SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

Formal Modelling of List Based Dynamic Memory Allocators

Bin Fang[1,2], Mihaela Sighireanu[2*], Geguang Pu[1*], Wen Su[3], Jean-Raymond Abrial[4], Mengfei Yang[5] & Lei Qiao[5]

[1] *Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai* 200062 *China;*
[2] *IRIF, Univ Paris Diderot and CNRS, Paris* 75013*, France;*
[3] *School of Computer Engineering and Science, Shanghai University, Shanghai, China;*
[4] *Marseille, France;*
[5] *Beijing Institute of Control Engineering, Beijing, China*

Abstract Existing implementations of dynamic memory allocators (DMA) employ a large spectrum of policies and techniques. The formal specifications of these techniques are quite complicated in isolation and very complex when combined. Therefore, the formal reasoning on a specific DMA implementation is difficult for automatic tools and mostly single-use. This paper proposes a solution to this problem by providing formal models for a full class of DMA, the class using various kinds of lists to manage the memory blocks controlled by the DMA. To obtain reusable formal models and tractable formal reasoning, we organise these models in a hierarchy ranked by refinement relations. We prove the soundness of models and the refinement relations using the modeling framework Event-B and the theorem prover Rodin. We demonstrate that our hierarchy is a basis for an algorithm theory for list based DMA: it abstracts various existing implementations of DMA and leads to new DMA implementations. The applications of this formalisation include model-based code generation, testing, and static analysis.

Keywords Dynamic memory allocators, Formal methods, Refinement, Event-B, Rodin, Model-based design

Citation Fang B, et al. Formal Modelling of List Based Dynamic Memory Allocators. Sci China Inf Sci, for review

1 Introduction

A dynamic memory allocator (DMA) is a piece of software managing a reserved region of the program memory. It appears in general purpose libraries (e.g., C standard library) or as part of applications where the dynamic allocation shall be controlled to avoid failure due to memory exhaustion (e.g., embedded critical software). A client program interacts with the DMA by requesting some amount of memory that it may free at any time. To offer this service, the DMA manages the reserved memory region by partitioning it into variable or fixed sized memory blocks, also called *chunks*. When a chunk is allocated to a client program, the DMA can not relocate it to compact the memory region (like in garbage collectors) and it is unaware about the kind (type or value) of data stored.

The existing implementations of DMA use various data structures to manage the set of chunks created in the memory region. In this paper, we focus on DMAs that record all chunks using a list, also called *heap list*. In this data structure, the chunks are stored in the increasing order of their start address and the successor relation between chunks is computed from some information stored in the start of the chunk, e.g., the size of the chunk. Notice that this data structures allows to manage both fixed or variable size chunks. To speed the allocation of a free chunk,

* Corresponding author (email: sighirea@irif.fr, ggpu@sei.ecnu.edu.cn)

DMA indexes the set of chunks not in use (*free chunks*) in an additional data structure. We focus here on *free list allocators* [15, 27] that record free chunks in a list. This class of list based DMA is widespread and it includes textbook examples [13, 15] and real-world allocators [16].

Providing DMAs which are optimal and formally proved correct is a challenging task for several reasons. Firstly, there is no optimal general solution to obtain DMAs that provide both low overhead for the management of the memory region and high speed in satisfying memory requests, as demonstrated in the survey [27]. Consequently, the design of a DMA shall take into account its specific use and adjusts the combination of techniques to obtain an optimal solution for this use. This leads to a wide variety of DMA implementations to be specified and proved correct. Secondly, the formal methods used to prove correctness shall deal with such optimized implementations which are usually combining low level code (e.g., pointer arithmetics, bit fields) with efficient high level data structures (e.g., hash tables with doubly linked lists). The difficulty to formally analyse particular DMA implementations has been demonstrated by several projects [5, 7, 14, 19, 26]. These projects make use of highly expressive logics to specify the memory organisation and content, e.g., second order logics or Separation Logic [21], which need sophisticated tools to be dealt with. Finally, there is no evidence that the techniques developed in these projects may be applied to verify the correctness of DMA implementations using different customizations.

This paper is a first step towards providing optimal and formally proved correct DMA implementations. We adopt a correct-by-construction approach, which is different from the most of research this area. In this approach, an abstract model is gradually refined to obtain a model that is detailed enough for code generation or code annotation. We apply this approach to the full class of list based DMA and therefore we obtain a set of formal models organised in a hierarchy ranked by refinement relations that establishes a *formally specified taxonomy of the techniques* employed by the implementations of list based DMA. This formally specified taxonomy forms an *algorithm theory* [24] for the free list DMA, i.e., a structure common to all implementations in this class, which abstracts away specific implementation concerns. To limit the complexity of this work, we consider DMA without support for concurrency, i.e., used in a sequential setting.

The most abstract specification of DMA is refined incrementally by introducing all the specific design tactics for heap and free list organisation reported in [27]. By studying the dependencies between the existing design tactics, we deduce an order for applying the refinement steps. We prove that this order increases the ability to conduct correctness and refinement proofs. For example, we first refine the most abstract specification of DMA into several models by modeling the techniques used to manage the heap list, in particular, the scheme for the management of successive free chunks, also called the *coalescing policy*. Indeed, we found out that the tactics for the free list are bound to specific coalescing tactics and therefore the refinement by free list tactics shall be done afterwards.

Our taxonomy includes more than one hundred models, some of them specifying interesting case studies, e.g., eight open source DMA implementations (e.g., [3, 4, 13, 16, 26]). Moreover, we obtain formal models for new combinations of techniques, not covered by the set of case studies we consider. Actually, our formal models are *implementation independent* due to the use of a formally defined signature $\mathbb{S}$ that abstracts the implementation details. For example, the taxonomy maps to the same model two implementations employing the same design tactics but keeping track of the size of chunks in different ways (e.g., as an integer or by the address of the next chunk). Therefore, the model-based analysis and code generation methods could use the specific implementation of the abstraction signature $\mathbb{S}$ to reach a specific DMA implementation.

Our work has a more theoretical consequence. It reveals the class of logics necessary to specify precisely each of the design tactics considered and thus it is a useful guide for the formal verification of DMA. For example, we identified the technique that requires second order logic to capture its precise state invariant: the use of a list of free chunks which is not sorted by the start address of chunks (see Section 4.1). Excepting this technique, the models proposed use only first order, universally quantified state invariants, which is a good class for automatic provers.

The formalisation work is done using Event-B [1] and the tool Rodin [2]. An Event-B model is a state machine with set-typed variables; it specifies the invariants to be satisfied in each state and the state changes, called events. The events are specified by defining their activation condition and the effect they produce on the state. A state machine M_1 may be refined to obtain a new machine M_2 by adding more state variables or events. Rodin generates automatically the proof obligations needed to prove the refinement relation (i.e., machine simulation) and it calls existing automatic solvers to discharge them or, if these solvers fail, the interactive proof system. Notice that the knowledge of these tools is not mandatory for the reader of this paper and we present the specifications such

that they may be exported to other systems providing powerful logic theories and means for refinement proofs, e.g., TVLA or Isabelle. The refined models may be used directly by developers to obtain *complete and sound* specifications (state invariants and pre/post conditions for methods) for DMA implementations in this class.

To sum up, this paper has the following contributions:

- We formalize a hierarchy of models for the class of list based DMA. The hierarchy is ranked by formally proved refinement relations and it includes complete and sound specifications of existing DMA implementations. Although extensible to other design tactics, our hierarchy covers actually all the techniques used for the management heap lists: we extend here the variable size heap list explored in [9] with array based and buddy DMA.
- We propose an algorithm theory for free list DMA. We identify a signature representing an abstraction from implementation details of DMA and links the formal models proposed with the concrete implementations.
- We illustrate the application of this work to model-based code generation, testing and verification techniques.

The paper is organised as follows. Section 2 introduces the DMA services and provides their first abstract specification. Then, it surveys the existing policies and techniques for free list DMA, presents our case studies, and concludes with our refinement principles. Section 3 defines a first refinement step and the models it produces for the low level management of the memory region, i.e., the heap list. Section 4 continues the refinement by introducing the policies for the free list. Section 5 illustrates the applications of this work to code generation, model-based testing and verification of DMA. We provide links with existing work and conclude in Section 6.

2 Dynamic Memory Allocators

In this section, we overview the interface provided by DMA as well as the techniques and policies used in existing implementations. Then, we present the first abstract specification of DMA and the refinement strategy.

