A Refinement-Based Methodology for Verifying Abstract Data Type Implementations

A Thesis
Submitted For the Degree of
Doctor of Philosophy
in the Faculty of Engineering

by

Sumesh Divakaran

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012

June 2015

Acknowledgements

I sincerely thank Prof. Deepak D'Souza, for being an inspiring mentor and a flexible supervisor with incredible patience. Being myself a teacher on deputation for pursuing PhD, I consider it a great opportunity to observe in him the qualities of a good teacher and learn from them. I was really lucky to work under his guidance.

I am also grateful to Prof. Deepak D'Souza for giving me opportunities to collaborate with renowned researchers from both academia and research labs. I express my gratitude to Prof. Jim Woodcock, Prof. Nigamanth Sridhar and Dr. Prahladavaradan Sampath for their valuable suggestions. I would also like to acknowledge the project funding agencies: UKIERI and Robert Bosh Centre for Cyber Physical Systems, IISc for their financial support on various occasions.

Education and research would not have been so enjoyable without good infra-structural facilities and motivating faculty around. I would like to extend my sincere thanks to Indian Institute of Science and the highly motivated faculty and staff in the Department of Computer Science & Automation, for providing me a wonderful environment with a nice academic ambiance. I have greatly benefited being a part of the Programming Languages research group discussions. My special thanks to the faculty members: Dr. K. V. Raghavan, Dr. Aditya Kanade and Dr. Murali Kishna Ramanathan and also my fellow student members of the group for all the helpful interactions and discussions.

I am grateful to Mr. Anirudh Kushwah, who helped me to complete a part of a case-study in this thesis.

I extend my sincere gratitude to Government Engineering College, Idukki; Department of Technical Education, Government of Kerala and All India Council for Technical Education, Government of India for giving me this opportunity to pursue my PhD on QIP deputation with full salary.

I would like to thank all my friends who made my IISc life really enjoyable. Discussions and debates with Rajmohan, Jasine, Deepak, Rogers, Dilip, Musa, Sunil Sir, Pradeesha and many others were thrilling and memorable.

A special thanks to my daughter Lakshmi, son Arjun, wife Remya and my parents for their loving support and for all their sacrifices on my behalf during the busy schedule of my PhD work.

In the last thirteen years of my career as a teacher, I always found motivation from the encouraging responses and appreciation of my students. It was the interactions with them that energized me to pursue higher education. Last but not the least I extend my sincere thanks to all my students for their support and care.

Abstract

This thesis is about techniques for proving the functional correctness of Abstract Data Type (ADT) implementations. We provide a framework for proving the functional correctness of imperative language implementations of ADTs, using a theory of refinement. We develop a theory of refinement to reason about both declarative and imperative language implementations of ADTs. Our theory facilitates compositional reasoning about complex implementations that may use several layers of sub-ADTs.

Based on our theory of refinement, we propose a methodology for proving the functional correctness of an existing imperative language implementation of an ADT. We propose a mechanizable translation from an abstract model in the Z language to an abstract implementation in VCC's ghost language. Then we present a technique to carry out the refinement checks completely within the VCC tool.

We apply our proposed methodology to prove the functional correctness of the scheduling-related functionality of FreeRTOS, a popular open-source real-time operating system. We focused on the scheduler-related functionality, found major deviations from the intended behavior, and did a machine-checked proof of the correctness of the fixed code.

We also present an efficient way to phrase the refinement conditions in VCC, which considerably improves VCC's performance. We evaluated this technique on a simplified version of FreeRTOS which we constructed for this verification exercise. Using our efficient approach, VCC always terminates and leads to a reduction of over 90% in the total time taken by a naive check, when evaluated on this case-study.

Publications from this thesis

1. Sumesh Divakaran, Deepak D'Souza and Nigamanth Sridhar. *Efficient Refinement Checking in VCC*. Verified Software: Theories, Tools and Experiments, 21-36 (VSTTE 2014). Springer -Verlag Lecture Notes in Computer Science 2014, pp 21-36.

2. Sumesh Divakaran, Deepak D'Souza, Anirudh Kushwah, Prahladavaradan Sampath,Nigamanth Sridhar and Jim Woodcock. *Refinement-Based Verification of the FreeRTOS scheduler in VCC* (submitted to ICFEM 2015).

3. Sumesh Divakaran, Deepak D'Souza, Prahladavaradan Sampath, Nigamanth Sridhar and Jim Woodcock. *A theory of refinement for Abstract Data Types with functional interfaces* (preprint).

Contents

Chapter 1

Introduction

This thesis presents techniques for proving the functional correctness of imperative language implementations of Abstract Data Types (ADTs). In the motivation section we discuss the term *verification* with an emphasis on the significance of verifying functional correctness. Next we describe different approaches to verifying functional correctness and highlight the advantages of a refinement-based approach. We then give an overview of the contributions made in this thesis.

1.1 Background and motivation

Program verification is the process of proving in a formal or mathematically precise way that a *model* of a program or software system, satisfies a certain *property*. There are a variety of properties considered in the literature, that one could classify in a spectrum ranging from "lightweight" to "deep".

We illustrate these properties via an example program which we will use as a running example in this thesis. Consider the program of Fig. 1.1, which gives an efficient implementation of a queue data-structure. It maintains the contents of the queue in the array `A` starting from the position "`beg`" and going up to "`end-1`", wrapping around to the start of the array if necessary. This C implementation satisfies the following properties: (i) the variables `beg` and `end` can take any value in the set $\{0, 1, \ldots, \texttt{MAXLEN} - 1\}$, (ii) the variable `len` can take any value in the set $\{0, 1, \ldots, \texttt{MAXLEN}\}$, (iii) the variables `beg`, `end` and `len` are related by the formula "$(\texttt{beg} + \texttt{len}) \ \% \ \texttt{MAXLEN} = \texttt{end}$" and (iv) the sequence of elements stored in the array can be accessed by the expression "`A[(beg+i) % MAXLEN]`", where $0 \leq \texttt{i} < \texttt{len}$. Fig. 1.2 shows two instances of the queue data-structure in the program of Fig. 1.1, where the left shows an instance after the sequence of operations: initialization followed by three insertions, and the right shows an instance after a sequence of insertions and deletions.

At one end of the spectrum of properties for program verification, we have lightweight properties. For example, the following are some interesting lightweight properties about the program `c-queue` of Fig. 1.1: "there is no array-index out of bounds error in `c-queue`", "`c-queue` satisfies the assertion

```
 1: int A[MAXLEN];              12: void enq(int t)
 2: unsigned beg, end, len;     13: {
 3:                             14:    if (len == MAXLEN)
 4: void init()                 15:       assert(0); /* exception */
 5: {                           16:    A[end] = t;
 6:    beg = 0;                 17:    if (end < MAXLEN-1)
 7:    end = 0;                 18:       end++;
 8:    len = 0;                 19:    else
 9: }                           20:       end = 0;
10:                             21:    len++;
11: int deq() { ... }           22: }
```

Figure 1.1: A C implementation of a queue data-structure called `c-queue`. The constant `MAXLEN` is a bound on the number of elements allowed in the queue.

Figure 1.2: Two instances of the queue data-structure in the program of Fig. 1.1.

at line `15`", or "`c-queue` satisfies the invariant that the number of elements in the array `A` is less than or equal to the constant `MAXLEN`".

A number of program verification techniques are available in the literature for verifying lightweight properties like above.

Model checking is a well known verification technique. In model checking, a transition system models the system to be verified and temporal logic is usually used for property specification. Model checking performs an exhaustive analysis of the model to prove the property of interest. Tools like Spin [28] and Sal [19] can be used for model checking. Model checking could be used to prove a property like, "a program satisfies an invariant".

Program analysis is another verification technique, which could be used to verify lightweight properties of a program. Abstract interpretation is a commonly used program analysis technique in which a data flow analysis is performed on an abstract model of the program, over-approximating its set of all behaviors. For example, an abstract interpretation framework could be used to prove the properties mentioned about the program `c-queue`.

At the other end of the spectrum of properties, we have deeper properties like *functional correctness*. Functional correctness is a property of a system, which ensures that the system satisfies its *intended purpose* (or *functionality*). For example, consider a program to sort an array of integers. The intended

$content$: `seq` $\mathbb{Z}$

$init()$:
  $content' = \langle\rangle$

$enq(x: \mathbb{Z})$:
  $\#content < k$
  $content' = content \frown \langle x \rangle$

$deq()$:
  $result$: $\mathbb{Z}$
  $content \neq \langle\rangle$
  $result = head(content)$
  $content' = tail(content)$

Figure 1.3: An abstract specification `z-queue`$_k$ of the queue ADT, parameterized by a constant k denoting the capacity of the queue.

purpose of this program is to return an output array which represents a *sorted permutation* of the elements present in the input array.

Functional correctness is an important property about a program, since it provides guarantees about the functionality of the program. For example, consider the case of a Real-Time Operating System (RTOS), which provides an operation to delay a task for a given period with respect to its previous wake time. Guarantees about the functionality of this operation could be used to prove that an RTOS application correctly implements a periodic task like releasing a unit of oxygen at a fixed time interval. Thus here we want guarantees about the functionality of the RTOS, which is deeper than the lightweight properties discussed above. Moreover verification guarantees about the functionality of an RTOS is vital for reasoning about applications developed over an RTOS, which are used in safety-critical systems in fields like: avionics, health-care and automotive.

The verification techniques discussed above are useful for proving lightweight properties about a program and these techniques scale to large program models. However these techniques cannot prove deeper properties about a program like functional correctness.

An Abstract Data Type (ADT) can be used to represent the functionality of a program, which provides an interface of operations for the client programs to interact with it. An ADT basically provides a set of interface operations in its Application Programmer Interface (API). For example, the program `c-queue` of Fig. 1.1, can be thought of as an implementation of a queue ADT with the set of interface operations: $\{init, enq, deq\}$.

An ADT can be modeled in a mathematical specification language like Z [46]. For example, Fig. 1.3 shows an abstract specification of a queue ADT in a Z-like language. The specification, which we call `z-queue`$_k$ is parameterized by a constant k representing the maximum length of the queue. The "type" of the queue ADT is the set of operations: $\{init, enq, deq\}$ in its API, and the associated input/output type for each operation. For example, the operation *enq* takes an integer argument and returns nothing (which we represent by a dummy return value "*ok*"). The ADT has a state, in this case the value of the variable *content* which is a finite sequence of integers denoting the contents of the queue. The body of each operation (like *enq*) represents a Before-After Predicate (BAP) on the input and the ADT states before and after the

operation. By convention, the primed variables like $content'$ denote the state component in the after-state.

Each operation on the ADT works as follows: when called on a state of the ADT with a given argument, it updates the state of the ADT and returns a value in its output type to the caller. Thus, the *enq* operation when called on a state s whose length is less than k, with an argument x, updates the state to append x to s and returns *ok*. When an operation is called on a state that lies outside its precondition (in the case of *enq* this happens when the length of the queue is k or more), the operation is assumed to return a special "exceptional" value "e" and update the state to a special "exceptional" state E. Once in an exceptional state, all operations on the ADT must maintain the exceptional state and return the exceptional value e. The exceptional state and the exceptional value represent situations like "division by zero error", "null-pointer dereference" and "infinite loop" which are possible in an imperative language implementation of an ADT.

Given a mathematical model and an implementation of an ADT, the implementation is said to be functionally correct with respect to the mathematical model, if the implementation "conforms" to the model. The exact meaning of what it means to conform to the model would vary according to the objective of the verification process, but it could mean for instance that every execution of the concrete implementation can be "matched" or "simulated" by an execution of the abstract model. For instance, the functional correctness of the implementation `c-queue` of Fig. 1.1, which implements a queue ADT could be proved with respect to the Z-like specification $\texttt{z-queue}_k$ of Fig. 1.3, whenever the constant `MAXLEN` in `c-queue` is less than or equal to the parameter k of the model.

1.2 Techniques for proving functional correctness

There are essentially two approaches in the literature for proving the functional correctness of an implementation of an ADT-like system.

The first approach uses *function contracts* for proving functional correctness. A function contract is a requirement on a method, which is typically specified using the `requires` and `ensures` annotations provided by a code verification tool like VCC [15]. The `requires` annotation is a condition on the before-state and input of a method and the `ensures` annotation is a predicate relating the after-state and output of a method to its before-state and input. For instance, Fig. 1.4 shows an example function contract required in VCC to prove the functionality of the method `init` in the program `c-queue` of Fig. 1.1. Tools like VCC support verification based on function contracts, which basically perform a weakest precondition analysis on an annotated program. Many recent verification efforts for functional correctness in the community [34, 7, 12, 42] have favored the use of function contracts.

