Formal Development of a Real-Time Operating System Memory Manager

Wen Su[1], Jean-Raymond Abrial[2], Geguang Pu[3], and Bin Fang[3]

[1] School of Computer Engineering and Science, Shanghai University
wsu@shu.edu.cn
[2] Marseille, France
jrabrial@neuf.fr
[3] Software Engineering Institute, East China Normal University
ggpu@sei.ecnu.edu.cn, fangbin@ecnu.cn

Abstract. This paper presents the complete development of the memory management module of a real time operating system. The interesting feature of this type of memory manager is that its dynamic memory allocation/deallocation mechanism behaves in O(1) (no loops). This brings a serious challenge on the "correct by construction" approach used to build this kind of system. This is due to the necessity to elaborate some delicate algorithms associated with complex data structures. To overcome this challenge, we follow the refinement principles of Event-B: we construct the proved final executable code from some initial requirements. This development is interesting because some of the encountered problems are rather rarely studied in formal proved developments, among which are a modular encapsulation development, the design pattern of a linked list, and the usage of guarded events to develop pre-conditioned operations, etc. It also gives us the opportunity to study a complex program construction in some general terms going beyond this specific example.

1 Introduction

There are many dynamic storage allocation algorithms proposed in the literature for memory management [24]. Among these algorithms, the Two-Level Segregate Fit (TLSF) algorithm [18] with a time cost O(1) is a good candidate, as the time performance of a memory manager is an important factor for real-time operating systems. TLSF has been widely used in many systems [29] such as Hypervisors [34], Amiga OS [5], Orocos [23], and plenty of real-time kernels. However, the complexity of this algorithm results in sophisticated data structures and operations. These complexities entail some difficulties for guaranteeing the correctness of TLSF implementation.

This very interesting example had been provided to us by our industrial partners who want to ensure that their operating system (embarked on board satellites) is bug free. This is the reason why they are quite interested to know whether some mathematical methods could be used for that purpose. When given to us initially, we thought this example was rather simple, but we found gradually that the formal handling of it raised some difficult questions that needed to be solved by means of the elaboration of new techniques.

Related Work. In this development we follow the TLSF approach informally described in [18, 19]. In these papers, the algorithm presentation is very descriptive only (no proof of correctness). This is the reason why it is very interesting to correctly construct this algorithm by means of refinements and proofs. We follow the work of Abrial and Leino on constructing the memory management of an hypervisor [3]. Some other related works are the following: the well-known work of seL4 team [14, 13] contains the formal verification of a virtual memory [16, 15]. Finally, the formal verification of a baby virtual memory based on abstraction is presented in [31].

Contributions. Our approach is presented by means of this dynamic memory management. The intent of this paper is to emphasise *four important new topics* in using formal methods for developing and proving complex software systems:

1. We experiment with a *modular construction* as advocated for a long time by programming language technologies (Modula, ADA, Object Oriented Languages, etc.). Modular constructions however have not been so far used very much in formal method approaches. Here we follow the early approach developed in [7] for Action System, and that developed for Object Z in [26].
2. Part of our development deals with the new construction of a *delicate algorithm* (for a doubly linked list handling). Formal developments have not been so far used very much in the construction of such algorithms. This algorithmic construction is made independently of the rest of the paper in sect. 3.
3. Our development will use the *Event-B technique* (using abstraction, refinement, and proofs) [2, 25, 32]. This technique makes a heavy usage of *events* defined by means of *guards* and *parallel actions*. Although very powerful (in particular for the formal verification proofs), this approach is not completely satisfactory to develop classical programs where the dynamic part is defined by means of *operations* dealing with *pre-conditions* and *sequential actions*. In this development, we show how a new technique allows us to use events to formally develop such classical operations. This approach has not been published yet. However, it has been presented in the Event-B workshop connected to the last ABZ Conference (2014). Some slides are available in [4].
4. In this development, we push the refinements to a final most concrete one that can then be easily translated into some *executable code* (C code in our case). Although the translation we propose here is only done manually, we hope to convince the reader that it could have been done by a tool. We followed the translation approaches already developed in [11, 21, 22, 30, 20, 33, 10, 8, 9]. But we also use some new techniques for handling sequential programs and modular organisation.

Outline. In this development, we follow the usual approach recommended for Event-B model construction [2, 28]:

- *Informal Requirement.* A very solid informal requirements definition of the system properties is proposed (sect. 2).
- *Design Pattern.* The handling a doubly linked list is presented (sect. 3).
- *Refinement Strategy.* Here we propose a well defined refinement strategy (sect. 4) explaining in which order the requirements are taken into account in the successive formal models. We also check that all requirements defined in sect. 2 are taken into account.

– *Formal Model.* We give a complete formal model construction with refinements and proofs (sect. 5). Model-checker and animator [6, 17] are also used.
– *Code Generation.* It is based on the final refinement (sect. 5.7).

All this is illustrated in Fig. 1. Interested readers can access our Event-B and Rodin development and the C code as well in the web site [1].

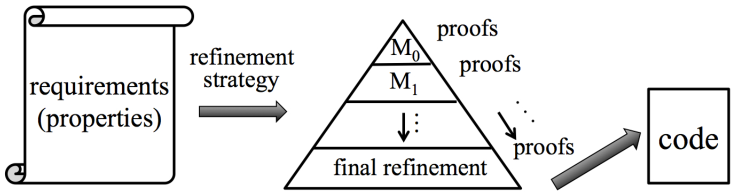


Fig. 1. Overall Approach

2 Requirement Document

Before starting the formal development, it is very important to make very clear what the functional requirements and environment assumptions are (hence the size of this section). Such requirements and assumptions, written in natural language, should not be an informal pseudo-implementation. Their roles are to give arguments able to be used to judge that the final constructed system is correct. They are labelled as follows: FUN, denoting functional requirements, and ENV, denoting environment assumptions.

2.1 The Memory

This system manages the memory of an operating system.	FUN-1

The memory is made of values stored at certain addresses.

The addresses of the memory all belong to a finite natural number interval starting at 0.	ENV-1

2.2 The Blocks

The entire memory is covered by blocks.	ENV-2

A block is made of an interval of contiguous addresses and related values.

The memory of a block is contiguous.	ENV-3

Blocks always contain some addresses (at least one memory address).

There is no empty block.	ENV-4

Each block has thus a first address and a positive size: they both identify completely the block in the whole memory pool.

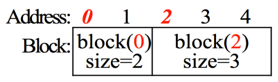


Fig. 2. First address and size

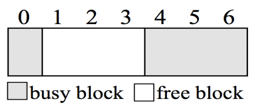


Fig. 3. Block is either busy or free

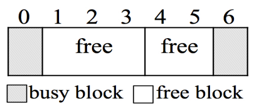


Fig. 4. Not possible: two free blocks adjacent

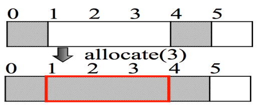


Fig. 5. free block 1 with a size exactly equal to the one required

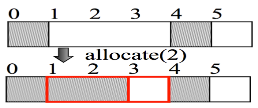


Fig. 6. free block 1 with a size greater than the one required

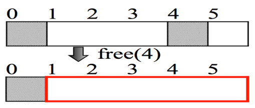


Fig. 7. Block 4 is made free and merged it adjacent free blocks

Each block has a first address and a size.	ENV-5

On Fig. 2, one can see two blocks covering the memory: blocks 0 and 2. Notice that we follow the convention that the name of a block is its first address.

The addresses of a block belong to the interval between its first and last addresses (first address plus size minus 1).	ENV-6

There is no gap between blocks: the blocks form a partition of the memory.

Each element of the memory belongs to a block.	ENV-7

A block's interval of addresses should not interfere with other block addresses.