2.1 DMA Interface

DMA manages a set of memory blocks and provides to its clients an interface allowing to allocate and deallocate memory. A memory block managed by DMA is called a *chunk*. Each chunk includes two parts, as illustrated in Figure 1: a *header* used to store information about the chunk (e.g., its status and its size) and a *body* to store clients' data. The body of a chunk shall be a contiguous memory region inside the chunk. The offset of the start address of the body with respect to the start address of the chunk is fixed in a given DMA implementation; therefore, the size of the body is easily obtained from the full size of the chunk.

The interface usually provided by a DMA is shown Table 1. The method `init` initialises the set of chunks managed and marks them all to be *free*. A call `alloc(n)` searches a free chunk whose body has size (in bytes) at least `n`. If such a free chunk is found, it is marked as *busy* and the call to `alloc` returns the start address of the chunk body; otherwise `alloc` returns an invalid address, which is denoted here by nil. A call `free(p)` succeeds if `p` is the start address of the body of a busy chunk; the chunk is marked as free and the call `free` returns true. Otherwise, the call does nothing and returns false. The size of a busy chunk may be changed to `n` using `realloc(p,n)`. If the size is decreased, a new free chunk is created at the end of the body pointed by `p` and the returned pointer is `p`. Otherwise, `realloc` either enlarges the chunk of `p` if it is enough free memory after the current chunk (and returns `p`) or it allocates a new chunk, frees the chunk of `p` after copying its body in the new chunk, and returns a pointer in the new chunk.

Table 1 Interface of DMA

```
1 void init();
2 void* alloc(size_t sz);
3 void* realloc(void* p, size_t sz);
4 bool free(void* p);
```

2.2 Abstract Specification of the Interface

Table 2 includes a very abstract formal specification of the above informal description. Because `realloc` is a composition of `alloc` and `free`, and for sake of space, we provide its formal specification in our technical report [10]. The memory region managed by the DMA is viewed as a sequence of bytes starting at the address hst and ending before the address hli. A state of the memory region is modelled by a tuple $\sigma \triangleq \langle H, F, csz, cst \rangle$ where:

- H models the set of start addresses of chunks managed.

• The size of the chunk, stored in its header, is modelled by the total mapping *csz*. The size of the header is modelled by the constant chd; it also gives the offset of the starting address of the chunk body.

• The status of the chunk (1 for free, 0 for busy), stored in its header, is modelled by the mapping *cst*. For readability of specifications, we denote by F the set $cst^{-1}(1)$, i.e., the *set of free chunks*.

We use the dotted notation, e.g., $\sigma.H$, for the above state components. Notice that the content of chunk body is abstracted out in this abstract specification.

Table 2 Most abstract specification A

	Constants		State σ
$\mathsf{hst}, \mathsf{hli}$	limits of the memory region	H, F	set of all chunks resp. free chunks
$\mathsf{chd}, \mathsf{cal} \in \mathbb{N}$	size of header resp. alignment	$csz : H \to \mathbb{N}$	size of a chunk
$\mathsf{fit} : H \times \mathbb{N} \to \mathbb{N}$	fitting a chunk with a request	$cst : H \to \{0,1\}$	status of a chunk (1–free, 0–busy)

Invariants for states $\sigma \triangleq \langle H, F, csz, cst \rangle$

$$I_1 : H \subseteq [\mathsf{hst}, \mathsf{hli}[\qquad I_2 : \forall c \in H \cdot c \ \mathtt{MOD}\ \mathsf{cal} = 0 \qquad I_3 : \mathsf{chd} > 0 \qquad I_4 : \forall c \in H \cdot csz(c) \geqslant \mathsf{chd}$$

$$I_5 : F \subseteq H \ \wedge \ \forall c \in F \cdot cst(c) = 1 \iff c \in F$$

$$I_6 : \forall b, c \in H \cdot c \neq b \Rightarrow [c, c + csz(c)[\cap [b, b + csz(b)[= \varnothing$$

Inference rules for methods

$$\text{Init}\ \frac{}{\sigma \xrightarrow{\mathtt{init}()} \sigma[F \leftarrow \sigma.H]}$$

$$\text{Free}^S\ \frac{\exists c \in \sigma.H \setminus \sigma.F \cdot p = c + \mathsf{chd}}{\sigma \xrightarrow{\mathtt{free}(p):\text{true}} \sigma[cst(c) \leftarrow 1]} \qquad \text{Free}^F\ \frac{\forall c \in \sigma.H \setminus \sigma.F \cdot p \neq c + \mathsf{chd}}{\sigma \xrightarrow{\mathtt{free}(p):\text{false}} \sigma}$$

$$\text{Alloc}^S\ \frac{c \in \sigma.F \qquad \mathsf{fit}(c, s) \leqslant \sigma.csz(c) \qquad p = c + \mathsf{chd}}{\sigma \xrightarrow{\mathtt{alloc}(s):p} \sigma[cst(c) \leftarrow 0]} \qquad \text{Alloc}^F\ \frac{\forall c \in \sigma.F \cdot \mathsf{fit}(c, s) > \sigma.csz(c)}{\sigma \xrightarrow{\mathtt{alloc}(s):\mathsf{nil}} \sigma}$$

The properties I_1–I_6 are correctness invariants for the abstract states. Property I_1 specifies that the elements of H shall be in the limits of the memory region managed. The alignment of the start addresses of chunks on multiples of the constant cal is specified by property I_2. Property I_6 requires that chunks in H occupy pairwise disjoint memory blocks. The relation between F and *cst* is specified by I_5.

The correct behaviours of the DMA methods are specified by inference rules in Table 2. We denote by $\sigma \xrightarrow{m(a):r} \sigma'$ the change of state from σ to σ' produced by the call of method m (from ones in Section 2.1) with parameter a and the returned value r (we omit r if void). We denote by $\sigma[f \leftarrow e]$ the state which is exactly σ except for the component f whose value is set to e. In the rules for `alloc`, we use the mapping fit to abstract the way of computing the number of bytes fitting a request. For example, $\mathsf{fit}(c, n) = (n + 3)/4$ aligns n to a multiple of four bytes; examples of fit mappings are given in Table 14 (page 14).

This specification reveals the main ingredients of the DMA without fixing a special policy or technique. Its high degree of abstraction and simplicity is obtained due to the following two hypotheses on abstract states:

• *Fixed set H of chunks managed:* This hypothesis is not satisfied in general, but it is present in implementations where the size of chunks is fixed to some constant, i.e., array implementation of the heap list.

• *External implementation of the set of chunks data structure:* By abstracting the implementation of data structure used for the set of chunks, we cannot specify the disposition of chunks inside the data segment, which controls important properties of DMA implementations, in particular, the absence of memory leaks.

2.3 Design Tactics for DMA

We faced a large variety of policies and techniques when removing the above simplifying hypotheses. Therefore, we focus on implementations using a *heap list* data structure for the set of chunks. In this section, we shortly

describe the design tactics relevant for this class. We extracted them from the case studies discussed in the next section and from comprehensive surveys of memory allocators like [27]. Table 3 summarises these design tactics employed by our case studies.

A heap list stores the chunks in sequence inside the memory region, like shown in Figure 1. Two techniques exist to encode the successor of a chunk in the heap list: (i) the header stores the start address of the next chunk or (ii) using the chunk size and address arithmetics, i.e., the expression $c + \sigma.csz(c)$. Moreover, some DMAs implement a doubly-linked heap list using the principle of "boundary tag" [15]. This feature is represented by the column "*linked*" in Table 3, where "→" (resp. "↔") means singly (resp. doubly) linked list, and "addr" (resp. "size") represents technique (i) (resp. (ii)) above.

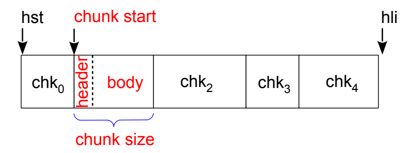

Figure 1 A heap list of five chunks

To simplify the management, some DMAs fix the size of all chunks to some constant. Thus, it only allocates memory blocks fitting inside the fixed size. Other DMAs require extra constraints on the size of the chunk. For example, the sizes of chunks in a buddy DMA shall be a power of two.

To fit a request, the DMA with variable size chunks may split a free chunk into two chunks: a chunk allocated for the request and a smaller free chunk. The order between these chunks is another parameter of our taxonomy, represented by the column "*split*" in Table 3, where the values indicate the position of the free chunk after splitting. Some DMA always splits a chunk into two same sized chunk, such as buddy system.

