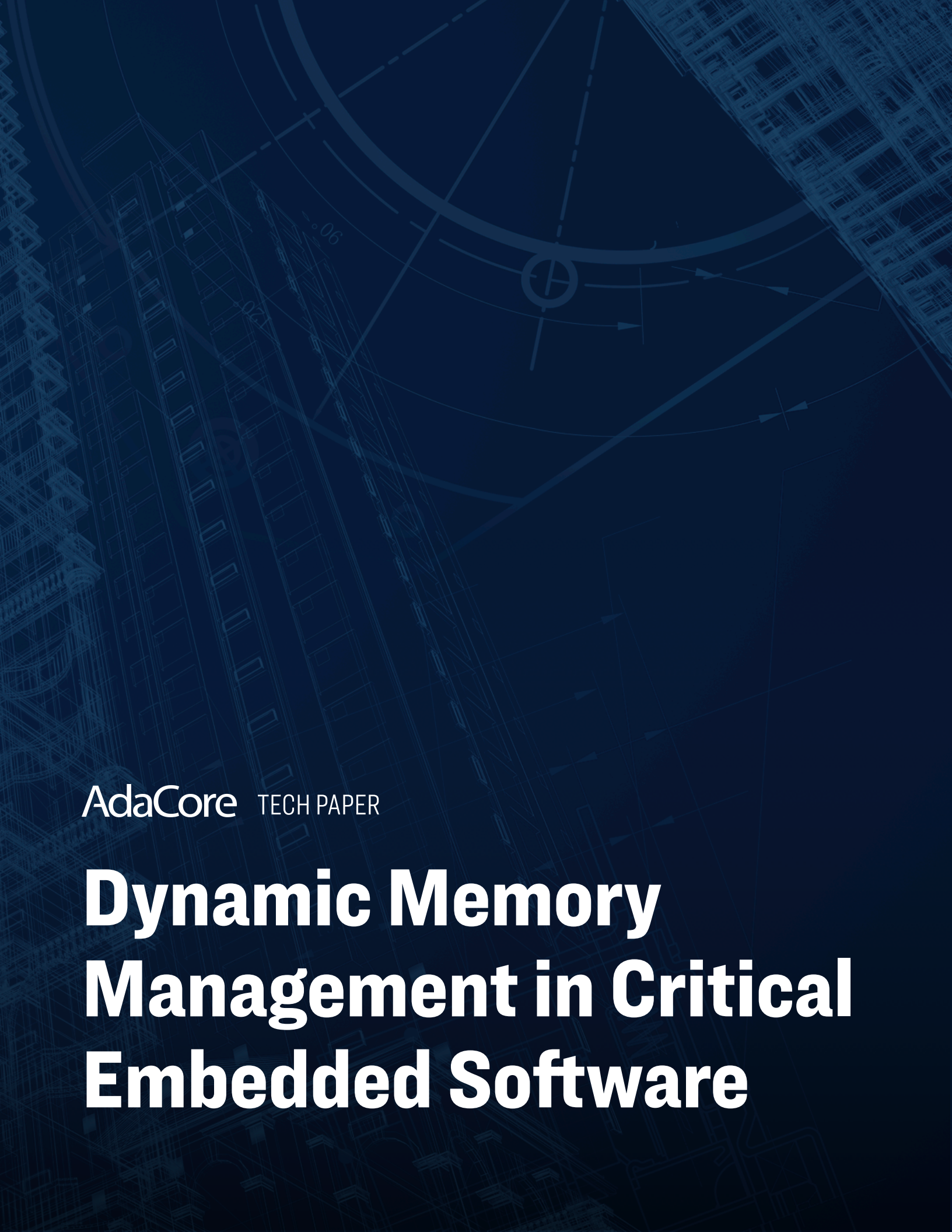

AdaCore TECH PAPER

Dynamic Memory Management in Critical Embedded Software

Dynamic Memory Management in Critical Embedded Software

Category: regular paper
Authors: Cyrille Comar (AdaCore), Claire Dross (AdaCore), Florian Gilcher (Ferrous Systems), Yannick Moy (AdaCore)
Keywords: critical embedded software, dynamic memory management, program proof

Memory management has always been a delicate issue in critical embedded software because memory is often a scarce resource and many of the typical software errors jeopardizing the integrity of the execution of the software are related to memory mismanagement. Furthermore, critical software has had a tendency to grow in size and complexity in recent years, because it is using more and more complex algorithms in the critical parts of a system. The push towards autonomous mobility is a good example of the drivers for complexity reaching the most critical parts of software controlling such systems. This added complexity requires added flexibility in memory management that is not compatible with the traditional memory management techniques used for critical embedded software. In this paper we will first go over the traditional memory management limitations and the reasons behind them, we will then explore possibilities for going beyond them while being able to provide a high level of guarantees of correctness with regard to memory usage.

