Efficient Refinement Checking in VCC

Sumesh Divakaran[1], Deepak D'Souza[1], and Nigamanth Sridhar[2]

[1] Indian Institute of Science, Bangalore, India
{sumeshd,deepakd}@csa.iisc.ernet.in
[2] Cleveland State University, Cleveland OH, USA
n.sridhar1@csuohio.edu

Abstract. We propose a methodology for carrying out refinement proofs across declarative abstract models and concrete implementations in C, using the VCC verification tool. The main idea is to first perform a systematic translation from the top-level abstract model to a ghost implementation in VCC. Subsequent refinement proofs between successively refined abstract models and between abstract and concrete implementations are carried out in VCC. We propose an efficient technique to carry out these refinement checks in VCC. We illustrate our methodology with a case study in which we verify a simplified C implementation of an RTOS scheduler, with respect to its abstract Z specification. Overall, our methodology leads to efficient and automatic refinement proofs for complex systems that would typically be beyond the capability of tools such as Z/Eves or Rodin.

1 Introduction

Refinement-based techniques are a well-developed approach to proving functional correctness of software systems. In a correct-by-construction approach using step-wise refinement, one begins with an abstract specification of the system's functionality, say $\mathcal{M}_1$, and successively refines it via some intermediate models, to a concrete implementation, say $\mathcal{P}_2$ in an imperative language. Similarly, in a post-facto proof of correctness, one begins with a concrete implementation $\mathcal{P}_2$, specifies its functionality abstractly in $\mathcal{M}_1$, and comes up with the intermediate models by simultaneously refining $\mathcal{M}_1$ towards $\mathcal{P}_2$ and abstracting $\mathcal{P}_2$ towards $\mathcal{M}_1$. This is depicted in Fig. 1(a). We note that it is convenient to have $\mathcal{M}_1$ specified in an abstract modelling language such as Z [16] or Event-B [1], since this gives us a concise yet readable, and mathematically precise specification of the system's behaviour, which serves as a specification of functional behaviour for users and clients of the system.

Refinement-based proofs of functional correctness have several advantages over an approach of directly phrasing and proving pre and post conditions on methods. To begin with, refinement-based approaches help to break down assertions on complex programs using successive refinement steps, leading to more modular and transparent proofs. Secondly, they provide a useful framework for verifying *clients* of a library more efficiently. In principle, one could reason about

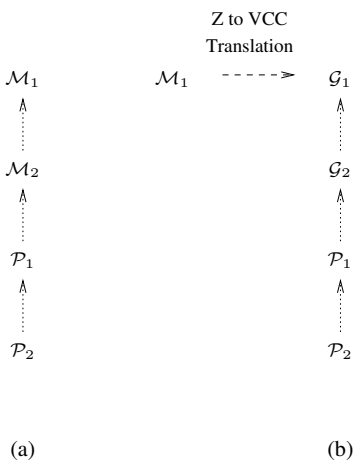


Fig. 1. (a) A typical refinement chain, with $\mathcal{M}_1$ and $\mathcal{M}_2$ being abstract models in a language like Z, and $\mathcal{P}_1$ and $\mathcal{P}_2$ being programs in a language like C. (b) The proposed translation and refinement chain, with $\mathcal{G}_1$ and $\mathcal{G}_2$ being "ghost" implementations in VCC. Dotted arrows denote the "refines" relationship.

assertions in a client program $\mathcal{C}$ that uses a concrete implementation of a library $\mathcal{P}_2$, by showing that $\mathcal{C}$ with a more abstract library $\mathcal{M}_1$ satisfies the same assertions. This could lead to considerable reductions in the verification effort as reported in [9]. In a similar way, if one replaces a library implementation by a more efficient one, one does not have to reprove certain properties of its possibly numerous clients if one has shown that the new implemention refines the old one.

There are nevertheless a couple of key difficulties faced in carrying out refinement proofs between the successive models in a refinement-based approach, in our experience. The first is that performing a refinement proof between the abstract models (such as a proof that $\mathcal{M}_1$ is refined by $\mathcal{M}_2$), is challenging because the level of automation in tools such as Z/Eves [14] and Rodin [2] is inadequate, and requires non-trivial human effort and expertise in theorem proving to get the prover to discharge the proof obligations. The second hurdle we encounter is in showing the refinement between the abstract model $\mathcal{M}_2$ and the imperative language model $\mathcal{P}_1$. The problem here is that there is no tool which understands *both* the modelling languages of $\mathcal{M}_2$ and $\mathcal{P}_1$. One way of getting around this is to "import" the before-after-predicates (BAP's) from $\mathcal{M}_2$ to $\mathcal{P}_1$, by using requires and ensures clauses that are equivalent to formulas in which the abstract state is existentially quantified away. But there are some disadvantages to this approach: (i) existential quantifications are difficult to handle for the theorem prover and can lead to excessive time requirement or can even cause the prover to run out of resources, and (ii) can be error-prone, and the equivalence should ideally be checked using a general-purpose theorem prover like Isabelle/HOL or PVS.