The second technique in the literature for proving functional correctness is

```
void init()
  _(requires \true)
  _(ensures beg = 0)
  _(ensures end = 0)
  _(ensures len = 0)
{
   //body of the method
}
```

Figure 1.4: Illustrating function contract for verifying the `init` method in the program `c-queue` of Fig. 1.1.

```
 1: tasks t;
 2: init();
 3: enq(0);
 4: enq(1);
 5: t = deq();
 6: while (true)
 8: {
 9:    if (*) // tick occured?
10:    {
11:       enq(t);
12:       t = deq();
13:    }
14: }
```

Figure 1.5: A client program `interp` that interprets two tasks of equal priority. The data-type `tasks` is assumed to be *integer*.

based on a notion of refinement. The idea of program refinement was proposed in the late 1970's as a methodology for developing correct-by-construction software systems. Here one begins with an abstract specification of the system's functionality in a concise and mathematically precise modeling language, and successively refines it by adding implementation details to finally obtain an implementation of the system which is guaranteed to "conform" to the high-level specification. For example, one could use refinement to obtain the implementation `c-queue` of Fig. 1.1, from the abstract mathematical model $\texttt{z-queue}_k$ of Fig. 1.3.

```
_(ghost int content[\natural])
_(ghost \natural beg, end, len)

void init(void)
{
   _(ghost beg = 0)
   _(ghost end = 0)
   _(ghost len = 0)
}
```

```
int deq() { ... }

void enq(int a)
  _(requires len < k)
{
   _(ghost content[end] = a)
   _(ghost len = len + 1)
   _(ghost end = end + 1)
}
```

Figure 1.6: A ghost version of $\texttt{z-queue}_k$ in VCC.

1.3 Advantages of refinement-based approaches

In our opinion, refinement-based approaches have several advantages over the approach based on function contracts.

The first advantage is that a refinement-based approach provides a stand-alone abstract specification (say $\mathcal{A}$) of the implementation (say $\mathcal{C}$), with the guarantee that certain properties proved about a client program P that uses $\mathcal{A}$ as an ADT (which we refer to as "P with $\mathcal{A}$" and denote by "$P[\mathcal{A}]$") also carry over for P with $\mathcal{C}$ (i.e. $P[\mathcal{C}]$). Thus, to verify that $P[\mathcal{C}]$ satisfies a certain property, it may be sufficient to check that $P[\mathcal{A}]$ satisfies the property. The latter check is in terms of a simpler component (namely $\mathcal{A}$) and can reduce the work of a prover by an order of magnitude [32].

To illustrate this, consider a client program of the `c-queue` ADT, shown in Fig. 1.5, which we call `interp`. With some imagination, one could view it as "interpreting" or executing two tasks of equal priority running on an operating system. Suppose we want to verify that the program `interp`, which uses the `c-queue` ADT (that is `interp`[`c-queue`]) does not encounter an exception while calling one of the queue operations, or that it satisfies an assertion on its local state (like the assertion: `(t == 0 || t == 1)` at line `6`). One could first prove that the program with the abstract $\texttt{z-queue}_k$ ADT (that is $\texttt{interp}[\texttt{z-queue}_k]$) verifies these properties and then infer by using a suitable theory of refinement that the program with the concrete `c-queue` ADT (that is `interp`[`c-queue`]) also satisfies these properties whenever `c-queue` is a refinement of $\texttt{z-queue}_k$. The above proof can be done in a prover like VCC for example by using a ghost implementation of the abstract $\texttt{z-queue}_k$ ADT called $\texttt{g-queue}_k$, shown in Fig. 1.6. Since $\texttt{g-queue}_k$ is a simpler program than `c-queue` the latter check is more tractable for a prover than the former.

The second advantage of using refinement is that it enables the existing clients of an ADT implementation to use a new, more efficient and refined ADT implementation without sacrificing their proved properties. For example, suppose we have an ADT implementation $\mathcal{C}$ that is used by several client programs $P_1, P_2, \ldots$ about each of which we have proved certain properties φ_i. We can now replace the ADT $\mathcal{C}$ in each of these clients by a more efficient one

$\mathcal{C}'$ refining $\mathcal{C}$, to obtain $P_i[\mathcal{C}']$. A suitable theory of refinement would let us infer that each $P_i[\mathcal{C}']$ continues to satisfy φ_i. Thus by using refinement, one can save the time and effort required to reprove the property φ_i for each $P_i[\mathcal{C}']$.

The third advantage of using refinement is that it makes the proof more modular and transparent, since it breaks up the task of reasoning about a complex implementation into smaller tasks, each of which is more manageable for both a human and a prover. There are two different techniques, which can be used in a refinement-based approach, to break up the complexity of proving functional correctness in a single step. We illustrate these techniques in the following paragraphs.

The first technique is to use a sequence of refinements to prove that a complex implementation refines a high-level mathematical model. Suppose we want to prove the functional correctness of a complex ADT implementation $\mathcal{C}$ with respect to a high-level model $\mathcal{M}$. In a verification based on function contracts, a user may need to use complex formulas in function contracts, when there is a large gap between the models $\mathcal{M}$ and $\mathcal{C}$. Tools like VCC may run out of memory when one tries to prove complex function contracts on large models. On the other hand, in a refinement-based approach, one could use a number of intermediate models to reduce the gap between $\mathcal{M}$ and $\mathcal{C}$. Thus a user could first divide the proof of refinement between $\mathcal{M}$ and $\mathcal{C}$ into a number of smaller steps, each of which is to prove the refinement between successive models in a refinement chain, starting with the high-level model $\mathcal{M}$ and ending with the implementation $\mathcal{C}$. Then the user could infer by using a suitable theory of refinement that $\mathcal{C}$ refines $\mathcal{M}$.

The second technique is to use simplified ADTs to reason about the correctness of a client program which calls operations from concrete ADT implementations. Suppose we want to prove a property φ about a client program $\mathcal{P}$ which uses two ADT implementations $\mathcal{C}_1$ and $\mathcal{C}_2$. Thus the client program is of the form $\mathcal{P}[\mathcal{C}_1][\mathcal{C}_2]$ and we need to prove that $\mathcal{P}[\mathcal{C}_1][\mathcal{C}_2]$ satisfies φ. Using a notion of refinement, one could divide this proof into three steps. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be simplified ghost implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. The first step is to prove that $\mathcal{P}[\mathcal{G}_1][\mathcal{G}_2]$ satisfies φ. The second and third steps need to prove that each $\mathcal{C}_i$ refines $\mathcal{G}_i$. Then the user could infer by using a suitable theory of refinement that $\mathcal{P}[\mathcal{C}_1][\mathcal{C}_2]$ refines $\mathcal{P}[\mathcal{G}_1][\mathcal{G}_2]$ and hence also that $\mathcal{P}[\mathcal{C}_1][\mathcal{C}_2]$ satisfies φ.

1.4 Selecting a notion of refinement

In this section we discuss different notions of refinement in the literature and attempt to justify why we prefer one of these notions - namely the VDM/Z style of refinement - over the others.

The notion of refinement in Event-B [4] is that the abstract simulates the concrete. This notion allows a concrete operation to *strengthen* its precondition [3] and hence provides rather weak guarantees in terms of the properties of $P[\mathcal{A}]$ that carry over to $P[\mathcal{C}]$. If $\mathcal{C}$ refines $\mathcal{A}$ in this notion of refinement, then it means that any execution of $\mathcal{C}$ that is "exception-free" – that is it does not

cause an exception due to say a null-dereference or divide-by-zero error, or due to non-termination of a loop – can be simulated by an execution of the abstract specification $\mathcal{A}$.

To illustrate why one would normally want a stronger guarantee than the above, consider a client program P that is "happy" with an ADT $\mathcal{A}$, in that $P[\mathcal{A}]$ never makes a call that causes an exception and that all its assertions on its local state are verifiable. Then this notion of refinement will *not* guarantee that P will be happy with a refinement $\mathcal{C}$ of $\mathcal{A}$ as well. In particular, $P[\mathcal{C}]$ can contain an exceptional behavior, even though $P[\mathcal{A}]$ did not. For example, $\texttt{c-queue}_2$ is a refinement of $\texttt{c-queue}_3$ in this notion of refinement, since $\texttt{c-queue}_3$ can simulate $\texttt{c-queue}_2$. However a client program which is happy with $\texttt{c-queue}_3$ will not be happy with $\texttt{c-queue}_2$, since $\texttt{c-queue}_3$ allows operation sequences like "$(\mathit{init}, \mathit{nil}, \mathit{ok}), (\mathit{enq}, 1, \mathit{ok}), (\mathit{enq}, 2, \mathit{ok}), (\mathit{enq}, 3, \mathit{ok})$" which $\texttt{c-queue}_2$ does not.

Liskov and Wing give a notion of refinement in the form of behavioral subtyping [37]. In a deterministic setting, their refinement notion requires *both* the abstract and concrete ADTs to simulate each other. This requirement is too strong as it does not allow certain refinements, which are valid in the sense that they preserve the properties required to satisfy the existing clients of the abstract ADT. For instance, $\texttt{c-queue}_3$ cannot refine $\texttt{c-queue}_2$ in this notion of refinement, since $\texttt{c-queue}_2$ cannot simulate $\texttt{c-queue}_3$.

The VDM/Z style of refinement [13, 30, 5, 46, 26] preserves the properties required to satisfy the existing clients of the abstract ADT in the above sense. In this notion of refinement a concrete ADT must allow an operation whenever the operation is allowed by the abstract ADT. Thus here refinement basically means reduction of non-determinism and it allows precondition *weakening*. In this notion an ADT is defined to have two special operations namely *initialization* and *finalization* whose purpose is respectively to convert a global (client's) state into an ADT state and convert an ADT state to a global state. Thus a client's interaction with the ADT is considered as a sequence of ADT operations starting with the initialization operation and ending with a finalization operation.

We prefer the VDM/Z style of refinement over the other notions of refinement discussed above. It is preferable over the notion of refinement in behavioral subtyping [37], since it allows more refinements than the other notion, while preserving the properties required to satisfy the existing clients of an abstract ADT. For instance the VDM/Z style allows $\texttt{c-queue}_3$ to refine $\texttt{c-queue}_2$ which is not allowed by the other notion. We prefer the VDM/Z style of refinement over the Event-B notion of refinement, since it is stronger than the latter in the sense of satisfying the clients of an abstract ADT. For example if $\mathcal{C}$ refines $\mathcal{A}$ in the VDM/Z style of refinement and if exception is non-reachable in $\mathcal{P}[\mathcal{A}]$, then $\mathcal{P}[\mathcal{C}]$ will never lead to an exception and every behaviour of $\mathcal{P}[\mathcal{C}]$ is also a behaviour of $\mathcal{P}[\mathcal{A}]$. On the other hand the Event-B notion of refinement cannot provide a guarantee as discussed above.

However there are some shortcomings in the VDM/Z style of refinement which we would like to rectify. The first problem is the way a client can

interact with an ADT. A client's interaction must start with an invocation to the initialization operation to convert a client state to an ADT state and end with an invocation to the finalization operation to convert an ADT state to a client state. Thus a client needs to remember the ADT state in this notion of refinement. But we need a more natural approach where a client need not remember the ADT state and a client can interact with the ADT by invoking an ADT operation by passing input if any and receiving output if any is returned from the ADT operation.

1.5 The notion of refinement we use and its theory

We use a notion of refinement similar in spirit to the VDM/Z style, except that we change the definition of an ADT (mainly its interface) to allow operations to have input arguments and output values. Thus a client program does not need the initialization/finalization constructs, but instead repeatedly interacts with the ADT by calling its operation, each time passing an argument and making use of the return value. Our definition of refinement is then in terms of the sequences of operations allowed by the abstract and concrete ADTs.

There is a natural notion of the set of (initialized) sequences of operation calls allowed by an ADT. Each element of such a sequence is of the form "(n, a, b)" where n is an operation name, and a and b are respectively input and output to the operation. For example, $\texttt{z-queue}_2$ allows the sequence: $(init, nil, ok), (enq, 1, ok), (deq, nil, 1)$ (here "*nil*" represents a dummy input value). This sequence of calls is *exception-free*. It also allows the sequence: $(init, nil, ok), (deq, nil, e)$, which however contains an exception. In a deterministic setting we say an ADT $\mathcal{C}$ refines another ADT $\mathcal{A}$ of the same type, if every exception-free sequence of operations allowed by $\mathcal{A}$ must also be allowed by $\mathcal{C}$. Thus, $\texttt{z-queue}_3$ refines $\texttt{z-queue}_2$, but not vice-versa since $\texttt{z-queue}_3$ allows the sequences like: $(init, nil, ok), (enq, 1, ok), (enq, 2, ok), (enq, 3, ok)$ which $\texttt{z-queue}_2$ does not.

This notion of refinement satisfies the requirements discussed above. In particular, if a client program is happy with an ADT $\mathcal{A}$ (in the above sense) then it will continue to be happy with an ADT $\mathcal{C}$ that refines $\mathcal{A}$. This notion of refinement is clearly also transitive.