Dynamic Memory Management in Critical Software

We usually distinguish different kinds of data depending on the complexity of their memory management life cycle. The simplest is statically-allocated data, which is allocated once at the start of the application and is never deallocated. The case of dynamically allocated data is far more complex. It includes both stack-allocated and heap-allocated data. Stack-allocated data is data locally allocated by a subprogram in its activation frame, so there might be multiple instances of the same data if the subprogram is recursive, and the maximum memory usage depends on the call-graph of the program. Heap-allocated data can be allocated at any point, and deallocated at any later point, with associated risks of accessing deallocated data, losing access to allocated data or even deallocating already deallocated data.

The only requirement for safe use of statically-allocated data is that it fits in memory. This is similar to the requirement that the code fits in memory, and is checked by comparing the size of the corresponding segment in the executable with the available memory size on the target platform. The corresponding requirement for stack-allocated data is that it fits in memory at all times during the execution of the application. This is checked by performing a worst-case analysis of local memory usage, taking into account the call-graph of the application, and comparing it with the size allocated

to the stack(s) on the target platform. Such a worst-case analysis can be made slightly difficult by the presence of cycles in the call graph, the existence of dynamically sized variables on the stack or the use of indirect calls but professional grade tools exist to perform this kind of analysis (e.g. GNATstack for Ada). As stack-allocated data is deallocated at subprogram exit, there is another requirement that such data is not accessed after subprogram exit, which could happen when the address of such local data is stored in pointers. This requirement can be enforced by programming languages (in Ada, Java, OCaml, Rust) or by static analysis (in C, C++). There are many more requirements for heap-allocated data, besides avoiding the errors previously mentioned. One also needs to make sure that fragmentation doesn't hinder memory allocation of large data pieces and more generally that enough memory is available at any time for the needs of the application.

Given the difficulty of guaranteeing that these requirements are satisfied, many coding standards for critical software forbid heap allocation either completely or after initialization. The most common pattern allowing heap allocation in critical software consists in an initialization phase where memory is dynamically allocated, and never deallocated. Thus, the only requirement for such a pattern is that there is enough memory for allocations to succeed during this initialization phase which is relatively easy to meet.

However, more liberal use of dynamic memory should also be possible in critical software, provided the associated requirements are adequately addressed. In the following, we refer to the subsections and risks of section OO.D.1.6.1 of the DO-332/ED-217 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A avionics standards. It lists the following risks, phrased here alternatively as criteria that a memory management should meet:

a. **Risk:** Ambiguous references. **Criterion:** The allocator returns a reference to a valid piece of memory, not otherwise in use.
b. **Risk:** Fragmentation starvation. **Criterion:** If enough space is available, allocations will not fail due to memory fragmentation.
c. **Risk:** Deallocation starvation. **Criterion:** An allocation cannot fail because of insufficient reclamation of inaccessible memory.
d. **Risk:** Heap memory exhaustion. **Criterion:** The total amount of memory needed by the application is available (that is, the application will not fail because of insufficient memory).
e. **Risk:** Premature deallocation. **Criterion:** An object is only deallocated after it is no longer used.
f. **Risk:** Lost update and stale reference. **Criterion:** If the memory management system moves objects to avoid fragmentation, inconsistent references are prevented.
g. **Risk:** Time-bound allocation or deallocation. **Criterion:** Allocations and deallocations complete in bounded time.

