
Proceedings Paper:
Woodcock, James Charles Paul, Divakaran, Sumesh, D'Souza, Deepak et al. (2015) Refinement-Based Verification of the FreeRTOS Scheduler in VCC. In: Butler, Michael, Conchon, Sylvain and Zaidi, Fatiha, (eds.) Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015. Lecture Notes in Computer Science. Springer, pp. 170-186.

https://doi.org/10.1007/978-3-319-25423-4_11

Refinement-Based Verification of the FreeRTOS Scheduler in VCC

Sumesh Divakaran[1,2], Deepak D'Souza[1], Anirudh Kushwah[1], Prahladavaradan Sampath[3], Nigamanth Sridhar[4], and Jim Woodcock[5(✉)]

[1] Indian Institute of Science, Bangalore, India
{sumeshd,deepakd,anirudhkushwah}@csa.iisc.ernet.in
[2] Government Engineering College, Idukki, India
[3] MathWorks India, Bangalore, India
prahlad.sampath@gmail.com
[4] Cleveland State University, Cleveland, USA
n.sridhar1@csuohio.edu
[5] University of York, York, UK
jim.woodcock@york.ac.uk

Abstract. We describe our experience with verifying the scheduler-related functionality of FreeRTOS, a popular open-source embedded real-time operating system. We propose a methodology for carrying out refinement-based proofs of functional correctness of abstract data types in the popular code-level verifier VCC. We then apply this methodology to carry out a full machine-checked proof of the functional correctness of the FreeRTOS scheduler. We describe the bugs found during this exercise, the fixes made, and the effort involved.

1 Introduction

The verification of the FreeRTOS real-time kernel was proposed in 2008 as one of the pilot projects of the Verified Software Initiative led by Hoare [19]. FreeRTOS [14] is a priority-based real-time scheduler and is an open-source representative of some of the commonly used kernels in the auto and aviation sectors, like the OSEC and ARINC 653 real-time operating systems. The correctness of applications (many of them safety-critical) that run on such kernels, as well as the analysis of such applications [25,30], crucially depend on the correctness of the kernel and its specification model. With this motivation in mind, we took up the goal of verifying the correctness of the scheduling-related functionality of FreeRTOS. This paper describes the choices made, the methodology developed, and the results achieved in this project.

The first choice to be made was about the kind of proof technique to adopt: one based on the direct use of code-level contracts or one based on the notion of

Supported by grants from the UK-India Education and Research Initiative (UKIERI) and the Robert Bosch Center for Cyber Physical Systems, IISc, Bangalore.

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 170–186, 2015.
DOI: 10.1007/978-3-319-25423-4_11

refinement. While recent verification efforts for functional correctness in the community [3,7,24,28] – with the prominent exception of the seL4 project [23] – have favoured the use of code-level contracts (in the form of `requires` and `ensures` annotations for methods in the program) over refinement-based approaches, we felt that the latter have the potential to ease the verification effort and provide stronger guarantees for verification.

In a refinement-based approach one views the system as an Abstract Data Type (ADT), and begins with an abstract specification of the system's functionality in a concise and mathematically precise modelling language. This specification is then successively refined by adding implementation details to finally obtain an implementation of the system which is guaranteed to "conform" to the high-level specification. The exact meaning of what it means to conform to the specification would vary according to the notion of refinement used, but it could mean for instance that every execution of the concrete implementation can be "matched" or "simulated" by an execution of the abstract model.

There were several reasons to favour a refinement-based approach. To begin with, a refinement-based approach provides a standalone abstract specification (say $\mathcal{A}$) of the implementation (say $\mathcal{C}$), with the guarantee that certain properties proved about a client program P that uses $\mathcal{A}$ as a library (which we refer to as "P with $\mathcal{A}$" and denote by "$P[\mathcal{A}]$") also carry over for P with $\mathcal{C}$ (i.e. $P[\mathcal{C}]$). Thus, to verify that $P[\mathcal{C}]$ satisfies a certain property, it may be sufficient to check that $P[\mathcal{A}]$ satisfies the property. The latter check involves reasoning about a simpler component (namely $\mathcal{A}$) and can reduce the work of a prover by an order of magnitude [22]. Finally, a refinement-based proof is more modular and transparent, since it breaks down the task of reasoning about a complex implementation into smaller tasks, each of which is more manageable for both a human and a prover.

We chose to use a notion of refinement similar to that of VDM [8,20] and Z [2,32], but adapted to a setting in which the client program interacts in a "functional" manner with the ADT (see also [18]). The details of this theory are spelt out in [11]. We propose a methodology for phrasing the refinement conditions from this theory across different models ranging from abstract Z models to concrete C implementations.

We then used this methodology to verify the FreeRTOS scheduler. We view the scheduler-related functionality of the kernel as an ADT, specify its intended behaviour in Z, and then verify that the implementation refines the high-level ADT. We used four levels of models (two in Z and one in VCC [10] ghost code, apart from the C implementation itself), and proved successive refinements between them. Barring a few manual steps, all our refinement conditions were phrased and proved in VCC, using its very useful ghost constructs.

We found a few subtle bugs which were acknowledged by the developers of FreeRTOS [5]. These bugs were fixed with minimal changes to the source code, and the verification of the fixed code was duly completed.

A natural question a VCC expert may ask is why we chose to build a "meta-theory" of refinement on top of VCC, instead of using its internal style of data

abstractions as illustrated in [9]. In the latter idiom, to prove an assertion about a client program P with a concrete data type implementation $\mathcal{C}$, one constructs a joint data type $\mathcal{AC}$ which contains a ghost version of the data type called $\mathcal{A}$, and includes a coupling constraint between the states of $\mathcal{A}$ and $\mathcal{C}$. One then proves the assertion in $P[\mathcal{AC}]$. By the restrictions imposed by VCC on ghost code, it follows that the assertion must continue to hold on the original program $P[\mathcal{C}]$ as well. While this style of verification has many of the advantages of a refinement-based approach, it loses out in a couple of aspects. Firstly, VCC must reason about P with the *joint* structure $\mathcal{AC}$ (instead of simply $P[\mathcal{A}]$ in a refinement-based approach). While it is possible to control the portion of the joint state exposed to the prover, this requires expert knowledge of VCC. Secondly, if we want to prove a property of $P[\mathcal{C}]$, like a temporal logic specification, which is not possible with VCC, this idiom is not of much use. On the other hand, using a meta-theory of refinement, we could use VCC to prove that $\mathcal{C}$ refines $\mathcal{A}$, prove the required property about $P[\mathcal{A}]$ using non-VCC means, and then infer the property for $P[\mathcal{C}]$.