In this paper we propose a method of performing step-wise refinement and proving the ensuing refinement conditions, fully within the VCC toolset [6], with the aim of overcoming some of the hurdles described above. Continuing the example above, the idea is to first translate the high-level specification $\mathcal{M}_1$ into a model $\mathcal{G}_1$ in VCC's "ghost" modelling language. Next we refine $\mathcal{G}_1$ to another ghost implementation $\mathcal{G}_2$ in VCC, which will play the role of $\mathcal{M}_2$ subsequently. How does this help us to get around the problems mentioned above? The first problem of proving refinement between the abstract models is alleviated as VCC is typically able to check the refinement between ghost models like $\mathcal{G}_1$ and $\mathcal{G}_2$ efficiently and automatically. The second problem of moving from an abstract model to an imperative implementation is also addressed because we now have both $\mathcal{G}_2$ and $\mathcal{P}_1$ in a language that VCC understands, and we can then proceed to phrase and check the refinement conditions (for instance by using a joint version of $\mathcal{G}_2$ and $\mathcal{P}_1$ together) within VCC.

Our contributions in this paper are the following. First, we provide a systematic and mechanizable translation procedure to translate specifications written in a subset of the Z modelling language to a ghost specification in VCC. The fragment of Z we target is chosen to cover the case study we describe next, and essentially comprises finite sequences and operations on them. There is an inevitable blow-up of around 10x in the number of specification lines while going from Z to VCC, as VCC does not support many data-types (such as sequences) and operators that Z supports. While refining one ghost model to another ($\mathcal{G}_1$ to $\mathcal{G}_2$), the size of the model is not a problem: typically only a few aspects of the models change in each refinement step.

Secondly, we propose a two-step technique of phrasing the refinement check between ghost models and C programs in VCC that improves VCC's efficiency considerably. A naïve encoding of the refinement conditions can cause VCC to run out of memory due to the size of the model and complexity of the verification conditions. Using our two-step refinement check, VCC always terminates and leads to a reduction of over 90% in the total time taken by a naïve check, when evaluated on our case-study.

The notion of refinement, theory and methodology for coming up with intermediate models used in this paper, are all based on the work in [7], where the functional correctness of a complex existing system — the FreeRTOS open-source real-time operating system [12] — was specified and verified. Experience with that case study, where we encountered the problems mentioned above, prompted us to explore these issues in a simpler setting. In this paper we use a simpler version of the FreeRTOS scheduler, which we built ourselves for this verification exercise. This scheduler, which we call `Simp-Sched` provides the same task-related API's as FreeRTOS (like `vtaskCreate` and `vtaskDelay`), but uses a task id (a number) instead of a full Task Control Block (TCB), and an array-based list library instead of the more complex circular doubly-linked `xList` library used in FreeRTOS. We begin with the Z specification of the scheduler API's that we used in [7], and apply the techniques described above to translate the initial model to VCC, and then carry out the refinement checks between

successive models completely within the VCC platform. We carry out the refinement checks using different approaches explained in Sec. 4 and report on the comparative improvements we obtain over other approaches.

2 Preliminaries

In this section we introduce the notion of refinement we will use in this paper and a running example to illustrate some of the techniques we propose.

Consider a C implementation of a queue Abstract Data Type (ADT) (or library) shown in Fig. 2, whose functional correctness we want to reason about. This example is taken from [7]. The library uses a integer array `A` to store the elements of the queue. The variables `beg` and `end` denote positions in the array and the elements of the queue are stored starting from `beg` to `end - 1` in the array, wrapping around to the beginning of the array if necessary. The library provides the operations *init*, *enq* and *deq* to respectively initialize, enqueue, and dequeue elements from the list. The *enq* operation inserts the given element into the position `end` in the array, and the *deq* operation returns the element at the position `beg` in the array. Both operations update the `len` variable and increment the `beg`/`end` pointer modulo `MAXLEN`.

```
1: int A[MAXLEN];              11: void enq(int t) {
2: unsigned beg, end, len;     12:   if (len == MAXLEN)
3:                             13:     assert(0); /* exception */
4: void init() {               14:   A[end] = t;
5:   beg = 0;                  15:   if (end < MAXLEN-1)
6:   end = 0;                  16:     end++;
7:   len = 0;                  17:   else
8: }                           18:     end = 0;
9:                             19:   len++;
10 int deq() { ...  }          20: }
```

Fig. 2. `c-queue`: a C implementation of a Queue library.

In a refinement-based approach we would begin by specifying the functionality of the queue abstractly. We could do this in the Z specification language for instance, as shown in Fig. 3. The model specifies the state of the ADT and how the operations update the state, using the convention that primed variables denote the post-state of the operation.

We now want to show that the queue implementation *refines* the abstract Z specification. Refinement notions are typically specified in terms of a *simulation* between the concrete and abstract models. The simulation is witnessed by an *abstraction relation*. In this case, a possible abstraction relation ρ we could use is roughly as follows:

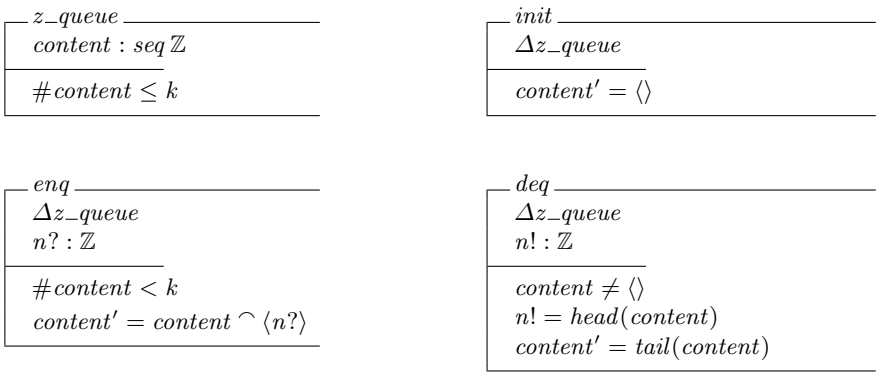


Fig. 3. A Z specification, z_queue_k, of a queue library which allows a maximum of k elements in the Queue. The notation ΔS for a Z schema S expands to the definition of S with an additional definition S' representing the post state with *primed* field names.