A trace-based notion of refinement similar to the above was discussed in the technical report [1] by Hoare et al, where they considered non-deterministic ADTs. In their notion of refinement a concrete ADT $\mathcal{C}$ refines an abstract ADT $\mathcal{A}$, if every legal sequence of interactions allowed by $\mathcal{C}$ is also allowed by the *totalized* version of $\mathcal{A}$. The totalized version of an ADT allows all possible transitions when an operation is called outside its precondition. However this notion appears to have been abandoned in favour of the one based on initialization/finalization in the final version [26].

We give a refinement condition (RC) based on a simulation relation, which is sufficient to ensure the proposed notion of refinement, and which is also

$ready, blocked : seq\ tasks$
...

$init()$: $ready' = blocked' = \langle\rangle$

$create(\text{t}: tasks)$:
  $ready' = ready \frown \langle t \rangle$

$resched(cur: tasks)$:
  $result: tasks$
  $result = head(ready \frown \langle cur \rangle)$
  $ready' = tail(ready \frown \langle cur \rangle)$

$delay(cur: tasks)$:
  ...

Figure 1.7: `z-sched`: An abstract specification of a scheduler ADT.

```
1: task resched(task cur)
2: {
3:    task t;
4:    enq(cur);
5:    t = deq();
6:    return t;
7: }
```

Figure 1.8: A part of the C implementation `c-sched` showing the method `resched`, which implements the reschedule operation of the scheduler ADT, that uses `c-queue` as a sub-ADT. The data-type `task` is assumed to denote the type $\mathbb{B}$, the set of bit values.

necessary in the case of deterministic ADTs. This is similar to the result in [1]. We extend our notion of refinement to deterministic transition systems modeling ADT implementations. Then we lift our equivalent refinement condition (RC) to ADT implementations. Thus we have a sound and complete characterization of refinement between ADT implementations as well. We also prove a substitutivity theorem for ADT implementations, which enables us to propose a compositional refinement methodology for reasoning about complex ADT implementations which make use of sub-ADTs. In the subsequent paragraphs we illustrate the use of our substitutivity theorem for proving the functional correctness of a complex ADT implementation.

Complex ADT implementations often take service from sub-ADTs to implement some operations in their APIs. For example, consider a simple OS scheduler, which maintains a set of ready tasks (ordered according to arrival time), and a set of blocked tasks, among other things. We can view the scheduler as an ADT that provides the following operations: (i) *init*, which initializes the lists to empty, (ii) *create*, which takes a new task and adds it to the end of the ready list, and (iii) *resched*, which takes the currently running task as input, adds it to the end of the ready list, removes the task at the head of the new ready list and returns it as the next task to run.

Fig. 1.7 shows an abstract specification of the scheduler ADT, called `z-sched` and Fig. 1.8 shows a method, which implements the reschedule operation which is a part of the C implementation called `c-sched` of the scheduler ADT.

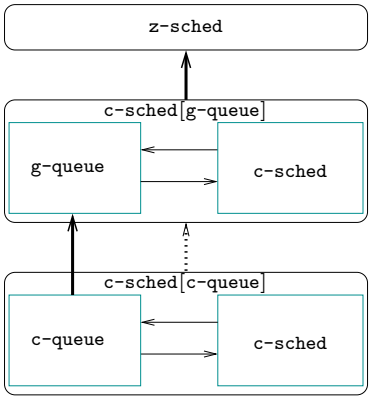


Figure 1.9: Strategy to prove that `c-sched`[`c-queue`] refines `z-sched`. Thick upward arrows denote refinements to be proved directly and the dotted arrow denotes the refinement which can be inferred using our substitutivity result.

Suppose we want to show that `c-sched` refines `z-sched`. We would like to reason about this in a step-by-step manner to reduce the complexity involved in doing this in a single step. As a first step, we could abstract the `c-queue` component and replace it by the simpler and more-abstract `g-queue` component (see Fig. 1.6), and argue that `c-sched`[`g-queue`] refines `z-sched`. As a second step we would need to argue that `c-sched`[`c-queue`] refines `c-sched`[`g-queue`]. This is depicted in Fig. 1.9. Our substitutivity result tells us that to do this second step, it is sufficient to show that `c-queue` refines `g-queue`. It is in this way that the substitutivity result adds compositionality to our verification process.

Conventional notions of refinement handle non-determinism which is a useful mechanism to ease the modeling process. Also it enables the user to delay the resolution of a non-deterministic choice to a later stage in refinement. To illustrate the flexibility provided by non-determinism, consider a simple operating system scheduler where tasks are of the same priority. Here the abstract model of the scheduler could be specified to schedule any one of the set of ready tasks. Thus in the abstract model, the user does not have to worry about the implementation details like the order in which the tasks are maintained. We propose an extension of our refinement theory to handle non-determinism, which also assures stronger guarantees to the clients. We extend our theory in such a manner to preserve exactly the same set of properties preserved by our notion of refinement for deterministic ADTs.

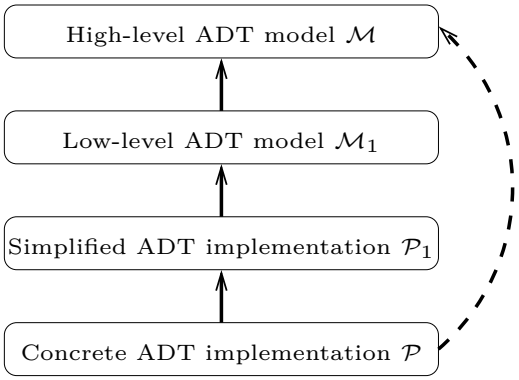


Figure 1.10: Illustrating the methodology for proving functional correctness. Thick arrows denote refinements to be proved directly and the dashed arrow denotes the refinement, which can be inferred using our transitivity result.

1.6 Methodology for proving functional correctness

We propose a methodology for proving the functional correctness of an imperative language implementation of an ADT-like system. The proposed methodology is based on our theory of refinement. We also propose a couple of techniques to use existing tools such as VCC for proving the refinement conditions which are required to prove the functional correctness of an ADT implementation.

Fig. 1.10 illustrates our methodology for proving the functional correctness of an ADT implementation $\mathcal{P}$.

The first step in the methodology is to specify the functionality of $\mathcal{P}$ in a mathematically precise specification language like Z. Let $\mathcal{M}$ be a high-level mathematical model in Z, specifying the intended behavior of $\mathcal{P}$.

The second step in the methodology is to obtain a simplified ADT implementation say $\mathcal{P}_1$ from $\mathcal{P}$ by replacing the sub-ADTs, if any, in $\mathcal{P}$ with simplified versions of the sub-ADTs. For instance, suppose $\mathcal{P}$ is the implementation of the scheduler ADT `c-sched` of Fig. 1.8. The ADT `c-sched` uses the sub-ADT `c-queue` and thus is of the form `c-sched`[`c-queue`]. Now we could replace `c-queue` in $\mathcal{P}$ with the ghost version `g-queue` of Fig. 1.6, to obtain the simplified ADT implementation $\mathcal{P}_1$ which is of the form `c-sched`[`g-queue`].

The next step in the methodology is to reduce the gap between the models $\mathcal{M}$ and $\mathcal{P}_1$ by using a number of intermediate ADT models. Let $\mathcal{M}_1$ be a low-level ADT model which refines the high-level ADT model $\mathcal{M}$ such that the model $\mathcal{M}_1$ is sufficiently close to the model $\mathcal{P}_1$. For instance, suppose the set of ready tasks (of same priority) is modeled as a sequence of maximum length 1024 in the high-level model $\mathcal{M}$, while these tasks are maintained in two arrays of length 512 each in the simplified scheduler implementation $\mathcal{P}_1$. In this case, we could refine $\mathcal{M}$ to $\mathcal{M}_1$ by dividing the single sequence of ready

tasks into two sequences to reduce the gap between $\mathcal{M}$ and $\mathcal{P}_1$.

The methodology now requires to prove that the simplified ADT implementation $\mathcal{P}_1$ refines the low-level ADT model $\mathcal{M}_1$. After proving this refinement, the next step is to prove the refinement between the concrete sub-ADTs and their simplified versions used in $\mathcal{P}_1$. For instance, in the above example we need to prove that `c-queue` refines `g-queue`.

Finally we can infer using our substitutivity and transitivity results that the existing implementation $\mathcal{P}$ refines the high-level ADT model $\mathcal{M}$.

A natural question a VCC expert may now ask is why we chose to build a "metatheory" of refinement on top of VCC, instead of using its internal style of data abstractions as illustrated in [14]. In the latter idiom, to prove an assertion about a client program $\mathcal{P}$ with a concrete ADT implementation $\mathcal{C}$, one constructs a joint ADT $\mathcal{AC}$ which contains a ghost version of the ADT called $\mathcal{A}$, and includes a coupling constraint between the states of $\mathcal{A}$ and $\mathcal{C}$. One then proves the assertion in $\mathcal{P}[\mathcal{AC}]$. By the restrictions imposed by VCC on ghost code, it follows that the assertion must continue to hold on the original program $\mathcal{P}[\mathcal{C}]$ as well. While this style of verification has many of the advantages of a refinement-based approach, it loses out in a couple of aspects. Firstly, VCC must reason about $\mathcal{P}$ with the joint structure $\mathcal{AC}$ (instead of simply $\mathcal{P}[\mathcal{A}]$ in a refinement-based approach). While it is possible to control the portion of the joint state exposed to the prover, this requires expert knowledge of VCC. Secondly, if we want to prove a property of $\mathcal{P}[\mathcal{C}]$, like a temporal logic specification, which is not possible with VCC, this idiom is not of much use. On the other hand, using a meta-theory of refinement, we could use VCC to prove that $\mathcal{C}$ refines $\mathcal{A}$, prove the required property about $\mathcal{P}[\mathcal{A}]$ using non-VCC means, and infer the property for $\mathcal{P}[\mathcal{C}]$.

We propose techniques for phrasing the refinement conditions between successive models proposed in our verification methodology. Our aim is to formulate these conditions in a way that enables us to use existing tools like Z/Eves [44] and VCC [15], or provers like PVS [41] and Z3 [18] to obtain machine-checked proofs of correctness of the refinement conditions. However there are some difficulties to use the existing tools to achieve this.

One problem is that performing a refinement proof between the abstract models, like a proof that $\mathcal{M}_1$ refines $\mathcal{M}$ is challenging because the level of automation in tools such as Z/Eves [44] and Rodin [4] is inadequate, and requires non-trivial human effort and expertise in theorem proving to get the prover to discharge the proof obligations. To overcome this difficulty, we propose a technique for using VCC to prove the refinement between abstract Z models. The idea is to first translate the Z models $\mathcal{M}$ and $\mathcal{M}_1$ respectively to the ghost implementations $\mathcal{G}$ and $\mathcal{G}_1$ in VCC, and then to prove in VCC that $\mathcal{G}_1$ refines $\mathcal{G}$. We propose a mechanizable translation procedure from a Z model to a ghost implementation in VCC. We also present a technique in VCC to check the refinement between ghost models.

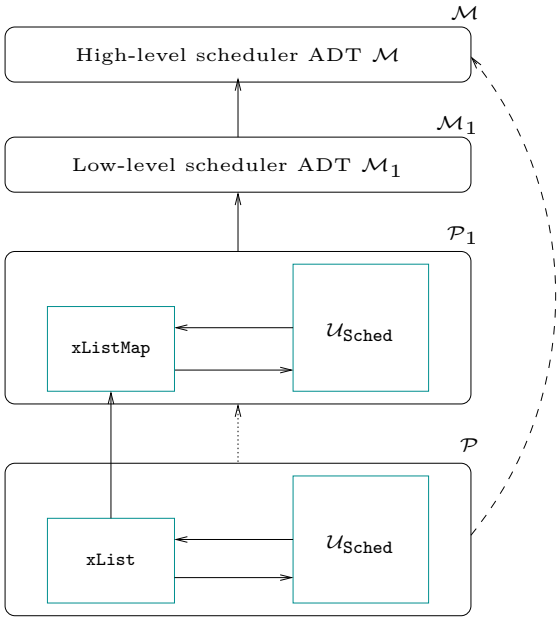


Figure 1.11: Illustrating the correctness proof of FreeRTOS. Solid upward-arrows denote directly proved refinements, the dotted arrow denotes the refinement inferred using our substitutivity result and the dashed arrow denotes the refinement inferred using our transitivity result.

1.7 Verifying FreeRTOS: a case-study

We apply our methodology on a case-study where we verify the functional correctness of the scheduler-related functionality of a popular open-source real-time operating system called FreeRTOS [38]. FreeRTOS has a large community of users. There are more than 100,000 downloads from SourceForge each year, putting it in the top 100 most-downloaded SourceForge codes. FreeRTOS is architected in a modular fashion. It has a portable layer which contains compiler/processor independent code (2,500 lines of C code), most of it in 3 C files `task.c`, `queue.c`, and `list.c`. The non-portable (or hardware-specific) layer is present in a separate directory associated with each compiler/processor pair, and is written in C and assembly.