In the next few sections we describe our refinement conditions and proposed methodology, before going on to the details of the FreeRTOS verification.

2 ADT's and Refinement

The notion of refinement we use in this paper is essentially that of Z [16,32]. We briefly recall this notion before describing the variant we use.

2.1 Refinement in Z

An ADT *type* is a finite set of operation names N along with a set of "global" states G. An *abstract data type* (ADT) of type (N, G) is a structure $\mathcal{A} = (Q, \mathit{init}, \mathit{fin}, \{op_n\}_{n \in N})$, where Q is the set of states of the ADT, $\mathit{init} \subseteq G \times Q$ is an initialization operation, $\mathit{fin} \subseteq Q \times G$ is a finalization operation, and each $op_n \subseteq Q \times Q$ is a realization of operation n. All operations are allowed to be non-deterministic. A program that makes use of an ADT of type (N, G), called an (N, G)-client program, is a sequence of operations P of the form $\mathit{init}; n_1; \cdots; n_k; \mathit{fin}$, with each $n_i \in N$. Given an ADT $\mathcal{A}$ of type (N, G), the program P with $\mathcal{A}$, written $P[\mathcal{A}]$, induces a relation from G to G in a natural way, obtained by composing the operations of $\mathcal{A}$ according to the sequence given by P. Now given two ADT's $\mathcal{A}$ and $\mathcal{C}$ of type (N, G), we say that $\mathcal{C}$ *refines* $\mathcal{A}$ if, for each (N, G)-client program P, we have $P[\dot{\mathcal{C}}] \subseteq P[\dot{\mathcal{A}}]$, where the "$\dot{}$" denotes the "totalized" version of the relation in which, essentially, elements outside the domain of the relation are related to all possible elements in the target set. Thus, if $\mathcal{C}$ refines $\mathcal{A}$, then when P uses $\mathcal{C}$ all the behaviours it could observe – in terms of initial global states being transformed to final global states – are also possible behaviours of P with $\mathcal{A}$.

Let $\mathcal{A} = (Q, \mathit{init}, \mathit{fin}, \{op_n\}_{n \in N})$ and $\mathcal{C} = (Q', \mathit{init}', \mathit{fin}', \{op'_n\}_{n \in N})$ be two ADT's of type (N, G). Then a sufficient (and also necessary [16]) condition for

$\mathcal{C}$ to refine $\mathcal{A}$, called "upwards simulation" in [16], which we denote by (RC_Z), is that there should exist an "abstraction" relation $\rho \subseteq Q' \times Q$, satisfying

1. For each $g \in G$, $p' \in \mathit{init}'(g)$, and $p \in Q$ such that $(p', p) \in \rho$: we have $p \in \mathit{init}(g)$.
2. For each $p' \in Q'$ and $p \in Q$, with $(p', p) \in \rho$: we have $\mathit{fin}'(p') \subseteq \mathit{fin}(p)$.
3. For each $n \in N$, $p', q' \in Q'$, and $p \in Q$, with $p \in \mathrm{dom}(op_n)$, $(p', p) \in \rho$, and $(p', q') \in op'_n$: we have there exists $q \in Q$ such that $q \in op_n(p)$ and $(q', q) \in \rho$.

2.2 Our Notion of Refinement

We would like to work in a setting where a client program interacts with an ADT in a *functional* manner, by periodically calling operations of the ADT, each time supplying an argument and using the value returned by the operation to update its local state. Thus, we no longer need a global set of states G in an ADT type, but instead require each operation name n to have an associated *input type* I_n and an *output type* O_n. A realization op_n of operation n in an ADT with state set Q is now a subset of $(Q \times I_n) \times (Q \times O_n)$. An N-client program is a transition system in which some transitions are labelled by local actions, and some by *calls* to the ADT operations, of the form (n, a, b), representing the fact that a call of operation n with argument a returned the value b. We use a notion of refinement based on the sequences of operation calls supported by an ADT, which essentially says that an ADT $\mathcal{C}$ refines an ADT $\mathcal{A}$, written $\mathcal{C} \preceq \mathcal{A}$, if the sequences of operation calls allowed by $\mathcal{C}$ are contained in those allowed by $\mathcal{A}$. Once again, if $\mathcal{C}$ refines $\mathcal{A}$, the "behaviours" seen by a client program using $\mathcal{C}$ are guaranteed to be present when using $\mathcal{A}$. The reader is referred to [11] for the details of the theory.

In the rest of this paper, we restrict our attention to *deterministic* ADT's. One reason for this is that our case study makes use of only deterministic models and implementations. Secondly, the presentation of our methodology is simpler with this assumption, while retaining the essence of what is needed to handle the general case. We model the deterministic operations as functions, by introducing a special *exceptional value*, denoted by e, in each output type O_n, and mapping a state-input pair which was undefined by the operation, to an exceptional state E and return value e. We formally define this below.

A (deterministic) *ADT* of type N is a structure of the form

$$\mathcal{A} = (Q, E, \mathit{init}, \{op_n\}_{n \in N})$$

where Q is the set of states of the ADT, $E \in Q$ is an *exceptional* state, $\mathit{init} : I_{\mathit{init}} \to (Q \times O_{\mathit{init}})$, and each op_n is a *realisation* of the operation n given by $op_n : Q \times I_n \to Q \times O_n$ such that $op_n(E, -) = (E, e)$ and $op_n(p, a) = (q, e) \implies q = E$. Thus if an operation returns the exceptional value the ADT moves to the exceptional state E, and all operations must keep it in E thereafter.

Let $\mathcal{A} = (Q, E, \mathit{init}, \{op_n\}_{n \in N})$ and $\mathcal{C} = (Q', E', \mathit{init}', \{op'_n\}_{n \in N})$ be ADT's of type N. We say $\mathcal{A}$ and $\mathcal{C}$ satisfy condition (RC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

(init) Let $a \in I_{init}$ and let (q_a, b) and (q'_a, b') be the resultant states and outputs after an $init(a)$ and $init'(a)$ operation in $\mathcal{A}$ and $\mathcal{C}$ respectively, with $b \neq e$. Then we require that $b = b'$ and $(q'_a, q_a) \in \rho$.