$$
\begin{array}{l}
\mathtt{len} = \#content \land \\
(\mathtt{beg} < \mathtt{end}) \implies \forall\, i \in \mathbb{N}.((n < \mathtt{end} - \mathtt{beg}) \implies A[\mathtt{beg} + i] = \mathtt{content}(\mathtt{beg} + i)) \land \\
(\mathtt{beg} > \mathtt{end} \lor (\mathtt{beg} = \mathtt{end} \land \mathtt{len} > 0)) \implies \\
\quad \forall\, i \in \mathbb{N}.((i < \mathtt{MAXLEN} - \mathtt{beg}) \implies A[\mathtt{beg} + i] = content(\mathtt{beg} + i)) \land \\
\quad \forall\, i \in \mathbb{N}.((i < \mathtt{end}) \implies A[i] = content(i)).
\end{array}
$$

The direction of simulation varies: a common notion of refinement used in the literature (for example in Event-B [1], and tools like Resolve [8], Dafny [11], and Jahob [17]), is to require the abstract to simulate the concrete. In this paper we use the notion from [7] where we require the *concrete* to simulate the *abstract*. We choose to use this notion as it gives us stronger verification guarantees. Nonetheless, the results we show in this paper are independent of the direction of simulation used, and apply for refinement notions with the other direction of simulation as well. We now briefly outline the notion of refinement used, and point the reader to [7] for more details.

An *ADT type* is a finite set N of *operation names*. Each operation name n in N has an associated *input type* I_n and an *output type* O_n, each of which is simply a set of values. We require that there is a special *exceptional value* denoted by e, which belongs to each output type O_n; and that the set of operations N includes a designated *initialization operation* called *init*. A (deterministic) ADT of type N is a structure of the form $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ where Q is the set of states of the ADT, $U \in Q$ is an arbitrary state in Q used as an *uninitialized* state, and $E \in Q$ is an *exceptional* state. 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 $op_n(p, a) = (q, e) \implies q = E$.

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ and $\mathcal{A}' = (Q', U', E', \{op_n\}_{n \in N})$ be ADT's of type N. We say $\mathcal{A}'$ *refines* $\mathcal{A}$ (written $\mathcal{A}' \preceq A$), 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 $init(a)$ operation in $\mathcal{A}$ and $\mathcal{A}'$ 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 visualized in Fig. 4.

The notation $p \xrightarrow{(n,a,b)} q$ denotes the fact that the ADT in state p can allow a call to operation n with argument a, return a value b, and transition to state q. We call this condition (RC).

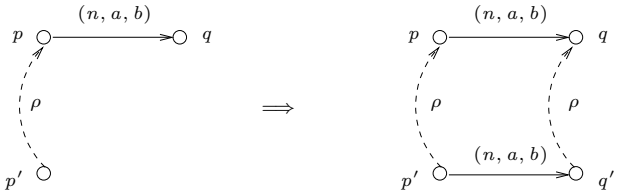


Fig. 4. Illustrating the condition (RC-sim) for refinement.

In the rest of the paper we describe our contributions with this background in mind. Our aim is to carry out the refinement checks across abstract models and C implementations, fully within the VCC tool, in a way that gets around some of the problems with checking refinements outlined in the introduction. In Sec. 3 we explain how we can systematically translate a Z model to a ghost implementation in VCC. In Sec. 4 we explain different techniques for carrying out refinement checking in VCC, beginning with the natural approaches, followed by our more efficient two-step approach. We evaluate our techniques in a case study invloving a simple RTOS scheduler called `Simp-Sched`, which we introduce in Sec. 5, and discuss the performance comparison in Sec. 6. Finally we conclude with some pointers to related work in Sec. 7.

3 Translating Z to VCC

The objective here is to translate an abstract Z model $\mathcal{M}$ into a ghost implementation $\mathcal{G}$ in VCC such that $\mathcal{G} \preceq \mathcal{M}$. The idea is to translate the state schema like that of *z_queue* in Fig. 3, comprising fields and invariants, into a structure in VCC with corresponding fields and invariants. Similarly, for the operations as well.

