Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 363 (2006) 318–337
Theoretical Computer Science
www.elsevier.com/locate/tcs

Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm

Dominique Cansell[a], Dominique Méry[b,*]

[a] *Université de Metz, 57045 Metz, France*
[b] *Université Henri Poincaré, LORIA, BP239, 54506 Vandœuvre-lès Nancy, France*

Abstract

The development of distributed algorithms and, more generally, distributed systems, is a complex, delicate and challenging process. Refinement techniques of (system) models improve the process by using a proof assistant, and by applying a design methodology aimed at starting from the most abstract model and leading, in an incremental way, to the most concrete model, for producing a distributed solution. We show, using the distributed reference counting (DRC) problem as our study, how models can be produced in an elegant and progressive way, thanks to the refinement and how the final distributed algorithm is built starting from these models. The development is carried out within the framework of the event B method and models are validated with a proof assistant.

Keywords: Specification; Modelling; Refinement; Abstraction; Distributed algorithms; Verification; Proof; Proof assistant

1. Introduction

Overview Developing distributed algorithms can be made simpler and safer by the use of refinement techniques. Refinement allows one to gradually develop a distributed algorithm step by step, and to tackle complex problems like the PCI Transaction Ordering Problem [9] or the IEEE 1394 Tree Identification Protocol [6]. The event B method [1] provides a framework integrating refinement for deriving models solving distributed problems, and the main contribution of this paper is the development of a distributed algorithm for the distributed reference counting problem. It is a general purpose technique, which may be used, for instance, to detect termination of distributed programs or to implement distributed garbage collection. Moreau and Duprat [17] present a distributed reference counting algorithm and a mechanical proof of correctness carried out using the proof assistant Coq, but they do not use any refinement technique, which would help to construct the required (inductive) invariant implying the safety property *if there exists a reference to a resource, then its reference counter will be strictly positive*. A second contribution relies on the analysis of a liveness property through the refinement; the event B refinement states explicit proof obligations ensuring that no new concrete event can take control over abstract ones forever; we analyse our models in order to reason about the liveness property holding under very strong assumptions over the environment.

* Corresponding author.
E-mail addresses: cansell@loria.fr (D. Cansell), mery@loria.fr (D. Méry)
URLs: http://www.loria.fr/cansell (D. Cansell), http://www.loria.fr/mery (D. Méry).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.08.015

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 detail 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* [8,2,12]. It is controlled by means of a number of, so-called, *proof obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine [13,4]. 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 role 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 our case studies, we use the environments B4free [13] and Click'n'Prove [3] for generating and proving proof obligations.

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 Back [8] and to UNITY programs [12]. The refinement of formal models plays a central role in these frameworks and is a key concept for developing algorithmic 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, 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.

The DRC algorithm: We present a complete and proved development of a distributed reference counting algorithm. This algorithm is used to share and remove resources in a distributed way and also in distributed garbage collection. A resource is created by a site (the owner) and can be used by other sites. The owner of a resource can remove it only when it is sure that this resource is not used by another site. Moreau and Duprat have developed such an algorithm and proved it using the proof assistant Coq. The first contribution of this paper is the incremental construction starting from a simple abstraction without any distribution through to a distributed and concrete algorithm which looks like that of Moreau and Duprat. This incremental construction allows us to validate step by step the real algorithm and to prove its correctness more easily. We conclude the refinement process with an explanation on the termination of this algorithm using specific constraints (limited messages like in [17]). The incrementation is a factor which increases the understanding of the algorithmic method.

Summary: Section 2 introduces the event B method (set-theoretical notations, model ling language, refinement). Section 3 contains the complete development of the distributed algorithm; it contains event B models validated by the proof assistant. Section 4 concludes the development by analysing the number of automatic and interactive proof obligations that were required.

2. Modelling (distributed) systems

The systems under consideration for our technique are general software systems, control systems, protocols, sequential and distributed algorithms, operating systems and circuits; these are generally very complex and have parts interacting with an environment. A discrete abstraction of such systems constitutes an adequate framework: such an abstraction is called a *discrete model*. A discrete model is more generally known as a *discrete transition system* and provides a view of the current system; the development of a model in B follows an incremental process validated by refinement. A system is modelled by a sequence of models related by the refinement and managed in a project. We limit the scope of our work to distributed algorithms modelled under the *local computation rule* [11] in graphs and we specialize the proof obligations with respect to the target of the development which is a distributed algorithm fitting safety and liveness requirements.

2.1. The event B modelling method

An event has a guard and is triggered in a state validating the guard. A state is characterized by state variables, which are allowing to observe the current state; x is the name for state variables and we use x standing for the current value of x and x' standing for the next values of x. Each event has a name, but an event has no input and output parameters. An event is observed or not observed; possible changes of variables should maintain the invariant of the current model:

Event : e	Before-After Predicate : $BA(e)\ (x, x')$
BEGIN $x : P(x_0, x)$ END	$P(x, x')$
WHEN $G(x)$ THEN $x : P(x_0, x)$ END	$G(x) \wedge P(x, x')$
ANY t WHERE $G(t, x)$ THEN $x : P(x_0, x, t)$ END	$\exists t \cdot (G(t, x) \wedge P(x, x', t))$

Fig. 1. Definition of events and before–after predicates of events.

Event : e	Guard: grd(e)
BEGIN S END	*TRUE*
WHEN $G(x)$ THEN T END	$G(x)$
ANY t WHERE $G(t, x)$ THEN T END	$\exists t \cdot G(t, x)$

Fig. 2. Definition of events and guards of events.

the specification/modelling style is called *defensive*. An event is characterized by a relation over state variables x and primed state variables x': a before–after predicate denoted $BA(e)(x, x')$. An event is essentially a reactive object and reacts with respect to its guard $\mathrm{grd}(e)(x)$. However, there is a restriction over the language used for defining events and we authorize only three kinds of events (see Fig. 1). In the definition of an event, three basic substitutions are used to write an event $(x := E(x),\ x :\in S(x),\ x : P(x_0, x))$ and the last substitution is the normal form of the three ones; $E(x)$ is an expression over the set-theoretical language, $S(x)$ is an expression defining a set parametrised by x and $P(x_0, x)$ denotes a relation between the value of x, when evaluating the relation and the value of x after the evaluation ($x := x+1$ is written as $x : (x = x_0 + 1)$). The notation x_0 is inherited from the classical B method and it continues to be useful with certain tools which accept it. An event should be *feasible*: some next state must be reachable from a given state; it may happen that a current context does not imply the existence of a required value in a given set and the feasibility condition is intended to express this kind of situation. Since events are reactive objects, related proof obligations should guarantee that the current state satisfying the invariant should be feasible. Fig. 2 contains the definition of guards of events.

When using the relational style for defining the semantics of events, we use the style advocated by Lamport [15] in TLA; an event is seen as a transformation between states before the transformation and states after the transformation. Lamport uses the priming of variables to separate before values from after values. Using this notation and supposing that x_0 denotes the value of x before the transition of the event, events are given a semantics defined over primed and unprimed variables as shown in Fig. 1.

Any event e has a guard defining the enabledness condition over the current state and it expresses the existence of a next state. For instance, the disjunction of all guards is used for strengthening the invariant of a B system of events to include the deadlock freedom of the current model. Before introducing B models, we give the expression stating the preservation of a property by a given event e:

$$I(x) \ \wedge \ BA(e)(x, x') \ \Rightarrow \ I(x'). \tag{1}$$

$BA(e)(x, x')$ is the before–after relation of the event e and $I(x)$ is a state predicate over variables x. Eq. (1) defines the proof obligation for the preservation of $I(x)$, while e is observed. Since the two approaches are semantically equivalent, the proof obligations generator of the Atelier B can be reused for generating those assertions in the B environment. In the next section, we detail abstract models, which are using events.