(sim) For each $n \in N$, $a \in I_n$, $b \in O_n$, and $p' \in Q'$, with $(p', p) \in \rho$, whenever $p \xrightarrow{(n,a,b)} q$ with $b \neq e$, then there exists $q' \in Q'$ such that $p' \xrightarrow{(n,a,b)} q'$ with $(q', q) \in \rho$. This is illustrated in the Fig. 1 below.

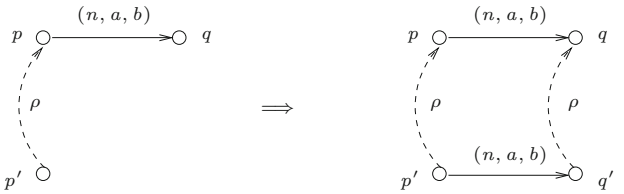


Fig. 1. Illustrating the condition (RC-sim) for refinement.

Notice that this condition is essentially a specialization of the condition (RC_Z) above for deterministic ADT's.

Finally, we will make use of a couple of properties of this notion of refinement from [11]. Firstly, refinement is *transitive*: if $\mathcal{C} \preceq \mathcal{B}$ and $\mathcal{B} \preceq \mathcal{A}$ then $\mathcal{C} \preceq \mathcal{A}$. Secondly, refinement is *substitutive*: if we have a client program $\mathcal{U}$ that implements an ADT, and itself uses a sub-ADT of type M, and if $\mathcal{B}$ and $\mathcal{C}$ are ADT's of type M such that $\mathcal{C} \preceq \mathcal{B}$. Then $\mathcal{U}[\mathcal{C}]$ refines $\mathcal{U}[\mathcal{B}]$.

3 Viewing Z and C Models as ADT's

In this section we show how to view models specified in different modelling languages as ADT's in our setting. We also phrase the refinement condition (RC) in a typical tool/environment for reasoning about these different models.

Z models. A specification $\mathcal{M}$ in the Z modelling language [32] essentially comprises the following: A finite set of variables $Var^{\mathcal{M}}$, with each $v \in Var^{\mathcal{M}}$ having a declared type (set of values) T_v. A state is a valuation s to these variables with $s(v) \in T_v$ for each $v \in Var^{\mathcal{M}}$, which satisfies a constraint $C^{\mathcal{M}}$ given as a first-order logic formula with free variables in $Var^{\mathcal{M}}$. The model has a finite set $Op^{\mathcal{M}}$ of operations. Each operation $n \in Op^{\mathcal{M}}$ has (for simplicity) a single formal input parameter x_n of type $X_n^{\mathcal{M}}$, and a single output variable y_n of type $Y_n^{\mathcal{M}}$; and a before-after-predicate $BAP_n^{\mathcal{M}}$ with free-variables in $Var^{\mathcal{M}} \cup \{x_n, y_n\} \cup Var^{\mathcal{M}'}$, where for a set of variables Var we use the convention that Var' denotes the set of variables $\{v' \mid v \in Var\}$. The set of operations $Op^{\mathcal{M}}$ includes an initialization operation called $init^{\mathcal{M}}$, whose BAP predicate is only on the input variable and primed variables (i.e. it only constrains the post-state). We say the Z model is

deterministic if for each operation $n \in Op^{\mathcal{M}}$, state p and input value $a \in X_n^{\mathcal{M}}$, we have at most one state q and output value $b \in Y_n^{\mathcal{M}}$ satisfying $BAP_n^{\mathcal{M}}(p, a, q, b)$.

A deterministic Z model like $\mathcal{M}$ above defines an ADT

$$\mathcal{A}_{\mathcal{M}} = (Q', E, init, \{op_n\}_{n \in N}),$$

of type N, where:

- N is the ADT type $Op^{\mathcal{M}}$ with $I_n = X_n^{\mathcal{M}}$ and $O_n = Y_n^{\mathcal{M}} \cup \{e\}$,
- $Q' = Q \cup \{E\}$ where Q is the set of states of $\mathcal{M}$, and E is a new exceptional state;
- the *init* operation is given by $init(a) = (q, b)$ iff $BAP_{init}^{\mathcal{M}}(a, q, b)$; and
- for each $n \in N$, we have $op_n : (Q' \times I_n) \to (Q' \times O_n)$ given by

$$op_n(p, a) = \begin{cases} (q, b) & \text{if } \exists (q, b) : BAP_n^{\mathcal{M}}(p, a, q, b) \\ (E, e) & \text{otherwise.} \end{cases}$$

Thus we view an operation as returning an exceptional value whenever it is called outside its pre-condition (namely pre_n which is the set of states and input pairs (p, a) such that there exists a state q and output b satisfying $BAP_n^{\mathcal{M}}(p, a, q, b)$).

Given two deterministic Z models $\mathcal{M}_1$ and $\mathcal{M}_2$, we say $\mathcal{M}_2$ *refines* $\mathcal{M}_1$ iff the induced ADT's $\mathcal{A}_{\mathcal{M}_2}$ and $\mathcal{A}_{\mathcal{M}_1}$ are such that $\mathcal{A}_{\mathcal{M}_2}$ refines $\mathcal{A}_{\mathcal{M}_1}$.

C implementations. We assume that an ADT implementation in C is a program P that comprises a set of global variables *Var* with each $v \in Var$ having a declared type T_v. It has a finite set of function names F, with an associated function definition $\texttt{func}_n$ for each $n \in F$, which could contain local variables. We can view P as an ADT in a natural way, as follows. A program state of P is a valuation for its global variables and local variables that are in scope, together with a location representing the statement number to be executed next. We use a special location "0" to represent the fact that an operation has completed, and the program is not in the middle of executing an operation. We call these program states with location "0" the *complete* program states of P. The states of the ADT induced by P is now the set of *complete* program states of P. As expected, we view each implementation of an operation as starting in a complete program state, taking an argument, transforming the program state – via a number of intermediate steps – from one complete state to another, and returning a value. If the function does not terminate (due to a buggy loop for example), or causes an exception (due to a null dereference for example), we view the operation as returning the exceptional value e.