	Requirement	Possible Z spec.	Equivalent VCC spec.
Set	Set of elements of type T.	A_T : $\mathbb{P}$ T	_(ghost \bool A_T[T])
	Set membership.	e $\in$ A_T	A_T[e]==\true
	Set Complement.	AComp_T = T \ A_T	\forall T t; AComp_T[t]<==>!A_T[t]
	Set Union.	C_T = A_T $\cup$ B_T	\forall T t;C_T[t]<==>A_T[t]||B_T[t]
Map	Partial function from X to Y	pMap : X $\pfun$ Y	_(ghost Y pMap[X]) _(ghost \bool pMapDom[X])
	Domain restriction for maps	g = A_X $\dres$ f	\forall gDom x;(g[x]==f[x])$\wedge$ \forall X x; gDom[x]<==>(fDom[x] && A_X[x])
	Domain subtraction for maps.	g = A_X $\ndres$ f	\forall gDom x; g[x]==f[x])$\wedge$ \forall X x;(gDom[x]<==>(fDom[x] && !A_X[x])
	Range restriction for maps.	g = f $\rres$ B_Y	\forall gDom x; g[x]==f[x])$\wedge$ \forall X x;(gDom[x]<==>(fDom[x] && B_Y[f[x]])
	Range subtraction for maps.	g = f $\nrres$ B_Y	\forall gDom x; g[x]==f[x])$\wedge$ \forall X x;(gDom[x]<==>(fDom[x] && !B_Y[f[x]])
	Relational overriding for maps (f:X$\pfun$Y, g:X$\pfun$Y).	h = f $\oplus$ g	\forall hDom x;((gDom[x]==>(h[x]==g[x]))$\wedge$(!gDom[x]==> (h[x]==f[x]))$\wedge$\forall X x;(hDom[x]<==>(fDom[x]||gDom[x]))
	Containment of relational image (f:X$\pfun$Y).	f($\!|$ A_X $|\!$) $\subseteq$ A_Y	\forall X x;A_X[x]==>A_Y[f[x]]
Seq	Injective sequence of elements of type T	s : iseq T	_(ghost T sElmnts[\natural]) _(ghost \natural sIndex[T]) _(ghost \natural sLength) _(invariant \forall \natural i;(i<sLength)==> (sIndex[sElmnts[i]]==i)) _(invariant \forall T e;(sIndex[e]<sLength)==> (sElmnts[sIndex[e]]==e))
	Membership in sequence.	e $\in$ ran s	sIndex[e] < sLength
	Disjoint sequences.	ran s $\cap$ ran t = $\emptyset$	\forall T e; ((sIndex[e]<sLength==>tIndex[e]>=tLength)$\wedge$ (tIndex[e]<tLength==>sIndex[e]>=sLength))
	Sequence - containment.	ran s $\subseteq$ ran t	\forall T e; sIndex[e]<sLength==>tIndex[e]<tLength
	Concatenation for injective sequences of the same type (s, t be disjoint sequences of type T).	u = s $\frown$ t	\forall \natural i;(i<sLength)==>(uElmnts[i]==sElmnts[i]) \forall \natural i;(i<tLength)==>(uElmnts[i+sLength]==tElmnts[i]) \forall T e;(sIndex[e]<sLength)==>(uIndex[e]==sIndex[e]) \forall T e;(tIndex[e]<tLength)==>(uIndex[e]==tIndex[e]+sLength) \forall T e;((sIndex[e]>=sLength)&&(tIndex[e]>=tLength))==> (uIndex[e]==sLength+tLength]) uLength==sLength+tLength
	Filter operation for sequence of type T.	t = s $\upharpoonright$ A_T	tLength<=sLength \forall \natural i;(i<tLength)==> ((sIndex[tElmnts[i]]<sLength)&&(A_T[tElmnts[i]])) \forall \T e;((sIndex[t]<sLength)&&A_T[e])==>(tIndex[e]<tLength) \forall \natural i,j;((i<j)&&(j<tLength))==> (sIndex[tElmnts[i]]<sIndex[tElmnts[j]])
	Ectract operation for sequence of type T.	t = s $\upharpoonright$ A_T	tLength<=sLength \forall \natural i;(i<tLength)==> ((sIndex[tElmnts[i]]<sLength)&&(!A_T[tElmnts[i]])) \forall \T e;((sIndex[t]<sLength)&&!A_T[e])==>(tIndex[e]<tLength) \forall \natural i,j;((i<j)&&(j<tLength))==> (sIndex[tElmnts[i]]<sIndex[tElmnts[j]])

Table 1. Table showing the translation of Z objects to VCC objects. It gives a *suitable* encoding of Z objects in VCC which enables fast verification. If X is a set then the notation A_X denotes an arbitrary subset of X.

We translate each operation schema $\mathcal{S}^{\mathcal{M}}_{op}$ of $\mathcal{M}$, corresponding to an operation *op* in the library, into *function contracts* (in terms of `requires` and `ensures` clauses) for the corresponding implementation of the function in VCC, say $\texttt{func}^{\mathcal{G}}_{op}$. In this translation we classify the set of predicates in $\mathcal{S}^{\mathcal{M}}_{op}$ into $pre^{\mathcal{M}}_{op}$ (*precondition*) and $BAP^{\mathcal{M}}_{op}$ (*before-after predicates*). Here $pre^{\mathcal{M}}_{op}$ is the set of predicates defined over the pre-state and input of $\mathcal{S}^{\mathcal{M}}_{op}$, and the remaining predicates relating the post-state to the pre-state are denoted by $BAP^{\mathcal{M}}_{op}$.

Table 1 presents a look-up procedure for encoding various Z objects in VCC. If X is a set then the notation A_X denotes an arbitrary subset of X. The Z

```
_(ghost int content[\natural])
_(ghost \natural contentLen)
_(invariant contentLen <= MAXLEN)

void init(void)
...
_(ensures contentLen == 0)
{
  _(ghost contentLen = 0)
}
```

```
void enq(int a)
_(requires contentLen < MAXLEN)
...
_(ensures contentLen == \old(contentLen)+1)
_(ensures (\forall \natural n; (n < \old(contentLen))
  => content[n] == \old(content[n])))
{
  _(ghost content[contentLen] = a)
  _(ghost contentLen = contentLen + 1)
}
```

Fig. 5. Part of the translation of the Z specification *z_queue* to a ghost version in VCC.

objects are encoded in VCC in a way that facilitates easy proofs for the required verification conditions. This is crucial for scalability.

Fig. 5 shows excerpts from the VCC code obtained by translating the Z schema of Fig. 3.