2.2. Modelling distributed systems in the event B approach

Reactive systems react to their environment with respect to external stimuli; abstract models of the event B approach integrate the reactivity to stimuli by promoting events. At most one event is observed at any time of the system. A (abstract) model is made up of a part defining mathematical structures related to the problem to solve, and a part containing behavioural elements such as state variables, transitions and (safety and invariance) properties of the model. Proof obligations are generated from the model and, if they are holding, the model is said to be *internally consistent*. A model is assumed to be closed and it means that every possible change over state variables is defined by transitions; transitions correspond to events observed by the specifier. A model m is defined by the following structure:

MODEL
m
SETS
s
CONSTANTS
c
PROPERTIES
$P(s, c)$
VARIABLES
x
INVARIANT
$I(x)$
SAFETY
$A(x)$
INITIALIZATION
⟨substitution⟩
EVENTS
⟨list of events⟩
END

A model has a name m; the clause SETS contains definitions of sets of the problem; the clause CONSTANTS allows one to introduce information related to the mathematical structure of the problem to solve and the clause PROPERTIES contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. It should be noted that sets and constants can be considered like parameters, and extensions of the B method exploit this aspect to introduce parameterization techniques in the development process of B models. The second part of the model defines dynamic aspects of state variables and properties over variables using the invariant—generally called the inductive invariant—and using assertions generally called safety properties. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions and which is preserved by events (or transitions) of the list of events. Conditions of verification, called proof obligations, are generated from the text of the model using the first part for defining the mathematical theory; and the second part is used to generate proof obligations for the preservation of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant. The predicate $A(x)$ states properties derivable from the model invariant. A model specifies that state variables are always in a given set of possible values defined by the invariant and it contains the only possible transitions operating over state variables.

A model is not a program and no control flow is related to it; however, it does require a validation (but we first must define the mathematics for stating sets, properties over sets, invariants and safety properties). Conditions of consistency of the model are called *proof obligations* and they express the preservation of invariant properties and avoidance of deadlock. (INV1) $Init(x) \Rightarrow I(x)$ and (INV2) $I(x) \wedge BA(e)(x, x') \Rightarrow I(x')$, where $e_1, \ldots, e_n$ is the list of events of the model m. (INV1) states that the initial condition establishes the invariant. (INV2) should be checked for every event e of the model, where $BA(e)(x, x')$ is the before–after predicate of e. Predicates $A(x)$ in the clause **safety** should be implied by the predicates of the clause **invariant** under assumptions summarized by $P(s, c)$; assumptions $P(s, c)$ are defined from the parts **sets**, **constants** and **properties** of the model and they are used by the proof assistant to generate the underlying mathematical model; the condition is simply formalized as follows: $P(s, c) \wedge I(x) \Rightarrow A(x)$.

Finally, the substitution of an event must be feasible; an event is feasible with respect to its guard and the invariant $I(x)$ if there is always a possible transition of this event or equivalently, there exists a next value x' satisfying the before–after predicate of the event. The feasibility of the initialization event requires that at least one value exists for the predicate defining the initial conditions. The feasibility of an event leads to a readability of the form of the event; the recognition of the guard in the text of the event simplifies the semantical reading of the event and it simplifies the translation process of the tool: no guard is hidden inside the event.

Proof obligations for a model are generated by the proof-obligations generator of the B environment; the sequent calculus is used to state the validity of the proof obligations in the current mathematical environment defined by constants and properties. Several proof techniques are available but the proof tool is not able to prove automatically every proof obligation and interaction with the prover should lead to the proof of every generated proof obligation. We say that the model is *internally consistent* when every proof obligation is proved. A model uses only three kinds of events; but the objectives are to provide a simple and powerful framework for modelling reactive systems. A B model expresses invariant and safety properties satisfied by state variables; the state of the current modelled system

is observed through the current values of the state variables. An underlying global fairness assumption guarantees that at east one enabled event is observed at any time and a trace semantics can be associated to each B model. The correspondence between trace semantics and wp semantics is very classical and the seminal work of Park [18] led to a wp characterization of weak and strong fairness assumptions; further results [16,14] provide elements on the link between wp semantics and trace semantics. Abrial and Mussat [7] explain the semantical foundations of the event B modelling language and relate them to the wp semantics.

Since the consistency of a model is defined, we should introduce the refinement of models using the refinement of events defined as the substitution refinement.

2.3. Refinement of event B models

The refinement of a formal model allows one to enrich a model in a *step-by-step* approach. Refinement provides a way to construct stronger invariants and also to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is 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* $J(x, y)$. The gluing invariant ensures the consistency between abstract and concrete variables; the concrete events should simulate the abstract model. A number of proof obligations ensure that (0) the initial concrete conditions should establish the initial abstract conditions and the gluing invariant (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved. A new event (refining *skip*) may take control over other (concrete) events, because it may simulate the stuttering at the abstract level and it may also contradict the relative progress property at the abstract level. A solution is to control the execution of the concrete event by a natural concrete variable called V. Another solution would be to impose additional fairness assumptions. We detail proof obligations of a refinement after the introduction of its syntax of a refinement.

REFINEMENT
  r
REFINES
  m
SETS
  t
CONSTANTS
  d
PROPERTIES
  $Q(t, d)$
VARIABLES
  y
INVARIANT
  $J(x, y)$
VARIANT
  $V(y)$
SAFETY
  $B(y)$
INITIALIZATION
  $y : INIT(y)$
EVENTS
  ⟨list of events⟩
END

A *refinement* has a name r; it is a model refining a model m in the clause **refines** and m can itself be a refinement of another model. New sets, new constants and new properties can be declared in the clauses **sets**, **constants** or **properties**. New variables y are declared in the clause **variables** and are intended to be the new concrete variables; variables x of the refined model m are called the abstract variables. The gluing invariant defines a mapping between abstract variables and concrete ones; when a concrete event occurs, there must be a corresponding one in the abstract model: the concrete model *simulates* the abstract model. The clause **variant** *controls* new events, which cannot take the control over other events of the system. In a refinement, new events may appear and are refining an event SKIP; events of the refined model can be strengthened and one should prove that the new model does not contain more deadlock configurations than the refined one: if a guard is strengthened too much it can lead to a dead refined event. The refinement r of a model m is a system; its trace semantics is based on traces of states over variables x and y and the projection of concrete traces on abstract traces is a stuttering-free traces semantics of the abstract model. The mapping between abstract and concrete traces is called a refinement mapping by Lamport [15] and the stuttering is the key concept for refining event systems. When an event e of m is triggered, it modifies variables y and the abstract event refining e modifies x. Proof obligations make precise the relationship between abstract model and concrete model.

The abstract system is m and the more concrete system is r; $INIT(y)$ denotes the initial condition of the concrete model; $I(x)$ is the invariant of the refined model m; $BAC(e)(y, y')$ is the concrete before–after relation of an event e

of the concrete system r and $BAA(e)(x, x')$ is the abstract before–after relation of the event e of the abstract system m; $G_1(x), \ldots G_n(x)$ are the guards of the n abstract events of m; $H_1(y), \ldots, H_k(y)$ are the guards of the k concrete events of r. Formally, the refinement of a model is defined as follows:

- (REF1) $INIT(y) \Rightarrow \exists x \cdot (Init(x) \wedge J(x, y))$: The initial condition of the refinement model implies that there exists an abstract value in the abstract model such that this value satisfies the initial conditions of the abstract model and implies the new invariant of the refinement model.
- (REF2) $I(x) \wedge J(x, y) \wedge BAC(e)(y, y') \Rightarrow \exists x'.(BAA(e)(x, x') \wedge J(x', y'))$: The invariant in the refinement model is preserved by the refined event and the activation of the refined event triggers the corresponding abstract event.
- (REF3) $I(x) \wedge J(x, y) \wedge BAC(e)(y, y') \Rightarrow J(x, y')$: The invariant in the refinement model is preserved by the refined event but the event of the refinement model is a new event which was not visible in the abstract model; the new event refines *skip*.
- (REF4) $I(x) \wedge J(x, y) \wedge (G_1(x) \vee \cdots \vee G_n(x)) \Rightarrow H_1(y) \vee \cdots \vee H_k(y)$: The guards of events in the refinement model are strengthened and we have to prove that the refinement model is not more blocked than the abstract model.
- (REF5) $I(x) \wedge J(x, y)) \Rightarrow V(y) \in \mathbb{N}$.
- (REF6) $I(x) \wedge J(x, y) \wedge BAC(e)(y, y') \Rightarrow V(y') < V(y)$: New events should not block abstract events forever.

