A Refinement Hierarchy for Free List Memory Allocators

Bin Fang[2,1] and Mihaela Sighireanu[1]

[1] IRIF, University Paris Diderot and CNRS, France
[2] Shanghai Key Laboratory of Trustworthy Computing, ECNU, 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 free list class. To obtain manageable formal reasoning and reusable formal models, we organise these models in a hierarchy ranked by refinement relations. We prove the soundness of models and refinement relations using an off-the-shelf theorem prover. We demonstrate that our hierarchy is a basis for an algorithm theory for the class of free list DMA: it abstracts various existing implementations of DMA and leads to new DMA implementations. We illustrate its application to model-based code generation, testing, run-time verification, and static analysis.

Keywords: Free list memory allocators, Formal methods, Refinement, Model-based design, Formal analysis

1 Introduction

A dynamic memory allocator (DMA) is a piece of software managing a reserved region of the heap. 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 blocks of memory of variable size that it may free at any time. To offer this service, the DMA manages the reserved memory region by partitioning it into arbitrary sized blocks of memory, 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 set of chunks not in use, also called *free chunks*, is managed using different techniques. In this paper, we focus on *free list allocators* [13,23], that record free chunks in a list. This class of DMA includes textbook examples [13,11] and real-world allocators [14].

The ultimate motivation of our work is the synthesis of code for DMA which is optimal and formally proved correct. This task presents some difficulties because, as shown in [23], there is no optimal general solution to obtain DMA that provide both low overhead for the management of the memory region and high speed in satisfying

memory requests. Consequently, the design of the DMA shall take into account its specific use and adjusts the combination of techniques to obtain an optimal solution for this use. The formal methods shall be able to deal with such customisation which usually combines highly optimised 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 [5,17,22,12,6]. These projects make use of highly expressive logics to specify the memory organisation and content, e.g., second order logics or Separation Logic [19], which need sophisticated tools to be dealt with.

This paper is a first step towards this ultimate target. It surveys a full class of DMA, the list based DMA, in order to provide an *algorithm theory* [20] for it, i.e., a structure common to the implementations in this class, which abstracts away specific implementation concerns. The algorithm theory for free list DMA is given as a set of formal models organised in a hierarchy ranked by refinement relations. It establishes a *formally specified taxonomy of the techniques* employed by the implementations of list based DMA. To limit the complexity of this work, we focus on DMA without support for concurrency, i.e., used in a sequential setting.

More precisely, we start from the most abstract specification of DMA and we refine it incrementally, by introducing specific design tactics (e.g., lazy and eager coalescing of chunks, policies for choosing the fitting chunk, boundary tag technique). The order in which the design tactics are considered is a consequence of a set of refinement principles we determine in this paper. These principles are dictated by the ability to conduct correctness and refinement proofs. Hence, given a formal model M and a set T of applicable tactics for refinement, we choose a technique $t \in T$ such that t is required to be fixed for the refinement of M by tactics in $T \setminus \{t\}$. For example, we first refine the most abstract specification of DMA by the specific techniques employed for the collapsing of free chunks. Indeed, we found out that the tactics used to manage the list of free chunks (e.g., boundary tags) are bound to specific collapsing tactics (for instance, early collapsing).

We show that our taxonomy covers eleven case studies, seven of them representing open source DMA implementations (e.g., [4,3,11,14]) and the remainder being our implementations of list based DMA described by Knuth in [13]. Moreover, we get 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 verification and code generation methods could use the specific implementation of the abstraction signature $\mathbb{S}$ to reach a 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 formulas for 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. To sum up, this paper has the following contributions:

- We propose an algorithm theory for the class of free list DMA by defining a set of formal models organized in a hierarchy ranked by (formally proved) refinement relations; this set of models provides a formal specification for correctness of DMA implementations. Our hierarchy is extensible to other design tactics for the same class of DMA.
- We identify a signature representing an abstraction from implementation details of DMA; this signature is the link between the formal models proposed and the concrete implementations.
- We show how that eleven existing DMA implementations are specified by models in our hierarchy.
- We illustrate the application of this work to model-based program synthesis, testing and verification techniques for DMA implementations, mainly test case generation, run-time verification, and static analysis.

The paper is organized 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 high level memory management using the free list. Section 5 presents the abstract signature and its concretisations to the case studies considered. Section 6 illustrates the applications of this work to model-based testing and verification of DMA. We provide links with existing work and conclude in Section 7.

2 Dynamic Memory Allocators

This section overviews the service provided by DMA and the techniques and policies used in existing implementations. This survey leads to a first formal model and a set of refinement principles.

2.1 DMA Service

A conventional DMA manages a set of memory blocks in the data segment of a process in order to satisfy requests for memory issued by its clients. We call a memory block managed by the DMA a *chunk*. The DMA splits each chunk into two parts (see Figure 2): (i) a *header* used to store information about the chunk (e.g., its size), and (ii) a *body* which may store clients' data. 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.

```
void init();
bool free(void* p);
void* alloc(size_t sz);
void* realloc(void* p, size_t sz);
```

Fig. 1. Interface of DMA

The DMA communicates with its clients through the interface provided in Figure 1. The method `init` initialises the set of chunks managed and marks them all to be *free*, i.e., available for allocation requests. A call `alloc(n)` searches a free chunk whose body has size (in bytes) at least `n`. Indeed, some DMA implementations reserve more space than the requested size to obtain address alignment or to avoid memory fragmentation. If such a chunk is found, it is marked as *busy* and `alloc` returns the start address of 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 room (and returns `p`) or 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. 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 $\sigma \triangleq \langle H, F, csz, cst \rangle$

$I_1 : H \subseteq [\mathsf{hst}, \mathsf{hli}[$	domain of chunks	$I_2 : \forall c \in H \cdot c \bmod \mathsf{cal} = 0$	chunk alignment
$I_3 : \mathsf{chd} > 0$	non empty header	$I_4 : \forall c \in H \cdot csz(c) \geq \mathsf{chd}$	size includes header
$I_5 : F \subseteq H \ \wedge \ \forall c \in F \cdot cst(c) = 1 \iff c \in F$			chunk status exactly represent F
$I_6 : \forall b, c \in H \cdot c \neq b \Rightarrow [c, c + csz(c)[\cap [b, b + csz(b)[= \varnothing$			chunks are pairwise disjoint

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

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

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

2.2 A Very Abstract Specification

Table 1 includes a formal specification of the above informal description. Because `realloc` is a composition of `alloc` and `free`, we provide its formal specification in the Appendix. The memory region managed by the DMA is seen as a sequence of bytes starting at the address hst and ending before address hli. A state of the memory region is modelled by a tuple $\sigma \triangleq \langle H, F, csz, cst \rangle$ where:

- The set H represents the start addresses of chunks managed. We denote by $c, c', \ldots$ the elements of H and they represent also (unique) identifiers of chunks.
- 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 *cst*. For readability of specifications, we denote by F the set $cst^{-1}(1)$, i.e., the *set of free chunks*.

Notice that the content of chunk body is abstracted out. Therefore, the specification of `realloc` is not precise in this abstraction. We use the dotted notation, e.g., $\sigma.H$, for state components.

The properties I_1–I_6 are correctness invariants for abstract states. 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 behaviors of the DMA methods are specified by inference rules in Table 1. We denote by $\sigma \xrightarrow{m(a):r} \sigma'$ the change of state from σ to σ' produced by the call of method m (in Figure 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 the component f which values is set to e. In 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 15.

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 (or set of constants).
- *External implementation of the set data structure:* By abstracting the implementation of data structure, 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 Free List DMA

When removing the above simplifying hypotheses, we face a large variety of policies and techniques for implementing sets of chunks. This paper focuses on implementations using a *heap list* inside the managed memory region; we discuss in Section 2.6

the modelling of other classes of DMA implementations. In this section, we shortly describe the design tactics relevant for this class; we encounter them in the case studies discussed in the next section and in comprehensive surveys of memory allocators like [23]. These design tactics are summarised as columns of Table 2.

A heap list stores the chunks in sequence inside the memory region, like shown in Figure 2. 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 DMA implement a doubly-linked heap list using the principle of "boundary tag" [13], which consists in using the last bytes of a chunk (i.e., the bytes just before the start of its successor in the list) to store a copy of its status and size. This feature is represented by the column "*linked*" in Table 2, where "$\rightarrow$" (resp. "$\leftrightarrow$") means singly (resp. doubly) linked list, and "addr" (resp. "size") represents technique (i) (resp. (ii)) above.

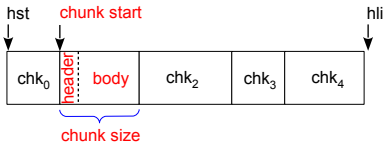


Fig. 2. Heap list

To fit a request, the DMA 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 2, where the values indicate the position of the free chunk after splitting.

Splitting of chunks may lead to memory fragmentation. There exist two main policies for defragmentation (column "*defrag.*" in Table 2), both of them join sequences of free chunks in one free chunk. *Early coalescing* policy does defragmentation during `free`; the newly free chunk is joined with its next and previous neighbours if they are free. Thus, the heap list never contains two adjacent free chunks. *Lazy coalescing* policy does defragmentation during the call of `alloc`, if none of the existing free chunks is large enough to satisfy the request.

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 DMA where the free chunks are kept in a list, called the *free list*. This 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 is the policy used to select a fitting free chunk when several are available. Possible options are: *first* fitting chunk in the heap list or the free list, chunk which fits (with smallest difference) the request, i.e., *best* fit, *next* fitting chunk with respect to the last allocation or deallocation.

2.4 Case Studies

Table 2 summarises the design tactics employed by eleven case studies implementing list based DMA. Four of these case studies (DKFF, DKBF, DKNF, DKBT) are our C implementations of algorithms proposed in [13]; the remainder are external and open source C implementations we found in research and survey papers on DMA and DMA verification. These case studies appear to us as representative for the list based DMA

Table 2. Design tactics employed in case studies

	heap list			*free list*		*fit*	*Model*
Case study	*linked*	*split*	*defrag.*	*shape*	*sorted*	*policy*	Figure 3
IBM [4]	addr, $\rightarrow$	–	–	–	–	first	**MH**
DL-small [14]	size, $\rightarrow$	–	–	–	–	first	**MH**
TOPSY [8]	size, $\rightarrow$	at end	lazy	–	–	first	**MHL**
DKFF [13]	size, $\rightarrow$	at start	early	acyclic, $\rightarrow$	yes	first	**MSAF**
DKBF [13]	size, $\rightarrow$	at start	early	acyclic, $\rightarrow$	yes	best	**MSAB**
LA [3]	size, $\rightarrow$	at start	early	acyclic, $\rightarrow$	yes	first	**MSAF**
DKNF [13]	size, $\rightarrow$	at start	early	acyclic, $\rightarrow$	yes	next	**MSAN**
KR [11]	size, $\rightarrow$	at start	early	cyclic, $\rightarrow$	yes	next	**MSC**
DKBT [13]	size, $\leftrightarrow$	at start	early	acyclic, $\leftrightarrow$	no	best	**MUAD**
DL-list [14]	size, $\leftrightarrow$	at start	early	acyclic, $\leftrightarrow$	no	best	**MUAD**
TLSF [18]	size, $\leftrightarrow$	at start	early	acyclic, $\leftrightarrow$	no	best	**MUAD**

because they illustrate all the design tactics we listed in the previous section. We provide a short overview of these allocators in the increasing order of complexity.

The first part of Table 2 contains pure (i.e., without free list) heap list DMA. The DMA TOPSY is the memory manager of the TOPSY operating system [8]. Some of its properties (e.g., heap list, chunk separation) have been already specified and proved on the code using Isabelle/HOL (see Section 7). The IBM allocator is provided in [4]. The DL-small allocator is extracted from Doug Lea's allocator [14]; it represents the part which deals with requests with size less than 256 bytes.