In this paper we present only those Z constructs that are used in the Z model of our case study in Sec. 5. Nevertheless other mathematical objects in Z can be handled in a similar way.

4 Phrasing refinement conditions in VCC

In this section we describe three ways to phrase the refinement condition (RC) of Sec. 2 as annotations in VCC. The first approach — which we call the "Direct-Import" approach — is useful when the abstract library is *not* available as a ghost model in VCC. Here one directly *imports* the abstract library as code level annotations in VCC. The second is the so-called "Combined" approach, which can be applied when the abstract library is available as a ghost implementation in VCC. Finally we describe our proposed "Two-Step" approach, which can again be applied when the abstract library is available as a ghost implementation, but which VCC discharges far more efficiently.

In each of these approaches we consider the case when the abstract model $\mathcal{M}$ is specified either as a Z specification or as a VCC ghost model, and the concrete model is given as an implementation in C, say $\mathcal{P}$. For clarity, we focus here only on the (sim) condition of (RC).

4.1 Direct-Import Approach

This approach is applicable when the abstract model $\mathcal{M}$ is specified in a specification language like Z. The idea is to existentially quantify away the abstract state from a glued joint (abstract and concrete) state, and phrase this as pre/post conditions on the concrete methods. The resulting `requires` and `ensures` conditions are independent of the abstract state.

Fig. 6 shows a schematic for how one can apply the direct-import method in VCC. We use s and s' to denote respectively the pre and post states of the

abstract model, and t and t' to represent the pre and post states of the concrete model. For an operation op, $pre_{op}^{\mathcal{M}}$ represents the precondition of op in library $\mathcal{M}$. We use inv_ρ to represent the abstraction relation which relates concrete and abstract states, and BAP to represent the predicates on pre and post states describing the transitions in the respective models.

```
op(X_op x)
_(requires ∃ s : pre_op^M(s) ∧ inv_ρ(t, s))
_(ensures ∃ s, s' : BAP_op^M(s, s') ∧ inv_ρ(s, t)
      ∧inv_ρ(s', t'))
_(ensures \result = s'.y) {
  // function body
}
```

Fig. 6. Directly importing abstract library ($\mathcal{M}$) using code level annotations in VCC.

```
int deq()
  _(requires len != 0)
  _(ensures \result == \old(A[\old(beg)])
  _(ensures len == \old(len) - 1)
  _(ensures \forall unsigned i; (i < len)
      ==> ((\old(beg) < end) => A[beg+i] == \old(A[beg+i]))
      ...
{
    // function body
}
```

Fig. 7. Manually transforming the directly-imported before-after-predicates from the Z specification of Fig. 3 into the queue implementation of of Fig. 2.

Unfortunately this approach is not feasible in VCC as it is difficult for the theorem prover to handle the existential quantification. A possible way out is to manually transform the annotations to remove the existential quantification, and get an equivalent condition on the concrete state. For instance, for the queue example of Sec. 2, we could phrase the directly imported annotations by eliminating the existential quantification, as shown in Fig. 7 for the *deq* operation. The before-after predicates from the *z_queue* model of Fig. 3 are phrased as annotations over data structures in the C implementation.

This approach has two disadvantages. Firstly, the manual transformation can be error prone and the equivalence should ideally be checked in a theorem prover like PVS or Isabelle/HOL. Secondly, the invariants and preconditions need to be specified *directly* on the concrete state. This can be quite complex for both the human and the tool, especially in the presence of potentially aliased data structures.

4.2 Combined Approach

A second technique can be used to prove the refinement between two libraries when both are available as ghost or concrete implementations in VCC. The refinement condition (RC) of Sec. 2 can be phrased in VCC by using a *combined function* to update the instance of a *joint structure* which combines the fields of abstract and concrete libraries. The abstraction relation ρ can be specified as an invariant in the joint structure which we call a *gluing invariant*. To illustrate this on a simple example, consider an abstract ghost implementation $\mathcal{G}_l$ of the queue library (sec. 2) and another ghost implementation $\mathcal{G}_k$ such that $k \geq l$, where the subscript represents the maximum size of the queue. Fig. 8 shows the

joint structure to phrase the refinement condition between $\mathcal{G}_l$ and $\mathcal{G}_k$ and Fig. 9 shows the combined function to check the refinement between the abstract and concrete implementation of the *deq* operation.

```
struct {
  _(ghost int lContent[\natural])
  _(ghost \natural lLen)
  -(invariant lLen <= l)

  _(ghost int kContent[\natural])
  _(ghost \natural kLen)
    // glueing invariant
  _(invariant (lLen == kLen) &&
     (\forall \natural i;(i < lLen)
     ==> (lContent[i] == kContent[i])))
} LK;
```

Fig. 8. Joint structure combining the states of $\mathcal{G}_l$ and $\mathcal{G}_k$.

```
void deqCombined()
  _(requires LK.lLen != 0)
  _(requires (LK.lLen == LK.kLen) &&
     (\forall \natural i;(i < LK.lLen)
     ==> (LK.lContent[i] == LK.kContent[i])))
  _(ensures (LK.lLen == LK.kLen) &&
     (\forall \natural i; (i < lLen)
     ==> (LK.lContent[i] == LK.kContent[i])))
  _(ensures lOut == kOut) {
     // function body of lDeq
     // function body of kDeq
}
```

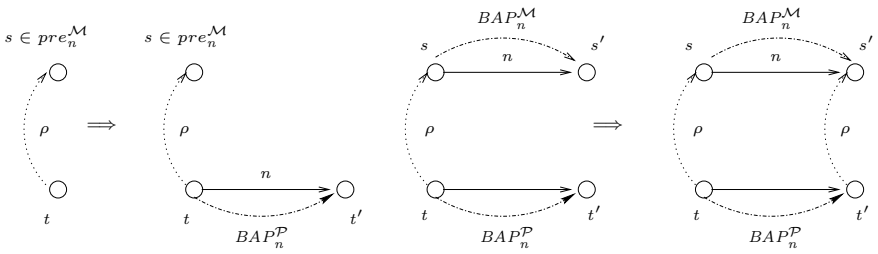
Fig. 9. Combined function to check refinement condition.