Splitting of chunks may lead to memory fragmentation. There exist two classes of policies for defragmentation (column "*defrag.*" in Table 3), both of them joining sequences of free chunks together in one free chunk:

• *Early coalescing* policy does defragmentation during `free`. The newly free chunk is joined with its neighbours if they are free. This process may be either total or partial. After a total coalescing, the heap list does not contain two adjacent free chunks. This is not the case for the *partial coalescing*, which joins the newly freed chunk with its free neighbours only if some additional constraints are satisfied. For example, the buddy system allocator only merges two adjacent free chunks if they are in the same logic block, called buddy. To simplify the vocabulary, we use early coalescing for the total coalescing.

• *Lazy coalescing* policy does defragmentation during the call of `alloc`, if none of the existing free chunks is large enough to satisfy the request. The coalescing may be total or partial.

A technique to accelerate the search of a fitting free chunk during `alloc` consists in indexing free chunks in an additional data structure. This paper focuses on so called *free list* DMA where the free chunks are kept in a list. The list is built using additional information in the chunk header and it may have several shapes: singly or doubly linked, acyclic or circular, etc. Moreover, some techniques keep the free list sorted by the start address of chunks in order to accelerate defragmentation. The last design tactic we consider for the free list is the policy used to select a fitting free chunk when several are available. Possible options are: (i) *first* fitting chunk in the heap list or the free list, (ii) the chunk which fits (with smallest difference) the request, i.e., *best* fit, (iii) *next* fitting chunk with respect to the last allocation or deallocation.

2.4 Case Studies

Table 3 summarises the design tactics employed by thirteen case studies implementing list based DMA we collected. These case studies appear to us as representative for the list based DMA because they illustrate all the design tactics we listed in the previous section.

The first part of Table 3 contains DMA that don't use a free list. The DMA TOPSY is the memory manager of the TOPSY operating system [11]. The IBM allocator is provided in [4]. The DL-small allocator is extracted from Doug Lea's allocator [16]; it represents the part which deals with requests for memory with size less than 256 bytes. The Buddy allocator is described in [15] to illustrate the buddy allocator systems.

The case studies in the second part of Table 3 use singly linked, address sorted, free lists. The memory allocator of the L4 microkernel [26] keeps the heap list as an array of fixed chunks and the free chunks in this tray are stored in an acyclic singly linked list. The DKFF and DKBF allocators are our implementations of algorithms A and B Section 2.5 of [15], for which we choose the first-fit and best-fit policies. The LA allocator [3] is the implementation of Knuth's algorithm A by Aldridge. The DKNF allocator is our implementation of the next-fit

Table 3 Design tactics employed in case studies

	heap list				*free list*		*fit*	*Model*
Case study	*linked*	*split*	*defrag.*	*array*	*shape*	*sorted*	*policy*	Figure 2
IBM [4]	addr, →	–	–	–	–	–	first	**MH**
DL-small [16]	size, →	–	–	yes	–	–	first	**MH**
TOPSY [11]	size, →	at end	lazy	–	–	–	first	**MHL**
Buddy [15]	size, ↔	at start	partial	–	–	–	first	**MHP**
L4 [26]	addr, →	–	lazy	yes	acyclic, →	yes	first	**MASAF**
DKFF [15]	size, →	at start	early	–	acyclic, →	yes	first	**MSAF**
DKBF [15]	size, →	at start	early	–	acyclic, →	yes	best	**MSAB**
LA [3]	size, →	at start	early	–	acyclic, →	yes	first	**MSAF**
DKNF [15]	size, →	at start	early	–	acyclic, →	yes	next	**MSAN**
KR [13]	size, →	at start	early	–	cyclic, →	yes	next	**MSC**
DKBT [15]	size, ↔	at start	early	–	acyclic, ↔	no	best	**MUAD**
DL-list [16]	size, ↔	at start	early	–	acyclic, ↔	no	best	**MUAD**
TLSF [20]	size, ↔	at start	early	–	acyclic, ↔	no	best	**MUAD**

policy using the "roving pointer" technique proposed in [15] (Exercice 6 in Section 2.5). The KR allocator is the code published in [13] for a next fit, circular free list DMA.

The third part of Table 3 contains case studies using doubly linked free lists. The DKBT allocator is our implementation of the "boundary tag" technique introduced in [15]. In C, this technique is implemented by setting as first field of the chunk header the information (status and start address) of the previous chunk in the heap list; this information is generally used only if the previous chunk is free, as part of the coalescing of adjacent free chunks. The DL allocator is a simplification[1)] of the part of Doug Lea's allocator [16] for medium size requests. The TLSF allocator described in [20] distributes the free chunks in several doubly linked lists, depending on their size; we consider here a simplified version with only one free list.

2.5 Hierarchy of Models

An important observation on the data collected in Table 3 is the relation between the design tactics employed and the basic operations on list data structure. Indeed, chunk splitting and defragmentation call two elementary operations on heap lists: inserting a new chunk and removing a chunk by merging it with some neighbour. For the free list, the operations required by different policies are exactly the same: insertion and removing of a free chunk, searching for a fitting chunk. Moreover, these basic operations may be implemented in different ways and their composition produces the variety of policies and techniques discussed. Thus, by identifying how the methods of the DMA are obtained from these basic operations, we could obtain different models of the DMA only by refining the elementary operations.

From the above observation and our experience with refinement proofs, we extract the following principles of refinement:

R_1 : *Refinements of the heap list precede the ones of the free list.*

R_2 : *Refinements of basic operations on heap (resp. free) list shall compose for the same state modelling.*

R_3 : *Refinements of the fit policy shall be done in the end.*

R_4 : *Refinements concern basic operations on heap (resp. free) list.*

We applied the above refinement principles to obtain a hierarchy of models, part of it presented in Figure 2. This hierarchy mainly includes two layers, called *heap list* and *free list*.

The first layer contains five models, each of them modelling a particular organisation of the the heap list. For example, the models **MH** and **MA** don't coalesce free chunks but they use variable size and fixed size chunks, respectively. The model **MHL** uses a heap list with lazy coalescing policy, while **MHP** and **MHE** use partial coalescing and eager coalescing, respectively. The models in the second layer refine the models in the first layer by introducing the design tactics for the free list. These design tactics are explicit in the box of each model. The black arrows between boxes are the refinement relations, e.g., the model **MUA** is a refinement of the model **MHE**.

1) We removed the code concerning concurrency, portability, and some optimisations.

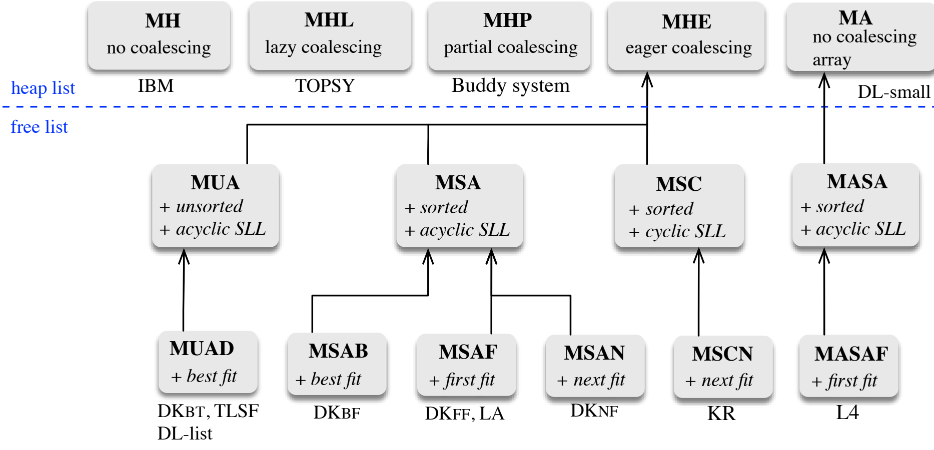

Figure 2 A partial view of the hierarchy of models and the case studies it covers

There can be several refining directions starting from a model, e.g., the model **MSA** has three different refinements. Therefore, the model **MSAF** specifies a DMA with an early coalescing heap list (it transitively refines **MHE**), a free list sorted by address and acyclic, and a first fit policy.

Actually, Figure 2 includes only the part of the hierarchy that covers our case studies, listed as labels of models. However, the refinement relations we define in the next sections allow to obtain more models. Indeed, the hierarchy in Figure 2 can be extended to specify more cases. The refinements for free list can start from any model in the heap list layer. In this paper, we mainly describe the branch refining **MHE** because it is the most complex model.

