A Compositional Refinement Technique for Verifying Abstract Data Type Implementations

Sumesh Divakaran\textsuperscript{1}, Deepak D’Souza\textsuperscript{1} Anirudh Kushwah\textsuperscript{1}, Prahladavaradan Sampath\textsuperscript{2}, Nigamanth Sridhar\textsuperscript{3}, and Jim Woodcock\textsuperscript{4}

\textsuperscript{1} Indian Institute of Science, Bangalore, \{sumeshd,deepakd,anirudhkushwah\}@csa.iisc.ernet.in
\textsuperscript{2} MathWorks India, prahlad.sampath@gmail.com
\textsuperscript{3} Cleveland State University, n.sridhari@csuohio.edu
\textsuperscript{4} University of York, jim.woodcock@york.ac.uk

Abstract. We propose a methodology for verifying the functional correctness of an imperative language implementation of a system that can be modelled as Abstract Data Type (ADT). The methodology is based on a novel theory of refinement that allows us to treat both declarative and imperative implementations of ADT’s on a common ground. The theory facilitates compositional reasoning about complex implementations that may use several layers of sub-ADT’s. We apply this technique to carry out a full machine-checked proof of functional correctness of the scheduler-related functionality of FreeRTOS, a popular open-source embedded operating system.

1 Introduction

Verification of the functional correctness of complex imperative language implementations is generally acknowledged by the verification community to be a difficult problem. Despite availability of powerful tools like Boogie/Spec\# \cite{20} and VCC \cite{6} that support reasoning about Floyd-Hoare-style annotations in imperative code, success stories are rare, with the Hyper-V \cite{13} and PikeOS \cite{5} projects being among those that have met with partial success. One of the reasons for this unsatisfactory state-of-the-art in our opinion is the lack of a theory of refinement for imperative languages that admits compositional reasoning.

In this paper we propose a theory that attempts to fill this gap for Abstract Data Type (ADT) implementations. Based on this theory we propose a methodology to specify and prove the correctness of complex (but modular!) imperative language implementations of systems that can be viewed as ADT’s.

The starting point of this theory is a simple notion of refinement between deterministic ADT’s where an ADT $B$ refines an ADT $A$ iff every exception-free sequence of operations on $A$ is also allowed in $B$. By an exception we mean an undesirable behaviour like null-pointer exception or non-termination, that causes control to not return to the calling client. Thus a client of $A$ (like a C program that uses $A$ as a library) would see exactly the same behaviour with a refinement $B$ of $A$, as it would with $A$, except that in cases where $A$ exhibited an exception, $B$ may do better and return a value.

Next we consider ADT’s implemented as transition systems – which could model an ADT implementation in C – as well as ADT’s implemented as transition systems
that make use of sub-ADT’s, and show that refinement is substitutive: If an ADT implementation \( U \) uses a sub-ADT \( A \), and we replace \( A \) by a refinement \( B \) of \( A \), then the ADT implementation \( U[B] \) refines \( U[A] \).

We then propose a methodology for specifying and proving the correctness of an implementation of an ADT-like system, by specifying its behaviour in a high-level modelling language like Z [22], and using our theory of substitution to successively refine it towards the implementation. We can use tools like Z-Eves [19] and VCC, or provers like PVS [17] and Z3 [7] to phrase our refinement conditions, and obtain machine-checked proofs of our correctness claim.

In the second part of the paper we apply our method to verify the functional correctness of FreeRTOS [21] which is an open source real-time kernel for embedded applications on low-memory processors. It has been ported to 34 architectures and receives more than 100,000 downloads a year. Its portable layer comprises about 3,000 lines of C code, and is very modularly written, though with no aim of verification in mind. 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 found a few subtle bugs which had to be fixed for the verification to go through.

While VCC uses a modular proof technique in that to prove a property about a client program it uses only the contract of the sub-function (and not its code), its traditional use is not compositional. To prove that a complex sub-ADT \( B \) refines a simpler more abstract one \( A \), one would use the technology in VCC to specify \( A \) via a “ghost” implementation in the program, and show that the complex implementaton \( B \) “conforms” to it. However the resulting contract on \( B \) is a complex one that talks about the joint state of \( A \) and \( B \). Using our theory, and assuming the client program and \( B \) interact in a modular fashion, we can work with a separate (and hence simpler) contract of \( A \). It is in this sense that our approach adds compositionality to the existing verification methodology.

To sum up, our contributions in this paper are (a) a new notion of refinement which provides a strong guarantee, (b) a compositional way of reasoning about ADT implementations, and (c) a post-facto, machine-checked, proof of functional correctness of a non-trivial system. Due to space constraints some material like proofs are moved to the appendix. All artifacts of our case-study are available at [8].

2 Transition system based ADT’s and refinement

We begin with some preliminary notions. A (labeled) transition system (TS) is a structure of the form \( S = (Q, \Sigma, s, \Delta) \) where \( Q \) is a set of states, \( \Sigma \) a set of action labels, \( s \) the start state, and \( \Delta \subseteq Q \times \Sigma \times Q \) the transition relation. \( S \) induces a language of (finite) sequences of action labels along execution paths denoted \( L(S) \). We say \( S \) is deterministic if for each \( p \in Q \), whenever \( p \xrightarrow{l} q \) and \( p \xrightarrow{l'} q' \) we have \( q = q' \). We say \( S \) is closed (or has no internal choice) if for each \( p \in Q \), whenever \( p \xrightarrow{l} q \) and \( p \xrightarrow{l'} q' \) we have \( l = l' \). We use standard notation to deal with strings over an alphabet, with \( \epsilon \) denoting the empty string and \( u \cdot v \) denoting the concatenation of strings \( u \) and \( v \).
2.1 Abstract data types and transition systems

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 $init$. We fix an ADT type $N$ for the next few sections.

A (deterministic) ADT of type $N$ is a structure of the form $A = (Q, U, E, \{ op_n \}_{n \in N})$ where $Q$ is the set of states of the ADT, $U \subseteq Q$ is an arbitrary state in $Q$ used as an uninitialized state, and $E \subseteq Q$ is an exceptional state. Each $op_n$ is a realisation of the operation $n$ given by $op_n : Q \times I_n \rightarrow Q \times O_n$ such that $op_n(E, -) = (E, e)$ and $op_n(p, a) = (q, e) \implies q = E$. Further, we require that the $init$ operation depends only on its argument and not on the originating state: thus $init(p) = init(q)$ for each $p, q \in Q \setminus \{ E \}$ and $a \in I_{init}$.

As an example consider the ADT type $QT\text{ype} = \{ init, \text{enq}, \text{deq} \}$ with $I_{init} = \{ \text{nil} \}$, $O_{init} = \{ \text{ok}, \text{fail}, \text{e} \}$, $I_{\text{enq}} = \mathbb{B}$, $O_{\text{enq}} = \{ \text{ok}, \text{fail}, e \}$, $I_{\text{deq}} = \{ \text{nil} \}$, and $O_{\text{deq}} = \mathbb{B} \cup \{ \text{fail}, \text{e} \}$. Here $\mathbb{B}$ is the set of bit values $\{0, 1\}$, and $\text{nil}$ is a "dummy" argument for the operations $init$ and $\text{deq}$. Fig. 1 shows an example ADT called $Q\text{ADT}_k$ of type $QT\text{ype}$.