The case studies in the second part of Table 2 uses singly linked, address sorted, free lists. The DKFF and DKBF allocators are our implementations of algorithms A and B Section 2.5 of [13], 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 policy using the "roving pointer" technique proposed in [13] (Exercice 6 in Section 2.5). The KR allocator is the code published in [11] for a next fit, circular free list DMA.

The third part of Table 2 contains case studies using doubly linked free lists. The DKBT allocator is our implementation of the "boundary tag" technique introduced in [13] (Algorithm C in Section 2.5). 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[3] of the part of Doug Lea's allocator [14] for medium size requests. The TLSF allocator is a simplification of the code described in [18]. It distributes the free chunks in several doubly linked lists, depending on their size, in order to reduce the search for the best fitting chunk.

2.5 Mining the Case Studies for Refinement

We discuss here our findings and provide a rationale of our refinement strategy.

[3] We removed the code concerning concurrency, portability, and some optimisations.

We identify three classes of implementations, represented by the three regions of Table 2. The first class, represented by the first three lines in the table, does not use a free list; the other classes use a free list but they mainly make different choices for the shape of the heap list and the sorting of the free list. An important observation relates the design tactics with the basic operations on lists. 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. The fit policy requires to do a search in the list. For the free list, the operations required by different policies 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 observations 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 3. This hierarchy mainly includes two layers called heap-list and free-list. The heap-list layer contains three models specifying DMA with heap list and lazy (**MHL**), eager (**MHE**), or absence of (**MH**) coalescing. The models in free-list layer refined from heap-list by add free list, e.g. **MUA** specifies DMA using unsorted, acyclic singly linked lists.

Actually, Figure 3 includes only the the part of the hierarchy that covers our case studies, listed as labels of models. However, the refinement relations we defined allows to obtain more models.

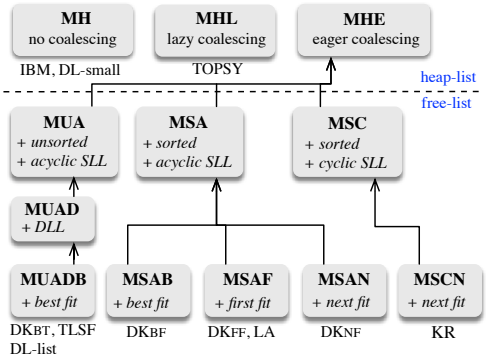


Fig. 3. The hierarchy of models and case studies it covers

We define formally the models and the refinement relations in the next sections. 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 abstract these low level design choices.

2.6 Attainable Extensions

The hypothesis of fixed limit for the data segment managed by the DMA, limit represented by hli, may be easily relaxed in our framework by including hli in the abstract state σ. This extension allows to specify precisely DMA implementations, e.g. IBM [4] and KR [11], where the method `init` is absent and the method `alloc` extends the data segment (using system calls) when the existing free chunks cannot satisfy a request.

The model **MUADB** may be further refined to specify precisely the policy proposed in [18], which indexes free chunks in several lists depending on their size.

Our hierarchy is not suitable to model the buddy allocation technique, which uses a tree organisation for the set of chunks.

3 Heap List Modelling

This section presents models of DMA using a heap list for the set of managed chunks. We show that these models may be obtained by composing refinements of basic operations on heap lists. These refinements are mainly discriminated by their handling of the chunk coalescing.

3.1 State and Invariants

The abstract state of a DMA using a heap list organisation of chunks is defined Table 3. It includes the set of chunks H and the information stored in the chunk header, i.e., *csz* and *cst* (or equivalently F) defined in Table 1. In addition, it stores 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 1, we introduce the invariants I_7 and I_7' in Table 3 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 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 2. A heap list satisfying I_1–I_9 has no memory leaks.

The invariant I_{ec} is not satisfied by all heap lists, but we introduce it here for sake of completeness. It asserts that two chunks, successive in the heap list, cannot be both free and it is a state invariant (correctness criteria) for DMA using early coalescing. Notice that I_{ec} may be temporary broken during the execution of methods `free` and `realloc`.

Table 3. Refinement of A for heap list

State refinement	
$cnx : H \to (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\}$	next chunk total mapping
$cpr : (H \setminus \{\mathsf{hst}\}) \cup \{\mathsf{hli}\} \to H$	previous chunk total mapping
$\sigma \triangleq \langle H, F, csz, cst, cnx, cpr \rangle$	state of DMA with heap list

Additional invariants

$I_7 : cnx$ is a bijection $I_8 : \mathsf{hst} \in H$ $I_9 : \forall c \in H \cdot cnx(c) = c + csz(c)$

$I'_7 : cpr = cnx^{-1}$ $I_{ec} : \forall c_1, c_2 \in H \cdot \big(cnx(c_1) = c_2\big) \Rightarrow \mathtt{NAND}(cst(c_1), cst(c_2))$

3.2 Basic Heap List Operations

For a heap list organisation of H, the inference rules in Table 1 (in particular "Alloc^S", "Alloc^F" and "Free^S") fail to model the splitting of a fitting chunk or the merging of adjacent free chunks in case of an allocation fail (in lazy coalescing) or free success (in early coalescing). To define the specifications of heap-list based DMA methods, we identified 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$_N$ merges a free chunk with its near free neighbours,
- hmerge$_\forall$ merges all sequences of free chunks, and
- hsearch searches a fitting free chunk.

Tables 4–7 provides the specifications of these basic operations for singly linked heap lists. For some operations, e.g., hsplit or hsearch, several refinements are shown. The main methods of the DMA are specified in Table 8 using these basic operations. In this section, we explain these specifications in detail. *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 notations.*

Table 4 specifies operations hremove and hinsert. The state of the chunk c given as parameter is updated accordingly.

Table 4. Refinement of remove/insert operations on heap list

hremove(c): hremove $\dfrac{c \in F}{\sigma \xrightarrow{\mathsf{hremove}(c)} \sigma[F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0]}$

hinsert(c): hinsert $\dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hinsert}(c)} \sigma[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1]}$

The operation hsplit 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. Notice that, this operation preserves the validity of the invariant I_{ec} because it sets one part of the split chunk to busy. Table 5 specifies two tactics for hsplit (hsplit$_B$ and hsplit$_E$) depending on the position of the chunk of size n (at begin resp. end of c).

Table 5. Refinement of split operation on heap list

$\mathsf{hsplit}(c,n) : c'$

$$\mathsf{hsplit}_B\ \frac{c \in F \quad 0 < n \leq csz(c) \quad c' = c + n \quad \left\langle \begin{array}{l} H \cup \{c'\}, F, \\ csz[c \leftarrow n, c' \leftarrow csz(c) - n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle = \sigma_1 \quad \sigma_1 \xrightarrow{\mathsf{hremove}(c)} \sigma_2 \xrightarrow{\mathsf{hinsert}(c')} \sigma_3}{\sigma \xrightarrow{\mathsf{hsplit}(c,n):c} \sigma_3} \qquad \mathsf{hsplit}_E\ \frac{c \in F \quad 0 < n \leq csz(c) \quad c' = c + csz(c) - n \quad \left\langle \begin{array}{l} H \cup \{c'\}, F, cst(c') \leftarrow 0, \\ csz[c \leftarrow csz(c) - n, c' \leftarrow n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle = \sigma_1}{\sigma \xrightarrow{\mathsf{hsplit}(c,s):c'} \sigma_1}$$

The operation hmerge is called in `free` or `realloc` to join two successive free chunks in one. But a state with two successive free chunks does not comply with the invariant I_{ec} satisfied in states of DMA with early coalescing. Therefore, the invariant I_{ec} is broken temporarily in a state before hmerge. As discussed at the end of the previous section, I_{ec} is reestablished at the return from `free` or `realloc`.

The first behaviour of hmerge is hmerge_R, defined in Table 6 by rules hmerge^S_R and hmerge^F_R. The operation joins its parameter, a chunk b, with its successor c (also right neighbour) if c is free; otherwise, the operation does nothing. For sake of symmetry with the second behaviour, hmerge_R returns its parameter. The second behaviour of hmerge is hmerge_L, defined by rules hmerge^S_L and hmerge^F_L. It joins its parameter b with its predecessor c (also left neighbour), if c is free and returns the address of this predecessor; otherwise, the operation returns b.

Table 6. Refinement of single merge operation on heap list

$\mathsf{hmerge}_R(b) : x$

$$\mathsf{hmerge}^S_R\ \frac{b \in F \quad c = cnx(b) \quad c \in F \quad \sigma \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \left[\begin{array}{l} H \leftarrow H \setminus \{c\}, \\ csz[b \leftarrow csz(b) + csz(c)], \\ cnx[b \leftarrow cnx(c)] \end{array} \right]} \qquad \mathsf{hmerge}^F_R\ \frac{b \in F \quad c = cnx(b) \quad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma}$$

$\mathsf{hmerge}_N(b)$

$$\mathsf{hmerge}^S_N\ \frac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \xrightarrow{\mathsf{hmerge}_L(b):c} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_2} \qquad \mathsf{hmerge}_\forall \qquad \mathsf{hmerge}^S_\forall\ \frac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_1 \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}$$

The above operations are called by hmerge_N and $\mathsf{hmerge}_\forall$, which are specified in Table **??**. Operation hmerge_N merges a free chunk b with its free neighbours. Operation $\mathsf{hmerge}_\forall$ merges any two successive free chunks in the entire memory region. Rule $\mathsf{hmerge}^F_\forall$ states property I_{ec} is satisfied and it is the terminal point of $\mathsf{hmerge}_\forall$. The operation hmerge_N will be used for eager coalescing policy, while $\mathsf{hmerge}_\forall$ is used for lazy coalescing policy.

Operation $\mathsf{hsearch}(n)$ looks through the heap list for a chunk fitting the requested size n and returns a fitting free chunk or nil. Table 7 defines refinements of this operation, $\mathsf{hsearch}_{FF}$ and $\mathsf{hsearch}_{BF}$ corresponding to the first-fit and best-fit policies. Notice that the first fit is computed from the start of the heap list, i.e. hst.

3.3 Models for Heap List DMA

The specifications of main DMA methods (from Figure 1) make use of basic operations presented above as follows.

Table 7. Refinement of search operation on heap list

$\mathsf{hsearch}(n) : c$	$\mathsf{hsearch}^{S}_{FF}\ \dfrac{\begin{array}{c} c \in F \qquad \mathsf{fit}(c,n) \leq csz(c) \\ \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b,n) \end{array}}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$ $\quad \mathsf{hsearch}^{F}_{*F}\ \dfrac{\forall b \in F \cdot csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} \sigma}$

The specification of the method `init` for all heap list models is given in section hinit in Table 8. Notice that the rule refines the abstract rule `init` in Table 1 by adding the information about the successor and predecessor of the initial chunk in the heap list.

The `alloc` method is refined to obtain two distinct behaviours: $\mathsf{halloc}_{\text{eager}}$ and $\mathsf{halloc}_{\text{lazy}}$ for allocation without resp. with coalescing. Both of these behaviours call the internal operation halloc_i, which does the main part of the work: it searches the free chunk fitting the request using hsearch and returns this chunk after changing its status. The rule $\mathsf{halloc}_{\text{fit}}$ specifies the behaviour where the fitting chunk is not split; the rule $\mathsf{halloc}_{\text{split}}$ specifies the splitting operations. Notice that the first rule allows to define models for allocation without splitting due to the use of the fit mapping which returns always the size of the chunk if it fits the request. For $\mathsf{halloc}_{\text{lazy}}$, i.e., allocation with coalescing, there are two cases, specified by rules $\mathsf{halloc}^{S}_{\text{lazy}}$ and $\mathsf{halloc}^{S'}_{\text{lazy}}$, depending on the returnee of hsearch. Rule $\mathsf{halloc}^{S}_{\text{lazy}}$ states that allocator merges two continuous free chunks when there is no big enough free chunk after searching (hsearch returns nil). The rule $\mathsf{halloc}^{S'}_{\text{lazy}}$ is applied when hsearch returns a fitting free chunk.

