# Verified Software: Theories, Tools, and Experiments
# VSTTE 2006, Workshop proceedings

K. Rustan M. Leino
leino@microsoft.com

Wolfram Schulte
schulte@microsoft.com

August 2006

Technical Report
MSR-TR-2006-117

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

This one-day FLoC 2006 workshop on Verified Software continues the discussion initiated at the IFIP Working Conference on Verified Software in Zurich, Switzerland in October 2005. Consisting of contributed papers, invited talks, and a discussion session, the workshop focuses on the development of systematic methods for specifying, building, and verifying high-quality software.

## Program committee

- Marieke Huisman, INRIA Sophia Antipolis, France
- K. Rustan M. Leino (co-chair), Microsoft Research, USA
- Claude Marché, Université Paris-Sud, France
- Wolfgang J. Paul, Saarland University, Germany
- Wolfram Schulte (co-chair), Microsoft Research, USA
- Cesare Tinelli, University of Iowa, USA
- Arnaud Venet, Kestrel Technologies, USA

## Invited speakers

- Jim Grundy, Research Scientist, Strategic CAD Labs, Intel Corporation, USA. *Challenges and Lessons for Software Verification*
- Sumit Gulwani, Researcher, Microsoft Research, USA. *Program Verification using Probabilistic Techniques*

## Discussion

Moderated by Tony Hoare, Microsoft Research, UK.

# Formalising a High-Performance Microkernel

Kevin Elphinstone Gerwin Klein Rafal Kolanski

National ICT Australia *, Sydney, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{kevin.elphinstone|gerwin.klein|rafal.kolanski}@nicta.com.au

## Abstract

This paper argues that a pragmatic approach is needed for integrating design and formalisation of complex systems. We report on our approach to designing the seL4 operating system microkernel API and its formalisation in Isabelle/HOL. The formalisation consists of the systematic translation of significant parts of the functional programming language Haskell into Isabelle/HOL, including monad-based code. We give an account of the experience, decisions and outcomes in this translation as well as the technical problems we encountered together with our solutions. The longer-term goal is to demonstrate that formalisation and verification of a large, complex, OS-level code base is feasible with current tools and methods and is in the order of magnitude of traditional development cost.

## 1. Introduction

Sometimes an incomplete engineering approach is better than a complete, precise mathematical solution. As for normal software, so also for formalisation and verification, seeking the perfect solution to a problem is at odds with the reality of limited development costs.

The overall aim of our project is to design and verify a microkernel-based operating system (Sect. 2). In this paper we argue that a pragmatic approach is essential for large-scale projects such as operating system (OS) verification. We aim to be pragmatic in the sense that we are using a method that on first sight is not suitable, because it will not work in general, because it does not provide a complete solution to the problem, and because it is not fully automatic where in theory it could be. Instead it is semi-automated, systematic, cheap, easy to employ, and still gives the desired result.

Another of our overall goals is to demonstrate that formalisation and verification of a large, complex OS-level code base is feasible with current tools and methods and that the cost of this is in the same order of magnitude as traditional development cost.

Our methodology is to develop an executable OS prototype in the high-level programming language Haskell (done by the OS team), then translate it to a formal specification (done by the theorem proving team). The result will be the basis for refinement into a high-performance C implementation of the OS, as well as the basis for a further abstraction to verify security properties.

This paper focuses on the techniques employed to achieve a practical translation process, the observations and lessons learnt. Its main contributions are:

- a simple, pragmatic method for making the benefits of formal verification and specification available to system architects in traditional system design. It retains traditional means for testing and validation while avoiding the need for unfamiliar specification languages (Sect. 2).
- a practical method for translating complex, real-life, monad-based Haskell [17] code to Isabelle/HOL, detailed in Sect. 3.
- experience from conducting a large scale verification project (Sect. 4), in particular creating a large, complex specification within a short time and with few resources, as would be common in an industrial setting.

At this early stage we can report that the methodology has been successful and beneficial so far. The formalisation cost was significantly lower than the implementation and testing cost, which is a significant improvement to our earlier experience on formalising and verifying parts of the C implementation of the L4 microkernel. The methodology resulted in a fully working microkernel prototype, implemented in Haskell, formalised in Isabelle/HOL, running normal ARM binaries through a simulator. The formalisation has uncovered a number of problems with the high-level language prototype, including a potentially unbounded operation (Sect. 4).

Although we have only concluded the formalisation stage so far, and have not proceeded to verification on that part of the project, the formalisation already implies one theorem: all system calls terminate.

## 2. seL4

A microkernel is an OS kernel designed to be minimal in code size and concepts. The kernel is the part of the OS that runs in the privileged mode of the hardware. L4 is a widely deployed second generation microkernel [20] providing the improved reliability and flexibility of the microkernel approach while overcoming the performance limitations of its predecessors. The current L4 implementation is on the order of 10,000 lines of C++ and assembler code.

The seL4 project is a descendant of L4, and aims to provide a secure foundation for high-end embedded systems development (e.g. mobile phones or PDAs). The security goals address two general areas that are lacking in the existing L4 API: communication control between applications, and kernel physical memory management. Control of communication is critical for both providing isolation guarantees between subsystems, and providing confinement guarantees of information possessed by an application. Control of physical memory consumed by the kernel is critical for providing availability guarantees for kernel services, and also for the predictability of their execution times.

On embarking on the seL4 project, we wanted an approach that had the following properties, while enabling the exploration of the design space of potential API solutions that address the issues outlined above

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

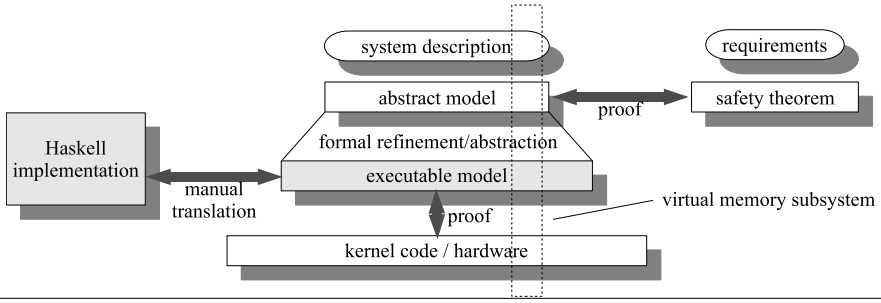



**Figure 1.** Overview

- The resulting API specification must be precise. Natural language manual-like descriptions are ambiguous and unsatisfactory.
- The approach must expose enough of the implementation details to allow the experimenter to be convinced a high performance implementation is possible.
- It should provide a method for gaining experience with the API by allowing construction of higher-level systems on top.
- It must be readily amenable to formalisation.
- The approach must be usable by kernel programmers who are not adept in formal methods.

The approach taken was to use literate Haskell [17] to specify and implement the seL4 API. Haskell, as a functional programming language, is not a large paradigm shift for typical kernel programmers. This might sound surprising as OS kernels are not usually developed in functional languages, but the proposition came from the OS group, not the verification group. Haskell is side-effect free, and at the same time allows us to explore implementation details of the kernel if desired.

This methodology is in part born out of a pilot project on L4 that we conducted to investigate two aspects of the feasibility of kernel verification: a formalisation of the L4 API using the B Method (topmost horizontal layer in Fig. 1) [19], taking about 6 person months, and a full refinement proof for a non-trivial subsystem (virtual memory) of L4 using Isabelle/HOL (vertical slice in Fig. 1) [26], taking about 18 person months. In the current work, the Haskell prototype serves as a solid, validated basis for the design, formalisation and verification of seL4. The formalisation in this paper occupies the horizontal layer between the most abstract model and the C code.

For validation, to enable the API to be used without requiring a real kernel implementation together with all the complexities of managing real hardware, we created a simulator that implements the ARM processor user-level instruction set and that transfers control to the Haskell kernel for hardware events like page faults and system calls. The simulator enables normal ARM application binaries, compiled for instance from C, to be executed on our kernel prototype.

At the present time, we have an initial seL4 API in Haskell together with the simulator executing ARM binaries. We have formalised this implementation using the mechanism described in Sect. 3 and are now further validating the API by attempting to prove the first security properties and at the same time porting our high-level application environment [15] to the new platform.

As observed in the design iterations so far, we expect to be able to readily adapt the API as we gain more experience in its use for building systems on top of the kernel, without the time consuming debugging usually associated with kernel programming and without large time investments for tracking changes in the formalisation.

## 3. Translating Haskell

This section gives an overview of our translation from Haskell to Isabelle/HOL. Although some of the solutions below are tailored to our specific problem, they are more generally useful and they demonstrate that interactive specification and theorem proving tools like Isabelle/HOL are suitable for pragmatic, large scale projects.

After discussing our choice of logic in Sect. 3.1, we describe the translation process itself in Sect. 3.2. Our main aim is to keep the translated code readable for interactive verification. Correctness, in particular with respect to partiality, is not our main concern. What will be implemented in the end is a refinement of the verified formal construct in Isabelle. The original Haskell code serves to validate the API design, not the implementation. Danielsson et al [5] show why partiality does not matter in this translation if the program is shown to terminate.

The last three subsections focus on particularly interesting parts of the translation: termination (Sect. 3.3), monads (Sect. 3.4), and the `Dynamic` extension of GHC (Sect. 3.5).

### 3.1 HOL or HOLCF?

Based on experience in our aforementioned pilot project, the verification tool of choice was the generic interactive theorem prover Isabelle which provides two logic instantiations that might be suitable: HOL and HOLCF.

As the name suggests, HOLCF is an implementation of Scott's logic of computable functions on top of Isabelle/HOL. It is well suited for faithfully describing features of Haskell such as partial functions and lazy evaluation. The Programatica project [9] attempts to automatically translate Haskell into Isabelle/HOLCF. Even though automatic translation would be ideal, we chose HOL over HOLCF, because Programatica at the time of writing was not able to parse our code base, because partial functions and lazy data structures do no play a major role in our code, and because HOLCF as a logic is more heavyweight than HOL, introducing reasoning about domains and continuous functions.

As the pilot study clearly showed that most of the effort will be spent in the later stages of the project, we made the trade-off towards more work in the specification phase instead of more complex reasoning later.

### 3.2 Types and Terms

The translation proceeded by creating one Isabelle theory per Haskell module. In the case of circular module dependencies which are possible with GHC, we created two Isabelle theories for one module — one with type and constant declarations only, the other with the corresponding definitions.

The bulk of the translation from Haskell to Isabelle/HOL consists of straightforward purely syntactic transformations, some of them just symbol replacements like converting Haskell's `->` function arrow into Isabelle's $\Rightarrow$, and Haskell's prefix notation for type constructors into postfix notation in Isabelle/HOL. The frequent simple case of algebraic data type declarations like `data D a = C1 a | C2 a Int` is trivial to translate; the general case with labelled fields like

```
data D a b = C1 { f1:a } | C2 {f1:a } { f2:Int }
```

has no direct counterpart in Isabelle, but can be simulated by defining separate field selector and update functions. If the appropriate naming and type conventions are respected, the update functions automatically become available in Isabelle as record update syntax.

For many basic terms no translation is required (Haskell and Isabelle syntax and semantics coincide apart from laziness), for most, simple token replacements already do the trick.

Among the more interesting constructs are `let`, where Isabelle does not allow recursive references. These would have to be lifted out and declared in a separate function, although this did not occur in our application. The list comprehensions `[e | pat <- xs, P pat ]` that can be translated directly into `[pat∈xs . P pat]` are only a small subset of the Haskell98 standard, but again were sufficient for our application. For more complex list comprehension expressions a separate function declaration might be necessary. Patterns in lambda, `case`, `let` and list comprehension expressions are restricted in Isabelle. Isabelle allows basic tuple patterns in `let` and lambda, and non-nested constructor patterns only for `case` and primitive recursion. More deeply nested patterns were translated to selectors. For example, for the `option` datatype with the constructors `Some 'a` and `None` and the selector `the (Some x) = x`, the expression `let Some x = f` would become `let x = the f`. More complex patterns in case construct are translated into one or more predicates combined with a let statement for name binding. For example, `case x of p1 -> t1; ...` is translated into `if is_p1 x then let p1 = x in t1 else ...` This is similar to the translation the Haskell98 [17] report gives into the Haskell core language. The difference here is that we use this expansion only when strictly necessary to keep the translated code as close as possible to the original.

Incomplete pattern matches are mapped to the value `undefined` in Haskell which is semantically equivalent to $\bot$, a non-terminating program. Compilers typically abort the program with an error message when `undefined` or `error`, which takes a message as its argument, are evaluated. We handle incomplete pattern matches in the standard Isabelle/HOL way: they are mapped to the value `arbitrary` which exists for every type, but is left unspecified. Since HOL is a logic of total functions, this value exists, but nothing is known about it. We also map explicit calls to `undefined` and `error` to `arbitrary`. This corresponds neatly with Haskell's lazy evaluation. In Haskell the error is only raised when `undefined` is evaluated, i.e. when it contributes to the result of a function. In Isabelle, proofs about the result of the same function only fail due to `arbitrary` when the constant contributes to the result.

Our first instance of this translation process was almost completely manual and still only took a small fraction of the original implementation cost in terms of effort. In the meantime, we have automated most of this process. Our tool is highly incomplete and manages an estimated 90% of the overall translation work automatically with the remaining 10% supplied manually as stubs or sections of translated code. The main lesson from this is that although Haskell is a very rich language and a complete, fully faithful translation is a sizeable project on its own, the easy, incomplete solution does work in practice. It is sufficient even for complex, real implementations of software on the OS level to be able to translate the part of Haskell with a relatively straightforward correspondence to HOL.

A trivial, but important detail was maintaining, as far as possible, a 1:1 correspondence with the Haskell code in both naming and the visual layout of functions. Since the translation was manual anyway, we had initially started out to translate concepts instead of syntax, along the way introducing slight abstractions. It quickly became clear that the better way for ease of translation and tracking change to the original was to translate purely syntactically first, and develop abstractions by proofs later if necessary.

As mentioned above, our application does not use lazy data structures and lazy evaluation as an essential feature. Laziness occurred in expressions like `zip xs [1..]` which could easily be translated to `zip xs [1..length xs]`.

### 3.3 Termination

Using HOL instead of HOLCF introduces the problem that all functions must be total. As mentioned before, this is our intention anyway, but it introduces an additional proof burden when writing the specification.

Fortunately termination for the bulk of the kernel code is obvious as it does not contain recursion. These parts can be handled by Isabelle's **constdef** which introduces a new constant as an abbreviation of existing constants. Apart from one instance, all recursion occurring in the code was easy to handle using Isabelle's primitive and well-founded recursion constructs. The one difficult instance concerns the one long-running operation of the seL4 kernel: revoking a capability. This is reflected in the code in a mutual recursion over four different functions that traverse the so-called mapping database which keeps track of capability derivations. Isabelle/HOL does support well-founded recursion in one argument, but it currently does not directly support arbitrary mutual recursion. To define these functions, we instead build one recursive function that takes the union of the original parameters together with an additional parameter that determines which of the branches is to be executed. This momentarily introduces more complexity and large, ugly terms, but the original function definitions as they appear in Haskell can then be easily derived as lemmas. For validation and proofs we use these lemmas, not the large construct containing all recursive branches. This technique was documented in detail by Slind [23].

Termination of this function still was nontrivial. The algorithm follows pointers in a data structure that models physical machine memory. We have shown a similar mechanism to terminate in the pilot study [18], but, since we still expect changes from the ongoing validation of the seL4 API in real systems, we would at this stage ideally like to avoid deep proofs that might be obsoleted faster than they were produced. A very simple method of at least guaranteeing that termination depends on the pointer parameter only is the observation that the set of machine words is finite and that traversing the tree will visit each pointer at most once. This termination criterion is easily accepted by Isabelle.

### 3.4 Monads

Since microkernels are inherently state-based, the Haskell implementation of seL4 uses monads [21] heavily to encapsulate this state. On the one hand this explicit state representation is much closer to HOL than for instance ML's implicit program state. On the other hand, faithfully representing Haskell monads in Isabelle/HOL is problematic.

The main difficulty is that Haskell uses type constructor classes for describing monads abstractly. A monad is a structure of type `'a m` where `'a` is a type variable and `m` a type constructor (like `list` or `option`), implementing two functions `return :: 'a ⇒ 'a m`

and `bind :: 'a m ⇒ ('a ⇒ 'b m) ⇒ 'b m`, written `_ >>= _`. Although there is no way to enforce this in Haskell, to form a monad, the two operations additionally have to satisfy the three monad laws.