The refinement of models by refining events is close to the refinement of action systems [8], the refinement of UNITY and the TLA refinement; even if there is no explicit semantics based on traces, one can consider the refinement of events like a relation between abstract traces and concrete traces. The stuttering plays a central role in the global process of development where new events can be added into the refinement model. When one refines a model, one can either refine an existing event by strengthening the guard and/or the before–after predicate (removing non-determinism), or add a new event which is supposed to refine the skip event. When one refines a model by another model, it means that the set of traces of the refined model contains the traces of the resulting model with respect to the stuttering relationship. Models and refined models are defined and can be validated through the proofs of proof obligations; the refinement supports the proof-based development.

3. The distributed reference counting (DRC) problem

3.1. Statement of the problem

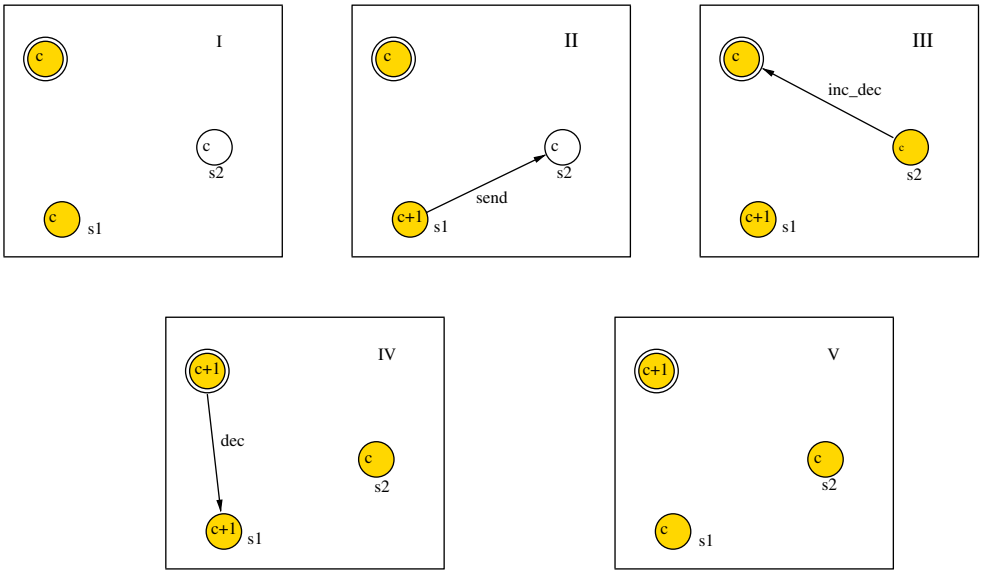
A finite set of sites interact in a distributed way; each site can create a resource and can use it. Our modelling does not concretely handle resource creation and the creator of a resource is its owner. Only the owner of the resource can delete it. The resource may be *shared* by other sites. However, the owner of a resource can delete it only if no other site is using the resource. Because of the distribution of sites, the problem for a site is to have a local knowledge of the current distributed use of its resource. This question is the main problem of the DRC algorithm, which counts (for the owner) the number of sites using a given resource.

Moreau and Duprat [17] provide a distributed algorithm and its correctness proof for safety and liveness properties using the Coq assistant and we summary the general process of the distributed algorithm. If a site (*owner*) has a resource, another site can use a copy of the resource; hence, a site ($s2 \neq owner$) can only obtain a copy from another site ($s1$), which has already got a copy; the site ($s1$) sends its copy to the other site ($s2 \neq s1$). When a site ($s2$) receives a copy from a site ($s1$), it informs the owner that it has received a copy from the site ($s1$) and it sends an `inc_dec` message to the owner ($s1$) that gave it the copy and that it can use the copy. When the owner receives an `inc_dec` message, it knows that ($s2$) has got a copy and sends to ($s1$) a `dec` message asserting that it can remove the message sent to ($s2$). When $s1$ receives from the owner such a `dec` message, it can remove the message sent to $s2$. A site ($s1$) can remove its copy, when it has not any waiting messages which sent the copy to another site. Then, it knows that all of its sent copies are processed by the owner. A site ($s2$) can remove its copy and it sends the owner a `remove` message, When the owner receives this message, it locally knows that ($s2$) will not use its copy, but before receiving this message ($s2$) can obtain another copy.

The assumptions over the network are as follows. All messages are sent from site ($s1$) to site ($s2$) using a buffer. Locally, a site uses a counter to get the number of messages sent to another site. The owner's counter counts also the number of sites that have a copy or have sent a `remove` message which it has not yet received. We assume that there exists a reliable message delivery (messages cannot be lost or corrupted); machines never crash and are never taken out of service; the entire domain is trusted. It is clear that the current development might be reused for further refinements by weakening some assumptions. The diagrams (I)–(V) describe the propagation of copies in the network. Blank sites have no copy of the resource; the double circled site is the owner of the resource and the value c of a node is the local value of the local counter.

(I) The site $s1$ and the owner have a copy of the resource and both nodes are yellow; the site $s2$ has no copy and it is a blank node.
(II) The site $s1$ sends a copy to $s2$ and its counter is incremented by 1; the current value for $s1$ is $c+1$.
(III) The site $s2$ receives the copy from $s1$ and sends to the owner an `inc_dec` message meaning that $s1$ has sent the copy; the `send` message is removed.
(IV) The owner receives the `inc_dec` message; it increments its counter by 1 and the value becomes $c+1$; it sends a `dec` message to $s1$; the `inc_dec` message is removed.
(V) The site $s1$ receives the `dec` message; it decrements its counter by 1 ($c+1-1=c$).

When the counter of a site is equal to 0, it can remove the resource and send a `dec` message to the owner. Both `inc_dec` and `dec` messages to the owner may not yet have been received by the owner. The owner needs to receive first the `inc_dec` message which is the case that arises when using a buffer.



3.2. *Incremental proof-based development in event B*

The incremental proof-based development of a distributed solution starts with a very abstract model integrating the essence of the safety and liveness properties. Next steps lead to the introduction of new events and/or the refinement of previous events; the refinement models are validated using the tools B4free and Click'n'Prove; each new refinement model adds further details to the current view of the target system. The two properties can be simply stated as follows:

- The safety property—*if there exists a reference to a resource, then its reference counter will be strictly positive.*
- The liveness property—*if all references to a resource are deleted, its reference counter will eventually become null.*

The complete development of DRC is made up of seven models (DRC0, …, DRC6) which are related by the refinement relationship:

- DRC0 specifies the view of the owner: *the owner removes the resource and the environment is left unspecified.*

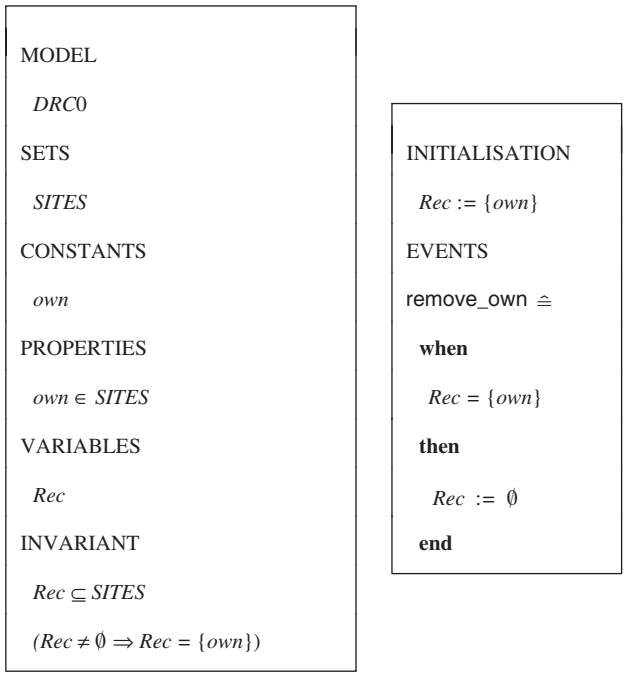


Fig. 3. Model DRC0.