Table 8. Refinements of methods for heap list

hinit()	$\mathsf{hinit}\ \dfrac{}{\sigma \xrightarrow{\mathsf{hinit}()} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst} \end{array} \right\rangle}$
$\mathsf{halloc}_i(s) : p$	$\mathsf{halloc}^{S}_{\text{split}}\ \dfrac{c \neq \mathsf{nil} \qquad p = b + \mathsf{chd} \qquad \mathsf{fit}(c,s) < csz(c) \qquad \sigma \xrightarrow{\mathsf{hsearch}(s):c} \sigma_1 \xrightarrow{\mathsf{hsplit}(c,\mathsf{fit}(c,s)):b} \sigma_2}{\sigma \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_2}$
$\mathsf{halloc}(s) : p$	$\mathsf{halloc}_{\text{eager}}\ \dfrac{\sigma \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_1}{\sigma \xrightarrow{\mathsf{halloc}(s):p} \sigma_1}$ $\quad \mathsf{halloc}^{S}_{\text{lazy}}\ \dfrac{\sigma \xrightarrow{\mathsf{halloc}_i(s):\mathsf{nil}} \sigma_1 \xrightarrow{\mathsf{hmerge}_{\forall}} \sigma_2 \xrightarrow{\mathsf{halloc}_i(s):p} \sigma_3}{\sigma \xrightarrow{\mathsf{halloc}(s):p} \sigma_3}$

The specification of the method `free` is refined similarly to obtain its behaviours for eager and lazy coalescing (see Appendix). We only show below the rule used for early coalescing. It recuperates the invariant I_{ec} by calling the merging of neighbours of the freed chunk before the return of `free`:

$$\mathsf{hfree}^{S}_{\text{eager}}\ \frac{p = b + \mathsf{chd} \qquad b \in H \setminus F \qquad \sigma \xrightarrow{\mathsf{hinsert}(b)} \sigma_1 \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_2}{\sigma \xrightarrow{\mathsf{hfree}(p):\mathtt{true}} \sigma_2}$$

We coded the above specifications in Event-B [1] and we proved with the Rodin tool [2] and the connected solvers the following correctness theorem. Table 9 provides statistics about size of the models and of proofs conducted to obtain their correctness.

Theorem 1. *Every operation of the DMA model **MH** (resp. **MHL** and **MHE**) preserves the invariants of the model.*

Table 9. Invariants / rules used in models and Statistics on proving of models

Models	`init`, `alloc`, `free`	remove	split	search
MH	hinit, $\mathsf{halloc}_{\text{eager}}$, $\mathsf{hfree}_{\text{lazy}}$	hremove	hsplit_B	$\mathsf{hsearch}_{FF}$
MHL	hinit, $\mathsf{halloc}_{\text{lazy}}$, $\mathsf{hfree}_{\text{lazy}}$	hremove	hsplit_B	$\mathsf{hsearch}_{FF}$
MHE	hinit, $\mathsf{halloc}_{\text{eager}}$, $\mathsf{hfree}_{\text{eager}}$	hremove	hsplit_E	$\mathsf{hsearch}_{FF}$

Model	LOC	Proof Obligations	Automatically discharged	Interactive proofs
MH	114	39	27(69%)	12(31%)
MHL	176	8	8(100%)	0(0%)
MHE	183	82	58(70%)	24(30%)

3.4 Model-based Translation to C Code

The rules presented in the previous section provide an operational semantics of DMA methods that we exploit to generate C code. Like in rules, the code of DMA methods calls basic operations on heap list, for which we provide several implementations, following the rules presented for them. We illustrate these principles on the `alloc` method because we presented almost all basic operations it uses. For model **MHE**, the code of `alloc` is given by the rule $\mathsf{halloc}_{\text{eager}}$ which is translated into a call of the halloc_i function whose code, provided in Figure 4, is obtained from the rules in Table 8. The type `hdr_t` is a C structure having as fields the header informations. The rules $\mathsf{halloc}^*_{\text{lazy}}$ for `alloc` method in model **MHL** lead to the code in Figure 4.

The direct translation in C of two rules for $\mathsf{hmerge}_\forall$ is a loop iterating over the chunks in the free list, picking a free chunk with free neighbours, and doing the merge; the loop stops when no merging is possible for any free chunk. For this, we add to the C functions translating the rules for merging (hmerge_N, hmerge_L, and hmerge_R) a boolean parameter indicating if any successful merge has been applied (using any of rules hmerge^S_L or hmerge^S_R). The interesting parts of the code generated is provided in Figure 5. Notice that this translation is not the most efficient, but it follows closely the rules given. The other basic operations are translated similarly.

```
void* halloc_i(size_t sz) {
  hdr_t* c = hsearch(sz);
  if (c == NULL) return NULL;
  if (fit(c,sz) == csz(c)) {
    hremove(c); return c+chd;
  } else {
    hdr_t* b = hsplit(c, fit(c,sz));
    return b+chd;
  }
}
```

```
void* alloc(size_t sz) {
  void* p = halloc_i(sz);
  if (p != NULL) return p;
  else {
    hmergeAll();
    return halloc_i(sz);
  }
}
```

Fig. 4. Code generated for `alloc` based on model **MHL**

```
void hmergeAll() {
  bool is_ec;
  do {
    is_ec = true;
    for (hdr_t* c=hst; c<hli; c=cnx(c))
      if (cst(c) == 1) { // c is free
        bool done_merge = false;
        hmergeN(c, &done_merge);
        is_ec = is_ec && !is_merge;
      }
  } while (!is_ec);
}
void hmergeN(hdr_t* b, bool* done_merge) {
  bool done_mergeR, done_MergeL;
  hmergeR(b, &done_mergeR);
  hmergeL(b, &done_mergeL);
  *done_merge = done_mergeR||done_mergeL;
}
```

```
hdr_t* hmergeR(hdr_t* b, bool* done_merge) {
  hdr_t* c = cnx(b);
  if (cst(b) == 1 && cst(c) == 1) {
    hremove(c);
    csz_update(b, csz(b)+csz(c));
    cnx_update(b, cnx(c));
    *done_merge = true; return c;
  } else {
    *done_merge = false; return b;
  }
}
```

Fig. 5. Code generated for $\mathsf{hmerge}_\forall$ and hmerge_N

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. To simplify the presentation, we provide only definitions for refinements of the **MHE** model, i.e., model using a heap list with early collapsing.

Table 10. Impact of refinement steps

Values	*State*	*Basic operations*	*Methods*
A	fnx_A	$\mathsf{fsearch}_A$	finit_A
C	fnx_C, I_C	$\mathsf{fsearch}_C$	finit_C
U		$\mathsf{finsert}_U, \mathsf{fmerge}_U$	
S	I_S	$\mathsf{finsert}_S, \mathsf{fmerge}_S$	
D	σ_D, I_{fpr}	$\mathsf{fremove}_D, \mathsf{finsert}_D$ $\mathsf{fmerge}_D, \mathsf{fsplit}_D$	finit_D
F,B		$\mathsf{fsearch}_F, \mathsf{fsearch}_B$	
N	σ_N, I_{rp}	$\mathsf{fsearch}_N$	$\mathsf{finit}_N, \mathsf{falloc}_N$

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_N, $\mathsf{fmerge}_\forall$, and fsearch. The refined models of the DMA methods (in Figure 1) employ these basic operations similarly to the models for heap list. We define four steps of refinement dealing with:

i. shape of the list, with values acyclic (A) and cyclic (C),
ii. ordering of chunks, with values unordered (U) and sorted (S),
iii. free cells linking, with values singly (default) and doubly (D),
iv. fit policy, with values first (F), best (B) and next (N) fit.

Each step adds new state elements, state invariants, or refinements of basic operations and DMA methods. Table 10 sums up the impact of each step using the notations introduced in the following.

4.1 States and Invariants

Table 11 defines the different states and the invariants used by the refinement steps. Notice that a free list state extends a state of the heap list model. The linking of free

Table 11. States and invariants used by free list refinements

Refined states	
$fbe, fen \notin H$ (constants used as flags)	
$fnx_A : (F \cup \{fbe\}) \rightarrow (F \cup \{fen\})$	$fpr_A : (F \cup \{fen\}) \rightarrow (F \cup \{fbe\})$
$fnx_C : (F \cup \{fbe, fen\}) \rightarrow (F \cup \{fbe, fen\})$	$fpr_C : (F \cup \{fbe, fen\}) \rightarrow (F \cup \{fbe, fen\})$
$\sigma \triangleq \langle H, F, csz, cst, cnx, fnx \rangle$	$\sigma_D \triangleq \langle H, F, csz, cst, cnx, fnx, fpr \rangle$
$\sigma_N \triangleq \langle H, F, csz, cst, cnx, fnx, rp \rangle$	

Additional invariants	
$I_{fnx} : fnx_*$ total bijection	$I_{fpr} : fpr = fnx^{-1}$
$I_\varnothing : fnx_*(fbe) = fen \iff F = \varnothing$	$I_{\ell s} : \forall F' \subseteq F \cdot F' \subseteq fnx_*^{-1}(F') \Rightarrow F' = \varnothing$
$I_C : fnx_C(fen) = fbe$	$I_{rp} : F \neq \varnothing \Rightarrow rp \in F$
$I_S : \forall c \in F \cdot fnx_*(fbe) \leq c \leq fnx_*^{-1}(fen) \wedge (fnx_*(c) = fen \vee c < fnx_*(c))$	

chunks is modelled using the bijective mappings *fnx* and *fpr* specified by invariants 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*, not chunks in H, which delimit the start resp. target (end) of the free list. (We provide intuition for free list in Figure B.3 of the Appendix.) Indeed, by using these flags, we could employ the invariant $I_{\ell s}$ for both cyclic and acyclic lists to ensure the following property:

Property 3. If a state satisfies I_{fnx}, $I_\varnothing$ (and I_C), and $I_{\ell s}$ then *fnx* defines an acyclic (resp. cyclic) list starting in $fnx(fbe)$ 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 one if the free list is address sorted, property specified by the invariant I_S. Indeed, the following property is a corollary of *fnx* being total, bijective and strictly increasing:

Property 4. If a state satisfies I_{fnx}, $I_\varnothing$ (and I_C), and I_S then *fnx* defines an acyclic (resp. cyclic) list starting in $fnx(fbe)$ and including all free chunks.

For models using unsorted free lists, we use the invariant $I_{\ell s}$. For the case studies in Table 2, these models are mainly using doubly linked free lists. Fortunately, Rodin provides means for proving second order logic properties on sets.

The variable *rp* is used by the state σ_N modelling the next fit policy to mark the last used free chunk.

4.2 Basic Operations

Table 12. Refinement of basic operations on free list

fremove(c)	$\mathsf{fremove}_A$	$\dfrac{c \in F}{\sigma \xrightarrow{\mathsf{fremove}(c)} \sigma\left[F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0, fnx(fnx^{-1}(c)) \leftarrow fnx(c)\right]}$
finsert(c)	$\mathsf{finsert}^B_U$	$\dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{finsert}(c)} \sigma\left[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, fnx(fbe) \leftarrow c, fnx(c) \leftarrow fnx(fbe)\right]}$
	$\mathsf{finsert}^B_S$	$\dfrac{c \in H \setminus F \qquad \forall b \in F \cdot c < b}{\sigma \xrightarrow{\mathsf{finsert}(c)} \sigma\left[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1, fnx(fbe) \leftarrow c, fnx(c) \leftarrow fnx(fbe)\right]}$
fsearch(n) : c	$\mathsf{fsearch}^S_{FF}$	$\dfrac{c \in F \qquad \mathsf{fit}(c,n) \leq csz(c) \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{fsearch}(n):c} \sigma}$
	$\mathsf{fsearch}^F_*$	$\dfrac{\forall b \in F \cdot csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{fsearch}(n):\mathsf{nil}} \sigma}$

To fix the parallel between the basic operations in heap and free lists, we provide the operations for free chunk removing, insertion and searching in Table 12. (The full specification of free list operations is given in Appendix B.)