An $N$-client transition system is a transition system whose action labels include “calls” to an ADT of type $N$. It is of the form $S = (Q, \Sigma, s, E, \Delta)$ where

- $Q$ is a set of states, with $s \in Q$ the start state
- $\Sigma$ is a finite set of internal or local action labels. Let $\Sigma_N = \{(n, a, b) \mid n \in N, a \in I_n, b \in O_n\}$ be the set of operation call labels corresponding to the ADT type $N$. The action label $(n, a, b)$ represents a call to operation $n$ with input $a$ that returns the value $b$. Let $\Sigma$ be the disjoint union of $\Sigma_l$ and $\Sigma_N$.
- $E \subseteq Q$ is an exceptional state reached when an exceptional value is returned.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation satisfying:
  - $(p, c, E) \in \Delta$ iff $c = (n, a, e)$ for some operation $n$ and input $a$ (thus an exceptional return value leads to the exceptional state and this is the only way to reach it).
  - $(p, \text{nil}, q) \in \Delta$ implies $p \neq E$ ($E$ is a “dead” state).
  - $(p, (n, a, b), q) \in \Delta$ implies for each $b' \in O_n$, there exists a $q'$ such that $(p, (n, a, b'), q') \in \Delta$ (calls from a state are “complete” with respect to return values).

Let $S = (Q, \Sigma, s, E, \Delta)$ be an $N$-client transition system and let $A = (Q', U', \Sigma', \{ op_n \}_{n \in N})$ be an ADT of type $N$. Then we can define the transition system obtained by using $A$ in $S$, denoted $S[A]$, to be the transition system $(Q' \times Q', \Sigma, (s, U'), \Delta')$ where $\Delta' \subseteq (Q' \times Q') \times Q \times (Q' \times Q')$ is given by

$$(p, p') \xrightarrow{(n, a, k)} (q, q') \quad \text{if} \quad l \in \Sigma_l \text{ and } p \xrightarrow{l} q \quad \text{and} \quad p \xrightarrow{(n, a, b)} q \text{ and } op_n(p', a) = (q', b).$$
2.2 Refinement between ADT’s

Let $A = (Q, U, E, \{op_n\}_{n \in N})$ be an ADT of type $N$. Then $A$ induces a (deterministic) transition system $S_A = (Q, \Sigma_N, U, \Delta)$ where $\Delta$ is given by $(p, (n, a, b), q) \in \Delta$ iff $op_n(p, a) = (q, b)$. We define the language of initialised sequences of operation calls of $A$, denoted $L_{\text{init}}(A)$, to be $L(S_A) \cap \langle (\text{init}, -, -) \cdot \Sigma_N^\ast \rangle$. We say a sequence of operation calls $w$ is exception-free if no call in it returns the exceptional value $e$ (i.e. $w$ does not contain a call of the form $(-, -, e)$).

Let $A$ and $B$ be ADT’s of type $N$. We say $B$ refines $A$, written $B \preceq A$, iff each exception-free sequence in $L_{\text{init}}(A)$ is also in $L_{\text{init}}(B)$.

With reference to the example queue ADT of Fig. 1, we could define another ADT say $QADT_k$ that refines $QADT_l$ by defining the $\text{enq}$ and $\text{deq}$ operations to return $\text{fail}$ (instead of failing with an exception) when the queue is full and empty respectively. Also, $QADT_k$ refines $QADT_l$ whenever $k \geq l$.

This notion of refinement gives us the following guarantee: Let $S$ be an $N$-client transition system and let $A$ and $B$ be ADT’s of type $N$ such that $B \preceq A$. Then the transition systems $S[A]$ and $S[B]$ are such that their computation trees are “isomorphic” except that at an exception node (arising out of a call returning an exception) in $S[A]$ we have an empty subtree, while the tree rooted at the corresponding node in $S[B]$ may have a non-trivial computation tree. Thus a client transition system that is happy with $A$ will also be happy with $B$.

It follows immediately from its definition that refinement is transitive:

**Proposition 1.** Let $A$, $B$, and $C$ be ADT’s of type $N$, such that $C \preceq B$, and $B \preceq A$. Then $C \preceq A$. □

We formulate an equivalent condition for $A'$ to refine $A$, based on an “abstraction relation” that relates states of $A'$ to states of $A$. We say $A$ and $A'$ satisfy condition (RC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

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

**(sim)** For each $n \in N$, $a \in I_n$, $b \in O_n$, and $p' \in Q'$, with $(p', p) \in \rho$, $\rho \Rightarrow \rho$ 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 $(q', q) \in \rho$.

**Theorem 1** Let $A$ and $A'$ be two ADT’s of type $N$. Then $A' \preceq A$ iff they satisfy condition (RC). □

2.3 ADT transition systems and client ADT’s

We are interested in reasoning about imperative language implementations of ADT’s. Consider a program $P$ that uses an ADT library $A$ which in turn uses a library $B$ for
its implementation. Then in the terminology of the previous section \( P \) is an “\( A \)-client TS”, while in what follows \( B \) is an “ADT TS” and \( A \) a “\( B \)-client ADT TS.”

An **ADT transition system** of type \( N \) is a structure of the form \( S = (Q_c, Q_l, U, \{ \delta_n \}_{n \in N}) \) where
- \( Q_c \) is the set of “complete” states of the ADT (where an ADT operation is complete) and \( Q_l \) is the set of “incomplete” or “local” states of the ADT. The set of states \( Q \) of the ADT TS is the disjoint union of \( Q_c \) and \( Q_l \).
- \( \Sigma_i \) is a finite set of internal or local action labels. Let \( \Gamma^i_c = \{ in(a) \mid n \in N \text{ and } a \in I_n \} \) be the set of input labels corresponding to the ADT of type \( N \). The action \( in(a) \) represents reading an argument with value \( a \). Let \( \Gamma^i_r = \{ ret(b) \mid n \in N \text{ and } b \in O_n \} \) be the set of return labels corresponding to the ADT of type \( N \). The action \( ret(b) \) represents a return of the value \( b \). Let \( \Sigma \) be the disjoint union of \( \Sigma_i, \Gamma^i_c \) and \( \Gamma^i_r \).
- \( U \in Q_c \) is an uninitialized state
- For each \( n \in N \), \( \delta_n \) is a transition relation of the form: \( \delta_n \subseteq Q \times \Sigma \times Q \), that implements the operation \( n \). It must satisfy the following constraints:
  - it is deterministic (but not necessarily complete)
  - it is closed, except for the input actions in \( \Gamma^i_c \) for which it must be complete.
  - Each transition labelled by an input action in \( \Gamma^i_c \) begins from a \( Q_c \) state and each transition labelled by a return action in \( \Gamma^i_r \) ends in a \( Q_c \) state. All other transitions begin and end in a \( Q_l \) state.
  - No transition is labeled \( \text{ret} \). Thus an ADT TS cannot explicitly return the exceptional value.

