\cap\, cnt \;=\; \emptyset \end{array}$$

When this "channel" contains a pair $x \mapsto y$, this means that y has discovered the contention with node x. When both pairs $x \mapsto y$ and $y \mapsto x$ are present in *cnt*, this means that both nodes x and y have discovered the contention. Notice that *cnt* is included in *req* and clearly disjoint with *ack*, as shown. We have two new events. The first one is called discover_cnt. The only difference with the guard of event send_ack concerns the condition $y \in \mathsf{dom}\,(req)$, which is true in discover_cnt and false in send_ack. The action of this event adds the pair $x \mapsto y$ to *cnt*. The second new event is called solve_cnt. It is enabled when both pairs $x \mapsto y$ and $y \mapsto x$ are present in *cnt*. The action removes these pairs from *req* and resets *cnt*. This formalizes what happens after the "very short delay". Notice that this event is not part of the protocol: it corresponds to a "deamon" acting when the very short delay has just passed. Here are the events

```
discover_cnt  ≙
  ANY  $x,y$  WHERE
    $x,y \in req - ack \quad \wedge$
    $y \in \mathsf{dom}\,(req)$
  THEN
    $cnt := cnt \;\cup\; \{x \mapsto y\}$
  END
```

```
solve_cnt  ≙
  ANY  $x,y$  WHERE
    $x,y \in cnt \quad \wedge$
    $y,x \in cnt$
  THEN
    $req, cnt := req - cnt, \emptyset$
  END
```

In order to prove the invariant $ack \cap cnt = \emptyset$, we need the following extra invariants

$$\forall (x,y) \cdot \begin{pmatrix} x,y \in req - ack \quad \wedge \\ y \in \mathsf{dom}\,(req) \\ \Rightarrow \\ y,x \in req - ack \end{pmatrix} \qquad \forall (x,y) \cdot \begin{pmatrix} x,y \in req - ack \quad \wedge \\ y \notin \mathsf{dom}\,(req) \\ \Rightarrow \\ x,y \notin cnt \end{pmatrix}$$

The complete formalization of the contention solution of the real IEEE 1394 protocol (involving the timers and the random choices) is not difficult, just a little too long to be presented within the framework of this paper.

5. Last Refinement: Localization

In the previous refinement, the guards of the various events were defined in terms of some *global* constants or variables such as g, tr, req, ack. A closer look at this refinement shows that these constants or variables are used in expressions of the following shapes: $g^{-1}[\{x\}]$, $tr^{-1}[\{x\}]$, $ack^{-1}[\{x\}]$, $\mathsf{dom}\,(req)$, and $\mathsf{dom}\,(tr)$. These shapes dictate the kind of *data refinement* we now undertake. We declare five new variables nb (for neighbours), ch (for children), ac (for acknowledged), dr (for domain of req), and dt (for domain of tr). Next are the declarations of these variables together with their simple definitions in terms of the global variables.

$$\begin{array}{l} nb \in ND \rightarrow \mathbb{P}(ND) \\ ch \in ND \rightarrow \mathbb{P}(ND) \\ ac \in ND \rightarrow \mathbb{P}(ND) \\ dr \subseteq ND \\ dt \subseteq ND \end{array} \qquad \begin{array}{l} \forall x \cdot (\, x \in ND \Rightarrow nb(x) = g^{-1}[\{x\}]\,) \\ \forall x \cdot (\, x \in ND \Rightarrow ch(x) \subseteq tr^{-1}[\{x\}]\,) \\ \forall x \cdot (\, x \in ND \Rightarrow ac(x) = ack^{-1}[\{x\}]\,) \\ dr = \mathsf{dom}\,(req) \\ dt = \mathsf{dom}\,(tr) \end{array}$$

Given a node x, the sets $nb(x)$, $ch(x)$, and $ac(x)$ are supposed to be "stored" locally within the node. As the varying sets $ch(x)$ and $ac(x)$ are subsets of the constant set $nb(x)$, it is certainly possible to further refine their encoding. Likewise the two sets dr and dt still appears to be global, but they can clearly be encoded locally in each node by means of local boolean variables.

