Specification and verification of the Nova microhypervisor

Short note on the selected approach and how it is applied during the specification of some selected modules of the microhypervisor source code (Robin deliverable D.6)

Hendrik Tews Bart Jacobs Erik Poll
Marko van Eekelen Peter van Rossum

Radboud Universiteit Nijmegen, The Netherlands*
http://www.sos.cs.ru.nl/

April 25, 2007

1 Executive Summary

The objective of the Robin project is to develop an open robust computing infrastructure. The Nova micro hypervisor is currently being developed as a basis for this robust infrastructure. Within the Robin project we plan to verify some selected module of Nova. Additionally we are going to develop a formal specification of Nova's interface.

This report **only** describes the results of the work done in this workpackage in the first 6 person month. In this this time we develop the approaches for the verification and the interface specification. **The other work performed so far in this workpackage will be fully described in our deliverables at the end of the project** (but see Section 5 and the end of this summary for an overview).

Approach for verification The Nova hypervisor is developed in C++. It is a piece of very low level systems software. For its proper functioning it is unavoidable that Nova does some manipulations that are usually considered as impossible to treat in a formal verification. Such manipulations are for instance: casting data into a different type (and thereby breaking the type system), using inline assembly to access certain CPU features and manipulating the virtual memory mapping during run time.

*This work has been supported by the European Union through PASR grant 104600.

To find a verification approach that meets our requirements we reviewed the LOOP-tool [1], the denotational semantics for C++ developed in the VFiasco project [6, 14], the approach of the Key-tool [2] and various other papers on C and C++ verification [11]. We evaluated all these approaches especially with respect to their ability to treat low-level systems code and finally designed our main verification approach.

Our main verification approach will exploit source-code verification on top of a carefully designed hardware model (see Figure 1 on page 6). We thereby use and extend the denotational semantics of (a subset) of C++ developed in the VFiasco project. This denotational semantics is specifically designed to handle type casts and changes in the virtual address mapping in a proper way. Special hardware features and operations can be easily incorporated.

A hardware model forms the basis of our approach to source-code verification. This hardware model provides the machine state for the denotational semantics. It models all the peculiar pieces of the hardware in a precise way to allow an accurate verification of Nova.

Our verification approach is similar to the one used in the LOOP project on Java verification [1]. Other approaches to software verification (e.g., [2, 4, 11]) do not fit our needs because they either assume type safety or are not suitable for low-level systems software.

Generation of verified source code The above described source-code verification is our main verification approach. As an alternative we explore the generation of verified source code on the example of a buddy allocator, which forms the heart of Nova's internal memory management. For that we develop an abstract model of the buddy allocator in Pvs and prove its correctness. The abstract model will then undergo a series of refinement steps, which all preserve the correctness properties. From the last refinement we plan to generate C++ sources for inclusion in Nova, which are then correct by construction.

Approach for specification The main challenge for the specification is that Nova is currently a moving target: We are going to develop the interface specification of Nova while Nova itself and its interface is being developed. It is therefore essential to tightly integrate the development of the specification with the Nova developers. We therefore plan to base all specification development on the *Nova interface base specification*, abbreviated as *Nibs*. Nibs will be integrated in the Nova documentation, it will be a formal specification in simple set theory, yet it will not be machine readable. The simple format of Nibs ensures that all involved parties, including the kernel developers, will be able to read the specification and notify us if it gets outdated.

Other work performed in the first year Apart from working out the approaches for specification and verification we achieved the following results during the first half of the Robin project:

C++ parser We developed Olmar [16] that connects the Elsa C++ parser [10] with

Ocaml. This makes it possible to manipulate abstract syntax trees of C++ programs from within Ocaml, for instance for a C++ semantics compiler.

Hardware Specification We developed parts of the hardware specification in PVS and designed the *plain-memory interface* [15].

Buddy allocator specification We finished the abstract specification for the buddy allocator.

Structure of this report The following section describes our approach to the Nova interface specification in more detail. Section 3 describes source-code verification as our main verification approach and Section 4 is on our exploration of source-code generation. Finally, Section 5 details the main work performed before the first Robin milestone.

2 Maintainable Kernel Interface Specification

There are two important non-technical challenges for the development of an interface specification of the Nova micro hypervisor. Firstly, Nova and its interface are currently developed in parallel in the Robin project. Any specification for it might therefore quickly become outdated. Secondly, there is a geographic separation between the Nova development (Dresden, Germany) and the specification development (Nijmegen, The Netherlands).

During the internship of Marcus Völp in Nijmegen we discussed the view of the kernel developers on an abstract description of Nova's interface. From an abstract point of view Nova maintains a number of relations (some of them functional) as internal state. System calls (that is, operations of the Nova interface) have typically relative complex preconditions.[1] The preconditions require that the arguments of the system call and some derived entities are members of some designated relations of the internal state. The effect of a system call can be easily described as an update of the relations that form the internal state. Relations and relation updates (i.e., adding/removing tuples from relations) can be sufficiently well described in simple set theory. However, writing an update of a relation is relatively verbose in simple set theory.

We also discussed the above mentioned non-technical problems with Marcus Völp and devised the following plan for the development of Nova's interface specification:

- The machine readable specification that we plan to develop within Robin will be based on the *Nova interface base specification*, called *Nibs* in the following.

- Nibs will be integrated in the Nova documentation. It will be a formal specification but it will not be written in a formal specification language. It will therefore not be machine readable.

[1] The kernel checks the preconditions of a system call and aborts the system call with an error code if the preconditions are not met.

- Nibs will be expressed in simple set theory, enhanced with some bits of syntactic sugar to make it more concise and easier to read
- Nibs describes the relations that form the internal state of Nova and the effect of the Nova system calls on them.

We rely on simple set theory, because this format can be easily read and understood without extra training. With the use of carefully designed syntactic sugar the specification will be as easy to read as simple updates on the kernel state, yet it will be formal. Augmenting the documentation with a formal specification will ensure it is preciseness and rigorousness. Because of the simple specification format the kernel developers can either adjust the specification themselves or flag parts of the specification as wrong when Nova evolves.

We have already successfully tested the feasibility of writing Nibs in the above described format. The Nova documentation draft [17] contains already formal specifications of the system calls `create`, `create_space_structures`, `map`, `copy`, `unmap`.

3 Source Code Verification

Currently, there seems to be no convincing alternative to C/C++ for kernel programming. Consequently the Nova hypervisor is written in C++ using assembly code for operations that are not directly available in C++. A big proportion of the assembly code is inlined, that is, directly embedded in a block of C++ statements. Inline assembly is used at least for the following purposes:[2]

- access to hardware registers, such as those from the APIC (Advanced Programmable Interrupt Controller) but also special CPU registers, such as CR3 (page directory base register), EFLAGS (the flags register), the global descriptor table, the interrupt descriptor table, the task segment register and the feature control registers CR0 and CR4.
- embedding special instructions in the code, such as IRET (return from interrupt), INVLPG (invalidate a TLB entry)
- manipulating the stack frame to access and modify parameters of system calls or for programming non-local exists similar to longjmp [9].
- to replace a bigger block of C++ statements with one equivalent assembly instruction, for instance using BSR (bit scan right) for determining the index of the most significant bit set in a pointer.

Except for the last point assembly code cannot be avoided. For assembly statements that can be replaced with equivalent C++ code it is still to be decided, whether the verified version of Nova will contain the assembly code or the much less efficient C++ version.

[2] A definite list can only be given when the Nova development is finished.

The embedded assembly in the Nova sources pose, of course, extraordinary challenges for the verification. However, it is crucial to understand that also seemingly well-behaved C++ code can complicate the verification in at least the same way. The reason for this effect is that CPU data structures such as page directories or stack frames are modelled with C++ types and are manipulated with innocent looking C++ statements. Assigning to such a data structure can for instance change the return address of a subroutine (in the case of stack frames)[3] or change the virtual address mapping (for page directories) and thereby changing the contents of variables in a certain address range.