Operation fremove for acyclic free list extends hremove by updating the relation *fnx*. For insertion in an unsorted free list, rules $\mathsf{finsert}_B$ and $\mathsf{finsert}_E$ specify the tactics of inserting at the start resp. end of the list. For a free list ordered by the start addresses of chunks, rules for $\mathsf{finsert}_S(c)$ search for the free chunk just before c: it can be at the start of the list or in the middle. It is obvious that fsearch refines hsearch for the first fit and best fit policies. The next fit policy uses the variable *rp* to start the search of the fitting chunk.

4.3 Models for Free List DMA

We developed several models refining the free list, including the models in Figure 3. The DMA methods are specified in a way very similar to the one used for heap lists, as could be seen on Table 13 for the methods `init` and `free` of the model **MSA**. The rule $\mathsf{ffree}^S_{\text{eager}}$ uses the operation finsert (instead of hinsert) to update the links used by the free list and then tries to merge the inserted chunk with its neighbours. Notice that the merging operation has a simpler formulation when the free lists are doubly linked.

Table 14 provides the main ingredients used by the refinement to obtain the models presented in Figure 3. Like for the heap list models, we prove with the Rodin tool the following correctness and refinement theorem. Table 14 provides statistics about models and proofs conducted to obtain the below theorem.

Table 13. Refinement of methods for free list

$$\mathsf{finit}() \quad \mathsf{finit}_A \;\frac{c \in F}{\sigma \xrightarrow{\mathsf{finit}_A()} \langle H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, \mathit{cnx}(\mathsf{hst}) \leftarrow \mathsf{hli}, \mathit{csz}(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \mathit{fnx}(\mathsf{hst}) \leftarrow \mathsf{nil} \rangle}$$

$$\mathsf{ffree}(p) : t \quad \mathsf{ffree}^S_{\mathsf{eager}} \;\frac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1 \xrightarrow{\mathsf{fmerge}_N(b)} \sigma_2}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{true}} \sigma_2}$$

$$\mathsf{ffree}^S_{\mathsf{lazy}} \;\frac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{true}} \sigma_1} \qquad \mathsf{ffree}^F_* \;\frac{\forall b \in H \setminus F \cdot p \neq b + \mathsf{chd}}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{false}} \sigma}$$

Table 14. Invariants / rules used in models and Statistics on proving of models

Models	State&Inv.	`init`	remove	search
MUA	σ	finit_A	$\mathsf{fremove}_A$	$\mathsf{fsearch}_{BF}$
MSA	σ, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{BF}$
MSCN	σ, I_C, I_S, I_{rp}	finit_C	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{NF}$
MUADB	σ_D, I_{fpr}	finit_D	$\mathsf{fremove}_D$	$\mathsf{fsearch}_{BF}$
MSAB	σ, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{BF}$
MSAF	σ, I_S	finit_A	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{FF}$
MSAN	σ_N, I_S, I_{rp}	finit_N	$\mathsf{fremove}_S$	$\mathsf{fsearch}_{NF}$

Model	LOC	Proof Obligations	Automatically discharged	Interactive proofs
MUA	219	36	30(83%)	6(17%)
MSA	197	41	27(66%)	14(34%)
MUC	205	37	30(82%)	7(18%)
MSCN	194	40	36(88%)	4(12%)
MUADB	241	9	9(100%)	0(0%)
MSAB	202	2	2(100%)	0(0%)
MSAF	202	2	2(100%)	0(0%)
MSAN	200	2	2(100%)	0(0%)

Theorem 2. *Every operation of a model for DMA with free list preserves the invariants of the model. Moreover, the refinement relations in Figure 3 are valid.*

4.4 Model-based Translation to C Code

We use principles similar to heap list in order generate code from the models. As expected, sorted (cyclic or acyclic) free list are well adapted for merging operations.

We observe that low level code (e.g., pointer arithmetics) is concentrated on basic operations for splitting or merging free chunks. Therefore, software verification tools without support for such low level code should (i) either focus on code generated for models without coalescing or chunk splitting (e.g., ones derived from **MH**), (ii) or abstract these operations to high level behaviour.

5 Refinement towards DMA Implementations

This section presents how to refine the models defined in the previous sections to obtain specifications for real code, including our case studies.

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 *flst*). Let $\mathbb{S}$ denote the above set of symbols.

The refinement to code relation is defined by the instantiation of each element in $\mathbb{S}$ by an expression using the types and variables of the DMA implementation such that the semantics of the element is fulfilled. We provide two examples of such refinement

relations in Table 15. Notice that some elements of $\mathbb{S}$ may be left unspecified (denoted by '$-$') if they are not used in the model (we omit the elements of $\mathbb{S}$ which are not specified in both examples). For example, the model **MSA** (using early coalescing and acyclic sorted free list) does not use the mappings heap previous *cpr* and free previous *fpr*. Therefore, the refinement relation provided for LA [3] does not use them. The refinement relations for every benchmark are provided in Appendix C. We obtain these refinement do code relations by inspecting the code of each allocator. However, we believe that some automatic analysis may be designed to extract some of this information. In all cases, the expression provided for some element of $\mathbb{S}$ shall be checked to ensure that it satisfies the typing constraints of its definition.

Table 15. Examples of refinement to code

$\mathbb{S}$	TOPSY [8] $\models$ **MHL**	LA [3] $\models$ **MSA**$_{FF}$	KR [11] $\models$ **MSC**$_{FF}$
hst	`start`	`_hsta`	`&begin`
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`
fit(c, n)	$(csz >$ `((n+3)&0x0F+8))?` `((n+3)&0x0F)):`csz	`(n+3)/4 + 1`	`(n+3)/4 + 1`

For model-based code generation, the elements of $\mathbb{S}$ are provided as C macro-definitions. We also generate code for the updating of each element, e.g., `csz_update` used in Figure 5.

6 Additional Applications

In the previous sections, we illustrated the application of our hierarchy of models to model-based code generation for free list DMA. We survey in this section other applications of our work for formal verification and monitoring.

Model-based testing We experimented model-based test case generation using the tool [16] 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 are able to generate test cases for models which are on top of our hierarchy in Figure 3. 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. Thus, 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 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, mainly because they are important for inferring the parameters for calls to `alloc`. The remaining elements of $\mathbb{S}$ can be dealt in a symbolic way by the tool.

Run-time verification In our case studies, we observed a common practice of DMA implementers which consists on coding methods checking that the heap list and the free list are well formed, e.g., chunks are separated, or all chunks flagged as free are registered in the free list. A systematic application of this practice leads to executable specifications for the invariants and the pre/post conditions of DMA methods. We explore this direction by translating manually in C code the state invariants presented in Tables 1, 3, and 11. For example, the code generated for the invariant I_S is:

```
for (c=fnx(fbe), i=0; c!=fen && i<MAX; c=fnx(c), i++)
      if (fnx(c)==fen || c<fnx(c)) continue;
      else error();
```

Notice the use of the counter to avoid infinite iteration due to a bad shape of the list. The manual translation of invariants is not a difficult task because these invariants are shared between models. Moreover, the use of the abstract signature $\mathbb{S}$ has the advantage to change easily the code for a particular implementation of DMA state. Of course, for sequential DMA, the run time monitoring we propose deprecates the DMA performances, so its use is limited test to validation phases.

We used this code to monitor the code we generated from models and the case studies we implemented, e.g. DKBT or DKBF. An interesting problem, which is out of the scope of this paper, is the automatic generation of this code, especially for the pre/post-conditions of DMA methods. Existing techniques for code generation from Event-B models, e.g. [9], consider a limited fragment of the modelling language, e.g., they do not translate properties like total bijection needed by *fnx*.

Static analysis Several static analysis techniques have been proposed to analyse DMA code (see related work). Due to the use of complex abstract domains, they are able to capture precise properties of DMA, e.g., shape of the lists including the overlapping between heap and free list. These domains are based on second order logics over graphs (to encode reachability), e.g., Separation Logic [19]. Ideally, these tools shall be able to infer the invariants and the pre/post-conditions of DMA methods. Therefore, our work may be exploited to provide the abstract model which fits the code analyzed. Moreover, our formal models of DMA indicate the kind of logic fragments needed to capture precisely the properties of DMA and thus is a source of inspiration for the design of more precise abstract domains.

7 Related Work and Conclusion

To our knowledge, this work is the first defining a hierarchy of models for the full class of list-based DMA. The work in [21] also does a top-down modelling but only to obtain the formal specification of the TLSF DMA [18]. They apply ten refinement steps and

use also Event-B for a DMA with a fixed size region. The most refined specification is used to (manually) generate code and to verify it using VCC. We are providing specification for a larger class of DMA, but we are not concerned by the code generation from our models; moreover, some modelling choice, e.g., use of basic operations on lists are different from ours.

Several projects report on the mechanical proofs of (partial) correctness of a specific memory allocator using theorem provers, e.g., [17,22,12,10,6]. [17] reports on a first attempt to verify the TOPSY DMA using the Coq theorem prover. For this, they developed a Coq library for the Separation Logic (SL) [19] and specify mainly the invariants of the heap list in this logic. The verification process revealed some bugs (issued from pointer arithmetics) of previous versions of this implementation. Although SL is very good to express properties like absence of chunk overlapping or memory leaks, it is not enough to specify some properties related with the content of the list, e.g., the fitting policy.

[22] proposes a formal memory model that captures both low level and abstract level of the memory organisation. The low level model is based on some set theory available in the prover language; the abstract level uses SL. The proof of abstract correctness properties are then either done at the abstract level for some operations using the proof assistant Isabelle/HOL or lifted automatically to the low level for low level operations. The approach was used to formally verify the memory allocator of the L4 microkernel and in the seL4 project [12]. Our work is complementary to this project: our specifications provide insights on the theories needed to cover other DMA and more complex properties.

The Bedrock framework [6] is a Coq library that allows to verify abstract specifications in SL over assembly and low level code. It has been applied to verify a free list based memory allocator with acyclic free list and without coalescing which is also specified in our hierarchy. The library includes the lemmas needed to automatically obtain the proof of the code.

Several works concern the static analysis of memory allocators [5,15,7]. They are design to infer properties of some allocators, e.g., DKFF, DKBF, DKNF, and KR for [5,7] and Minix 1.1 for [15]. These properties belong to SL or to some logic over arrays which are not expressive enough alone to cover fully the invariants of the DMA analysed (e.g., the fit policy). [7] proposes an analysis based on abstract interpretation using domains based on SL that is able to infer some invariants proposed in our models, e.g., I_S.

Conclusion: We propose an original methodology based on refinement to obtain first-order specifications for a large class of DMA implementations, i.e., list-based DMA. We apply this methodology and obtain a hierarchy of models that may be concretised to obtain new models and models for existing DMA implementations. We prove the correctness of the models in this hierarchy and of the refinement relation. 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.

References

1. J.-R. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
2. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.
3. L. Aldridge. Memory allocation in C. *Embedded Systems Programming*, pages 35–42, August 2008.
4. J. Bartlett. Inside memory management. `http://www.ibm.com/developerworks/library/l-memory/sidefile.html`, 2004.
5. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.
6. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.
7. B. Fang and M. Sighireanu. Hierarchical shape abstraction for analysis of free-list memory allocators. In *LOPSTR*, LNCS. Springer, 2016.
8. G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner. Topsy – A Teachable Operating System. Technical report, Computer Engineering and Networks Laboratory, ETH Zürich, Switzerland, 2000.
9. A. Fürst, T. S. Hoang, D. Basin, K. Desai, N. Sato, and K. Miyazaki. Code generation for event-b. In *iFM*, pages 323–338. Springer, 2014.
10. C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, pages 441–453. ACM, 2009.
11. B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.
12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.
13. D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
14. D. Lea. `dlmalloc`. `ftp://gee.cs.oswego.edu/pub/misc/malloc.c`, 2012.
15. J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.
16. Q. A. Malik, J. Lilius, and L. Laibinis. Model-based testing using scenarios and Event-B refinements. In *Methods, Models and Tools for Fault Tolerance*, pages 177–195. Springer, 2009.
17. N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *ICFEM*, volume 4260 of *LNCS*, pages 400–419. Springer, 2006.
18. M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *ECRTS*, pages 79–86. IEEE Computer Society, 2004.
19. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, LNCS, pages 1–19. Springer, 2001.
20. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2):305 – 321, 1990.
21. W. Su, J. Abrial, G. Pu, and B. Fang. Formal development of a real-time operating system memory manager. In *ICECCS*, pages 130–139. IEEE, 2015.

22. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108. ACM, 2007.

23. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, volume 986 of *LNCS*, pages 1–116. Springer, 1995.

A Heap List Modeling

A.1 Rules of basic operations in heap-list

Table 16. Refinements of remove/insert operations on heap list

$\mathsf{hremove}(c)$	$\mathsf{hremove}\ \dfrac{c \in F}{\sigma \xrightarrow{\mathsf{hremove}(c)} \sigma[F \leftarrow F \setminus \{c\}, csz(c) \leftarrow 0]}$	$\mathsf{hinsert}(c)$	$\mathsf{hinsert}\ \dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hinsert}(c)} \sigma[F \leftarrow F \cup \{c\}, csz(c) \leftarrow 1]}$

Table 17. Refinements of split operation on heap list

$\mathsf{hsplit}(c,n) : c'$	
	$\mathsf{hsplit}_B\ \dfrac{c \in F \qquad 0 < n < csz(c) \qquad c' = c + n \qquad \left\langle \begin{array}{l} H \cup \{c'\}, F, csz[c \leftarrow n, c' \leftarrow csz(c) - n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle \xrightarrow{\mathsf{hremove}(c);\mathsf{hinsert}(c')} \sigma_1}{\sigma \xrightarrow{\mathsf{hsplit}(c,n):c'} \sigma_1}$
	$\mathsf{hsplit}_E\ \dfrac{c \in F \qquad 0 < n < csz(c) \qquad c' = c + csz(c) - n}{\sigma \xrightarrow{\mathsf{hsplit}(c,s):c'} \left\langle \begin{array}{l} H \cup \{c'\}, F, cst(c') \leftarrow 0, \\ csz[c \leftarrow csz(c) - n, c' \leftarrow n], cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle}$

Table 18. Refinements of merge operation on heap list

hmerge$(b) : x$

$$\mathsf{hmerge}_R^S \;\frac{b \in F \qquad c = cnx(b) \qquad c \in F \qquad \langle\, H \setminus \{c\}, F, cnx[b \leftarrow cnx(c)], csz[b \leftarrow csz(b) + csz(c)] \,\rangle \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma_1}$$

$$\mathsf{hmerge}_R^F \;\frac{b \in F \qquad c = cnx(b) \qquad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma}$$

$$\mathsf{hmerge}_L^S \;\frac{b \in F \qquad cnx(c) = b \qquad c \in F \qquad \langle\, H \setminus \{b\}, F, csz[b \leftarrow csz(b) + csz(c)], cnx[b \leftarrow cnx(c)] \,\rangle \xrightarrow{\mathsf{hremove}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}(b):c} \sigma_1}$$

$$\mathsf{hmerge}_L^F \;\frac{b \in F \qquad cnx(c) = b \qquad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma}$$

hmerge$_N(b)$

$$\mathsf{hmerge}_N^S \;\frac{b \in F \qquad \sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \qquad \sigma_1 \xrightarrow{\mathsf{hmerge}_L(b):c} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_N(b):b \mid c} \sigma_2}$$

hmerge$_\forall$

$$\mathsf{hmerge}_\forall^S \;\frac{b \in F \qquad \sigma \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_1 \qquad \sigma_1 \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}$$

$$\mathsf{hmerge}_\forall^F \;\frac{I_{ec}}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma}$$

Table 19. Refinements of search operations on heap list

$\mathsf{hsearch}(n) : c$	$\mathsf{hsearch}^S_{FF}\ \dfrac{c \in F \quad \mathsf{fit}(c,n) \leq csz(c) \quad \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$ $\mathsf{hsearch}^S_{BF}\ \dfrac{c \in F \quad \mathsf{fit}(c,n) \leq csz(c) \quad \forall b \in F \cdot (c \neq b \wedge \mathsf{fit}(b,n) \leq csz(b)) \Rightarrow (csz(b) - \mathsf{fit}(b,n) \geq csz(c) - \mathsf{fit}(c,n))}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$ $\mathsf{hsearch}^F_{*F}\ \dfrac{\forall b \in F \cdot csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} \sigma}$

Table 20. Refinements of remove/insert operations on heap list

$\mathsf{hremove}(c)$	$\mathsf{hremove}\ \dfrac{c \in F}{\sigma \xrightarrow{\mathsf{hremove}(c)} \sigma[F \leftarrow F \setminus \{c\}, cst(c) \leftarrow 0]}$	$\mathsf{hinsert}(c)$ $\mathsf{hinsert}\ \dfrac{c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hinsert}(c)} \sigma[F \leftarrow F \cup \{c\}, cst(c) \leftarrow 1]}$
$\mathsf{hsplit}(c,n) : c'$	$\mathsf{hsplit}_B\ \dfrac{c \in F \quad 0 < n \leq csz(c) \quad c' = c+n \quad \left\langle \begin{array}{l} H \cup \{c'\}, F, \\ csz[c \leftarrow n, c' \leftarrow csz(c) - n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle \xrightarrow{\mathsf{hremove}(c);\mathsf{hinsert}(c')} \sigma_1}{\sigma \xrightarrow{\mathsf{hsplit}(c,n):c'} \sigma_1}$	$\mathsf{hsplit}_E\ \dfrac{c \in F \quad 0 < n \leq csz(c) \quad c' = c + csz(c) - n}{\sigma \xrightarrow{\mathsf{hsplit}(c,s):c'} \left\langle \begin{array}{l} H \cup \{c'\}, F, \\ csz[c \leftarrow csz(c) - n, c' \leftarrow n], \\ cnx[c \leftarrow c', c' \leftarrow cnx(c)] \end{array} \right\rangle}$
$\mathsf{hmerge}(b) : x$	$\mathsf{hmerge}^S_R\ \dfrac{b \in F \quad c = cnx(b) \quad c \in F \quad \left\langle \begin{array}{l} H \setminus \{c\}, F, \\ csz[b \leftarrow csz(b) + csz(c)], \\ cnx[b \leftarrow cnx(c)] \end{array} \right\rangle \xrightarrow{\mathsf{hremove}(c)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma_1}$ $\mathsf{hmerge}^S_L\ \dfrac{b \in F \quad cnx(c) = b \quad c \in F \quad \left\langle \begin{array}{l} H \setminus \{b\}, F \\ csz[b \leftarrow csz(b) + csz(c)], \\ cnx[b \leftarrow cnx(c)] \end{array} \right\rangle \xrightarrow{\mathsf{hremove}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{hmerge}(b):c} \sigma_1}$	$\mathsf{hmerge}^F_R\ \dfrac{b \in F \quad c = cnx(b) \quad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma}$ $\mathsf{hmerge}^F_L\ \dfrac{b \in F \quad cnx(c) = b \quad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{hmerge}(b):b} \sigma}$
$\mathsf{hmerge}_N(b)$	$\mathsf{hmerge}^S_N\ \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_R(b):b} \sigma_1 \quad \sigma_1 \xrightarrow{\mathsf{hmerge}_L(b):c} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_2}$	
$\mathsf{hmerge}_\forall$	$\mathsf{hmerge}^S_\forall\ \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{hmerge}_N(b)} \sigma_1 \quad \sigma_1 \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma_2}$	$\mathsf{hmerge}^F_\forall\ \dfrac{I_{ec}}{\sigma \xrightarrow{\mathsf{hmerge}_\forall} \sigma}$
$\mathsf{hsearch}(n) : c$	$\mathsf{hsearch}^S_{FF}\ \dfrac{c \in F \quad \mathsf{fit}(c,n) \leq csz(c) \quad \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$ $\mathsf{hsearch}^F_{*F}\ \dfrac{\forall b \in F \cdot csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{hsearch}(n):\mathsf{nil}} \sigma}$	$\mathsf{hsearch}^S_{BF}\ \dfrac{\begin{array}{c} c \in F \quad \mathsf{fit}(c,n) \leq csz(c) \\ \forall b \in F \cdot (c \neq b \wedge \mathsf{fit}(b,n) \leq csz(b)) \\ \Rightarrow (csz(b) - \mathsf{fit}(b,n) \geq csz(c) - \mathsf{fit}(c,n)) \end{array}}{\sigma \xrightarrow{\mathsf{hsearch}(n):c} \sigma}$

Table 20 specifies operations hremove and hinsert. The state of the chunk c given as parameter is updated accordingly.

The operation hsplit 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. Notice that by creating a new busy chunk, this operation preserves the validity of the invariant I_{ec}. Table 17 includes two distinct specifications for hsplit: hsplit_B and hsplit_E for the creation of the new chunk at start resp. end of the chunk c.

The operation hmerge is called in `free` or `realloc` to join two successive free chunks in one. But a state with two successive free chunks does not comply with the invariant I_{ec} satisfied in states of DMA with early coalescing. Therefore, the invariant I_{ec} is broken temporarily in a state before hmerge.

The first behaviour of hmerge is hmerge_R, defined in Table 18 by rules hmerge_R^S and hmerge_R^F. The operation joins its parameter, a chunk b, with its successor c (also right neighbour) if c is free; otherwise, the operation does nothing. For sake of symmetry with the second behaviour, hmerge_R returns its parameter. The second behaviour of hmerge is hmerge_L, defined by rules hmerge_L^S and hmerge_L^F. It joins its parameter b with its predecessor c (also left neighbour), if c is free and returns the address of this predecessor; otherwise, the operation returns b.

The above operations are called by hmerge_N and $\mathsf{hmerge}_\forall$. Operation hmerge_N merges a free chunk b with its free neighbours. Operation $\mathsf{hmerge}_\forall$ merges any two successive free chunks in the entire memory region. Rule $\mathsf{hmerge}_\forall^F$ states property I_{ec} is satisfied and it is the terminal point of $\mathsf{hmerge}_\forall$. The operation hmerge_N will be used for eager coalescing policy, while $\mathsf{hmerge}_\forall$ is used for lazy coalescing policy.

The search of a fitting chunk in a heap list is done using operation hsearch, shown In Table 19, which receives the requested size and returns a fitting free chunk or nil. We define refinements of this operation, $\mathsf{hsearch}_{FF}$ (resp. $\mathsf{hsearch}_{BF}$) for first-fit (resp. best-fit) policy.

A.2 Rules of methods in heap-list

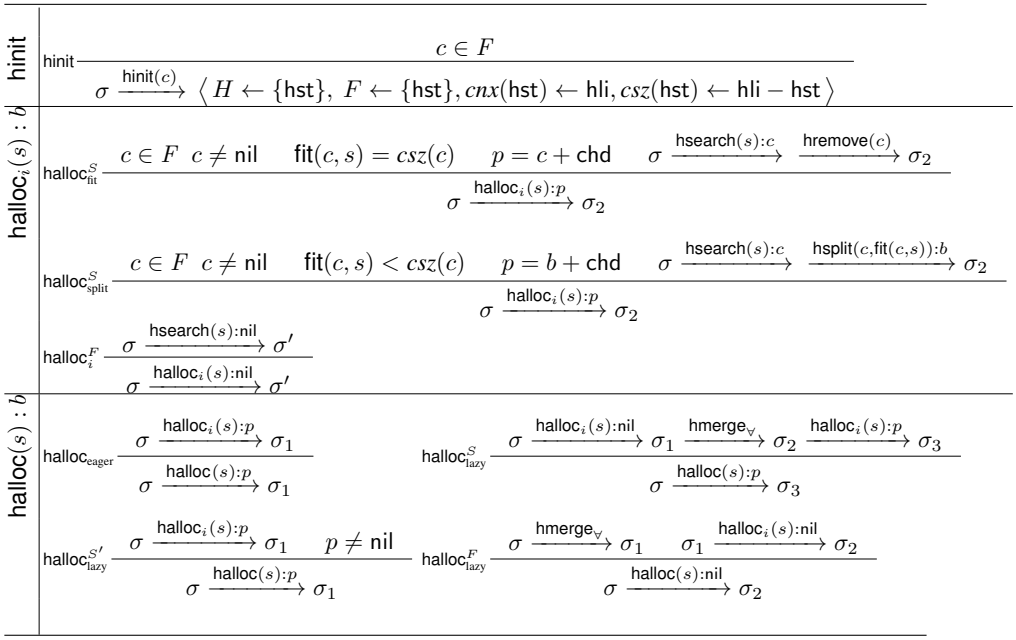
The rule for method `init` in heap-list level is hinit shown in Table 21, which initializes the memory state. It refines abstract rule `init` in Table 1. In the initial memory state, the set of heap chunks has only one chunk which represents the whole memory region and is identified as hst. The size of hst is fixed and the next chunk of hst is hli.