- DRC1 refines the *environment* by introducing new events controlling the use of the resource by other nodes: *the owner removes the resource, when no other copy is in use elsewhere*; it specifies the propagation of copies in a global way.
- DRC2 introduces a tree structure for the control of copies in the network; the tree structure over the network provides a way to record the propagation of copies; the model takes into account only the last `inc_dec` messages which are not followed by the related `dec` message.
- DRC3 is a technical refinement step; in a first attempt, we wrote a first version of DRC4 and we were re not able to discharge a given proof obligation; a stronger invariant was required and it lead us to an intermediate refinement model, namely DRC3.
- DRC4 introduces Moreau's messages using the triangle ($s1$, $s2$, *owner*) described in previous figures; no counter is used.
- DRC5 reorganizes the format of messages to make easier the introduction of counters in the final step.
- DRC6 introduces Moreau's counters for expressing the cardinality of abstract sets.

3.2.1. The model DRC0

The first model DRC0 introduces sites, including a special one called the *owner*; the site *owner* is supposed to propagate the resource, which is not explicitly defined. This model is as simple as possible but sufficiently expressive to handle the essence of the problem: the management of the resource. The goal is to state the condition or the guard for removing the copy of the resource by the owner. The model has only one central event remove_own and a second one stuttering for modelling the environment.

The model DRC0 (Fig. 3) is the starting point of the development and it should express in an obvious way both safety and liveness properties; a set of sites is given by *SITES* and is assumed to be non-empty. A constant called *own* models a given site. The site *own* is supposed to be the site owner of the resource. Only one state variable *Rec* records the fact that either the site *own* has the resource ($own \in Rec$), or the site *own* has not the resource ($own \notin Rec$). The variable *Rec* is initially set to $\{own\}$, since initially the site *own* has the resource. $Rec = \{own\}$ means that the site *owner* still has the resource; when *Rec* is empty, the site *owner* has released the resource.

The safety property—*if there exists a reference to a resource, then its reference counter will be strictly positive*—is not yet visible in the model, since we have no reference counter; the abstract safety property is summarized by the following statement: *if a site owns the resource, the set ReC is not empty and contains own*. DRC0 has only one event called remove_own, which models the possible removing action by the owner, if it owns it. The safety property is still expressed at a very abstract level, since we have not yet obtained an algorithm but we have the view of the owner without mentioning the environment. The event remove_own may occur and we should be careful about the conditions for ensuring the eventuality of that event. In fact, the model expresses that there is a possible starvation of the event remove_own with respect to an environment which may behave badly. A solution would be to control the environment, and this solution was followed by Moreau and Duprat [17] by restricting some actions of the environment. However, these actions are not yet visible at this level of modelling; but the problem is effective.

3.2.2. The refinement model DRC1

The next step is to show how other sites can obtain the resource and how the resource is managed through the network of sites. The model DRC1 introduces a new variable *REC* which is intended to contain the set of sites which are currently using the resource or a copy of the resource. Remember that the resource is shared from the site *owner* and the question is to ensure that the site *owner* knows when the resource is released by other sites. At this step, the information is clearly supposed to be global and the distribution will be handled later. The event *stuttering* is refined into two new events which simulate the propagation/release of the resource.

MODEL
$DRC1$
VARIABLES
REC
INVARIANT
$REC \subseteq SITES$
$Rec \subseteq REC$
$(REC \neq \emptyset \Rightarrow Rec \neq \emptyset)$
INITIALIZATION
$REC := \{own\}$
EVENTS
remove_own $\widehat{=}$
when
$REC = \{own\}$
then
$REC := \emptyset$
end

receive_copy $\widehat{=}$
any s **where**
$REC \neq \emptyset$
$s \in SITES - REC$
then
$REC := REC \cup \{s\}$
end
remove_copy $\widehat{=}$
any s **where**
$s \in REC$
$s \neq own$
then
$REC := REC - \{s\}$
end
END

The model DRC0 takes into account the use of the resource by a given site called *own*; the resource can be propagated to others sites, which can further propagate their copy to other sites. The model DRC1 is a refinement of the model DRC0 and it introduces two new events:

(1) receive_copy for the propagation of the resource or the reception of a copy of the resource.
(2) remove_copy for the release of the resource by a site other than the site *owner*.

The two events are refining the environment and, clearly, provide a hint for understanding why the event remove_own might be infinitely delayed. The two new events can be alternatively executed and the system might enter a live-lock cycle. A solution would be to require a fixed number of these events in order to ensure the liveness property.

In DRC1, the owner sees all other sites and knows what sites are using the resource; the new variable *REC* contains all sites, which have (a copy of) the resource. The owner can remove its resource, when it knows that no other site has

any copy of the resource. The invariant expresses the relationship between *Rec* and *REC*: *REC* contains sites which have a copy of the resource and obviously the owner has a copy of the resource. The remove_own event is refined by modifying the guard which states a property over the variable *REC* instead of *Rec*; thanks to the invariant which allows the validation of the refinement.

3.2.3. The refinement model DRC2

The next step introduces a tree structure, which we call a forest; the forest is built from the traffic of messages sent by sites propagating the resource. The variable *Inc_Dec* is a binary relation over sites and, if a pair $s1 \mapsto s2$ is in the set *Inc_Dec*, it means that $s2$ has got the resource from $s1$ and $s1$ owned the resource. Note that *Inc_Dec* is not itself a tree structure but the inverse relation Inc_Dec^{-1} is a tree-like structure or a forest. *Roots* is a new variable which is expressing the local knowledge of the site *owner*; the site *owner* knows that the sites in *Roots* have the resource but it does not yet know that the sites of ran(*Inc_Dec*) have the resource, since messages are on the way to the site *owner*. Consequently, the variable *REC* is the union of both sets *Roots* and ran(*Inc_Dec*), i.e. the set of sites which have the resource.

The diffusion of copies from the owner follows the forest structure of *Inc_Dec*; a site, which has a copy, can give it to another site without copy. We do not model the *sending* message at this level. We model only a reception of the message, which adds a new site in the copy owner. A new variable called *Inc_Dec* is introduced to model the diffusion tree and a short delay between the owner knowledge (still abstract and exact) and the diffusion (through the forest) of the resource. The owner's knowledge is the set of roots of the forest. The variable *REC* is decomposed into two disjunct subsets, *Roots* (owner's knowledge about copies) and ran(*Inc_Dec*) (the complement). The property over *Inc_Dec* states that it is a forest containing *Roots* and the assertion expresses the acyclicity of the relation. Moreau and Duprat explain that the *inc*/*dec* messages give a tree structure to the node which owns a copy. The challenge of the refinement proof is to prove that the concrete guard of remove_own implies the abstract one:

$$\left(\begin{array}{l} Roots = \{own\} \ \wedge \\ own \notin \mathrm{dom}(Inc\_Dec) \end{array}\right) \Rightarrow REC = \{own\}$$

The refinement is proved by instantiation of the variable q with the singleton {own} to deduce that $REC \subseteq \{own\}$. All antecedents are easily discharged, because $\{own\} \subseteq REC$, $\{own\} \neq \emptyset$, $Roots \subseteq \{own\}$ (because $Roots = \{own\}$ and $Inc\_Dec[\{own\}] \subseteq \{own\}$ (since $own \notin dom(Inc\_Dec)$). The events are modified to take into account the new variables (Fig. 4).

The event remove_own is transformed to handle the new variables; the guard implies the guard of the same event in the model DRC1. The gluing invariant helps to prove the implication.

The event receive_copy modifies the forest structure, by creating a new link between $s1$ and $s2$ in *Inc_Dec*. The acyclicity is preserved, since the site $s2$ had no copy according to the guard.

The event remove_copy models the release of a copy of the resource by a site which is a leaf ($s \notin \mathrm{dom}(Inc\_Dec)$). The event is a global vision of a removal of the copy: the copy is removed and the message is sent to the site *owner*.

The event receive_Inc models the reception of an *inc* message by the owner; a new root with a copy of the resource is created and the forest structure is preserved. $s2$ is added to the set of roots.

3.2.4. The refinement model DRC3

One key-point of the refinement-based development is to manage the complexity of the proof obligations; when a model is too difficult to validate by discharging proof obligations, one should introduce an intermediate model which simplifies the proof process. Unfortunately, our first attempt for the current model leads us to an unproved proof obligation (refinement of remove_copy) and it requires us to enrich the invariant. In the model DRC3, the variables *Rootsf* and *Own* are introduced, and the variables *REC* and *Inc_Dec* remain visible and modifiable by the events of the model DRC3. The variable *Roots* of DRC2 is refined by two new variables:

(1) *Own*, which is empty after execution of remove_own, may contain only *own*; it is a concrete version of the variable *Rec* in the model DRC0.
(2) ran(*Rootsf*) where *Rootsf* has a similar type to *Inc_Dec*. The main challenge is to prove that $Rootsf^{-1}$ is a function.