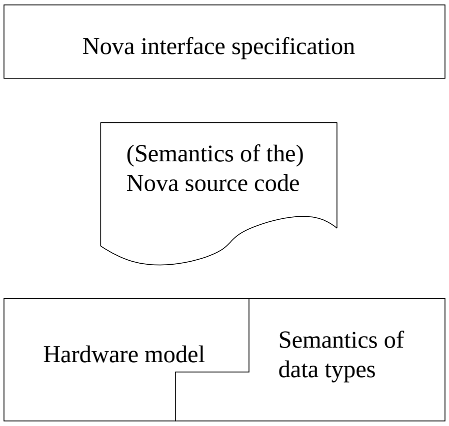
For some CPU data structures one has to obey additional rules. Up to now we now about two of such rules that affect the Nova verification. Firstly, some CPU data structures contain reserved bits. Some of those bits are checked to contain 0 in some CPU brands. It seems therefore best to verify that reserved bits are always 0. The second rule concerns the TLB (Translation Look-aside Buffer). The TLB is a special kind of cache that caches page-table traversals. As a cache the TLB is not transparent, this means, when changing a page-directory or page-table entry one must manually invalidate the TLB before using the new address mapping. Otherwise, depending on the execution history, the old mapping, still cached in the TLB, might be used (resulting in strange errors).

A final complication for the verification of Nova comes from the use of type casts. Nova must contain its own memory management and for that it is necessary to cast addresses into pointers and also to cast between different pointer types. The behaviour of such casts is left implementation specific in the C++ standard. The correctness of Nova relies therefore also on properties of the C++ compiler.

For all the just discussed subtleties and complications it is the case that that an incorrect treatment of these points in Nova might lead to undefined behaviour (read crashes) during runtime. The verification approach for Nova must treat all the above complications in a suitable manner. In detail this means:

- The hardware registers that Nova uses must be modelled with their special behaviour. For instance enabling the PGE bit (page table enable, CR4, bit 7) must change the behaviour of implicit TLB flushes (global pages will survive).
- Special assembler instructions, such as IRET, must be modelled with all their subtleties, for instance switching stacks depending on the privilege level.
- Non-local jumps and exits must be treated according to the specification of the functions that implement these things.
- Checks for reserved bits in CPU data structures.
- Check for potential TLB errors, that is, for all TLB implementations (current and future ones) and all possible TLB sizes it must be checked that TLB entries are invalided correctly.

[3] A change of the return address might be necessary for certain CPU traps when the kernel decides not to follow the hardware defined behaviour or retrying or skipping the instruction that caused the trap.



The hardware model and the semantics of data types provide the basic operations and properties for the verification of Nova. For hardware data types the hardware model relies on the semantics of data types. Technically

$$\Phi_{data\_types}, \ \ \Phi_{hardware} \ \ \vdash \ \ \varphi(\text{Nova})$$

where φ is one property from the specification, such as *termination without runtime type errors*.

Figure 1: Robin verification approach

- Check for casting errors.
- Check for errors resulting from wrong virtual address mappings.
- It must be possible to incorporate compiler specific behaviour in a modular and orthogonal way.
- Finally, of course, model the program semantics such that termination, possible abnormal conditions, and computation results can be verified.

To find a verification approach that fulfils these requirements we reviewed the LOOP-tool [1], the denotational semantics for C++ developed in the VFiasco project [6, 14], the approach of the Key-tool [2] and various other papers on C and C++ verification [11].

Apart from the solutions developed in the VFiasco project none of the proposed approaches fits with our requirements. The dynamic logic used in the KeY tool rests on the assumption that memory allocation is correct and that therefore different variables are backed up with disjunct memory. However, for Nova correct memory allocation is one of the most interesting proof goals! The LOOP tool and the work by Nipkow and Metha [11] model use a typed heap for the program state. These approaches can therefore not model implicit casts that occur when pointers of different type point to the same memory location and are used for reading and writing.

After this evaluation process we decided to further develop the superior approach from the VFiasco project [7]. The approach is depicted in Figure 1. We will use source code verification. The denotational semantics of C++ developed in the VFiasco project will be used to translate the Nova source code into its semantics in Pvs. The hardware model and the semantics of data types provide the basic building blocks of the semantics. In this design the three components (C++ semantics, hardware model and data types) are relatively independent from each other. It is therefore possible

- to add additional axioms to the data types, for instance, to model a compiler specific assumption about the size of some data types or the precise behaviour of some type casts.

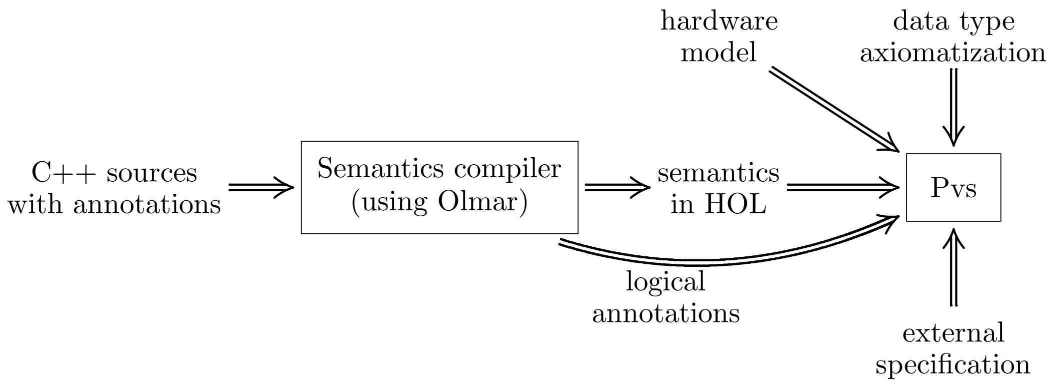


Figure 2: Approach for source code verification

- to add new operations to the hardware model
- to use different versions of the hardware model for different pieces of Nova. The boot code of Nova can be verified against physical memory and the hardware independent parts of Nova can be verified against a traditional, untyped memory model.
- to adopt the semantics for new C++ features or compiler specific C++ constructs.

Figure 2 depicts the data flow of our verification approach. A semantics compiler translates the C++ source code into its semantics in higher-order logic and outputs this semantics as Pvs source code. There will be two kinds of annotations in the source code. The first kind influences the syntactic form of the output. It will for instance be possible to place the semantics of a block or a group of statements into a separate HOL function, in order to make it possible to modularise the verification. The second kind of annotations contains specifications for the code similar to JML [3]. The semantics compiler translates these specifications into Pvs proof obligations.

The hardware model and the data type axiomatisation are directly developed in Pvs. From the Pvs point of view they provide declarations for all the function that appear in the output of the semantics compiler. With all the different pieces loaded in Pvs one can start to prove hand-written, external specifications of Nova.

In our approach a semantics compiler translates the sources of the program into semantic functions that precisely model the behaviour of the original source code. A semantic function is a state transformer of the following form[4]

$$\mathit{State} \quad\longrightarrow\quad \overset{\mathit{ok}:}{\mathit{State}} \uplus \overset{\mathit{pagefault}:}{\mathit{State} \times \mathit{Page\_fault\_info}} \uplus \overset{\mathit{hang}:}{\mathbf{1}} \uplus \overset{\mathit{fatal}:}{\mathbf{1}} \uplus \cdots$$

[4]The symbol $\uplus$ depicts disjoint union, defined as $A \uplus B \overset{\text{def}}{=} \{0\} \times A \cup \{1\} \times B$. The symbol $\mathbf{1}$ stands for the one element set. It is the mathematical counterpart of the unit and void types found in many programming languages.

Here *State* is a set of machine states provided by the hardware model. Every state contains the contents of the physical memory and the contents of some important control registers (such as virtual address mapping or the stack pointer). The disjoint union on the right hand side describes the possible results of a state transformer. If no abnormal condition occurs it yields a successor state tagged with *ok*. If a page fault occurs it yields a successor state plus some additional information. A result tagged with *hang* means that the program did not terminate (for instance of a nonterminating while loop or a page fault that keeps occurring at the same instruction). The result *fatal* is reserved for serious errors such as TLB inconsistency or reserved bit violations.