Operation halloc_i has as parameter a request size s and returns a suitable chunk represented by b. It defined by two rules $\mathsf{halloc}_{\text{fit}}$ and $\mathsf{halloc}_{\text{split}}$. Operation $\mathsf{halloc}_{\text{fit}}$ allocates the exact suitable free chunk while $\mathsf{halloc}_{\text{split}}$ splits a big chunk into two parts and allocates one.

The `alloc` in abstract specification is refined by two rules, $\mathsf{halloc}_{\text{eager}}$ and $\mathsf{halloc}_{\text{lazy}}$, representing allocation with eager coalescing and lazy coalescing respectively. Both of them have a request size parameter (s), a return value (b) and call the internal operation halloc_i.

As to lazy coalescing policy, there are two cases, specified by $\mathsf{halloc}_{\text{lazy}}^S$ and $\mathsf{halloc}_{\text{lazy}}^{S'}$, which depend on the returnee of hsearch. Rule $\mathsf{halloc}_{\text{lazy}}^S$ states that allocator merges

Table 21. Refinements of `init` and `alloc` on heap list



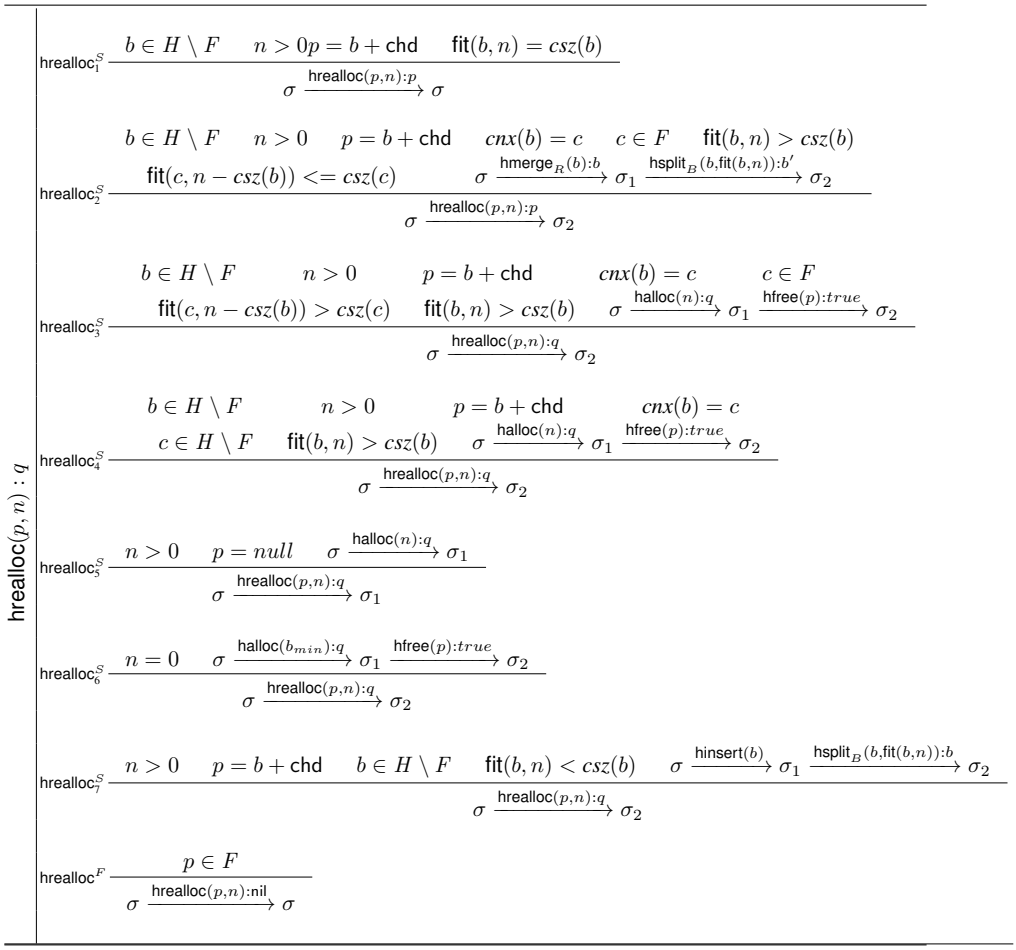
two continuous free chunks when there is no big enough free chunk after searching (hsearch returns nil). The rule $\mathsf{halloc}^{S'}_{\mathsf{lazy}}$ is applied when hsearch returns a fitting free chunk.

In Table 22, we defined rules for method `realloc`. There are seven cases represent different procedures of `realloc`.

The rule for method `free` is specified by hfree, shown in Table 23. It includes two behaviors $\mathsf{hfree}_{\mathsf{eager}}$ and $\mathsf{hfree}_{\mathsf{lazy}}$ for two coalescing policies. The behavior of $\mathsf{hfree}_{\mathsf{eager}}$ includes rules $\mathsf{hfree}^S_{\mathsf{eager}}$ and $\mathsf{hfree}^F_{\mathsf{eager}}$, which means that we free a chunk successfully if the parameter is valid and do nothing if the parameter is not valid.

To satisfy property I_{ec} for DMA with eager coalescing policy, we should pay attention to changing statuses of chunks. The operation hmerge merges two free successive chunks in our specifications. Thus, we need to change the status of the chunk to be released by operation hinsert then do hmerge_N operation. Notice that operation hinsert is called in $\mathsf{hfree}^S_{\mathsf{eager}}$ to change the status of chunk b. In this way, I_{ec} cannot be satisfied in state σ_1 of rule $\mathsf{hfree}^S_{\mathsf{eager}}$. We can only ensure that I_{ec} is satisfied after method hfree. An alternative way is to remove the free neighbor of chunk b from free list firstly, then we merge the free neighbor with b together. Property I_{ec} will be satisfied at any intermediate steps of method hfree. Therefore, we need to specify that operation hmerge merges two 'busy' chunks. However, the state 'busy' is a fake state. It doesn't mean that the 'busy' chunk is the allocated one, and the merge operation doesn't touch the contents stored inside the chunks.

Table 22. Refinements of realloc method on heap list



B Free List Modeling

B.1 Formalize acyclic unsorted free list (MUA, MUAD)

Singly-linked list To formalize the free list, we define a map (relation) to present the link-relation between free chunks. For acyclic free list, we define invariants shown as follows:

$$\begin{array}{l} \textit{inv1} : \textit{fnx} \in F \cup \{\textit{fbe}\} \rightarrowtail\!\!\!\!\!\rightarrow\ F \cup \{\textit{fen}\} \\ \textit{inv2} :\ \forall u \cdot u \subseteq \textit{fnx}^{-1}[u] \Rightarrow u = \varnothing \end{array}$$

In the above invariants, the relation *fnx* specifying points-to is defined as a bijection. We use two extra variables *fbe* and *fen* to specify the head and the end of free list. However, *fbe* and *fen* are not free chunks. Invariant *inv2* specifies that there is no loop in the free list. Thus, we can avoid the error state, e.g., $\{\textit{fbe} \mapsto \textit{fen}, n_1 \mapsto n_2, n_2 \mapsto n_3, n_3 \mapsto n_1\}$.

Table 23. Refinements of `free` method on heap list

$$\mathsf{hfree}(p):t$$

$$\mathsf{hfree}^{S}_{\mathsf{eager}}\ \frac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{hinsert}(b):b} \sigma_1\ \sigma_1 \xrightarrow{\mathsf{hmerge}_N(b):b} \sigma_2}{\sigma \xrightarrow{\mathsf{hfree}(p):\mathrm{true}} \sigma_2}$$

$$\mathsf{hfree}^{S}_{\mathsf{lazy}}\ \frac{p = b + \mathsf{chd} \quad b \in H \setminus F \quad \sigma \xrightarrow{\mathsf{hinsert}(b)} \sigma'}{\sigma \xrightarrow{\mathsf{hfree}(p):\mathrm{true}} \sigma'} \qquad \mathsf{hfree}^{F}_{*}\ \frac{\forall b \in H \setminus F \cdot p \neq b + \mathsf{chd}}{\sigma \xrightarrow{\mathsf{hfree}(p):\mathrm{false}} \sigma}$$

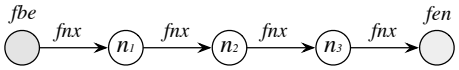


Fig. 6. Shape of acyclic free list

For example, in the above figure, the state of *fnx* is $\{\mathit{fbe} \mapsto n_1, n_1 \mapsto n_2, n_2 \mapsto n_3, n_3 \mapsto \mathit{fen}\}$. $\mathit{fnx}(\mathit{fen})$ is not defined because *fen* is not in the domain of *fnx*. We can get the first free chunk of free list by $\mathit{fnx}(\mathit{fbe})$ and last free chunk by $\mathit{fnx}^{-1}(\mathit{fen})$.

Doubly-linked list The backward points-to relation is specified by *inv3* which is the reverse of *fnx*. Thus, we can specify the doubly-linked acyclic list by combining *inv1*, *inv2* with *inv3*.

$$\begin{array}{l} \mathit{inv1} : \mathit{fnx} \in F \cup \{\mathit{fbe}\} \rightarrowtail\!\!\!\!\!\rightarrow\ F \cup \{\mathit{fen}\} \\ \mathit{inv2} :\ \forall u \cdot u \subseteq \mathit{fnx}^{-1}[u] \Rightarrow u = \varnothing \\ \mathit{inv3} : \mathit{fpr} = \mathit{fnx}^{-1} \end{array}$$

Initial state The initial state of singly-linked acyclic unsorted free list is: $< \mathit{fnx} \leftarrow \{\mathit{fbe} \mapsto \mathsf{hst}, \mathsf{hst} \mapsto \mathit{fen}\} >$. The unique free chunk is represented by hst. The initial state of doubly-linked acyclic unsorted free list is: $< \mathit{fnx} \leftarrow \{\mathit{fbe} \mapsto \mathsf{hst}, \mathsf{hst} \mapsto \mathit{fen}\}, \mathit{fpr} \leftarrow \{\mathit{fen} \mapsto \mathsf{hst}, \mathsf{hst} \mapsto \mathit{fbe}\} >$.

B.2 Formalize acyclic sorted free list (MSA,MSAB,MSAF)

As to acyclic sorted free list, based on *fnx* we defined above, we use extra invariants to specify the list is in addressed order.

$$\begin{array}{l} \mathit{inv1} : \mathit{fnx} \in F \cup \{\mathit{fbe}\} \rightarrowtail\!\!\!\!\!\rightarrow\ F \cup \{\mathit{fen}\} \\ \mathit{inv2} :\ \forall u \cdot u \subseteq \mathit{fnx}^{-1}[u] \Rightarrow u = \varnothing \\ \mathit{inv3} : \mathit{fpr} = \mathit{fnx}^{-1} \\ \mathit{inv4} : \forall c \in F \cdot \mathit{fnx}(\mathit{fbe}) \leq c \leq \mathit{fnx}^{-1}(\mathit{fen}) \\ \qquad \wedge (\mathit{fnx}(c) = \mathit{fen} \vee c < \mathit{fnx}(c)) \end{array}$$

The invariant *inv4* constraints the next free chunk which a free chunk c points to should be bigger then c.