Fig 2(a) shows a part of an ADT transition system induced by a C implementation of a QType ADT, shown in Fig. 3 and described later in Sec. 3.

An ADT transition system like \( S \) above induces an ADT \( A_S \) of type \( N \) given by \( A_S = (Q_c \cup \{ E \}, U, E, \{ op_n \}_{n \in N}) \) where for each \( n \in N \), \( p \in Q_c \cup \{ E \} \), and \( a \in I_n \), we have:

\[
\text{op}_n(p, a) = \begin{cases} 
(q, b) & \text{if there exists a path of the form } p \xrightarrow{\text{in}(a)} r_1 \xrightarrow{b_1} \cdots \xrightarrow{b_{k-1}} r_k \xrightarrow{\text{ret}(b)} q \text{ in } S \\
(E, e) & \text{otherwise}.
\end{cases}
\]

We say that an ADT TS \( S' \) refines another ADT TS \( S \) iff \( A_{S'} \) refines \( A_S \).

Let \( M \) and \( N \) be ADT types. Then an **\( M \)-client ADT transition system of type \( N \)** is similar to a ADT transition system of type \( N \), except that it makes calls to a sub-ADT of type \( M \). It is a structure of the form \( \mathcal{U} = (Q_c, Q_l, \Sigma_i, U, E, \{ \delta_n \}_{n \in N}) \) where \( Q_c, Q_l, \Sigma_i, \) and \( U \) are as in an ADT transition system. \( E \in Q_l \) is an exceptional state that arises when a call to a sub-ADT returns an exceptional value. Let \( \Sigma \) be the disjoint union of \( \Sigma_i, \Gamma^i_c, \Gamma^i_r \) and \( \Sigma_M \) (recall that \( \Sigma_M \) is the set of operation calls of type \( M \)). Then, for each operation \( n \) in \( N \), \( \delta_n \) is a transition relation of the form \( \delta_n \subseteq Q \times \Sigma \times Q \) satisfying similar constraints as in an ADT transition system, except that in addition we require that

- \( E \) is a dead state (i.e. \( \delta_n \) has no transition of the form \( (E, -, -) \)).
Let transition systems with multiple sub-ADT’s as well.

Theorem 2

We can extend the definition of client transition systems to allow them to have multiple sub-ADT’s. Thus an \((M_1, \ldots, M_n)-\)client transition system makes calls to ADT’s of type \(M_1, \ldots, M_n\). Thm. 2 implies that the congruence property holds for client ADT transition systems with multiple sub-ADT’s as well.

We lift the sufficient condition for refinement to ADT transition systems as well. Let \(S = (Q_c, Q_t, \Sigma_t, U, \{\delta_n\}_{n \in N})\) and \(S' = (Q'_c, Q'_t, \Sigma'_t, U', \{\delta'_n\}_{n \in N})\) be two ADT transition systems of type \(N\). We say \(S\) and \(S'\) satisfy the condition (RC-TS) if there exists a relation \(\rho \subseteq Q'_c \times Q_c\) such that:

**Fig. 2.** (a) Part of an ADT-TS representing a queue implementation from Fig.3, with solid edges representing \(\delta_{\text{init}}\) and dashed edges representing \(\delta_{\text{enq}}\); and (b) part of a \(Q\text{Type}\)-client ADT transition system realising a reschedule operation.

- \(\delta_n\) is “closed” with respect to a given \(M\)-operation and input value (thus if \(l \xrightarrow{(m,a,b)} l' \in \delta_n\) and \(l \xrightarrow{(m',a',b')} l'' \in \delta_n\), then \(m = m'\) and \(a = a'\).
- The transition relation \(\delta_{\text{init}}\) is such that along any sequence of \(\delta_{\text{init}}\) transitions from a \(Q_c\) state, there is a call to the \(\text{init}\) operation of the sub-ADT, and the first call of a sub-ADT operation is an \(\text{init}\) operation.

Fig 2(b) shows part of a \(Q\text{Type}\)-client ADT TS corresponding to \(\delta_{\text{resched}}\) for an operation \(\text{resched}\) of the C implementation of a scheduler ADT in Fig. 3.

Let \(U\) be an \(M\)-client ADT transition system of type \(N\), and \(A\) be an ADT of type \(M\). Then the ADT transition system obtained by using \(A\) in \(U\), denoted \(U[A]\), is defined in the expected way as a product of the transition systems \(A\) and \(U\). The following theorem says that refinement is “substitutive” and gives us a compositional way of reasoning about ADT implementations:

**Theorem 2** Let \(U\) be an \(M\)-client ADT transition system of type \(N\), and \(B\) and \(C\) be ADT’s of type \(M\) such that \(C \preceq B\). Then we have \(A_{\text{U}[\text{C}]} \preceq A_{\text{U}[\text{B}]}\).
(init) Let $a \in I_{\text{init}}$ and let $(q_a, b)$, with $b \neq e$, be the resultant complete state and output after an $\text{init}(a)$ operation in $S$ (thus, starting from an arbitrary complete state $q$, there is a sequence of transitions starting with $\text{in}(a)$ and ending with a $\text{ret}(b)$ in state $q_a$). Then, on doing an $\text{init}$ operation with input $a$ from any complete state in $S'$, (a) the run in $S'$ must terminate, (b) the output should be $b$, and (c) the resultant complete state $q_a'$ must be such that $(q_a', q_a) \in \rho$.

(sim) For each $n \in N$, $a \in I_n$, $b \in O_n$, $p, q \in Q$, and $p' \in Q'$, with $(p', p) \in \rho$, whenever $\delta_n$ has a terminating run in $S$ starting in state $p$ with a transition labelled $\text{in}(a)$, and ending in state $q$ with a $\text{ret}(b)$; then there must exist a complete state $q' \in Q'$ such that $\delta_n'$ has a terminating run in $S'$ starting from the state $p'$, which begins with a transition labelled $\text{in}(a)$, and ends with a $\text{ret}(b)$ in a state $q'$, with $(q', q) \in \rho$.

### 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 $M$ in the Z modelling language [22] essentially comprises the following: A finite set of variables $\text{Var}^M$, with each $v \in \text{Var}^M$ 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 $v \in \text{Var}^M$, which satisfies a constraint $C^M$ given as a first-order logic formula with free variables in $\text{Var}^M$. The model has a finite set $\text{Op}^M$ of operations. Each operation $n \in \text{Op}^M$ has (for simplicity) a single formal input parameter $x_n$ of type $X_n^M$, and a single output variable $y_n$ of type $Y_n^M$; and a before-after-predicate $\text{BAP}_n^M$ with free-variables in $\text{Var}^M \cup \{x_n, y_n\} \cup \text{Var}^{M'}$, where for a set of variables $\text{Var}$ we use the convention that $\text{Var}'$ denotes the set of variables $\{v' \mid v \in \text{Var}\}$. The set of operations $\text{Op}^M$ includes an initialization operation called $\text{init}$, whose $\text{BAP}$ 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 $n \in \text{Op}^M$, state $p$ and input value $a \in X_n^M$, we have at most one state $q$ and output value $b \in Y_n^M$ satisfying $\text{BAP}_n^M(p, a, q, b)$.