State transformers can be composed in the obvious way: If the first state transformer yields a result tagged with *OK* the result state is passed into the second state transformer. If any abnormal conditions occurs the second state transformer is skipped and the result of the composition is the abnormal result of the first state transformer.

The hardware model provides the basic state transformers for reading and writing to and from memory and for reading and writing the control registers that are supported by the hardware model. The semantics compiler composes the basic state transformers from the hardware model to build the semantics of its input program.

Program verification proceeds by reasoning in Pvs over the nontrivial state transformer that represents the semantics of the sources. This is mostly done by applying some start state to the state transformer and proving properties of the result (for instance that the result *is not* tagged *fatal*. Such a verification could equivalently performed by computing the weakest precondition of the verification goal with respect to the program.

A slightly different view on verification is as follows: The hardware model defines a state machine. The basic state transformers of the hardware model describe the actions of the state machine. The program is symbolically executed on top of the state machine. Properties are derived from the state changes that one observes. The LOOP tool and the KeY tool actually use the same approach of symbolic execution.

3.1 Goals for the semantics and the verification correctness

The preceding explanations made very clear that the precision of the verification hinges on the hardware model. With precision of the verification we refer to the amount and kind of errors whose absence is proved with a successful verification.

Our aim is to make the hardware model precise enough to let it catch the following kinds of errors:

- *Common errors*, such as dereferencing a null-pointer, nonterminating loops, wrong results (wrt. the functional specification)
- Type errors. A type error occurs when one attempts to read an instance of some type from a memory location where nothing or an instance of a different type has been stored. The hardware model is precise enough to catch type errors for user code (for instance resulting from wrong pointer or address calculations) *and* for implicit hardware memory accesses (for instance reading page directories)

- Virtual memory aliasing errors. A virtual memory aliasing occurs when the virtual memory of two distinct variables is mapped to the same (or overlapping) physical memory regions. A virtual memory aliasing error happens if one has virtual memory aliasing for two variables that are used at the same time.
- TLB errors (accessing a linear address[5] for which the page directory or page table entry might be inconsistent with the translation look aside buffer)
- allocation errors (using the same memory for different variables at the same time)
- mutating the kernel object code
- executing non-kernel object code with kernel privileges
- unwanted side effects in the kernel page fault handler
- nontermination through a faulty page-fault handler

Within Robin we are not targeting the following errors:

- any kind of hardware error
- errors that can only occur on systems with more that one logical processors (i.e., on systems with multiple CPU's or with active hyper-threading)

4 Generating Verified Sources

The verification approach described in Section 3 is our main verification approach. As an alternative we explore the possibilities of generating verified source code. The Generation of verified source code is very similar to the model driven architecture approach [12]. Starting from an abstract model or abstract specification a series of refinement steps is developed towards an actual implementation. All desired properties are proved for the abstract model or specification. The refinement steps maintain these properties. Finally source code will be automatically generated from the last and most implementation like model in the refinement. The generated code is then correct by construction.

A possible advantage of the source-code–generation approach is that it allows to obtain a fully verified code with much less effort. A disadvantage is that the code is hard to maintain: a change in the implementation requires changes in the abstract model and all refinements, and all refinement proofs must be done again. Also, the generated source code may not be as efficient as the system developers expect it to be.

Our plan is to explore the potential of the source code generation for the Nova microhypervisor. The abstract model and all proofs and refinements will be done in PVS.

[5]On the IA32 architecture virtual addresses, which are page-wise mapped to physical addresses, are called linear addresses. Virtual addresses in the sense of IA32 (i.e., the addresses that appear in the object code) are first subject to an address translation defined by the segment registers. This translation yields a linear address which is further translated using the page directory. In practice segments are not actively used so that virtual address and linear address are identical.

The C++ source will be generated from the PVS code of the most refined model. As a target module we use the buddy allocator inside Nova. The buddy allocator has a clear interface and a well-defined behaviour. On an abstract level it can be modelled in an elegant way using binary trees.

The abstract model will use a final model of a coalgebraic specification of a particular kind of finite binary trees. On top of this model an algebraic extension provides more primitive operations for manipulating the binary trees. The operations of the buddy allocator (i.e., allocate and free) are then defined on top of the algebraic extension. In this design all three levels (the final tree model, the algebraic extension and the allocator interface functions) are clearly separated and relatively independent from each other. This allows a fine-grained control over refinement steps. For instance, one such step could introduce a new final tree model, with leaving other levels intact.

Our final aim is of course to replace the buddy allocator source code in the Nova sources with automatically generated code.

5 Other Work performed in the first year

In this section we comment on the work that we performed in the first year apart from working out the specification and verification approach. The appendix (starting on page 14) contains pieces of the relevant source code and other material as illustration.

Olmar For the source-code verification we need a semantics compiler that translates Novas C++ sources into their semantics in Pvs. Apart from the C++ parser the semantics compiler performs mainly symbolic computations that are best done in a function programming language using pattern matching. We therefore developed an ocaml backend to the Elsa C++ parser suite [10], which is now part of the Oink C++ static analysis tools [18]. This Ocaml backend is freely available as a fork of the Elsa sources under the name Olmar [16]. Olmar is fully functional already. As proof of concept we implemented a tool for visualising C++ abstract syntax trees.

There was a big need for infrastructure that permits to manipulate C++ abstract syntax trees from within a clean functional programming language with pattern matching. That Olmar satisfied that need is nicely illustrated by the following: Soon after the first announcement of Olmar we got in contact with the Mozilla corporation. They plan to exploit Olmar for complex code restructuring of the Mozilla code base.

After initial ignorance the Elsa/Oink maintainers signalled interest to incorporate the Olmar features into the main Elsa distribution. We are currently in discussions about the best way to achieve this.

A detail of the abstract syntax tree of the Nova sources of the buddy allocator is in Appendix A (on page 15).

Development of the hardware model We improved and further developed the hardware model that was developed within the VFiasco project [5]. As explained above in our approach the exactness of the verification depends in a crucial way on the hardware model. We therefore improved the treatment of hardware data types in the model. As hardware data types we regard here types that are documented in the IA32 documentation [8] and that are used to control certain functions of the CPU. As hardware data types we consider for instance the type of page directory and page table entries, but also the various descriptors for call and interrupt gates.

The hardware model in [5] contained an explicit decoding of hardware data out of sequences of bytes. When interpreting a string as a page directory entry this model would probably show strange behaviour would otherwise continue to function. As improvement we applied now the ideas from [6] also to hardware types. Now the attempt to interbred a string as a page directory entry will immediately result in the abnormal *fatal* state (compare page 7), because one cannot prove that the implicit conversion of a string into a page directory entry will terminate successfully.

We further added the plain memory abstraction to the hardware model. This abstraction captures the common properties of memory, namely that reading in memory does not change anything and that writing to a variable location changes that variable and nothing else. Note that these properties do not hold in a kernel environment. Reading might trigger the page fault handler, which in turn might change the complete memory. When writing to one address an arbitrary number of addresses might change because of virtual memory aliasing and of memory mapped devices. However, for most of the Nova code these fancy effects are irrelevant and therefore the code will be verified with respect to the much more well-behaved plain memory abstraction.

The source code of the plain-memory definition, some utility lemmas and the important invariant results is included in Appendix B (on page 16).

For the construction of the hardware data types we started to developed our own Pvs library for operations on bit vectors. In our experience the Pvs provided bit vector library does not work well in applications that frequently cut out parts of bit vectors and merge them again to form other bit vectors. The Pvs bit vector library uses a dependent type for bit vectors, where the length of the bit vector appears in the type. For applications like ours that frequently use cut and merge one gets non-trivial arithmetic expressions in the types. These arithmetic expressions in types makes the development in Pvs much more complicated. One gets more and more complicated TCC's. Further, proof obligations are typically harder and require more interaction in order to simplify the arithmetic on the type level.