Blocks do not overlap.	ENV-8

2.3 Allocating and Freeing Blocks

When a block is allocated it is said to be busy, otherwise it is free.

Each block is either busy or free.	FUN-2

On Fig. 3, we see three block: 0, 1, and 4. Blocks 0 and 4 are busy whereas block 1 is free. An important property of this system is that free blocks are not surrounded (to the right or to the left) by other free blocks.

A free block is always surrounded by busy blocks.	FUN-3

On Fig. 4, we see a bad situation : two free blocks are next to each other.

Initially, there is a unique free block covering the entire memory.	FUN-4

Two operations are provided to make busy a free block or free a busy block.

The system provides two operations: "Allocate a new block of memory" and "Free a block of memory".	FUN-5

2.4 The Allocating Operation

The effect of the allocating operation is to transform totally or partially a free block into a busy one. It has one parameter: the size of the new busy block.

The size of the required memory block is the unique parameter of the allocating operation.	FUN-6

Next is the details of the external behaviour of the allocating operation. A free block is made totally or partially busy as indicated.

When the allocating operation is called with a parameter q: – if a free block b with a size exactly equal to q is found, then block b is made busy and the operation succeeds. – if a free block b with a size greater than q is found, then the block b, limited to the size q, is made busy, the remaining part of b forms a new free block, and the operation succeeds. – if no free block with a size greater than or equal to q is found, then the operation fails.	FUN-7

The behaviour described in FUN-7 is illustrated in Fig.5 and Fig.6. The allocating operation is optimised to find the most accurate free block to become busy.

The allocating operation should find a block whose size is closed to the required size.	FUN-8

2.5 The Freeing Operation

The effect of the freeing operation is to make free a busy block which is a correct parameter of this operation.

The busy block to be freed is a parameter of the freeing operation.	FUN-9

If the parameter of the freeing operation is not a busy block, then the operation fails otherwise it always succeeds.	FUN-10

A busy block being made free, it might be the case that this block is merged with surrounding blocks (if any) in order to avoid having free blocks next to each other.

When a busy block is freed by the freeing operation, some free surrounding blocks (if any) of the new free block are merged with it.	FUN-11

This is illustrated in Fig. 7.

2.6 Implementation of the Arrangement of Free Blocks

Free blocks are grouped according to their size. More precisely, we have a number of "groups", each of which corresponds to an interval of block sizes.

Each free block belongs to a certain group corresponding to its size.	FUN-12

The sizes of the groups covers all possible sizes.

Each group is defined by means of a size interval. The various size intervals of all groups partition the entire possible block sizes.	FUN-13

Next is the allocation strategy to search for a free block of the relevant size.

When searching for a free block of a certain size q to make it busy, the allocation operation makes a search in the smallest group whose interval lower bound is not smaller than the size q. If this group is not empty then a free block in it is chosen. Otherwise a group with bigger sizes is searched, and so on.	FUN-14

FUN-14 guarantees that in the group selected by a search, all blocks have a size greater than or equal to q. Therefore, there is no need to search any more for a block inside that group: a block can be chosen randomly. In other words, the search for a free block is simple: one has to find the smallest group which is not empty and whose lower bound is not smaller than the size q. In the implementation of the TLSF algorithm, instead of searching groups one by one, this group can be located directly by some hash functions. In order to locate the group faster than with a one dimensional data structure, a two dimensional one is used to organise the groups. TLSF algorithm can find a “good fit” free block with a time cost in O(1). More explanation of O(1) implementation can be seen in sect. 5.8.

Free blocks in a group are stored by means of a doubly linked list.	FUN-15

An implementation of the free blocks in each groups is done according to a doubly linked list as illustrated in Fig. 8. The double link allows for an easy removal of a block from a list.

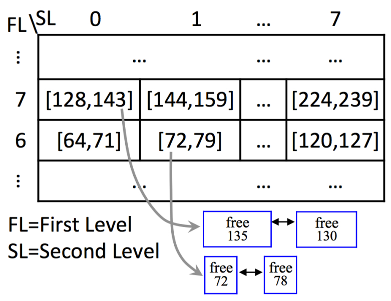


Fig. 8. Free block in a size interval group are stored with a doubly linked list

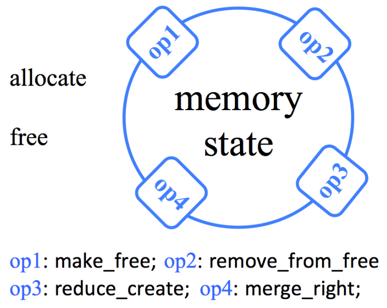

allocate
free
memory state
op1
op2
op3
op4
op1: make_free; op2: remove_from_free
op3: reduce_create; op4: merge_right;

Fig. 9. Basic operations forming a module around the memory state

3 A Design Pattern: a Doubly Linked List and its Operations

In this section, we develop an independent model for a dynamic doubly linked list made of distinct elements (this model is different from the static one presented in [2]). This model will be used as a *design pattern* [12] in our development in sect. 5.6. It contains an initial model defining the forward link and then a refinement defining the backward link.

3.1 Initial Model: Forward Link

The state. We first define a carrier set S and the constant nil which is a member of S. The state is defined by three variables: s, f, and nx.

- The variable s is a subset of S (**inv1**). It is supposed to record the distinct elements of the list. Notice that nil is not part of s (**inv2**).
- The variable f points to the first element of the list when it is not empty. It is equal to nil otherwise (**inv3, inv4**).
- The variable nx records the forward link of the list: it is a bijection from the set s to the set $(s \cup \{nil\}) \setminus \{f\}$ (**inv5**). Finally, we state that the list contains no cycles and is not infinite (**inv6**).

inv1: $s \subseteq S$
inv2: $nil \notin s$
inv3: $f \in s \cup \{nil\}$

inv4: $f = nil \;\Rightarrow\; s = \varnothing$
inv5: $nx \in s \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} (s \cup \{nil\}) \setminus \{f\}$
inv6: $\forall p \cdot p \subseteq nx^{-1}[p] \;\Rightarrow\; p = \varnothing$

The Events. We have an event for adding a new element at the beginning of the list (add) and two events for removing an element from the list: rmv1 when the removed element is not the first one in the list and rmv2 otherwise. These events are shown as below. Initially, $s := \varnothing$, $f := nil$, $nx := \varnothing$.

```
add ≙
any
  $x$
where
  $x \notin s$
  $x \neq nil$
then
  $nx := nx \cup \{x \mapsto f\}$
  $s := s \cup \{x\}$
  $f := x$
end
```

```
rmv1 ≙
any
  $x$
where
  $x \in s$
  $f \neq x$
then
  $s := s \setminus \{x\}$
  $nx := (\{x\} \mathbin{\lhd\!\!\!-} nx \mathbin{\rhd\!\!\!-} \{x\}) \;\cup$
          $\{nx^{-1}(x) \mapsto nx(x)\}$
end
```

```
rmv2 ≙
any
  $x$
where
  $x \in s$
  $f = x$
then
  $s := s \setminus \{x\}$
  $nx := \{x\} \mathbin{\lhd\!\!\!-} nx$
  $f := nx(x)$
end
```

3.2 Refinement: Backward Link

The State. We have a new variable pr denoting the backward link. It is a function defined in terms of the variable nx (**inv1**).