A deterministic Z model like $M$ above defines an ADT $A_M = (Q', U, E, \{ op_n \}_{n \in N})$ of type $N$, where:

- $N$ is the ADT type $\text{Op}^M$ with $I_n = X_n^M$ and $O_n = Y_n^M \cup \{e\}$,
- $Q' = Q \cup \{E\}$ where $Q$ is the set of states of $M$, $E$ is a new exceptional state, and $U$ is an arbitrary state in $Q$, and
- for each $n \in N$, we have $op_n : (Q' \times I_n) \rightarrow (Q' \times O_n)$ given by
  
  $op_n(p, a) = \begin{cases} (q, b) & \text{if } \exists(q, b) : \text{BAP}_n^M(p, a, q, b) \\ (E, e) & \text{otherwise.} \end{cases}$

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

Given two deterministic Z models $Y$ and $M$ we say $Y$ refines $M$ if the induced ADT’s $A_Y$ and $A_M$ are such that $A_Y$ refines $A_M$. We can also phrase the sufficient condition (RC) of Section 2.2 logically as detailed in Appendix B.
We assume that an ADT implementation in C is a program $P$ that comprises a set of global variables $Var$ with each $v \in Var$ of a declared type $T_v$. It has a finite set of function names $F$, with an associated function definition $func_n$ for each $n \in F$, which could contain local variables. The program $P$ can be translated to an ADT transition system along standard lines: a state of the program is a statement number $l$ along with a valuation $s$ to the variables (both global and local), and a heap state $h$ that maps heap locations to values, and a stack $k$ to model procedure calls. We assume a special statement number 0, so that states of the form $(0, s, h)$ (with $s$ mapping local variables to an uninitialised value “$u$”) form the complete states $Q_c$, while the rest form the internal states $Q_l$ (which may include values for local variables); the local actions labels are simply the statements of the program; on returning from any function in $F$ we return to the statement number 0 with the global state and heap state being unchanged and the local variables and stack being reset to $u$ and empty respectively.

\begin{verbatim}
1: typedef struct queue { 12: void task enq(task t){ 1: task reschedule(task cur){
2: task A[MAXLEN]; 13: if (q->len == MAXLEN) 2: task t;
3: int begin, end, len; 14: assert(0); /* exception */ 3: enq(cur);
4: } queue; 15: q->A[q->end] = t; 4: t = deq();
5: } queue q; 16: if (q->end < MAXLEN-1) 5: return t;
6: void init() { 17: q->end++; 6: }
7: void init() { 18: else
8: q->begin = 0; 19: q->end = 0;
9: q->end = 0; 20: q->len++;
10: q->len = 0; 21: }
11: }
12: void task enq(task t){
13: if (q->len == MAXLEN) 14: assert(0); /* exception */
15: q->A[q->end] = t; 16: if (q->end < MAXLEN-1) 17: q->end++;
18: else
19: q->end = 0;
20: q->len++;
21: }
22: task deq() { ... }
23: task deq() { ... }
\end{verbatim}

Fig. 3. (a) A C implementation of a $QType$ ADT and (b) an implementation of the reschedule operation of a Scheduler type ADT, that uses the queue as a sub-ADT. Here task is assumed to be the type B.

Fig. 3(a) shows a C implementation of a $QType$ ADT. A part of the induced ADT transition system is shown in Fig. 2(a), where we have named states as per the convention described above, with the states at the top being the complete ones. Let us now consider a simple scheduler that uses a queue sub-ADT. Fig. 3(b) shows the code for a reschedule operation that receives the id of the currently running task, pushes it into the queue, and pops and returns the task at the head of the queue as the task to be scheduled next. This scheduler can be represented as a $QType$-client transition system which calls operations of $QType$, and is depicted in Fig. 2(b).

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 function as an ADT transition system that is defined as before, except that for complete states and inputs that don’t satisfy $pre_n$ the ADT transitions to a “dead” local state.
4 Directed refinement methodology

We now propose a methodology for proving the 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 $\mathcal{N}$. This may require us to elide certain code from $\mathcal{P}$, or to transform some parts of it to reflect this view.
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 specification language like Z, that captures the intended behaviour of the implementation.
3. In general $\mathcal{P}$ may make use of several sub-ADT’s, say $\mathcal{B}_1, \ldots, \mathcal{B}_n$ of type $\mathcal{M}_1, \ldots, \mathcal{M}_n$ respectively. $\mathcal{P}$ can thus be viewed as $\mathcal{U}[\mathcal{B}_1, \ldots, \mathcal{B}_n]$, where $\mathcal{U}$ is an $(\mathcal{M}_1, \ldots, \mathcal{M}_n)$-client ADT transition system of type $\mathcal{N}$. 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 call 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 successively add 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 [19] or other tools [1, 17, 16], or by a suitable encoding in VCC.
5. Check that $\mathcal{P}_1$ refines $\mathcal{M}_2$. We can do this by manually importing the before-after predicates from $\mathcal{M}_2$, as described in Sec. B.2, and then checking the associated verification conditions in a tool like VCC. At the end of this step, we would have contracts in the form of requires and ensures predicates, for the ghost implementations of the sub-ADT’s, that were used to prove that $\mathcal{P}_1$ refines $\mathcal{M}_2$.
6. Take each sub-ADT $\mathcal{A}_i$ along with its associated precondition (from the requires clause of its contract), and check that it is refined by $\mathcal{B}_i$.

If these checks are successful, we can conclude using Prop. 1 and Thm. 2, that $\mathcal{P} = \mathcal{U}[\mathcal{B}_1, \ldots, \mathcal{B}_n] \preceq \mathcal{U}[\mathcal{A}_1, \ldots, \mathcal{A}_n] = \mathcal{P}_1 \preceq \mathcal{M}_2 \preceq \mathcal{M}_1$. We note that the implementation $\mathcal{P}$ may be incorrect and we would need to work with a suitably fixed version for proofs to go through.