We view the scheduler-related functionality of the FreeRTOS kernel as an ADT, specify its intended behavior in Z, and then verify that the implementation refines the Z model. The code verification tool, VCC is used to verify the refinement conditions in this case-study.

Fig. 1.11 illustrates the strategy used to prove the functional correctness of the FreeRTOS scheduler. We started with a high-level Z model called $\mathcal{M}$, specifying the intended functionality of FreeRTOS. We then obtained a simplified FreeRTOS implementation say $\mathcal{P}_1$ from the existing FreeRTOS implementation $\mathcal{P}$ by replacing a linked list sub-ADT called `xList` in $\mathcal{P}$ with an abstract ADT in VCC's ghost language called `xListMap`. Next we refined the high-level Z model $\mathcal{M}$ to a low-level Z model called $\mathcal{M}_1$ to capture some implementation details from $\mathcal{P}_1$.

We then tried to prove that $\mathcal{P}_1$ refines $\mathcal{M}_1$. In this step, we have found a number of subtle bugs (deviations from the high-level model $\mathcal{M}$) in the implementation of the FreeRTOS scheduler. We reported these bugs to the developers of FreeRTOS who acknowledged that the reported behaviors were indeed deviations from what they had intended. We fixed all the bugs found in the FreeRTOS scheduler and proved that the fixed code say $\mathcal{P}_1'$ is a refinement of $\mathcal{M}_1$.

Finally we proved that the concrete sub-ADT `xList` is a refinement of the abstract sub-ADT `xListMap`. We then concluded using our substitutivity and transitivity results that $\mathcal{P}'$ refines $\mathcal{M}$. This version of FreeRTOS ($\mathcal{P}'$) can thus be viewed as a piece of software in which the sequential behaviour of the task-related API's has been formally specified and verified. By "sequential behaviour" we mean that each API behaves correctly in the absence of interleaving with other API's.

1.8 Checking refinement conditions efficiently

In our FreeRTOS case-study, we encountered a number of difficulties in carrying out the proofs of successive refinements.

The first problem was the difficulty to prove the refinement between the low-level Z model and the simplified scheduler implementation. The requirements from the Z model were imported as function contracts for the methods in the implementation, by existentially quantifying away the components from the Z model. However these function contracts make use of existential quantifications which are difficult to handle for a verification tool like VCC. To solve this problem, we transformed each of these formulas to a logically equivalent formula which does not use existential quantifications. Nonetheless this technique also has the disadvantage that a manual translation can be error prone and hence the equivalence should ideally be checked in a theorem prover like PVS or Isabelle/HOL.

The second problem we identified was the issue with the scalability of a code verification tool like VCC for carrying out the refinement checks. VCC may take a lot of time to prove the refinement conditions or may even run out of memory for large imperative language models.

We propose a technique to overcome the difficulties mentioned above. The idea is to first translate the high-level Z model to a high-level ghost model in VCC, by applying our Z-to-VCC translation procedure mentioned in Sec. 1.6. Then the proofs of successive refinements from the high-level ghost model to the existing implementation can be performed in VCC. We use the techniques proposed in our methodology to prove these refinement conditions in VCC.

We also propose an efficient technique in VCC, which considerably improves VCC's performance for checking refinement conditions for large imperative language models.

We evaluated our translation procedure and the efficient refinement checking technique on a simplified version of FreeRTOS, which we constructed for this verification exercise. Using our efficient refinement check, VCC always

terminates and leads to a reduction of over 90% in the total time taken by a naive check, when evaluated on this case-study.

1.9 Outline

The rest of this thesis is organized as follows. Chap. 2 describes ADTs and a notion of refinement between ADTs. A transition system model for an ADT implementation is presented in Chap. 3, where we also describe the refinement between ADT implementations. In Chap. 4, we describe the mathematical ADTs induced by ADT specifications in different modeling languages. Chap. 5 describes a methodology for proving the functional correctness of a given ADT implementation and also presents a couple of techniques for checking refinement conditions using off-the-shelf tools. The concept of conformal ADTs is presented in Chap. 6, where we also describe the refinement between conformal ADTs. The FreeRTOS case-study is presented in Chap. 7, which is followed by a chapter explaining an efficient technique in VCC to check refinement conditions. Chap. 9 concludes this thesis with a discussion on future work. An extension of our refinement theory to handle non-determinism is given in Appendix A.

Chapter 2

Abstract Data Types and Refinement

This chapter begins with a section introducing the common symbols and notation used in the rest of this thesis. We then define Abstract Data Types (ADTs) and discuss transition systems as clients of ADTs. Next we introduce a notion of refinement between ADTs and describe the verification guarantees provided to the clients by the proposed notion of refinement. We formulate an *equivalent* refinement condition for a concrete ADT to refine an abstract ADT in the proposed notion of refinement. The chapter concludes with a discussion on the related work in the literature.

2.1 Preliminaries

A (labeled) *transition system* (TS) is a structure of the form $\mathcal{S} = (Q, \Sigma, s, \Delta)$ where Q is a set of states, Σ a set of action labels, s the start state, and $\Delta \subseteq Q \times \Sigma \times Q$ the transition relation. We use the notation "$s \xrightarrow{l} q$" to denote a transition from a state s to a state q in a transition system with the action label l. Two states p and q are called adjacent if there exists a transition label l such that $p \xrightarrow{l} q$. Two transitions $p \xrightarrow{l_1} q$ and $r \xrightarrow{l_2} s$ are called adjacent if $q = r$. That is, adjacent transitions are connected by a state. For example, $s \xrightarrow{a} q$ and $q \xrightarrow{b} s$ are adjacent transitions in the transition system of Fig. 2.1(i).

A "path" is a sequence of adjacent transitions in a transition system. We

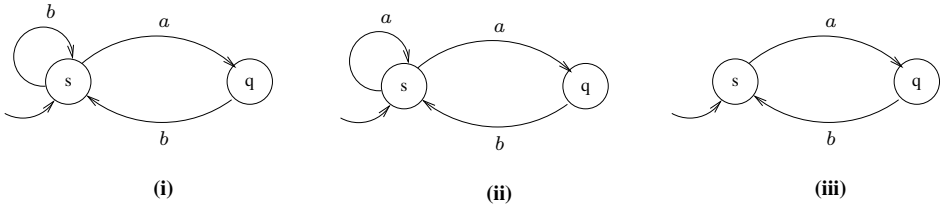


Figure 2.1: Example transition systems: (i) deterministic but not closed, (ii) non-deterministic but closed and (iii) deterministic and closed.

use the notation "$p \xrightarrow{l_1} q \xrightarrow{l_2} r$" to denote a path with the sequence of transitions: $p \xrightarrow{l_1} q$ and $q \xrightarrow{l_2} r$. This notation can be extended to paths of higher lengths in a natural way. A path is called *finite* if it is a finite sequence of transitions. For example, $q \xrightarrow{b} s \xrightarrow{b} s$ is a finite path in the transition system of Fig. 2.1(i). A path is called *initial* if it begins with the start state. For example, $s \xrightarrow{a} s \xrightarrow{a} q$ is an initial path in the transition system of Fig. 2.1(ii). We use the notation "$p_1 \cdot p_2$" to denote the concatenation of the paths p_1 and p_2, where p_1 is finite.

We use the notation "$\langle a, b, \ldots, k \rangle$" to denote a finite sequence of elements: $a,\ b \ldots, k$ and the notation "$s_1 \cdot s_2$", to denote the concatenation of the sequences s_1 and s_2.

A "trace" in a transition system is a sequence of adjacent states in the transition system. For example, $\langle s, q, s \rangle$ is an example trace in the transition system of Fig. 2.1(ii). The trace associated with a path p, denoted "$trace(p)$" is the sequence of states in it. For example, $trace(s \xrightarrow{a} s \xrightarrow{a} q) = \langle s, s, q \rangle$.

We use standard notation to deal with strings over an alphabet. For an alphabet Σ, we use the notations: ϵ to denote the empty string, Σ^* to denote the set of all finite strings over Σ, Σ^ω to denote the set of all infinite strings over Σ, Σ^∞ to denote the union of Σ^* and Σ^ω, and $u \cdot v$ or simply uv to denote the concatenation of strings u and v.

The "word" associated with a path p, denoted "$word(p)$" is the concatenation of edge labels in p. For example, $word(s \xrightarrow{a} s \xrightarrow{a} q) = aa$. We use the shorthand notation "$s \xrightarrow{w} q$" to denote a set of paths P from the state s to the state q in a transition system with $word(p) = w$ for each path p in P. For example, $s \xrightarrow{aa} q$ denotes the set of paths $\{s \xrightarrow{a} s \xrightarrow{a} q\}$ in the transition system of Fig. 2.1(ii).

The "language" associated with a transition system $\mathcal{S}$, denoted "$L(\mathcal{S})$" is the set of words associated with the set of all initial and finite paths in $\mathcal{S}$ (that is, $L(\mathcal{S}) = \bigcup_{p \in P} word(p)$, where P is the set of all initial and finite paths in $\mathcal{S}$). For example, the set $\{(ab)^*, (ab)^* \cdot a\}$ represents the language associated with the transition system of Fig. 2.1(iii).

We say a transition system $\mathcal{S}$ is *deterministic* if for each $p \in Q$ and $l \in \Sigma$, whenever $p \xrightarrow{l} q$ and $p \xrightarrow{l} q'$ we have $q = q'$. We say a transition system $\mathcal{S}$ is *closed* (or has no internal choice) if for each $p \in Q$ and $l, l' \in \Sigma$, whenever $p \xrightarrow{l} q$ and $p \xrightarrow{l'} q'$ we have $l = l'$. Fig. 2.1 illustrates different types of transition systems.

We use the standard symbols: $\mathbb{B}$, $\mathbb{N}$ and $\mathbb{Z}$ respectively to denote the set $\{0, 1\}$ of bit values, the set of natural numbers and the set of integers. For a finite set X, we use the notation X^i to denote the set of sequences/strings of length i over X.

$$
\begin{aligned}
QType_{\mathbb{B}} &= \{init, enq, deq, I_{init}, O_{init}, I_{enq}, O_{enq}, I_{deq}, O_{deq}\} \text{ where:} \\
I_{init} &= \{nil\} \\
O_{init} &= \{ok, e\} \\
I_{enq} &= \mathbb{B} \\
O_{enq} &= \{ok, fail, e\} \\
I_{deq} &= \{nil\} \text{ and} \\
O_{deq} &= \mathbb{B} \cup \{fail, e\}
\end{aligned}
$$

Figure 2.2: Example ADT type $QType_{\mathbb{B}}$.

2.2 Abstract data types

This section gives a formal definition of an Abstract Data Type (ADT). An ADT essentially gives a set of operations in its Application Programmer Interface (API). The clients of an ADT can interact with the ADT by calling operations in its API.

Definition 2.1 (ADT Type)**.** *An* ADT type *is a structure of the form:* $\mathcal{N}=(N, (I_n)_{n\in N}, (O_n)_{n\in N})$*, where* N *is a finite set of* operation names*,* I_n *and* O_n *are respectively the* input type *and* output type *associated with an operation* n *in* N*. An input (or output) type 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*.*

The *type* of an ADT defined above is considered as a *sort* in the literature on algebraic specifications of ADTs. However we will use the term *ADT type* to represent a tuple with a template as defined above, in the rest of this thesis.

We fix an ADT type $\mathcal{N}=(N, (I_n)_{n\in N}, (O_n)_{n\in N})$ for the rest of this chapter. As an example of an ADT type, consider a version of the queue example from Sec. 1.1, that stores bits rather than integers. Fig. 2.2 shows this ADT type which we call $QType_{\mathbb{B}}$. Here *nil* is a "dummy" argument for the operations *init* and *deq*, and *ok* is a dummy return value for the operations *init* and *enq*.