Finally, we would also like to consider C implementations that have a *precondition* for each operation. We assume that the precondition for operation n is a predicate pre_n on the complete state and input of the operation. We view such a C program as inducing an ADT as defined above, except that for complete states and inputs that don't satisfy pre_n the ADT transitions to a "dead" local state.

With this view of Z and C ADT models we can phrase the refinement conditions (RC) as theorems in tools like Z/Eves or Rodin, or as `requires` and `ensures` clauses in a tool like VCC (see for instance [12]).

4 Directed Refinement Methodology

We now propose a methodology based on our theory of refinement for proving the functional correctness of an imperative language implementation $\mathcal{P}$ of an ADT-like system.

1. To begin with we view $\mathcal{P}$ as implementing an ADT of a certain type N.
2. Based on a high-level understanding of the code, and documentation like user manual and comments in code, construct an ADT $\mathcal{M}_1$ in a high-level specification language like Z, that captures the intended behaviour of $\mathcal{P}$.
3. In general $\mathcal{P}$ may use several sub-ADT's, say $\mathcal{B}_1, \ldots, \mathcal{B}_n$ of type $M_1, \ldots, M_n$ respectively, and can be viewed as $\mathcal{U}[\mathcal{B}_1, \ldots, \mathcal{B}_n]$, where $\mathcal{U}$ is an $(M_1, \ldots, M_n)$-client program, itself providing an ADT of type N. We now replace each sub-ADT implementation $\mathcal{B}_i$ by a version $\mathcal{A}_i$ of it expressed using the high-level constructs like maps of the ghost language available in tools like VCC. We refer to this abstraction $\mathcal{U}[\mathcal{A}_1, \ldots, \mathcal{A}_n]$ of the implementation as $\mathcal{P}_1$.
4. Refine $\mathcal{M}_1$ towards the implementation $\mathcal{P}_1$, via a sequence of successively refined Z models, that add increasing details of the implementation. Let $\mathcal{M}_2$ be the resulting Z model that is sufficiently "close" to $\mathcal{P}_1$. The refinement conditions for the successive Z models could be checked in Z-Eves [29] or other tools [1,26,27], or by a suitable encoding in VCC.
5. Check that $\mathcal{P}_1$ refines $\mathcal{M}_2$. We can do this by either using a ghost version $\mathcal{M}_2$ if one is available, or by directly importing the before-after predicates from $\mathcal{M}_2$ (see [12] for example), and then checking the resulting annotations in a tool like VCC. At the end of this step, we would have contracts in the form of `requires` and `ensures` annotations, for each ghost implementation $\mathcal{A}_i$ of the sub-ADT's that were used to prove that $\mathcal{P}_1$ refines $\mathcal{M}_2$.
6. Check that each sub-ADT $\mathcal{A}_i$ along with its associated precondition (from the `requires` clause of its contract), is refined by $\mathcal{B}_i$.

If these checks are successful, we can conclude using the transitivity and substitutivity property of refinement, that $\mathcal{P} = \mathcal{U}[\mathcal{B}_1, \ldots, \mathcal{B}_n] \preceq \mathcal{U}[\mathcal{A}_1, \ldots, \mathcal{A}_n] = \mathcal{P}_1 \preceq \mathcal{M}_2 \preceq \mathcal{M}_1$.

5 About FreeRTOS

In the next few sections we describe the case-study (FreeRTOS V6.1.1) on which we apply our verification methodology. FreeRTOS [14] is a real-time kernel meant for use in embedded applications that run on microcontrollers with small to mid-sized memory. It allows an application to organise itself into multiple independent tasks (or threads) that will be executed according to a priority-based preemptive scheduling policy. It is implemented as a set of API functions written in about 3,000 lines of C code, that an application programmer can include with their code and invoke as function calls. These API's provide the programmer ways to create and schedule tasks, communicate between tasks (via message queues, semaphores, etc.), and carry out time-constrained blocking of tasks.

It has been ported to 34 architectures and receives more than 100,000 downloads a year.

Figure 2 shows a simple application that uses FreeRTOS. The application creates two tasks "A1" and "B2" with priorities 1 and 2 respectively (a higher number indicates a higher priority), and starts the FreeRTOS scheduler. We use a naming convention that indicates the task's priority in its name. The scheduler then runs task B2, which immediately asks to be delayed for 2 time units. B2 is now blocked and the lower priority task A1 gets to execute. After 2 time units, B2 is ready to execute and preempts A1. This behaviour continues forever.

```
int main(void) {
  xTaskCreate(foo, "A1", 1,...);
  xTaskCreate(bar, "B2", 2,...);
  vTaskStartScheduler();
}
void foo(void* params) {
  for(;;) { }
}
void bar(void* params) {
  for(;;) {
    vTaskDelay(2);
  }
}
```

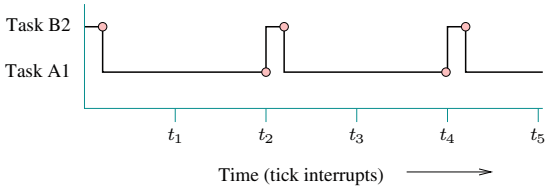


Fig. 2. An example FreeRTOS application and its timing diagram.

```
void vTaskDelay(portTickType xTicksToDelay){...
  if(xTicksToDelay > (portTickType) 0){
    xTimeToWake = xTickCount + xTicksToDelay;
    vListRemove(&(pxCurrentTCB->xGenListItem));
    listSET_LIST_ITEM_VALUE(
     &(pxCurrentTCB->xGenListItem),xTimeToWake);
    vListInsert(pxDelayedTaskList,
     &(pxCurrentTCB->xGenListItem));
    ...
  }
}
```

```
void vListInsert(xList *pxList,
                 xListItem *pxNewItem) {
  ...
  xValOfInsertion = pxNewItem->xItemValue;
  for(pxIterator = &(pxList->xListEnd);
    pxIterator->pxNext->xItemValue
                 <= xValOfInsertion;
    pxIterator = pxIterator->pxNext) {
  }
  pxNewItem->pxNext = pxIterator->pxNext;
  pxNewItem->pxNext->pxPrevious = pxNewItem;
  ...
}
```

Fig. 3. Excerpts from the `vTaskDelay` API and the `xList` operation `vListInsert`.