Some readers concerned by implementation details may get worried about some design choices that are not covered by the above presentation, e.g., alignment of start addresses for chunks, encoding of the free status of chunks in the header, the unit on which the size of the chunk is measured, the fitting size. For them, we apply the abstraction principle used in the model presented in Section 2.2, more precisely, we define in Section 5 a signature $\mathbb{S}$ that abstracts these low level design choices.

3 Heap List Modelling

This section presents the models in the heap list level of the hierarchy in Figure 2. We mainly show that these models may be obtained by composing refinements of basic operations on heap lists and that these refinements are discriminated by their handling of the chunk coalescing.

Table 4 Refinement of A for heap list models

State refinement			
$cnx : H \rightarrow (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\}$	next chunk	$cpr : (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\} \rightarrow H$	previous chunk
$\sigma \triangleq \langle H, F, csz, cst, cnx, cpr \rangle$	state of DMA with heap list		
Additional invariants			
$I_7 : cnx$ is a bijection	linked heap list	$I'_7 : cpr = cnx^{-1}$	doubly linked list
$I_8 : \mathsf{hst} \in H$	start in hst	$I_9 : \forall c \in H \cdot cnx(c) = c + csz(c)$	no leaks
$I_{ar} : \forall c \in H \cdot csz(c) = kb$			array heap list
$I_{ec} : \forall c_1, c_2 \in H \cdot \big(cnx(c_1) = c_2\big) \Rightarrow \mathrm{NAND}(cst(c_1), cst(c_2))$			early coalescing
$I_{pc} : \forall c_1, c_2 \in H \cdot buddy(c_1, c_2) \Rightarrow \mathrm{NAND}(cst(c_1), cst(c_2))$			partial coalescing
$I_{by} : \forall c \in H \cdot \exists k \in \mathbb{N} \cdot csz(c) \ = 2^k \wedge (c \bmod 2^k = 0)$			buddy size constraint

3.1 State and Invariants

The abstract state of a heap list DMA is defined in Table 4. It includes, in addition to elements of the abstract state of the model A defined in Table 2, the successor and predecessor relations between chunks, *cnx* resp. cpr. The mapping cpr is specified only for doubly linked heap lists. In addition to invariants in Table 2, we introduce the invariants I_7 and I_7' in Table 4 to characterise the two new relations. They assert that *cnx* is a bijection, and if cpr is defined, it is the inverse of *cnx*. A consequence of invariants I_7–I_8 is the following expected property:

Property 3.1. *The heap list is acyclic, starts in* hst*, and ends in* hli*.*

The invariant I_9 asserts that a chunk occupies exactly the space between its start and the next chunk, which leads to the following property:

Property 3.2. *A heap list satisfying* I_1–I_9 *has no memory leaks.*

Each model in the heap list layer satisfies the invariants I_1–I_9 and a subset of the last four invariants in Table 4. For example, the model **MA** includes only the invariant I_{ar} that fixes the size of chunks to some constant kb to specifies the class of DMA that manages an array of chunks. The invariant I_{ec} is added only for the model **MHE** to characterise the state of DMA with early coalescing policy. It asserts that any two chunks, successive in the heap list, cannot be both free. The invariant I_{pc} is satisfied by the model **MHP** for heap list with partial coalescing. It specifies that two adjacent free chunks can not be both free if they belong to the same buddy, which is expressed the predicate $buddy(c_1, c_2)$, defined by $cnx(c_1) = c_2 \wedge csz(c_1) = csz(c_2) \wedge c_1 \texttt{ MOD } (2 \times csz(c_1)) = 0$. Notice that both I_{ec} and I_{pc} may be temporary broken during the execution of methods `free` and `realloc`. In addition to I_{pc}, the model **MHP** includes the invariant I_{by} that constrains the size of each chunk to be a power of two and its address to be aligned to this size.

3.2 Basic Heap List Operations

The inference rules in Table 2 abstract away the implementation details of DMA methods and don't capture the splitting of a fitting chunk or the merging of adjacent free chunks during allocation and deallocation. To refine these rules into ones that specify precisely the behaviour of methods of heap list DMA, we use the following basic operations on the heap lists:

hremove	removes a free chunk (from F)	hinsert	inserts a free chunk (into F)
hsplit	splits a free chunk	hmerge_L	merges a free chunk with its free left neighbour
$\mathsf{hmerge}_\forall$	merges all sequences of free chunks	hmerge_R	merges a free chunk with its free right neighbour
hsearch	searches a fitting free chunk	hmerge_P	merges free chunks in same buddy

We explain the specifications of these basic operations for singly linked heap lists in this section. Notice that only remove, insert, and search operations are relevant for array based DMA. ***To simplify the presentation of rules, we adopt the convention that the elements of state*** σ ***(the source state of the defined rule) appear without the dotted notation in the rule.*** For some operations, e.g., hsplit or hsearch, several refinements are provided. The main methods of the DMA are specified in Table 8 using these basic operations.

Table 5 specifies the operations hremove, hinsert, and hsearch. The status (free or busy) of the chunk c given as parameter is updated accordingly for removing and insertion in the free set. Operation $\mathsf{hsearch}(n)$ has two refinements given by the rules $\mathsf{hsearch}_{FF}$ and $\mathsf{hsearch}_{BF}$ that specify the first fit resp. best fit policies for the search of the chunk fitting the requested size n. The rule $\mathsf{hsearch}^F_{*F}$ specifies the failure behaviour for both policies. Table 6 specifies the refinements for the operation hsplit. This operations has as parameters a free chunk c and a natural n representing the size of the new chunk to be created inside c; this new chunk is set as busy and returned as result of hsplit. The three refinements of hsplit, represented by behaviours hsplit_M, hsplit_B and hsplit_E, choose different ways to split the chunk: in two equal size parts, with n bytes at the beginning, or at the end respectively. The behaviour hsplit_P refines hsplit for buddy DMA: it applies repeatedly hsplit_M (rule hsplit^S_P) until the requested size n fits in the chunk and it is bigger than the half of candidate chunk (rule hsplit^F_P).

Table 7 provides two refinements of the operation hmerge that is called in `free` or `realloc` to join neighbouring free chunks in one. The invariants for early or partial coalescing (I_{ec} resp. I_{pc}) are broken temporarily in the state before calling hmerge. The behaviour hmerge_R joins the chunk parameter b with its right neighbour

Table 5 Refinements of heap list operations for remove, insert, and search

Operation	Rule
$\mathsf{hremove}(c)$	$\mathsf{hremove}\ \dfrac{c \in F}{\sigma \xrightarrow{\mathsf{hremove}(c)} \sigma[F \leftarrow F \setminus \{c\}, \mathit{cst}(c) \leftarrow 0]}$
$\mathsf{hinsert}(c)$	$\mathsf{hinsert}\ \dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hinsert}(c)} \sigma[F \leftarrow F \cup \{c\}, \mathit{cst}(c) \leftarrow 1]}$
$\mathsf{hsearch}(n) : c$	$\mathsf{hsearch}^S_{FF}\ \dfrac{c \in F \quad \mathsf{fit}(c,n) \leqslant \mathit{csz}(c) \quad \forall b \in F \cdot b < c \Rightarrow \mathit{csz}(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$
	$\mathsf{hsearch}^F_{*F}\ \dfrac{\forall b \in F \cdot \mathit{csz}(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} \sigma}$
	$\mathsf{hsearch}^S_{BF}\ \dfrac{c \in F \quad \mathsf{fit}(c,n) \leqslant \mathit{csz}(c) \quad \forall b \in F \cdot (c \neq b \wedge \mathsf{fit}(b,n) \leqslant \mathit{csz}(b)) \Rightarrow (\mathit{csz}(b) - \mathsf{fit}(b,n) \geqslant \mathit{csz}(c) - \mathsf{fit}(c,n))}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$