Our own library for bit vectors uses natural numbers as bit vectors (thus we have variable size bit vectors) and implements shift and cut operations via division and multiplication with suitable powers of 2.

Technically the plain memory abstraction is an invariant for the memory access functions that is parametric in an address set. This address set describes the part of the memory on which the plain memory properties hold, once the invariant is established. Certain parts of the page directory will always be outside the address set. For the verification of code that modifies these parts of the page directory entry one has to leaf the plain memory invariant, perform the change of the page directory, and establish the plain memory invariant again (possibly with a different address set).

Abstract buddy allocator model We worked on the code generation approach for the buddy allocator described in Section 4 and achieved important intermediate results. We studied the C++ source code of Nova's buddy allocator and derived from it an abstract model in Pvs. The abstract model clearly separates separate issues in the buddy allocator such as maintaining the sets of allocated and free memory blocks and the computation from memory blocks to addresses and vice versa. Furthermore the model is structured in three layers:

- A final model of a coalgebraic specification of finite, boolean labelled binary trees forms the bottom layer. The specification is written in CCSL [13] from which the CCSL compiler generates Pvs sources. The binary trees mimic the tree structure of blocks in a buddy allocator. The booleans in the leafs indicate whether a block is free or allocated.

  The source code of the CCSL specification is in Appendix C (on page 28).

- The second layer contains additional utility functions for binary trees such as expanding free leafs into subtrees, folding subtrees and flipping a leaf from the free state into allocated and vice versa. These utility functions are defined exploiting the finality of layer 1.

- Layer 3 defines the internal state of the buddy allocator and the operations of its external interface: allocate and These functions are defined in an ad hoc manner.

So far we proved the following properties about the abstract buddy allocator model:

- The obvious model of binary trees forms a final model of the specification of layer 1. This result establishes the correctness of the specification and justifies the assumption of an arbitrary final model in the other layers.

- Correctness of the allocation: If allocation is successful then the return address is properly aligned and the allocated block was free before.

- Correctness and liveness of free: For an allocated block the free operation will be successful and after this operation the block will be free.

- Liveness of allocate: the allocate operation will be successful if there is a large enough, sufficiently aligned block of memory.

The Pvs source code for layer 3 of the abstract buddy allocator model together with the last three of the just mentioned theorems is in Appendix D (on page 30).

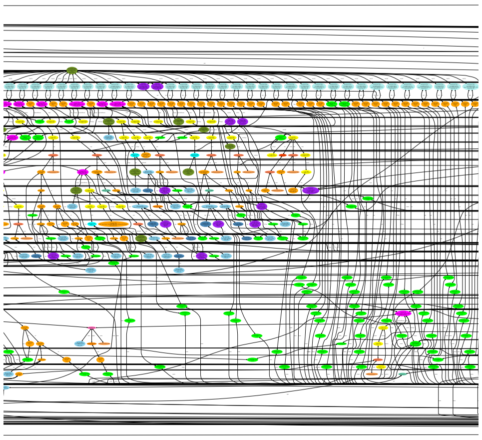
6 Conclusion

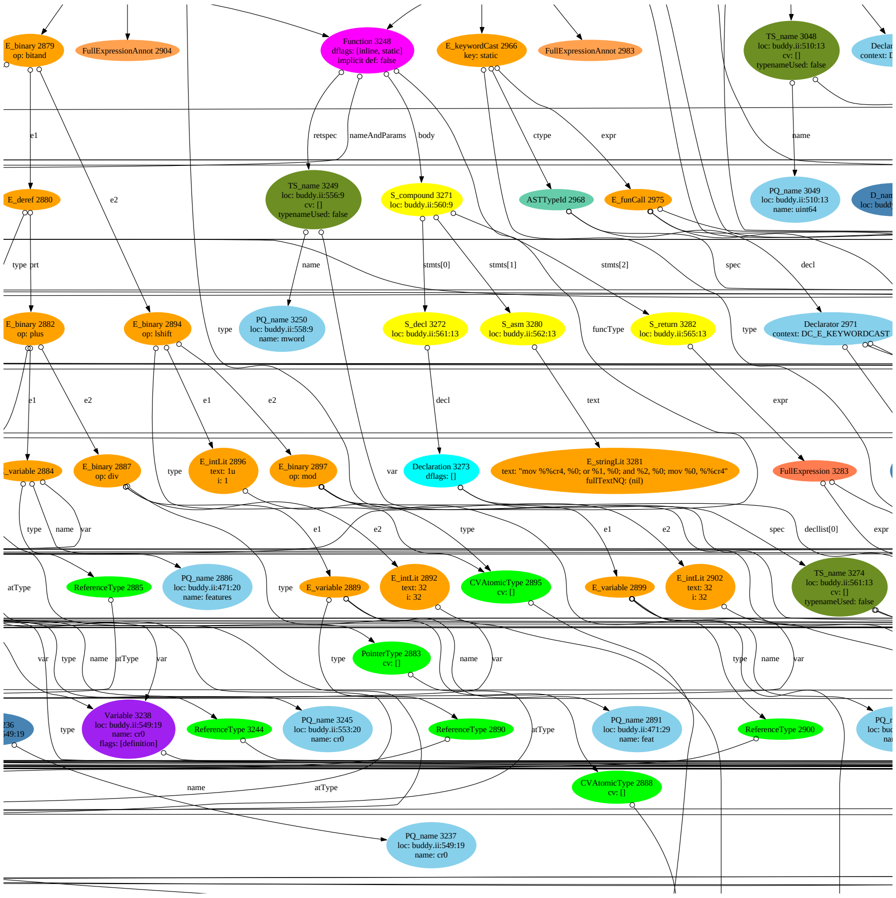
This paper outlined the approach taken for the specification and the verification of the Nova microhypervisor in the Robin project, which aims at an open robust computing infrastructure. For Nova's interface we will develop a specification in simple set theory that is closely interlocked with the interface documentation of Nova in order to minimise the danger of the specification getting outdated. To prove properties of Nova we will use source-code verification, which is based on a (partial) denotational semantics of C++. Additionally we will explore the generation of verified code for Nova, and in particular the generation of the buddy-allocator module.

Appendix

A Details of the Abstract Syntax Tree of the Buddy Allocator

Below there are two details from the abstract syntax tree from the source code of Nova's buddy allocator. The images were produced with the ast_graph tool in Olmar and rendered with dot. In the resolution of the lower picture the complete syntax tree would have a size of approximately 14 × 2 metres.





B Selected parts of the PVS sources of the plain memory abstraction

```
Plain_Memory[ State : Type ] : Theory
% plain memory definition and immediate utility results
Begin

  IMPORTING Memory_Change_2

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % Plain Memory Definition
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Plain_Memory : Type = [#
    mem : Memory_struct[State],
    states : PRED[State],
    ro_addr : PRED[Address],
    rw_addr : PRED[Address],
    other_actions : PRED[[State -> Result[State, Unit]]]
  #]

  pm : Var Plain_Memory

  addr : Var Address
  size : Var nat


  plain_memory?(pm) : bool =
    % there is an invariant hidden, see below
    unchanged_memory_invariant?(pm`mem, pm`states,
      union(pm`other_actions,
        memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr))),
      union(pm`ro_addr, pm`rw_addr))
    AND
    unchanged_memory_invariant?(pm`mem, pm`states,
                memory_write_transformers(pm`mem, pm`rw_addr),
                pm`ro_addr)
    AND
    unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)
    AND
```

```
  changed_memory_invariant?(pm‘mem, pm‘states, pm‘rw_addr)
  AND
  transformers_ok?(pm‘states,
    union(memory_read_transformers(pm‘mem, union(pm‘ro_addr, pm‘rw_addr)),
          memory_write_transformers(pm‘mem, pm‘rw_addr)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Utility results
%
% transformer invariants
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_invariant_read_transformers_ro_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm‘states,
                    memory_read_transformers(pm‘mem, pm‘ro_addr))

% proof status :-)
plain_memory_invariant_read_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm‘states,
                    memory_read_transformers(pm‘mem, pm‘rw_addr))

% proof status :-)
plain_memory_invariant_read_transformers_ro_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm‘states,
      memory_read_transformers(pm‘mem, union(pm‘ro_addr, pm‘rw_addr)))

% proof status :-)
plain_memory_invariant_write_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm‘states,
                    memory_write_transformers(pm‘mem, pm‘rw_addr))


% proof status :-)
plain_memory_invariant : Lemma
  plain_memory?(pm) Implies
```

```
    transformer_invariant?(pm`states,
      union(
        union(pm`other_actions,
              memory_read_transformers(pm`mem,
                                       union(pm`ro_addr, pm`rw_addr))),
        memory_write_transformers(pm`mem, pm`rw_addr)))
```