Figure 3 shows an excerpt from the code of the `vTaskDelay` API function. It computes the time-to-awake, removes the current task from the ready queue, updates its key value to the time-to-awake, and inserts it in the delayed queue. The last 3 steps are done using calls to a list data-structure called `xList` which is the core data-structure used in FreeRTOS. It is a circular doubly-linked list of `xListItem` nodes each of which contains a key field called `xItemValue`. Based on the invariants it satisfies an `xList` can be used as a priority queue, a FIFO queue, or a generic list. It provides 13 different operations, including enqueue in a priority queue (`vListInsert`), head of a FIFO/priority queue, and rotate left.

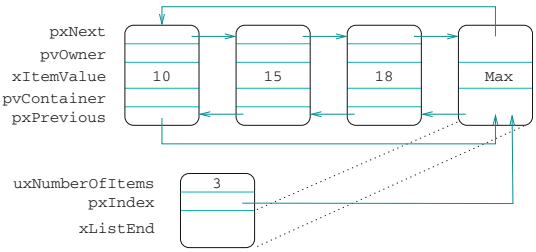


Fig. 4. An example `xList` representing a priority queue with values 10,15, and 18.

Figure 4 shows an instance of `xList`, that represents a (non-decreasing order) priority queue with item values 10,15, and 18. The head of the queue is the node pointed to by the `pxNext` field of the `xListEnd` node of the list header. The second part of Fig. 3 shows part of the `vListInsert` operation of `xList`.

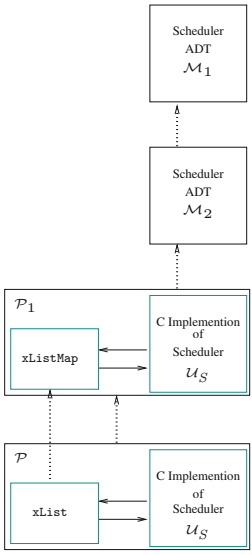
FreeRTOS is architected in a modular fashion. It has a portable part which contains compiler/processor independent code, most of it in 3 C files `tasks.c`, `queue.c`, and `list.c`. The port-specific part is present in a separate directory associated with each compiler/processor pair, and is written in C and assembly.

6 Overview of FreeRTOS Verification

We view the system corresponding to a FreeRTOS application as conceptually having two components: one is an *interpreter* for the application program, which keeps track of the local states of each task, the currently running task, etc.; the other is a component which we call the *scheduler*, whose job it is to maintain the scheduling-related state of the FreeRTOS kernel (the set of tasks created and their priorities, the contents of the ready and delayed queues, the current tick count, etc.). The interpreter component makes calls to the operations (API's) provided by the scheduler (for example `vTaskDelay(d)`), and gets back a return value which typically indicates the task to be run next. Thus, in the terminology of Sect. 2 the interpreter is a scheduler-type-client program, that uses the scheduler component as an ADT.

While in an actual execution of an application API calls could be interleaved in a non-atomic fashion (for example while the `vTaskDelay` function is running, a tick interrupt might arrive causing the `vTaskIncrementTick` to execute before the call to `vTaskDelay` finishes), we assume a limited form of preemption in which interleaving happens only at API boundaries.

In this work our interest lies in this conceptual scheduler component. We restrict ourselves to the task-related API's in the file `task.c` of the FreeRTOS code, and consider the relevant parts of this code to be the implementation $\mathcal{P}$ of the scheduler component. Our aim is to specify and verify this ADT implementation using the methodology outlined in Sects. 2 and 4.



Following the methodology, we first build a high-level deterministic model $\mathcal{M}_1$ of the scheduler in the Z specification language. This model maintains the tick count as a number bounded by *maxNumVal* and has a single delayed list. Next we observe that the scheduler implementation $\mathcal{P}$ uses a sub-ADT, namely `xList`, and thus is of the form $\mathcal{U}_S[\texttt{xList}]$ where $\mathcal{U}_S$ is a `xList`-type-client program that itself implements an ADT. We replace the sub-ADT `xList` by a ghost implementation in VCC which we call `xListMap`. Thus $\mathcal{P}_1$ is a version of the implementation of the form $\mathcal{U}_S[\texttt{xListMap}]$. Next, we bring $\mathcal{M}_1$ closer to $\mathcal{P}_1$ by adding a separate "overflow-delayed" list to store tasks whose time-to-awake is beyond *maxNumVal*. We call this model $\mathcal{M}_2$. The models $\mathcal{M}_2$ and $\mathcal{P}_1$ are very similar and hence we can import the before-after-predicates from $\mathcal{M}_2$ to $\mathcal{P}_1$, to phrase the refinement conditions. To check these conditions in VCC we come up with pre-conditions in `xListMap`. Finally we show that `xList` refines `xListMap` with its given pre-conditions. The components in the methodology used to verify FreeRTOS are shown in the figure alongside.

Provided we can check the associated verification conditions (which we address in the next section), we can conclude that $\mathcal{P}$ refines $\mathcal{M}_1$, since $\mathcal{P} = \mathcal{U}_S[\texttt{xList}] \preceq \mathcal{U}_S[\texttt{xListMap}] \preceq \mathcal{M}_2 \preceq \mathcal{M}_1$.

7 Details of Steps in the Verification of FreeRTOS

We now describe in some detail the main steps and results of our case-study. The artifacts of this project are available at www.csa.iisc.ernet.in/~deepakd/FreeRTOS/.

7.1 Z Models

We begin by describing our high-level models of the scheduler in Z. To begin with, we tried to understand the "intended" behaviour of the FreeRTOS scheduler. The main input for this understanding was the FreeRTOS user guide [4]. For some API's we had to look at the code and the comments therein to infer the meaning. We also had to re-group some of the functionality in the implementation: for instance, FreeRTOS does not have an explicit API for initialization, but initialization is done partly in the first call to `vTaskCreate` (calling a private function) and partly in `vTaskStartScheduler`; so we collected this functionality into a separate initialization API function.