Table 6 Refinements of heap list operation for chunk splitting

Operation	Rule
$\mathsf{hsplit}(c,n) : b$	$\mathsf{hsplit}_M\ \dfrac{c \in F \quad 0 < n < \mathit{csz}(c)/2 \quad c' = c + \mathit{csz}(c)/2 \quad \left\langle H \cup \{c'\}, F, \mathit{csz}[c, c' \leftarrow \mathit{csz}(c)/2], \mathit{cnx}[c \leftarrow c', c' \leftarrow \mathit{cnx}(c)] \right\rangle = \sigma_1 \xrightarrow{\mathsf{hremove}(c)} \sigma_2 \xrightarrow{\mathsf{hinsert}(c')} \sigma_3}{\sigma \xrightarrow{\mathsf{hsplit}_M(c,n):c} \sigma_3}$
	$\mathsf{hsplit}_B\ \dfrac{c \in F \quad 0 < n < \mathit{csz}(c) \quad c' = c + n \quad \left\langle H \cup \{c'\}, F, \mathit{csz}[c \leftarrow n, c' \leftarrow \mathit{csz}(c) - n], \mathit{cnx}[c \leftarrow c', c' \leftarrow \mathit{cnx}(c)] \right\rangle = \sigma_1 \xrightarrow{\mathsf{hremove}(c)} \sigma_2 \xrightarrow{\mathsf{hinsert}(c')} \sigma_3}{\sigma \xrightarrow{\mathsf{hsplit}(c,n):c} \sigma_3}$
	$\mathsf{hsplit}_E\ \dfrac{c \in F \quad 0 < n < \mathit{csz}(c) \quad c' = c + \mathit{csz}(c) - n \quad \left\langle H \cup \{c'\}, F, \mathit{cst}(c') \leftarrow 0, \mathit{csz}[c \leftarrow \mathit{csz}(c) - n, c' \leftarrow n], \mathit{cnx}[c \leftarrow c', c' \leftarrow \mathit{cnx}(c)] \right\rangle = \sigma_1}{\sigma \xrightarrow{\mathsf{hsplit}(c,s):c'} \sigma_1}$
$\mathsf{hsplit}_P(c,n) : b$	$\mathsf{hsplit}^S_P\ \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{hsplit}_M(c,n):b} \sigma_1 \xrightarrow{\mathsf{hsplit}_P(b,n):b'} \sigma_2}{\sigma \xrightarrow{\mathsf{hsplit}_P(c,n):b'} \sigma_2}$
	$\mathsf{hsplit}^F_P\ \dfrac{c \in F \quad \mathit{csz}(c)/2 < n < \mathit{csz}(c)}{\sigma \xrightarrow{\mathsf{hsplit}_P(c,n):c} \sigma}$

c if c is free; otherwise, the operation does nothing. For sake of symmetry with the second behaviour, hmerge_R returns its parameter. Similarly, the refinement hmerge_L merges a free chunk with its left free neighbour. The behaviour $\mathsf{hmerge}_\forall$ merges any two successive free chunks in the entire memory region. The rule $\mathsf{hmerge}^F_\forall$ states that I_{ec} is satisfied and therefore the merging operation terminates. The refinement of hmerge for DMA with partial coalescing is specified by the rules of hmerge_{PN} that joins only chunks in the same buddy. These operations are combined in operation hmerge_{PN} which is called repeatedly by hmerge_P until the invariant I_{pc} is satisfied.

3.3 Models for Heap List DMA

The specifications of DMA methods make use of basic operations presented above as shown in Table 8.

We provide two refinements for the method `init`: the rule $\mathsf{hinit}_{\mathrm{hl}}$ initialises abstract state for DMA with variable size chunks, while the rule $\mathsf{hinit}_{\mathrm{ar}}$ does initialisation for fixed size chunks, i.e., array based DMA.

The `alloc` method is refined to obtain three distinct behaviours: for allocation in fixed chunk sized DMA (rule $\mathsf{halloc}_{\mathrm{ar}}$), without coalescing for variable chunk sizes (rule $\mathsf{halloc}_{\mathrm{eager}}$ because used in eager coalescing DMA), or for allocation with (lazy) coalescing (rule $\mathsf{halloc}_{\mathrm{lazy}}$). The last two behaviours call the internal operation halloc_i, which does the main part of the work: it searches the free chunk fitting the request using $\mathsf{hsearch}$ and returns this chunk after changing its status. The rule $\mathsf{halloc}^S_{\mathrm{fit}}$ specifies the case where the fitting chunk does not need splitting; the rule $\mathsf{halloc}^S_{\mathrm{split}}$ specifies the splitting operations. Notice that $\mathsf{halloc}^S_{\mathrm{fit}}$ allows to define behaviours for allocation without splitting: if $\mathsf{fit}(c,s)$ returns $\mathit{csz}(c)$ for $\mathit{csz}(c) \geqslant s$. Table 8 includes only some rules for `alloc`, the full specification is given in [10].

Table 7 Refinements of heap list operation for chunk merging

$\mathsf{hmerge}_R(b):x$	$\mathsf{hmerge}^S_R\ \dfrac{b \in F \quad c \in F \quad c = cnx(b) \quad \sigma \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \left[H \leftarrow H \setminus \{c\}, csz[b \leftarrow csz(b) + csz(c)], cnx[b \leftarrow cnx(c)] \right]}$	$\mathsf{hmerge}^F_R\ \dfrac{b \in F \quad cnx(b) \notin F}{\sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma}$
$\mathsf{hmerge}_L(b):x$	$\mathsf{hmerge}^S_L\ \dfrac{b \in F \quad c \in F \quad cnx(c) = b \quad \sigma \xrightarrow{\mathsf{hremove}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}_L(b):b} \sigma_1 \left[H \leftarrow H \setminus \{b\}, csz[b \leftarrow csz(b) + csz(c)], cnx[c \leftarrow cnx(b)] \right]}$	$\mathsf{hmerge}^F_L\ \dfrac{b \in F \quad cpr(b) \notin F}{\sigma \xrightarrow{\mathsf{hmerge}_L(b):b} \sigma}$
$\mathsf{hmerge}_\forall$	$\mathsf{hmerge}^S_\forall\ \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \xrightarrow{\mathsf{hmerge}_L(b):c} \sigma_2 \xrightarrow{\mathsf{hmerge}_\forall} \sigma_3}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma_3}$	$\mathsf{hmerge}^F_\forall\ \dfrac{I_{ec}}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma}$
$\mathsf{hmerge}_{PN}(b):x$	$\mathsf{hmerge}^S_{PN}\ \dfrac{b \in F \quad c \in F \quad buddy(b,c) \quad \sigma \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}_{PN}(b):b} \sigma_1 \begin{bmatrix} H \leftarrow H \setminus \{c\}, \\ csz[b \leftarrow csz(b) + csz(c)], \\ cnx[b \leftarrow cnx(c)] \end{bmatrix}}$	$\mathsf{hmerge}^{S'}_{PN}\ \dfrac{b \in F \quad c \in F \quad buddy(c,b) \quad \sigma \xrightarrow{\mathsf{hremove}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}_{PN}(b):c} \sigma_1 \begin{bmatrix} H \leftarrow H \setminus \{b\}, \\ csz[c \leftarrow csz(b) + csz(c)], \\ cnx[c \leftarrow cnx(b)] \end{bmatrix}}$
$\mathsf{hmerge}_P(b)$	$\mathsf{hmerge}^S_P\ \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_{PN}(b):c} \sigma_1 \xrightarrow{\mathsf{hmerge}_P(c)} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_P(b)} \sigma_2}$	$\mathsf{hmerge}^F_P\ \dfrac{I_{pc}}{\sigma \xrightarrow{\mathsf{hmerge}_P(b)} \sigma}$

The specification of the method `free` is refined similarly to obtain its behaviours for eager and lazy coalescing. We only show below the rule used for early coalescing; the other rules may be found in [10]. After freeing the chunk, the invariant I_{ec} is established by calling the merging with the free neighbours (if any):

$$\mathsf{hfree}^S_{\text{eager}}\ \frac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{hinsert}(b)} \sigma_1 \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_2 \xrightarrow{\mathsf{hmerge}_L(b):c} \sigma_3}{\sigma \xrightarrow{\mathsf{hfree}(p):\mathtt{true}} \sigma_3}$$

Table 9 sums up the main characteristics of each model resulting from the refining of the heap list: the specific invariants, the heap list operations used, and the size of the model. We coded these specifications in Event-B [1] machines. The correctness of the refinement, stated by the following theorem, is translated into a set of proof obligations which are proved with the Rodin tool [2] and the connected solvers. Table 9 provides statistics about the proofs conducted to obtain this theorem.