%%
%
% transformer invariants on blocks
%
%%

% proof status :-)

```
plain_memory_invariant_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformer_invariant?(pm`states, memory_read_transformers(pm`mem,
                                             address_block(addr, size)))
```

% proof status :-)

```
plain_memory_invariant_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size),  pm`rw_addr)
    Implies
      transformer_invariant?(pm`states,
              memory_write_transformers(pm`mem, address_block(addr, size)))
```

%%
%
% transformers_ok?
%
%%

% proof status :-)

```
plain_memory_transformers_ok_read_ro_rw : Lemma
  plain_memory?(pm)
    Implies
      transformers_ok?(pm`states,
          memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)))
```

% proof status :-)

```
plain_memory_transformers_ok_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformers_ok?(pm`states, memory_read_transformers(pm`mem,
                                        address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size),  pm`rw_addr)
    Implies
      transformers_ok?(pm`states,
              memory_write_transformers(pm`mem, address_block(addr, size)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged_memory_invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_unchanged_read_block_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_transformers(pm`mem, address_block(addr, size)),
              pm`rw_addr)

% proof status :-)
plain_memory_unchanged_read_block_ro_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_transformers(pm`mem, address_block(addr, size)),
              union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_write_block_ro_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
```

```
      unchanged_memory_invariant?(pm‘mem, pm‘states,
              memory_write_transformers(pm‘mem, address_block(addr, size)),
              pm‘ro_addr)

  % unchanged invariants of reading
  % proof status :-)
  plain_memory_unchanged_read_block : Lemma
    plain_memory?(pm) And
    subset?(address_block(addr, size), union(pm‘ro_addr, pm‘rw_addr))
      Implies
        unchanged_memory_invariant?(pm‘mem, pm‘states,
              memory_read_transformers(pm‘mem, address_block(addr, size)),
              address_block(addr, size))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % unchanged_memory_write_invariant
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  plain_memory_unchanged_block : Lemma
    plain_memory?(pm) And
    subset?(address_block(addr, size), pm‘rw_addr)
      Implies
        unchanged_memory_write_invariant?(pm‘mem, pm‘states,
                                          address_block(addr, size))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % changed_memory_invariant
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  plain_memory_changed_block : Lemma
    plain_memory?(pm) And
    subset?(address_block(addr, size), pm‘rw_addr)
      Implies
        changed_memory_invariant?(pm‘mem, pm‘states, address_block(addr, size))

End Plain_Memory
```

```
Plain_Mem_Properties[State : Type] : Theory
%
% primitive (non-datatype) read and writes on plain memory
Begin

  IMPORTING Plain_Memory[State]

  pm : Var Plain_Memory[State]

  s : Var State
  addr : Var Address
  size : Var nat

  % reading a block of memory is a transformer invariant

  % writing to a block of memory is a transformer invariant

  % proof status :-)
  plain_mem_write_list_read_list : Lemma
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) AND
      plain_memory?(pm) And
      pm`states(s) And
      subset?(address_block(addr, size), pm`rw_addr) And
      OK?(memory_write_list(pm`mem)(addr, bl)(s)) And
      OK?(memory_read_list(pm`mem)(size, addr)
          (next_state(memory_write_list(pm`mem) (addr, bl)(s))))
        Implies
          get_data(memory_read_list(pm`mem)(size, addr)
                  (next_state(memory_write_list (pm`mem) (addr, bl) (s))))
            = bl

  % proof status :-)
  plain_mem_q_read_list : Lemma Forall(q : [State -> Result[State, Unit]]) :
    OK?(memory_read_list(pm`mem)(size, addr)(s)) And
    OK?(q(s)) And
    OK?(memory_read_list(pm`mem)(size, addr)(next_state(q(s)))) And
    plain_memory?(pm) And
    unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                address_block(addr, size)) And
    addr + size <= max_linear And
    pm`states(s) And
```

```
    subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
      Implies
        get_data(memory_read_list(pm`mem)(size, addr) (next_state(q(s))))
          = get_data(memory_read_list(pm`mem)(size, addr) (s))

End Plain_Mem_Properties


Plain_Mem_Properties_2[State, Data : Type] : Theory
Begin
  IMPORTING Plain_Mem_Properties, Abstract_Read_Write

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
  s : Var State
  addr : Var Address
  data : Var Data

  % proof status :-)
  plain_memory_inv_pred_write_data : Lemma
    plain_memory?(pm) And
    pm`states(s) And
    subset?(address_block(addr, size(uidt(dt))), pm`rw_addr) And
    OK?(write_data(pm`mem, dt)(addr, data)(s))
      Implies
        pm`states(next_state(write_data(pm`mem, dt)(addr, data)(s)))

  % proof status :-)
  plain_memory_read_data_ok : Lemma
    plain_memory?(pm) AND
    pm`states(s) And
    in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
    valid_in_mem(pm`mem, dt)(addr)(s)
      Implies
        OK?(read_data(pm`mem,dt)(addr)(s))

  % proof status :-)
  plain_memory_write_data_ok : Lemma
    plain_memory?(pm) AND
    pm`states(s) And
    in_blessed_memory?(dt, addr, pm`rw_addr)
      Implies
        OK?(write_data(pm`mem,dt)(addr, data)(s))
```

```
% proof status :-)
% plain_memory_write_data_valid_TCC1 : TCC Obligation

% proof status :-)
plain_memory_write_data_valid : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      valid_in_mem(pm`mem, dt)(addr)
              (next_state(write_data(pm`mem, dt)(addr, data)(s)))

% proof status :-)
plain_memory_transformers_ok_write_data : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      transformers_ok?(pm`states,
              singleton(write_data(pm`mem, dt) (addr, data)))

% proof status :-)
plain_memory_transformer_invariant_write_data : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      transformer_invariant?(pm`states,
              singleton(write_data(pm`mem, dt) (addr, data)))

% proof status :-)
plain_memory_transformer_invariant_read_data : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr))
    Implies
      transformer_invariant?(pm`states,
                      singleton(read_data(pm`mem, dt) (addr)))

% proof status :-)
plain_memory_unchanged_memory_invariant_write_data : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
```

```
  in_blessed_memory?(dt, addr, pm`rw_addr)
    IMPLIES
      unchanged_memory_invariant?(pm`mem, pm`states,
          singleton(write_data(pm`mem, dt)(addr, data)),
          difference(union(pm`ro_addr, pm`rw_addr),
                              address_block(addr, size(uidt(dt)))))

% proof status :-)
plain_memory_unchanged_memory_invariant_read_data : Lemma
  plain_memory?(pm) AND
  in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr))
    IMPLIES
      unchanged_memory_invariant?(pm`mem, pm`states,
          singleton(read_data(pm`mem, dt)(addr)),
          union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_read_data_q_ok : Lemma
  Forall(q : [State -> Result[State, Unit]]) :
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) AND
    unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                              address_block(addr, size(uidt(dt)))) And
    pm`states(s) And
    OK?(q(s)) And
    valid_in_mem(pm`mem,dt)(addr)(s) Implies
      OK?(read_data(pm`mem, dt)(addr)(next_state(q(s))))

% proof status :-)
% plain_memory_read_data_q_same_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_data_q_same_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_data_q_same : Lemma
  Forall(q : [State -> Result[State, Unit]]) :
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) AND
    unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                              address_block(addr, size(uidt(dt)))) And
    pm`states(s) And
    OK?(q(s)) And
    valid_in_mem(pm`mem,dt)(addr)(s) Implies
```

```
        get_data(read_data(pm`mem, dt)(addr)(next_state(q(s)))) =
          get_data(read_data(pm`mem, dt)(addr)(s))


  % proof status :-)
  plain_memory_read_write_ok : Lemma
    plain_memory?(pm) AND
    pm`states(s) And
    in_blessed_memory?(dt, addr, pm`rw_addr)
      IMPLIES
        OK?((
                write_data(pm`mem,dt)(addr, data) ##
                read_data(pm`mem,dt)(addr)
            )(s)
           )

  % proof status :-)
  % plain_memory_read_write_res_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_read_write_res : Lemma
    plain_memory?(pm) AND
    pm`states(s) And
    in_blessed_memory?(dt, addr, pm`rw_addr)
      IMPLIES
        get_data((
                  write_data(pm`mem,dt)(addr, data) ##
                  read_data(pm`mem,dt)(addr)
                 )(s))
          = data

end Plain_Mem_Properties_2



Plain_Mem_Properties_3[State, Data1, Data2 : Type] : Theory
%
% reads after write at different places return the previous item
Begin
  Importing Plain_Mem_Properties_2, Memory_Change_To_Unit

  pm : Var Plain_Memory[State]

  dt1 : Var (interpreted_data_type?[Data1])
```

```
dt2 : Var (interpreted_data_type?[Data2])
s : Var State
addr1, addr2 : Var Address
data1 : Var Data1


% proof status :-)
plain_memory_read_write_other_ok : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) And
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(pm`mem,dt2)(addr2)(s)
    IMPLIES
      OK?
         ((
              write_data(pm`mem,dt1)(addr1, data1) ##
              read_data(pm`mem,dt2)(addr2)
          )(s)
         )


% proof status :-)
% plain_memory_read_write_other_res_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_write_other_res_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(pm`mem,dt2)(addr2)(s)
    IMPLIES
      get_data((
                  write_data(pm`mem,dt1)(addr1, data1) ##
                  read_data(pm`mem,dt2)(addr2)
               )(s))
         =
      get_data(read_data(pm`mem,dt2)(addr2)(s))
```

```
% proof status :-)
plain_memory_read_read_ok: Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  valid_in_mem(pm`mem,dt1)(addr1)(s) And
  valid_in_mem(pm`mem,dt2)(addr2)(s)
    IMPLIES
      OK?(
              ( read_data(pm`mem, dt1)(addr1) ##
                read_data(pm`mem, dt2)(addr2)
              )(s)
        )

% proof status :-)
% plain_memory_read_read_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_read_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_read: Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  valid_in_mem(pm`mem,dt1)(addr1)(s) And
  valid_in_mem(pm`mem,dt2)(addr2)(s)
    IMPLIES
      get_data(
              ( read_data(pm`mem, dt1)(addr1) ##
                read_data(pm`mem, dt2)(addr2)
              )(s)
             )
      = get_data(read_data(pm`mem, dt2)(addr2)(s))

End Plain_Mem_Properties_3
```