Isabelle does provide single parameter axiomatic type classes, but it does not provide constructor classes, and can hence not express monads in the same abstract fashion. There is, however, nothing stopping us from defining concrete monads in Isabelle. The seL4 implementation uses three monads: a state transformer, a state transformer with an exception (`ErrorT`) monad on top, and a state transformer with two exception monads on top. They are easily formalised:

```
('s,'a) state_monad = 's ⇒ 'a × 's
return a ≡ λs. (a, s)
f >>= g ≡ λs. let (v, s') = f s in g v s'
gets f ≡ λs. (f s, s)
modify f ≡ λs. ((), f s)
('s,'a,'b) error_monad = ('s, 'a + 'b) state_monad
returnOk ≡ return ∘ Inr
throwError ≡ return ∘ Inl
lift f v ≡ case v of Inl e ⇒ throwError e
           | Inr v' ⇒ f v'
f >>=E g ≡ f >>= lift g
```

Note that we did not formalise the monad transformer `ErrorT`, but instead the result, an `ErrorT StateMonad`. The functions `Inl` and `Inr` are the projections into the sum type. We leave out the formal definition of the `ErrorT (ErrorT StateMonad)`, it is analogous and introduces another sum type in the result.

We initially defined `('s,'a) state_monad` in the usual way as a data type with the only constructor `State 's ⇒ 'a × 's`. This ensures that `state_monad` is different from the function space `'s ⇒ 'a × 's` and makes conversions between them explicit. Later, for reasoning about the state monad, these explicit conversions got in the way. Apart from purely algebraic reasoning, we often showed equality of two state monads by extensionality (being equal if they yield the same results for all start states). Stating this without explicit conversions was more natural and provided smoother automation.

It was easy to show that the three monad laws hold for all of the instantiations, and it was also not hard to provide a slightly modified `do`-notation where `do x ← f; g x od` stands for `bind f (λx. g)`.

We opted not to use Isabelle's constant overloading, but instead chose different names for each `bind` and `return` implementation with corresponding `do`-notation (`doE`, `doEE`). The reason is again the absence of constructor classes. To express the type of `return` for all implementations, we would have to generalise it to `'a ⇒ 'b` which in turn dilutes the value of type checking the specification. That means we traded off a small notational overhead against higher assurance through type checking. In fact, we found the notational overhead made the specification clearer than the original Haskell code because in its nested `do`-blocks it was often not obvious in which monad the operations are performed.

Fig. 2 shows a typical example of translated monadic code and demonstrates how more complex case patterns are resolved.

For specification purposes, this concrete treatment of monads proved fully adequate. The main disadvantage is that we cannot reason abstractly about monads just in term of monad laws, which could lead to duplication of theorems. So far this did not turn out to be a problem. We mostly had to reason about the behaviour of the state monad, which involved lemmas specific to state monads, not lemmas about monads in general. Scalability was not a problem. For some programs it might turn out inconvenient to not have monad transformers available as such, but only their results. Applying significantly more than three transformers (as in our case) is unlikely to occur in practice, though.

**Haskell:**

```
activateThread = do
  thread <- getCurThread
  state <- getWaitState thread
    case state of
      NotWaiting -> return ()
      WaitingToSend { pendingReceiveCap = Nothing } ->
        doIPCTransfer thread (waitingIPCPartner state)
      WaitingToReceive {} ->
        doIPCTransfer (waitingIPCPartner state) thread
      _ -> error "Current thread is blocked"
```

**Isabelle/HOL:**

```
activateThread ≡
do thread ← getCurThread;
   state ← getWaitState thread;
   case state of
     NotWaiting ⇒ return ()
   | WaitingToSend eptr badge fault cap ⇒
       if cap = None then
         doIPCTransfer thread (waitingIPCPartner state)
       else arbitrary
   | WaitingToReceive eptr ⇒
       doIPCTransfer (waitingIPCPartner state) thread
   | _ ⇒ arbitrary
od
```

**Figure 2.** Typical monad code translation

Although the method described above is very lightweight and proved adequate so far, it would be more satisfactory and scalable to be able to directly emulate constructor classes in Isabelle.

Dawson [6] shows that abstract reasoning about monads is possible in Isabelle/HOL by declaring a new type `'a m` that encodes type constructor application. This makes the types of `return` and `bind` and the corresponding laws directly expressible. Unfortunately, this technique prohibits instantiation.

Lüth et al [3, 16] have extended Isabelle with parameterised theories. They show how this mechanism enables an abstract treatment of monads together with a convenient instantiation mechanism. The only drawback of the method is scalability of another kind: it relies on Isabelle's proof terms to produce the required instantiations. Proof terms currently consume a significant amount of additional resources (mainly memory). For small to medium-sized developments this does not pose a problem, but we expect the size of our proofs to go beyond the limits of current ML systems if proof terms are switched on.

Huffman [14] uses a method similar to Dawson's for modelling monads in Isabelle/HOLCF which he recently extended to Isabelle/HOL [13]. Instead of declaring a new type for all type constructors, he creates an axiomatic class of type constructors and defines a new type for each specific type constructor. For example, instead of showing that `'a option` is of class `monad`, one declares a new type `Option` and instead shows that this is of class `monad`. It can then be used as `'a · Option` where `·` is a new operator for applying type constructors to types. The argument type `'a` is restricted to the class of representable types which are basically types whose values can be enumerated. The approach is flexible, allows abstract reasoning, generic `do`-notation, monad transformers, and instantiation. It is, however, cumbersome to use because it requires explicit conversion between e.g. `'a option` and `'a · Option` that are not present in the Haskell code.

### 3.5 Dynamic

The `Dynamic` extension of GHC to Haskell98 allows a limited form of type casting: automatic conversion of monomorphic types to the type `Dynamic` and back.

This extension is used in the kernel implementation to model physical memory as a map from addresses (machine words) to a tuple of dynamic objects and their size:

```
psMap :: Map (PPtr w) (Int, Dynamic)
```

Kernel objects belong to the `Storable` type class and implement operations which among others allow their storage to (`storeObject`) and from (`loadObject`) this physical memory.

The question therefore is how to define this type class storable and how to represent `Dynamic` in Isabelle/HOL. These two points are related: had we not wanted to define a type class, the naïve solution of modelling `Dynamic` as the union (a sum or datatype) of all types we possibly might want to store would work. Because some of the types to be stored have parameters, so would the union. Since the class `storable` already describes at least one type variable, this conflicts with the fact that Isabelle supports single parameter type classes only.

We therefore chose a concrete type that is large enough to support an injection of all storable objects: `word8 list` where `word8` is the type of 8 bit machine words. This choice was arbitrary, we could just as well have chosen natural numbers or anything else large enough. We picked `word8 list`, because we already had some of the infrastructure for encoding/decoding other types into it available from our work on a memory model for C pointers [27]. What is new here is lifting these encodings to more complex data structures by using parser combinators.

The axiomatic type class `storable` is built up as follows in Isabelle/HOL.

```
axclass to_from_byte < type
to_byte :: 'a::to_from_byte ⇒ word8 list
from_byte :: word8 list ⇒
             ('a::to_from_byte × word8 list) option

axclass storable < to_from_byte
from_byte (to_byte x @ xs) = Some (x, xs)
```

We use the class `to_from_byte`, a subclass of Isabelle's default `type`, to restrict the type of the two overloaded constants `to_byte` and `from_byte`. The subclass `storable` introduces the defining axiom. The constant `from_byte` has a slightly more complex type than might be expected, because we are interested in what remains of the stream when we have read an object (`@` is the append operator).

We can now define a combinator and an extractor:

```
(f1 -- f2) bs ≡ let res1 = f1 bs;
                    res2 =
                      case res1 of None ⇒ None
                      | Some (obj, rem) ⇒ f2 rem
                in case res2 of None ⇒ None
                   | Some (obj2, rem2) ⇒
                       Some
                        ((fst (the res1), obj2), rem2)

x ▷ f ≡ case x of None ⇒ None
        | Some (y, rem) ⇒ Some (f y, rem)
```

This allows us to build more complex types from existing ones. For example, if we have already proved that `bool::storable`, the datatype used to model capability rights is introduced easily:

```
datatype cap_rights = CapRights bool bool bool bool

to_byte (CapRights b1 b2 b3 b4) =
to_byte b1 @ to_byte b2 @ to_byte b3 @ to_byte b4

from_byte bs ≡ (from_byte --
                 from_byte -- from_byte -- from_byte)
                 bs ▷
               (λ(b1, b2, b3, b4).
                   CapRights b1 b2 b3 b4)
```

Type inference and overloading saves us from specifying which `to_byte` and `from_byte` are to be used. We only need to give the structure of the encoding.

After showing that arbitrarily sized machine words are storable, we encoded natural numbers using repeated modulo/division by 255, with 255 itself as the terminator of the stream. Boolean values were stored as byte values of 0 or 1; similarly for the `option` type, but with the encapsulated object following it in the stream. Lists were encoded as their length (a natural number) followed by their contents. Functions can be encoded as long as their domains can be shown to be finite enumerations; this is done by iterating over the domain and encoding only the range. All other components were based upon these primitives.

We found this approach to scale well beyond primitive types; once these were defined, the build-up of all other storable data types and records was swift, and the instantiation proofs automatic.

## 4. Experience

As mentioned above, the overall project goal is not to show that microkernels can be verified in principle. The goal is to show that and how it can be done with time and resources in the order of traditional system implementation (within maybe a factor of 2 or 3). The goal is not to verify a toy implementation or simplified abstraction, but a high-performance, binary compatible version of seL4 that can directly be used on embedded devices such as mobile phones.

The basic philosophy in planning and conducting this project is not much different from conducting a software development project. We are aiming to be pragmatic, to use existing tools as far as possible, to employ automation when possible, and to use systematic methods if not.

Translating Haskell in part manually instead of fully automatically was a pragmatic decision. The cost of manual translation was with about 1 person month significantly below that of implementing the Haskell kernel in the first place. The total development cost (including API design ca. 10 person months) was less than our formalisations in the pilot project had suggested. Change tracking using dependency tags in the source files together with normal version control and shell scripts to pick out changes automatically proved to be effective and again well below the cost of the implementing the changes in Haskell in the first place. The overall cost of creating a fully automated tool would have been significantly higher.

So far, the design and formalisation task has gone smoothly and largely according to expectation.

This process of designing a new OS kernel API and formalising it at the same time was highly interactive and interwoven with many iterations and it has not concluded yet. The translation to Isabelle/HOL started relatively early, when the Haskell API was nearing a first stable point and first user-level binaries could be run through the machine simulator on top of seL4. Already during the translation process, we found and fixed a number of problems, for example an unintentionally unbounded runtime of the IPC send operation. It was discovered because Isabelle demanded termina-

tion proofs for operations that were supposed to execute in constant time.

## 5. Related Work

We have already mentioned work related to the translation of Haskell [9, 14] and the treatment of monads in Isabelle [6, 14, 13, 3, 16] in Sect. 3.

Thompson [25] translates Miranda to Isabelle, but uses first order logic as base and does not treat advanced features like monads or `Dynamic`. Abel et al [1] give an automatic translation of the GHC core language (into which Haskell is transformed for compilation) into the type-theory based Agda system and into first order logic. Some of our manual translations to Isabelle/HOL are similar to those of Haskell to GHC core. Harrison and Kieburtz [11] present P-logic, a program logic for Haskell that focuses on the strict and lazy aspects of the Haskell semantics.

Hallgren et al [10] also produced a microkernel in Haskell. The difference to our work is that they are interested in providing a production kernel in Haskell running directly on the hardware. We are producing a series of design prototypes that are later to result in a high-performance C implementation.

Earlier work on OS verification includes PSOS [22] and UCLA Secure Unix [30]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel with far simpler and less general abstractions than modern microkernels. A number of case studies [7, 4, 29] describe the IPC and scheduling subsystems of microkernels in PROMELA and verify them with the SPIN model checker. Manually constructed, these abstractions are not necessarily sound, and so while useful for discovering concurrency bugs, they cannot provide guarantees of correctness. The VFiasco project [12] is verifying parts of the L4-based Fiasco micro kernel directly on the implementation level without a more abstract specification of its behaviour. The VeriSoft project [8] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS.

Spies [24] uses the high level specification language *FOCUS* to describe the behaviour of an operating system. The difference to our approach is that we use a language that is foremost a programming language and that can being used as such by the kernel design team without expert knowledge in formal methods. It is one contribution of this work to show that this choice does not sacrifice the ability to arrive at an exact formalisation with little effort.

Our approach occupies the middle ground between two extremes: the a priori approach where the kernel is designed formally from the start, and the a posteriori approach where a traditional (C/C++) implementation is created first and formalised later. Both can be found in the literature, e.g. the formal design process of PSOS [22] and implementation verifications such as [7, 4, 29, 12].

In our setting, the a priori approach would design the kernel directly in the theorem prover and extract a program to be used for validation. This requires that the OS designers are intimately familiar with the formal specification language, which they are usually not. They also would be restricted in their use of the language by the executable fragment of HOL, since validation of low-level design decisions is necessary to distinguish between those designs that can possibly be implemented efficiently and those that cannot. This restriction is significant, because even full Isabelle/HOL, while perfectly suited for specification, is not a comfortable programming language yet, certainly not one for rapid development, testing and prototyping of sizeable, low-level, and largely imperative systems.

The a posteriori approach would create a traditional C implementation first. Folklore says and our own experience [28] shows that the effort for formalisation here is significantly higher and correspondence to the prototype much less obvious. Additionally, the effort for implementation is significantly higher as well — we estimate the effort for creating a micro-kernel prototype the traditional way in our OS group to be about 1 person year. This does not include the numerous iterative changes to the API that we went through in our process.

Our approach lies in between. Compared to the a priori method, we enjoy the richness and expressiveness of a full functional programming language and keep the intricacies of formalisation from the OS designers. Compared to the a posteriori method, we arrive at a precise formalisation very quickly and easily. We also significantly speed up development and make an iterative prototyping process possible that in a few months has gone through more API changes than what would otherwise have taken years to implement.

## 6. Conclusion

In this paper, we have shown how formalisation can be included in the creation of an OS microkernel while maintaining OS design concerns as the main driving factors. We are convinced that this approach is amenable to spreading software verification wider into industry use, because it makes it easier to achieve scale, and to achieve formalisations quickly and systematically. Note that the formalisation activity can be carried out by a separate group of people — there is no need to replace an established design team with people who are trained in formal methods. This is an important difference to other techniques such as creating design prototypes in the theorem prover directly.

We have shown how a significant part of Haskell98, including a number of common GHC extensions that occur in practice can be systematically translated into Isabelle/HOL. It was not our aim to provide a complete translation for all language features, as our target language HOL is not suited to this task. The programs that are likely to work well with our method are those that terminate and do not make essential use of laziness. Programs that are likely to be problematic are those that make heavy use of laziness and advanced type system features like multi-parameter type classes. As we have shown, though, some of these problems can be solved, at least for specific applications.

We have received feedback on earlier version of this paper in both directions: on the one hand that it is obvious that one would want to be pragmatic in this way and on the other hand that it will not work in the longer term, because the lack of automation will introduce inconsistencies. With longer experience in tracking change and staying in synchronisation with Haskell over many prototype iterations we can safely disagree with the latter. The former criticism is harder to rebut. It is, of course, obvious that you want to be pragmatic when you know with hindsight that the approach works. The main message of this paper is that it does work, and that it does work well. One can in fact formalise large (5.000 loc literate Haskell), real-world programs at a low cost, even though the translation is not fully automatic, HOL and Haskell do not quite match, and the method we use is incomplete. This fact in our view is far from obvious.

We have additionally created (but not shown here) a more abstract specification that is suitable for proofs on security and invariants on kernel data structures. We are currently proving that it is indeed a formal abstraction of the result of the Haskell translation. Again, starting this process has already helped uncover problems in the Haskell implementation that slipped through code reviews and tests (such as an incomplete test in the revoke-capability operation).

This shows that formalisation and the use of theorem proving tools is beneficial even if full verification is not yet performed or is not even planned. In our setting the formalisation cost so far has been significantly lower than the implementation and testing cost, while the design team did not have to switch to completely new methods or notations.

We currently have a fully working microkernel, implemented in Haskell, running normal ARM binaries through a simulator. We have fully formalised this microkernel in Isabelle/HOL by systematic translation. Next to continuing validation and development of the kernel, this formalisation is the basis for future verification of properties of the system (which we have already started), and a formal refinement of the formalisation down to high-performance C code.