5 About FreeRTOS

In the next few sections we describe the case-study (FreeRTOS V6.1.1) on which we apply our verification method. FreeRTOS [21] 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 Application Programmer Interface (API) functions written in C, 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.
Fig. 4 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.

```c
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. 4. An example FreeRTOS application and its timing diagram.](image)

Fig. 5(a) 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. The figure alongside 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. Fig. 5(b) 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 C files task.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, which task is currently running, 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 Sec. 2 the interpreter is a client transition system, 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 of the scheduler component. Our aim is to specify and verify this ADT implementation using the methodology outlined in Secs. 2 and 4.

Following the methodology, we first build a high-level deterministic model $M_1$ 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 $P$ uses a sub-ADT, namely xList, and thus is of the form $U_x[xList]$ where $U_x$ is a xList-type-client ADT transition system. We replace the sub-ADT xList by a ghost implementation in VCC which we call xListMap. Thus $P_1$ is a version of the implementation of the form $U_x[xListMap]$. Next, we bring $M_1$ closer to $P_1$ by adding a separate “overflow-delayed” list to store tasks whose time-to-awake is beyond maxNumVal. We call this model $M_2$. The models $M_2$ and $P_1$ are very similar and hence we can import the before-after-predicates from $M_2$ to $P_1$, 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 Fig. 6.
Provided we can check the associated verification conditions (which we address in the next section), we can conclude that \( P \) refines \( M_1 \), since \( P = U_S[xList] \preceq U_S[xListMap] \preceq M_2 \preceq M_1 \).

7 Details of steps in verification of 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 [3]. For some APIs 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 \( M_1 \). 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. Fig. 7 shows the main elements of the data state of the scheduler and invariants on the state. The variables \( maxPrio \) and \( maxNumVal \) represent respectively the maximum priority and a common bound on values like tick count and time-to-delay. These variables represent corresponding configurable constants in FreeRTOS, and are initialized in the model as shown in Fig. 7.

\[
\begin{align*}
\text{Scheduler} & \\
\text{maxPrio, maxNumVal, tickCount, topReadyPriority : } & \\
\text{tasks : } & \\
\text{priority : TASK } & \rightarrow \mathbb{N} \\
\text{running_task, idle : TASK } & \\
\text{ready : seq (iset TASK) } & \\
\text{delayed : seq TASK } & \times \mathbb{N} \\
\text{blocked : seq TASK } & \\
\ldots & \\
\text{idle } & \in \text{tasks } \wedge \text{idle } \in \text{ran } \wedge (\text{ran } \setminus \text{ready}) \\
\forall i, j : & \text{dom delayed } \wedge (i < j) \bullet \text{delayed}(i).2 \leq \text{delayed}(j).2 \\
\forall tcn : & \text{ran delayed } \wedge tcn.2 > \text{tickCount} \\
\text{running_task = head ready(topReadyPriority) } & \\
\text{dom priority = tasks } \wedge \text{tickCount } \leq \text{maxNumVal} \\
\forall i, j : & \text{dom blocked } \wedge (i < j) \Rightarrow \text{priority}(\text{blocked}(i)) \geq \text{priority}(\text{blocked}(j)) \\
\ldots & \\
\text{Inst} & \\
\text{maxP? : N } & \\
\text{maxN? : N } & \\
\text{maxN? > 0 } & \\
\text{maxN? } \geq \text{maxP? } & > 0 \\
\text{maxP? } = \text{maxP? } & \\
\text{maxNumVal } = \text{maxN? } & \\
\text{tasks' } = \{ \text{idle} \} \\
\text{running_task' } = \text{idle} \\
\text{tickCount' } = 0 \\
\text{ready' }(1) = \{ \text{idle} \} \\
\ldots &
\end{align*}
\]

Fig. 7. Data and invariants of the Scheduler and Init schema.

Fig. 8 in App. C shows the schema for the \( \text{vTaskDelay} \) API, for the case when there is another ready task of the top ready priority, apart from the running task. The argument \( \text{delay} \) to the operation is required to be at most \( \text{maxNumVal} \). Since the value of tick count is bounded by \( \text{maxNumVal} \) the time-to-awake for the running task will be in the range \([0, 2 \cdot \text{maxNumVal}]\). The operation for increment-tick increments the value of the tick count modulo \( (\text{maxNumVal} + 1) \). When it resets the tick count to 0, it reduces the time-to-awake values of the delayed tasks by \( \text{maxNumVal} + 1 \).
The model $M_2$ refines $M_1$ 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 $\text{maxNumVal}$. These tasks are stored in this list with time-to-awake values reduced by $\text{maxNumVal} + 1$. This is modelled in $M_2$ by adding a corresponding list called $\text{oDelayed}$. Secondly, the set of tasks blocked on an event (like message arrival in a queue) is modeled in $M_1$ 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 $\text{maxPrio}$. This is done so that a single insert operation of $\text{xList}$ can be used for both the delayed and blocked lists. $M_2$ models this by changing the invariant on the blocked list.

We checked that $M_2$ is a refinement of $M_1$ using the refinement condition of Sec. 2. The abstraction relation is as follows: the delayed list in $M_1$ is obtained by increasing the time-to-awake values in $\text{oDelayed}$ by $\text{maxNumVal} + 1$ and appending it to $\text{delayed}_{M_2}$. The corresponding verification conditions for the affected operations were checked using VCC by modelling the relevant parts of $M_1$ and $M_2$ in VCC.

### 7.2 Verifying that $P_1$ refines $M_2$

We now address the task of showing that $P_1$ (namely the FreeRTOS scheduler C code, with the $\text{xList}$ library replaced by the VCC ghost library $\text{xListMap}$) refines $M_2$, the Z model of the scheduler. As mentioned in Sec.6, we define a simple list ADT using the ghost programming constructs of VCC, called $\text{xListMap}$, that provides the same intended functionality of $\text{xList}$. Fig. 10 in App. C shows a part of its definition. Like $\text{xList}$ it maintains a list of pointers to $\text{xListItem}$ nodes, but as a mathematical “map” from integers to $\text{xListItem}$ pointers. The component $\text{length}$ records the number of items in the list. The element $\text{type}$ keeps track of whether the list is meant to be a FIFO or priority queue. The figure also shows the definition of the operation $\text{vListInsert}$ using a lambda construct provided by VCC’s ghost language.