inv1: $pr = \{nil\} \mathbin{\lhd\!\!\!-} (nx^{-1} \cup \{f \mapsto nil\})$

The Events. We refined the events defined in the initial model in a straightforward way. Initially, $s := \varnothing$, $f := nil$, $nx := \varnothing$, $pr := \varnothing$.

```
add $\widehat{=}$
refine
  $add$
any
  $x$
where
  $x \notin s$
  $x \neq nil$
then
  $nx := nx \cup \{x \mapsto f\}$
  $s := s \cup \{x\}$
  $f := x$
  $pr := (\{f\} \lhd pr) \cup$
          $(\{nil\} \lhd w)$
end
```

```
rmv1 $\widehat{=}$
any
  $x$
where
  $x \in s$
  $f \neq x$
then
  $s := s \setminus \{x\}$
  $nx := (\{x\} \lhd nx \rhd \{x\}) \cup$
          $\{pr(x) \mapsto nx(x)\}$

  $pr :=$  $(\{x\} \lhd pr \rhd \{x\}) \cup$
          $(\{nil\} \lhd \{n \mapsto p\})$
end
```

```
rmv2 $\widehat{=}$
any
  $x$
where
  $x \in s$
  $f = x$
then
  $s := s \setminus \{x\}$
  $nx := \{x\} \lhd nx$
  $f := nx(x)$
  $pr := (\{x, nx(x)\} \lhd pr) \cup$
          $(\{nil\} \lhd \{n \mapsto p\})$
end
```

where $\quad n = nx(x), \quad p = pr(x), \quad w = \{f \mapsto x, x \mapsto nil\}$

Table 1. Animation of the refined doubly linked list

Event	s	f	nx	pr
	∅	nil	∅	∅
add(a1) ⇓				
	{a1}	a1	{a1↦ nil}	{a1↦ nil}
add(a3) ⇓				
	{a1,a3}	a3	{a1↦nil, a3↦a1}	{a1 ↦ a3, a3 ↦ nil }
rmv1(a1) ⇓				
	{a3}	a3	{a3↦nil}	{a3↦nil}
rmv2(a3) ⇓				
	∅	nil	∅	∅

The two models (initial and refined) required 33 proof obligations, all proved automatically except 4 of them that were proved interactively.

4 Refinement Strategy

In this section we explain the order with which the requirements are taken into account in the successive formal models.

4.1 Principles Followed in this Development

In this development we follow a *modular approach*, summarised as follows:

1. We first define carefully the *static state* of the memory and its blocks, that is: the memory of a block is contiguous, blocks cover completely the entire memory, blocks do not overlap, there are no empty blocks, free blocks are not surrounded by other free blocks, etc.
2. We then define some *basic operations* on the memory (see sect. 5.1). These operations are the following: make_free, remove_from_free, reduce_create, and merge_right. They modify the memory in a way that is convenient for the complete definition and development of the allocating and freeing operations defined in item 3 below. They are proved to maintain the static properties of the state as defined in item 1 above.

3. Finally, we define the *two main operations* for allocating and freeing blocks in the memory, These operations have *no access* to the state of the memory, they can only *call the basic operations* defined in item 2 above.

This modular structure is illustrated in Fig. 9.

4.2 The Initial Model to the Fifth Refinement: Basic Operations

In the initial model, we first introduce the state of the memory. The main data structure for a block is defined by means of two functions recording its size and its right neighbour, and a set recording whether this block is free. Notice again that the "name" of a block is its first address.

Four basic operations are defined in the initial model. As mentioned in the previous section, these operations are proved to preserve the main properties of the state of the memory. Here are some informal definitions of these operations:

- The operation make_free takes a busy block as its parameter and transform it into a free block.
- The operation remove_from_free takes a free block as its parameter and transforms it into a busy block.
- The operation reduce_create takes a busy block b as its parameter together with a positive natural number q supposed to be *strictly smaller* than the size of b. This operation splits b into two blocks b and c. The new block b is the old block b reduced to the size q. The remaining part of the old block b is the new block c.
- The operation merge_right takes a busy block b as its parameter. Moreover, the right neighbour c of block b is supposed to be a busy block as well. The effect of this operation is to merge the two blocks (forming a new busy block b) thus removing completely the old busy block c.

The refinements from the first to the fifth introduce the doubly linked lists (as explained in sect. 3) and some minor details.

4.3 The Sixth and Seventh Refinement: Allocate and Free.

These refinements introduce "calls" made by the allocating or freeing operations to the previous basic operations.

4.4 The Eighth to the Tenth Refinement: Search Algorithm.

In these refinements, we add more data and events in order to develop the search of an available space to built a new block from some free ones. We take account of requirement FUN-14 defining the way size intervals of free blocks are recorded. A two dimensional data structure is used to organise the size intervals of groups.

4.5 Refinement Strategy Synthesis

In this section, we show carefully in which refinement (or in the initial model) the various requirements are considered. It is summarised in Table 2 where FUN denotes functional requirements and ENV denotes environment assumptions. The purpose of this synthesis is thus to show the mapping between requirements and refinements and to make sure that all requirements have been taken into account.

Table 2. Mapping Initial model and the eight refinements to the labels of requirements

	Initial	First	Second	Third	Forth	Fifth	Sixth-Seventh	Eighth-Tenth
FUN-	1,2(p), 4,11	3	12(p),13	15	12	2	5,6,7,9,10	8,14
ENV-	1-8	-	-	-	-	-	-	-

(p) means the related requirement is PARTIALLY taken into account.

5 Formal Models

The formal developments follow the principles of the modular approach explained in sect. 4.1. Below is a short outline presenting the new techniques used in the following formal models.

1. *Modular construction*: we first define the basic operations on the memory from the initial model to the fifth refinement (sect. 5.1 to 5.6). We then define the two main external operations (allocating and freeing) in sect. 5.7. Such external operations can only call the basic operations .
2. *Doubly linked list*: the independent model of a dynamic doubly linked list has already been introduced in sect. 3. It is used as a design pattern in the formal development of sect. 5.4.
3. *Pre-condition* and *sequential actions*: Event-B makes a heavy usage of events defined by means of guards and parallel actions (instead of pre-conditions and sequential actions). However, the dynamic part of classical programs is always defined by means of operations dealing with pre-conditions and sequential actions. Sect. 5.7 shows how to use events to formally develop such classical operations.
4. *Code generation*: the new technique of code generation is presented in sect. 5.7, where it explains how to handle sequential programs and modular organisation in code generation.
5. *Search algorithm*: the formal model of TLSF search algorithm is introduced in sect. 5.8.

5.1 Initial Model: Construction of Basic Operations

The State. In this initial model, we first construct the state of the memory. Notice again that the "name" of a block is its first address. The main data structure for a block is defined by means of its *size* and *right* neighbour. We also define the set of *free* blocks.

To begin with, we define the positive constant m denoting the size of the memory (from 1 to m). However, the memory starts at address 0 and ends at address $m+1$: we shall see that blocks 0 and $m+1$ are convenient dummy busy blocks that are never touched. Here are the invariants of *size*:

inv1: $size \in 0\,..\,m+1 \nrightarrow 1\,..\,m$	**inv3:** $size(0) = 1$
inv2: $\{0, m+1\} \subseteq dom(size)$	**inv4:** $size(m+1) = 1$

Notice that the domain of *size* defines the set of blocks. Also notice that the *size* of a block is a positive number (ranging from 1 to m): therefore, there are no empty blocks (**inv1**). The two dummy blocks 0 and $m+1$ have both a *size* equal to 1 (**inv2**, **inv3**, and **inv4**). Here are the invariants for *right*:

$$\begin{array}{ll}
\textbf{inv5:} & right \in 0 \mathbin{..} m \mathrel{+\!\!\!\!\rightarrow} 1 \mathbin{..} m+1 \\
\textbf{inv6:} & dom(right) = dom(size) \setminus \{m+1\} \\
\textbf{inv7:} & \forall b \cdot b \in dom(right) \;\Rightarrow\; right(b) = b + size(b) \\
\textbf{inv8:} & \forall b, c \cdot \; b \in dom(size) \land c \in dom(size) \land b \neq c \\
 & \qquad \Rightarrow\; (c \mathbin{..} c + size(c) - 1) \cap (b \mathbin{..} b + size(b) - 1) = \varnothing
\end{array}$$