Theorem 1. *For any model of the heap list DMA (i.e.,* ***MH****,* ***MHA****,* ***MHL****,* ***MHE****), the operations specifying a DMA method preserve the invariants of the model.*

4 Free List Modelling

This section defines the refinements applied to capture the different design choices related with the use of a list for the set of free chunks. Following the principle R_1, these refinements are applied to models obtained by refinements of the heap list. Because they are the most interesting and for sake of space, we comment only the refinements of the **MHE** model, i.e., models dealing for DMA with early coalescing.

To conform to principle R_4, we define a set of basic operations on free list. These operations are the counterpart of the ones defined for the heap list in Section 3.2: fremove, finsert, fsplit, fmerge, and fsearch. We define four directions of refinement, each dealing with a specific feature of the free list: (1) shape of the free list, with values acyclic (A) and cyclic (C), (2) ordering of chunks by addresses in the free list, with values unordered (U) and sorted (S), (3) cells linking, with values singly (default) and doubly (D), and (4) searching for fit policy, with values first (F), best (B) and next (N) fit. Each direction corresponds to specific state elements, state invariants, or

Table 8 Refinements of methods for heap list

Method	Rule	Refinement
$\mathsf{hinit}()$	$\mathsf{hinit}_{\mathrm{hl}}$	$\dfrac{}{\sigma \xrightarrow{\mathsf{hinit}()} \left\langle H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, \mathit{cnx}(\mathsf{hst}) \leftarrow \mathsf{hli}, \mathit{cst}(\mathsf{hst}) \leftarrow 1, \mathit{csz}(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst} \right\rangle}$
	$\mathsf{hinit}_{\mathrm{ar}}$	$\dfrac{}{\sigma \xrightarrow{\mathsf{hinit}()} \left\langle H \leftarrow \{\mathsf{hst} + i \times \mathit{kb} \mid 0 \leqslant i\}, F \leftarrow \{\mathsf{hst} + i \times \mathit{kb} \mid 0 \leqslant i\}, \mathit{cnx}(c) \leftarrow c + \mathit{kb}, \mathit{csz}(c) \leftarrow \mathit{kb} \right\rangle}$
$\mathsf{halloc}_i(s) : p$	$\mathsf{halloc}^S_{\mathrm{fit}}$	$\dfrac{c \neq \mathsf{nil} \quad p = c + \mathsf{chd} \quad \mathsf{fit}(c, s) = \mathit{csz}(c) \quad \sigma \xrightarrow{\mathsf{hsearch}(s):c} \sigma \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_1}$
	halloc^F_i	$\dfrac{\sigma \xrightarrow{\mathsf{hsearch}(s):\mathsf{nil}} \sigma}{\sigma \xrightarrow{\mathsf{halloc}_i(s):\mathsf{nil}} \sigma}$
	$\mathsf{halloc}^S_{\mathrm{split}}$	$\dfrac{c \neq \mathsf{nil} \quad p = b + \mathsf{chd} \quad \mathsf{fit}(c, s) < \mathit{csz}(c) \quad \sigma \xrightarrow{\mathsf{hsearch}(s):c} \sigma \xrightarrow{\mathsf{hsplit}(c, \mathsf{fit}(c,s)):b} \sigma_1}{\sigma \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_1}$
$\mathsf{halloc}(s) : p$	$\mathsf{halloc}^S_{\mathrm{eager}}$	$\dfrac{\sigma \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_1}{\sigma \xrightarrow{\mathsf{halloc}(s):p} \sigma_1}$
	$\mathsf{halloc}^S_{\mathrm{ar}}$	$\dfrac{\sigma \xrightarrow{\mathsf{hsearch}(s):p} \sigma \xrightarrow{\mathsf{hremove}(p)} \sigma_1 \quad p \neq \mathsf{nil}}{\sigma \xrightarrow{\mathsf{halloc}(s):p} \sigma_1}$
	$\mathsf{halloc}^F_{\mathrm{ar}}$	$\dfrac{\sigma \xrightarrow{\mathsf{hsearch}(s):\mathsf{nil}} \sigma}{\sigma \xrightarrow{\mathsf{halloc}(s):\mathsf{nil}} \sigma}$
	$\mathsf{halloc}_{\mathrm{lazy}}$	$\dfrac{\sigma \xrightarrow{\mathsf{halloc}_i(s):\mathsf{nil}} \sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma_1 \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_2}{\sigma \xrightarrow{\mathsf{halloc}(s):p} \sigma_2}$

Table 9 Overview of heap list models and statistics on proofs

Models	Specific invariants	Rules: `init`, `alloc`, `free`	Rules: split	Rules: search	LOC	Proof obligations	Automatically discharged	Interactive proofs
MH	none	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{eager}}, \mathsf{hfree}_{\mathrm{lazy}}$	hsplit_B	$\mathsf{hsearch}_{FF}$	114	39	27(69%)	12(31%)
MHL	none	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{lazy}}, \mathsf{hfree}_{\mathrm{lazy}}$	hsplit_B	$\mathsf{hsearch}_{FF}$	176	8	8(100%)	0(0%)
MHE	I_{ec}	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{eager}}, \mathsf{hfree}_{\mathrm{eager}}$	hsplit_E	$\mathsf{hsearch}_{FF}$	183	82	58(70%)	24(30%)
MHP	I_{pc}, I_{by}	$\mathsf{hinit}_{\mathrm{hl}}, \mathsf{halloc}_{\mathrm{eager}}, \mathsf{hfree}_{\mathrm{partial}}$	hsplit_P	$\mathsf{hsearch}_{FF}$	383	143	140(98%)	3(2%)
MA	I_{ar}	$\mathsf{hinit}_{\mathrm{ar}}, \mathsf{halloc}_{\mathrm{ar}}, \mathsf{hfree}_{\mathrm{lazy}}$	–	$\mathsf{hsearch}_{FF}$	168	20	20(100 %)	0 (0%)

refinements of basic operations and DMA methods as summarised up in Table 13; the notations used in this table are introduced in the following section.

4.1 States and Invariants

Table 10 defines the states and the invariants used by the refinement directions. Notice that a free list state extends a state of the heap list model with at least one mapping, the bijection *fnx*, that models the linking in the free list. For doubly linked lists, the linking backward is modelled by the mapping *fpr*. The invariants satisfied by the linking mappings are I_{fnx} and I_{fpr}. To capture easily all the shapes of the free lists in our modelling framework, we use two constants, *fbe* and *fen* which delimit the start resp. target (end) of the free list. The variable *rp* is used by the state σ_N modelling the next fit policy to mark the last used free chunk. Thus, we could employ the invariant $I_{\ell s}$ for both cyclic and acyclic lists to ensure the following property:

Property 4.1. *If a state satisfies* I_{fnx}, $I_\varnothing$ *(and* I_C*), and* $I_{\ell s}$ *then the mapping fnx defines an acyclic (resp. cyclic) list starting in* $\mathit{fnx}(\mathit{fbe})$ *and including all free chunks.*

Notice that reachability is a second order property. $I_{\ell s}$ is a manner to express this property, inspired by [1]; it states that *fnx* does not define a clique inside F. This is the only place where we need a second order property. For tools with support limited to first-order logic, $I_{\ell s}$ may be replaced by a first order invariant if the free list is address sorted, property specified by the invariant I_S. Indeed, the following property is a corollary of *fnx* being bijective and strictly increasing:

Property 4.2. *If a state satisfies* I_{fnx}, $I_\varnothing$ *(and* I_C*), and* I_S *then the mapping fnx defines an acyclic (resp. cyclic) list starting in* $\mathit{fnx}(\mathit{fbe})$ *and including all free chunks.*

Table 10 States and invariants used by free list refinements; $x \in \{A, C\}$ denotes refinements for the shape of the free list

Refined states		Additional invariants	
$fbe, fen \notin H$	constants	I_{fnx} : fnx_x total bijection	
$F^+ = F \cup \{fbe, fen\}$	extended free set	$I_{fpr} : fpr_x = fnx_x^{-1}$	
$fnx_A : (F \cup \{fbe\}) \to (F \cup \{fen\})$	next free chunk	$I_\varnothing : fnx_x(fbe) = fen \iff F = \varnothing$	empty list
$fpr_A : (F \cup \{fen\}) \to (F \cup \{fbe\})$	previous free chunk	$I_{\ell s} : \forall F' \subseteq F \cdot F' \subseteq fnx_x^{-1}(F') \Rightarrow F' = \varnothing$	no clique
$fnx_C, fpr_C : F^+ \to F^+$	cyclic next resp. previous	$I_C : fnx_C(fen) = fbe$	*fen* ends the cycle
$\sigma \triangleq \langle H, F, csz, cst, cnx, fnx_x \rangle$	state for SLL	$I_S : \forall c \in F \cdot fnx_x(fbe) \leqslant c \leqslant fnx_x^{-1}(fen)$	
$\sigma_D \triangleq \langle H, F, csz, cst, cnx, fnx_x, fpr_x \rangle$	state for DLL	$\wedge (fnx_x(c) = fen \vee c < fnx_x(c))$	sorted list
$\sigma_N \triangleq \langle H, F, csz, cst, cnx, fnx_x, rp \rangle$	state for SLL, next fit policy	$I_{rp} : F \neq \varnothing \Rightarrow rp \in F$	*rp* is free

For models using unsorted free lists, we use the invariant $I_{\ell s}$ due to the fast that Rodin provides means for dealing with second order logic properties on sets.