Those risks and associated criteria can be grouped as follows:

- Criteria related to temporal memory safety, to ensure that accessed data is allocated (a.k.a. use-after-free, or Risk (e) “Premature deallocation”) and that data is not deallocated multiple times (a.k.a. double-free)
- Criteria related to memory availability, which requires in particular that access to allocated data is not lost (see Risks (b) “Fragmentation starvation”, (c) “Deallocation starvation” and (d) “Heap memory exhaustion”)
- Criteria related specifically to the garbage collection techniques. The time at which memory is deallocated, and the running time for deallocation, are in general unpredictable, which is a problem for real-time critical software. See also Risk (g) “Time-bound allocation or deallocation”. The garbage collector may move data to avoid memory fragmentation, which requires all references to the data to be updated to its new location. See also Risk (f) “Lost update and stale reference”.

DO-332 lists three main techniques for dynamic memory management: object pooling, activation frame based object management (divided into stack allocation and scope allocation), and heap based object management (divided into manual heap allocation and automatic heap allocation). The following table summarizes how criteria are addressed with each technique, detailing whether the application code (AC) or the memory management infrastructure (MMI) is responsible for it:

Table OO.D.1.6.3 Where Activities Are Addressed For DMM

Technique	**Activities (OO.6.8.2)**						
	a	**b**	**c**	**d**	**e**	**f**	**g**
Object pooling	AC	AC	AC	AC	AC	N/A	MMI
Stack allocation	AC	MMI	MMI	AC	AC	N/A	MMI
Scope allocation	MMI	MMI	MMI	AC	AC	MMI	MMI
Manual heap allocation	AC	AC*	AC	AC	AC	N/A	MMI
Automatic heap allocation	MMI	MMI	MMI	AC	MMI	MMI	MMI

*AC = application, MMI = memory management infrastructure, N/A = not applicable, and * = difficult to ensure by either application or MMI*.

Ease of use correlates with the larger dependency on MMI for addressing criteria: automatic heap allocation (garbage collection) only leaves to AC the need to verify that there is indeed enough memory for the application to run, while manual heap allocation only leaves to MMI the need to verify that allocation and deallocation operate in bounded time. Garbage collection is thus far superior in terms of usability, at the cost of offsetting a lot of responsibilities to the MMI, which comes with major challenges for certification.

Use Case: Message Handling

In order to explain the main memory management techniques, we consider a use case of message handling. The application receives a message which it stores as an array of bytes. For efficiency, this array of bytes should not be copied, but instead a pointer to the dynamically allocated array should be passed to parts of the application that need access to the message. When handling of the message is complete, the corresponding memory should be reclaimed. Let's consider how the five memory management techniques introduced previously address this use case.

In object pooling, a different memory pool must be allocated statically for all objects of a given type. Thus, this requires creating a memory pool for all the different sizes of arrays, or at least enough intermediate values of sizes so that not too much memory is wasted as padding. Then, it's entirely up to the AC to ensure correct pool usage, making sure stale references to arrays are not used for example after the corresponding array has been reclaimed. When objects of a pool are of the same size, object pooling helps address Risk (b) "Fragmentation starvation" as it eliminates fragmentation within a pool, but the memory remains fragmented between pools.

Stack allocation and scope allocation are not applicable here, as we need to return the allocated array from the activation frame in which it was created.

In manual heap allocation, we are in a similar situation as with object pooling, only with one pool. In particular, objects are not all of the same size, so Risk (b) "Fragmentation starvation" is particularly difficult to address with this technique, as outlined in the table from DO-332. Allocators like jemalloc mitigate fragmentation by internally using multiple arenas for different allocation sizes [JEMALLOC].

In automatic heap allocation, memory is allocated as needed when creating the data structure from the incoming message, and from that point on, the AC needs not be concerned with dynamic memory management. The MMI is in charge of ensuring that all other criteria (except availability of enough memory) are ensured. Addressing fragmentation in particular requires moving allocated data to ensure larger contiguous patches of memory are available for allocation, which greatly increases the complexity of the MMI. Similarly for addressing time-bound allocation and deallocation, which makes it necessary for the garbage collector to operate in chunks instead of in one go.

Use Case: Current Practice and Challenges

In cases where it applies, garbage collection clearly provides the best user experience. But the cost of certifying the MMI prevents this technique from being used in many cases. Real time Java was the only effort to adapt garbage collection to the needs of critical real-time software, by providing the means to exempt regions of code from garbage collection and use region-based memory management instead, but it has not seen much adoption. [SCJAVA, USCJAVA]

This leaves object pooling and manual heap allocations as the only two applicable techniques, with the same assignment of most responsibilities to AC for ensuring criteria a-b-c-d-e are satisfied. This is a major challenge in certification, which explains why dynamic memory management remains difficult to use in critical software, beyond the previously mentioned pattern of use only during initialization.