Every block, except block $m+1$, has a right neighbour and block 0 is not the right neighbour of a block (**inv5** and **inv6**). Notice the simple definition of $right(b)$ (**inv7**): we add the size of the block b to the name, b, of this block (its first address). Finally, we state that blocks do not overlap (**inv8**). From these invariants, we can prove the following three important theorems:

$$\begin{array}{ll}
\textbf{thm1:} & (\bigcup i \cdot i \in dom(size) \mid i \mathbin{..} i + size(i) - 1) = 0 \mathbin{..} m+1 \\
\textbf{thm2:} & \forall b, c \cdot b \in dom(right) \land c \in dom(right) \land right(b) = right(c) \;\Rightarrow\; b = c \\
\textbf{thm3:} & right \in 0 \mathbin{..} m \mathrel{\rightarrowtail\!\!\!\!\!\!\!+\;\;} 1 \mathbin{..} m+1
\end{array}$$

Theorem **thm1** states that blocks cover the entire memory. Invariant **inv8** allows us to prove that $right$ is an injective (one-one) function (**thm2** and **thm3**). Since $right$ is an injective function, we can refer to $right^{-1}(b)$ when it is well defined. Here are the invariants for $free$:

$$\begin{array}{ll}
\textbf{inv9:} & free \subseteq dom(size) \\
\textbf{inv10:} & 0 \notin free \\
\textbf{inv11:} & m+1 \notin free
\end{array}$$

$$\begin{array}{ll}
\textbf{inv12:} & \forall b \cdot b \in free \\
 & \quad \Rightarrow \\
 & \quad right(b) \notin free \land right^{-1}(b) \notin free
\end{array}$$

We first state that $free$ blocks are indeed blocks (**inv9**) and then that dummy blocks 0 and $m+1$ are not free blocks (**inv10** and **inv11**). Finally, we state the very important property of free blocks: they cannot be surrounded by free blocks (**inv12**).

The Events. Initially, we have a single free block: this is block 1 of size m. Blocks 0 and $m+1$ are dummy busy blocks of size 1. The right block of block 0 is block 1 and the right block of block 1 is block $m+1$. So, the initialisation is defined as indicated. We introduce the four operations mentioned in sect. 5.1. The operation make_free and remove_from_free are used to handle free blocks.

```
INITIALISATION ≙
 begin
   $free := \{1\}$
   $size := \{0 \mapsto 1, 1 \mapsto m,$
            $m+1 \mapsto 1\}$
   $right := \{0 \mapsto 1,$
             $1 \mapsto m+1\}$
 end
```

```
make_free ≙
 any
   $b$
 where
   $b \in dom(size) \setminus free$
   $b \notin \{0, m+1\}$
 then
   $free := free \cup \{b\}$
 end
```

```
remove_from_free ≙
 any
   $b$
 where
   $b \in free$
 then
   $free := free \setminus \{b\}$
 end
```

The operation reduce_create is used to reduce the size of a block, whereas the operation merge_right is used to merge a block with its right neighbour.

```
reduce_create $\widehat{=}$
  any
    $b$
    $q$
  where
    $b \in dom(size) \setminus free$
    $q < size(b)$
    $q > 0$
    $b \notin \{0, m+1\}$
  then
    $size := (\{b\} \lhd size) \; \cup \; \{b \mapsto q\}$
          $\cup \; \{b + q \mapsto size(b) - q\}$
    $right := (\{b\} \lhd right) \; \cup$
          $\{b \mapsto b + q\} \cup \{b + q \mapsto right(b)\}$
  end
```

```
merge_right $\widehat{=}$
  any
    $b$
  where
    $b \in dom(size) \setminus free$
    $b \notin \{0, m+1\}$
    $rb \notin free$
    $rb \notin \{0, m+1\}$
  then
    $size := (\{rb, b\} \lhd size) \; \cup$
          $\{b \mapsto size(b) + size(rb)\}$
    $right := (\{rb\} \lhd right \rhd \{rb\})$
          $\cup \{b \mapsto right(rb)\}$
  end
```

where rb stands for $right(b)$)

5.2 First Refinement: Gluing Invariant Between *left* and *right*

The State. In this very simple refinement we introduce a new variable $left$, defined as $right^{-1}$ as indicated in the following invariant:

$$\boxed{\textbf{inv1:} \quad left = right^{-1}}$$

The variable $right$ of previous model is removed and replaced by $left$. In [18], the state used in the data structure of block is $left$, however, since $right(b) = b + size(b)$, the introduction of $right$ makes the proofs and modeling simpler. This is why we first introduce $right$ in the initial model and prove that it is injective. Then $right$ is replaced by $left$ in this refinement.

The Refined Events. The related event reduce_create and merge_right are refined by replacing the assignment to $right$ by that to $left$:

```
reduce_create $\widehat{=}$
  ...
  then
    ...
    $size := (\{b\} \lhd size) \; \cup \; \{b \mapsto q\}$
          $\cup \; \{b + q \mapsto size(b) - q\}$
    $left := (\{left^{-1}(b)\} \lhd left) \; \cup$
          $\{b + q \mapsto b\} \; \cup \; \{left^{-1}(b) \mapsto b + q\}$
  end
```

```
merge_right $\widehat{=}$
  ...
  then
    ...
    $size := (\{lb, b\} \lhd size) \; \cup$
          $\{b \mapsto size(b) + size(lb)\}$
    $left := (\{lb\} \lhd left \rhd \{lb\}) \; \cup$
          $\{left^{-1}(lb) \mapsto b\}$
  end
```

where lb stands for $left^{-1}(b)$. Notice that we have:
$lb = b + size(b), \qquad left^{-1}(lb) = b + size(b) + size(b + size(b))$

5.3 Second Refinement: Size Groups of Blocks

The State. This refinement takes care of the requirement that each free block belongs to a block size group. We have a fixed number, d, of group (**axm1**). The mapping linking each block size with a group is defined by means of a constant function g (**axm2**). The group corresponding to the biggest size m is d (**axm3**).

axm1: $d \in \mathbb{N}_1$
axm2: $g \in 1 \,..\, m \twoheadrightarrow 1 \,..\, d$
axm3: $d = g(m)$

inv1: $box \in free \rightarrow 1 \,..\, d$
inv2: $\forall b \cdot b \in free \;\Rightarrow\; box(b) = g(size(b))$

A new variable, *box*, is introduced (**inv1**). The function *box* maps each free block with the group corresponding to its size (**inv2**).

The Refined Events. The variable *box* is initialized to $\{1 \mapsto d\}$ (since initially, the unique free block is block 1 whose size is m). The two operations dealing with *box* are refined accordingly:

```
make_free ≙
  any
    $b$
  where
    $b \in dom(size) \setminus free$
    $b \notin \{0, m+1\}$
    $left^{-1}(b) \notin free$
    $left(b) \notin free$
  then
    $free := free \cup \{b\}$
    $\underline{box(b) := g(size(b))}$
  end
```

```
remove_from_free ≙
  any
    $b$
  where
    $b \in free$
  then
    $free := free \setminus \{b\}$
    $\underline{box := \{b\} \lhd\!\!\!- box}$
  end
```

5.4 Third Refinements: Doubly Linked List

There are two third refinements. Here, we promote the construction of the doubly linked list as defined in sect. 3 (it had an initial model and a refinement). We introduce the variable *sigma*. It handles, for each group, the set of free blocks stored in that group. It generalises the variable s defined in sect. 3. So, we have the following invariant:

$$\forall i \cdot i \in 1 \,..\, d \;\Rightarrow\; sigma(i) = box^{-1}[\{i\}]$$

We also introduce the variable f. It handles, for each group, the head of the corresponding list. It generalises the variable f defined in sect. 3. Finally, we define both variables *nx* and *pr* generalising similar variables defined in sect. 3. We update accordingly events make_free and remove_from_free. The two other events are not modified.