***Acknowledgements*** We thank Jeremy Dawson and Brian Huffman for discussions on their monad formalisations. We are also grateful to Manuel Chakravarty, Michael Norrish, and Kai Engelhardt for reading earlier drafts of this paper.

## References

[1] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *Haskell'05, Tallinn, Estonia*, 2005.

[2] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[3] E. Broch Johnsen and C. Lüth. Theorem reuse by proof term transformation. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *LNCS*, pages 152–167. Springer, Sept. 2004.

[4] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.

[5] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 206–217. ACM, 2006.

[6] J. Dawson. Compound monads and the kleisli category. `http://users.rsise.anu.edu.au/~jeremy/pubs/cmkc/`, 2006. Draft.

[7] G. Duval and J. Julliand. Modelling and verification of the RUBIS $\mu$-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.

[8] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, pages 1–16, Oxford, UK, 2005.

[9] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. High Confidence Software and Systems Conference, HCSS04, `http://www.cse.ogi.edu/~hallgren/Programatica/HCSS04`, 2004.

[10] T. Hallgren, M. P. Jones, R. Leslie, and A. P. Tolmach. A principled approach to operating system construction in haskell. In O. Danvy and B. C. Pierce, editors, *ICFP*, pages 116–128. ACM, 2005.

[11] W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.

[12] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[13] B. Huffman. Constructor classes in Isabelle/HOL. `http://www.csee.ogi.edu/~brianh/`, 2006. Formal Proof Development.

[14] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 147–162. Springer, 2005.

[15] The Iguana operating system. `http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/`, 2006.

[16] E. B. Johnsen, C. Lüth, and M. Bortin. Formal software development with Isabelle. `http://www.tzi.de/~cxl/awe/sfd.pdf`, 2006.

[17] S. P. Jones. Haskell98 language and libraries, revised report, Dec 2002.

[18] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[19] R. Kolanski and G. Klein. Formalising the L4 microkernel API. In B. Jay and J. Gudmundsson, editors, *Computing: The Australasian Theory Symposium (CATS 06)*, volume 51 of *Conferences in Research and Practice in Information Technology*, pages 53–68, Hobart, Australia, Jan. 2006.

[20] J. Liedtke. Towards real $\mu$-kernels. *CACM*, 39(9):70–77, 1996.

[21] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[22] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[23] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technische Universität München, Institut für Informatik, 1999.

[24] K. Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten*. Phdrep, Technische Univerität München, 1998.

[25] S. Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, (7), March 1995.

[26] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In *Proc. NICTA FM Workshop on OS Verification*, pages 73–97. Technical Report 0401005T-1, National ICT Australia, 2004.

[27] H. Tuch and G. Klein. A unified memory model for pointers. In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *LNCS*, pages 474–488, Jamaica, Dec. 2005.

[28] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[29] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.

[30] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

# Automated Verification of UPC Memory Consistency

Øystein Thorsen Charles Wallace

Michigan Technological University

## 1 Introduction

The *UPC* programming language [3] is a shared memory extension to ANSI C. Its memory consistency model is relaxed, allowing for a high degree of optimization, but also permitting behavior which may be surprising to the naïve programmer. To allow better understanding of this memory model, we present a tool [4] for analyzing the behavior of UPC programs. Given an execution trace, the tool determines whether the results are compatible with the UPC memory model. The tool is targeted at newcomers to UPC who want to learn about its memory model and at developers who want to verify possible behaviors of their programs.

This work was inspired by Yang's verification tool [5] for the Intel Itanium memory model. Based on the memory model definition in the UPC specification [3], we have devised a procedure for converting an execution trace into a propositional logic formula that is satisfiable if and only if the execution is compatible with the memory model's notion of consistency. We then use the SAT solver zChaff [6] to determine consistency.

## 2 UPC and its memory model

In UPC, multiple *threads* (computing agents) execute a common program concurrently. In keeping with the shared memory paradigm, threads access shared memory addresses concurrently through standard read and write instructions, rather than through explicit message passing. Program behavior is affected by the order in which memory operations are made visible to threads. By constraining the order of operations, a memory model determines which values may be returned by each read operation. UPC introduces

two *memory modes*, *strict* and *relaxed*. The mode of each shared memory access within a UPC program affects the visibility of those accesses during program execution. As the names suggest, program behavior under strict consistency is more constrained than that under relaxed consistency.

UPC includes support for global synchronization. The upc_barrier statement causes a thread to halt execution until all other threads have executed their corresponding barriers. Once each thread has reached its barrier, all threads may resume execution. From an implementation standpoint, a barrier is composed of two distinct operations. A thread first *notifies* all other threads that it has reached the barrier and then waits until all threads have reached the barrier. UPC also offers a *split-phase* barrier, consisting of the paired statements upc_notify and upc_wait. Separating a barrier into two separate instructions gives the programmer the opportunity to do local computations after notifying but before having to wait.

Every *UPC-consistent* execution is "explainable" in terms of a (global) precedence relation and thread-local precedence relations on the memory accesses in the execution. In this paper, we restrict attention to shared memory accesses. The global relation $<_{strict}$ linearly orders strict operations, including operations by different threads. This serves as a globally agreed-upon sequential order on strict accesses. This ensures sequential consistency for strict consistency mode. Also, for each strict access $S$ by a given thread $t$, $<_{strict}$ orders $S$ after $t$'s preceding relaxed accesses and before $t$'s following relaxed accesses. Thus a strict access serves to globally order certain relaxed accesses before others.

Each local relation $<_t$ is a linear order, on $t$'s operations and all writes and strict operations, that represents thread $t$'s view of the execution. (Notably, $<_t$ does *not* include relaxed reads by threads *other than* $t$.) This relation must agree with $<_{strict}$. Furthermore, it must agree with the observable behavior of the execution (the values returned by reads): each read operation must return the value written by the *most recent* write operation to the given location (a well-defined notion, since $<_t$ is linear).

Fence and barrier statements are defined as null strict operations. The difference between a fence and a barrier is that the latter is *collective*; that is, assumed to be executed by all threads. Hence a fence brings a *single* thread up to date in terms of memory consistency, while a barrier does the same for *all* threads.

## 3 Input and output

The input to the tool includes the "program order" of operations executed by each thread, the values returned by all read operations, and initial values for each memory location. In a valid execution trace, all threads call the same collective operations in the same order, each upc_wait is preceded by a unique upc_notify, and no collective calls are allowed between a upc_notify and a upc_wait.

If the execution trace is UPC-consistent, we want to show a graph displaying the observed ordering for each thread, and if it is not UPC-consistent we want to show why. For this we use a graph drawing package [1] that displays the operations as nodes in a directed graph. Operations are color-coded and organized in columns based on the observing thread. Two operations may have a directed edge between them showing which was observed before the other. Strict operations (including barriers) are displayed in black in a separate column, since they are part of the common strict order for all threads.

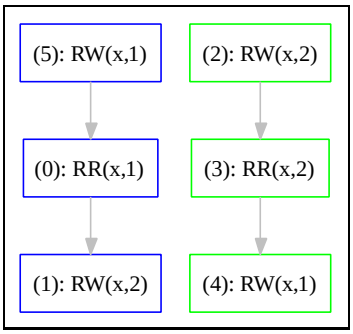
Relaxed write operations have a special status. Each relaxed write is observed by threads other than the thread that issued it; however, threads may observe relaxed writes in different orders. The following example (corresponding to Example 1 in §B5 of the UPC Language Specification) demonstrates this property.

```
startvalues
x = 0

numthreads = 2

thread
RR(x,1)
RW(x,2)

thread
RR(x,2)
RW(x,1)
```




This is an example of UPC-consistent behavior. The tool input is shown in the left-hand box. The memory location x is given the initial value 0. Two threads participate in this execution. One thread performs a relaxed read (RR) of x (returning the value 1), followed by a relaxed write (RW)

to x (assigning the value 2). The other thread performs a relaxed read of x (returning 2), followed by a relaxed write to x (assigning the value 1). The result may be counterintuitive to programmers accustomed to sequential consistency, since no global order of all operations can explain the values read.

One way to conceive of this is to define each relaxed write as a "family" of writes: the "real" write, observed by the thread that issued it, and for every other thread, a "virtual" write that it observes. In an explanatory order $<_t$ for a thread $t$, all of the virtual relaxed writes, issued by other threads, must be included. (In contrast, each relaxed read is only included in the explanatory order of the thread that issued it, so there is no need for virtual copies of relaxed reads.)

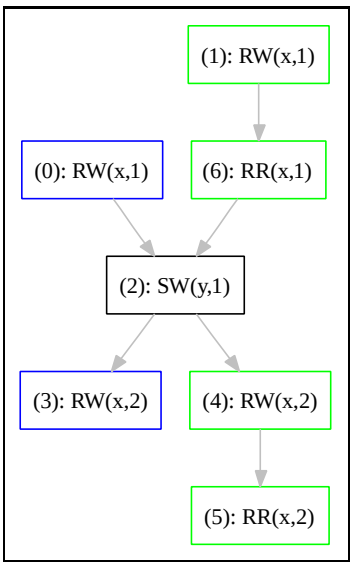
The tool output in the right-hand box shows explanatory orderings for each thread. The operations labeled (1) and (2) are copies of the same relaxed write (similarly, (4) and (5)). The threads order the two relaxed writes differently. Moreover, each thread's explanatory ordering includes only the relaxed reads issued by that thread. Informally speaking, it is not the responsiblity of a thread to explain the relaxed read values of *other* threads.

The following example (corresponding to Example 6 in §B5 of the UPC Language Specification) illustrates the representation of strict operations.

```
numthreads = 2

thread
RW(x,1)
SW(y,1)
RW(x,2)

thread
RR(x,2)
RR(x,1)
```




This is another UPC-consistent example. In the first thread's execution, the presence of the intervening strict write (SW) implies that all threads

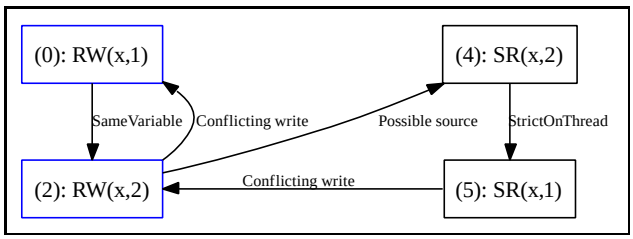
must observe the two relaxed writes in program order (*i.e.,* (0) must precede (3) and (1) must precede (4)). The (potentially counterintuitive) read results are due to the fact that the second thread may observe its own reads out of order, as indicated in the output.

Since the graph represents a temporal order of operations, the presence of a cycle indicates UPC-inconsistency. In the case of UPC consistency, we remove edges implied by transitivity from the displayed graph. For UPC-inconsistent execution traces, we know there must be a cycle preventing this temporal order, so we show all the edges involved in that cycle (or cycles). The following example (corresponding to Example 8 in §B5 of the UPC Language Specification) shows how UPC-inconsistent results are displayed.

```
numthreads = 2

thread
RW(x,1)
RW(x,2)

thread
SR(x,2)
SR(x,1)
```




This behavior is UPC-inconsistent. The relaxed writes must be observed in program order since they operate on the same location, and the reads must be observed in program order since they are strict. The first read result (the value 2) implies a write-read dependency that forms a cycle in the output graph.

# 4 Implementation

Each operation is encoded as a tuple containing all the relevant attributes (*e.g.,* issuing thread, memory mode, memory location, value read/written). The program reads the user input and converts it into a set of tuples, using a bison/flex parser. The tuples are then fed into the SAT generation program. These tuples, along with constraints derived from the memory model, provide the input for generation of the SAT instance.

Clause generation proceeds according to the following high-level characterization of the UPC memory model [2], which is derived from the definition in the UPC specification [3]. We use two predicates: (**order** $i$ $j$) to indicate

that operation $i$ is ordered before operation $j$, and (**reads** $i$ $j$) to indicate that read operation $i$ gets its value from write operation $j$. The predicates **order** and **reads** are each encoded as sets of variables in the SAT encoding.

We begin with constraints on the explaining order. It is irreflexive and transitive. Each strict operation is ordered with respect to every other operation, and the relaxed operations observed by each thread $t$ (as denoted by (**obs** $i = t$)) are totally ordered.

**requireIrreflexiveOrder** *tuples order* $\equiv$
$\forall i \in \mathit{tuples}.\ \neg$(**order** $i$ $i$)

**requireTransitiveOrder** *tuples order* $\equiv$
$\forall i, j, k \in \mathit{tuples}.$ (**order** $i$ $j$ $\wedge$ **order** $j$ $k$) $\Rightarrow$ **order** $i$ $k$

**requireDeterministicOrder** *tuples order* $\equiv$
$\forall t \in$ Threads. $\forall i, j \in \mathit{tuples}.$
((**obs** $i = t$) $\wedge$ (**obs** $j = t$)) $\vee$ (**isStrict** $i$ $\vee$ **isStrict** $j$)
$\Rightarrow$ **order** $i$ $j$ $\vee$ **order** $j$ $i$

Restricting attention to operations by a single thread (*i.e.,* all $i, j$ for which the issuer attributes are equal: (**iss** $i$ = **iss** $j$)), the program order of operations (as implied by the program counter value (**pc** $i$) for each instruction $i$) must be maintained in two cases: conflicting operations (*i.e.,* operations on the same location, where one of the two operations is a write), and cases where one operation is strict.

**conflicting** $i$ $j$ $\equiv$
(**iss** $i$ = **iss** $j$) $\wedge$ (**var** $i$ = **var** $j$) $\wedge$ (**isWrite** $i$ $\vee$ **isWrite** $j$)

**strictOnThreads** $i$ $j$ $\equiv$ (**iss** $i$ = **iss** $j$) $\wedge$ (**isStrict** $i$ $\vee$ **isStrict** $j$)

**dependOnThreads** $i$ $j$ $\equiv$
(**pc** $i <$ **pc** $j$) $\wedge$ (**conflicting** $i$ $j$ $\vee$ **strictOnThreads** $i$ $j$)

**requireProgramOrder** *tuples order* $\equiv$
$\forall i, j \in \mathit{tuples}.$ **dependOnThreads** $i$ $j$ $\Rightarrow$ **order** $i$ $j$

The definition of split-phase barrier implies that for a given notify-wait phase, notify operations across all threads must complete before any waits may complete. Restricting attention to a single thread, this is implied by

**requireProgramOrder**, since notify and wait operations are strict and hence program order must be maintained. To enforce the ordering across different threads, we need a further constraint. (Here, (**src** $i$ = **src** $j$) means that the notify $i$ and wait $j$ are paired together in the same phase.)

**collectiveOrder** *tuples order* $\equiv$
$\forall i, j \in$ *tuples*.
$(\textbf{op}\ i = \textbf{notify}) \ \wedge \ (\textbf{op}\ j = \textbf{wait}) \ \wedge \ (\textbf{src}\ i = \textbf{src}\ j) \Rightarrow \textbf{order}\ i\ j$

Next, we specify the conditions on read operations. The value returned by each read must be explainable in terms of the value assigned by some write. We say that a read operation $k$ *reads from* a write operation $i$ if the locations and values of the operations match, and there is no intervening write that overwrites the value written by $i$.

To ensure that every read operation $k$ has an explaining write, we add a clause which is a disjunction of the form (**reads** $k\ i \ \vee$ **reads** $k\ i' \ \vee \ \ldots$), where $i, i', \ldots$ are writes in the trace that match $k$ in location and value attributes. This is satisfied if and only if (**reads** $k\ i$) is true for some $i$.

If $k$ reads from $i$, then $i$ must precede $k$.

**requireReadOrder** *tuples order* $\equiv$
$\forall i, k \in$ *tuples*.
$\textbf{isRead}\ k \ \wedge \ \textbf{isWrite}\ i \ \wedge \ (\textbf{var}\ i = \textbf{var}\ k) \ \wedge \ (\textbf{val}\ i = \textbf{val}\ k)$
$\Rightarrow \neg\textbf{reads}\ k\ i \ \vee \ \textbf{order}\ i\ k$

Furthermore, if $k$ reads from $i$, any write $j$ that would obliterate the value of $i$ must be ordered either before $i$ or after $k$.