4.2 Basic Operations

For sake of space, we give in Table 11 a sample of rules defining the refinements of basic operations on the free list for a free chunk removing, insertion, and searching. The rule $\mathsf{fremove}_A$ specifies the $\mathsf{hremove}$ basic operation for acyclic singly linked free list; it simply updates the relation *fnx*. The rule $\mathsf{finsert}_B$ specifies a refinement of an insertion operation for singly linked free lists which are unsorted. The new chunk is inserted at the end of the list in this rule, but a similar rule specifies the insertion at the beginning of the list in [10]. This is the case for the rule $\mathsf{finsert}_S^B$ which specifies the case of the insertion of a chunk in a free list sorted by the start addresses of chunks. Because the inserted chunk has an address smaller than all the free chunks in the list, it is inserted at the beginning of the list. The rule $\mathsf{fsearch}_{FF}^S$ refines the rule for $\mathsf{hsearch}$ with the first fit policy.

Table 11 Some refinements of basic operations on free list

Operation	Rule
$\mathsf{fremove}(c)$	$\mathsf{fremove}_A \dfrac{c \in F}{\sigma \xrightarrow{\mathsf{fremove}(c)} \sigma\left[F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0, fnx(fnx^{-1}(c)) \leftarrow fnx(c)\right]}$
$\mathsf{finsert}(c)$	$\mathsf{finsert}_U^E \dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{finsert}(c)} \sigma\left[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, fnx^{-1}(fen) \leftarrow c, fnx(c) \leftarrow fen\right]}$
	$\mathsf{finsert}_S^B \dfrac{c \in H \setminus F \qquad \forall b \in F \cdot c < b}{\sigma \xrightarrow{\mathsf{finsert}(c)} \sigma\left[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, fnx(fbe) \leftarrow c, fnx(c) \leftarrow fnx(fbe)\right]}$
$\mathsf{fsearch}(n) : c$	$\mathsf{fsearch}_{FF}^S \dfrac{c \in F \qquad \mathsf{fit}(c, n) \leqslant csz(c) \qquad \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b, n)}{\sigma \xrightarrow{\mathsf{fsearch}(n):c} \sigma}$

4.3 Models for Free List DMA

We developed several models by refining the free list in the heap list models, including the models in Figure 2. The DMA methods are specified in a way very similar to the one used for heap lists models, as could be seen on Table 12 for the methods `init` and `free` of the model **MSA**. The rule $\mathsf{ffree}_{\text{eager}}^S$ uses the operation $\mathsf{finsert}$ (instead of $\mathsf{hinsert}$) to update the links used by the free list and then tries to merge the inserted chunk with its neighbours using the operation fmerge_N, which refines hmerge for left and right neighbours (see the details in [10]).

Table 13 sums up the main ingredients used by the refinement directions to obtain the models presented in Figure 2. Like for the heap list models, we translate these models into Event-B machines and we prove with the Rodin

Table 12 Refinement of methods for free list

finit()	$\mathsf{finit}_A\ \dfrac{c \in F}{\sigma \xrightarrow{\mathsf{finit}_A()} \left\langle H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, cst(\mathsf{hst}) \leftarrow 1, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, fnx(\mathsf{hst}) \leftarrow \mathsf{nil} \right\rangle}$
ffree$(p) : b$	$\mathsf{ffree}^S_{\text{eager}}\ \dfrac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1 \xrightarrow{\mathsf{fmerge}_N(b)} \sigma_2}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathtt{true}} \sigma_2}$
	$\mathsf{ffree}^S_{\text{partial}}\ \dfrac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1 \xrightarrow{\mathsf{fmerge}_N(b)} \sigma_2}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathtt{true}} \sigma_2}$
	$\mathsf{ffree}^S_{\text{lazy/no}}\ \dfrac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathtt{true}} \sigma_1}$ $\qquad \mathsf{ffree}^F_*\ \dfrac{\forall b \in H \setminus F \cdot p \neq b + \mathsf{chd}}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathtt{false}} \sigma}$

Table 13 Overview of free list models and statistics on proofs

Models	State&New invariants	Rules: `init`	Rules: remove	Rules: search	LOC	Proof obligations	Automatically discharged	Interactive proofs
MUA	σ, fnx_A	finit_A	$\mathsf{fremove}_A$	$\mathsf{fsearch}_{BF}$	219	36	30(83%)	6(17%)
MSA	σ, fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{BF}$	197	41	27(66%)	14(34%)
MSC	σ, fnx_C, I_C, I_S	finit_C	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{BF}$	205	37	30(82%)	7(18%)
MSCN	$\sigma_N, fnx_C, I_C, I_S, I_{rp}$	finit_C	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{NF}$	194	40	36(88%)	4(12%)
MUAD	$\sigma_D, fnx_A, fpr_A, I_{fpr}$	finit_D	$\mathsf{fremove}_D$	$\mathsf{fsearch}_{BF}$	241	9	9(100%)	0(0%)
MSAB	σ, fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{BF}$	202	2	2(100%)	0(0%)
MSAF	σ, fnx_A, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{FF}$	202	2	2(100%)	0(0%)
MSAN	$\sigma_N, fnx_A, I_S, I_{rp}$	finit_N	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{NF}$	200	2	2(100%)	0(0%)

tool the following correctness and refinement theorem. Table 13 provides statistics about the proofs conducted to obtain the below theorem.

Theorem 2. *Every operation of a model for DMA with free list preserves the invariants of the model. Moreover, the refinement relations in Figure 2 are valid.*

5 Applications of the Formal Hierarchy of Models

Refinement towards DMA implementations In our models, the constants and state elements abstract the following implementation details: the boundaries of the memory region used by the DMA (variables hst and hli), the type of the header (constants chd, cal, mappings *csz*, *cst*, *cnx*, *cpr*, *fnx*, *fpr*), the algorithm deciding which is the number of bytes needed to satisfy a client request (mapping fit), and the boundaries of the free list (variables *fbe* and *fen*). Let $\mathbb{S}$ denote the above set of symbols.

The refinement to code is defined by associating to each element in $\mathbb{S}$ an expression using the types and variables of the DMA implementation such that the semantics of the element is fulfilled. We provide three examples of such refinements in Table 14. Notice that some elements of $\mathbb{S}$ may be left unspecified (entries with '$-$') if they are not used in the model (we omit the elements of $\mathbb{S}$ which are not specified in all examples in the table). The refinement relations for our benchmark are provided in [10]. We obtain these relations by inspecting the code of each allocator. However, we believe that some automatic analysis may be designed to extract automatically such information.

Code generation The elements of $\mathbb{S}$ and the rules presented in the previous sections may be exploited to generate code for DMA modelled by our specifications. In particular, the rules provide an operational semantics of DMA methods and a decomposition of these methods into calls to list (heap or free) operations. The invariants specified for each state may be translated into code and therefore provide means for run time verification of the correctness of a particular state of the DMA.