Unfortunately when the concrete model is a C program, this approach could cause the prover to take lot of time or even run out of resources. In our opinion this is mainly due to the fact that a large number of extra annotations are required when reasoning about a joint (abstract and concrete) state that are both mutable. These annotations are required as loop invariants and as function contracts, to specify that each ghost object in the system is kept unmodified by a loop or function to modify the concrete data object. For instance, there should be a loop invariant in the combined function for updating the array `A` in the queue example of Fig. 2, which essentially says that each element in the abstract map (like *lContent* above) is kept unchanged. Similar predicates are required in the function contract if functions are used to update the concrete state. In our case study (Sec. 5), the number of such annotations required in a loop or function contract is about twice the number of annotations required in the proposed Two-Step approach to prove refinement conditions.

4.3 Two-Step Approach

We now propose an efficient approach, which we call the *Two-Step* approach, which overcomes some of the difficulties of the previous two approaches. The idea is to divide the refinement check into two steps. The first step is to prove the *BAP*s for the abstract and concrete functions separately by manually supplying the *BAP*s. The second step is to prove that the output states as defined by the *BAP*s satisfies the gluing invariant. The problem with the combined approach is avoided as in Step 1, we are interested in proving *only* the concrete *BAP* as the post condition of the concrete function and hence there is no need to specify the extra set of predicates in loops and concrete function contract to specify preservation of the abstract state.

Fig. 10 illustrates the two steps of our approach. Fig. 11 shows the skeleton of the function in VCC to prove the abstract *BAP* for an library operation *op*.



(a) Step 1 for concrete model (b) Step 2

Fig. 10. Illustrating the conditions checked in the two-step approach.

```
op^A( )
  _(requires pre^A)
  _(ensures BAP^A_op)
{
   // body of  op^A
}
```

Fig. 11. Step 1 of the Two-Step approach for proving the abstract *BAP*.

```
op^P( )
  _(requires pre^A ∧ inv_ρ)
  _(decreases 0)
  _(ensures BAP^B_op)
{
   // body of  op^P
}
```

Fig. 12. Step 1 of the Two-Step approach for proving the concrete *BAP*.

Here $\mathcal{A}$ and $\mathcal{B}$ represent the abstract and concrete libraries respectively, $pre^{\mathcal{A}}$ represents the abstract precondition (like $lLen \neq 0$ for *deq*) and inv_ρ represents the GI. For instance, the *BAP* of the *deq* operation (see s=Sec. 2, Fig. 5) is $(ret = \texttt{\textbackslash old}(G.content[0]) \wedge (G.len = old(G.len) - 1) \wedge (0 \leq i < G.len \mid G.content[i] = \texttt{\textbackslash old}(G.content[i+1]))$. Fig. 12 shows the skeleton of the function in VCC to prove the concrete *BAP*. The annotation `_(decreases 0)` says that the function *terminates*.

The second step of the Two-Step approach is checking the validity of the following implication and one can use a dummy function in VCC to check its validity.

$$pre_{op}^{\mathcal{A}} \wedge inv_\rho \wedge BAP_{op}^{\mathcal{A}} \wedge BAP_{op}^{\mathcal{B}} \implies inv'_\rho \wedge ret^{\mathcal{A}} = ret^{\mathcal{B}}.$$

5 Case study: `Simp-Sched`

Here we describe our experiences with building specifications and a correct-by-construction implementation of a software system. The system we chose to work with is the scheduler component of FreeRTOS, which is a popular real-time operating system (RTOS) that is widely used in the embedded systems community both in academic settings as well as in industry [3].

The FreeRTOS scheduler API provides operations to create, schedule, and remove tasks, as well as to delay and resume task operation (see Figure 13 for a summary of task states). Application programmers can use these operations to implement application functionality using these tasks as units of behaviour. For

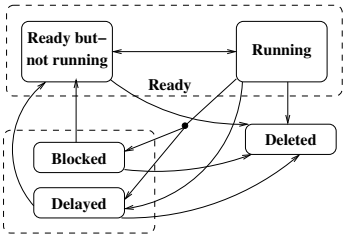


Fig. 13. Task states in FreeRTOS/`Simp-Sched`

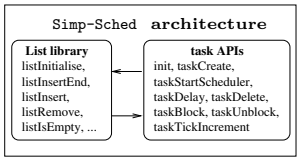


Fig. 14. Components in the scheduler implementation

our case study, we create a simplified version of the FreeRTOS scheduler, called `Simp-Sched`. Our new implementation maintains all key aspects of timing and scheduling. The simplification is based on two things:

1. Tasks in the FreeRTOS scheduler are maintained in a struct called a *task control block*, which includes pointers to function behaviour. In `Simp-Sched`, we simply use an integer *task ID* to represent the TCB of a task.
2. All task lists (ready, delayed, etc.) are maintained in a data structure called `xlist`, which is implemented as a circular doubly-linked list. In `Simp-Sched`, we replace this data structure with array implementations of the task lists.