**requireReadValue** *tuples order* $\equiv$
$\forall\ t \in$ Threads. $\forall i, j, k \in$ (**observed** *tuples* $t$).
$\textbf{isRead}\ k \ \wedge$
$\textbf{isWrite}\ i \ \wedge \ (\textbf{var}\ i = \textbf{var}\ k) \ \wedge \ (\textbf{val}\ i = \textbf{val}\ k) \ \wedge$
$\textbf{isWrite}\ j \ \wedge \ (\textbf{var}\ j = \textbf{var}\ k) \ \wedge \ (\textbf{val}\ j \neq \textbf{val}\ i)$
$\Rightarrow \neg\textbf{reads}\ k\ i \ \ \vee \ \textbf{order}\ j\ i \ \vee \ \textbf{order}\ k\ j$

When all the clauses are generated the SAT solver tries to find a satisfying variable assignment [7]. If a solution is found, the program prints a graph showing a possible ordering. If a solution is not found, we extract the cycle from the SAT encoding and output a graph showing the cycle.

## 5 Future Work

Currently, the tool operates at a level of abstraction that is rather removed from that of UPC code. This is in keeping with the memory model definition in the UPC specification. Our primary goal so far has been faithful adherence to the official specification. However, this requires a manual conversion from actual UPC source code to an idealized execution trace. For programs of significant size, this process is unwieldy. Thus an imperative for us is to derive execution traces automatically from UPC code. We feel that this enhancement will make the application a feasible tool both for developers and students.

## References


[1] GraphViz home page. `http://www.graphviz.org`.

[2] Øystein Thorsen. Automated verification of UPC memory consistency. Master's thesis, Michigan Technological University, 2006. Available at `http://www.upc.mtu.edu/papers/UPCVerifier.pdf`.

[3] The UPC Consortium. UPC language specifications v.1.2, May 2005.

[4] UPCVerifier software. `http://www.upc.mtu.edu/applications/UPCVerifier.html`.

[5] Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, University of Utah, 2004.

[6] zChaff home page. `http://www.princeton.edu/.chaff/zchaff.html`.

[7] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 582–595, 2002.

# Automated Model-based Verification of Object-Oriented Code

Jonathan Ostroff, Chen-Wei (Jackie) Wang, Eric Kerfoot and Faraz Ahmadi Torshizi *
Department of Computer Science and Engineering, York University,
4700 Keele St., Toronto, ON M3J 1P3, Canada.

**Abstract**

ESpec is a suite of tools that facilitates the testing and verification of object-oriented Eiffel programs in an integrated environment. The suite includes unit testing tools (ES-Test) and Fit tables (ES-Fit for customer requirements) that report contract failures. This paper describes ES-Verify (part of ESpec) for automatically verifying a significant subset of Eiffel constructs written with a value semantics. The tool includes a mathematical model library (sequences, sets, bags and maps) for writing high-level specifications, and a translator that converts the Eiffel code into the language used by the Perfect Developer (PD) theorem prover. Preliminary experience indicates that the vast majority of verification conditions are quickly and automatically discharged, including loop variants and invariants. ES-Verify is the first automated Eiffel verification tool and allows the developer to use the clean syntax and object-oriented structures of Eiffel, together with its mature industrial-strength design by contract (DbC) mechanism.

## 1 Introduction

A software product is reliable if it is correct (performs its tasks according to specification) and robust (reacts appropriately to abnormal conditions). How should specifications be provided and how do we check that software behaves according to its specification? Design by Contract (DbC) is a promising method for answering these questions. A class can be specified via expressive pre-conditions, post-conditions and class invariants [19].

A variety of object-oriented languages have followed this contracting approach to software quality such as Eiffel [19], Spec# [4, 3], JML [17] tools like ESC/Java2 [10, 7], and UML/OCL [5]. A "lightweight" formal approach to checking the correctness of code works by runtime assertion checking, i.e. the contracts are checked as the code is executed and an exception is raised if there is a contract violation. However, we would also like to reason formally about the correctness of programs and to mechanize such process. Automated verification of object-oriented code has been pursued in systems such as Spec# and JML tools like ESC/Java2.

ESpec (Eiffel Specification) toolset is a unified environment allowing software developers to combine Fit tables (ES-Fit for customer requirements and acceptance tests) with contracts and unit testing tools (ES-Test). This means that a single integrated tool can be used to specify, develop, test, and verify the requirements and design of a software product. Formal verification is a substantial addition to the capabilities of the ESpec toolset, allowing for a combination of lightweight validation and automated deductive verification.

In this paper we describe the automated model-based verification for a significant subset of Eiffel. The following three components, which together we call the ES-Verify, are under development as part of the ESpec suite:

*Email: {jonathan, faraz}@cs.yorku.ca. Eric.Kerfoot@comlab.ox.ac.uk. Supported by a grant from NSERC.

- An Eiffel Model Library (ML) for specifying the abstract state of a program without exposing its implementation details. This library is similar to the model-based specifications as in B [1] and Z [20], except that it is object-oriented. ML contains classes such as ML_SEQ, ML_SET, ML_BAG, and ML_MAP. These classes are both immutable and executable. They are immutable so that software properties specified in the pre- and post- conditions as well as the class invariants can be based on them. They are executable so that contract violations will be reported (if any). This mathematical library is thus useful for lightweight verification even in the absence of a theorem prover.

- An Eiffel base library (ES_BASE) of data structures (classes such as ESV_ARRAY, ESV_LIST, ESV_SET, and ESV_TABLE) for the efficient implementation of software products. The prefix "ESV" stands for an "ESpec Value" structure, which is part of the ESpec base library (built on top of the Eiffel base library via inheritance) for implementing code. These ESV classes apply a value semantics [12], but for efficiency they are mutable. While class features are contracted via ML (which are executable but inefficient due to their mathematical immutability), their bodies are implemented via the ES_BASE classes (which are mutable and hence efficient, but not as suitable for specifications as ML ones).

- A translator that will convert Eiffel code implemented via ES_BASE and specified via ML into an equivalence written in a specification language Perfect [14]. The advantage of this translator is that there is, associated with the Perfect language, a fully-automated reasoning tool - Perfect Developer (PD) - that fits well for our source Eiffel code. PD supports object-oriented, model-driven, and DbC software development as well as its verification [11]. PD converts its specification (written in the Perfect Language) into complete verification conditions and attempts to automatically discharge their proofs.

As stated, ES-Verify uses the PD tools (the Perfect language and its associated theorem prover). Although we are impressed by the expressiveness and power of the PD tools, we have not used them in the intended fashion. The intended use of PD tools is that developers write their specifications in the Perfect Language, which is then used to automatically generate executable code (e.g. Java or C++). In this respect, Perfect is akin to model-driven development (MDD) methods. Perfect also has a notion of refinement that can be used to improve the efficiency of the generated code.

We have examined the Java code and found that the generated code - much longer and more complex than the original contract-based specification - is not intended to be read. The MDD approach is useful if there is never a need to deal with the generated code. However, Perfect specifications are neither directly executable nor is there a debugger at the model level. As a result, our preference is to write code in Eiffel. Eiffel has a mature industrial-strength contracting mechanism with a full set of tools such as debuggers, profilers, documentation, and browsing capabilities. The language is admired for its clear syntax and expressive use of a full range of object-oriented constructs such as multiple inheritance.

Our approach is to write the code in Eiffel and thus retaining the simple but expressive use of its language constructs. The Eiffel code is then translated into Perfect using (a) the Perfect refinement constructs for Eiffel feature implementations and (b) the Perfect contracting mechanism for Eiffel contracts. The Eiffel model library (ML) was designed in order to avoid mismatches between itself and the Perfect data structures. Theorem proving program involving genericity and loops (with their invariants) is a non-trivial task, and this work shows that model libraries (such as ML) must be designed with the target theorem prover in mind. In the sequel we will use the abbreviation PD for the combination the Perfect specification language and its associated theorem prover.

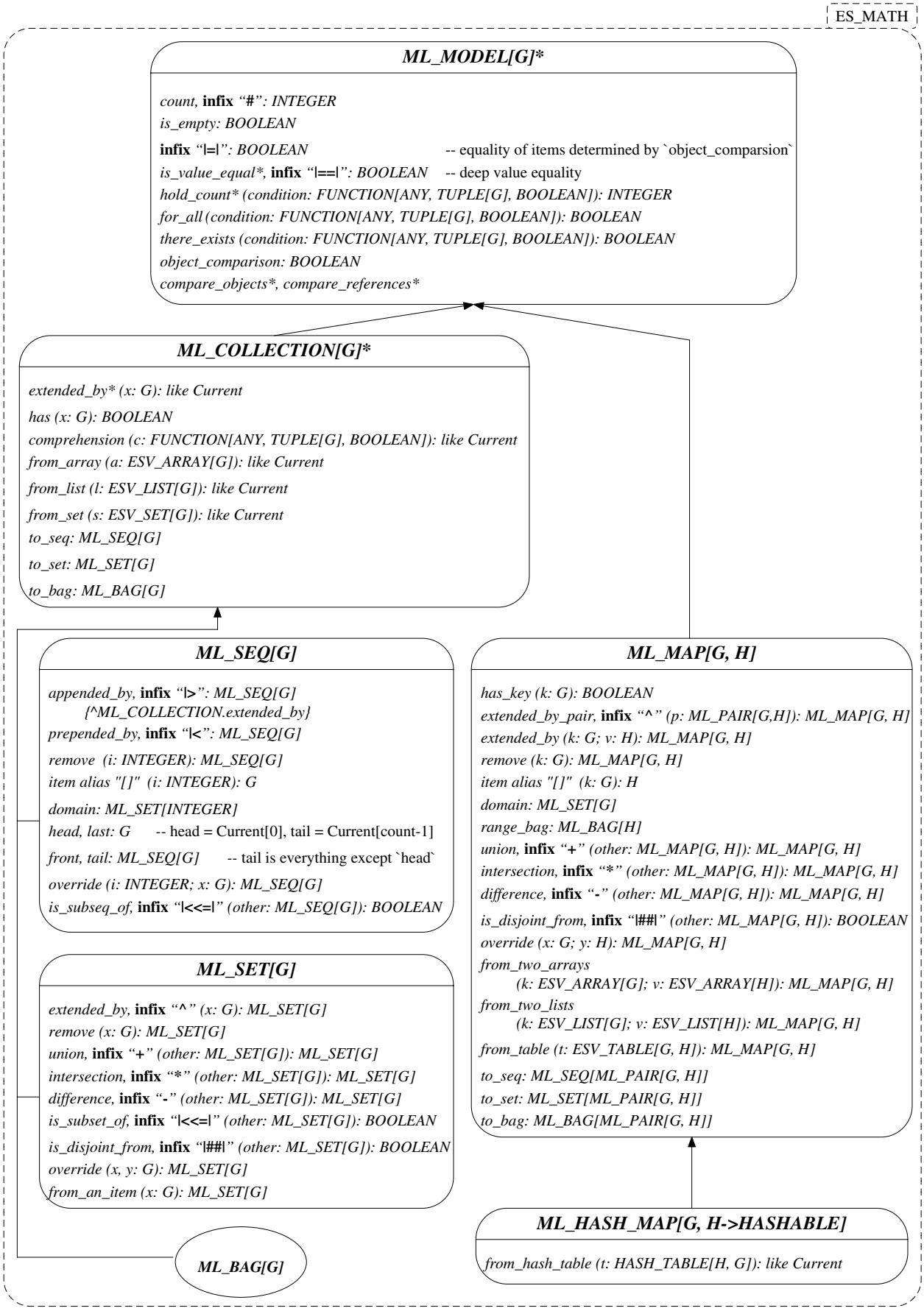
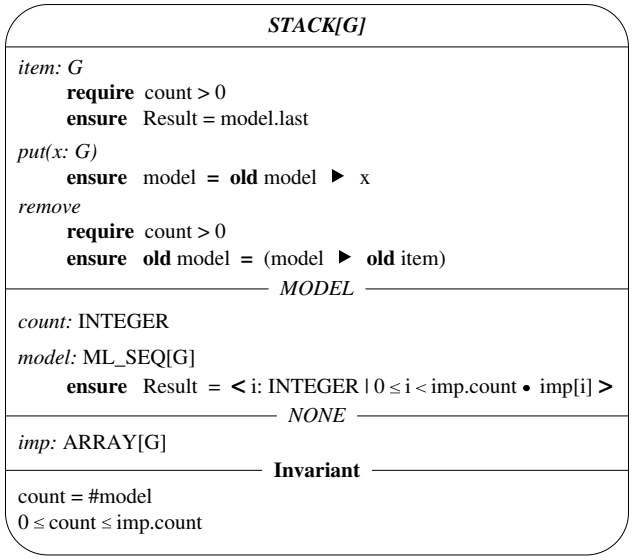



Figure 1: Core Classes in the Mathematical Library (ML) for Model-based Specification




(a) BON Diagram of STACK

```
class STACK[G] feature

    put (x: G) is
        do
            imp.force (x, imp.count)
        ensure
            model |=| old model |> x
        end
        ...
end
```

(b) *put* feature of STACK

```
class STACK_PROPERTIES[G] feature

    lifo (s: STACK[G] ; x: G) is
        require
            s /= void
        do
            s.put (x)
            s.remove
        ensure
            s.model |=| old s.model
        end
        ...
end
```

(c) Stack LIFO property

Figure 2: STACK[G] modelled by ML_SEQ[G]

# 2 Models via ML

As explained in [20] with reference to Z, formal specifications use mathematical notation to describe, in a precise way, the properties which a software product must have, without unduly constraining the way in which these properties are achieved. We may call the mathematical description an abstract *model* of the system under development. The model describes *what* the system must do without saying *how* it is to be done. Models allow questions about what the system does to be answered confidently, without the need to either disentangle the information from a mass of detailed program code, or speculate about the meaning of phrases in an imprecisely-worded prose description.

In Z, the mathematical models are based on predicate logic and the set theory, and thus obey a rich collection of mathematical laws which makes it possible to effectively reason about the way a specified system will behave. But these models are not oriented towards computer representation.

The model library (ML) described in this paper encodes predicate logic acting on sets, sequences, bags, and maps (as in Z), but the mathematical theories are structured as classes (instantiated to immutable objects needed for mathematical specification) whose features (e.g. $\forall, \exists, \in$, set comprehension, etc.) are pure functions executable in the object-oriented style. The Eiffel agent mechanism for iteratively applying a supplied expression to a collection is much used.

The classes of ML are shown in Fig. 1. Contracts may be specified using ML and these contracts are executable. When runtime assertion checking is turned on, contract violations (if any) are signalled via exceptions, thus indicating an inconsistency between the implementation and its specification. The complete specification of a system and its implementation can be provided in the same compilable and executable Eiffel text (e.g. see class `STACK[G]` in Fig. 4). The immutable ML classes will be inefficient (due to its re-construction of a new ML object every time a feature such as `appended_by` is invoked), by comparison to the mutable classes in the Eiffel or ES base library (such as `ARRAY` and `LIST`). But this is acceptable as contract checking may be turned off in the final delivered code which will only use the efficient base library for implementation.

As a simple example, consider the BON [22] contract view of a generic stack as shown in Fig. 2a.

The model of the stack consists of a `ML_SEQ[G]` (i.e. a sequence of items of type `G`, where `G` is a generic parameter) and *count* (the number of items in the stack). The contracts of all the other features of the stack can be described in terms of the sequence and *count*. In the absence of a sequence to model the stack (i.e. with just the model attribute *count*), the best post-condition for the stack push operation `put` is

$$count = \textbf{old}\ count + 1 \wedge item = x \tag{1}$$

However, such abstract specification violates Einstein's maxim to "make everything as simple as possible, but not simpler" because it is incomplete. For example, an implementor can satisfy the above specification yet change old values of the stack that are not at the top. Therefore, we need a frame condition that says the old part of the stack remains unchanged. By adding a sequence to the model we can now express the complete contract as

$$model = \textbf{old}\ model \blacktriangleright x \tag{2}$$

where $\blacktriangleright$ is the `appended_by` (pure) function of a mathematical sequence that returns a new sequence same as the old one, but with the argument item appended to the end. Since (2) $\Rightarrow$ (1), there is then no need to write (1) as it is entailed by the model post-condition. With the full model we can then provide the complete contracts for the pop operation `remove` and the query `item` that returns the top of the stack. The Eiffel notation follows the BON notation quite closely as shown in Fig. 2b. For $\blacktriangleright$, we may use either the `appended_by` function or alternatively the infix operator `|>` as shown in class `ML_SEQ` in Fig. 1.