The Ownership Approach to Dynamic Memory Management

Ownership is an approach whereby, at any program point, an entity within the program is statically identified as the “owner” of a piece of dynamically allocated memory. Only the owner of a piece of memory can deallocate it, and an analysis tool ensures statically that the deallocated memory cannot be accessed afterwards through the owning pointer or one of its aliases. Ownership rules are generally useful to describe data structures that only create lists or trees, and never direct acyclic graphs or cycles or arbitrary graphs.

The central principle in the ownership approach is that assigning *moves* the ownership from the source to the target of the assignment. The owner of a piece of memory has exclusive read-write access to the memory, which includes the (implicit or explicit) right to deallocate the memory. The owner of a piece of memory can also grant temporary read-write access to a single other object, or temporary read-only access to multiple objects, after which ownership is transferred back to the original owner, this is often called *borrowing* or *lending*. This is typically done when passing a pointer as a parameter to a call.

Adhering to the ownership approach has several benefits for ensuring the safe use of dynamic memory:

- It guarantees temporal memory safety (no use-after-free or double-free) - Risks (a) and (e).
- It guarantees that memory is not lost (no memory leak) - Risk (c).
- It does not rely on garbage collection for releasing memory safely - Risks (f) and (g).

Thus, ownership only leaves Risks (b) and (d) that should be addressed at the level of the AC. In addition, it provides the basis for addressing safe concurrency (no data races) and formal verification (with pointers). Two programming languages used in critical software, Rust and SPARK, have implemented the ownership approach, with a focus on safe concurrency in Rust, and on formal verification in SPARK. We’re going to consider both.

Rust Approach to Ownership

Rust is a general purpose programming language originally developed by Mozilla Research. Coming from an organisation with a large C++ codebase, Rust aims to make systems programming at scale safer by enforcing strict rules around memory usage. It does so at a *type level*.

Rust is often characterised around the borrow checker, allowing safe referencing of values. But Rust uses ownership as the governing principle for the whole programming language. Every value introduced into a program has exactly one *owner*. Ownership is gained by introducing the value (independent of the location) and lost by *dropping* the value or *passing* it. In general, dropping happens if a value reaches the end of a scope without being passed on. The regions where a value is alive define the *lifetime* of the value. Those lifetimes are later used by the borrow checker to prove that references are safe. Rust strictly bans double ownership of an allocation. Rust also disallows *giving up ownership* while there are still references existing on the owned value.

As an example, Rust provides the *Box<T>* type for allocating a single value on the heap. The Box type *logically owns* its contents and is responsible in its implementation to ensure that it behaves like an owner. The implementation of Box is private and internally unsafe. Consider a possible expression in Rust of the message handling use case:

```
use std::sync::mpsc::channel;

type Message = [u8];

fn main() {
	let (sender, rcvr) = channel::<Box<Message>>();
	let task1 = move || {
		let msg: Box<[u8]> = Box::from([1,2,3,4]);
		sender.send(msg); // ownership is moved
		// payload is inaccessible here
	};
	let task2 = move || {
		if let Ok(msg) = rcvr.recv() { // ownership is gained
			println!("{:?}", msg);
			drop(msg); // drop is inserted implicitly, inserted for clarity
		}
	};
	task1();
	task2();
}
```

The use of the “move” keyword moves ownership of the referenced “sender” and “rcvr” endpoints of the channel to task1 and task2 respectively. Ownership of the message is passed on from task1 to task2 through the channel, ending with the message being deallocated by calling “drop” in task2, an

action automatically inserted by the compiler when not done explicitly like here. This example illustrates a number of points: The correctness of this code relies on the correctness of the implementation of *Box* and *mpsc::channel. Box<u8>* expresses ownership of a range of bytes of dynamic size on the heap. Giving up ownership of a *Box* triggers immediate deallocation. *mpsc::channel* on the other hand is a primitive that passes owned types between potentially concurrent components. To the user, this is expressed through owning a *sender* and a *receiver.* The internals of both primitives are private and often internally unsafe.