5.5 Fourth Refinement

In this simple refinement, we remove variables *sigma* and *box* since they do not appear in guards. They were introduced in previous refinement to define the invariants inherited from those defined in the little independent development of sect. 3.

5.6 Fifth Refinement: Variable $free\_bit$

In this simple refinement, we introduce the variable $free\_bit$ for the main data structure of blocks. It is a boolean value associated with each block: TRUE when the block is free and FALSE otherwise

inv1: $free\_bit \in dom(size) \to BOOL$
inv2: $\forall b \cdot b \in dom(size) \Rightarrow free\_bit(b) = \text{bool}(b \in free)$

It is initialized to $free\_bit := \{0 \mapsto FALSE, 1 \mapsto TRUE, m+1 \mapsto FALSE\}$. All the four operations surround this variable. They are refined as below:

```
make_free ≙
  status
    $ordinary$
  refine
    $make\_free$
  any
    $b$
  where
    $b \in dom(size)$
    $b \notin \{0, m+1\}$
    $free\_bit(left^{-1}(b)) = FALSE$
    $free\_bit(left(b)) = FALSE$
    $free\_bit(b) = FALSE$
  then
    $free\_bit(b) := TRUE$
    $nx(g0)) := (nx(g0) \lhd\!\!\!- \{(nx(g0))^{-1}(ls(g0)) \mapsto b\}) \cup \{b \mapsto ls(g0)\}$
  end
```

```
remove_from_free ≙
  status
    $ordinary$
  refine
    $remove\_from\_free$
  any
    $b$
  where
    $b \in dom(size)$
    $free\_bit(b) = TRUE$
  then
    $free\_bit(b) := FALSE$
    $nx(g0) := (\{b\} \lhd nx(g0) \rhd \{b\}) \cup \{(nx(g0))^{-1}(b) \mapsto nx(g0)(b)\}$
  end
```

where $g0 = g(size(b)$

```
reduce_create ≙
  status
    $ordinary$
  refine
    $reduce\_create$
  any
    $b$
    $q$
  where
    $b \in dom(size)$
    $q < size(b)$
    $q > 0$
    $b \notin \{0, m+1\}$
    $free\_bit(b) = FALSE$
  then
    $size := (\{b\} \lhd size) \cup \{b \mapsto q\} \cup \{b+q \mapsto size(b) - q\}$
    $left := (\{left^{-1}(b)\} \lhd left) \cup \{b+q \mapsto b\} \cup \{left^{-1}(b) \mapsto b+q\}$
    $free\_bit(b+q) := FALSE$
  end
```

```
merge_right ≙
  status
    $ordinary$
  refine
    $merge\_right$
  any
    $b$
  where
    $b \in dom(size)$
    $b \notin \{0, m+1\}$
    $left^{-1}(b) \notin \{0, m+1\}$
    $free\_bit(left^{-1}(b)) = FALSE$
    $free\_bit(b) = FALSE$
  then
    $size := (\{left^{-1}(b), b\} \lhd size) \cup \{b \mapsto size(b) + size(left^{-1}(b))\}$
    $left := (\{left^{-1}(b)\} \lhd left \rhd \{left^{-1}(b)\}) \cup \{left^{-1}(left^{-1}(b)) \mapsto b\}$
    $free\_bit := \{left^{-1}(b)\} \lhd free\_bit$
  end
```

5.7 Sixth to Tenth Refinements: Final Model and Code Generation

In previous refinements, we constructed our four basic operations: make_free, remove_from_free, reduce_create, and merge_right. Now we define two operations for allocating and freeing blocks in the memory. These two operations have no access to the state of the memory. They can only call the four basic operations as shown in Fig. 9. In order to achieve this, the four basic operations defined as guarded events have to be transformed into pre-conditioned operations. We follow the approach explained in the following subsection.

Transforming Guarded Events into Pre-Conditioned Operations. Here are some rules for the transformation.

1. We define an enumerated set P containing *call* and *return* values for each basic operation, and an $undefined$ value as well:

$$\begin{aligned} P = \{ & call\_makefree, \quad return\_make\_free, \\ & \ldots \\ & call\_merge\_right, \quad return\_merge\_right, \\ & undefined\} \end{aligned}$$

2. We have a new variable *prog* of type P and initialised to $undefined$. When an event wants to "call" a basic operation, it assigns *prog* to the corresponding call. The mention of this call is made in the guard of the basic operation (this is illustrated in the event make_free below and in others as well). The event corresponding to the basic operation assigns then *prog* to the corresponding return value (again, it is illustrated below in the event make_free and others as well). This new value of *prog* is used in the guard of the event supposed to take control after the return from the called basic operation.
3. Notice that the events corresponding to the basic operations have *no guards* anymore (except the call value). This means that the guards we had in previous refinements for these operations are now mere theorems that are proved by means of some invariants: we have indeed transformed these guards into pre-conditions that must be proved before the call.

This technique will be explained in details in a coming paper to be published later. However, this technique has already been presented by means of slides during the last ABZ Conference (2014) in the Event-B Workshop [4].

Final Construction of the Four Basic operations. The Event-B models presented in this subsection show how to construct basic operations which can be called by other events. It follows the rule of how to transform guarded events into pre-conditioned operations presented in previous subsection.

The final versions of these operations in Event-B are given as indicated below. Each of them is presented together with the related C code. An *important remark* about these C codes is made in sect. 5.9.

The make_free *Operation.* This event now has been transformed into a pre-conditioned operation: it has no guards anymore except the call value.

As can be seen, the C code for make_free event is clearly very close to the event itself. However, a very special attention must be made in this translation: the "parallel statements" of the events are transformed into "sequential statements" in the C code. We have thus to be sure that this can be done in a straightforward way in this case, that is without introducing some local variables or some particular ordering (we shall see that in operations reduce_create and merge_right some special ordering are needed for the C code). It is clearly something that could have been analysed and decided automatically. Note the way we made the hand translation of the various assignment. In the C code the variables "free_bit", "size", etc. are all arrays whereas in the Event-B refinement they are functions. Clearly some translation rules are needed here to justified what we have done: this will be systematically developed in some future work.

```
make_free  ≙
 when
   $prog = call\_make\_free$
 then
   $free\_bit(b) := TRUE$
   $f(g(size(b))) := b$
   $next := next \cup \{b \mapsto a\}$
   $prev := (\{a\} \lhd prev) \ \cup$
             $(\{-1\} \lhd (\{a \mapsto b, b \mapsto -1\}))$
   $prog := return\_make\_free$
 end
```

```c
int make_free(int b){
    int a=f[g[size[b]]];
    free_bit[b] = TRUE;
    f[g[size[b]]] = b;
    next[b]=a;
    prev[b]=-1;
    if (a!=-1) prev[a] = b;
    return(0);
}
```

where (in event make_free:)

$$b = \texttt{b\_mf}, \quad next = nx(g(size(b))), \quad a = f(g(size(b))), \quad prev = pr(g(size(b)))$$