MODEL

$DRC2$

VARIABLES

$REC, Roots, Inc\_Dec$

INVARIANT

$Inc\_Dec^{-1} \in REC - \{own\} \nrightarrow REC$

$REC = Roots \cup \mathsf{ran}(Inc\_Dec)$

$$\forall q \cdot \left(\left(\begin{array}{l} q \subseteq REC \ \wedge \\ q \neq \emptyset \ \wedge \\ Roots \subseteq q \ \wedge \\ Inc\_Dec[q] \subseteq q \end{array} \right) \Rightarrow REC \subseteq q \right)$$

$Roots \cap \mathsf{ran}(Inc\_Dec) = \emptyset$

$(Roots \neq \emptyset \Rightarrow own \in Roots)$

$(Roots = \emptyset \Rightarrow Inc\_Dec = \emptyset)$

INITIALISATION

$Roots := \{own\} \parallel REC := \{own\} \parallel$

$Inc\_Dec := \{own\}$

EVENTS

remove_own $\widehat{=}$

when

$Roots = \{own\}$

$own \notin \mathsf{dom}(Inc\_Dec)$

then

$REC := \emptyset \parallel Roots := \emptyset$

end

receive_copy $\widehat{=}$

any $s1, s2$ **where**

$s1 \in REC$

$s2 \in SITES - REC$

then

$Inc\_Dec := Inc\_Dec \cup \{s1 \mapsto s2\}$

$REC := REC \cup \{s2\}$

end

remove_copy $\widehat{=}$

any s **where**

$s \in REC$

$s \neq own$

$s \notin \mathsf{dom}(Inc\_Dec)$

then

$Roots := Roots - \{s\}$

$Inc\_Dec := Inc\_Dec \rhd \{s\}$

$REC := REC - \{s\}$

end

receive_Inc $\widehat{=}$

any $s1, s2$ **where**

$s1 \mapsto s2 \in Inc\_Dec$

then

$Inc\_Dec := Inc\_Dec - \{s1 \mapsto s2\}$

$Roots := Roots \cup \{s2\}$

end

Fig. 4. Model DRC2.

MODEL
$DRC3$
VARIABLES
$REC, Inc\_Dec, Rootsf, Own$
INVARIANT
$Own \subseteq SITES$
$Rootsf^{-1} \in SITES \nrightarrow SITES$
$(Own \neq \emptyset \Rightarrow Own = \{own\})$
$Roots = Own \cup \mathsf{ran}(Rootsf)$
INITIALIZATION
$Roots := \{own\}, REC := \{own\}$
$Inc\_Dec := \{own\}, Rootsf := \emptyset$
$Own := \{own\}$
EVENTS
remove_own $\widehat{=}$
when
$Own = \{own\}$
$Rootsf = \emptyset$
$own \notin \mathsf{dom}(Inc\_Dec)$
then
$REC := \emptyset$
$Own := \emptyset$
end

receive_copy $\widehat{=}$
any $s1, s2$ **where**
$s1 \in REC$
$s2 \in SITES - (REC)$
then
$Inc\_Dec := Inc\_Dec \cup \{s1 \mapsto s2\}$
$REC := REC \cup \{s2\}$
end
remove_copy $\widehat{=}$
any s **where**
$s \in REC$
$s \neq own$
$s \notin \mathsf{dom}(Inc\_Dec)$
then
$Rootsf := Rootsf \rhd \{s\}$
$Inc\_Dec := Inc\_Dec \rhd \{s\}$
$REC := REC - \{s\}$
end
receive_Inc $\widehat{=}$
any $s1, s2$ **where**
$s1 \mapsto s2 \in Inc\_Dec$
then
$Inc\_Dec := Inc\_Dec - \{s1 \mapsto s2\}$
$Rootsf := Rootsf \cup \{s1 \mapsto s2\}$
end

The events of DRC2 are modified to take into account the new variables and to ensure the gluing invariant. The event remove_own of DRC2 has a sub-expression $Roots = \{own\}$ which is now substituted by $Own = \{own\}, Rootsf = \emptyset$ according to the gluing invariant. The remove_copy event updates the new variable *Rootsf*, because *Rootsf* is linked to *Roots*. Finally, the event receive_Inc models the reception of an *inc* message by the owner; a new root with a copy of the resource is created and the forest structure is preserved. $s2$ is added to the set of roots; in DRC3, we use the variable *Rootsf* to keep the information that the message has been sent. Basically, the events are modified according to the new variables and the relationship among abstract and concrete variables.

3.2.5. The refinement model DRC4

At this step, we introduce *send* messages to propagate copies of the resource. A site with a copy can send it, using a *send* message, to another site (which may already have a copy). A site sending a copy, keeps this message in *SendLoc*, until the owner asks it to remove the message (waiting message). When a site receives a *send* message, it can send to the owner an *Inc* message, if it has no copy, or else it sends a *SendDec* message to the sender. When a site wants to remove its copy, it sends a *Dec* message to the owner. Our *SendDec* and *Dec* messages correspond exactly to the `dec` messages of Moreau and Duprat. We are now very close to their algorithm. Hopefully, all these messages can be modelled using pairs of *SITES*, like in the previous abstraction. Unfortunately, a site $s1$ can send more than one message to the same site $s2$. Moreau and Duprat solve this question by using a buffer between both sites (messages are indexed in the buffer). Following the refinement methodology, it is too early to introduce buffers in this model, because proofs are more difficult to manage automatically and interactively. The complexity of proofs is strongly related to the complexity of data structures.

We introduce an abstract buffer which is a set in which messages are stamped by counters and this allows us to distinguish two identical messages. *COUNT* is a new set of counters (or dates) and the variable *count* is a subset which contains all counters already used. Each message looks like the following example $(c \mapsto (s1 \mapsto s2))$, where c is a

counter ($c \in count$), $s1$ is the sender and $s2$ is the receiver. New indexed messages are generated by the new event Send_copy. A message m (= $(c \mapsto (s1 \mapsto s2))$) is sent by $s1$ to $s2$ through the *Send* variable and it reaches the site *owner* through the variable *Inc*; finally, it reaches $s1$ through the variable *SendDec*.

Any *Inc* message always arrives before a *dec* message and then both messages can travel to the owner. The site *owner* has not yet a precise view of copies which are used by other sites. The new knowledge of the owner is stored (as a message) in the new variable *RecOwn*, the abstract knowledge of the owner is included in the concrete, because some site can remove the resource. A figure is attached to each event and sketches the scheduling of messages. $s1$ has a copy but the site *owner* does not know that it has a copy. $s1$ can send a copy to $s2$. $s2$ can send a *inc* message to the site *owner*. When the site *owner* receives the message, the site *owner* knows that it has a copy. Then, the site *owner* sends a *dec* message to $s1$. When it receives it, it removes the *SendLoc* message.

The invariant provides typing information for the new variables *RecOwn*, *count*, *Inc*, *Send*, *SendDec*, *Dec* and *SendLoc*.

INVARIANT

$$
\begin{array}{l}
count \subseteq COUNT \\
Send \in count \leftrightarrow (SITES \times SITES) \\
SendLoc \in count \twoheadrightarrow (REC \times SITES) \\
Inc \in count \leftrightarrow (SITES \times SITES) \\
SendDec \in count \leftrightarrow (SITES \times SITES) \\
Dec \in count \leftrightarrow (SITES \times SITES) \\
RecOwn \in count \leftrightarrow (SITES \times SITES)
\end{array}
$$

Each variable is used to model the following information:

(1) The variable *count* contains the counters currently used by the indexed messages.
(2) The variable *Send* contains the messages sent for propagating the resource.
(3) The variable *SendLoc* allows to keep locally the messages sent for propagating the resource and it plays the role of witness.
(4) The variable *Inc* contains the set of *inc* messages sent to the owner for information on the status of the resource.
(5) The variable *RecOwn* models the local knowledge of the site *owner* on the status of the copies in the network.
(6) The variable *SendDec* is used to handle the fact that, when a site has sent an inc message to the site *owner*, the owner should send a *SendDec* message to the site which has sent the copy.
(7) The variable *Dec* is used to manage the *dec* message.