All other aspects of the scheduler implementation are maintained. As such, the stock FreeRTOS scheduler implementation is a refinement of our `Simp-Sched` implementation. Given this, one of the key uses of our `Simp-Sched` implementation is use in a runtime monitor that can be used to identify potential scheduling inconsistencies and errors. Each API operation implementation in FreeRTOS can be instrumented to include a call to the corresponding operation in `Simp-Sched`, so that the two scheduler implementations are running in parallel.

The C implementation of `Simp-Sched` includes 769 lines of C code and 106 lines of comments [15]. The task lists are implemented as a separate library in which list is implemented using arrays in C. Fig. 14 shows the components in the `Simp-Sched` implementation with interface operations.

5.1 Refinement strategy for `Simp-Sched`

We now describe our methodology for constructing a *correct* C program from a mathematical specification of `Simp-Sched` by applying the refinement theory from [7]. The methodology involves five stages.

1. We start with a mathematical model in Z which we call $\mathcal{M}_1$ capturing the high-level functionality of `Simp-Sched`.
2. We apply our mechanizable procedure explained in Sec. 3 to translate $\mathcal{M}_1$ to a declarative model in VCC, which we call $\mathcal{G}_1$. Note that the translation guarantees that $\mathcal{G}_1 \preceq \mathcal{M}_1$.

3. We then refine $\mathcal{G}_1$ to a more concrete model $\mathcal{G}_2$ in VCC to capture some machine level requirements. For example, the system clock is unbounded in $\mathcal{G}_1$, which is not directly realizable in the C language. In $\mathcal{G}_2$ the clock value cycles in the interval [`0`, `maxNumVal`] where `maxNumVal` is the maximum value that an `unsigned int` in C can take. This change has another effect: the *delayed* tasks are maintained in a single delayed list in $\mathcal{G}_1$, which has to be broken into two lists in $\mathcal{G}_2$ to cope with the bounding of the clock value. The refinement between $\mathcal{G}_2$ and $\mathcal{G}_1$ is verified using the *combined* approach explained in Sec. 4.2.
4. Next, we refine $\mathcal{G}_2$ to $\mathcal{P}_1$ where every data object except task lists in $\mathcal{G}_2$ is implemented using executable objects and functions in C. The refinement between $\mathcal{P}_1$ and $\mathcal{G}_2$ is verified using the *Two-Step* approach explained in Sec. 4.3.
5. Finally we refine $\mathcal{P}_1$ to $\mathcal{P}_2$ where the map-based abstract implementation of the list library (`lMap`) is replaced with an array-based list implementation (`lArray`). The refinement between $\mathcal{P}_2$ and $\mathcal{P}_1$ is verified using the *Two-Step* approach explained in Sec. 4.3.

$\mathcal{P}_2$ is a C program and we conclude using the transitivity and Substitutability theorems from [7] that $\mathcal{P}_2 \preceq \mathcal{M}_1$. The verification artifacts from this case study are available at [15].

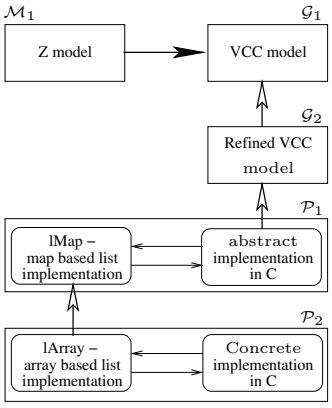


Fig. 15. Overview of the construction of `Simp-Sched`.

Model	LOA	LOC	Model	LOA	LOC
$\mathcal{M}_1$	*222*	-	$\mathcal{G}_2 \preceq \mathcal{G}_1$	*741*	*3271*
$\mathcal{G}_1$	*1580*	*1317*	$\mathcal{P}_1 \preceq \mathcal{G}_2$	*7639*	*24*
$\mathcal{G}_2$	*2287*	*1954*	$L_1 \preceq L_2$	*837*	*20*
$\mathcal{P}_1$	*293*	*609*	L_1	*602*	*240*
$\mathcal{P}_2$	-	*769*	L_2	*56*	*104*

Fig. 16. Code metrics and human effort involved. Here LOA and LOC respectively represent the number of Lines Of Annotations and number of Lines Of Code without comments and blank lines, L_1 and L_2 represent *lMap* and *lArray* respectively.

5.2 Code metrics and human effort involved

We spent two human-months to complete this work. The code metrics presented in the table in Fig. 16. Even though there are around 22500 lines of