Model classes such as `ML_SEQ` hold items that may be stored either by reference or by value. Eiffel has the **expanded** construct for constructing a value semantics. We thus introduce the notion of model equality (infix operator `|=|`) which depends on what type of comparison is requested (see `ML_MODEL` in Fig. 1). The default is that two model sequences (say $s1$ and $s2$) are compared for their stored items via reference equality (i.e. $s1$ |=| $s2$ iff the two sequences have the same size and the items stored at each index both refer to the same object). A specifier may invoke feature `compare_objects` (see `ML_MODEL`), in which case the items stored at each index will be compared based on how the inherited feature `is_equal` (of the actual generic type `G`) is defined [1].

With our contracts complete, and even in the absence of implementation details, we may already begin to validate our specification based only on the model. For example, the last-in-first-out (LIFO) property of the stack can be specified as shown in Fig. 2c. In the absence of implementation, we cannot execute or unit test the LIFO property. However, with the translator and theorem prover, the LIFO property will prove with a warning that the body of `put` and `remove` must be refined with an implementation.

We must now refine the specification to an efficient implementation. We choose mutable structures such as an array or linked list. We may use `ARRAY` from the Eiffel base library, or from the ES base library if a value semantics rather than a reference semantics is desired (i.e. by declaring `imp:ESV_ARRAY[G]`).

Next we need to define the abstraction relation between the abstract space in which the abstract program is written (i.e. `model`) and the space of the concrete representation (i.e. `imp`). This can be accomplished by giving an abstraction function which maps the concrete variables into the abstract objects which they represent. We may do this as follows. The body of the query `model` (a `ML_SEQ[G]`) for the stack in Fig. 2 could be a loop that iterates through the implementation array and returns an equivalent sequence with the same elements as the array. That is, we "lift" the mutable array into a mathematical immutable sequence. The abstraction function [16] is captured by the post-condition of query `model` as follows:

[1] `is_equal` in Eiffel is similar to `equals` in Java

$$Result = \langle i : INTEGER \mid 0 \leq i < imp.count \bullet imp[i] \rangle \quad (3)$$

where the angle brackets $\langle\,\rangle$ stand for sequence comprehension in the same way that $\{\,\}$ stands for set comprehension. For example, $\{i : INT \mid 0 \leq i \leq 2 \bullet i + 1\} = \{1, 2, 3\}$. Set, bag, sequence or map comprehension presents expressive notation for abstraction functions and is supported in ML. The Eiffel ML library uses the agent construct for writing comprehension (see Fig. 1). However, for the post-condition of `model` we may use one of the pre-defined ML functions `from_array` that "lifts" an efficient mutable array to a mathematical sequence. Function `from_array` returns a new sequence whose items refer to the same items as in the array `imp` between $0 \cdots count - 1$. So the post-condition (3) writen in ML becomes:

```
Result |=| Result.from_array(imp.subarray(0,count-1))
```

which asserts that the resulting sequence returned by the model is model-equal to the implementation array treated as a sequence. The contracts of all other features remain the same as they are all described in terms of `model`.

## 2.1 The Birthday Book example – ML specifications and loop invariants

The author of [21] reports that a web-enabled database system, consisting of 35,799 lines of Perfect, generated 9810 proof obligations and proved automatically in 4.5 hours (1.6 seconds per proof) on a modest laptop. We believe that the above performance is sustainable for reasonable chunks of code but there is minimal refinement and PD does the code generation. However, in our case there is refinement from high level models to more complex constructs (e.g. loops their variants and invariants), and thus the demands on PD are much greater. Nevertheless, by means of careful matching between ML and PD data structures as well as tuning of the translator, we can achieve proofs of the vast majority (if not all) verification conditions.

The birthday book example [20] nicely illustrates refinement to loops and more intensive use of ML as shown by the BON diagram in Fig. 3a.

The model for the birthday book is a combination of the number of name-and-date pairs stored (i.e. `count`) and a `ML_MAP[NAME, DATE]` (i.e. a set of name-and-date pairs). Alternatively, this map is a function whose domain is a set of names and whose range is a bag of dates. The features of the birthday book include the ability to add a new pair (e.g. $[Peter, (March\ 1)]$), find a birthday given a name, and a `remind` function that for a given date $d$ returns the set of names whose birthday is on $d$.

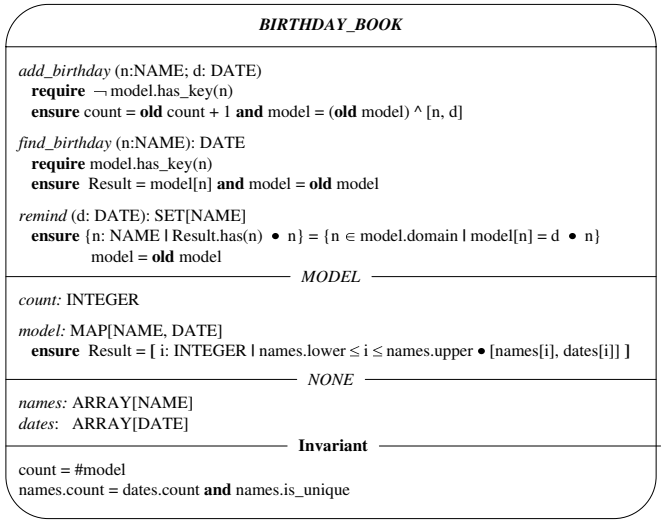
The `remind` function returns a set of names (`SET[NAME]`) where `SET` is an efficient mutable structure from either the Eiffel or ES base library. The birthday book is implemented as two arrays: one for names and the other for dates. The post-condition of the `remind` query is

$$\{n : NAME \mid Result.has(n) \bullet n\} = \{n \in model.domain \mid model[n] = d \bullet n\} \quad (4)$$

where the RHS expression means the set of all names, from the domain of the model map, whose birthday is on the provided date $d$. And this must be equal to the LHS expression which represents the set of all names returned by the `remind` function. The Eiffel notation for the `remind` function is shown in Fig. 3b. The Eiffel post-condition of the `remind` query in (4) now becomes:

```
model_set.from_set(Result) |=| model.comprehension(agent date_matches (?, ?, d)).domain
```

The agent function used in the post-condition (and loop invariant) of the `remind` query is:




(a) BON Diagram of BIRTHDAY_BOOK

```
class BRITHDAY_BOOK feature

 remind (n: NAME ; d: DATE): SET[NAME] is
  local
   i : INTEGER
  do
   create Result.make
   from
    i := dates.lower
   invariant
    pd_modify ("i, Result")
    i >= 0 and then i <= names.count
    i < names.count implies names.valid_index (i)
    inv: -- see text
   variant
    dates.count - i
   until
    i = dates.count
   loop
    if dates.item (i).is_equal (today) then
     Result.extend (names[i])
    end
    i := i + 1
   end
  ensure
   model_set.from_set (Result) |=|
   model.comprehension (agent date_matches (?, ?, d)).domain
  end
  ...
end
```

(b) *remind* feature of BIRTHDAY_BOOK

Figure 3: Birthday Book

```
date_matches (x: NAME; y, date: DATE): BOOLEAN is
    do
        if y.is_equal (date) then
            Result := true
        end
    end
```

By defining a slice of the model map, according to the current loop counter $i$ as well as arrays *names* and *dates*, as follows:

$$\text{mSlice}(i, names, dates) \;\widehat{=}\; \langle\!\langle j : INTEGER \mid 0 \leq j < i \bullet [names[j], dates[j]] \rangle\!\rangle \tag{5}$$

we can show that the loop invariant for the `remind` query has been constructed to approximate and hence similar to its post-condition:

$$\{n : NAME \mid Result.has(n) \bullet n\} = \{n \in \text{mSlice}(i, names, dates).domain \mid model[n] = d \bullet n\} \tag{6}$$

And the equivalent Eiffel loop invariant (`inv` in Fig. 3b) is:

```
model_set.from_set (Result) |=|
model.from_two_arrays(names.subarray (0, i-1),dates.subarray(0, i-1)).
    comprehension(agent date_matches (?, ?, today)).domain
```

# 3 The Eiffel to PD Translator

### Underlying Theorem Prover

Our goal is to automatically verify Eiffel code specified via ML as in the stack and birthday book examples. The question would be, which theorem prover do we use? The *Perfect Developer* (PD)

specification language and theorem prover [12] is a technically mature product that is aligned with the object-orientation and design by contract paradigms. PD theorem prover has about the same level of power and automation as *Simplify* [13] that is used for static verification in Spec# and ESC/Java2. *Simplify* handles integers and booleans at the primitive level while PD has a greater repertoire (e.g. reals, characters, and strings). PD specification language also has a library of generic sequences, sets, bags, and maps well-suited to ML [14]. A limitation of PD is that it discourages reference semantics [12]. It is well-known that the presence of multiple references to a common object causes aliasing and makes sound and complete static verification problematic. Therefore, PD, unlike say Java and Eiffel, adopts a value semantics by default and discourages the use of reference semantics [2]. Despite these limitations, we have adopted PD for automated deduction in our ES-Verify tool, and we are in the process of constructing a library of base Eiffel classes with a value semantics (see Introduction) using the Eiffel **expanded** construct. As a future goal we have to expand our tool to handle verification of reference aliasing and inheritance.

The theoretical foundations of PD are Floyd-Hoare logic and Dijkstra's weakest pre-condition calculus and it has the power of first-order predicate calculus, as well as a few higher-order constructs [11]. The prover generates verification conditions and aims for verifying the total correctness (termination and refinement satisfying specification) of the input code. It delivers either a proof, upon success in discharging all verification conditions, or otherwise a list of warnings, possibly accompanied by useful fix suggestions. Output from the prover can be in formats such as HTML or Tex. From an academic point of view, there is a lack of information about the inner workings of the PD theorem prover (as opposed to an interactive theorem-proving system such as *Isabelle* [5]). Ideally, the logical rules used in correctness proofs should be open for inspection so that independent trust can be established. However, the PD theorem prover does provide the complete proof, and thus the product is robust and suitable for engineering use [15].

### Outline of Class Translation

Fig. 4 shows how the Eiffel generic stack example is translated into its equivalent PD specifications. The translator assumes that all Eiffel classes to be translated have already been compiled and type-checked. On the Eiffel side (left of Fig. 4), there are three different **feature** declarations: the *public* feature declaration[3], the *model* feature declaration[4], and the *implementation* feature declaration[5]. And on the PD side, there are three corresponding sections: **abstract**, **internal** and **interface**.

We first consider the Eiffel *public* feature declaration. Each Eiffel public attribute (e.g. `count`) becomes a variable (i.e. **var** declaration) in the PD abstract section. In order to allow client classes to access this variable, it must also be redeclared as a function in the PD interface section (hence the first line in the PD interface section reads `function` count). Each Eiffel public command (e.g. `put`) and public query (e.g. `item`) become a **schema** and a **function** in the PD interface section, respectively.

We then consider the Eiffel *model* feature declaration. In stack we only have the query `model`, but in general we may have attributes and queries (but no commands) in this declaration. Each Eiffel model attribute becomes a variable in the PD abstract section. Each Eiffel model query (which is essentially the abstraction function), not only becomes a variable in the PD abstract section, but also becomes two functions in the PD internal section. The first PD function uses the same name as the Eiffel model query and its definition (expression following symbols ^ =,

[2]In PD, if a reference semantics is adopted, then, roughly speaking, a `heap` declaration, e.g. `heap` MyHeap, would be required. Although we have several simple PD examples on basic aliasing effect, we have not yet experienced much the power of the prover on handling reference semantics. Escher Technologies Ltd. is in the process of developing a new beta intending to properly handle the issue.

[3]The part under the label **feature**{ANY}.

[4]The part under the label **feature**{ML_MODEL, ANY}.

[5]The part under the label **feature**{ML_MODEL}.

i.e. is-defined-as) corresponds to the translated post-condition[6] of the that query. The second PD function is a twin function with a `_verification` name suffix. This twin function has the same definition but with a refinement (`via...end` segment) underneath which is the translated body of the Eiffel model query. This twin function is needed because future versions of PD will disallow refinement/implementations of abstraction functions. Since we desire to verify that the `model` implementation satisfies its post-condition, we need this twin function. In stack the Eiffel query `model` becomes (a) a variable `model` in the PD abstract section, and (b) a function `model` and its twin refined function `model_verification` in the PD internal section.

Now we consider the Eiffel *implementation* feature declaration. All features under this declaration appear in the PD internal section in the obvious way, i.e. Eiffel attributes become PD variables, Eiffel queries become PD functions, and Eiffel commands become PD schemas. Moreover, since Eiffel agent expressions in loop invariants are private, they should be declared in this feature declaration; however, agent expressions in pre-/post-conditions may be declared in either the public or model feature declaration part for access from clients. One such example is the agent function `date_matches` occurring in the loop invariant and post-condition of the `remind` feature in birthday book.

Finally we consider the Eiffel class invariants. Those clauses that only refer to public or model attributes become equivalent invariants in the PD abstract section; otherwise, they become equivalent invariants in the PD internal section.

**Outline of Routine Translation**:

As stated, Eiffel commands and queries become PD schemas and functions, respectively. For an Eiffel command that may modify the current object, frame constraints are needed. In order to specify frame constraints, PD supports a **change** clause[7]. For translation into PD, we use in Eiffel specification a `pd_modify`[8] declaration with its string argument passed as a list of attributes that the PD schema may change. For an Eiffel command or query, its require clause (for pre- condition) and ensure clause (for post-condition) appear as equivalent PD **pre** and **satisfy** clauses, respectively. For Eiffel command, its ensure clause (with its pd_modify declaration) appears as the equivalent PD change and satisfy clauses under a **post** declaration. For Eiffel query, it is translated in the same way as it for a command except there is no pd_modify declaration in its post-condition, and thus there exists no change list and post declaration for its translation in PD. Moreover, the Eiffel **old** notation for the value of expressions in a pre-state is converted into the equivalent PD primed notation. Finally, the body of an Eiffel command or query appears as an equivalent PD **via ... end** refinement segment.

# 4 Comparison with other tools

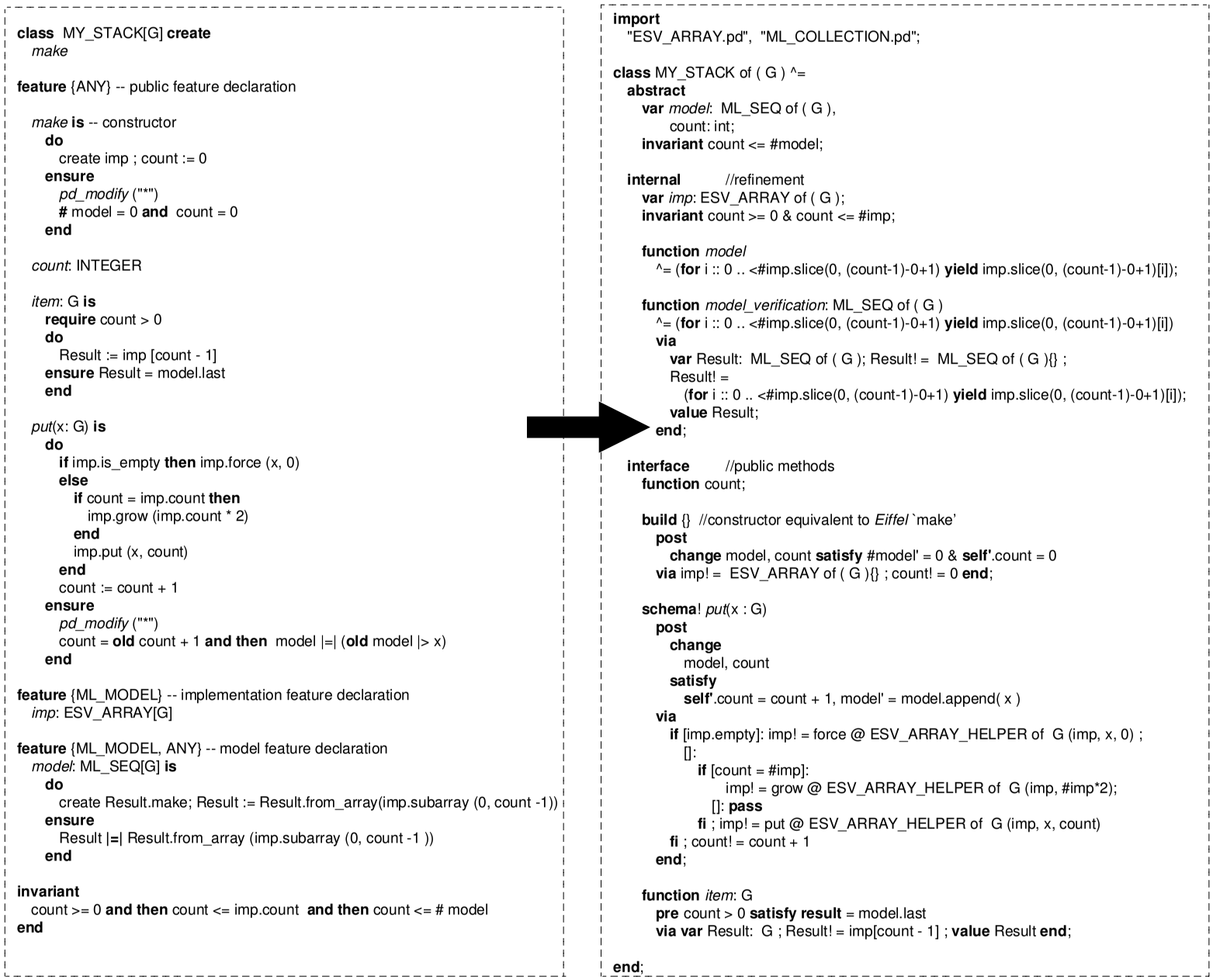
We compare ES-Verify with the other two similar software verification tools: ESC/Java2 and Spec#.