The Rust language expresses certain assumptions (that Box *owns* the type it is constructed with its contents), the implementation of *Box<T>* ensures this by transparently heap-allocating the value and deallocating it when ownership of the Box is given up. It is also the burden of the library programmer to make sure that the internal pointer is always valid to dereference, never dangling while in use, always pointing to a valid heap allocation and that there is only one way to deallocate the value (by letting it run out of scope, giving it up). It is also the burden of the library programmer to ensure that destructors of logically owned values are called, if necessary. If correctly implemented in the library, the type cannot be misused.

Rust as such pushes the burden of implementing valid, memory-safe types to library implementers, giving them a number of rules to uphold to implement safe components. It then constructs a *facade* around the internal complexity, by using visibility rules - the internal are inaccessible to the user. Because *Box<[u8]>* only has a single owner, by proxy, we also know that the byte field (*[u8]) only* has a single owner (the owner of the box).

Rust's background of being developed for large codebases informs its outside-in approach: it trades ease of systems construction against implementation complexity of components. Heap-allocating types like Box and Vec have far more rules to uphold in their interface than in other programming languages such as C, for example, they are also not allowed to be moved while referenced. Rust strongly suggests reuse of known-safe components, a practice that is both found in its standard library and outside of it.

We can use this information to build a generic and safe to use API to express passing data between components:

```
fn send<T: ‘static>(t: Box<T>) {
    // ‘static declares that T  is not allowed to hold inner references
    // internal implementation
}
```

This signature does not only declare that a value of *Box<T>* is taken, but also that if passed, the caller *gives up* ownership, passing it to the sending component (which later passes it to the receiver). This holds true in both single-threaded and concurrent scenarios. The ability to fully pass ownership between components is crucial to Rust's concurrency guarantees, as it allows to avoid pessimistic

locking and confusion at deallocation time. Other usages include resource modelling approaches like RTIC [RTIC-BOOK] that use ownership to model exclusive access to hardware resources.

Rust addresses Risk (a) by requiring the same behavior of an allocation system in use (that is, the underlying call to the memory allocator should not return a pointer to memory already in use). Rust prevents Risks (c) and (e) by providing a type system that makes it easy to pair allocations with deallocations and clarifies the responsible component for the deallocation operation. Risk (f) is mitigated by the Rust language strictly disallowing references existing on data being moved. Given a memory allocator with the required behaviour, Risk (g) is prevented by Rust providing clear deallocation points and not deferring memory deallocation.

SPARK Approach to Ownership

Ada is a general purpose language designed for safety critical applications. SPARK is a subset of Ada aimed at formal verification. One of the major restrictions imposed by SPARK over Ada is the absence of aliases. The latest releases of SPARK support pointers without introducing aliasing, thanks to the use of an ownership model [CAV].

In SPARK, ownership is only concerned with pointers. Pointers in Ada are called *access types*. They are basically references, pointer arithmetics is only possible through a library and is not supported in SPARK. Their design is low-level, a pointer is null at declaration, it is possible to manually allocate data on the heap, or to take a reference to a stack-allocated object which has been explicitly marked as potentially aliased. The data allocated on the heap is not reclaimed automatically (Ada does not have a garbage collector), it has to be explicitly deallocated. In addition, to enforce some level of safety, Ada introduces a notion of *accessibility level* associated to all access types. This level is used to statically enforce that data allocated on the stack in a function can never be referenced by a pointer of a type declared outside of the function, therefore ensuring that the data will not be accessed once it has been popped from the stack. Finally, Ada makes a difference between *pool-specific access types*, which are necessarily heap-allocated, and *general access types* which may be either heap-allocated, stack-allocated, or even statically allocated.

Pointer support in SPARK was designed to stay compatible with the usual semantics of pointers in Ada. As a result, pointers are allowed to be null, and, if they have been allocated on the heap, they should be deallocated manually (no automatic reclamation is done when the scope of an object is exited). To be able to verify that stack-allocated data is never deallocated, deallocation is only allowed in SPARK for pool-specific access types.