C CCSL sources for the specification of finite, boolean labelled binary trees

```
Begin NodeDescription : Adt
% enumeration all the possible observations that you can
% make when descending into a binary tree with boolean labeled leafs.
  Constructor
    Free : Carrier;          % found a true-labeled leaf
    Allocated : Carrier;     % found a false-labeled leaf
    Split : Carrier;         % found a node (without label)
    NoNode : Carrier         % error / path too long / tree too short
End NodeDescription

Type nat
Constant (>) : [nat,nat -> bool]

Begin BinaryWordSig : GroundSignature
% declare the necessary constants for binary words
  Importing BinaryWord
  Type BinaryWord

  Constant
    empty_word : BinaryWord;
    length : [BinaryWord -> nat];
    append : [BinaryWord, bool -> BinaryWord]
End BinaryWordSig

Begin Tree : Final ClassSpec
  % we have just one observation: what do you see, if you use the
  % binary word to descend into the tree.
  method
    node : [Self, BinaryWord] -> NodeDescription

  Assertion SelfVar bt : Self
    % assert a tree structure: nodes (Split) are never directly followed
    % by a NoNode (error). There must be at least a leaf inbetween.
    split_next:
      Forall(p : BinaryWord, b : bool) :
        node(bt, p) = Split IFF
          Not node(bt, append(p, b)) = NoNode;

    finite:
```

```
    Exists(n : nat) : Forall(p : BinaryWord) :
      length(p) > n Implies node(bt, p) = NoNode;

  % exclude the empty tree (without any nodes and leafs)
  non_empty:
      not node(bt, empty_word) = NoNode;
End Tree
```

D PVS sources of the abstract buddy allocator model

```
FinalAbstractBuddy[X : Type] : Theory
Begin
  Importing TreeOperations, Alignment, WordAddressConversion, Expt_Lemmas
  Importing finite_sets@finite_sets_minmax % for max over finite sets
  Importing more_finite_sets_lemmas

  c : Var (x_unit_final[X])

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % specific utility functions/predicates on trees
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  depth_upto?(c)(max : nat) : PRED[X] = Lambda(t : X) : depth(c)(t) <= max


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % free nodes
  %

  % type predicate for trees that have a free node at a certain depth
  % (certain functions below only work on trees with a free node)
  has_free_node_at_depth(c)(depth : nat) : PRED[X] =
    Lambda(t : X) : Exists(p : BinaryWord) :
        length(p) = depth And
        node(c)(t, p) = Free

  % proof status :-)
  % choose_free_path_TCC1 : TCC Obligation

  choose_free_path(c)(depth : nat, t : (has_free_node_at_depth(c)(depth)))
                                                                  : BinaryWord =
    choose(Lambda(p : BinaryWord) :
                length(p) = depth And
                node(c)(t, p) = Free)

  % proof status :-)
  length_choose_free_path : Lemma
    Forall(depth : nat, t : (has_free_node_at_depth(c)(depth))) :
      length(choose_free_path(c)(depth, t)) = depth
```

```
% proof status :-)
node_choose_free_path : Lemma
  Forall(depth : nat, t : (has_free_node_at_depth(c)(depth))) :
    node(c)(t, choose_free_path(c)(depth, t)) = Free

% proof status :-)
% find_free_above_depth_TCC1 : TCC Obligation


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% find_free_above_depth
%
% Tries to find a depth less than or equal to depth with a free node.
% If there is one, returns the maximal depth with a free node
% (which is the node that we split and allocate in).
find_free_above_depth(c)(t : X, depth : nat) : lift[nat] =
  IF Exists(n : upto(depth)) : has_free_node_at_depth(c)(n)(t)
  Then up(max[nat, <=]
      (Lambda(n : nat) : n <= depth And has_free_node_at_depth(c)(n)(t)))
  Else bottom
  Endif

% proof status :-)
find_free_above_depth_above : Lemma
  Forall(t : X, depth : nat) :
    up?(find_free_above_depth(c)(t, depth)) Implies
      down(find_free_above_depth(c)(t, depth)) <= depth

% proof status :-)
find_free_above_depth_has_free : Lemma
  Forall(t : X, depth : nat) :
    up?(find_free_above_depth(c)(t, depth)) Implies
      has_free_node_at_depth(c)(
              down(find_free_above_depth(c)(t, depth)))(t)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% make_node
%
% replace a free node on split_depth with a (non-minimal) tree
% with depth depth (containing just free nodes)
% returns the modified tree
```

```
make_node(c)(split_depth, depth : nat,
                     t : (has_free_node_at_depth(c)(split_depth))) : X =
  plug_subtree(c)(t, choose_free_path(c)(split_depth, t),
                                    make_empty_tree(c)(depth))

% proof status :-)
depth_make_node : Lemma
  Forall(split_depth, depth : nat, t : X, max_depth : nat) :
    has_free_node_at_depth(c)(split_depth)(t) And
    depth(c)(t) <= max_depth And
    split_depth + depth <= max_depth
      Implies
        depth(c)(make_node(c)(split_depth, depth, t)) <= max_depth

% proof status :-)
make_node_has_free : Lemma
  Forall(split_depth, depth : nat, t : X) :
    has_free_node_at_depth(c)(split_depth)(t) Implies
      has_free_node_at_depth(c)(split_depth + depth)
                            (make_node(c)(split_depth, depth, t))

% proof status :-)
make_node_allocated : Lemma
  Forall(path : BinaryWord, split_depth, depth : nat,
                       t : (has_free_node_at_depth(c)(split_depth))) :
    node(c)(t, path) = Allocated IFF
      node(c)(make_node(c)(split_depth, depth, t), path) = Allocated

% proof status :-)
make_node_compact_allocated : Lemma
  Forall(path : BinaryWord, split_depth, depth : nat, t : X) :
    has_free_node_at_depth(c)(split_depth)(compact_tree(c)(t)) Implies
      (node(c)(t, path) = Allocated IFF
         node(c)(make_node(c)(split_depth, depth,
                               compact_tree(c)(t)), path) = Allocated)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% State and Methods of the buddy allocator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% A state of the abstract buddy allocator consists of a
```

```
% start address (of the managed memory region),
% a maximal depth (which determines the size as size = 2 ** max_depth)
% and a tree. There are two invariants that must hold on a
% correct state:
% 1. the depth of the tree must be less than or equal than max_depth
% 2. start_addr must be size aligned (ie divisible by 2 ** level)
%   (otherwise nodes in the tree do not correspond to aligned
%    memory blocks,
%    needed for the alignment property (allocate_aligned)
%    and the liveness property allocate_live)
%
% Point 1 (depth less than max_depth) is needed because the utility
% predicate allocated_points (defined below, returns (addr, size) pairs
% for all allocated reagions) ignores Allocated leafs
% below max_depth. It ignores those leafs because the corresponding
% size would be a fraction of 1. To fix this problem one would
% either have to switch to sets of allocated addresses or have a
% separate predicate for allocated points (without size) or allow for
% fractional sizes. The problem shows only up in the proof of
% allocated_alive, because with allocated_points ignoring Allocated
% leafs below max_depth, you might have a free block, which contains
% somewhere in the middle an Allocated leaft below max_depth. Such a
% tree is of course not compacted by compact_tree, therefore allocation
% would fail.

State(c) : Type = [#
  max_depth : nat, % determines size = 2 ** max_depth
  start_addr : (aligned(max_depth)),
  % leave the two invariants aside for a while
  % wait until we really need them
  % tree : (minimal_depth_upto?(c)(max_depth))
  tree : (depth_upto?(c)(max_depth))
#]


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Towards the definition of allocate and free
%
% utility functions and TCC lemmas
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
```