As described in Sec. 2, to check that $P_1$ refines $M_2$ we directly import the before-after-conditions from $M_2$ as requires/ensures conditions on the API functions in $P_1$. In App. C, Fig. 11 shows the annotations for the $\text{vTaskDelay}$ API, corresponding to the case shown in the $M_2$ schema of Fig. 9. VCC was able to check most of the annotations in the API’s in $P_1$, except for the $\text{xTaskCreate}$ API, and a couple of other API’s we mention in Sec. 7.4. The problem with $\text{xTaskCreate}$ 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 ⟨$B1$, $A1$⟩. Now suppose $A1$ creates a task $C1$. The $\text{xTaskCreate}$ function uses the $\text{xList}$ operation $\text{ListInsertEnd}$ to add $C1$ to the end of the queue, to get ⟨$B1$, $A1$, $C1$⟩. 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 $\text{xList}$ (and $\text{xListMap}$) library: $\text{list-rotate-left}$ and $\text{list_GET_FIRST_ENTRY}$ that respectively rotate a FIFO queue by one position to the left, and return the node at the head of the list. The function $\text{list-rotate-left}$ is used in the case of preemption (time slicing within tasks of the top priority), while $\text{list_GET_FIRST_ENTRY}$ is used to find the next running task.

With these changes and other fixes mentioned in Sec. 7.4 VCC verifies all the API functions of $P_1$. This part of the proof required considerable effort, as shown in Tab. 7.3. As described in Sec. 2 we also need to check that the operations in $P_1$ all terminate in state-input pairs.
that satisfy their preconditions. In \( P_1 \) all calls to the sub-ADT namely \( \text{xListMap} \) terminate since they are defined declaratively. Further, the only loops present in the \( P_1 \) code are in the call to the function \( \text{vTaskSwitchContext} \) whose job it 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 \( \text{topReadyPriority} \) variable), and proved that its value decreases in each iteration of the loop, using VCC.

### 7.3 Verifying that \( \text{xList} \) refines \( \text{xListMap} \)

We now focus on showing that \( \text{xList} \) is a refinement of \( \text{xListMap} \) based on the methodology described in Section B.2. Recall that the preconditions of the \( \text{xListMap} \) operations are derived from the contract (see Fig. 10) used to prove the correctness of \( P_1 \) in the previous section. It is sufficient to consider a single pair of instances of \( \text{xList} \) and \( \text{xListMap} \), and phrase the refinement conditions (RC-TS) on it. We first create a joint structure containing the state components of both \( \text{xList} \) and \( \text{xListMap} \), and their invariants. In addition we add “gluing” invariants that represent the abstraction map between the two components. These invariants crucially use the \( \text{type} \) field of the \( \text{xListMap} \) component to say how the elements in the two lists correspond. For example, for a non-empty list of type \( \text{FIFO} \), then \( \text{pxIndex} \) points to the \textit{end} of the list, and hence the first element of the list is the one pointed to by \( \text{pxIndex} \rightarrow \text{pxNext} \).

For a priority queue however, the first item is the one after \( \text{xListEnd} \). In addition, a node in the \( i \)-th position of \( \text{list} \) has its \( \text{pxNext} \) field pointing to the one at position \( i+1 \) in \( \text{list} \):

\[
(\text{invariant } ((\text{type} == \text{FIFO}) \land (\text{length} > 0) \land (\text{pxIndex} \rightarrow \text{pxNext}) != (\text{xListEnd}) ) \implies (\text{list}[0] == \text{pxIndex} \rightarrow \text{pxNext}))
\]

\[
(\text{invariant } ((\text{type} == \text{PQ}) \land (\text{length} > 0)) \implies \text{list}[0] == ((\text{xListEnd}) \rightarrow \text{pxNext}))
\]

\[
(\text{invariant } \forall \text{unsigned } i; \ldots (i < (\text{length}-1) \implies (\text{list}[i] == \text{list}[i+1] \rightarrow \text{pxNext}))
\]

Next, for each list operation we create a joint version of the operation, containing the updates for both \( \text{xListMap} \) and \( \text{xList} \). The precondition for this operation is inherited from the \( \text{xListMap} \) version, and additionally requires the joint list argument to be \textit{wrapped} (that the invariants on the structure hold). The \texttt{ensures} clause simply asks for the joint structure to be \textit{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 and annotation effort 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 \texttt{list.c} and \texttt{task.c}. This includes 17 core API’s from \texttt{task.c} (many of the remaining 20 task API’s are to do with tracing and other non-core functionality).

### 7.4 Bugs found

Apart from the previously mentioned problem with \( \text{xTaskCreate} \), 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 \( \text{pxCurrentTCB} \) (the running task) is updated.

A more serious bug was in the \( \text{vTaskPrioritySet} \) 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 \textit{not} adjusted. A similar bug exists in the \( \text{vTaskPriorityInherit} \) 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 [2]. 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 and Conclusion

We begin with the work on ADT refinement that is most closely related to ours. Kapur [11] proposes a behavioural and algebraic notion of ADT’s, but the emphasis is on proving properties about them rather than refinement. The notion of refinement of Liskov and Wing [14], and Event-B [1] is based on a gluing relation between the concrete and abstract states, unlike ours which is more “semantic” being based on the sequences of operations allowed. Furthermore, in their notion the abstract simulates the concrete, while in ours it is the opposite. The notion of refinement in Z [22] and of [9], is closest to our notion when restricted to deterministic ADT’s. However they don’t consider transition system implementemntations of ADT’s and hence lack the associated substitutivity results that are essential in our setting.

Coming now to practical verification work, we first consider the design-for-verification projects. The most prominent work here is the seL4 project [12], where a formally verified micro kernel was developed. The scope of their work is larger than ours, addressing among other things memory allocation and interrupts. They also show correctness of the C implementation with respect to a high-level specification in Isabelle/HOL. However, the notion of refinement they use is again based on a gluing relation that ensures that the abstract model simulates the concrete implementation. This gives a weaker guarantee than ours, since a concrete implementation that does nothing would also refine the abstract. Finally we use existing tools like VCC, which are more reliable with a larger user base, than hand-coded translation of C semantics into Isabelle/HOL. Another related work is ExpressOS [15] which develops a security-verified operating system for mobile devices. They verify lightweight properties like memory/storage insulation rather than functional correctness.