The remove_from_free *Operation.* There are two events associated with the operation remove_from_free. They correspond to the expression $f(g(size(b))$ being equal to or not equal to b. In the code below this will impact the assignment to $next(prev(b))$ and that to $f(g(size(b)))$. The same remark as done for the previous operation applies here as well: we have to make clearer the rules we need to perform the translation of the various assignments.

```
remove_from_free_1  ≙
 when
   $prog = call\_remove\_from\_free$
   $f(g(size(b))) \neq b$
 then
   $free\_bit(b) := FALSE$
   $next := (\{b\} \lhd next \rhd \{b\}) \cup$
            $\{(prev(b) \mapsto next(b)\}$
   $prev := (\{b\} \lhd prev \rhd \{b\}) \cup$
            $(\{-1\} \lhd \{next(b) \mapsto prev(b)\})$
   $prog := return\_remove\_from\_free$
 end
```

```
remove_from_free_2  ≙
 when
   $prog = call\_remove\_from\_free$
   $f(g(size(b))) = b$
 then
   $free\_bit(b) := FALSE$
   $next := \{b\} \lhd next$
   $f(g(size(b))) := next(b)$
   $prev := (\{b, next(b)\} \lhd prev) \cup$
            $(\{-1\} \lhd \{next(b) \mapsto prev(b)\})$
   $prog := return\_remove\_from\_free$
 end
```

where: $b = \texttt{b\_rff}$, $next = nx(g(size(b)))$, $prev = pr(g(size(b)))$

```c
int remove_from_free(int b){
    free_bit[b] = FALSE;
    if (f[g[size[b]]]!=b)
        next[prev[b]]=next[b];
    else
        f[g[size[b]]]=next[b];
    if (next[b]!=NIL) prev[next[b]]=prev[b];
    next[b] = -1;
    prev[b] = -1;
    return(0); }
```

The reduce_create *and* merge_right *Operation.* In the C code, notice that the assignment to $size[b]$ is put AFTER that of $size[b+q]$. This is because the assignment to $size[b+q]$ use the value of $size[b]$.

```
reduce_create ≙
  when
    $prog = call\_reduce\_create$
  then
    $size := (\{b\} \lhd size) \cup \{b \mapsto q\} \cup$
             $\{b+q \mapsto size(b) - q\}$
    $left := (\{b + size(b))\} \lhd left) \cup$
             $\{b+q \mapsto b\} \cup$
             $\{b + size(b) \mapsto b+q\}$
    $free\_bit(b+q) := FALSE$
    $prog := return\_reduce\_create$
  end
```

```
int reduce_create(int b, int q)
{   size[b+q]=size[b]-q;
    left[b+q]=b;
    free_bit[b+q]=FALSE;
    left[b+size[b]]=b+q;
    size[b]=q;
    return(0);
}
```

where (in the event reduce_create): $b = \texttt{b\_rc}$, $b + size(b) = left^{-1}(b)$, $q = \texttt{q\_rc}$.
The merge_right Operation is similar.
The merge_right *Operation.* In the C code, we have the same remark as the one done for the previous operation. The assignment to $size[b+s]$ has to be put AFTER those to $size[b]$ and $left[b+s+size[b+s]]$.

```
merge_right ≙
  when
    $prog = call\_merge\_right$
  then
    $size := (\{b+s, b\} \lhd size) \cup$
             $\{b \mapsto s + size(b+s)\}$
    $left := (\{b+s\} \lhd left \rhd \{b+s)\}) \cup$
             $\{b+s+size(b+s) \mapsto b\}$
    $free\_bit := \{b+s\} \lhd free\_bit$
    $prog := return\_merge\_right$
  end
```

```
 int merge_right(int b){
    int s;
    s=size[b];
    left[b+s+size[b+s]]=b;
    size[b]=s+size[b+s];
    size[b+s]=-1;
    left[b+s]=-1;
    free_bit[b+s]=-1;
    return(0);
}
```

where (in the event merge_right)
$b = \texttt{b\_mr}$, $b+s = left^{-1}(b)$, $s = size(b)$, $b+s+size(b+s) = left^{-1}(left^{-1}(b))$

Allocating and Freeing. The events below show how to construct events which can call the basic operations. It follows the rules of the transformation as explained in previous subsection. These events also show how to construct sequential process by events using an address (*adrp* in the model).

Here we have two groups of events of allocation: first, events allocate_1_1 to allocate_1_4, and second, events allocate_2_1 and allocate_2_2. The first group corresponds to the required quantity of memory, q_loc_0, being strictly smaller than the $size$ of the chosen free bloc, $bloc\_0$. The second group deals with the case where the required quantity of memory is exactly equal to the size of the chosen block. The events below and related C code are explained as follows:

1. Each of these events calls some basic operations by means of the assignment "$prog := call\_...$" and possibly returns from some basic operation by means of the guard "$prog = return\_...$".

2. Each event works with an address counter *adrp* which makes the sequential translation easy: we just follow the successive values of the address counter to translate the calls sequentially.

```
allocate_1_1 ≙
 when
   $adrp = 1$
   $bloc\_0 \neq -1$
   $q\_loc\_0 < size(bloc\_0)$
 then
   $prog := \texttt{call\_rff}$
   $\texttt{b\_rff} := bloc\_0$
   $\texttt{q\_loc} := q\_loc\_0$
   $adrp := 3$
 end
```

```
allocate_1_2 ≙
 when
   $prog = \texttt{return\_rff}$
   $adrp = 3$
 then
   $prog := \texttt{call\_rc}$
   $\texttt{b\_rc} := \texttt{b\_rff}$
   $\texttt{q\_rc} := \texttt{q\_loc}$
   $adrp := 4$
 end
```

```
allocate_1_3 ≙
 when
   $prog = \texttt{return\_rc}$
   $adrp = 4$
 then
   $prog := \texttt{call\_mf}$
   $\texttt{b\_mf} := \texttt{b\_rc} + \texttt{q\_loc}$
   $adrp := 5$
 end
```

```
allocate_1_4 ≙
 when
   $prog = \texttt{return\_mf}$
   $adrp = 5$
 then
   $prog := undefined$
   $q\_loc\_0 := 0$
   $bloc\_0 := -1$
   $adrp := 0$
 end
```

```
allocate_2_1 ≙
 when
   $adrp = 1$
   $bloc\_0 \neq -1$
   $q\_loc\_0 = size(bloc\_0)$
 then
   $prog := \texttt{call\_rff}$
   $\texttt{b\_rff} := bloc\_0$
   $adrp := 2$
 end
```

```
allocate_2_2 ≙
 when
   $prog = \texttt{return\_rff}$
   $adrp = 2$
 then
   $prog := undefined$
   $q\_loc\_0 := 0$
   $bloc\_0 := -1$
   $adrp := 0$
 end
```

where (in allocate events),
  $\texttt{rff} = \texttt{remove\_from\_free}$, $\texttt{rc} = \texttt{reduce\_create}$, $\texttt{mf} = \texttt{make\_free}$.
In these events, $bloc\_0 \neq -1$ denotes the success of searching a free block. The model of search is shown in next subsection.