Definition 2.2 (ADT)**.** *A (deterministic)* ADT *of type* $\mathcal{N}$ *can be defined as a 4-tuple:* $\mathcal{A} = (Q, U, E, \{op_n\}_{n\in N})$*, 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* realization *of the operation* n *given by* $op_n : Q \times I_n \to Q \times O_n$ *such that* $op_n(E, -) = (E, e)$ *and* $op_n(p, a) = (q, e) \implies q = E$*. Thus if an operation returns the exceptional value the ADT moves to the exceptional state* E*, and all operations must keep it in* E *thereafter. Further, we require that the init operation depends only on its argument and not on the originating state: thus* $init(p, a) = init(q, a)$ *for each* $p, q \in Q \setminus \{E\}$ *and* $a \in I_{init}$*.*

$$
\begin{array}{lcl}
QADT_k & = & (Q, U, E, \{op_n\}_{n \in QType_{\mathbb{B}}}) \text{ where:} \\
Q & = & \{\epsilon\} \cup \bigcup_{i=1}^{k} \mathbb{B}^i \cup \{E\} \\
op_{init}(q, nil) & = & \begin{cases} (\epsilon, ok) & \text{if } q \neq E \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{enq}(q, a) & = & \begin{cases} (q \cdot a, ok) & \text{if } q \neq E \text{ and } |q| < k \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{deq}(q, nil) & = & \begin{cases} (q', b) & \text{if } q \neq E \text{ and } q = b \cdot q' \\ (E, e) & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 2.3: An ADT $QADT_k$ (parameterized by a length k) of type $QType_{\mathbb{B}}$.

Fig. 2.3 shows an example ADT called $QADT_k$ of type $QType_{\mathbb{B}}$. The subscript k denotes the maximum number of elements allowed in the queue. The state set Q comprises the set of all binary strings of lengths at most k, representing the contents of the queue; and the exception state E. The ADT $QADT_k$ enters the exception state in the following cases: (i) the *enq* operation is invoked when the queue is *full* ($|q| = k$) or (ii) the *deq* operation is invoked when the queue is *empty* ($|q| = 0$).

2.3 Client transition systems

In this section we describe transition systems whose action labels include calls to operations from an ADT type $\mathcal{N}$. Such transition systems are called $\mathcal{N}$-*client transition systems*. These transition systems are meant to model client programs of an ADT, like the C program `interp` of Fig. 1.5, which is a client of the `c-queue` ADT of Fig. 1.1.

Definition 2.3 ($\mathcal{N}$-client transition system)**.** *An $\mathcal{N}$-client transition system is a transition system whose action labels include "calls" to an ADT of type $\mathcal{N}$. It is of the form $\mathcal{S} = (Q, \Sigma, s, E, \Delta)$ where:*

- *Q is a set of states, with $s \in Q$ the start state.*
- *$\Sigma = \Sigma_l \cup \Sigma_{\mathcal{N}}$, where:*
  - *Σ_l is a finite set of* internal *or* local *action labels.*
  - *$\Sigma_{\mathcal{N}} = \{(n, a, b) \mid n \in N, a \in I_n, b \in O_n\}$ is the set of* operation call *labels corresponding to the ADT type $\mathcal{N}$. The action label (n, a, b) represents a call to operation n with input a that returns b.*
- *$E \in Q$ is an* exceptional *state, which is reached when an exceptional value is returned by the ADT.*
- *$\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation satisfying:*

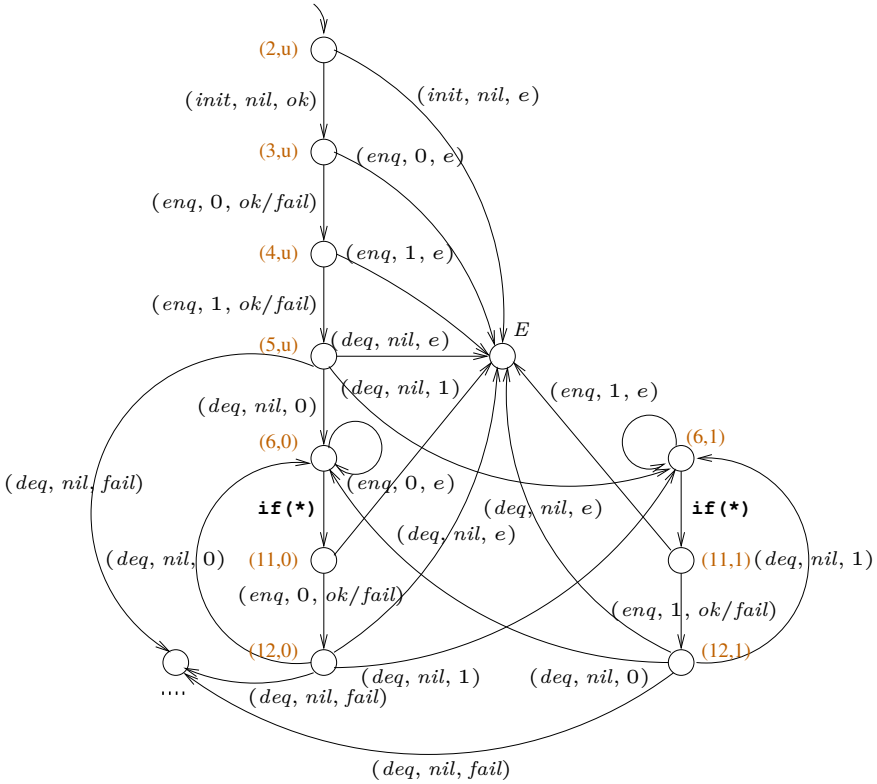


Figure 2.4: A $QType_{\mathbb{B}}$-client transition system $\mathcal{S}_{\texttt{interp}}$, which represents the C program `interp` of Fig. 1.5.

- $(p, c, E) \in \Delta$ *iff* $c = (n, a, e)$ *for some operation* n *in* N *and input* a *in* I_n *(thus an exceptional return value leads to the exceptional state and this is the only way to reach it).*
- $(p, -, q) \in \Delta$ *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, or in other words: an* $\mathcal{N}$*-client transition system must be ready to receive any value permitted by the output type of an operation* n *in* N*, whenever it calls* n*).*

Fig. 2.4 shows a $QType_{\mathbb{B}}$-client transition system called $\mathcal{S}_{\texttt{interp}}$, which represents the `interp` program of Fig. 1.5. In the sequel we will assume that client transition systems always initialize the ADT they are using before making calls to other operations on it. We note that an $\mathcal{N}$-client transition system is ready to receive any value in the output type O_n of an operation n in N, when it calls the ADT operation n. This is because, an implementation of an ADT operation n in N is allowed to return any value in the output type O_n.

Now we describe a transition system which obtained from an $\mathcal{N}$-client transition system $\mathcal{S}$, by plugging-in an ADT $\mathcal{A}$ of type $\mathcal{N}$. We can take the "product" of the transition systems $\mathcal{S}$ and $\mathcal{A}$, denoted $\mathcal{S}[\mathcal{A}]$, to obtain a new transition system.

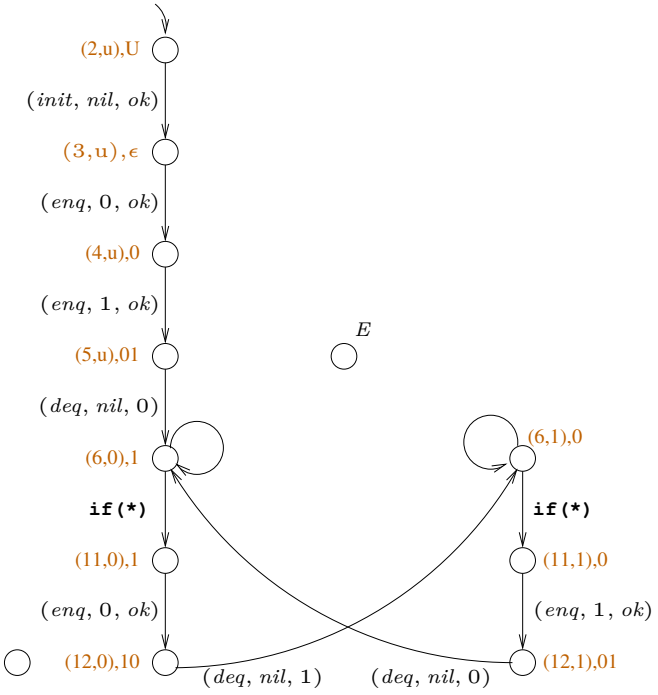


Figure 2.5: The transition system $\mathcal{S}_{\texttt{interp}}[QADT_2]$.

Definition 2.4. *Let $\mathcal{S} = (Q, \Sigma, s, E, \Delta)$ be an $\mathcal{N}$-client transition system and let $\mathcal{A} = (Q', U', E', \{op_n\}_{n \in N})$ be an ADT of type $\mathcal{N}$. Then we can define the transition system obtained by using $\mathcal{A}$ in $\mathcal{S}$, denoted "$\mathcal{S}[\mathcal{A}]$", to be the transition system $(Q \times Q', \Sigma, (s, U'), \Delta')$, where Δ' is given by:*

$$
\begin{array}{lll}
(p, p') \xrightarrow{l} (q, p') & \text{if} & l \in \Sigma_l \text{ and } p \xrightarrow{l} q \text{ in } \mathcal{S}. \\
(p, p') \xrightarrow{(n,a,b)} (q, q') & \text{if} & (n, a, b) \in \Sigma_{\mathcal{N}},\ p \xrightarrow{(n,a,b)} q \text{ in } \mathcal{S} \text{ and} \\
 & & op_n(p', a) = (q', b).
\end{array}
$$

Fig. 2.5 shows the transition system $\mathcal{S}_{\texttt{interp}}[QADT_2]$, which is obtained by plugging-in $QADT_2$ in the $QType_{\mathbb{B}}$-client transition system, $\mathcal{S}_{\texttt{interp}}$ of Fig. 2.4. Plugging-in an ADT of type $\mathcal{N}$ in an $\mathcal{N}$-client transition system makes it a closed transition system. For example, the transition system $\mathcal{S}_{\texttt{interp}}[QADT_2]$ of Fig. 2.5 is a closed transition system (see Sec. 2.1).

2.4 Refinement between ADTs

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ be an ADT of type $\mathcal{N}$. Then $\mathcal{A}$ induces a (deterministic) transition system $\mathcal{S}_{\mathcal{A}} = (Q, \Sigma_{\mathcal{N}}, U, \Delta)$, where the transition relation Δ is given by: $(p, (n, a, b), q) \in \Delta$ iff $op_n(p, a) = (q, b)$ and $(p, (n, a, e), E) \in \Delta$ iff $op_n(p, a) = (E, e)$. For example, Fig. 2.6 shows the transition system $\mathcal{S}_{QADT_2}$ induced by the ADT $QADT_2$ (see Fig. 2.3), where for simplicity, we do not show uninitialized sequences of operations.

We define the language of *initialized sequences of operation calls* of $\mathcal{A}$,

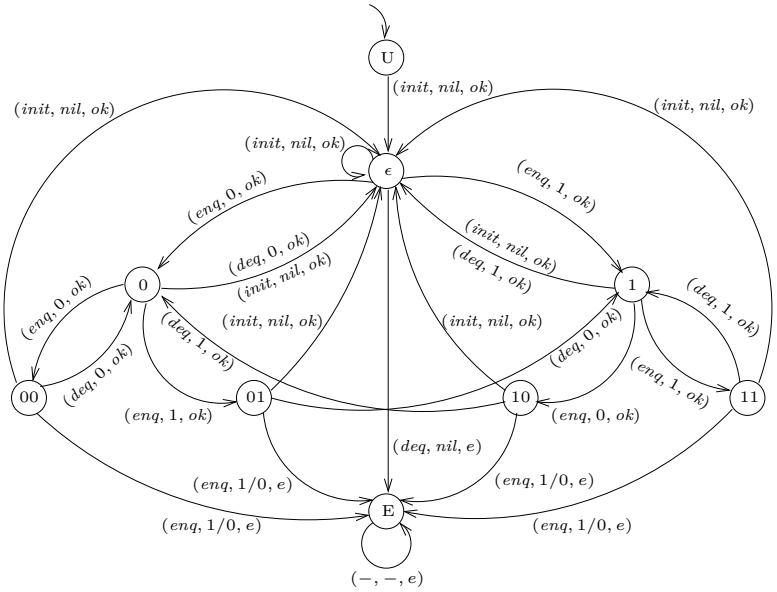


Figure 2.6: The transition system $\mathcal{S}_{QADT_2}$ induced by the ADT $QADT_2$.

denoted "$L_{init}(\mathcal{A})$", to be $L(\mathcal{S}_{\mathcal{A}}) \cap ((init, -, -) \cdot \Sigma^*_{\mathcal{N}})$. 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 $(-, -, e)$).

Definition 2.5 (Refinement)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be ADTs of type $\mathcal{N}$. We say $\mathcal{B}$* refines *$\mathcal{A}$, written "$\mathcal{B} \preceq \mathcal{A}$", iff each exception-free sequence of operations in $L_{init}(\mathcal{A})$ is also in $L_{init}(\mathcal{B})$.*

For example, Fig. 2.7 shows an ADT $QADT'_k$ which is a refinement of the ADT of Fig. 2.3. The operations *enq* and *deq* in $QADT'_k$ return *fail* when the queue is *full* or *empty* respectively. Note that $QADT_k$ fails with an exception value in such situations. Also, $QADT_k$ refines $QADT_l$ whenever $k \geq l$.