Among the work on post-facto verification, the most related is the Verisoft XT project [5] 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 [13]) PikeOS [4] is an embedded OS, similar in nature to FreeRTOS. The verification uses VCC and specifications are annotations and correctness is in terms of conformance to ghost code. In contrast, we have an abstract specification, give a clear guarantee in terms of conforming to the abstract model, and provide a theory that facilitates compositional verification. In [18] the authors verify properties like data-race freedom of a Linux USB keyboard driver using Verifast, but do not address functional correctness.
References


2. Richard Barry. personal communication.


A Proofs of Theorems

Proof of Theorem 1:

Proof. Let \( \mathcal{A} = (Q, U, E, \{ op_n \}_{n \in \mathbb{N}}) \) and \( \mathcal{A}' = (Q', U', E', \{ op'_n \}_{n \in \mathbb{N}}) \) be two ADTs of type \( N \), and \( \rho \subseteq Q' \times Q \) an abstraction relation, such that \( \mathcal{A} \) and \( \mathcal{A}' \) satisfy condition (RC) wrt \( \rho \).

We prove that for any states \( p, q \in Q \) and \( p' \in Q' \), if \( p \xrightarrow{w} q \) in \( \mathcal{A} \) for an initialized error-free sequence of operation calls \( w \), then there exists a state \( q' \in Q' \) such that \( p' \xrightarrow{w} q' \) in \( \mathcal{A}' \) and \( (q', q) \in \rho \).

Let \( w = (\text{init}, a, b) \cdot u \). The proof proceeds by induction on the length of \( u \). For the base case, \( u = \epsilon \). By part (init) of the condition (RC), we know that there exists \( q' \in Q' \) such that \( p' \xrightarrow{(\text{init}, a, b)} q' \) and \( (q', q) \in \rho \), and we are done.

For the induction step, suppose \( u = u' \cdot (n, a', b') \). Let \( r \in Q \) such that \( p \xrightarrow{(\text{init}, a, b)} r \xrightarrow{(n, a', b') \cdot u'} q \). Then by the induction hypothesis, we have \( r' \in Q' \) such that \( p' \xrightarrow{(\text{init}, a, b)} r' \) and \( (r', r) \in \rho \). Now by part (sim) of the condition (RC), we know that there exists a \( q' \in Q' \) such that \( r' \xrightarrow{(n, a', b')} q' \) and \( (q', q) \in \rho \). Thus we have \( p' \xrightarrow{(\text{init}, a, b)} w' \xrightarrow{(n, a', b')} q' \) with \( (q', q) \in \rho \). This completes the proof of our claim, and the only-if direction follows.

(If direction): Conversely suppose \( \mathcal{A}' \preceq \mathcal{A} \). Let \( \rho \subseteq Q' \times Q \) be defined as follows: \( (q', q) \in \rho \) iff there exists an exception-free initial sequence of operations \( w \) such that \( U \xrightarrow{w} q \) in \( \mathcal{A} \) and \( U' \xrightarrow{w} q' \) in \( \mathcal{A}' \). We claim that \( \mathcal{A}' \) and \( \mathcal{A} \) satisfy condition (RC) with respect to this \( \rho \). For the (init) part, suppose \( p \xrightarrow{(\text{init}, a, b)} q \) in \( \mathcal{A} \). Then since \( \mathcal{A}' \) refines \( \mathcal{A} \), we must have \( p' \xrightarrow{(\text{init}, a, b)} q' \) for some \( q' \in Q' \). Also, by definition of \( \rho \), we have \( (q', q) \in \rho \). For the (sim) part, suppose \( p' \in \mathcal{A}' \), and \( p \xrightarrow{(n, a, b)} q \) in \( \mathcal{A} \). By definition of \( \rho \), we know that there exists an exception-free initial sequence \( w \) such that \( U \xrightarrow{w} p \) and \( U' \xrightarrow{w} p' \). Since \( p \xrightarrow{(n, a, b)} q \) by assumption, we have \( U \xrightarrow{w} q \) by definition of \( \rho \). Hence, \( q' \in Q' \) and \( (q', q) \in \rho \), and hence also that \( p' \xrightarrow{(n, a, b)} q' \). This implies that \( (q', q) \in \rho \), and we are done.

Proof of Theorem 2:

Proof. We prove the theorem by contradiction. For the purpose of contradiction, suppose that \( \mathcal{A}_{l[C]} \not\preceq \mathcal{A}_{l[B]} \). Then there exists \( w \in L_{\text{init}}(\mathcal{A}_{l[B]}) \) such that \( w \notin L_{\text{init}}(\mathcal{A}_{l[C]}) \). Let \( w = (n_1, a_1, b_1) \cdot (n_2, a_2, b_2) \cdots (n_k, a_k, b_k) \) and let \( w_0 = (n_1, a_1, b_1) \cdots (n_i, a_i, b_i) \), \( 0 \leq i < k \) be the maximal prefix of \( w \) present in \( L_{\text{init}}(\mathcal{A}_{l[C]}) \). Where \( w_0 \) is well defined with \( w_0 = \epsilon \). Let \( p \) be the state in which the path corresponding to \( w_0 \) ends in \( \mathcal{A}_{l[C]} \). In \( \mathcal{A}_{l[C]} \), \( \rho_{n_{i+1}}(p, a_{i+1}) = (q', b'_{i+1}) \) and \( b'_{i+1} \neq b_{i+1} \) (because \( w_0 \) is the maximal prefix of \( w \) in \( L_{\text{init}}(\mathcal{A}_{l[C]}) \)).

Now consider the implementations of \( n_{i+1} \) say \( \delta_{n_{i+1}} \) and \( \delta'_{n_{i+1}} \) in \( \mathcal{U}[B] \) and \( \mathcal{U}[C] \) respectively. Let

\[
\delta_{n_{i+1}} = p \xrightarrow{\text{init}(a_{i+1})} r_1 \cdots r_k \xrightarrow{\text{ret}(b_{i+1})} q \quad \text{and} \quad \\
\delta'_{n_{i+1}} = p \xrightarrow{\text{init}(a_{i+1})} l_1 \cdots l_k \xrightarrow{\text{ret}(b'_{i+1})} q'.
\]

Let \( x = \text{init}(a_{i+1}) \cdot l_2 \cdots l_k \cdot \text{ret}(b_{i+1}) \) and \( x' = \text{init}(a_{i+1}) \cdot l'_2 \cdots l'_k \cdot \text{ret}(b'_{i+1}) \) be the strings of transition labels in \( \delta_{n_{i+1}} \) and \( \delta'_{n_{i+1}} \) respectively. Let \( y \) be the maximal prefix of \( x \) such that \( x = y \cdot (m, a, b) \cdot z \) and \( x' = y \cdot (m, a, b') \cdot z' \). Note that \( x \) and \( x' \) can differ only in the operation calls made to the ADT of type \( m \).

Let \( u \) be the initialized sequence of type \( M \) operation calls made in \( y \). Now we have \( v = u \cdot (m, a, b) \) in \( L_{\text{init}}(B) \) and \( u \cdot (m, a, b') \) in \( L_{\text{init}}(C) \). Thus there exists a \( v \) in \( L_{\text{init}}(B) \) such that \( v \) is not in \( L_{\text{init}}(C) \) (because \( C \) is deterministic). But this contradicts the assumption that \( C \preceq B \). Hence done. \( \square \)
B Phrasing refinement conditions

B.1 Refinement conditions for Z models

We can also phrase the sufficient condition (RC) of Section 2.2 for the Z models Y and M logically as follows:

- \( Op^Y = Op^M \), and input/output types for each \( n \in Op^M \) match (i.e. \( X_n^Y = X_n^M \) and \( Y_n^Y = Y_n^M \)).
- There exists a predicate \( \rho \) on \( \text{Var}^Y \cup \text{Var}^M \) that satisfies the following conditions:
  - For each \( n \in Op^M \), for each \( a \in X_n^M \text{init} \), for each \( q \in Q^M \), \( b, b' \in Y_n^M \text{init} \), and \( q' \in Q^Y \):
    \[
    (BAP^M_{\text{init}}(a, q, b) \land BAP^Y_{\text{init}}(a, q', b')) \implies b = b' \land \rho(q', q).
    \]
  - and for each \( n \in Op^M \), for each \( a \in X_n^M \), for each \( p, q \in Q^M \), \( b \in Y_n^M \), and \( p' \in Q^Y \):
    \[
    [C^M(p) \land C^Y(p') \land \rho(p', p) \land BAP^M_n(p, a, q, b)] \implies \exists q' \in Q^Y \text{ s.t. } BAP^n_Y(p', a, q', b) \land \rho(q, q').