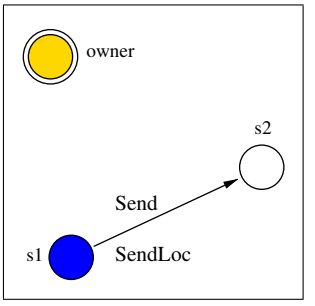
The relation over variables is specified as follows. The range of $Inc - Dec$ is exactly *Inc_Dec* and the range of *RecOnw-Dec* is exactly *Rootsf*. Using the following safety property: $REC = (Own \cup \mathrm{ran}(\mathrm{ran}(RecOwn - Dec)) \cup \mathrm{ran}(\mathrm{ran}(Inc - Dec)))$, we have simplified the guards of events. We summarize the resulting invariant of DRC4:

INVARIANT

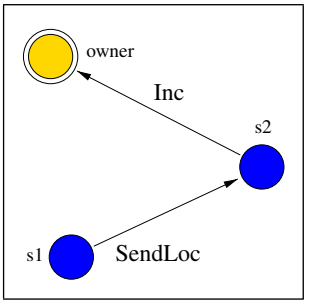
$$
\begin{array}{l}
(RecOwn \cup Inc) - Dec \in count \rightarrowtail\!\!\!\!\rightarrow (SITES \times SITES) \\
Inc\_Dec = \mathrm{ran}(Inc - Dec) \\
Rootsf = \mathrm{ran}(RecOwn - Dec) \\
Send \cup Inc \cup SendDec \subseteq SendLoc \\
Dec \subseteq RecOwn \cup Inc \\
Inc \cap Send = \emptyset \\
Inc \cap SendDec = \emptyset \\
Send \cap SendDec = \emptyset \\
\mathrm{dom}(Send) \cap \mathrm{dom}(RecOwn \cup Inc) = \emptyset
\end{array}
$$

New events for managing the messages are introduced and we used a diagram for explaining the role of several events.

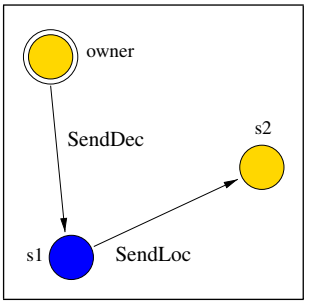
send_copy $\widehat{=}$
any $s1, s2, c$ **where**
$s1 \in REC$
$s2 \in SITES$
$c \in COUNT - count$
then
$Send := Send \cup \{c \mapsto (s1 \mapsto s2)\}$
$SendLoc := SendLoc \cup \{c \mapsto (s1 \mapsto s2)\}$
$count := count \cup \{c\}$
end



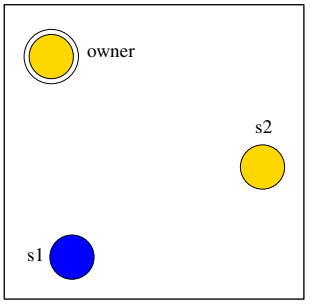
receive_copy $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in Send$
$s2 \notin REC$
then
$Inc := Inc \cup \{c \mapsto (s1 \mapsto s2)\}$
$Send := Send - \{c \mapsto (s1 \mapsto s2)\}$
$REC := REC \cup \{s2\}$
end



receive_Inc $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in Inc - Dec$
then
$Inc := Inc - \{c \mapsto (s1 \mapsto s2)\}$
$RecOwn := RecOwn \cup \{c \mapsto (s1 \mapsto s2)\}$
$SendDec := SendDec \cup \{c \mapsto (s1 \mapsto s2)\}$
end



receive_SendDec $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in SendDec$
then
$SendDec := SendDec - \{c \mapsto (s1 \mapsto s2)\}$
$SendLoc := SendLoc - \{c \mapsto (s1 \mapsto s2)\}$
end



remove_copy $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in (RecOwn \cup Inc) - Dec$
$s2 \neq own$
$s2 \notin \mathrm{dom}(\mathrm{ran}(SendLoc))$
then
$Dec := Dec \cup \{c \mapsto (s1 \mapsto s2)\}$
$REC := REC - \{s2\}$
end

receive_Dec $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in Dec - Inc$
then
$Dec := Dec - \{c \mapsto (s1 \mapsto s2)\}$
$RecOwn := RecOwn - \{c \mapsto (s1 \mapsto s2)\}$
end

remove_own $\widehat{=}$
when
$Own = \{own\}$
$ReCOwn = \emptyset$
$own \notin \mathsf{dom}(\mathsf{ran}(SendLoc))$
then
$REC := \emptyset$
$Own := \emptyset$
end

receive_SendDec $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in SendDec$
then
$SendDec := SendDec - \{c \mapsto (s1 \mapsto s2)\}$
$SendLoc := SendLoc - \{c \mapsto (s1 \mapsto s2)\}$
end

The question is to have a complete description of the model

receive_notcopy $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in Send$
$s2 \in REC$
then
$SendDec := SendDec \cup \{c \mapsto (s1 \mapsto s2)\}$
$Send := Send - \{c \mapsto (s1 \mapsto s2)\}$
end

When a site receives a *Send* message, it may already have a copy of the message and a *SendDec* message is sent using the following event: unfortunately, our development work is not yet over. Our current model may be deadlocked.

If we analyse guards, *Inc* and *Dec* messages are received by the owner, when the other one is not present: $c \mapsto (s1 \mapsto s2) \in Inc - Dec$ in the guard of receive_Inc and $c \mapsto (s1 \mapsto s2) \in Dec - Inc$ in the guard of receive_Dec. Hopefully, the guard of receive_Inc can be changed into $c \mapsto (s1 \mapsto s2) \in Inc$: however, in this case the refinement proof fails. In fact, another event is missing and it acts like receive_Inc:

receive_Incbis $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in Inc \cap Dec$
then
$Inc := Inc - \{c \mapsto (s1 \mapsto s2)\}$
$RecOwn := RecOwn \cup \{c \mapsto (s1 \mapsto s2)\}$
$SendDec := SendDec \cup \{c \mapsto (s1 \mapsto s2)\}$
end

It should be noted that in the abstraction *ReC* (or *ReCf*) are found to be the effective view of all copies. In this refinement, we have a delay between the effective view and the site view of the *owner*.

3.2.6. The refinement model DRC5

The main difficulties of the problem have been overcome and we modify the data structures by transformation as follows: *SendLoc* messages, such as like $c \mapsto (s1 \mapsto s2)$ are transformed into messages, such as $s1 \mapsto (c \mapsto s2)$ of the variable *LocSend*. We prove that the set *LocSend* can replace *SendLoc* and then we can use $LocSend[\{s1\}]$ to get each message sent by $s1$.

Invariant

$$LocSend \in SITES \leftrightarrow (COUNT \times SITES)$$

$$\forall (s1, s2, c) \cdot \left(\begin{array}{l} s1 \in SITES \ \wedge \\ s2 \in SITES \ \wedge \\ c \in COUNT \\ \Rightarrow \\ (c \mapsto (s1 \mapsto s2) \in SendLoc \Leftrightarrow s1 \mapsto (c \mapsto s2) \in LocSend) \end{array} \right)$$

Only four events are modified to take into account the transformations over the *SendLoc* messages.

send_copy $\widehat{=}$
any $s1, s2, c$ **where**
$s1 \in REC$
$s2 \in SITES$
$c \in COUNT - count$
then
$Send := Send \cup \{c \mapsto (s1 \mapsto s2)\}$
$LocSend := LocSend \cup \{s1 \mapsto (c \mapsto s2)\}$
$count := count \cup \{c\}$
end

receive_SendDec $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in SendDec$
then
$SendDec := SendDec - \{c \mapsto (s1 \mapsto s2)\}$
$LocSend := LocSend - \{s1 \mapsto (c \mapsto s2)\}$
end

remove_copy $\widehat{=}$
any $s1, s2, c$ **where**
$c \mapsto (s1 \mapsto s2) \in (RecOwn \cup Inc) - Dec$
$s2 \neq own$
$s2 \notin \mathsf{dom}(LocSend)$
then
$Dec := Dec \cup \{c \mapsto (s1 \mapsto s2)\}$
$REC := REC - \{s2\}$
end

remove_own $\widehat{=}$
when
$Own = \{own\}$
$ReCOwn = \emptyset$
$own \notin \mathsf{dom}(LocSend)$
then
$REC := \emptyset$
$Own := \emptyset$
end

The resulting model is in fact a technical intermediate model, which is more convenient for incrementally introducing the real traffic of messages.