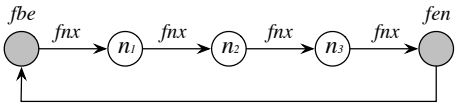


Fig. 7. Shape of cyclic free list

B.3 Formalize cyclic sorted free list (MSC)

To formalize the cyclic sorted free list, we will have several cases if the cycle is arbitrary. We specify the case shown in the figure above. The relation *inv1* in B.1 is not enough, because *fbe* should be also in the range of the relation. Thus, we define invariants for cyclic sorted free:

$$\begin{array}{l} \mathit{inv1} : \mathit{fnx} \in F \cup \{\mathit{fbe}, \mathit{fen}\} \rightarrowtail\!\!\!\!\!\rightarrow\ F \cup \{\mathit{fbe}, \mathit{fen}\} \\ \mathit{inv2} : \forall u \cdot u \subseteq \mathit{fnx}^{-1}[u] \Rightarrow u = \varnothing \\ \mathit{inv3} : \forall c \in F \cdot \mathit{fnx}(\mathit{fbe}) \leq c \leq \mathit{fnx}^{-1}(\mathit{fen}) \\ \qquad \wedge (\mathit{fnx}(c) = \mathit{fen} \vee c < \mathit{fnx}(c)) \end{array}$$

Initial state The initial state of cyclic sorted free list is: $< \mathit{fnx} \leftarrow \{\mathit{fbe} \mapsto \mathsf{hst}, \mathsf{hst} \mapsto \mathit{fen}, \mathit{fen} \mapsto \mathit{fbe}\} >$.

B.4 Rules of basic operations in free-list

In free-list level, operation fremove refines hremove of heap-list level. It has an assignment which updates *fnx*. The operation fremove has as parameter the chunk to be removed. The state of this removed chunk changed to busy (as 0) and the link relation (*fnx*) of free list are updated after fremove.

The operation finsert in free-list level, shown in Table 24, is the refinement of hinsert. It has as parameter the chunk to be inserted. It defined by several rules because insertion position depends on the shape of the free list.

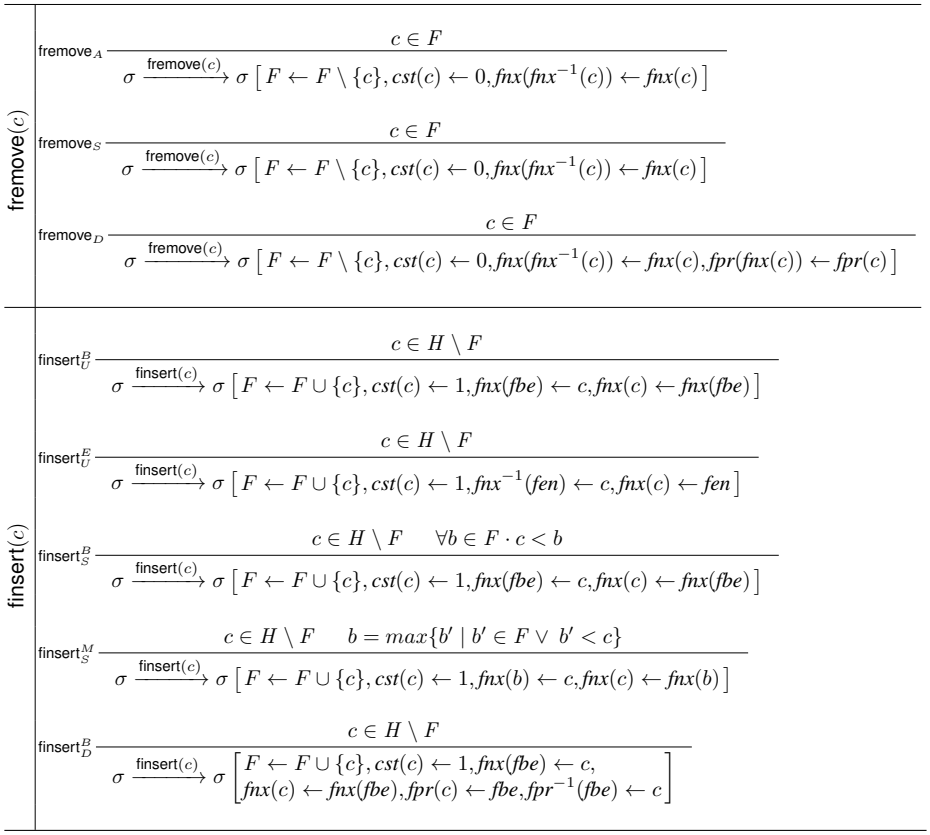
The operation fsearch refines hsearch and has a similar profile as hsearch, but it iterates only the chunks in the free list. We show the operations with three kinds of fit policies, $\mathsf{fsearch}_{BF}$, $\mathsf{fsearch}_{FF}$ and $\mathsf{fsearch}_{NF}$. $\mathsf{fsearch}_{NF}$ is for next-fit policy of which the searching position is represented by rp.

The rest operations in free-list level refine corresponding operations in heap-list by replacing the operations called with refined operations. For instance, hmerge operation calls hremove in heap-list while operation fmerge calls fremove in free-list level. However, the operation fmerge_R^S is a special case. In free-list level, the refined operation fmerge_R^S will call one more operation finsert to insert the new merged free chunk into free list.

B.5 Rules of methods in free-list

In Table 28, the rule for method `init` is specified by finit. The rule for method `free` is specified by ffree. It includes two behaviors $\mathsf{ffree}_{\text{eager}}$ and $\mathsf{ffree}_{\text{lazy}}$ for two coalescing

Table 24. Refinements of remove/insert on free list



policies. The behavior of $\mathsf{ffree}_{\text{eager}}$ defined by two rules $\mathsf{ffree}^S_{\text{eager}}$ and $\mathsf{ffree}^F_{\text{eager}}$, which means that we free a chunk successfully if the parameter is valid and do nothing if the parameter is not valid.

In Table 30, the behavior of $\mathsf{ffree}^S_{\text{lazy}}$ inserts a free chunk into free list directly and doesn't call merge operation. The rule $\mathsf{ffree}^F_{\text{lazy}}$ has the same definition as rule $\mathsf{ffree}^F_{\text{eager}}$.

The rules for `alloc` in free-list have similar profiles as halloc in heap-list, but they call basic operations of free-list.

Table 25. Refinements of split operation on free list

$\mathsf{fsplit}(c, n) : c'$	
	fsplit_B $\dfrac{c \in F \qquad 0 < n \leq \mathit{csz}(c) \qquad c' = c + n \qquad \left\langle H \cup \{c'\}, F, \mathit{csz}[c \leftarrow n, c' \leftarrow \mathit{csz}(c) - n], \mathit{cnx}[c \leftarrow c', c' \leftarrow \mathit{cnx}(c)] \right\rangle \xrightarrow{\mathsf{fremove}(c);\mathsf{finsert}(c')} \sigma_1}{\sigma \xrightarrow{\mathsf{fsplit}(c,n):c'} \sigma_1}$
	fsplit_E $\dfrac{c \in F \qquad 0 < n \leq \mathit{csz}(c) \qquad c' = c + \mathit{csz}(c) - n}{\sigma \xrightarrow{\mathsf{fsplit}(c,s):c'} \left\langle H \cup \{c'\}, F, \mathit{csz}[c \leftarrow \mathit{csz}(c) - n, c' \leftarrow n], \mathit{cnx}[c \leftarrow c', c' \leftarrow \mathit{cnx}(c)] \right\rangle}$

C Refinement to implementation

Table 26. Refinements of merge operation on free list

$\mathsf{fmerge}(b) : x$	$\mathsf{fmerge}^S_R \dfrac{b \in F \quad c = cnx(b) \quad c \in F \quad \left\langle \begin{array}{l} H \setminus \{c\}, F, cnx[b \leftarrow cnx(c)] \\ csz[b \leftarrow csz(b) + csz(c)] \end{array} \right\rangle \xrightarrow{\mathsf{fremove}(c)} \sigma_1 \xrightarrow{\mathsf{finsert}(b)} \sigma_2}{\sigma \xrightarrow{\mathsf{fmerge}(b):b} \sigma_2}$
	$\mathsf{fmerge}^F_R \dfrac{b \in F \quad c = cnx(b) \quad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{fmerge}(b):b} \sigma}$
	$\mathsf{fmerge}^S_L \dfrac{b \in F \quad cnx(c) = b \quad c \in F \quad \langle H \setminus \{b\}, F, cnx[b \leftarrow cnx(c)], csz[b \leftarrow csz(b) + csz(c)] \rangle \xrightarrow{\mathsf{fremove}(b)} \sigma_1}{\sigma \xrightarrow{\mathsf{fmerge}(b):c} \sigma_1}$
	$\mathsf{fmerge}^F_L \dfrac{b \in F \quad cnx(c) = b \quad c \in H \setminus F}{\sigma \xrightarrow{\mathsf{fmerge}(b):b} \sigma}$
$\mathsf{fmerge}_N(b)$	$\mathsf{fmerge}^S_N \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{fmerge}_R(b):b} \sigma_1 \xrightarrow{\mathsf{fmerge}_L(b):c} \sigma_2}{\sigma \xrightarrow{\mathsf{fmerge}_N(b)} \sigma_2}$
$\mathsf{hmerge}_\forall$	$\mathsf{hmerge}^S_\forall \dfrac{b \in F \quad \sigma \xrightarrow{\mathsf{fmerge}_N(b)} \sigma_1 \xrightarrow{\mathsf{fmerge}_\forall} \sigma_2}{\sigma \xrightarrow{\mathsf{fmerge}_\forall} \sigma_2} \quad \mathsf{fmerge}^F_\forall \dfrac{I_{ec}}{\sigma \xrightarrow{\mathsf{fmerge}_\forall} \sigma}$

Table 27. Refinements of search operation on free list

$\mathsf{fsearch}(n) : c$

$$\mathsf{fsearch}^{S}_{FF}\ \frac{c \in F \qquad \mathsf{fit}(c,n) \leq csz(c) \qquad \forall b \in F \cdot b < c \Rightarrow csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{fsearch}(n):c} \sigma}$$

$$\mathsf{fsearch}^{S}_{BF}\ \frac{\begin{array}{cc} c \in F & \mathsf{fit}(c,n) \leq csz(c) \\ \multicolumn{2}{c}{\forall b \in F \cdot (c \neq b \wedge\ \mathsf{fit}(b,n) \leq csz(b)) \Rightarrow (csz(b) - \mathsf{fit}(b,n) \geq csz(c) - \mathsf{fit}(c,n))} \end{array}}{\sigma \xrightarrow{\mathsf{fsearch}(n):c} \sigma}$$

$$\mathsf{fsearch}^{S}_{NF}\ \frac{\begin{array}{cc} c \in F & \mathsf{fit}(c,n) \leq csz(c) \\ \multicolumn{2}{c}{\forall k, \ell \geq 0, b \in F \cdot (c = fnx^{k}(rp) \wedge\ b = fnx^{\ell}(rp) \wedge\ \mathsf{fit}(b,n) \leq csz(b)) \Rightarrow (k < \ell)} \end{array}}{\sigma \xrightarrow{\mathsf{fsearch}(n):c} \sigma}$$

$$\mathsf{fsearch}^{F}_{*}\ \frac{\forall b \in F \cdot csz(b) < \mathsf{fit}(b,n)}{\sigma \xrightarrow{\mathsf{fsearch}(n):\mathsf{nil}} \sigma}$$