Next we specified this behaviour in a Z model which we call $\mathcal{M}_1$. To represent the state of the scheduler we adopted the basic design of the FreeRTOS implementation, in particular we chose to represent the ready queue as a sequence of sequences resembling the array of FIFO queues (indexed by priorities) used in

FreeRTOS. Figure 5 shows the main elements of the data state of the scheduler and invariants on the state. The variable *maxPrio* represents the maximum priority, and *maxNumVal* represents a common bound on values like tick count and time-to-delay, as well as the maximum length of queues like the *ready* queues. These variables represent corresponding configurable constants in FreeRTOS, and are initialized in the model as shown in Fig. 5.

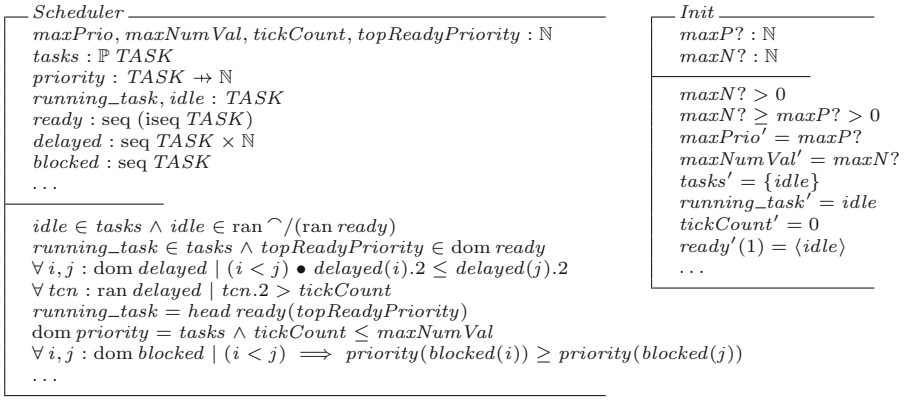


Fig. 5. Data and invariants of the Scheduler and Init schema.

Figure 6 shows the schema for the `vTaskDelay` API, for the case when there is another ready task of the top ready priority, apart from the running task. The argument `delay` to the operation is required to be at most *maxNumVal*. Since the value of tick count is bounded by *maxNumVal* the time-to-awake for the running task will be in the range $[0, 2 \cdot maxNumVal]$. The operation for increment-tick increments the value of the tick count modulo $(maxNumVal + 1)$. When it resets the tick count to 0, it reduces the time-to-awake values of the delayed tasks by $maxNumVal + 1$.

The model $\mathcal{M}_2$ refines $\mathcal{M}_1$ by adding two details from the FreeRTOS implementation. FreeRTOS maintains a separate list called "overflow-delayed" for tasks whose time-to-awake values are beyond *maxNumVal*. These tasks are stored in this list with time-to-awake values reduced by $maxNumVal + 1$. This is modelled in $\mathcal{M}_2$ by adding a corresponding list called *oDelayed*. Secondly, the set of tasks blocked on an event (like message arrival in a queue) is modeled in $\mathcal{M}_1$ as a list *blocked* in which tasks are stored in decreasing order of their priority. In FreeRTOS however they are enqueued with a key value that is the *complement* of their priority in *maxPrio*. This is done so that a single insert operation of `xList` can be used for both the delayed and blocked lists. $\mathcal{M}_2$ models this by changing the invariant on the *blocked* list.

We checked that $\mathcal{M}_2$ is a refinement of $\mathcal{M}_1$ using the refinement condition of Sect. 2. The abstraction relation is as follows: the *delayed* list in $\mathcal{M}_1$ is obtained by increasing the time-to-awake values in *oDelayed* by $maxNumVal + 1$ and

TaskDelay

$\Delta Scheduler$
$delay? : \mathbb{N}$
$delayedPrefix, delayedSuffix : \text{seq } TASK \times \mathbb{N}$
$running! : TASK$

$delay > 0 \land delay \leq maxNumVal \land running\_task \neq idle$
$\#delayed < maxNumVal$
$tail\ ready(topReadyPriority) \neq \langle\rangle \land delayed = delayedPrefix \frown delayedSuffix$
$\forall\, tcn : \text{ran } delayedPrefix \mid tcn.2 \leq (tickCount + delay?)$
$delayedSuffix \neq \langle\rangle \implies (head\ delayedSuffix).2 > (tickCount + delay?)$
$running\_task' = head\ tail\ ready(topReadyPriority)$
$ready' = ready \oplus \{ (\ topReadyPriority \mapsto tail\ ready(topReadyPriority)\ ) \}$
$delayed' = delayedPrefix \frown \langle (running\_task, (tickCount + delay?)) \rangle \frown delayedSuffix$
$\ldots$

Fig. 6. Operation schema for API *vTaskDelay* when another ready task of same priority is available.

appending it to $delayed_{\mathcal{M}_2}$. The corresponding verification conditions for the affected operations were checked using VCC by modelling the relevant parts of $\mathcal{M}_1$ and $\mathcal{M}_2$ in VCC.

7.2 Verifying that $\mathcal{P}_1$ Refines $\mathcal{M}_2$

We now address the task of showing that $\mathcal{P}_1$ (namely the FreeRTOS scheduler C code, with the `xList` library replaced by the VCC ghost library `xListMap`) refines $\mathcal{M}_2$, the Z model of the scheduler. As mentioned in Sect. 6, we define a simple list ADT using the ghost programming constructs of VCC, called `xListMap`, that provides the same intended functionality of `xList`. Figure 7 shows a part of its definition. Like `xList` it maintains a `list` of pointers to `xListItem` nodes, but as a mathematical "map" from integers to `xListItem` pointers. The component `length` records the number of items in the list. The element `type` keeps track of whether the list is meant to be a FIFO or priority queue. The figure also shows the definition of the operation `vListInsert` using a lambda construct provided by VCC's ghost language.

```
typedef struct xListMap {
  _(ghost xListItem *list[unsigned])
  _(ghost unsigned length)
  _(ghost enum xListType type)
  _(invariant length <= maxNumVal)
  _(invariant (type==PQ)==> (\forall
     unsigned i,j; (j<length && i<j)
       ==> (list[i]->xItemValue
            <= list[j]->xItemValue)))
    ...
} xListMap;
```

```
void vListInsert(xListMap *mlist, xListItem *xli)
   _(requires \wrapped(mlist))
   _(requires mlist->length < maxNumVal) {
      unsigned index;
      _(ghost mlist->list = \lambda unsigned i;
         (i<=mlist->length)?
         ((i<index)? mlist->list[i] : ((i == index)?
         xli: mlist->list[i-1])) : (xListItem*) NULL)
      _(ghost mlist->length++)
      ...
}
```

Fig. 7. Excerpts from `xListMap` and `vListInsert`. The ghost variable `index` is constrained to be the required position of `xli`.