An ownership policy is used to ensure that pointers cannot create visible aliases in the program, namely, either there exists only one pointer that can be used to access the data and modify it, or there exist several pointers that can access the data but only for reading. As discussed above, this is necessary so that formal analysis remains tractable and efficient (the analysis tool can continue to

assume that there can be no aliases in the program). Together with the Ada semantics, this allows to ensure the memory integrity of SPARK programs and prevent Risk (e) as follows:

- *A pointer designating stack-allocated or statically allocated data is never freed.* This comes from the fact that objects of a general access type can never be deallocated in SPARK.
- *A pointer which is not null always designates valid data.* This follows from the accessibility rules of Ada for pointers designating stack-allocated data. For heap-allocated data, this is a result of the ownership policy, which ensures that when something is deallocated through a pointer, it cannot be accessed through any other pointers, together with the fact that deallocation nullifies the deallocated pointer in Ada.

The properties above are ensured in SPARK by construction, that is, by the semantic checking of the SPARK language, without running the formal verification tool.

In addition, the static verification tool for SPARK programs considers both dereferencing a null pointer and exiting the scope of an object when it still owns some heap-allocated memory as run-time errors, and will therefore attempt to verify that it never occurs. This prevents Risk (c).

SPARK addresses Risks (a) and (g) with the same requirements as Rust on the underlying allocation system to return valid pointers on allocation, and perform allocation and deallocation in bounded time. As just seen, SPARK verification prevents Risks (c) and (e). Risk (f) is mitigated by the SPARK language strictly disallowing references existing on data being moved. That leaves Risks (b) and (d) that should be addressed at the level of the AC.

Thanks to ownership, it is possible in the verification tool associated with the SPARK language to completely ignore the indirection associated to the pointer, and instead to consider pointers as composite types which are either null, or hold the value that they reference (similarly to *option* or *maybe* types commonly used in functional programming languages). This allows users to verify relatively complex heap-manipulating programs, involving for example recursive data-structures such as lists and trees, in an efficient way. The ownership policy allows the specifications to remain simple, as separation of memory segments remains implicit (contrary to explicitly having to state that all owned memory blocks are distinct from one another), at the cost of only being able to verify alias-free programs (no doubly linked lists or Directed Acyclic Graphs).

Here is how we could implement in SPARK the small example presented as a case-study. We define a type Bytes to be an array of an unknown number of elements of type Byte. We then define a type Ptr for pointers to such an array of bytes. Here we want to allocate data on the heap, so we use a pool-specific access type.

```
type Byte is mod 2**8;
type Bytes is array (Natural range 1 .. <>) of Byte;
type Ptr is access Bytes;
```

The function Alloc allocates an array of bytes of a specific size given as a parameter. The rules of SPARK require that the newly created pointer is stored in an object. During formal verification, the tool will make sure that this value is never discarded before being moved away or properly deallocated, so we can know that the memory will never be leaked.

```
function Alloc (Size : Natural) return Ptr is
   (new Bytes'(1 .. Size => 0));
```

The procedure Free deallocates a non-null pointer. Using an instance of the Ada.Unchecked_Deallocation generic is the normal way to deallocate data in Ada. It also sets its pointer parameter to null. Because of the ownership rule, we know that no other object can hold a reference to a pointer when it is given to Free, so the deallocated memory cannot be accessed after the free. So, while deallocation is indeed “unchecked” in Ada, it is fully formally verified in SPARK.

```
procedure Free is new Ada.Unchecked_Deallocation (Bytes, Ptr);
```

As an example of use of pointers, Swap exchanges two pointers without copying the memory around. The ownership of the memory initially designated by X is moved to Tmp and then Y, after the ownership of the memory initially designated by Y has been moved to X. Note that the ownership rules of SPARK will require that Swap is always called on distinct pointers, even though its body would also handle the case where X and Y are the same correctly.

```
procedure Swap (X, Y : in out Ptr) is
   Tmp : constant Ptr := X;
begin
   X := Y;
   Y := Tmp;
end Swap;
```

SPARK formal verification tools can be used on this procedure to show that it correctly implements swapping of the underlying values of two non-null pointers:

```
procedure Swap (X, Y : in out Ptr) with
  Pre  => (X /= null) and (Y /= null),
  Post => (X.all = Y.all'Old) and (Y.all = X.all'Old);
```

Comparison Between the Rust and SPARK Approaches

Rust and SPARK approaches, while closely related, bring different benefits, which we will explore in this section.