Table 14 Examples of refinement to code

$\mathbb{S}$	TOPSY [11] $\models$ **MHL**	LA [3] $\models$ $\mathbf{MSA}_{FF}$	KR [13] $\models$ $\mathbf{MSC}_{FF}$
hst	`start`	`_hsta`	
hli	`end`	`sbrk(0)`	`sbrk(0)`
chd	`sizeof(HmEntryDesc)`	`sizeof(HDR)`	`sizeof(Header)`
cal	4	`sizeof(HDR)`	`sizeof(Header)`
$csz(x)$	`(long)x->next-(long)x`	`x->size*sizeof(HDR)`	`x->s.size*sizeof(Header)`
$cst(x)$	`x->status`	–	–
$cnx(x)$	`x->next`	`(HDR*)x+x->size`	`(Header*)x + x->s.size`
fnx	–	`x->ptr`	`x->s.ptr`
fbe	–	`frhd`	`freep`
$\mathsf{fit}(c, n)$	($csz(c)$ `>((n+3)&0x0F+8))?` `((n+3)&0x0F)):` $csz(c)$)	`(n+3)/4 + 1`	`(n+3)/4 + 1`

Model-based testing We experimented model-based test case generation using the tool published in [18] which implements several methods for Event-B models. We focused on the generation of test cases that are finite sequences of calls to `alloc` and `free` and end in a fail behaviour of `free`. A first observation concerns the scalability of this tool, which is not related with its particular implementation, but with the methodology it employs, which is based on queries to SMT-solvers. We were able to generate test cases for models which are on top of our hierarchy in Figure 2. The models in the lower part, which have more complex invariants, cannot be dealt by the theories available in the SMT-solvers connected with this tool. We expect that this situation is reproduced in other model-based test case generators using different input languages. Our hierarchy is a solution for this scalability problem because it provides reasonable size abstractions for the complex models of free list DMA. A second observation is related with the concretisation of the signature $\mathbb{S}$ to the code under test. Not all the elements of $\mathbb{S}$ shall be instantiated to apply the tool: only the mappings *csz* and fit shall be fixed because they are important for inferring the parameters for the calls to `alloc`; the other elements of $\mathbb{S}$ can be dealt in a symbolic way by the tool.

Static analysis Several static analysis techniques have been developed to analyse particular DMA implementations, e.g. [5, 8, 17]. They employ complex abstractions of DMA state to capture precisely some properties of DMA, e.g., the shape of the lists, the overlapping between heap and free list. These abstractions are usually based on second order logics over graphs to capture reachability between locations and shapes of data structures. The analyses aim to infer the invariants and the pre/post-conditions of DMA methods. In this context, our models provide a sound reference for the inferred specifications and highlight the logic fragments needed to capture precisely the DMA properties. These logic fragments may inspire the design of new abstractions for such analysis.

6 Related Work and Conclusion

To our knowledge, this work is the first defining a complete hierarchy of models for the full class of list based DMA. The same approach of top-down modelling is employed in [25] to obtain the formal specification of one DMA, the TLSF DMA [20]. Our set of specifications is complete for the techniques utilized in the list based DMA; it extends our former work [9] with the buddy techniques for the heap lists.

Several projects report on the mechanical proofs using theorem provers of (partial) correctness of code for specific purpose DMA, e.g., [7, 12, 14, 19, 26]. Most of these works use Separation Logic (SL) [21] which provides a scalable and expressive reasoning framework [23]. [19] targets the verification of the TOPSY DMA using the Coq theorem prover. For this, they developed a Coq library for SL which is employed to specify only some of the invariants we provide for the heap list. The Bedrock framework [7] is another Coq library that has been used to verify DMA code with only acyclic free list and no coalescing. [26] proposes a formal memory model that captures both the low level (heap list) and the abstract level (free list) of the memory organisation in DMA. The low level model is based on the set theory available in Isabelle/HOL; the abstract level uses a fragment of SL encoded in Isabelle/HOL. The approach was used to formally verify the code of the DMA used by the L4 microkernel [14]. [12] employs Boogie and Z3 to verify a realistic garbage collector whose code has been annotated with a particular region logic. Our work is complementary to these projects. We provide reusable and complete specifications for all list based DMA by applying several refinement steps, while they focus on the verification of specifications for a particular DMA code.

Verification of DMA code by static analysis has been considered in [5, 8, 17]. All these methods infer only some properties for particular allocators. Indeed, they employ fragments of SL or some logics over arrays which are not expressive enough to cover fully the invariants of the DMA analysed (e.g., the fit policy). Our work provides reference specifications to compare with the inferred ones, in a logic fragment more general than SL. It could motivate the extension or the direct application of general purpose methods based on SL, e.g., [6, 22].

Conclusion: We propose an original methodology based on refinement to obtain formal specifications for a large class of DMA implementations, i.e., list based DMA. Our set of specifications is complete towards the set of targeted policies. We prove the correctness of models in this hierarchy and of the refinement relations between these models. We show that this hierarchy is useful to obtain code for new combination of DMA policies or to help tools for formal verification and monitoring targeting this class of DMA.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Abrial J-R. Modeling in Event-B: system and software engineering. Cambridge University Press, 2010
2 Abrial JR, Michael B, Stefan H, et al. Rodin: an open toolset for modelling and reasoning in Event-B. International journal on Software Tools for Technology Transfer, 2010. 12(6): 447-466
3 Leslie A. Memory allocation in C. Embedded Systems Programming, 2008. 35-42
4 Jonathan B. Inside memory management. http://www.ibm.com/developerworks/library/l-memory/sidefile.html, 2004
5 Cristiano C, Dino D, Peter W O, et al. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Proceedings of Static Analysis Symposium, Seoul, 2006. 4134: 282-203
6 Wei-Ngan C, Cristina D, Huu H N, et al. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program, 2012, 77(9): 1006-1036
7 Adam C. Mostly-automated verification of low-level programs in computational separation logic. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Jose, 2011. 234-245
8 Fang B and Sighireanu M. Hierarchical shape abstraction for analysis of free-list memory allocators. In: Proceedings of International Symposium on Logic-based Program Synthesis and Transformation, Edinburgh, 2016. 151-167
9 Fang B and Sighireanu M. A refinement hierarchy for free list memory allocators. In: Proceedings of ACM SIGPLAN International Symposium on Memory Management, Barcelona, 2017. 104-114
10 Fang B and Sighireanu M. A refinement hierarchy for free list memory allocators. Research Report hal-01510166, IRIF, 2017
11 George F, Christian C, Eckart Z, et al. Topsy A Teachable Operating System. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland, 2000
12 Chris H and Erez P. Automated verification of practical garbage collectors. In: Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages, Savannah, 2009. 441-453
13 Brian W K and Dennis R. The C Programming Language, Second Edition. Prentice-Hall, 1988
14 Gerwin K, Kevin E, Gernot H, et al. seL4: formal verification of an OS kernel. In: Proceedings of ACM Symposium on Operating Systems Principles, Big Sky Resort, 2009. 207-220
15 Donald E K. The Art of Computer Programming, Volume I: Fundamental Algorithms Addison-Wesley. Reading, Mass. 1973
16 Doug L. dlmalloc. ftp://gee.cs.oswego.edu/pub/misc/malloc.c, 2012
17 Liu J C and Xavier R. Abstraction of arrays based on non contiguous partitions. In: Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation, Paris, 2015. 8931: 282-299
18 Qaisar A M, Johan L, and Linas L. Model-based testing using scenarios and Event-B refinements. In Methods, Models and Tools for Fault Tolerance, Berlin: Springer, 2009. 177-195
19 Nicolas M, Reynald A, and Akinori Y. Formal verification of the heap manager of an operating system using separation logic. In: Proceedings of International Conference on Formal Engineering Methods, Macao, 2006. 4260: 400-419
20 Miguel M, Ismael R, Alfons C, et al. TLSF: A new dynamic memory allocator for real-time systems. In: Proceedings of Euromicro Conference on Real-Time Systems, Catania, 2004. 79-86
21 Peter W O, John C R, and Yang H . Local reasoning about programs that alter data structures. In: Proceedings of European Association for Computer Science Logic, Paris, 2001. 1-19
22 Qin S C , He G H, Luo C G, et al. Automatically refining partial specifications for heap-manipulating programs. Sci. Comput. Program, 2014. 82: 56-76
23 Qin S C, XU, and Ming Z. Survey of research on program verification via separation logic. Journal of Software, 2017
24 Douglas R S and Michael R L. Algorithm theories and design tactics. Sci. Comput. Program, 1990. 14(2): 305 321
25 Su W, Abrial J R, Pu G G, et al. Formal development of a real-time operating system memory manager. In: Proceedings of International Conference on Engineering of Complex Computer Systems, Gold Coast, 2015. 130-139
26 Harvey T, Gerwin K, and Michael N. Types, bytes, and separation logic. In: Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages, Nice, 2007. 97-108
27 Paul R W, Mark S J, Michael N, et al. Dynamic storage allocation: A survey and critical review. In: Proceedings of International Workshop on Memory Management, Kinross, 1995. 986: 1-116