Tools like ESC/Java2 and Spec# allow the developer to increase the confidence of already existing Java and C# code following an Annotated Development approach by adding specifications as annotations [12]. Spark Ada [2] is a successful example of Annotated Development, but the specifications are usually only partial (in particular, expressing data refinement is difficult)[12]. When applied to an object-oriented language that uses reference semantics or makes heavy use of pointers, correctness has to be sacrificed in order to allow more potential bugs to be spotted, otherwise very little can be verified. Spark Ada instead preserves correctness by subsetting the Ada language. This is the strategy that ES-Verify has assumed where references and inheritance are for now not used.

[6] More precisely, RHS of the first post-condition clause which has a matching type with it of that query.

[7] The new ECMA specification for Eiffel has a somewhat equivalent **only** clause.

[8] A boolean function that takes as argument a string and always returns true, and thus can always pass the run-time contract checking. Expression `pd_modify("*")` is an abbreviation meaning all attributes may change.



```
class  MY_STACK[G] create
   make

feature {ANY} -- public feature declaration

   make is -- constructor
      do
         create imp ; count := 0
      ensure
         pd_modify ("*")
         # model = 0 and  count = 0
      end

   count: INTEGER

   item: G is
      require count > 0
      do
         Result := imp [count - 1]
      ensure Result = model.last
      end

   put(x: G) is
      do
         if imp.is_empty then imp.force (x, 0)
         else
            if count = imp.count then
               imp.grow (imp.count * 2)
            end
            imp.put (x, count)
         end
         count := count + 1
      ensure
         pd_modify ("*")
         count = old count + 1 and then  model |=| (old model |> x)
      end

feature {ML_MODEL} -- implementation feature declaration
   imp: ESV_ARRAY[G]

feature {ML_MODEL, ANY} -- model feature declaration
   model: ML_SEQ[G] is
      do
         create Result.make; Result := Result.from_array(imp.subarray (0, count -1))
      ensure
         Result |=| Result.from_array (imp.subarray (0, count -1 ))
      end

invariant
   count >= 0 and then count <= imp.count  and then count <= # model
end
```

```
import
   "ESV_ARRAY.pd",  "ML_COLLECTION.pd";

class MY_STACK of ( G ) ^=
   abstract
      var model:  ML_SEQ of ( G ),
            count: int;
      invariant count <= #model;

   internal          //refinement
      var imp: ESV_ARRAY of ( G );
      invariant count >= 0 & count <= #imp;

      function model
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);

      function model_verification: ML_SEQ of ( G )
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i])
         via
            var Result:  ML_SEQ of ( G ); Result! =  ML_SEQ of ( G ){} ;
            Result! =
               (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);
            value Result;
         end;

   interface         //public methods
      function count;

      build {}  //constructor equivalent to Eiffel `make'
         post
            change model, count satisfy #model' = 0 & self'.count = 0
         via imp! =  ESV_ARRAY of ( G ){} ; count! = 0 end;

      schema! put(x : G)
         post
            change
               model, count
            satisfy
               self'.count = count + 1, model' = model.append( x )
         via
            if [imp.empty]: imp! = force @ ESV_ARRAY_HELPER of  G (imp, x, 0) ;
               []:
                  if [count = #imp]:
                        imp! = grow @ ESV_ARRAY_HELPER of  G (imp, #imp*2);
                     []: pass
                  fi ; imp! = put @ ESV_ARRAY_HELPER of  G (imp, x, count)
            fi ; count! = count + 1
         end;

      function item: G
         pre count > 0 satisfy result = model.last
         via var Result:  G ; Result! = imp[count - 1] ; value Result end;

end;
```

Figure 4: STACK example: The Translation Layout from Eiffel into Perfect Language

ES-Verify takes advantage of the built-in Eiffel DbC constructs, enabling it to both immediately use the run-time debugging as well as the ES-Test tool and formalize these specifications by translating them into PD code. JML tools like ESC/Java2 and `jmlc` [8] exist separately to support static verification and runtime assertion checking. The current release of ESC/Java2 claims it now runs in the JML runtime assertion checker. The Spec# system supports both static verification and runtime assertion checking in Microsoft Visual Studio. That is, the code accompanied with its specifications in ESC/Java2, Spec#, and ES-Verify are immediately executable. However, the goal of ESC/Java2, Spec#, and ES-Verify is to find bugs rather than prove total correctness. An interesting property of these tools is that they neither warn about *all* errors nor do they warn *only* about actual errors [6] and may raise false alarms due to the nature of logical proof.

The PD theorem prover has approximately the same level of proving power as the B theorem prover. It is capable of dealing with all the primitive types including reals, quantification, and set theory. Also, PD is used to verify itself with about 130,000 verification conditions. In proving data-intensive applications like the Birthday Book example (with loop invariants and model contracts in both pre- and post- conditions), PD has been able to discharge all the verification conditions. When we attempted to code the Birthday Book example in Spec#, with the same set of model specifications, it did not even translate the code into the intermediate one and start verifying. A number of specifications had to be removed in order to make it verify. Unlike ESC/Java2 and Spec#, ES-Verify can reason about generic classes. The updated version of ESC/Java2 is compatible with Java version 1.4, but not 1.5 which includes generic types. Spec# compiles generic types but did not yet verify them at the time we attempted to code the Stack example in it.

ES-Verify follows an Abstract Specification and Refinement approach as in the B-method. This approach requires a notation that can adequately express both an abstract specification and an implementation. Data refinement is a key feature, i.e. you develop the specification using an abstract data model (in our case ML), then refine the data model if necessary with an efficient implementation. The notation and the semantics are designed for correctness and provability unlike traditional programming languages, so there is no need to sacrifice correctness. Our declared model, if not deferred, is defined in terms of other fields, and thus is used as a model field not accessible to clients. ESC/Java2 and Spec# (following JML) include the ability to declare specification-only model fields [9] and abstraction functions for representing the relationship between the value of the model field and the implementation so that refinements can be proved. The authors in [18] claim the existing techniques for JML model variables suffer from soundness, modularity, expressiveness, or practical problems. They present a simpler but more expressive methodology for model fields, but it has not yet been implemented.

ESC/Java2 and Spec# provide precise feedback as to where errors occur. Our tool does not yet provide such precise feedback. However, the output html file produced by the PD theorem prover is informative. That line number easily associates with the Eiffel feature having the same name or assertion tag, so that it is relatively easy to track back to where the problem was. We have to improve this feedback reporting in future versions.

## 5 Conclusion

We have presented in this paper a system where we make use of the mathematical but executable ML library and the translator to convert clean and expressive Eiffel code into PD for automated verification. The translation process transforms each Eiffel construct into an equivalent PD one so that this one-to-one relation between Eiffel and PD constructs allows us to assign the semantics of the PD language to that of Eiffel. Of course such semantics depends upon the soundness of PD.

When the ES-Verify translator is applied to the Eiffel code for the birthday book example, the

PD theorem prover generates 158 verification conditions which are *all* automatically discharged. This includes proof of termination via the loop variant. We used a value semantics class `ESV_ARRAY` for the two implementation arrays. Preliminary experience with other examples indicates that the vast majority of verification conditions are quickly and automatically discharged, including loop variants and invariants, without any interaction with the user. The user may add axioms (with the danger of introducing inconsistencies) or assertions to help the theorem prover, but this is mostly unnecessary. Future work aims to extend the verification to handle the issue of reference aliasing and inheritance.

**Acknowledgements:** We deeply appreciate the help we have received from David Crocker of Escher Technologies with the Perfect toolset. Likewise we would like to acknowledge helpful feedback from Bertrand Meyer and Bernd Schoeller of ETH Zurich. This work was funded by a Discovery Grant from NSERC.

# References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003. With Praxis Critical Systems Limited.

[3] Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fhndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. *Position paper at VSTTE*, 2005.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. 2004.

[5] Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal Ocl Semantics in Isabelle/Hol. In *Theorem Proving in Higher Order Logics*, volume LNCS 2410. Springer-Verlag, 2002.

[6] 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. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Springer-Verlag, editor, *Formal Methods for Components and Objects (FMCO'2005)*, LNCS, 2006.

[8] Yoonsik Cheon. A runtime assertion checker for the java modeling language. TR 03-09, Department of Computer Science, Iowa State University, April 2003.

[9] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[10] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Technical Report NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

[11] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. In *Tools Exhibition Notes at Formal Methods Europe*, 2003.

[12] David Crocker. Safe Object-Oriented Software: The Verified Desing-By-Contract Paradigm. In F.Redmill & T.Anderson, editor, *Twelfth Safety-Critical Systems Symposium*, pages 19–41. Springer-Verlag, London, 2004.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[14] Escher Technologies. *Perfect Developer Language Reference Manual*, 3.0 edition, December 2004. Available from www.eschertech.com.

[15] Ingo Feinerer. Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations. Master's thesis, Vienna University of Technology, January 2005.

[16] C. A. R. Hoare. Proof of Correctness of Data Representations. In *Acta Informatica*, volume 1, pages 271–281. Springer-Verlag, February 1972.

[17] Gary T. Leavens, K. Rustan M. Leino, and Peter Mller. Specification and verification challenges for sequential object-oriented programs. TR 06-14, Department of Computer Science, Iowa State University, May 2006.

[18] K. Rustan M. Leino and Peter Mller. A verification methodology for model fields. ESOP 2006.

[19] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[20] J.M. Spivey. *The Z Notation: A Reference Manual (2nd edition).* Prentice-Hall, Englewood Cliffs, N.J., 1992.

[21] Brian Stevens. Implementing Object-Z with PerfectDeveloper. *Journal of Object Technology*, 6(2):189–202, March-April 2006.

[22] Kim Walden and Jean-Marc Nerson. *Seamless Object Oriented Software and Architecture.* Prentice Hall, 1995. Seamless Object Oriented Software and Architecture.

# Cross-Verification of JML Tools: An ESC/Java2 Case Study

Patrice Chalin and Perry James

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
www.dsrg.org

**Abstract**. This paper presents a case study in the use of the JML Run-time Assertion Checker (RAC) compiler on ESC/Java2, an extended static checker for the Java Modeling Language (JML). We believe that overall product quality is maximized by the use of complementary verification tools. Use of the JML RAC allowed us to uncover deeper problems with the design of ESC/Java2 than was possible with static analysis alone. Some problems that were found with the RAC are discussed, along with tentative and implemented solutions.

## 1 Introduction

The two main components of the Dependable Systems Evolution Grand Challenge (also named GC6) [1, 2] are the Verifying Compiler (VC) project and the Verified Software Repository (VSR). A Verifying Compiler is envisioned as a tool to be used by mainstream developers to statically prove that a program is correct. The VSR is meant to hold, among other things, examples of early VC prototypes and "challenge codes," i.e., realistic programs in the form of source code, specifications, and documentation that will be usable as benchmarks for the purpose of exercising proposed VC candidate technologies [3].

This paper reports on our progress in preparing the verification tools of the Java Modeling Language (JML) [4] as potential candidates for inclusion in the VSR. Since the verification tools themselves are written using JML-annotated Java, they can serve as challenge codes as well. More precisely, this paper presents a case study in the use of the JML Run-time Assertion Checker (RAC) compiler on ESC/Java2, an extended static checker for JML.

The main thesis of this paper is that overall product quality is maximized by the use of *complementary* verification tools. For example, routine application of the ESC/Java2 to itself has resulted in the elimination of common coding and specification errors. The strength of ESC/Java2 is that it performs fully automated verification and offers a familiar compiler-like interface to developers. As is typical with fully automatic checkers, it has compromised completeness and soundness for efficacy. On the other hand, use of the JML RAC allows specifiers to verify (albeit at runtime) most assertions, and hence can achieve a much higher degree of completeness. As a consequence, use of the JML RAC has allowed us to uncover deeper problems with the design of ESC/Java2.

The remainder of the paper is organized as follows: In Section 2 we give a brief introduction to JML and its supporting verification tools. Section 3 covers the main design issues in ESC/Java2 that have been uncovered by using the JML RAC. We conclude in Section 4.

# 2 JML and its tools

The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) for Java that supports Design By Contract (DBC) [5, 6] as well as more advanced features such as frame properties and specification-only fields [7]. JML enjoys support from a wide range of tools that are useful for verification, including the following [8]:

- JML RAC, also called the JML Compiler (`jmlc`). Compiling JML annotated Java files using `jmlc` instruments the code with runtime checks of the correctness of the class contracts.
- ESC/Java2 can perform extended static checking—i.e., a form of fully automated verification of contracts that is neither sound nor complete.
- LOOP compiler can be used to compile JML-annotated Java classes into PVS theories containing proof obligations. Discharging the proofs using PVS establishes the (total) correctness of the classes.

As can be imagined, use of the LOOP tool and PVS in conducting verifications requires a high level of sophistication on the part of their users. The JML RAC and ESC/Java2, on the other hand, offer a familiar compiler-like interface and conduct verification fully automatically and tend to be the main verification tools used by developers.

When both tools are used during development, the following informal process has proven useful. First, use is made of ESC/Java2 to eliminate obvious errors. When a specification becomes too involved, ESC/Java2 will report that it is unable to prove, e.g., that a method body satisfies its contract. In this case, one must resort to using the JML compiler. Even though some assertion expressions are not executable (e.g., some forms of quantified expression), the RAC can generally check more specification statements than ESC/Java2. The caveat, of course, is that the compiled code must be run so that as many input cases as possible are exercised. Another tool, JMLUnit, can be used as a test oracle, automatically creating JUnit test cases from JML specifications. In the next section we explain how we made use of the JML RAC to further verify the contracts of the ESC/Java2 application classes.

# 3 Case study

## 3.1 Compiling ESC/Java2 source with the JML RAC

This project was initiated in the summer of 2005. Prior to that time, the ESC/Java2 source had not been compiled with the JML RAC. Overall, it took approximately four developer-weeks to make the necessary updates to both the ESC/Java2 source based on problems reported by the static checking component of the RAC. Most of these changes were due to slight incompatibilities between the syntax accepted by the JML compiler and that used in the source files. A few bugs were removed from (or, more accurately, enhancements were made to) the repository of API specifications (e.g., `java.lang`, `java.lang.util`, etc.) that are distributed with ESC/Java2.