% allocate_aux_TCC1 : TCC Obligation

% proof status :-)
% allocate_aux_TCC2 : TCC Obligation

% proof status :-)
% allocate_aux_TCC3 : TCC Obligation

% proof status :-)
% allocate_aux_TCC4 : TCC Obligation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% allocate_aux does everything necessary for allocation, except
% the binary word into address translation. It first compacts the tree.
% This is different to Udo's code, where compactification is performed
% after freeing and Udo has the additional invariant that trees are
% always compacted.
allocate_aux(c)(bs : State(c), ln2size : nat) : lift[[X, BinaryWord]] =
  Let ct = compact_tree(c)(bs‘tree)
  IN
    IF ln2size > bs‘max_depth Then bottom
    Else
      Cases find_free_above_depth(c)(ct, bs‘max_depth − ln2size) OF
        bottom : bottom,
        up(depth) :
          Let t2 = make_node(c)(depth,
                                bs‘max_depth − ln2size − depth, ct),
              alloc_path = choose_free_path(c)(bs‘max_depth − ln2size, t2)
          IN
            up(insert_alloc_leaf(c)(t2, alloc_path), alloc_path)
      EndCases
    Endif

% proof status :-)
depth_allocate_aux : Lemma Forall(bs : State(c), ln2size : nat) :
  up?(allocate_aux(c)(bs, ln2size)) Implies
    depth_upto?(c)(bs‘max_depth)(down(allocate_aux(c)(bs, ln2size))‘1)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% allocate

% proof status :-)
```

```
% allocate_TCC1 : TCC Obligation

% proof status :-)
% allocate_TCC2 : TCC Obligation

allocate(c)(bs : State(c), ln2size : nat) : lift[[State(c), nat]] =
  Cases allocate_aux(c)(bs, ln2size) OF
    up(tree_path) :
      up(bs With ['tree := tree_path'1],
         bs'start_addr +
              word_2_nat(pad_word_right(tree_path'2, bs'max_depth))),
    bottom : bottom
  EndCases


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% free

% proof status :-)
% get_alloc_path_TCC1 : TCC Obligation

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The address where a block is freed doesn't immediately determine
% a path to an Allocated leaf. To get the path one first has to chop off
% some padding on the right. The amount of padding depends on the
% unknown size of the freed reagion (or the unknown level of the
% Allocated leaf). Of course, when you look in the tree there is at most
% one suitable Allocated leaf (if there is none then the address to
% be freed was never allocated). The following function looks into
% the tree and guesses the path to the Allocated leaf (if there is one).
get_alloc_path(c)(t : X, p : BinaryWord) : lift[BinaryWord] =
  IF Exists(op : (non_zero_prefixes(p))) : node(c)(t, op) = Allocated
  Then up(choose(Lambda(op : (non_zero_prefixes(p))) :
                                   node(c)(t, op) = Allocated))
  Else bottom
  Endif

% proof status :-)
% free_TCC1 : TCC Obligation

% proof status :-)
% free_TCC2 : TCC Obligation

% proof status :-)
```

% free_TCC3 : TCC Obligation

```
free(c)(bs : State(c), addr : nat) : lift[State(c)] =
  IF addr < bs`start_addr OR
      addr >= bs`start_addr + expt(2, bs`max_depth) OR
      bottom?(get_alloc_path(c)(bs`tree,
                      nat_2_word_len(addr - bs`start_addr, bs`max_depth)))
  Then bottom
  Else
    up(bs With [`tree :=
      insert_free_leaf(c)(bs`tree,
        down(get_alloc_path(c)(bs`tree,
                      nat_2_word_len(addr - bs`start_addr, bs`max_depth))))])
  Endif
```

%%%
%
% Properties
%
%%%

%%%
% allocated_points
%
% returns a set of (addr, ln2size) pairs for everything allocated in
% the tree bs`tree
% ln2size is actually the level, so the real size is
*% 2 ** ln2size * base_block_size*
% addresses are block addresses, that is they are implicitely
% taken modulo base_block_size
%
% Comment about the quantifier "Exists(p : BinaryWord)":
% In principle the addr determines the path p, so one could question
% why not using "nat_2_word(addr - bs`start_addr)" instead of p.
% The problem with the latter approach is that one needs additional
% checks to make sure addr is inside the heap of the buddy allocator.
% Once you converted the addr to a BinaryWord, you cannot check this
% anymore. The version below avoids these checks, however, it still
% ensures
% bs`start_addr <= addr < bs`start_addr + expt(2, bs`max_depth)
% and
% length(p) <= bs`max_depth
%

```
allocated_points(c)(bs : State(c)) : PRED[[nat, nat]] =
  Lambda(addr, ln2size : nat) :
    Exists(p : BinaryWord) :
      node(c)(bs`tree, p) = Allocated And
      length(p) = bs`max_depth - ln2size And
      addr = bs`start_addr + word_2_nat(pad_word_right(p, bs`max_depth))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% free_block
%
% Think about it: The negation of allocated_points does not tell
% you about free blocks!
% Looking at "not allocated_points" you only see that a certain
% (addr, size) pair is not allocated, you don't see if the region
% is really free, ie. if there are other allocated blocks inside
% or overlapping.