Rust was the first language to popularize the ownership approach. It has evolved around this core principle, and consequently offers the most features around ownership. Ownership in Rust is

implemented as a core concept of the language, enabling it in all contexts, allowing the modelling of lifetimes everywhere. Rust takes advantage of ownership to provide automatic deallocation of dynamically allocated memory. Rust aims at providing safe concurrent access to dynamically allocated memory and similar resources, and defines several standard libraries implementing standard approaches to dynamic memory management such as collections and a pointer module. A key benefit of the Rust approach is that it is implemented in the compiler, hence is enforced on all programs.

SPARK has adapted the ownership approach to the existing Ada rules regarding pointers and dynamic memory, using the existing notions of scopes and types of pointers. Thus, SPARK does not provide automatic deallocation of dynamically allocated memory, instead it allows checking that explicit deallocation does not introduce errors and does not leak memory. A key benefit of the SPARK approach is that it allows to prove properties of programs with pointers, starting with absence of runtime errors and extending to arbitrary security or safety properties expressed as contracts.

In a nutshell, program objects subject to ownership are not the same in Rust and SPARK: this concerns all objects in Rust, and only objects containing pointers in SPARK. Deallocation of dynamically allocated memory is automatic in Rust, while its correction is verified in SPARK. And enforcement of ownership is done by the compiler in Rust, by the verifier in SPARK. But both programming languages make it possible to create and reclaim objects subject to ownership via various means: this can be through a standard library like std::boxed::Box in Rust or a user library; this can be through standard allocation/deallocation in SPARK or a user library. Finally, it is possible in both Rust and SPARK to use libraries that internally violate the ownership principles, but should expose an API to client code in Rust or SPARK that is safe to use from code subject to ownership principles. This is done in Rust by using so-called “unsafe code” and in SPARK by using plain Ada code.

While Rust focuses on safe concurrency and SPARK focuses on formal verification, these shared benefits of ownership apply to both. The Rust example of sending a message across tasks can be written in SPARK too, and there are academic formal verification systems that can prove the Swap function in Rust too.

Applicability of Ownership to Other Languages

The term *ownership* was picked by Rust and later adopted in SPARK as it is a frequent concern also expressed in other languages, such as C/C++ and Java.

For C++, a common concern is the following question: given a pointer to a memory location, is the code handling that pointer allowed to deallocate (implying ownership) and mutate it (implying the question of aliasing). Both are sources of mistakes. Structured solutions for this exist. Particularly

Rust's notion of ownership being paired with destruction is bringing the common C++ concept of RAII (Resource Acquisition Is Initialization) into the language directly. Rust also implements *moves* as an implicit core language operation coupled with passing ownership, expressed in C++ through the *std::move* function. Still, a lot of concepts exist in C++ as library concepts, such as *std::unique_ptr* for pointers that should not alias and hence should destroy the pointee when they are themselves destroyed (similar to the above mentioned Box), and *std::shared_ptr* for reference counted objects. C++ is lacking a general solution for ensuring reference validity, while having a very structured approach to using its modern std library components to signal intent. Pitfalls in those API still exist, as illustrated in [UNIQUEPTR].

For C, the situation is similar to SPARK, namely, there is little support in the language for checking memory safety of programs, but some can be added using an external analysis tool. There exist several tools targeting the C language, be they general-purpose analysis tools [FRAMAC] or specific tools targeting memory safety [CLOUSEAU], which could be used for that purpose.

Ownership concerns are also a large issue in concurrent programming, particularly the ability to cleanly move data and responsibilities from one component and thread to another. Early approaches to solving this problem can for example be found in introducing the notion of owned pools into Java in JCoBox [JCOBOX], where passing objects between actors in Java also forces them to give up all references to those objects for the passing actors. This work points to Java lacking a general solution for unwanted aliasing.

Given that ownership is an implicit concern in many languages and settings, raising it to a language level concern if possible is useful. This can be done as part of the core language like in Rust, or as part of a language subset like in SPARK. The former requires influence over the committee presiding over the language evolution, which is a large endeavour for most programming languages. The latter is easier to adopt, but it puts constraints on the features supporting ownership, as they must be compatible with the base programming language.

Application of Ownership to Critical Software

The ownership approach does not address all issues with the use of dynamic memory in critical embedded software. Memory availability remains an issue, both to ensure that there is enough available memory at all times, and that memory fragmentation does not make it unavailable for allocating larger objects. Traditional solutions to address these issues have relied on static memory pools for objects of different sizes, with a suitable analysis of the program needs for each memory pool.