The exercise also allowed us to uncover and file reports on 8 bugs in the JML RAC. A major problem, which was only recently resolved, prevented the JML RAC from creating instrumented .class files for three classes because the checking code had a try/catch block that is larger than the limits allowed by the JVM [9]. With this problem overcome, we were able to compile the 550 classes of the ESC/Java2 application with jmlc in a little over 7 minutes (on a 2GHz P4). These classes are distributed over 3 main packages:

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInvariantError:
by method PrimitiveType.<init>@post
File "Javafe/java/javafe/ast/PrimitiveType.java", line 128, character 15
regarding specifications at
File "Javafe/java/javafe/ast/PrimitiveType.java", line 35, character 17 when
'tag' is 0
'this' is [PrimitiveType tmodifiers = null tag = 0 loc = 0]
at javafe.ast.PrimitiveType.checkInv$instance$PrimitiveType(PrimitiveType.java:958)
at javafe.ast.PrimitiveType.<init>(PrimitiveType.java:210)
at javafe.ast.PrimitiveType.internal$makeNonSyntax(PrimitiveType.java:97)
at javafe.ast.PrimitiveType.makeNonSyntax(PrimitiveType.java:3029)
at javafe.tc.Types.internal$makePrimitiveType(Types.java:154)
at javafe.tc.Types.makePrimitiveType(Types.java:4016)
at javafe.tc.Types.<clinit>(Types.java:19)
at escjava.Main.<init>(Main.java:78)
at escjava.Main.compile(Main.java:215)
at escjava.Main.main(Main.java:177)
```

**Figure 1. Run-time assertion violation reported by ESC/Java2 compiled with the JML RAC**

- `javafe`, a common front end used by ESC/Java2 and other tools, such as Houdini [10].
- `escjava`, a package that builds on the services provided by the Javafe to implement the extended static checking functionality.
- `junitutils`, various support utilities, particularly with automated testing.

In the subsections that follow, we detail some of the major problems reported by ESC/Java2 when running the RAC compiled version. Note that all of these errors were identified during static initialization. That is, these errors report inconsistencies between the static initialization code and the JML specifications of the ESC/Java2 classes. The errors are presented essentially in the order in which they were discovered. As was mentioned in the introduction, we will see that the errors identified have fairly deep design implications.

## 3.2 AST node invariants not established by constructors

The first error to be reported by the RAC instrumented ESC/Java2 is shown in Figure 1. While possibly intimidating to the uninitiated, just a little training is generally sufficient to make sense of the error report. We can see that the class invariant (`JMLInvariantError`) of the `PrimitiveType` class was violated on exit (i.e., during the verification of the postcondition) of the `PrimitiveType` constructor (represented by `<init>`). How is it that ESC/Java2 did not report this error? We will see why this is so shortly.

The violation occurred at line 128 of the file `Primitivetype.java`. This file is part of the collection of `javafe` Abstract Syntax Tree (AST) node classes. An excerpt of the file is given in Figure 2. The figure shows only one sample invariant clause (constraining the value of the `tag` field) at the start of the file. A static `make()` method and the problematic constructor are also shown. At line 128, we see that the body of `PrimitiveType()` is empty. Its associated Javadoc comment explains why. Apparently, a fundamental design decision for the AST node class hierarchy had been to have all AST nodes created via maker methods (generally named `make()`). The maker methods first invoke a default constructor having an empty body and then proceed to initialize the object fields. The AST node instance returned by this maker method is meant to satisfy its class invariant.

Of course, class invariants are meant to hold for *all* instances of the class including those created by default constructors with empty bodies. Since this is clearly not the case for the AST node constructors, use is made of the ESC/Java2 `nowarn` pragma. This pragma allows developers to instruct ESC/Java2 to ignore certain kinds of errors—e.g., invariant errors in the case of `PrimitiveType`. As a reminder to developers of the obligation to establish the class

```
public class PrimitiveType extends Type
{
  /*@ invariant (tag == TagConstants.BOOLEANTYPE || ...); */
  public int tag;

  //@ requires (tag == TagConstants.BOOLEANTYPE || ...);
  //@ ensures ...
  public static /*@ non_null */ PrimitiveType
      make(TypeModifierPragmaVec tmodifiers, int tag, int loc)
  {
      //@ set I_will_establish_invariants_afterwards = true;
      PrimitiveType result = new PrimitiveType();
      result.tag = tag;
      result.loc = loc;
      result.tmodifiers = tmodifiers;
      //...
      return result;
  }

  /**
   * Construct a raw PrimitiveType whose class invariant(s) have not
   * yet been established.  It is the caller's job to initialize the
   * returned node's fields so that any class invariants hold.
   */
  //@ requires I_will_establish_invariants_afterwards;
  protected PrimitiveType() {}   //@ nowarn Invariant,NonNullInit; // ** LINE 128 **
  ...
}
```

**Figure 2. Excerpt of javafe/ast/PrimitiveType.java**

invariant after calling the default constructor, a specification-only (ghost) variable named "`I_will_establish_invariants_afterwards`" was created. We see in Figure 2 how the `make()` method uses the default constructor, sets the instance fields, and sets the special-purpose ghost variable to `true`.

While there are a number of solutions to this problem, the simplest was to eliminate the default constructor in favor of constructors that establish invariants right from the start. In doing so, we simplified the design by consolidating the instance creation process, eliminating the `I_will_establish_invariants_afterwards` variable and the `nowarn` pragma. In this way, both ESC/Java2 and the JML RAC can process the resulting specifications. While our new design impacted almost two hundred classes, most of the changes were confined to AST node generation routines and templates.

We note that there are generally two main reasons for using `nowarn` pragma:

1. When a specifier believes something to be true but the verifier is unable to confirm its truth. In such a case, the RAC facility can confirm that the specification does indeed hold at runtime for the exercised test cases.
2. When a specifier knows something to be false, but wants to ignore it for the moment and continue making progress (in verifying other parts of the program). The RAC will catch these violations and prevent the system from being usable.

It would be helpful to developers if all `nowarn`s were commented with the reason for their presence. Instances of (2) should be resolved as quickly as possible so that all of our tools can be used in support of our development efforts. It would appear that the case treated in this subsection is an instance of (2)—maybe there was a belief that the use of non-default constructors was not feasible, when in fact it turns out to be straightforward.

## 3.3 Internal AST node instances vs. AST node class invariants

```
public static /*@ non_null */ FieldDecl lengthFieldDecl
  = FieldDecl.make(..., lenId, Types.intType, Location.NULL,...);
```

**Figure 3. Declaration of length field for arrays in javafe.tc.Types**

```
package javafe.ast;

public abstract class GenericVarDecl extends ASTNode
{
  ...
  public /*@ non_null @*/ Type type;
  //@ invariant type.syntax;
  ...
}

public class FieldDecl extends GenericVarDecl implements ...
{
  ...
  //@ requires locId != javafe.util.Location.NULL;
  //@ ensures \result != null;
  public static FieldDecl make(...,
                               /*@ non_null @*/ Type type,
                               int locId, ...)
  {
     //...
  }
}
```

**Figure 4. Excerpts from GenericVarDecl and FieldDecl of javafe.ast**

The next two problems reported by the RAC are related to the creation of an internal field for the length of arrays (viz., `lengthFieldDecl`), itself of type `int` (Figure 3). The violations were, firstly, of the invariant of `GenericVarDecl` that `type.syntax` be true (i.e., that the type be an AST node read from a file, not an internally create type like `Types.intType`)—see Figure 4. The second violation had do to with the with the `locId` of the `FiedlDecl` maker method: it was required to be different from `Location.NULL`. Of course, neither of these conditions is satisfied by the call to `make()` in Figure 3.

After some analysis, and two unsatisfactory attempted solutions, the approach we implemented was to create a new maker method and constructors for `FieldDecl` and `GenericVarDecl` that do not take a location. These would set the `GenericVarDecl`'s `locId` to `Location.NULL`. To capture the idea of an internal field, an `isInternal()` method was added to `GenericVarDecl`. This method returns `true` exactly when the location is not equal to `Location.NULL`. Because of these changes, `FieldDecl`'s new maker method no longer mentions location, and the old one remains unchanged. The invariant of `GenericVarDecl`, `FieldDecl`'s super class, was changed to reflect that `syntax` is `true` exactly when `isInternal()` is `false`. To reflect that there are no internal AST nodes of subclasses other than `GenericVarDecl` (viz., `FormalParaDecl` and `LocalVarDecl`), all other AST node classes have `!isInternal()` as an invariant.

## 3.4 Specification and polymorphic structures

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError: by method
PrimitiveType.makeNonSyntax regarding specifications at
File "Javafe/java/javafe/ast/PrimitiveType.java", line 78, character 16
when
        'tag' is 247
        at escjava.Main.compile(Main.java:4138)
        at escjava.Main.internal$main(Main.java:118)
        at escjava.Main.main(Main.java:3479)
```

**Figure 5. RAC error: violation of PrimitiveType maker method precondition**

```
package javafe.ast;
public class PrimitiveType extends Type {
  //@ invariant (tag == TagConstants.BOOLEANTYPE || ...); // ???
  public int tag;

  //@ requies ???
  protected PrimitiveType(…, int tag, int loc) {
     this.tag = tag;
     ...
  }
}

package escjava.ast;
public class EscPrimitiveType extends PrimitiveType
{
  //@ requires (* tag is a valid javafe tag or an esc tag *);
  protected EscPrimitiveType(…, int tag, int loc)  {
      // tag might not be a valid PrimitiveType tag!
      super(tmodifiers, tag, loc);
  }
  // ...
}
```

**Figure 6. Sample (invalid) solution: excerpts of PrimitiveType and EscPrimitiveType**

A very interesting design problem that runtime assertion checking highlighted involved (a violation of) *behavioral subtyping* [11]. As mentioned above, Java's primitive types are represented using instances of `PrimitiveType`. This class belongs to the `javafe` package, which is common to tools that need a Java front end. Primitive types are distinguished by a `tag` attribute. The maker methods require that a valid tag be used when creating a new instance of `PrimitiveType`, and an invariant ensures that the tag remains valid. A valid tag is defined in `PrimitiveType` to be one of ten given tag values. The tags themselves are defined in the class `javafe.ast.TagConstants`.

As was mentioned earlier, the `escjava` package makes use of services of the Java front-end package. In particular, it makes direct use of the `PrimitiveType` class to define ESC/Java2- and JML-specific primitive types such as `lockset` and `\bigint`. To do so, new tags are defined in `escjava.ast.TagConstants`. Unfortunately, the static creation of, e.g., the `escjava lockset` primitive type results in a violation of the `PrimitiveType` maker method's precondition—see Figure 5—since the maker is given a tag value that is not one of the expected ten "valid" values.

One approach considered was to define a subtype of `javafe.ast.PrimitiveType` named `escjava.ast.EscPrimitiveType` and represent the ESC/Java2 and JML primitive types with instances of this new class. Unfortunately, the semantics of class invariants and the enforcing of behavioral subtyping in JML make it impossible to write any useful class contracts for `PrimtiveType` and `EscPrimitiveType` in such a case (even if, for example, we use an auxiliary boolean method `isValidTag`). The problem is illustrated in Figure 6. The first problem to be noticed is that it is difficult to choose an appropriate class invariant restricting the

value of `tag`. E.g., it cannot be limited to only `javafe` tags, otherwise `EscPrimitiveType`s could not be created. We cannot say in `javafe.ast.PrimitiveType` that the legal tags also include those of the `escjava.ast` package since this would create circular dependencies between `javafe` and `escjava`. Similarly, notice how the `EscPrimitiveType` constructor invokes the `PrimitiveType` constructor (via `super`). For this call to be permitted, what precondition must the `PrimitiveType` constructor have with respect to its tag parameter?

Instead of trying to work with the untenable solution consisting of two classes, we decided to extract an interface and allow both the Java front end and ESC tools to implement this common interface. The interface has both a code and model version of an `isValidTag` method. By specifying that the code version result is the same as the model version's we are able to statically verify the invariant that `isValidTag` always returns `true`.

The two implementations of the interface (viz., `JavaFePrimitiveType` and `EscPrimitiveType`) have implementations of `isValidTag` that compare against the appropriate values in each case. Their makers and constructors require that the tag value passed to them be valid, as determined by their local versions of `isValidTag`. Since the value passed to the makers and constructor is valid, and since this value is stored as the type's `tag`, the invariant can be statically shown to hold. This solution is illustrated in Figure 7.

```
package javafe.ast;
public interface PrimitiveType
{
  //@ public model instance int _tag;
  //@ pure model boolean specIsValidTag(int tag);

  //@ ensures \result == specIsValidTag(_tag);
  /*@ pure */ boolean isValidTag();

  //@ public invariant isValidTag();

  //@ ensures \result == _tag;
  /*@ pure */ int getTag();
}

package escjava.ast;
public class EscPrimitiveType implements PrimitiveType
{
  /*@ spec_public */ private int tag;
  //@ public represents _tag <- this.tag;

  /*@ public normal_behavior
    @   ensures \result == (JfePrimitiveType.isValidTag(tag) ||
    @         tag == TagConstants.LOCKSET || ...);
    @*/
  public static /*@pure*/ boolean isValidTag(int tag) {
     return (JfePrimitiveType.isValidTag(tag) ||
             tag == TagConstants2.LOCKSET || ...);
  }

  /*@ also
    @ public normal_behavior
    @   ensures \result == EscPrimitiveType.isValidTag(tag);
    @*/
  public /*@pure*/ boolean isValidTag() {
     return isValidTag(tag);
  }

  /*@ public normal_behavior
    @   ensures \result == EscPrimitiveType.isValidTag(tag);
    @ public model pure boolean specIsValidTag (int tag) {
    @    return EscPrimitiveType.isValidTag(tag);
    @ }
    @*/

  /*@ protected normal_behavior
    @   requires EscPrimitiveType.isValidTag(tag);
    @   ensures  this.tag == tag && ...;
    @*/
  protected /*@pure*/ EscPrimitiveType(..., int tag, int loc)  {
     this.tag = tag; ...
  }
}
```

**Figure 7. Excerpt of correct redesign of PrimitiveType**

# 4 Conclusion

Use of the JML RAC has enabled us to uncover significant design flaws and inconsistencies that ESC/Java2 was unable to report (either due to its unsoundness or incompleteness). Trying to detect these through a manual code review would have been very tedious. Having a tool with another verification approach allowed us to rapidly see problems buried in the code. As expected, the problems that the RAC reported were non-trivial: ESC/Java2 had already been run on itself and the trivial problems exposed had been dealt with. The issues raised by the RAC had deep implications (e.g., violations of behavioral subtyping). Careful engineering analysis of both the specification violations and the related source code allowed us to resolve the issues without falling into deeper traps.

This next step in our verification efforts has allowed us to uncover design issues in the ESC/Java2 front end and type system whose resolution has resulted in a system that has higher consistency between its specification and implementation. We will continue to run RAC-

instrumented versions of ESC/Java2 to uncover further bugs. We hope to soon begin using ESC/Java2 to analyze the source for the JML Compiler, as it also has a large JML-annotated code base. Using the tools iteratively to analyze both themselves and each other should allow us to enhance their quality, hence making them more likely potential candidates for inclusion in the Verified Software Repository.

# References


[1] UKCRC, "Grand Challenges for Computer Research, ," UK Computing Research Committee (UKCRC) 2006.

[2] J. C. P. Woodcock, "Grand Challenge 6:Dependable Systems Evolution," 2006.

[3] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock, "The verified software repository: a step towards the verifying compiler," *Formal Aspects of Computing*, 2006.

[4] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual," 2006.

[5] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, pp. 40-51, 1992.

[6] G. T. Leavens and Y. Cheon, "Design by Contract with JML," Draft paper 2005.

[7] P. Chalin, J. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2 (Tutorial Paper)," in *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05)*, 2005.

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, pp. 212-232, 2005.

[9] F. Rioux, "Effective and Efficient Design by Contract for Java," in *Dependable Software Research Group (DSRG), Faculty of Engineering and Computer Science, Department of Computer Science and Software Engineering*. Montreal, Quebec: Concordia University, 2006.

[10] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," presented at International Symposium of Formal Methods Europe, Berlin, Germany, 2001.

[11] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1811-1841, 1994.

# Static Stability Analysis of Embedded, Autocoded Software

Eric Feron* and Arnaud Venet†

## 1 Introduction

Embedded software-based control systems are commonly constructed using model-based design environments such as MATLAB/Simulink™from MathWorks. These environments allow the system designer to establish critical properties ensuring the reliability of the system (stability, disturbance rejection, etc.) directly at the model level, using a rich mathematical toolset. However, the software implementation substantially transforms the mathematical model by introducing numerous programming artifacts (aggregate data structures, pointers) and altering the numerical representation (platform-dependent floating/fixed-point arithmetic, and, in the most extreme cases, conversion from continuous-time dynamics to discrete-time dynamics). Verifying that the reliability properties of the system are preserved by the implementation is extremely challenging, yet in many cases critically important. Model-based design environments usually come with an *autocoder* i.e., a code generation tool that automatically synthesizes an implementation of the embedded controller from the specification of its model. Autocoders are getting increasingly used in practical applications for they greatly simplify the implementation process. In many corporations however, including aerospace and automotive industries, autocoding is essentially precluded because its properties are considered to be not adequately trustworthy.