As described in Sect. 3, to check that $\mathcal{P}_1$ refines $\mathcal{M}_2$ we directly import the before-after-conditions from $\mathcal{M}_2$ as `requires` and `ensures` conditions on the API functions in $\mathcal{P}_1$. We manually simplified these conditions to remove the existential quantifiers making use of the fact that $\mathcal{M}_2$ and $\mathcal{P}_1$ were closely related. VCC was able to check most of the annotations in the API's in $\mathcal{P}_1$, except for the `xTaskCreate` API, and a couple of other API's we mention in Sect. 7.4. The problem with `xTaskCreate` was as follows. FreeRTOS follows a convention of keeping the running task at the *end* of the ready queue corresponding to its priority. However this convention leads to inconsistencies like the following. Consider the scenario where tasks A1, B1 (both of priority 1) are ready, with A1 currently executing. By the FreeRTOS convention, the ready queue is the list $\langle \text{B1}, \text{A1} \rangle$. Now suppose A1 creates a task C1. The `xTaskCreate` function uses the `xList` operation `vListInsertEnd` to add C1 to the *end* of the queue, to get $\langle \text{B1}, \text{A1}, \text{C1} \rangle$. Thus the running task A1 is no longer at the end of the queue. If a couple of tick interrupts now arrive, causing A1 and then B1 to be preempted, it will be A1 that runs again (instead of C1!).

We chose to fix this problem in the design of FreeRTOS by following the convention of the Z models to keep the running task at the head of its ready queue. However to do this we needed to add two new functions to the `xList` (and `xListMap`) library: `list-rotate-left` and `list_GET_FIRST_ENTRY` that respectively rotate a FIFO queue by one position to the left, and return the node at the head of the list. The function `list-rotate-left` is used in the case of preemption (time slicing within tasks of the top priority), while `list_GET_FIRST_ENTRY` is used to find the next running task.

With these changes and other fixes mentioned in Sect. 7.4 VCC verifies all the API functions of $\mathcal{P}_1$. This part of the proof required considerable effort, as shown in the table of Fig. 8. As described in Sect. 3 we also need to check that the operations in $\mathcal{P}_1$ all terminate in state-input pairs that satisfy their preconditions. In $\mathcal{P}_1$ all calls to the sub-ADT namely `xListMap` terminate since they are defined declaratively. Further, the only loops present in the $\mathcal{P}_1$ code are in the call to the function `vTaskSwitchContext` whose job is to find the new top ready priority, and consequently the new running task. To verify termination of this function we used a simple ranking function (the value of the `topReadyPriority` variable), and proved that its value decreases in each iteration of the loop, using VCC.

7.3 Verifying that `xList` Refines `xListMap`

We now focus on showing that `xList` is a refinement of `xListMap`. Recall that the preconditions of the `xListMap` operations are derived from the contract (see Fig. 7) used to prove the correctness of $\mathcal{P}_1$ in the previous section. It is sufficient to consider a single pair of instances of `xList` and `xListMap`, and phrase the refinement conditions (RC) on it. We first create a joint structure containing the state components of both `xList` and `xListMap`, and their invariants. In addition we add "gluing" invariants that represent the abstraction map between

the two components. These invariants crucially use the `type` field of the `xListMap` component to say how the elements in the two lists correspond. For example, for a non-empty list of type `FIFO`, `pxIndex` points to the *end* of the list, and hence the first element of the list is the one pointed to by `pxIndex->pxNext`. For a priority queue however, the first item is the one after `xListEnd`. In addition, a node in the i-th position of `list` has its `pxNext` field pointing to the one at position $i + 1$ in `list`:

```
_(invariant ((type == FIFO) && (length > 0) && (pxIndex->pxNext != (&xListEnd)) ==>
                                                    (list[0] == pxIndex->pxNext)))
_(invariant ((type == PQ) && (length > 0)) ==> (list[0] == ((&xListEnd)->pxNext)))
_(invariant (\forall unsigned i; (i < (length-1) ==> (list[i+1] == list[i]->pxNext)))
```

Next, for each list operation we create a joint version of the operation, containing the updates for both `xListMap` and `xList`. The precondition for this operation is inherited from the `xListMap` version, and additionally requires the joint list argument to be "`wrapped`" (that the invariants on the structure hold). The `ensures` clause simply asks for the joint structure to be `wrapped` at the end and return values to be equal. All the assertions were successfully proved by VCC.

The table alongside summarises the number of lines of code (LOC) and annotation effort (LOA) in our case study. The numbers reported exclude comments and blank lines. Of the 2514 LOC in the portable code of FreeRTOS, we have verified 482 LOC mainly from the files `list.c` and `task.c`. This includes 17 core API's from `task.c` (many of the remaining 20 task API's are to do with tracing and other non-core functionality).

Z Model $\mathcal{M}_1$		Z Model $\mathcal{M}_2$		API funcs in $\mathcal{P}$		
Schemas	LOC	Schemas	LOC	Funcs	LOC	LOA
50	766	60	1239	17	361	2347

xListMap			xList		
Funcs	LOC	LOA	Funcs	LOC	LOA (`xListJoint`)
15	306	1033	15	121	1450

Fig. 8. Size of artifacts in FreeRTOS verification

7.4 Bugs Found

Apart from the previously mentioned problem with `xTaskCreate`, another related problem is that if the main program creates tasks A1 followed by B1, and then starts the scheduler, the task that runs is B1 (instead of A1). This is due to a problem with the way the `pxCurrentTCB` (the running task) is updated.

A more serious bug was in the `vTaskPrioritySet` function which changes the priority of a given task. When the given task is in the blocked queue (say waiting to receive a message from a message queue), then its priority is updated but its position in the event queue (which is a priority queue) is *not* adjusted. A similar bug exists in the `vTaskPriorityInherit` API function which is used to increase the priority of a task holding a mutex, when a higher priority task wants the mutex. The idea is that the lower priority task temporarily *inherits* the priority of the higher priority task that is waiting for a resource it is holding, so that it can complete sooner and release the resource for the higher priority task.