    \]

Such a condition can be checked in a theorem prover for Z like Z-Eves or even by a suitable translation in VCC.

B.2 Refinement conditions for C models

In this section we discuss how one could phrase the sufficient conditions for refinement as annotations in a C implementation.

We begin with considering a Z model \( \mathcal{M} \) and a candidate C implementation as its refinement. Within this, we first consider the case when the Z model is closely aligned to the C implementation. In this case we could directly “import” the conditions from the Z model and phrase them as preconditions in the C program. We denote these imported conditions with a superscript “\( \mathcal{M} \rightarrow C \)”, below. We use the \texttt{requires} annotation (which specifies a condition on the state and input that is expected to hold when the function is invoked), and the \texttt{ensures} annotation (which asserts the condition expected to hold when the function returns), as commonly used in verification tools like VCC [6] and Verifast [10].

(\textbf{init-a}) \texttt{func_{init}} must terminate on all state-input pairs satisfying \( pre^Z_{\text{init}} \rightarrow C \).

(\textbf{init-b}) \texttt{func_{init}(x_{\text{init}} x)}
\hspace{1em} _(\texttt{requires } pre^Z_{\text{init}} \rightarrow C)\hspace{1em}
\hspace{1em} _(\texttt{ensures } BAP^Z_{\text{init}} \rightarrow C) \hspace{1em} \{} \hspace{1em}
\hspace{2em} // function body \hspace{1em} \}\hspace{1em}

(\textbf{sim-a}) For each operation \( n \), \texttt{func_n} must terminate on all state-input pairs satisfying \( pre^n_{\text{init}} \rightarrow C \).

(\textbf{sim-b}) For each operation \( n \):
\texttt{func_n(x, x)}
\hspace{1em} _(\texttt{requires } pre^n_{\text{init}} \rightarrow C)\hspace{1em}
\hspace{1em} _(\texttt{ensures } BAP^n_{\text{init}} \rightarrow C) \hspace{1em} \{} \hspace{2em} // function body \hspace{1em} \} \hspace{1em}

In the second case, we assume that we have a “ghost” implementation of the Z model $\mathcal{M}$. A ghost implementation is similar to a C implementation, except that it uses “ghost” variables that are declared separately, and can be embedded into a C program ensuring that it does not interfere with the program state.

To show that a C implementation $\mathcal{C}$ refines a ghost implementation $\mathcal{M}$, we can encode the sufficient condition for refinement as follows. We construct a “joint” program $P_{\mathcal{M},\mathcal{C}}$ that has global variables of both $\mathcal{M}$ and $\mathcal{C}$. For each operation $n$, the joint function definition $\text{func}^{\mathcal{M},\mathcal{C}}_n$ has the same argument and return value as $\text{func}^{\mathcal{C}}_n$, but executes the bodies of both functions. We need a predicate $\text{inv}_\rho$ on the joint state of the ghost implementation and the C implementation, describing an abstraction relation $\rho$ from the states of the C implementation to the states of the ghost implementation. We now assert the conditions on the joint program as described below:

\begin{enumerate}
  \item[\text{(init-a)}] $\text{func}^{\mathcal{M},\mathcal{C}}_n$ terminates on all joint state-input pairs satisfying $\text{pre}^\mathcal{M}_\text{init}$.
  \item[\text{(init-b)}] $\text{func}^{\mathcal{M},\mathcal{C}}_\text{init} (x, x_\text{init})$
      \begin{itemize}
        \item[\text{_(requires)}] $\text{pre}^\mathcal{M}_\text{init}$
        \item[\text{_(ensures)}] $\text{inv}_\rho \land y^\mathcal{M}_\text{init} = y^\mathcal{C}_\text{init}$
        \hspace{1em} \{ \hspace{1em}
          \begin{itemize}
            \item[\text{// body of func}^{\mathcal{M}}_\text{init}]
            \item[\text{// body of func}^{\mathcal{C}}_\text{init}]
        \end{itemize}
      \}
  \end{itemize}
  \item[\text{(sim-a)}] For each operation $n$, $\text{func}^{\mathcal{M},\mathcal{C}}_n$ must terminate on all state-input pairs satisfying $\text{pre}^\mathcal{M}_n \land \text{inv}_\rho$.
  \item[\text{(sim-b)}] For each operation $n$:
      \begin{itemize}
        \item[\text{func}^{\mathcal{M},\mathcal{C}}_n (x, x_n) ]
        \item[\text{_(requires)}] $\text{pre}^\mathcal{M}_n \land \text{inv}_\rho$ \\
        \item[\text{_(ensures)}] $\text{inv}_\rho \land y^\mathcal{C}_n = y^\mathcal{C}$
        \hspace{1em} \{ \hspace{1em}
          \begin{itemize}
            \item[\text{// body of func}^{\mathcal{M}}_n]
            \item[\text{// body of func}^{\mathcal{C}}_n]
        \end{itemize}
      \}
  \end{itemize}
\end{enumerate}

Finally, for refinement between two C implementations $\mathcal{C}_1$ and $\mathcal{C}_2$, with $\mathcal{C}_1$ possibly having preconditions, we can phrase the sufficient condition for refinement in a similar way to the case above (C implementation refines ghost implementation), except for the following:

1. Let $\text{term}^{\mathcal{C}_1}_n$ denote a predicate describing the set of state-input pairs on which $\text{func}^{\mathcal{C}_1}_n$ terminates. Then the condition $\text{pre}^{\mathcal{C}_1}_\text{init}$ in (init-a) and (init-b) can replaced by $\text{pre}^{\mathcal{C}_1}_\text{init} \land \text{term}^{\mathcal{C}_1}_\text{init}$, and similarly $\text{pre}^{\mathcal{C}_1}_n$ in (sim-a) and (sim-b) can be replaced by $\text{pre}^{\mathcal{C}_1}_n \land \text{term}^{\mathcal{C}_1}_n$.
2. It must be the case that the function implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$ don’t “interfere” with each other.

\section*{C \hspace{1em} Excerpts from FreeRTOS models}

Fig. 9 shows the Scheduler schema in $\mathcal{M}_2$ and the schema for $\text{vTaskDelay}$ for the case corresponding to Fig.8, but where the time-to-awake is greater than $\text{maxNumVal}$. The only operation schemas that change from $\mathcal{M}_1$ are $\text{vTaskDelay}$, $\text{vTaskDelayUntil}$ and $\text{vTaskIncrementTick}$.

Fig. 11 shows the annotations for the $\text{vTaskDelay}$ API, corresponding to the case shown in the $\mathcal{M}_2$ schema of Fig. 9.
Fig. 8. Operation schema for API vTaskDelay when another ready task of same priority is available.

Fig. 9. Scheduler schema in $M_2$ and operation schema for vTaskDelay in $M_2$ when time-to-awake is greater than the value of maxNumVal.

Fig. 10. Excerpts from xListMap and listInsert. The ghost variable index is constrained to be the required position of xli.
void vTaskDelay(unsigned xTicksToDelay)
_(requires xTicksToDelay <= maxNumVal)
_(requires xTicksToDelay > (maxNumVal - xTickCount))
_(requires oDelayed->length < maxNumVal)
_(ensures delayed == \old(delayed))
_(ensures oDelayed->length == (\old(oDelayed->length) + 1))
_(ensures \forall unsigned i; (i<oDelayed->position[pxCurrentTCB->pGenericListItem]) ==> (oDelayed->list[i]==\old(oDelayed->list[i])))
_(ensures oDelayed->list[oDelayed->position[pxCurrentTCB->pGenericListItem]] == pxCurrentTCB->pxGenericListItem)
...

Fig. 11. Excerpt from \texttt{vTaskDelay} verification in $P_1$. 