Proposition 2.1. *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be ADTs of type $\mathcal{N}$, such that $\mathcal{C} \preceq \mathcal{B}$, and $\mathcal{B} \preceq \mathcal{A}$. Then $\mathcal{C} \preceq \mathcal{A}$.*

Proof. Let $w \in (\Sigma \setminus \{-, -, e\})^*$ be an arbitrary exception-free sequence of operation calls admitted by the ADT $\mathcal{A}$. Then

$$\begin{aligned} w \in L_{init}(\mathcal{A}) &\implies w \in L_{init}(\mathcal{B}) \text{ [Since } \mathcal{B} \preceq \mathcal{A}] \\ &\implies w \in L_{init}(\mathcal{C}) \text{ [Since } \mathcal{C} \preceq \mathcal{B}]. \end{aligned}$$

Thus every exception-free sequence of operation calls in $L_{init}(\mathcal{A})$ is also in $L_{init}(\mathcal{C})$ and hence $\mathcal{C} \preceq \mathcal{A}$. □

2.5 Verification guarantee

We now describe the verification guarantee given by the above definition of refinement. Let $\mathcal{P}$ be a client program which makes calls to an ADT of type

$$
\begin{array}{lcl}
QADT'_k & = & (Q, U, E, \{op_n\}_{n \in QType_{\mathbb{B}}}) \text{ where} \\
Q & = & \{\epsilon\} \cup \bigcup_{i=1}^{k} \mathbb{B}^i \cup \{E\} \\
op_{init}(q, nil) & = & \begin{cases} (\epsilon, ok) & \text{if } q \neq E \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{enq}(q, a) & = & \begin{cases} (q \cdot a, ok) & \text{if } q \neq E \text{ and } |q| < k \\ (q, fail) & \text{if } q \neq E \text{ and } |q| = k \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{deq}(q, nil) & = & \begin{cases} (q', b) & \text{if } q \neq E \text{ and } q = b \cdot q' \\ (q, fail) & \text{if } q \neq E \text{ and } q = \epsilon \\ (E, e) & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 2.7: An ADT $QADT'_k$, which refines $QADT_k$ by returning *fail* in some of the cases where $QADT_k$ returns an exception.

$\mathcal{N}$, and let $\mathcal{A}$ and $\mathcal{A}'$ be ADTs of the same type $\mathcal{N}$ such that $\mathcal{A}' \preceq \mathcal{A}$. We show in this section that $\mathcal{P}[\mathcal{A}]$ and $\mathcal{P}[A']$ satisfy exactly the same set of temporal properties when the exception state is non-reachable in $\mathcal{P}[\mathcal{A}]$.

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ and $\mathcal{A}' = (Q', U', E', \{op'_n\}_{n \in N})$ be two ADTs of type $\mathcal{N}$ such that $\mathcal{A}'$ refines $\mathcal{A}$ and let $\mathcal{T} = (R, \Sigma_l \cup \Sigma_{\mathcal{N}}, s, E_{\mathcal{T}}, \Delta)$ be an $\mathcal{N}$-client transition system. We fix $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{T}$ as above for the rest of this section. There is a natural relation σ between the states of $\mathcal{A}'$ and $\mathcal{A}$ such that $(q', q) \in \sigma$, 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 say that $U \xrightarrow{w} q$ in $\mathcal{A}$ iff $U \xrightarrow{w} q$ in $\mathcal{S}_{\mathcal{A}}$. We can use this relation to define a kind of isomorphism σ' between $\mathcal{T}[\mathcal{A}']$ and $\mathcal{T}[\mathcal{A}]$: a state (r, q') of $\mathcal{T}[\mathcal{A}']$ and a state (s, q) of $\mathcal{T}[\mathcal{A}]$ are related by σ', iff $r = s$ and $\sigma(q', q)$ holds. Thus when two states are related by σ' the local states of the client transition system $\mathcal{T}$ in them are the same.

Theorem 2.1. *Let $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{T}$ and σ' be as above. Then σ' is an isomorphism in the following sense:*

1. *if $((r, p'), (r, p)) \in \sigma'$, and $(r, p) \xrightarrow{l} (s, q)$ in $\mathcal{T}[\mathcal{A}]$ with l a non-exception action label, then there exists (s, q') in $\mathcal{T}[\mathcal{A}']$ such that $(r, p') \xrightarrow{l} (s, q')$ and $((s, q'), (s, q)) \in \sigma'$.*

2. *Conversely, if $((r, p'), (r, p)) \in \sigma'$, and $(r, p') \xrightarrow{l} (s, q')$ in $\mathcal{T}[\mathcal{A}']$, then either there exists a state (s, q) in $\mathcal{T}[\mathcal{A}]$ such that $(r, p) \xrightarrow{l} (s, q)$ and $((s, q'), (s, q)) \in \sigma'$, or l is of the form (n, a, b) and $(r, p) \xrightarrow{(n,a,e)} (-, E)$ in $\mathcal{T}[\mathcal{A}]$.*

Proof.

1. Suppose $((r, p'), (r, p)) \in \sigma'$, and $(r, p) \xrightarrow{l} (s, q)$ in $\mathcal{T}[\mathcal{A}]$ with l a non-exception action label. There are two possibilities to consider here:

(a) l is a local action label in $\mathcal{T}$. This means that $r \xrightarrow{l} s$ in $\mathcal{T}$ and then it follows from the definition of $\mathcal{T}[\mathcal{A}]$ that $p = q$. Now it follows from the definition of $\mathcal{T}[\mathcal{A}']$ that $(r, p') \xrightarrow{l} (s, p')$. Also we have $((s, p'), (s, p)) \in \sigma'$, since $(p', p) \in \sigma$ by assumption. Hence we are done.

(b) l is of the form (n, a, b). Thus we have $(r, p) \overset{(n,a,b)}{\rightarrow} (s, q)$ in $\mathcal{T}[\mathcal{A}]$. Then by definition of $\mathcal{T}[\mathcal{A}]$, we have $r \overset{(n,a,b)}{\rightarrow} s$ in $\mathcal{T}$ and $p \overset{(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$. It follows from the definition of σ that there exists an exception-free initial sequence of operations w such that $U \xrightarrow{w} p$ in $\mathcal{A}$ and $U \xrightarrow{w} p'$ in $\mathcal{A}'$, since we have $(p', p) \in \sigma$. Thus we have $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$. This implies that there exists a q' in Q' such that $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$, since $\mathcal{A}' \preceq \mathcal{A}$ by assumption. Therefore it follows that $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$, since $\mathcal{A}'$ is deterministic, $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$ and $U' \xrightarrow{w} p'$ in $\mathcal{A}'$. Now it follows from the definition of $\mathcal{T}[\mathcal{A}']$ that $(r, p') \overset{(n,a,b)}{\rightarrow} (s, q')$ in $\mathcal{T}[\mathcal{A}']$, since we have $r \overset{(n,a,b)}{\rightarrow} s$ in $\mathcal{T}$. Also it follows from the definition of σ that $(q', q) \in \sigma$, since we have $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$ and $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$. This implies that $((s, q'), (s, q)) \in \sigma'$ and hence we are done.

2. Conversely, suppose $((r, p'), (r, p)) \in \sigma'$, and $(r, p') \xrightarrow{l} (s, q')$ in $\mathcal{T}[\mathcal{A}']$. Here also we need to consider two cases:

(a) l is a local action label in $\mathcal{T}$. This means that $r \xrightarrow{l} s$ in $\mathcal{T}$ and then it follows from the definition of $\mathcal{T}[\mathcal{A}']$ that $p' = q'$. Now it follows from the definition of $\mathcal{T}[\mathcal{A}]$ that $(r, p) \xrightarrow{l} (s, p)$. Also we have $((s, p'), (s, p)) \in \sigma'$, since $(p', p) \in \sigma$ by assumption. Hence we are done.

(b) l is of the form (n, a, b). Thus we have $(r, p') \overset{(n,a,b)}{\rightarrow} (s, q')$ in $\mathcal{T}[\mathcal{A}']$. Then by definition of $\mathcal{T}[\mathcal{A}']$, we have $r \overset{(n,a,b)}{\rightarrow} s$ in $\mathcal{T}$ and $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$. It follows from the definition of σ that there exists an initial exception-free sequence of operations w such that $U \xrightarrow{w} p$ in $\mathcal{A}$ and $U' \xrightarrow{w} p'$ in $\mathcal{A}'$, since we have $(p', p) \in \sigma$. Thus we have $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$. This implies that either there exists a q in Q such that $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$, or that $U \overset{w\cdot(n,a,e)}{\rightarrow} E$ in $\mathcal{A}$, since $\mathcal{A}' \preceq \mathcal{A}$ by assumption and $U \xrightarrow{w} p$ in $\mathcal{A}$. The latter implies that $p \overset{(n,a,e)}{\rightarrow} E$ in $\mathcal{A}$, since $\mathcal{A}$ is deterministic and hence the result immediately follows from the definition of $\mathcal{T}[\mathcal{A}]$. The former implies that $p \overset{(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$, since $\mathcal{A}$ is deterministic, $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$ and $U \xrightarrow{w} p$ in $\mathcal{A}$. Hence it follows from the definition of $\mathcal{T}[\mathcal{A}]$ that $(r, p) \overset{(n,a,b)}{\rightarrow} (s, q)$ in $\mathcal{T}[\mathcal{A}]$, since we have $r \overset{(n,a,b)}{\rightarrow} s$ in $\mathcal{T}$. Also it follows from the definition of

σ that $(q', q) \in \sigma$, since we have $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$ and $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$. Hence we have $((s, q'), (s, q)) \in \sigma'$ and we are done.

□

Let $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{T}$ and σ' be as above. Then a path $p' = v'_0 \overset{a'_1}{\rightarrow} v'_1 \cdots \overset{a'_m}{\rightarrow} v'_m$ in $\mathcal{T}[\mathcal{A}']$ is said to be σ'-*equivalent* to a path $p = v_0 \overset{a_1}{\rightarrow} v_1 \cdots \overset{a_n}{\rightarrow} v_n$ in $\mathcal{T}[\mathcal{A}]$, written "$p' \overset{\sigma'}{\equiv} p$", iff $word(p') = word(p)$ (see Sec. 2.1) and $(v'_i, v_i) \in \sigma'$. We say that two traces: $t' = \langle v'_0, v'_1, \cdots, v'_m \rangle$ and $t = \langle v_0, v_1, \cdots, v_n \rangle$ in $\mathcal{T}[\mathcal{A}']$ and $\mathcal{T}[\mathcal{A}]$ respectively are σ'-equivalent, written "$t' \overset{\sigma'}{\equiv} t$", iff $m = n$ and $(v'_i, v_i) \in \sigma'$.

Corollary 2.1. *Let $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{T}$ and σ' be as above. Then Theorem 2.1 implies the following:*

1. *For every path*[1] *p in $\mathcal{T}[\mathcal{A}]$, there exists a path p' in $\mathcal{T}[\mathcal{A}']$ such that $p' \overset{\sigma'}{\equiv} p$.*
2. *For every path p' in $\mathcal{T}[\mathcal{A}']$, either there exists a path p in $\mathcal{T}[\mathcal{A}]$ such that $p' \overset{\sigma'}{\equiv} p$ or there exists a path p_{pref} in $\mathcal{T}[\mathcal{A}]$ such that p' is of the form $p'_{pref} \overset{(n,a,b)}{\rightarrow} p'_{suf}$ with $p'_{pref} \overset{\sigma'}{\equiv} p_{pref}$ and $p_{pref} \overset{(n,a,e)}{\rightarrow} E \in \mathcal{T}[\mathcal{A}]$.*

The condition 1 above follows from condition 1 of Theorem 2.1 and the condition 2 follows from condition 2 of Theorem 2.1, since the start states of $\mathcal{T}[\mathcal{A}]$ and $\mathcal{T}[\mathcal{A}']$ are related by σ'.

We follow a notion of *Linear Time* (LT) properties similar to the one given in *Principles of model checking* [8], to formalize the properties preserved by our notion of refinement. Given a vocabulary Σ, an LT property φ is a subset of the set of all strings in Σ^∞ (see Sec. 2.1). For instance, consider the vocabulary Σ as the state set Q in the transition system of Fig. 2.6, and an LT property ψ: "it is always the case that the state label is not E and the length of the state label (binary string) is at most 2". This may be an interesting LT property that we want to verify about the transition system of Fig. 2.6. Given an LT property φ over Σ and a string t in Σ^∞, t is in φ if t satisfies φ.

A given path p in a transition system satisfies φ, if $trace(p)$ (sequence of states in p) satisfies φ. For instance, the path corresponding to the trace $\langle U \rangle \cdot \langle \epsilon, 0 \rangle^\omega$ satisfies the property ψ above. A transition system satisfies an LT property φ, if all its initial paths satisfy φ. The transition system of Fig. 2.6 does not satisfy the above property ψ, since some of its initial paths like the path corresponding to the trace $\langle U, \epsilon, E \rangle$ do not satisfy ψ.