Sl.No.	API	Time taken by VCC (in seconds)				
		Direct Import	Combined	Two-Step		
				step1	step2	Total
1.	*init*	**257.56**	**89.63**	231.01	3.52	**234.53**
2.	*taskCreate*	**357.09**	**780.88**	9.41	4.28	**13.69**
3.	*taskStartScheduler*	**10.36**	**13.95**	5.13	4.55	**9.68**
4.	*taskDelay*	**285.09**	**18773.61**	22.19	8.47	**30.66**
5.	*taskDelete*	**436**	**18391.04**	68.23	7.86	**76.09**
6.	*taskBlock*	**422.7**	**20699.25**	21.64	5.28	**26.92**
7.	*taskUnblock*	**227.06**	**16838.05**	27.44	6.02	**33.46**
8.	*listInitialise*	**2.11**	**2.7**	2.34	1.88	**4.22**
9.	*listGetNumberOfElements*	**2.02**	**2.19**	2.06	1.89	**3.95**
10.	*listIsEmpty*	**1.97**	**2.34**	3.86	2.14	**6.00**
11.	*listIsContainedIn*	**2.31**	**2.19**	2.83	4.44	**7.27**
12.	*listGetIDofFirstFIFOtask*	**3.05**	**2.3**	2.26	2.97	**5.23**
13.	*listGetIDofFirstPQtask*	**1.69**	**2.59**	2.52	4.39	**6.91**
14.	*listGetKeyOfFirstPQtask*	**1.83**	**3.08**	2.36	1.97	**4.33**
15.	*listInsertEnd*	**2.49**	**2.69**	2.19	4.52	**6.71**
16.	*listInsert*	**31.77**	**8.89**	2.77	2.22	**4.99**
17.	*listRemove*	**4447.67**	**42.7**	3.7	2.23	**5.39**
Total time taken by each technique		**6492.77**	**75658.08**			**489.94**

Table 2. Time taken by VCC to prove refinement conditions with the different techniques.

code/annotations there is only a small modification required in successive refinements and hence the size of the initial model $\mathcal{G}_1$ and Ł$_1$ model extracted from $\mathcal{G}_2$ are the important parts deciding the human effort required. The size of $\mathcal{G}_1$ and Ł$_1$ comes to 2422 lines of annotation in VCC and that is about 3 times the size of the executable code $\mathcal{P}_2$.

6 Performance Comparison

We report the time taken by VCC to prove the refinement conditions between different models in the case study. Table 2 shows the time taken under the three different approaches, namely the *Direct-Import*, *Combined*, and *Two-Step* approaches described in Sec. 4. Our *Two-Step* approach takes only 7.4% of the total time taken by the *Direct-Import* approach. The time taken by the *Combined* approach is much longer than the time taken by the *Direct-Import* approach. This is because of the presence of the abstract objects, abstract invariants, gluing relation and abstract function body in addition to the overhead involved in the *Direct-Import* approach.

7 Related work and Conclusion

As already mentioned, the work in this paper uses the foundation laid in [7], in terms of the theory of refinement and methodology used. There again, VCC is the main tool used for refinement checking: first for checking the refinement conditions between abstract Z models by translating them to VCC, and secondly for

checking the refinements between the refined Z model and the concrete implementation. However the Z to VCC translation was *partial*, as they only needed to check the refinement between the *changed* API's. As a result the approach used for phrasing the refinement conditions for the abstract to imperative implementation step was the "Direct-Import" technique described in Sec. 4.1. In contrast, in this paper we have (a) given a systematic translation from Z to VCC and (b) proposed a two-step refinement check to phrase the refinement conditions in VCC that we show leads to significant improvements in our case study.

VCC was used extensively in the Verisoft XT project [5] at Microsoft, where the goal was verification of Hyper-V hypervisor [10] and PikeOS [4] operating systems. The methodology used there appears to have been to define an abstract specification as a ghost model in VCC, and to prove conformance of the C implementation to this abstract model. However it is not clear if these works make use of a formal theory of refinement and if so, how the refinement conditions are checked in VCC.

In future work, we plan to automate the Z to VCC translation and expand the subset of the language we handle. We would also like to explore the further translation of the VCC ghost model to a simple executable implementation in C, with the aim of acting as a simulator for the model along the lines of ProZ animator [13].

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. 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. Int. J. Softw. Tools Technol. Transf. 12(6), 447–466 (Nov 2010), http://dx.doi.org/10.1007/s10009-010-0145-y
3. Barry, R.: **U**sing the FreeRTOS Real Time Kernel – A Practical Guide (2010)
4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: Cassez, F., Huuck, R., Klein, G., Schlich, B. (eds.) SSV. EPTCS, vol. 102, pp. 18–32 (2012)
5. Beckert, B., Moskal, M.: Deductive Verification of System Software in the Verisoft XT Project. KI 24(1), 57–61 (2010)
6. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs. pp. 23–42 (2009)
7. Divakaran, S., D'Souza, D., Kushwah, A., Sampath, P., Sridhar, N., Woodcock, J.: A Theory of Refinement with Strong Verification Guarantees . Tech. Rep. TR-520, Department of Computer Science and Automation, Indian Institute of Science (06 2014)
8. Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Part ii: specifying components in resolve. SIGSOFT Softw. Eng. Notes 19(4), 29–39 (Oct 1994), http://doi.acm.org/10.1145/190679.190682
9. 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)

10. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: 16th International Symposium on Formal Methods (FM 2009). Lecture Notes in Computer Science, vol. 5850, pp. 806–809. Springer, Eindhoven, the Netherlands (2009)
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR (Dakar). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
12. Ltd., R.T.E.: The FreeRTOS Real Time Operating System. `www.freertos.org` (2014)
13. Plagge, D., Leuschel, M.: Validating z specifications using the probanimator and model checker. In: Davies, J., Gibbons, J. (eds.) IFM. Lecture Notes in Computer Science, vol. 4591, pp. 480–500. Springer (2007)
14. Saaltink, M.: The Z/Eves system. In: ZUM97: Z Formal Specification Notation. pp. 72–85. Springer-Verlag (1997)
15. Efficient refinement check in VCC: Project artifacts. `www.csa.iisc.ernet.in/~deepakd/SimpSched` (2014)
16. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall (1996)
17. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. pp. 349–361. ACM (2008)