We will review the use of Rust and SPARK in critical software, and how ownership plays a role, concluding with a roadmap to further the applicability of both technologies to critical software.

Use of Rust in Critical Software

While Rust is not used in safety critical applications with regulatory concerns to date, it is used at many locations with major security concerns today (often referred to as “mission critical”). Rust being built for use in Firefox (a program with a massive user base and strong security concerns as an entry point for exploits) informs the ethos and the goals of the language. This reflects in project policies such as very strong requirements for supported software targets of the highest tier [TARGETPOLICY], requiring committed developers and fully automated testing of all changes, blocking release of the whole compiler, should bugs arise. Examples of further usage include use at many cloud providers such as AWS and Microsoft Azure. There is an expressed interest in moving Rust further into the critical spaces, with gaps to be filled, particularly enabling developers to better understand the tradeoffs that Rust makes and enabling them to easier implement safe base abstractions. The Ferrocene project [FERROUS] is currently underway to ensure that.

Use of SPARK in Critical Software

Since its inception in 1987, SPARK has been adopted in numerous large industrial projects to get as close as possible to zero-defect software. Typically, only critical parts of the software were proved “correct” with respect to full functional specification. More generally, SPARK was used to prove specific properties of interest about the software, like the absence of all possible run-time errors (no division by zero, no buffer overflow, etc.) and some user-specified safety or security properties. Thanks to its benefits for increasing software quality (and thus safety and security), SPARK has been a language of choice for the most stringent levels of certification domains: level A for avionics (DO-178), space (ECSS-E-ST40C), level 4 for railway (EN 50128) and automotive (ISO-26262).

Conclusion

To this day, certification standards for developing critical software applications strongly discourage the use of dynamic memory, due to the associated risks and the difficulty of demonstrating with sufficient confidence that these risks are adequately addressed. The avionics certification standard supplement DO-132 published in 2011 was the first effort to describe in detail the risks associated with the use of dynamic memory, and acceptable means of compliance. Since then, Rust has emerged as a new language for mission critical software, and it has popularized the ownership approach to dynamic memory management. This approach is not limited to Rust, as the example of SPARK has shown that it can be adapted for adoption in other languages, even supporting formal verification of pointer programs in SPARK.

Industrial users of Rust and SPARK are currently adopting this approach in the automotive domain. It remains to see how easily the objectives of ISO-26262 can be addressed when using ownership in

both programming languages. The extension to other certification domains is a challenge for the near future, that others have started investigating for the avionics domain [VFS].

References

[SCJAVA] Thomas Henties, Siemens Ag, James Hunt, Doug Locke, Kelvin Nilsen, Aonix Na, Martin Schoeberl, and Jan Vitek. Java for Safety-Critical Applications. Electronic Notes in Theoretical Computer Science - ENTCS 2009.
https://www.researchgate.net/publication/228806774_Java_for_safety-critical_applications

[USCJAVA] Kelvin Nilsen. Unification of Safety-Critical Java. Embedded Real Time Software and Systems (ERTS2012), Feb 2012, Toulouse, France. https://hal.inria.fr/ERTS2012/hal-02263468v1

[FERROUS] https://ferrous-systems.com/ferrocene/

[SPARKPRO] https://www.adacore.com/sparkpro

[TARGETPOLICY] https://doc.rust-lang.org/rustc/target-tier-policy.html#tier-1-with-host-tools

[CAV] Claire Dross and Johannes Kanig. Recursive Data Structures in SPARK. CAV 2020

[JCOBOX] Jan Schäfer and Arnd Poetsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. 24th European Conference on Object-Oriented Programming (ECOOP 2010). https://softech.informatik.uni-kl.de/homepage/publications/SchaeferPoetzschHeffter10jcobox.pdf

[UNIQUEPTR] https://bartoszmilewski.com/2009/05/21/unique_ptr-how-unique-is-it/

[JEMALLOC]
https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/

[VFS] Max Taylor, Josh Ehlinger, Jeff Imig, Massimiliano De Otto: Rust for Safe and Secure Avionics and Mission System Software, Vertical Flight Society Forum, May 2021

[RTIC] https://rtic.rs/1.0/book/en/

[CLOUSEAU] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.

[FRAMAC] Florent Kirchner, Nikolai Kosmatov, Virgiles Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. Formal Aspects of Computing.