For the free operation, we have a similar situation as the one seen in the operation allocate. The translation uses the same technique as for allocate.

```c
int allocate_mem(int q){
  int bloc_0=search(q);
  if (bloc_0!=-1) {
    if (q<size[bloc_0]){
        remove_from_free(bloc_0);
        reduce_create(bloc_0, q);
        make_free(bloc_0+q);
    }else if (q==size[bloc_0]){
        remove_from_free(bloc_0);
    }
    printf("SUCCESS");
  }
  return(0);
}
```

The free Operation. For the free operation, we have a similar situation as the one seen in the operation allocate. The translation uses the same technique as before for allocate.

```
free_1_1 $\widehat{=}$
any
   $b$
where
   $b \in dom(size)$
   $free\_bit(b) = FALSE$
   $b \notin \{0, m+1\}$
   $left(b) \in dom(size)$
   $free\_bit(left(b)) = TRUE$
   $adrp = 0$
then
   $prog :=$ call_rff
   b_rff $:= left(b)$
   $bloc := b$
   $adrp := 6$
end
```

```
free_1_2 $\widehat{=}$
when
   $prog =$ return_rff
   $adrp = 6$
then
   $prog :=$ call_mr
   b_mr $:= left(bloc)$
   $bloc := left(bloc)$
   $adrp := 7$
end
```

```
free_1_3 $\widehat{=}$
when
   $prog =$ return_mr
   $adrp = 7$
then
   $prog := undefined$
   $adrp := 8$
end
```

```
free_2 $\widehat{=}$
   any
      $b$
   where
      $b \in dom(size)$
      $free\_bit(b) = FALSE$
      $b \notin \{0, m+1\}$
      $left(b) \in dom(size)$
      $free\_bit(left(b)) = FALSE$
      $adrp = 0$
   then
      $bloc := b$
      $adrp := 8$
   end
```

```
free_3_1 $\widehat{=}$
   when
      $adrp = 8$
      $left^{-1}(bloc) = bloc + size(bloc)$
      $free\_bit(bloc + size(bloc)) = TRUE$
   then
      $prog :=$ call_rff
      b_rff $:= bloc + size(bloc)$
      $adrp := 9$
   end
```

```
free_3_2 $\widehat{=}$
when
   $prog =$ return_rff
   $adrp = 9$
then
   $prog :=$ call_mr
   b_mr $:= bloc$
   $adrp := 10$
end
```

```
free_3_3 $\widehat{=}$
when
   $prog =$ return_mr
   $adrp = 10$
then
   $prog := undefined$
   $adrp := 11$
end
```

```
free_4 $\widehat{=}$
when
   $adpr = 8$
   $left^{-1}(bloc) = bloc + size(bloc)$
   $bloc + size(bloc) \in dom(size)$
   $free\_bit(bloc + size(bloc))$
$= FALSE$
then
   $adrp := 11$
end
```

```
free_5 $\widehat{=}$
   when
      $adrp = 11$
   then
      $prog :=$ call_mf
      b_mf $:= bloc$
      $adrp := 12$
   end
```

```
free_6 $\widehat{=}$
   when
      $prog =$ return_mf
      $adrp = 12$
   then
      $prog := undefined$
      $adrp := 0$
   end
```

```
int free_mem(int b){
    int bloc;
        if (size[b]!=-1 && free_bit[b]==0 && b!=0 && b!=m+1
                                                && size[left[b]]!=-1) {
        if (free_bit[left[b]]==TRUE) {
            bloc=b;
            remove_from_free(left[b]);
            bloc = left[bloc];
            merge_right(bloc);
        }else{
            bloc=b;
        }
        if (free_bit[bloc+size[bloc]]==TRUE) {
            remove_from_free(bloc+size[bloc]);
            merge_right(bloc);
        }
        make_free(bloc);
        printf("SUCCESS");
    }else{
        printf("FAILURE");
    }
    return(0);
}
```

5.8 The search Operation

In the eighth refinement, we define two constants *upper* and *lower* defining the limit of the sizes of each group as follows:

axm1:	$lower \in 1\,..\,d \rightarrow 1\,..\,m$	**axm3:**	$lower(1) = 1$
axm2:	$upper \in 1\,..\,d \rightarrow 1\,..\,m$	**axm4:**	$upper(d) = m$
axm5:	$\forall i \cdot i \in 1\,..\,d \;\Rightarrow\; g^{-1}[\{i\}] = lower(i)\,..\,upper(i)$		
axm6:	$\forall i \cdot i \in 1\,..\,d-1 \;\Rightarrow\; lower(i+1) = upper(i)+1$		

In this abstraction, the groups (from 1 to d) are defined in one dimension. As can be seen, the sizes assigned to the various groups partition the range of group sizes (from 1 to m). There are two functions to assign a size to a group: $g(q)$ assigns the size q to the group where free block of size q are stored; $g\_srh(q)$ assigns the size q to the group where the search for a block of size q should start from.

axm7: $g \in 1\,..\,m \rightarrow 1\,..\,d$
axm8: $g\_srh \in 1\,..\,m \rightarrow 1\,..\,d$
axm9: $\forall q \cdot q \in 1\,..\,m \;\Rightarrow\; g(q) = min(\{i | i \in 1\,..\,d \wedge q \leq upper(i)\})$
axm10: $\forall q \cdot q \in 1\,..\,lower(d) \;\Rightarrow\; g\_srh(q) = min(\{i | i \in 1\,..\,d \wedge q \leq lower(i)\})$
axm11: $\forall q \cdot q \in 1\,..\,lower(d) \;\Rightarrow\; q \leq lower(g\_srh(q))$
axm12: $\forall q, j \cdot q \in 1\,..\,lower(d) \wedge j \in 1\,..\,d \wedge g\_srh(q) < j \;\Rightarrow\; q < lower(j)$

Example. Suppose groups i and $i+1$ relates to the size interval $[52, 55]$ and $[56, 59]$ respectively. A block with size 54 is located in the group holding intervals $[52, 55]$. This is done using the function g. But if a block of size 54 is requested, the search using function g_srh has to be used in order to locate a block in the interval $[56, 59]$. This is because if the search is done in the group $[52, 56]$, blocks of sizes 52 or 53 have to be discarded. Therefore, in order to guarantee that the search is made without loop, it is started from the group where the lower bound is as large as the requested size.

Refinement. A two-dimensional data structure is introduced in this refinement in order to locate free blocks faster than with a one dimensional structure. This data structure is explained as in Fig. 10. Note that constants mf and ms are the length of the first dimension and second dimension respectively. Functions $fl(q)$ and $ft(q)$ get the first level and second level index of size q (size between 0 and m), respectively.

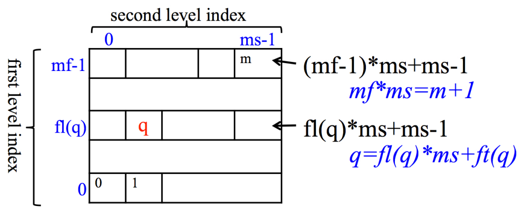


Fig. 10: Two Dimension for Groups

$g\_srh(q)$ is implemented by $g(nxt(q))$ where $\forall q \cdot q \in 1 \,..\, m \Rightarrow nxt(q) = q + ms - 1$. If a requested size is q_0, it is used to locate to the group $g(q)$ (where $q = nxt(q_0)$). There are two cases for the search to succeed.

Success Case 1. In this case, the group j containing available free block is found when searching the blocks between group $g(q)$ and the last group of the first level $fl(q)$. The guard below is thus included in this event:

$$\{i | i \in g(q) \,..\, g(fl(q) * ms + ms - 1) \wedge f(i) \neq -1\} \neq \varnothing$$

In this case, the founded group j for the requested size is the following:

$$j = min(\{i | i \in g(q) \,..\, g(fl(q) * ms + ms - 1) \wedge f(i) \neq -1\})$$

Success Case 2. In this case, in the first level of $fl(q)$, no free block can be found. The founded group j has a greater first level index than $fl(q)$. The guards below are included in this event:

$$\{i | i \in g(q) \,..\, g(fl(q) * ms + ms - 1) \wedge f(i) \neq -1\} = \varnothing$$

$$\{r | r \in fl(q) + 1 \,..\, mf - 1 \wedge \{i | i \in g(r * ms) \,..\, g(r * ms + ms - 1) \wedge f(i) \neq -1\} \neq \varnothing\} \neq \varnothing$$

In this case, the founded group j for the requested size is as below:

$$k = min(\{r | r \in fl(q) + 1 .. mf - 1 \wedge \{i | i \in g(r * ms) .. g(r * ms + ms - 1) \wedge f(i) \neq -1\} \neq \varnothing\})$$

$$j = min(\{i | i \in g(k * ms) \,..\, g(k * ms + ms - 1) \wedge f(i) \neq -1\})$$

The explanation for O(1) (no loop) is now very simple. As can be seen, in both cases above, we use various occurrences of the min operator to determine the value of the concerned block j. If the value of mf and ms are correctly determined then implementations of these min operators can be done by means of single *machine instructions*. In fact, these machine instructions are able to perform the loops involved in the min operators: this is the important idea proposed in the TLSF paper [18].