free_block(c)(bs : State(c), addr, ln2size : nat) : bool =
  bs`start_addr <= addr And
  addr + expt(2, ln2size) <= bs`start_addr + expt(2, bs`max_depth) And
    Forall(oaddr, osize : nat) :
      allocated_points(c)(bs)(oaddr, osize)
        Implies
          oaddr + expt(2, osize) <= addr OR
          addr + expt(2, ln2size) <= oaddr

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% aligned_free_block
%
% Check if there is a free block of a certain size
% Size is again the level, ie real_size = 2 ** size * base_block_size
% (only needed below in allocate_alive).

aligned_free_block(c)(bs : State(c), ln2size : nat) : bool =
  Exists(addr : nat) :
    aligned(ln2size)(addr) And          % addr is size aligned
    free_block(c)(bs, addr, ln2size)


% proof status :-)
allocate_correct : Theorem Forall(bs : State(c), ln2size : nat) :
  up?(allocate(c)(bs, ln2size)) Implies
    free_block(c)(bs, down(allocate(c)(bs, ln2size))`2, ln2size) And
```

```
    allocated_points(c)(down(allocate(c)(bs,ln2size))`1) =
      union(allocated_points(c)(bs),
              singleton(down(allocate(c)(bs,ln2size))`2, ln2size))


% proof status :-)
allocate_aligned : Theorem Forall(bs : State(c), ln2size : nat) :
  up?(allocate(c)(bs, ln2size)) Implies
    aligned(ln2size)(down(allocate(c)(bs, ln2size))`2)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Lemmas towards free_correct
%

% proof status :-)
get_alloc_path_allocated : Lemma Forall(t : X, p : BinaryWord, len : nat) :
  node(c)(t, p) = Allocated Implies
    get_alloc_path(c)(t, pad_word_right(p, len)) = up(p)


% proof status :-)
free_correct : Theorem Forall(bs : State(c), addr, ln2size : nat) :
  allocated_points(c)(bs)(addr, ln2size) Implies
    up?(free(c)(bs, addr)) And
    allocated_points(c)(down(free(c)(bs, addr))) =
      remove((addr, ln2size), allocated_points(c)(bs))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Lemmas towards allocate_live
%

% proof status :-)
free_block_start_addr : Lemma
  Forall(bs : State(c), addr, ln2size : nat) :
    free_block(c)(bs, addr, ln2size) Implies
      bs`start_addr <= addr

% proof status :-)
free_block_end_addr : Lemma
  Forall(bs : State(c), addr, ln2size : nat) :
    free_block(c)(bs, addr, ln2size) Implies
      addr < bs`start_addr + expt(2, bs`max_depth)
```

```
% proof status :-)
free_block_size_less : Lemma
  Forall(bs : State(c), addr, ln2size : nat) :
    free_block(c)(bs, addr, ln2size) Implies
      ln2size <= bs`max_depth

% proof status :-)
% free_block_not_allocated_TCC1 : TCC Obligation

% proof status :-)
% free_block_not_allocated_TCC2 : TCC Obligation

% proof status :-)
% This lemma says, that on the path that leads to a free memory block,
% there is no Allocated leaf. Further there is no Allocated leaf
% inside the free block.
% proof status :-)
free_block_not_allocated : Lemma
  Forall(bs : State(c), addr, ln2size : nat, p : BinaryWord) :
    aligned(ln2size)(addr) And
    free_block(c)(bs, addr, ln2size) Implies
    Let addr_path = nat_2_word_len(addr - bs`start_addr, bs`max_depth)
    IN
      is_prefix(take(p, bs`max_depth - ln2size), addr_path)
        Implies
          Not node(c)(bs`tree, p) = Allocated

% proof status :-)
% Strengthen the previous result: There is a completely empty subtree
% somewhere on the path that leads to a free block.
% (A subtree is completely empty if all leafs in it are Free leafs.)
% proof status :-)
free_block_exists_completely_empty : Lemma
  Forall(bs : State(c), addr, ln2size : nat) :
    aligned(ln2size)(addr) And
    free_block(c)(bs, addr, ln2size) Implies
      Let addr_path = nat_2_word_len(addr - bs`start_addr, bs`max_depth)
      IN
        nonempty?(Lambda(len: nat):
                      completely_empty?(c)(bs`tree, take(addr_path, len)))

% proof status :-)
% free_block_min_completely_free_path_TCC1 : TCC Obligation
```

```
% proof status :-)
% Now that we know there is a completely empty subtree, we can take
% the biggest such (which compact_tree will replaced with a Free leaf).
% It will cover the free block (it might be bigger) and the path to it
% ends either in a Split node or a free leaf.
% proof status :-)
free_block_min_completely_free_path : Lemma
  Forall(bs : State(c), addr, ln2size : nat) :
    aligned(ln2size)(addr) And
    free_block(c)(bs, addr, ln2size) Implies
      Let addr_path = nat_2_word_len(addr - bs`start_addr, bs`max_depth),
          free_depth = min_nat.min(Lambda(len : nat) :
              completely_empty?(c)(bs`tree, take(addr_path, len)))
      IN
        free_depth <= bs`max_depth - ln2size And
        (node(c)(bs`tree, take(addr_path, free_depth)) = Free Or
         node(c)(bs`tree, take(addr_path, free_depth)) = Split)


% proof status :-)
has_free_node_at_depth_compact_tree : Lemma
  Forall(bs : State(c), ln2size : nat) :
    ln2size <= bs`max_depth And
    aligned_free_block(c)(bs, ln2size) Implies
      Exists(n: upto(bs`max_depth - ln2size)):
        has_free_node_at_depth(c)(n)(compact_tree(c)(bs`tree))


% proof status :-)
allocate_alive : Theorem Forall(bs : State(c), ln2size : nat) :
  aligned_free_block(c)(bs, ln2size) Implies
      up?(allocate(c)(bs, ln2size))


% proof status :-)
free_alive : Theorem Forall(bs : State(c), addr, ln2size : nat) :
  allocated_points(c)(bs)(addr, ln2size) Implies
    up?(free(c)(bs, addr))

End FinalAbstractBuddy
```

E References

[1] E. Poll B. Jacobs. Java program verification at nijmegen: Developments and perspective. In N. Yonezaki K. Futatsugi, F. Mizoguchi, editor, *Software Security – Theories and Systems (ISSS'03)*, volume 3233 of *LNCS*, pages 134–153. Springer, 2004.

[2] Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007. To appear.

[3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[4] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[5] Sarah Hoffmann. Formalising pc hardware: A model of the x86 architecture. Master's thesis, TU Dresden, 2003.

[6] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg.

[7] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, 2005.

[8] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, March 2006. available via http://www.intel.com.

[9] longjmp, siglongjmp — non-local jump to a saved stack context. Linux manual page.

[10] Scott McPeak. Elsa: The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/.

[11] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[12] Model-driven architecture. Article in Wikipedia, the free encyclopedia, see http://en.wikipedia.org/wiki/Model-driven_architecture.

[13] H. Tews. *Coalgebraic Methods for Object-Oriented Specification*. PhD thesis, University of Dresden, 2002. Available via wwwtcs.inf.tu–dresden/∼tews/PhD.

[14] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps. Draft, 2004. Available from `wwwtcs.inf.tu-dresden.de/∼tews/science.html`.

[15] H. Tews. Wellbehaved memory on top of virtual memory. Presentation on Nicta's International Workshop on System Verification, August 2007. Slides available via www.cs.ru.nl/∼tews.

[16] Hendrik Tews. Olmar : Process C++ Programs in Ocaml. http://www.cs.ru.nl/∼tews/olmar/.

[17] Marcus Völp Udo Steinberg, Hendrik Tews. Nova hypervisor & vmm. draft documentation in preparation.

[18] Daniel S. Wilkerson, Karl Chen, and Scott McPeak. Oink: a Collaboration of C++ Static Analysis Tools. http://www.cubewano.org/oink/.