Table 28. Refinements of `init` on free list

finit

$$\mathsf{finit}_{A}\ \frac{c \in F}{\sigma \xrightarrow{\mathsf{finit}(c)} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, \\ cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, fnx(fbe) \leftarrow\ \mathsf{hst}, fnx(\mathsf{hst}) \leftarrow\ fen \end{array} \right\rangle}$$

$$\mathsf{finit}_{D}\ \frac{c \in F}{\sigma \xrightarrow{\mathsf{finit}(c)} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fpr(fen) \leftarrow\ \mathsf{hst}, fpr(\mathsf{hst}) \leftarrow\ fbe, fnx(fbe) \leftarrow\ \mathsf{hst}, fnx(\mathsf{hst}) \leftarrow\ fen \end{array} \right\rangle}$$

$$\mathsf{finit}_{C}\ \frac{c \in F}{\sigma \xrightarrow{\mathsf{finit}(c)} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fnx(\mathsf{hst}) \leftarrow\ fen, fnx(fbe) \leftarrow\ \mathsf{hst}, fnx(fen) \leftarrow\ fbe, rp \leftarrow \mathsf{hst} \end{array} \right\rangle}$$

$$\mathsf{finit}_{N}\ \frac{c \in F}{\sigma \xrightarrow{\mathsf{finit}(c)} \left\langle \begin{array}{l} H \leftarrow \{\mathsf{hst}\}, F \leftarrow \{\mathsf{hst}\}, cnx(\mathsf{hst}) \leftarrow \mathsf{hli}, csz(\mathsf{hst}) \leftarrow \mathsf{hli} - \mathsf{hst}, \\ fnx(fbe) \leftarrow\ \mathsf{hst}, fnx(\mathsf{hst}) \leftarrow\ fen, rp \leftarrow \mathsf{hst} \end{array} \right\rangle}$$

$\mathsf{ffree}(p) : t$

$$\mathsf{ffree}^{S}_{\mathrm{eager}}\ \frac{p = b + \mathsf{chd} \qquad b \in H \setminus F \qquad \sigma \xrightarrow{\mathsf{finsert}(b):b} \sigma_1 \xrightarrow{\mathsf{fmerge}_N(b):b} \sigma_2}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{true}} \sigma_2}$$

$$\mathsf{ffree}^{S}_{\mathrm{eager}}{}^{F}\ \frac{\forall b \in H \setminus F \cdot p \neq b + \mathsf{chd}}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{false}} \sigma}$$

$$\mathsf{ffree}^{S}_{\mathrm{lazy}}\ \frac{p = b + \mathsf{chd} \qquad b \in H \setminus F, \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma'}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{true}} \sigma'} \qquad \mathsf{ffree}^{S}_{\mathrm{lazy}}{}^{F}\ \frac{\forall b \in H \setminus F \cdot p \neq b + \mathsf{chd}}{\sigma \xrightarrow{\mathsf{ffree}(p):\mathrm{false}} \sigma}$$

Table 29. Internal allocation operation on free list

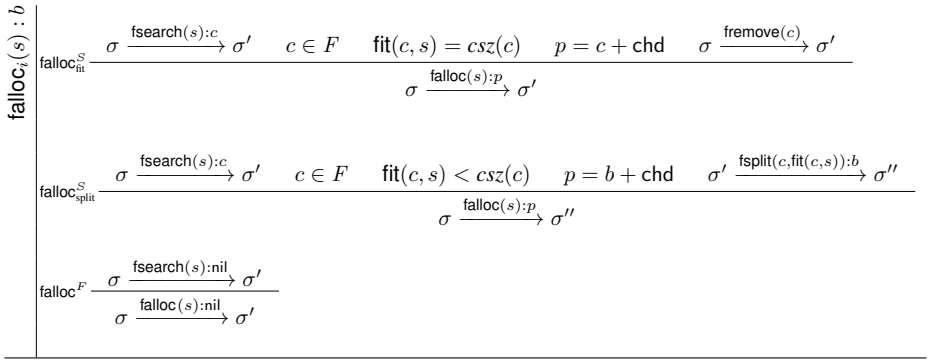


Table 30. Refinements of `alloc` method on free list

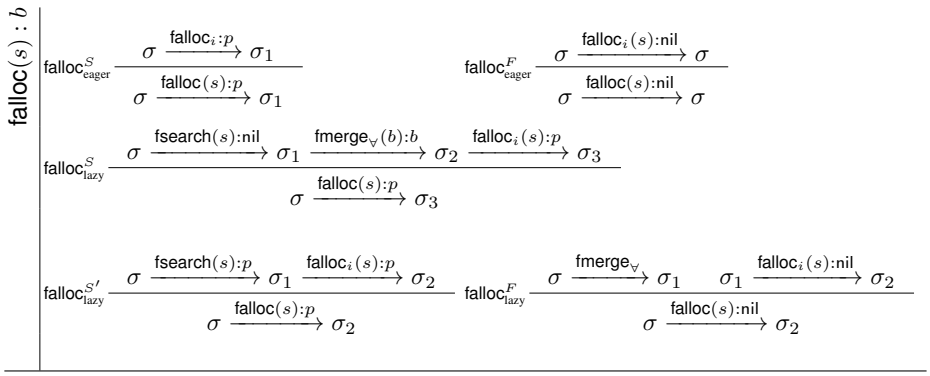


Table 31. Refinements of realloc method on free list

$\mathsf{realloc}(p,n):q$

$$\mathsf{realloc}_1^S\ \frac{b \in H \setminus F \quad n > 0 \quad p = b + \mathsf{chd} \quad \mathsf{fit}(b,n) = csz(b)}{\sigma \xrightarrow{\mathsf{realloc}(p,n):p} \sigma}$$

$$\mathsf{realloc}_2^S\ \frac{\begin{array}{c} b \in H \setminus F \quad n > 0 \quad p = b + \mathsf{chd} \quad cnx(b) = c \quad c \in F \\ \mathsf{fit}(b,n) > csz(b) \qquad \mathsf{fit}(c, n - csz(b)) <= csz(c) \\ \sigma \xrightarrow{\mathsf{fmerge}_R(b):b} \sigma_1 \xrightarrow{\mathsf{fsplit}_B(b,\mathsf{fit}(b,n)):b'} \sigma_2 \end{array}}{\sigma \xrightarrow{\mathsf{realloc}(p,n):p} \sigma_2}$$

$$\mathsf{realloc}_3^S\ \frac{\begin{array}{c} b \in H \setminus F \qquad n > 0 \qquad p = b + \mathsf{chd} \\ cnx(b) = c \quad \mathsf{fit}(c, n - csz(b)) > csz(c) \quad c \in F \\ \mathsf{fit}(b,n) > csz(b) \quad \sigma \xrightarrow{\mathsf{falloc}(n):q} \sigma_1 \xrightarrow{\mathsf{ffree}(p):true} \sigma_2 \end{array}}{\sigma \xrightarrow{\mathsf{realloc}(p,n):q} \sigma_2}$$

$$\mathsf{realloc}_4^S\ \frac{\begin{array}{c} b \in H \setminus F \qquad n > 0 \qquad p = b + \mathsf{chd} \\ cnx(b) = c \qquad c \in H \setminus F \\ \mathsf{fit}(b,n) > csz(b) \quad \sigma \xrightarrow{\mathsf{falloc}(n):q} \sigma_1 \xrightarrow{\mathsf{ffree}(p):true} \sigma_2 \end{array}}{\sigma \xrightarrow{\mathsf{realloc}(p,n):q} \sigma_2}$$

$$\mathsf{realloc}_5^S\ \frac{n > 0 \quad p = null \quad \sigma \xrightarrow{\mathsf{falloc}(n):q} \sigma_1}{\sigma \xrightarrow{\mathsf{realloc}(p,n):q} \sigma_1}$$

$$\mathsf{realloc}_6^S\ \frac{n = 0 \quad \sigma \xrightarrow{\mathsf{falloc}(b_{min}):q} \sigma_1 \quad \sigma_1 \xrightarrow{\mathsf{ffree}(p):true} \sigma_2}{\sigma \xrightarrow{\mathsf{realloc}(p,n):q} \sigma_2}$$

$$\mathsf{realloc}_7^S\ \frac{\begin{array}{c} n > 0 \quad p = b + \mathsf{chd} \quad b \in H \setminus F \quad \mathsf{fit}(b,n) < csz(b) \\ \sigma \xrightarrow{\mathsf{finsert}(b)} \sigma_1 \xrightarrow{\mathsf{fsplit}_B(b,\mathsf{fit}(b,n)):b} \sigma_2 \end{array}}{\sigma \xrightarrow{\mathsf{realloc}(p,n):q} \sigma_2}$$

$$\mathsf{realloc}^F\ \frac{p \in F}{\sigma \xrightarrow{\mathsf{realloc}(p,n):\mathsf{nil}} \sigma}$$

$\mathbb{S}$	DKBT [13] $\models$ **MUAD**	DKBF [13] $\models$ **MSAB**	DKFF [13] $\models$ **MSAF**	DKNF [13] $\models$ **MSAN**	DL [14] $\models$ **MUAD**
hst	`&base`	`&base`	`&base`	`&base`	`msegment.base`
hli	`sbrk(0)`	`sbrk(0)`	`sbrk(0)`	`sbrk(0)`	`sbrk(0)`
chd	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(malloc_chunk)`
cal	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(Header)`	**`sizeof`**`(malloc_chunk)`
$csz(x)$	`(x).size & ~1`	`x->s.size*`**`sizeof`**`(Header)`	`x->s.size*`**`sizeof`**`(Header)`	`x->s.size*`**`sizeof`**`(Header)`	`(x).size & 3`
$cst(x)$	`(x).size & 1`	—	—	—	`(x).size & 1`
$cnx(x)$	`x+(x.size)/2`	`(Header*)x + x->s.size`	`(Header*)x + x->s.size`	`(Header*)x + x->s.size`	`(x-base + x->s.size`
fnx	`x.nlink`	`x.link`	`x.link`	`x.link`	`x->next`
fbe	`AVAIL`	`AVAIL`	`AVAIL`	`AVAIL`	`msegment`
$\mathsf{fit}(c,n)$	`(n+3)/4 + 1`	`(n+3)/4 + 1`	`(n+3)/4 + 1`	`(n+3)/4 + 1`	`(n+3)/4 + 1`

$\mathbb{S}$	TOPSY [8] $\models$ **MHL**	IBM [4] $\models$ **MH**	LA [3] $\models$ **MSAF**	KR [11] $\models$ **MSC**	TLSF [18] $\models$ **MUAD**
hst	`start`	`start`	`_hsta`	`&begin`	—
hli	`end`	`end`	`sbrk(0)`	`sbrk(0)`	`sbrk(0)`
chd	**`sizeof`**`(HmEntryDesc)`	**`sizeof`**`(MCB)`	**`sizeof`**`(HDR)`	**`sizeof`**`(Header)`	`x+offset(fnxt)`
cal	4	**`sizeof`**`(MCB)`	**`sizeof`**`(HDR)`	**`sizeof`**`(Header)`	**`sizeof`**`(block_header)`
$csz(x)$	**`(long)`**`x->next-`**`(long)`**`x`	`x->size*`**`sizeof`**`(MCB)`	`x->size*`**`sizeof`**`(HDR)`	`x->s.size*`**`sizeof`**`(Header)`	`(x).size & ~3`
$cst(x)$	`x->status`	`x->status`	—	—	`(x).size & 1`
$cnx(x)$	`x->next`	`(MCB*)x+x->size`	`(HDR*)x+x->size`	`(Header*)x + x->s.size`	`x+x->size`
fnx	—	—	`x->ptr`	`x->s.ptr`	`x->fnxt`
fbe	—	—	`frhd`	`freep`	`blocks`
$\mathsf{fit}(c,n)$	$(csz$`>((n+3)&0x0F+8))?((n+3)&0x0F)):`csz	`n+4`	`(n+3)/4 + 1`	`(n+3)/4 + 1`	`(n+3)/4 + 1`