Let t be a string in $(R \times S)^\infty$. Then the notation "$t \restriction R$" represents the string obtained by projecting each symbol in the string t to the first component. Thus, if $t = \langle (r_1, s_1), (r_2, s_2), \cdots, (r_k, s_k) \rangle$, then $t \restriction R = \langle r_1, r_2, \cdots, r_k \rangle$. Let $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{T}$ be as above. Then a trace t' in $\mathcal{T}[\mathcal{A}']$ is said to be *locally-equivalent* to a trace t in $\mathcal{T}[\mathcal{A}]$, written "$t' \overset{l}{\equiv} t$" iff $t' \restriction R = t \restriction R$. Recall that R is the set of states in $\mathcal{T}$. That is, two traces t' and t in $\mathcal{T}[\mathcal{A}']$ and $\mathcal{T}[\mathcal{A}]$ respectively are

[1] *a path in these conditions is assumed to not contain an exception (a transition label of the form $(-, -, e)$).*

locally equivalent if they have the same number of states and also they have the same local state (state in $\mathcal{T}$) at the i^{th} position in t' and t for all i in the set $\{1, 2, \ldots |t|\}$.

Definition 2.6 (Local LT property)**.** *Let $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{T}$ be as above. Then a* local LT property *over a state vocabulary $(R \times S)$ is an LT property φ over $(R \times S)$ such that for any two traces t' and t in $(R \times S)^\infty$ with $t' \overset{l}{\equiv} t$, t' satisfies φ iff t satisfies φ.*

The properties about a client transition system preserved by our notion of refinement is captured in the following theorem.

Theorem 2.2. *Let $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{T}$ be as above. Let φ be a local LT property over the vocabulary $(R \times (Q \cup Q'))$. Then if $\mathcal{T}[\mathcal{A}]$ satisfy φ, either $\mathcal{T}[\mathcal{A}']$ will also satisfy φ or each trace[2] violating φ in $\mathcal{T}[\mathcal{A}']$ contains a prefix with a locally-equivalent trace leading to the exception state in $\mathcal{T}[\mathcal{A}]$. In particular, if the client $\mathcal{T}$ does not see an exception with the abstract ADT $\mathcal{A}$, then both $\mathcal{T}[\mathcal{A}]$ and $\mathcal{T}[\mathcal{A}']$ satisfy exactly the same set of local LT properties.*

Proof. Suppose t' is a trace in $\mathcal{T}[\mathcal{A}']$ violating φ. It follows from Corollary 2.1 that one of the following conditions is true:

1. there exists a trace t in $\mathcal{T}[\mathcal{A}]$ such that $t' \overset{l}{\equiv} t$.
2. there exists a trace t_{pref} in $\mathcal{T}[\mathcal{A}]$ such that t' is of the form $t'_{pref} \cdot t'_{suf}$ with $t'_{pref} \overset{l}{\equiv} t_{pref}$ and $t_{pref} \cdot \langle E \rangle \in \mathcal{T}[\mathcal{A}]$.

Now condition 1 above cannot be true since it contradicts the assumption that $\mathcal{T}[\mathcal{A}]$ satisfies φ. Hence the condition 2 must be true and which gives a prefix for t' as required in the theorem and hence we are done. □

For example, suppose the `interp` program of Fig. 1.5, satisfies the following properties when it calls the queue operations from the `z-queue` ADT of Fig. 1.3: (i) the exception state is not reachable and (ii) the value of `t` is either `0` or `1` at line `6`. This program is of the form `interp`[`z-queue`]. Now we can infer from our theory of refinement that `interp`[`c-queue`] satisfies the above properties when `c-queue` refines `z-queue`.

Let $\mathcal{A}$ be an ADT of type $\mathcal{N}$ and let $\mathcal{P}$ be a client program that calls operations from the ADT $\mathcal{A}$ such that every ADT call returns a non-exception value. We mention below some example properties about the client program with the abstract ADT ($\mathcal{P}[\mathcal{A}]$), preserved by our notion of refinement.

1. Exception state is not reachable.
2. Satisfies a local assertion ψ.

[2] *a trace here is assumed to begin with the start state of the transition system.*

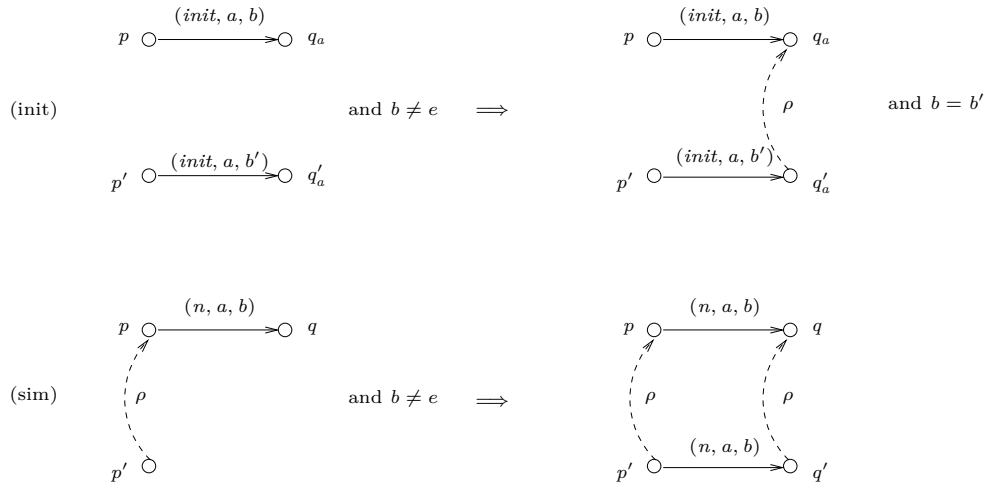


Figure 2.8: Illustrating the equivalent condition (RC) for refinement.

3. *Fairness*: Every occurrence of the event "request", which is represented by a state in which a local variable say `requested` has the value *true*, is eventually followed by an event "granted", which is represented by a state in which a local variable say `granted` has the value *true*.

4. *Mutual exclusion*: At most one process is allowed to enter a critical section (this may be encoded as a condition on a local counter variable representing the number of processes in a critical section).

5. Satisfies a local invariant ϕ.

It is easy to check that the above properties are local LT properties.

2.6 Equivalent refinement condition

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n\in N})$ and $\mathcal{A}' = (Q', U', E', \{op_n\}_{n\in N})$ be ADTs of type $\mathcal{N} = (N, (I_n)_{n\in N}, (O_n)_{n\in N})$. We formulate an *equivalent* condition for $\mathcal{A}'$ to refine $\mathcal{A}$, based on an "abstraction relation" that relates states of $\mathcal{A}'$ to states of $\mathcal{A}$. We say $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (RC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

(init) Let $a \in I_{init}$ and let (q_a, b) and (q'_a, b') be the resultant states and outputs after an $init(a)$ 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 $p \overset{(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$ with $b \neq e$, then there exists $q' \in Q'$ such that $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$ with $(q', q) \in \rho$.

Fig. 2.8 illustrates the equivalent refinement condition (RC).

Theorem 2.3. *Let $\mathcal{A}$ and $\mathcal{A}'$ be two ADTs of type $\mathcal{N}$. Then $\mathcal{A}' \preceq \mathcal{A}$ iff they satisfy condition (RC).*

Proof.

($\Leftarrow$) Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n\in N})$ and $\mathcal{A}' = (Q', U', E', \{op'_n\}_{n\in N})$ be two ADTs of type $\mathcal{N}$, and let $\rho \subseteq Q' \times Q$ be an abstraction relation, such that $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (RC) with respect to ρ. We first prove the following claim.

Claim 2.1. *For any states $p, q \in Q$, $p' \in Q'$ and an exception-free initial sequence of operations w of the form $w = (init, a, b) \cdot u$, if $p \xrightarrow{w} q$ in $\mathcal{A}$, then there exists a state q' in Q' such that $p' \xrightarrow{w} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$.*

Proof. We prove this claim by induction on the length of u.

(Basis) Let $|u| = 0$. Then $w = (init, a, b)$, where $a \in I_{init}$ and $b \in O_{init}$. Now it follows from $(init)$ of condition (RC) that there exists a q' in Q' such that $p' \xrightarrow{(init,a,b)} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$. Hence we are done.

(Ind. step) Let $|u| = k+1$. Then w is of the form: $w = (init, a, b) \cdot v \cdot (n, a_n, b_n)$, where $|v| = k$, $n \in N$, $a_n \in I_n$, $b_n \in O_n$ and let $p \xrightarrow{(init,a,b)\cdot v} r \xrightarrow{(n,a_n,b_n)} q$ be the path corresponding to w in $\mathcal{A}$. It follows from the induction hypothesis that there exists a state $r' \in Q'$ such that $p' \xrightarrow{(init,a,b)\cdot v} r'$ in $\mathcal{A}'$ and $(r', r) \in \rho$. Now by (sim) of condition (RC), there exists a state $q' \in Q'$ such that $r' \xrightarrow{(n,a_n,b_n)} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$. Thus we have $p' \xrightarrow{w} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$. Hence we are done.

□

Now it follows from Claim 2.1 that whenever $w \in L_{init}(\mathcal{A})$ for any exception-free sequence of operations w, we also have $w \in L_{init}(\mathcal{A}')$. This proves that $\mathcal{A}' \preceq \mathcal{A}$.

($\Rightarrow$) Conversely suppose $\mathcal{A}' \preceq \mathcal{A}$. Let ρ be the relation $\sigma \subseteq Q' \times Q$ defined in the proof of our verification guarantee in Sec. 2.5. Recall that $(q', q) \in \rho$, iff there exists an exception-free initial sequence of operation calls w such that $U \xrightarrow{w} q$ in $\mathcal{A}$ and $U' \xrightarrow{w} q'$ in $\mathcal{A}'$.

To show that ρ satisfies (RC-init), suppose $p \xrightarrow{(init,a,b)} q$ in $\mathcal{A}$. Then since $\mathcal{A}'$ refines $\mathcal{A}$, we must have $p' \xrightarrow{(init,a,b)} q'$ in $\mathcal{A}'$ for some $q' \in Q'$. Also, by definition of ρ, we have $(q', q) \in \rho$. Hence ρ satisfies (RC-init). Recall that the *init* operation is independent of the state on which it is invoked.

We now show that ρ satisfies the condition (RC-sim). Suppose $(p', p) \in \rho$, and $p \xrightarrow{(n,a,b)} q$ in $\mathcal{A}$ with $b \neq e$. By definition of ρ, we know that there exists an exception-free initial sequence of operations w such that $U \xrightarrow{w} p$ in $\mathcal{A}$ and $U' \xrightarrow{w} p'$ in $\mathcal{A}'$. Since $p \xrightarrow{(n,a,b)} q$ in $\mathcal{A}$ by assumption, we have

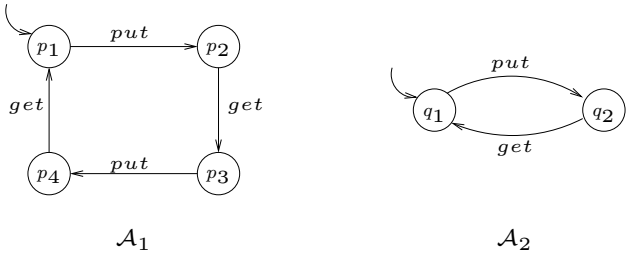


Figure 2.9: A two-state ADT and a four-state ADT supporting the same set of operation sequences $\{(put \cdot get)^*, (put \cdot get)^* \cdot put\}$.

$U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$. Then by the assumption $\mathcal{A}' \preceq \mathcal{A}$, we have $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$ for some $q' \in Q'$. Now it follows that $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$, since $\mathcal{A}'$ is deterministic and we have $U' \overset{w}{\rightarrow} p'$ in $\mathcal{A}'$. Now by definition of ρ we have $(q', q) \in \rho$, since we have $U \overset{w\cdot(n,a,b)}{\rightarrow} q$ in $\mathcal{A}$ and $U' \overset{w\cdot(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}'$. Hence ρ satisfies (RC-sim).

□

In the above, ρ needs to be a relation and a function does not suffice. For example, consider an ADT type with the set of operations $\{put, get\}$. There can be a two-state ADT and a four-state ADT, each of which permits the set of operation sequences $\{(put \cdot get)^*, (put \cdot get)^* \cdot put\}$ (see Fig. 2.9). Each one of these ADTs refines the other, since they both allows the same set of operation sequences. However there is no abstraction function possible from the states of the two-state ADT to the states of the four-state ADT. Similarly a bijective relation does not suffice, since for example there is no bijective relation possible between the states of the ADTs in this example.