Note that the C code we provide does not implement the complete system as described in [18]. The O(1) search operation described in sect. 5.8 is part of our Rodin formal development but not part of the code. Some further refinements would be necessary in order to perform the complete O(1) achievement in the generated C code.

5.9 Discussion about the C code

The C-Codes presented in the previous section for the basic operations and the external operations are not the final codes one could find in a real operating system as presented in Fig. 11. In fact, we used different arrays for handling the $size$, $left$, $free\_bit$, nx, and pr "fields" of each block. In a real program for these fields, we could have used a C "structure types" and also some C "pointer types" for nx and pr. We preferred to present the C code in the form we used in this paper because it is simpler to read and also closer to the Event-B formal development.

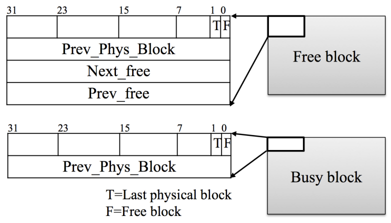
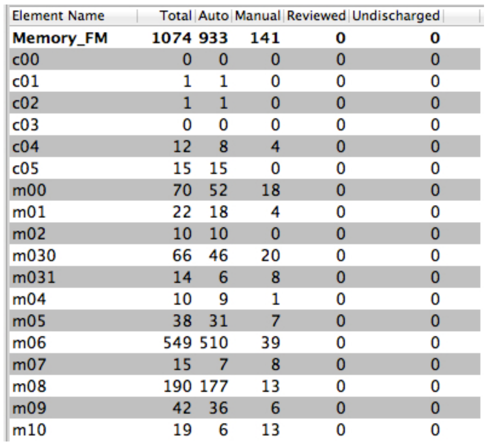


Fig. 11: Structure of Blocks in [18]

The transformation of our C codes to more concrete ones must be done by means of further data refinements (and proofs) on the Event-B development. In other words, we insist to perform these ultimate C code developments by using Event-B because in performing these transformations directly on the present C code to the more concrete one, we could introduce some bugs.

5.10 Statistics.

The overall proof effort needed to develop this system with the Rodin Platform is 1074 proofs where 933 were proved automatically (87%) as shown in Fig 12. The remaining proofs were done interactively: they are not difficult, only a bit tedious. Besides the basic inference rules encoded in the Rodin platform, some provers such as SMT (CVC3 and veriT), Atelier P0, and Atelier B ML are also used. Moreover, model-checker and animator are used to complement proving as explained in [27, 28].



Element Name	Total	Auto	Manual	Reviewed	Undischarged
Memory_FM	**1074**	**933**	**141**	**0**	**0**
c00	0	0	0	0	0
c01	1	1	0	0	0
c02	1	1	0	0	0
c03	0	0	0	0	0
c04	12	8	4	0	0
c05	15	15	0	0	0
m00	70	52	18	0	0
m01	22	18	4	0	0
m02	10	10	0	0	0
m030	66	46	20	0	0
m031	14	6	8	0	0
m04	10	9	1	0	0
m05	38	31	7	0	0
m06	549	510	39	0	0
m07	15	7	8	0	0
m08	190	177	13	0	0
m09	42	36	6	0	0
m10	19	6	13	0	0

Fig. 12. Statistics of the Proofs

6 Conclusion

In this paper, we presented the formal development of a short but complex software system. In doing this, we propose various novel techniques for a modular software structure, corresponding code generation, and related Event-B approaches. Interested readers can access our Event-B and Rodin development and the C code as well in the web site [1]. Some further refinements would be necessary to obtain a final code.

References

1. *Rodin Development and Code for this Paper.* http://www.lab205.org/memory4fm.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
3. Jean-Raymond Abrial and Rustan Leino. Mini-course around Event-B and Rodin: hypervisor. http://research.microsoft.com/apps/video/default.aspx?id=151665, 2011.
4. Jean-Raymond Abrial and Wen Su. Transforming guarded events into preconditioned operations. http://wiki.event-b.org/images/Abrial2RUDW2014.pdf, 2014.
5. Amiga Operating System, 2014. http://www.amiga.com.
6. AnimB. *http://www.animb.org.*
7. R. J. R. Back and K. Sere. From action systems to modular systems, 1996.
8. Pontus Boström. Creating sequential programs from Event-B models. In *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, pages 74–88, 2010.
9. Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina A. Waldén. Derivation of concurrent programs by stepwise scheduling of Event-B models. *Formal Asp. Comput.*, 26(2):281–303, 2014.
10. Andrew Edmunds, Michael Butler, Issam Maamria, Renato Silva, and Chris Lovell. Event-B code generation: type extension with theories. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 365–368. Springer, 2012.
11. Andreas Fürst, Thai Son Hoang, David A. Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code generation for Event-B. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 323–338, 2014.
12. Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. *Software and System Modeling*, 12(2):229–244, 2013.
13. Gerwin Klein. Operating system verification an overview, February 2009.
14. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220, 2009.
15. Gerwin Klein and Harvey Tuch. Towards verified virtual memory in l4. In *TPHOLS EMERGING TRENDS'04, PARK CITY*, 2004.
16. Rafal Kolanski. A logic for virtual memory. *Electr. Notes Theor. Comput. Sci.*, 217:61–77, 2008.
17. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
18. M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *ECRTS 2004*, pages 79–88.
19. Miguel Masmano, Ismael Ripoll, Patricia Balbastre, and Alfons Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179, 2008.
20. Dominique Méry and Rosemary Monahan. Transforming event B models into verified c# implementations. In *First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, July 12-13, 2013*, pages 57–73, 2013.
21. Dominique Méry and Neeraj Kumar Singh. Automatic code generation from Event-B models. In *Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011*, pages 179–188, 2011.

22. Dominique Méry and Neeraj Kumar Singh. A generic framework: from modeling to code. *ISSE*, 7(4):227–235, 2011.
23. Open Robot Control Software, 2013. http://www.orocos.org.
24. Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 41–49. IEEE, 2002.
25. Rodin. *http://www.event-b.org/*.
26. Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
27. Wen Su and Jean-Raymond Abrial. Aircraft landing gear system: Approaches with Event-B to the modeling of an industrial system. In *ABZ 2014: The Landing Gear Case Study*. Springer, 2014.
28. Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Complementary methodologies for developing hybrid systems with Event-B. In *ICFEM*, pages 230–248, 2012.
29. TLSF, 2013. http://www.gii.upv.es/tlsf/main/used.
30. Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. From Event-B specifications to programs for distributed algorithms. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, June 17-20, 2013*, pages 104–109, 2013.
31. Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, pages 143–159, 2012.
32. Laurent Voisin and Jean-Raymond Abrial. The rodin platform has turned ten. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 1–8, 2014.
33. Steve Wright. Automatic generation of C from Event-B. In *Workshop on integration of model-based formal methods and tools*, page 14. Citeseer, 2009.
34. XtratuM: one kind of Hypervisors, 2014. http://www.xtratum.org.