3.2.7. The refinement model DRC6

The model DRC5 results from a transformation over Cartesian products for the variable *SendLoc* and, consequently, we can implement the set *LocSend* using a total function on sites leading to an implementation of counter. In DRC6, *LocSend*, *RecOwn* and *Own* are removed, but are still in the abstraction, thanks to the refinement. Each site s evaluates the number of *send* messages (in $LocSend[\{s\}]$) and *own* contains the sum of the cardinality of *send* messages (in $LocSend[\{s\}]$) and the cardinality *RecOwn*. Messages (*Send*, *Inc*, *Dec*, *SendDec*) are messages of the abstraction. When removing copy, a site $s2$ should know the complete information on the received messages and then it can send a *Dec* message. The variable *cREC* plays this role. The (gluing) invariant is a simple transformation over the previous one:

Gluing Invariant
$$CLocSend \in SITES \to \mathbb{N}$$
$$\forall s \cdot \left(\begin{array}{l} s \in SITES - \{own\} \\ \Rightarrow \\ CLocSend(s) = \mathsf{card}(LocSend[\{s\}]) \end{array} \right)$$
$$CLocSend(own) = \mathsf{card}(LocSend[\{own\}]) + card(RecOwn)$$
$$cREC \in REC - \{own\} \to (RecOwn \cup Inc) - Dec$$
$$\forall s \cdot \left(\begin{array}{l} s \in REC - \{own\} \\ \Rightarrow \\ \exists (c, s1) \cdot (c \in count \ \wedge \ s1 \in SITES \ \wedge \ cREC(s) = c \mapsto (s1 \mapsto s)) \end{array} \right)$$

At this point of the development, buffers are not introduced but we can translate each event into an imperative action by indicating the site which is locally executing the action. Each event can be a piece of the real algorithm if we give the local site. The set *REC* looks like a boolean. A site s is in *REC* if and only if s has a copy (or is the owner). With transformations we can easily obtain the five events from Moreau and Duprat's algorithm.

```
remove_own ≙
 when
  own ∈ REC
  CLocSend(own) = 0
 then
  REC := ∅
 end
```

However, our remove_own event is not present in their paper. This event is the starting event of our work. In the sequel, we sketch the way to translate events into local actions. The event remove_own is local to the site *own* and the variable *CLocSend* is localized at *own*. The occurrence of $CLocSend(own)$ is substituted by *CLocSend*, meaning that *CLocSend* is locally defined in the site *own* and is a natural:

```
send_copy ≙
 any s1, s2, c where
  s1 ∈ REC
  s2 ∈ SITES
  c ∈ COUNT−count
 then
  Send := Send ∪ {c ↦ (s1 ↦ s2)}
  CLocSend(s1) := CLocSend(s1) + 1
  count := count ∪ {c}
 end
```

The event send_copy is localized at $s1 \in SITES$; $s1$ sends a message $(Send(resource))$ to $s2$. We recall that the communication medium is supposed to be reliable and is managed according to the FIFO policy. The action propagates the copy of the resource. We can use a variable *resource* to express the value of the resource or copy, since $s1 \in REC$.
Events receive_copy and receive_notcopy can be combined into one action. The local site is $s2$ (the sender is $s1$) and the condition is $s2 \in REC$.

```
receive_copy ≙
 any s1, s2, c where
  c ↦ (s1 ↦ s2) ∈ Send ∧ s2 ∉ REC
 then
  Inc := Inc ∪ {c ↦ (s1 ↦ s2)}
  Send := Send−{c ↦ (s1 ↦ s2)}
  REC := REC ∪ {s2}
  cREC(s2) := c ↦ (s1 ↦ s2)
 end
```

```
receive_notcopy ≙
 any s1, s2, c where
c ↦ (s1 ↦ s2) ∈ Send ∧ s2 ∈ REC
 then
  SendDec := SendDec ∪
          {c ↦ (s1 ↦ s2)}
  Send := Send−
          {c ↦ (s1 ↦ s2)}
 end
```

```
receive_Inc_and_Incbis ≙
 any s1, s2, c where
  c ↦ (s1 ↦ s2) ∈ Inc
 then
  Inc := Inc−{c ↦ (s1 ↦ s2)}
  CLocSend(own) := CLocSend(own) + 1
  SendDec := SendDec ∪ {c ↦ (s1 ↦ s2)}
 end
```

Events receive_Inc and receive_Incbis are closely related; when both events are translated, they can be combined into an event: the set *Inc* satisfies the following property $(Inc−Dec \cup (Inc \cap Dec)) = Inc$ which is the guard of both events. The site which receives the *Inc* message is the owner of the resource.

```
remove_copy ≙
 any s2 where
  s2 ∈ REC−{own}
  CLocSend(s2) = 0
 then
  Dec := Dec ∪ {cREC(s2)}
  REC := REC−{s2}
  cREC := {s2} ⩤ cREC
 end
```

Events receive_Dec and receive_SendDec are similar only the owner receives *Dec* messages but always after the corresponding *Inc* message (thanks to buffers). The *SendDec* message is sent by the site $s2$ and received by $s1$ which decrement its counter.

```
receive_Dec ≙
 any s1, s2, c where
  c ↦ (s1 ↦ s2) ∈ Dec−Inc
 then
 Dec := Dec−{c ↦ (s1 ↦ s2)}
 CLocSend(own) := CLocSend(own)−1
 end
```

```
receive_SendDec ≙
 any s1, s2, c where
  c ↦ (s1 ↦ s2) ∈ SendDec
 then
 SendDec := SendDec−
                {c ↦ (s1 ↦ s2)}
 CLocSend(s1) := CLocSend(s1)−1
 end
```

Translation of remove_own
LOCAL NODE $owner \in SITES$
IF REC **and** $CLocSend = 0$ **THEN** $REC := FALSE$

Translation of send_copy($s2 \in SITES$)
LOCAL NODE $s1 \in SITES$
IF REC **and** $s1 \neq s2$ **THEN**
$CLocSend := CLocSend{+}1$
$SEND(Send(resource))$ TO $s2$ IN $BUF(s1, s2)$

Translation of receive_copy, receive_notcopy
LOCAL NODE $s2 \in SITES$
precondition $First(BUF(s1, s2) = Send(ressource)$
IF REC **THEN**
$SEND(dec)$ TO $s1$ IN $BUF(s2, s1)$
ELSE
$REC := TRUE$
$SEND(inc(s1))$ TO $owner$ IN $BUF(s2, owner)$

Translation of receive_DeC, receive_SendDec
LOCAL NODE $s \in SITES$
precondition $First(BUF(s1, s) = dec$
$CLocSend := CLocSend{-}1$

Translation of receive_Inc, receive_Incbis
LOCAL NODE $owner \in SITES$
precondition $First(BUF(s2, owner) = inc(s1)$
$CLocSend := CLocSend{+}1$
$SEND(dec)$ TO $s1$ IN $BUF(owner, s1)$

Translation of remove_copy
LOCAL NODE $s \in SITES$
precondition$s \neq owner$
IF REC **and** $CLocSend = 0$ **THEN**
$REC := FALSE$
$SEND(dec)$ TO $owner$ IN $BUF(s, owner)$

Fig. 5. The algorithm DRC.

Some events are translated into the local actions in Fig. 5 from the events of DRC6. The key-idea is to sufficiently localize the events. Each site has two local variables: *REC* a boolean which is true iff the site has the resource, and a counter *CLocSend* ($\in \mathbb{N}$). *BUF* is a global variable and $BUF(s1, s2)$ is a buffer between both sites $s1$ and $s2$. Some actions have a precondition on the buffer and $First(BUF(s1, s2)) = x$ means that x is a message sent by $s1$ to $s2$ which is not received in the FIFO policy. $First(BUF(s1, s2))$ provides the message which is removed from the buffer.