These functions in turn call `list_SET_ITEM_VALUE`, which however does not have the desired effect when the lower priority task is in the blocked queue. A simple fix is to implement these API's by first removing the concerned node from the blocked queue, update its priority using `list_SET_ITEM_VALUE`, and then insert it back in the queue using `vListInsert`.

We communicated these issues to the developers of FreeRTOS who acknowledged that our understanding of the intended behaviour was correct and that the said behaviours were indeed deviations [5]. They would like to make the proposed fixes provided they do not conflict with other design choices in FreeRTOS: for example a time-consuming priority-based insert operation is ok to do in a lightweight critical section where the scheduler is suspended, but *not* when interrupts are disabled. Finally, the fixes made to obtain the fully verified version of the API's involved only a small part of the code: 19 lines in the API code were modified and 7 lines added to `xList`.

8 Related Work

We discuss some of the OS verification projects in the literature that are most closely related to ours. In the design-for-verification projects, the most prominent work is the seL4 project [23], where a formally verified microkernel was developed. The scope of their work is larger than ours, addressing among other things memory allocation and interrupts. They also use a refinement-based approach to prove functional correctness of the C implementation with respect to a high-level specification, in Isabelle/HOL. The translation of the C semantics to Isabelle/HOL is validated by checking that the compiled kernel refines the translation to Isabelle/HOL [31]. In contrast, our verification – though far more modest in scope – is "post-facto," and is built on an existing code verification tool like VCC, which has a large user base and hence provides a different dimension of confidence in the verification.

Among the works in post-facto verification, the most related is the Verisoft XT project [7,33] at Microsoft, where the goal was proving the functional correctness of the Hyper-V hypervisor and PikeOS operating systems. While details of the Hyper-V effort are not publicly available (see [21,24]) PikeOS [6] is an embedded OS, similar in nature to FreeRTOS though with a few more features like virtualization. The verification uses VCC and specifications are annotations and correctness is in terms of conformance to ghost code. In contrast, we use a refinement-based approach, and as a result have a standalone abstract specification that can be used to verify clients in other environments.

In a recent and closely related piece of work, Ferreira et al. [13] prove functional correctness and memory safety of some of the FreeRTOS list and task API's, in the HIP/SLEEK verification tool. Their specifications are pre/post annotations on the API code. In contrast we verify all the list API's and the core task API's. We use an abstract specification and correctness is in terms of conformance to the abstract specification. As part of this conformance proof we prove all the functional and safety properties mentioned in [13].

Gotsman and Yang [15] propose a modular way of reasoning about preemptive kernel code by separately arguing correctness of the context-switching and the uninterrupted kernel code. We currently do not model context-switching since this is part of the "interpreter" component that we don't model, but this would be a useful approach in extending this work to a concurrent setting.

In work related to phrasing refinement conditions in code-level verifiers, the work in [17] translates refinement conditions to annotations in C code for the purpose of proving a separation property for an embedded device. Finally, in recent work [12] we propose an efficient 2-step approach to phrasing refinement checks in VCC, and evaluate it against the two approaches proposed here, on a simplified version of FreeRTOS.

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
2. Abrial, J.R., Schuman, S.A., Meyer, B.: Specification language. In: McKeag, R.M., Macnaughlen, A.M. (eds.) On the Construction of Programs, pp. 343–410. Cambridge University Press, Cambridge (1980)
3. Alkassar, E., Hillebrand, M.A., Paul, W., Petrova, E.: Automated verification of a small hypervisor. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010)
4. Barry, R.: Using the FreeRTOS Real Time Kernel - A Practical Guide (2010)
5. Barry, R.: Personal communication by email (2013)
6. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification - specification is the new bottleneck. In: SSV, pp. 18–32 (2012)
7. Beckert, B., Moskal, M.: Deductive verification of system software in the verisoft XT project. KI **24**(1), 57–61 (2010)
8. Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Berlin (1978)
9. Cohen, E.: Data abstraction in VCC. In: Broy, M., Peled, D., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 34, pp. 79–114. IOS Press, Amsterdam (2013)
10. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
11. Divakaran, S., D'Souza, D., Sampath, P., Sridhar, N., Woodcock, J.: A theory of refinement for ADTs with functional interfaces. Technical report TR-2015-4, Department of Computer Science and Automation, IISc, Bangalore (2015)
12. Divakaran, S., D'Souza, D., Sridhar, N.: Efficient refinement checking in VCC. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 21–36. Springer, Heidelberg (2014)
13. Ferreira, J.F., Gherghina, C., He, G., Qin, S., Chin, W.: Automated verification of the FreeRTOS scheduler in Hip/Sleek. STTT **16**(4), 381–397 (2014)
14. The FreeRTOS Project. www.freertos.org (Accessed on 10 April 2012)

15. Gotsman, A., Yang, H.: Modular verification of preemptive OS kernels. In: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 404–417 (2011)
16. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
17. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.D.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: 13th ACM Computer and Communications Security (CCS), pp. 346–355 (2006)
18. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Data Refinement Refined (DRAFT). Technical report, Oxford University Computing Laboratory, Oxford, UK, May 1985
19. Hoare, C., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: a manifesto. ACM Comput. Surv. **41**(4), 22:1–22:8 (2009)
20. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River (1986)
21. Klein, G.: Operating system verification – an overview. Sādhanā **34**(1), 27–69 (2009)
22. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014)
23. Klein, G., Elphinstone, K., Heiser, G., et al.: sel4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) SOSP, pp. 207–220. ACM (2009)
24. Leinenbach, D., Santen, T.: Verifying the microsoft hyper-V hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
25. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 398–418. Springer, Heidelberg (2011)
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
27. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS(LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
28. Penninckx, W., Mühlberg, J.T., Smans, J., Jacobs, B., Piessens, F.: Sound formal verification of Linux's USB BP keyboard driver. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012)
29. Saaltink, M.: The Z/EVES system. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
30. Schwarz, M.D., Seidl, H., Vojdani, V., Lammich, P., Müller-Olm, M.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: POPL 2011, pp. 93–104. ACM (2011)
31. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, 16–19 June 2013, Seattle, WA, USA, pp. 471–482. ACM (2013)
32. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River (1996)
33. Verisoft XT Project (2010). http://www.verisoftxt.de/