It is worth noticing that the "definition" of variable ch above is not given in terms of an equality, rather in terms of an inclusion (this is thus not really a definition). This is due to the fact that the set $ch(y)$ cannot be updated while the event progress takes place: this is because this event can only act on its *local* data. A new event, receive_cnf (for receive confirmation) is thus necessary to update the set $ch(y)$. Next are the refinement of the various events.

```
elect ≙
  ANY $x$ WHERE
    $x \in ND \wedge$
    $nb(x) = ch(x)$
  THEN
    $ld := x$
  END
```

```
send_req ≙
  ANY $x, y$ WHERE
    $x \in ND - dr \wedge$
    $y \in ND - ac(x) \wedge$
    $nb(x) = ch(x) \cup \{y\}$
  THEN
    $req := req \cup \{x \mapsto y\} \;||$
    $dr := dr \cup \{x\}$
  END
```

```
send_ack ≙
  ANY $x, y$ WHERE
    $x, y \in req \quad \wedge$
    $x \notin ac(y) \quad \wedge$
    $y \notin dr$
  THEN
    $ack := ack \cup \{x \mapsto y\} \;||$
    $ac(y) := ac(y) \cup \{x\}$
  END
```

```
progress ≙
  ANY $x, y$ WHERE
    $x, y \in ack$   $\wedge$
    $x \notin bt$   THEN
    $tr := tr \cup \{x \mapsto y\}$ ||
    $dt := dt \cup \{x\}$
  END
```

```
receive_cnf ≙
  ANY $x, y$ WHERE
    $x, y \in tr$ $\wedge$
    $x \notin ch(y)$
  THEN
    $ch(y) := ch(y) \cup \{x\}$
  END
```

The proofs that these events correctly refine their respective abstractions are technically trivial. We now give in the following table, the *local node* "in charge" of each event as encoded above

event	*node*
elect	x
send_req	x
send_ack	y
progress	x
receive_cnf	y

The reader could be surprised to still see formulas such as $req := req \cup \{x \mapsto y\}$ or $x, y \in req$. They correspond in fact to writing and reading operations done by corresponding local nodes as explained in the following table:

formula	*explanation*
$req := req \cup \{x \mapsto y\}$	x sends a request to y
$x, y \in req$	y reads a request from x
$ack := ack \cup \{x \mapsto y\}$	y sends an acknowledgement to x
$x, y \in ack$	x reads an acknowledgement from y
$tr := tr \cup \{x \mapsto y\}$	x sends a confirmation to y
$x, y \in tr$	y reads a confirmation from y

6. Concluding Remarks

The total number of proofs (all done mechanically with Atelier B) amounts to 106, where 24 required an easy interaction. Proofs help us to understand the contention problem and the rôle of graph properties in the correctness of the solution. The refinements gradually introduce the various invariants of the system. No assumption is made on the size of the network. The proof leads us to the discovery of the confirmation event to get the complete correctness, which was not the case of the I/O automata modelling.

In our opinion, this text, whose notation is very close to that of classical mathematics, is very simple to understand (provided, of course, the corresponding mathematical concepts, namely sets, functions, relations, and the like are well mastered), with the exception of our formulation of tree structures described under the form of the father function together with a universal quantification formalizing the corresponding induction rule. This formulation requires some more mathematical background. The question concerning the mythical *average programmer* understanding our solution is a bit irrelevant here: this problem is first, we believe, an abstract algorithm problem requiring a certain background in discrete mathematics. The lack of such background may lead to very awkward solutions due to the fact that they precisely try to convince the famous *average programmer*. In fact, in these solutions, the mathematical essence of the problem is hidden behind a curtain of technicalities all presented in a flat manner (no abstraction, thus no refinement, hence proof obligation explosion).

The essence of our approach is the methodology of *separation of concerns*: first prove the algorithm at an abstract (mathematical) level, then, and only then, gradually introduce the peculiarity of the specific protocol. What is important about our approach is that the fundamental properties we have proved at the beginning, namely the reachability and the uniqueness of a solution, are kept through the refinement process (provided, of course, the required proofs are done). It seems to us that this sort of approach is highly ignored in the literature of protocol developments where, most of the time, things are presented in a flat manner directly at the level of the final protocol itself.

References

[Abr96a] J.-R. Abrial. *The B book - Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abr96b] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190, November 1996.

[ACM01] J.-R. Abrial, D. Cansell, and D. Méry. The Complete B Project for the Development of the IEEE 1394 Tree Identify Protocol . http://www.loria.fr/ mery/ieee1394, March 2001.

[AM98] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 :Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[Bac79] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation.* Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[CLE01] ClearSy, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 2001. Version 3.6.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DGRV00] M. Devillers, D. Griffioen, J. Romin, and F. Vaandrager. Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394. *Formal Methods in System Design*, 16:307–320, 2000. Kluwer Academic Publishers.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1), January 1983.

[IEE95] IEEE. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, August 1995.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.