4. Comments and comparisons with other work

4.1. On the proof process

The source code for the proof in Coq is approximately 13 000 lines long [17], but a similar measure, based on the number of proof lines is not adequate for the fair comparison with the B models development. In fact, the complexity of proofs is due to the inherent complexity of data structures used in formal models expressed in Coq. Our belief is that refinement, and the choice of the right abstraction make more tractable proof obligations generated automatically from the text of B models. Proofs (of proof obligations) provide an objective reference or explanation that an invariant is in

fact an inductive property. The proof assistant (Click'n'Prove) is really used to improve the search for the invariant. It may happen that an assertion of a given (abstract) model is not invariant (or inductive) and a proof obligation is not discharged: either the property is not an invariant, or the tool is not powerful enough. The more complex the model is, the more complex is the discovery of the invariant. If the user (or the specifier) is not able to interpret failures of the proof assistant, he/she will struggle. However, the refinement guarantees the possibility to obtain models sufficiently expressive and simple.

Refinement, together with a cognitive incrementability of models, help in organizing and structuring the systematic construction of the final solution. The modelling of distributed algorithms is based on the classical semantical model of non-deterministic interleaving and fairness requirements may be added to the models, like in TLA. The style of modelling is often related to the kind of specification language being used. For instance, the B modelling language permits one to manipulate sets, relations and partial functions; the partiality of functions (and relations) facilitates the development of our formal models. On the other hand, Coq provides a typed language for describing models and is based on total functions. Coq had no automatic procedures when Moreau and Duprat constructed interactively proofs of invariance; Moreau mentions that he spent three weeks at this time and later he redeveloped the proof in 10 h using a more automated version of Coq. From DRC0 to DRC2, the duration for proved development is evaluated to 3 h: the proved development includes the writing of B models and the verification of B models; each proof obligation is automatic or requires little interaction (e.g.: instantiation of the inductive property on the tree structure). DRC4 introduces communications which cannot be introduced separately and it explains why we have 9 proof obligations to discharge. Proof obligations generated from B models are easily discharged. The main point of interest is that we have distributed difficulties of proofs using refinement. Half of interactive proofs are done on the last most concrete model. There are technical proofs on cardinalities: refinement proof such $card(s) = 0 \Rightarrow s = \emptyset, x \notin s \Rightarrow card(sx) = card(s) + 1, x \in s \Rightarrow card(s-x) = card(s)-1$, which can be automated, if the prover was improved.

4.2. On the modelling process

The incremental proof-based development allows us to explain how models are working; each refinement step adds new details either in the events or in the invariant. The previous invariant properties are preserved and are not interfering with the current discharge of proof obligations generated in the current refinement step. The final algorithm is progressively explained. The invariant of Moreau and Duprat is expressed on the whole set of variables of the algorithm. It is thus much more difficult to be proven. The majority of our 'technical' invariants were detected during the proof. We thus progressed at the same time in the comprehension of the algorithm and its proofs. These even technical invariants often provide us with better understanding of the algorithm. Our second model highlights in a simple way the problem of termination of the algorithm. It improves the resolution of these problems by limiting the triggering of an abstract event. In our third model, the variable *Inc_Dec* models exactly the set of last messages `inc_dec` in the buffer but which is not followed by a message `dec`. Our modelling being more abstract, the required properties are much simpler to prove. Our event remove_own, which was the only event of our first model was not present in the algorithm of Moreau and Duprat.

5. Conclusion and future work

Fig. 6 gives details of the number of proof obligations for the complete development; the model DRC4 is the most difficult to prove, since it required 9 interactive proofs (the main reason is that it introduces messages). If we consider liveness properties, we understand very early that the algorithm may not terminate; in fact, in DRC1, the two new events can take control forever and maintain the global process in a live-lock status. The proof of termination is ensured by limiting occurrence of receive_copy event in DRC1; in DRC4, we limit the number of send_copy events and it guarantees the control of receive_copy events. 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 problem. What is important about our approach is that the fundamental properties we have proved at the beginning are kept throughout the refinement process (provided, of course, the required proofs are done). The current development adds a new element to a library of proof-based developed (distributed) algorithms [6,5] and further work will explore new problems of the very challenging domain of distributed algorithms. Probabilistic distributed algorithms bring new questions to be solved with respect to the notion of refinement.

Model	Number of proof obligation	Interactive proved
DRC0	1	0
DRC1	6	0
DRC2	37	5
DRC3	14	2
DRC4	94	9
DRC5	19	5
DRC6	48	21
TOTAL	219	42

Fig. 6. Summary of proof obligations for the DRC development.

Acknowledgements

We are grateful to referees for detailed comments on previous versions; they have kindly suggested improvements of our French English version and have pointed out technical problems.

References

[1] J.-R. Abrial, B#: toward a synthesis between z and b, in: D. Bert, J.P. Bowen, S. King, M.A. Waldén (Eds.), ZB, Lecture Notes in Computer Science, Vol. 2651, Springer, Berlin, 2003, pp. 168–177.
[2] J.R. Abrial, The B Book–Assigning Programs to Meanings, Cambridge University Press, Cambridge, 1996 ISBN 0-521-49619-5.
[3] J.-R. Abrial, D. Cansell, Click'n'Prove, 2002. ⟨www.loria.fr/cansell/cnp.html⟩.
[4] J.-R. Abrial, D. Cansell, Click'n'prove: interactive proofs within set theory, in: D.A. Basin, B. Wolff (Eds.), TPHOLs, Lecture Notes in Computer Science, Vol. 2758, Springer, Berlin, 2003, pp. 1–24.
[5] J.-R. Abrial, D. Cansell, Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity), J. Universal Comput. Sci. 11 (5) (2005) 744–770.
[6] J.-R. Abrial, D. Cansell, D. Méry, A mechanically proved and incremental development of ieee 1394 tree identify protocol, Formal Asp. Comput. 14 (3) (2003) 215–227.
[7] J.-R. Abrial, L. Mussat, Introducing dynamic constraints in B, in: D. Bert (Ed.), B, Lecture Notes in Computer Science, Vol. 1393, Springer, Berlin, 1998, pp. 83–128.
[8] R.J.R. Back, On correct refinement of programs, J. Comput. System Sci. 23 (1) (1979) 49–68.
[9] D. Cansell, G. Gopalakrishnan, M.D. Jones, D. Méry, A. Weinzoepflen, Incremental proof of the producer/consumer property for the pci protocol, in: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (Eds.), ZB, Lecture Notes in Computer Science, Vol. 2272, Springer, Berlin, 2002, pp. 22–41.
[10] D. Cansell, D. Méry, The B archive for the DRC development, http://www.loria.fr/∼cansell/drc, January 2006.
[11] J. Chalopin, Y. Métivier, A bridge between the asynchronous message passing model and local computations in graphs, in: J. Jedrzejowicz, A. Szepietowski (Eds.), MFCS, Lecture Notes in Computer Science, Vol. 3618, Springer, Berlin, 2005, pp. 212–223.
[12] K.M. Chandy, J. Misra, Parallel Program Design A Foundation, Addison-Wesley Publishing Company, Reading, MA, 1988 ISBN 0-201-05866-9.
[13] ClearSy, Web site B4free set of tools for development of B models, 2004.
[14] N. Francez, Fairness, Texts and Monographs in Computer Science, Springer, Berlin, 1986.
[15] L. Lamport, A temporal logic of actions, ACM Trans. Programming Languages Systems 16 (3) (1994) 872–923.
[16] D. Méry, A proof system to derive eventually properties under justice hypothesis, in: J. Gruska, B. Rovan, J. Wiedermann (Eds.), MFCS, Lecture Notes in Computer Science, Vol. 233, Springer, Berlin, 1986, pp. 536–544.
[17] L. Moreau, J. Duprat, A construction of distributed reference counting, Acta Inform. 37 (2001) 563–595.
[18] D. Park, A predicate transformer for weak fair iteration, in: IBM (Ed.), Proc. of the IBM Conference, IBM, 1981, pp. 259–275.