Static program analysis tools have recently proven successful in tackling the certification of embedded software-based control systems. ASTREE [3], developed by P. Cousot's team in France, can automatically verify the consistency of floating-point arithmetic in the electric-command control system of the A380, Airbus' super-jumbo carrier. C Global Surveyor [7], developed by Kestrel Technology LLC, can verify the absence of pointer manipulation errors in the mission-control software of NASA's Mars Exploration Rovers. However, the scope of static analysis has been essentially limited to robustness properties, i.e., ensuring the absence of runtime errors during the execution of the program. Verifying functional properties by static analysis for a system to be used in the field requires (1) translating a reliability property of the model into the implementation setting, using the appropriate data structures and numerical

*Georgia Institute of Technology, feron@gatech.edu
†Kestrel Technology, 4984 El Camino Real #230, Los Altos, CA 94022, arnaud@kestreltechnology.com

libraries, and (2) tracking the evolution of this property over all execution paths using abstract interpretation techniques. This process requires a tight coupling between the model description and its implementation. While such tight coupling is rarely achieved in practice, it can exist at least when the implementation is automatically generated from the model. Autocoders, like Real-Time Studio$^{TM}$for Simulink$^{TM}$, are increasingly being used in industry for the development of embedded control software. This means that static analysis techniques specialized for codes automatically generated from high-level models can be developed to meet market needs.

The approach we are investigating consists of translating the formal proof of reliability properties of an embedded logic into a dedicated static analyzer that automatically carries out the corresponding proof on the code generated from the model. Whereas today's commercial (and even known academic) static analyzers for embedded mission- and safety-critical software are handcrafted, we propose to use our prior research to construct the dedicated static analyzer automatically. Ultimately, the system designer would be provided with a fully automated engine that performs verification of the generated code without requiring any additional information other than the high-level model specification.

The paper is organized as follows. In Sect. 2 we describe the challenges posed by the analysis of autocoded embedded control software and how they are addressed in our approach. In Sect 3 we propose a research agenda toward the analysis of reliability properties for embedded control software. Section 4 discusses current issues that come up during the design of the static analysis.

## 2 Challenges

There are two major challenges in developing a tool for the automated verification of reliability properties of embedded control software automatically generated from a high-level model:

1. How do we translate a property of the high-level model into a property of the code generated from that model? What are the features of the autocoder we must know to effectively build this translator?
2. How do we specify the basic components of the static analyzer required to verify the desired property? How do we specialize the analyzer to the code generated by a given autocoder?

We discuss these challenges in the following subsections.

### 2.1 Property Translator

The autocoder of the model-based design environment generates code from the model in a predictable way. Therefore, we expect to be able to map a property of the model's variables

into a property of the data structures used in the model's implementation. Commercial model-based design environments offer rich libraries of basic components for building systems and the properties of interest may greatly vary depending on the nature of the system designed.

The model-based design environment we have chosen is the Matlab/Simulink$^{TM}$tool suite; the family of systems we are beginning with is that of closed-loop dynamical systems, represented on the one hand by a family of differential equations that capture the system's physics, and on the other hand the closed-loop control algorithm and its associated code. The functional properties we are interested in include closed-loop systems stability, closed-loop system performance (eg tracking performance), and reachability analyses. We are currently interested in describing how to map that property to the implementation by conducting an extensive study of the code generated from a benchmark of systems in that family.

### 2.2 Static Analysis Specification Framework

Checking the transposed property of the model on the code may require a specific analysis algorithm that takes into account the underlying computational model (floating/fixed-point) and the nature of the reliability property (linear or ellipsoidal invariants). Moreover, the code generated by the model-based design environment may use programming language constructs (like pointers or union types in C) that pose a difficulty for the static analyzer. These constructs may require dedicated analysis algorithms (pointer analysis, type analysis) specially tailored for the particular structure of code produced by the autocoder. These static analyzers are strongly specialized toward the family of models and properties considered. This does not require rewriting the analyses from scratch for each of these configurations. We need a library of baseline static analysis algorithms (pointer analysis, floating-point analysis, type analysis, etc.) and a framework for combining them in a way to address each configuration's unique requirements. We are in the process of establishing a taxonomy of static analysis algorithms and describing a specification framework for expressing arbitrary combinations of these algorithms.

## 3 Research Agenda

### 3.1 Overview of the problem: Illustrative example

Consider a dynamical system described by the following equations of motion

$$\begin{array}{l} \frac{d}{dt}x = y \\ \frac{d}{dt}y = u + w \end{array} \tag{1}$$

where $x$ is the variable to be controlled, $u$ is a control variable, and $w$ a disturbance whose value is unknown (for example wind gusts). We assume a sensor system is able to read $x$.

A typical embedded system design problem is to come up with a logic that, based upon successive readings of $x(t)$ over time, is able to generate a sequence of control inputs $u(t)$ such that the "closed-loop" system is stable (that is, the variable $x$ never grows out of bounds), and achieves proper *disturbance rejection*, that is, minimizes the deviations of $x$ generated by $w$. Control systems engineering provides a convenient framework to achieve such performance requirements, by using linear dynamical systems as a modeling framework for such a logic. For example, the logic may be specified by the dynamical system:

$$\begin{array}{l} \frac{d}{dt}x_1 = y, \\ \frac{d}{dt}x_2 = -10x_2 + y, \\ u = -0.01x_1 + \frac{9}{10}x_2 - 10y \end{array} \tag{2}$$

and it is indeed easy to check (via a simple eigenvalue computation, for example) that the system *specification* consisting of (3) and (2) is stable, and therefore correctly achieves the most important requirement for the embedded system. An alternative and more powerful stability criterion consists of computing an *invariant* for (3) together with (2). Such an invariant might easily be obtained by considering the evolution over time of $V(x, y, x_1, x_2)$, with

$$V(x, y, x_1, x_2) = \begin{bmatrix} x \\ y \\ x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 768.5818 & -0.5000 & 0.8254 & -6.3254 \\ -0.5000 & 82.5378 & 50.0000 & 6.7959 \\ 0.8254 & 50.0000 & 495.5018 & 4.4932 \\ -6.3254 & 6.7959 & 4.4932 & 0.6616 \end{bmatrix} \begin{bmatrix} x \\ y \\ x_1 \\ x_2 \end{bmatrix}$$

The level sets of $V$ are ellipsoids, and, in the absence of perturbations $w$, the system (3) together with the logic specification (2) satisfies

$$\frac{d}{dt}V \leq 0$$

for any initial condition. This in turn ensures all variables of the closed-loop system specification remain bounded, and therefore the system is stable.

Consider now the *implementation* of the logic (2): This implementation must replace the continuous-time system (3) by a difference scheme instead, with or without variable time step. Other significant differences may exist, including the coding of all variables in floating-point arithmetic. The invariant function $V$ may, however, still be used to prove the stability of the system (3) together with the implementation of the controller specification (2). This is what we propose to achieve by conducting static analysis on the code generated from the model, using Abstract Interpretation techniques. Kestrel Technology has a substantial body of implemented analyzers based on abstract interpretation and libraries [7, 6] available. These algorithms are being incorporated in *CodeHawk*, a generic static analysis development framework that enables the automated construction of static analyzers from a set of requirements.

### 3.2 From system-level properties to implementation-level properties

In the case of the reliability property presented in the previous section (stability), we have to prove the preservation of an ellipsoidal invariant. This property is expressed on a continuous

model and has to be transformed so that it can be checked directly on the implementation of a discrete model extracted from the continuous formulation. This process can be split up in two separate steps:

(1) Generate a discrete executable model from a continuous formulation.

(2) Translate the discrete executable model into an actual program.

The first step is a standard mathematical transformation that is fairly independent from the choice of a particular autocoder. In order to assess the feasibility of our approach, we are designing a static analysis framework that operates on the executable model of (1). This executable model can be seen as the most detailed operational formulation that can be stated independently from the characteristics of the target programming language. Since this formulation does not use low-level programming language constructs we can focus on the algorithmic aspects of the static analysis. We assign a syntax and semantics to this operational model in order to conduct formal reasoning on it and define its abstract interpretation rigorously. The connection between this intermediate operational model and the actual implementation is investigated independently by studying the code generated from a benchmark of representative models that we are building build for that purpose.

### 3.3 Tailoring the analyzer to the property to verify

The executable model of the system introduces a complex control-flow with loops that represents the iterative computation of the control logic over discretized time. This has the effect of breaking down the simple formulation of the original logic into a number of intermediate steps. This means that the original invariant that holds at some points in the executable code does not hold at some others. Therefore, the analysis must be able to model how the invariant is transformed by elementary operations of the executable model. We need what is called in the jargon of Abstract Interpretation an *abstract domain*. In the case of ellipsoidal invariants, we need to construct an abstract domain that can represent all possible ellipsoidal invariants over a given set of variables as well as the associated semantic transformers. The semantic transformers model the effect of elementary operations like initializing a variable or incrementing its value.

The executable model on which we are working hides low-level implementation details, there is one feature of the target platform that we must take into account at this level because of its impact on the design of the abstract domain: the computational model of reals (floating/fixed-point arithmetic). How roundoffs are performed and imprecision is propagated is extremely important, because the accumulation of roundoff errors can cause a violation of the invariant and compromise the stability of the control system. This issue extends to other discretizations that may be performed automatically, such as time discretization. Tracking the propagation of roundoff and discretization errors requires a proper abstraction of the floating-point computational model that is amenable to mechanized analysis.

The design of the abstract domain is the most critical aspect of our approach. It requires expertise both in Abstract Interpretation and Control Theory in order to devise a representation

of invariants that will effectively enable checking the correctness of the executable model with good performance. Fortunately, there is a substantial body of work already published that addresses many the problems involved in the design of such abstract domains, like the inference of numerical invariants for floating-point computations [5] or the discovery of ellipsoidal invariants for the analysis of linear digital filters [4], with extensions to uncertain dynamical systems [1]. Abstract domains specialized for certain properties can be combined in various ways in order to obtain more expressive ones that can handle more complex properties. Therefore, the effort of building a new abstract domain is incremental. We plan to use *CodeHawk* as a repository of abstract domains as well as a generator for building new abstract domains from combination of existing ones.

### 3.4 Analyzing implementation artifacts

Studying the analysis of the executable model provides us with the core concepts for carrying out the verification at the implementation level. In order to translate these concepts into an actual analyzer that operates at the generated code level, we need a number of auxiliary analyses whose purpose is to recover structural data from the implementation artifacts. For example, consider the following function which has been taken out of the C code generated by Real-Time Studio from the sample continuous model described in Section 3.1.

```
static void rt_ertODEUpdateContinuousStates(RTWSolverInfo *si , int_T tid)
{
  time_T tnew = rtsiGetSolverStopTime(si);
  time_T h = rtsiGetStepSize(si);
  real_T *x = rtsiGetContStates(si);
  ODE1_IntgData *id = rtsiGetSolverData(si);
  real_T *f0 = id->f[0];
  int_T i;

  int_T nXc = 2;

  rtsiSetSimTimeStep(si,MINOR_TIME_STEP);

  rtsiSetdX(si, f0);
  logic_derivatives();
  rtsiSetT(si, tnew);

  for (i = 0; i < nXc; i++) {
    *x += h * f0[i];
    x++;
  }

  rtsiSetSimTimeStep(si,MAJOR_TIME_STEP);
}
```

This function updates the control variables at each time step accordingly to the specified logic. In order to recover the original components of the continuous model we must perform a pointer analysis that is able to distinguish between elements of arrays accessed through pointers (like `x` and `f0`) and a numerical analysis that is able to infer the range of index variables (like `i`) used for manipulating the model data. If we consider pointer analysis for example, there is a broad spectrum of existing algorithms that greatly differ in terms of precision and scalability. Depending on the structure of the code produced by the generator or just the family of models considered, we may have to choose different algorithms. In order to achieve this we need a high level of flexibility for specifying the set of analyses to be performed on the code.

We plan to use the *CodeHawk* static analysis generator that precisely enables the rapid development of efficient analyzers from a list of requirements. What has to be studied, however, is how to state these requirements for this specific application, i.e., how to recover the structure of the high-level model. This means that we have to determine a formalism that both specifies which combination of analysis to use and how to tie the results of the analysis back to the model components. This formalism would ultimately be integrated within *CodeHawk*. We are now studying the features that such a formalism should possess and we will propose a tentative definition from experiments conducted on a benchmark of models using *CodeHawk*.

## 4 Some Current Issues: Designing the Collecting Semantics

When analyzing code, a central issue is the design of what is called the *collecting semantics* [2]. The collecting semantics describes how much information from the original program must be retained in order to verify the desired property. The collecting semantics forms the base model on which static analysis is conducted. Higher levels of semantic collection allow one to define more compact models of the software execution, but this task may also be more complex, since information must be collected over several lines of code and then linked into a compact model. Thus, lower semantic levels (like "line-by-line" analyses) are more desirable from the standpoint of analyzer simplicity and adaptability. We now show on a simple example that testing typical control-related invariants raises interesting issues with respect to the collecting semantics.

Consider a C code implementation of the simple recursion

$$\begin{array}{l} \texttt{for(;;) \{} \\ \quad x_+ = Ax; \\ \quad x = x_+; \\ \} \end{array} \tag{3}$$

In this infinite recursion, $x \in \mathbf{R}^n$ is the state of the system, and the matrix $A \in \mathbf{R}^{n \times n}$ is a square matrix. We assume the matrix $A$ to be stable, that is, all trajectories $x$ converge to zero, which is equivalent to the spectral radius of $A$ being strictly less than 1. A typical invariant used to prove this property is a quadratic function of the state $x$, that is, a function of the form $V(x) = x^T P x$, where $P \in \mathbf{R}^{n \times n}$ is a positive definite, symmetric matrix. Such quadratic

invariants can always be found if the matrix $A$ is stable, and the monotonic decay of $V$ is equivalent to the matrix inequality (in the sense of the partial order of symmetric matrices):

$$A^T P A - P \leq 0$$

Assume such a matrix $P$ is known. A software implementation of (3) might include several lines of code to carry the line $x_+ = Ax$, and then several other lines of code to implement $x = x_+$. When $n > 1$, the latter command can raise significant issues, especially if they are distributed throughout the code as is often done. Indeed, each line of code corresponds to the substitution of *one entry of the state vector $x$ at a time*. The effect on the invariant is, most often, disastrous. Indeed, considering for example a substitution of pointer values of the kind

```
for (i =0; i < n; i++) {
  *(x + i) = *(x_+ + i);
}
```

Then the value of the invariant $V$ after the first line of this recursion is

$$\begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T P \begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where $a_1$ is the first row of the matrix $A$, and the difference between this and $V(x) = x^T P x$ is

$$\begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T P \begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - x^T P x,$$

or

$$\begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T P x + x^T P \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T P \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

the last of the three terms is positive, such that the candidate invariant $x^T P x$ decays if and only if, writing $a_1 = [a_1 1 \; a_1 2 \; \ldots a_1^n]$ and introducing $b = [a_1 1 - 1 \; a_1 2 \; \ldots a_1^n]$, we have, for any $x$:

$$x^T b^T p_1 x + x^T p_1^T b x \leq 0$$

where $p_1$ is the first row of $P$. However, it is well known from linear algebra that such an inequality is possible only if $b$ and $p_1$ are collinear and oriented in the same direction, which is usually not the case. Thus, the invariant $x^T P x$ does not decay line-by-line, and we must first collect the *entire* substitution of $x$ by $x_+$ before the invariant decays. This substantially constrains the design of the collecting semantics and the static analysis, since all possible execution traces of a loop must be collected and represented by one semantic object.

## Conclusion

Static analysis of embedded software arising from autocoded specifications is a necessary step towards broad and safe acceptance by the industrial community. This paper has outlined a research program aimed at achieving this task by exploiting the underlying structure arising from this type of software, using abstract interpretation and control systems analysis methods. The execution of this research is an oingoing activity and its findings will be reported in future publications.

## References


[1] Boyd, S., El Ghaoui, L., Feron, E., and Balakrishnan, V. Linear matrix inequalities in system and control theory. *SIAM Studies in Applied Mathematics 15* (1994).

[2] Cousot, P. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, 1981, ch. 10, pp. 303–342.

[3] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)* (2005), vol. 3444 of *Lecture Notes in Computer Science*, pp. 21–30.

[4] Feret, J. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)* (2004), no. 2986 in LNCS, Springer-Verlag.

[5] Miné, A. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04* (2004), vol. 2986 of *LNCS*, Springer, pp. 3–17.

[6] Venet, A. A scalable nonuniform pointer analysis for embedded programs. In *Proceedings of the International Static Analysis Symposium, SAS 04* (2004), vol. 3148 of *Lecture Notes in Computer Science*, Springer, pp. 149–164.

[7] Venet, A., and Brat, G. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation* (2004), pp. 231–242.