2.7 Related work

The notion of refinement in Event-B [4] satisfies the requirement that every legal behaviour of a concrete ADT is also a legal behaviour of the abstract ADT. One of the two conditions for refinement in this notion requires that the abstract precondition is satisfied whenever the concrete precondition is satisfied [3]. Thus precondition strengthening is allowed in this notion of refinement and hence a concrete ADT may not allow some legal operations allowed by an abstract ADT. In our opinion, this notion is not strong enough to satisfy the existing clients of an abstract ADT, since this notion gives the concrete ADT the liberty to not allow certain exception-free behaviors allowed by the abstract ADT. For instance, in this notion of refinement $QADT_2$ refines $QADT_3$, since $QADT_3$ can simulate $QADT_2$. However clients of $QADT_3$ may not be happy with $QADT_2$, since it cannot allow certain behaviors like "$(init, nil, ok)(enq, 1, ok)(enq, 1, ok)(enq, 0, ok)$" which $QADT_3$ allows. Moreover this notion does not allow what we consider to be valid refinements like

$QADT_3$ refining $QADT_2$, since $QADT_2$ cannot simulate $QADT_3$.

Liskov and Wing give a notion of refinement in the form of behavioral subtyping [37]. In a deterministic setting, their refinement notion requires both the abstract and concrete ADTs to simulate each other. This requirement is too strong as it does not allow certain refinements, which are valid in the sense that they preserve the properties required to satisfy the existing clients of the abstract ADT. For instance, $QADT_3$ cannot refine $QADT_2$, since $QADT_2$ cannot simulate $QADT_3$.

The notion of refinement in VDM [13, 30], Z [5, 46] and [26] preserves the properties required to satisfy the existing clients of the abstract ADT. However in this notion of refinement, a program (or application context) is considered as a sequence of ADT operations starting with an *initialization* operation and ending with a *finalization* operation, where an initialization operation converts a global state (client's state) into a local ADT state and a finalization operation converts a local ADT state back to a global state. Thus in this notion of refinement a client program needs to remember the local state of an ADT in the client's state in order to invoke the initialization operation whenever the client requires to interact with the ADT at a later point of time. On the other hand our notion does not require the client to maintain the ADT state. Also we proved a substitutivity result for ADTs which enables us to propose a compositional way of reasoning about complex ADT implementations which make use of sub-ADTs.

A trace-based notion of refinement similar to our notion was discussed in the technical report [1] by Hoare et al. However they found to have abandoned this notion and have presented a notion similar to the Z notion of refinement in the published version [26]. Also they lack a compositional way of reasoning about complex ADT implementations.

A trace-based notion of refinement between software libraries is presented in [45] in the form of backward compatibility, where the concrete simulates the observable behaviors allowed by the abstract. This notion preserves the properties required to satisfy the existing clients of the abstract library in the above sense. However their theory lacks the presence of an abstract model of the library and hence the properties about the clients need to be proved using the concrete library implementation. Also their theory cannot handle liveness properties. They make an imprecise claim that there is a bijective relation between the states of compatible libraries. This is not true in general as illustrated by the above example in terms of a two-state ADT and a four-state ADT supporting the same set of operation sequences.

Chapter 3

ADT Transition Systems

This chapter begins with the definition of a special kind of transition system called an ADT Transition System (ADT-TS), which models an imperative language implementation of an ADT. ADT transition systems help us to extend our trace-based notion of refinement for ADTs to ADT implementations as well. We define a notion of refinement between ADT transition systems and also describe an equivalent condition for refinement between ADT transition systems.

We are interested in arguing about complex ADT implementations which use several layers of *sub-ADT*s. By the term sub-ADT we simply mean a stand-alone ADT that is used in the implementation of another ADT, and not in the sense of a *sub-algebraic* structure.

We introduce the notion of a client ADT-TS to model an ADT implementation which in turn calls operations of a sub-ADT. We then state and prove a substitutivity theorem, which gives us a compositional way of reasoning about complex ADT implementations.

3.1 ADT transition system

Consider the program `c-queue` of Fig. 1.1. This program implements an ADT of a type $QType_{\mathbb{Z}}$, which is similar to the ADT type $QType_{\mathbb{B}}$ defined in Fig. 2.2, except that the input and output values are integers rather than bits. We use the term "method" to denote an imperative language implementation of an ADT operation. For example, the `init` method in the program `c-queue` of Fig. 1.1, implements the operation *init* in the ADT type $QType_{\mathbb{Z}}$.

A program like `c-queue` which implements an ADT can be modeled by what we call an "ADT transition system". How can we model a program like `c-queue` as a transition system? We could represent a state in the transition system as an n-tuple "$(loc, v(x_1), \ldots, v(x_{n-1}))$", where *loc* is the line number of the next statement to execute and v is a valuation to the variables $x_1, \ldots, x_{n-1}$ in the program. For example, the six-tuple $(8, \langle\rangle, 0, 0, u, nil)$, represents the state of the program `c-queue` just before executing the statement at line `8` in the `init` method. In this state, the array `A` is empty, the variables `beg` and `end` have the same value 0, the variable `len` has the uninitialized value u and

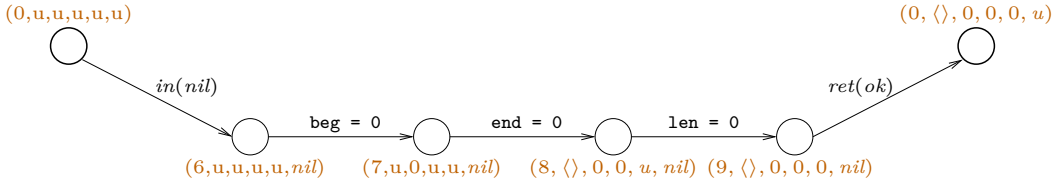


Figure 3.1: Transition system induced by the `init` method in the program `c-queue` of Fig. 1.1.

the argument has the dummy value *nil*. We can use transitions to represent the state changes induced by the statements of the program. For example, the transition $(\texttt{8}, \langle\rangle, 0, 0, u, nil) \xrightarrow{\texttt{len = 0}} (\texttt{9}, \langle\rangle, 0, 0, 0, nil)$, represents the state change induced by the statement at line `8` in the program `c-queue`.

It is not difficult to see that, for a given input, each method induces a deterministic and closed transition system (see Sec. 2.1). Fig. 3.1 shows the transition system induced by the `init` method in the program `c-queue`. The `init` method does not take any input. The transition labels: "$in(nil)$" and "$ret(ok)$" respectively model the method reading the dummy input (argument) value *nil* and the method returning the dummy output value *ok*. In the transition system of Fig. 3.1, the first and the last states are the only states *visible* to the clients of the ADT, which we show at the top level in an ADT-TS as depicted in this figure. The visible states of an ADT-TS are called *complete*, in the sense that the ADT operations are complete in such states. We use statement number 0 to represent a complete state. Other states are internal states of the ADT-TS and are of no interest to the clients of the ADT.

We fix the ADT type $\mathcal{N}=(N, (I_n)_{n\in N}, (O_n)_{n\in N})$ for the rest of this chapter.

Definition 3.1 (ADT transition system)**.** *An* ADT transition system *of type* $\mathcal{N}$ *can be defined as a 5-tuple:* $\mathcal{S} = (Q_c, Q_l, \Sigma, U, \{\delta_n\}_{n\in N})$*, where:*

- Q_c *is the set of "complete" states of the ADT-TS (where an ADT operation is complete) and* Q_l *is the set of "incomplete" or "local" states of the ADT-TS. The set of states* Q *of the ADT-TS is the disjoint union of* Q_c *and* Q_l*.*
- $\Sigma = \Sigma_l \cup \Gamma^i_{\mathcal{N}} \cup \Gamma^o_{\mathcal{N}}$*, where:*
  - Σ_l *is a finite set of* internal *or* local *action labels (these basically model executable statements in a method, like* `beg = 0` *in the* `init` *method of Fig. 1.1).*
  - $\Gamma^i_{\mathcal{N}} = \{in(a) \mid \exists\, n \in N \text{ with } a \in I_n\}$ *is the set of* input *labels corresponding to the ADT type* $\mathcal{N}$*. The action label* $in(a)$ *models a method reading an input* a*.*
  - $\Gamma^o_{\mathcal{N}} = \{ret(b) \mid \exists\, n \in N \text{ with } b \in O_n\}$ *is the set of* return *labels corresponding to the ADT type* $\mathcal{N}$*. The action label* $ret(b)$ *models a method returning an output* b*.*
- $U \in Q_c$ *is an* uninitialized *state.*

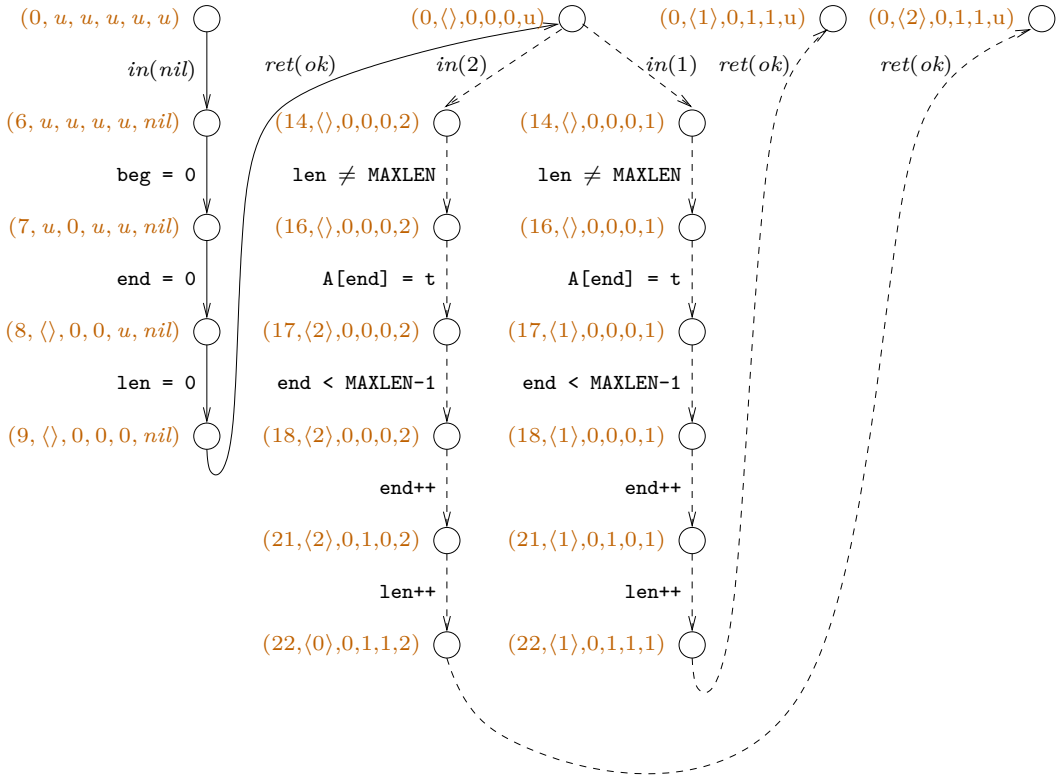


Figure 3.2: A part of the ADT-TS modeling the program `c-queue` of Fig. 1.1, with solid edges representing $\delta_{\texttt{init}}$ and dashed edges representing $\delta_{\texttt{enq}}$. Here the value of the constant *MAXLEN* is assumed to be greater than one. The state label is a six-tuple representing a valuation for: the *line number*, the array `A`, the variables `beg`, `end` and `len`, and the argument for the concerned function.

- *For each $n \in N$, $\delta_n \subseteq Q \times \Sigma \times Q$, is a transition relation that implements the operation n. This is meant to model a method like the* `init` *method of Fig. 1.1. It must satisfy the following constraints:*
  - *it is deterministic (see Sec. 2.1).*
  - *it is closed, except for the input actions in $\Gamma^i_{\mathcal{N}}$ for which it must be complete (that is there exists an input transition for each input $a \in I_n$ - in other words: δ_n is ready to accept any value in the input type I_n).*
  - *Each transition labeled by an input action in $\Gamma^i_{\mathcal{N}}$ begins from a Q_c state and each transition labeled by a return action in $\Gamma^o_{\mathcal{N}}$ ends in a Q_c state. All other transitions begin and end in Q_l states.*
  - *No transition is labeled ret(e). Thus an ADT-TS cannot explicitly return the exceptional value.*

Fig. 3.2 shows a part of the ADT-TS of type $QType_{\mathbb{Z}}$, induced by the program `c-queue` of Fig. 1.1, which implements an ADT of type $QType_{\mathbb{Z}}$.

We fix $\mathcal{S} = (Q_c, Q_l, \Sigma, U, \{\delta_n\}_{n \in N})$ for the rest of this chapter.

$$
\begin{aligned}
op_{init}((0, u, u, u, u, u), nil) &= ((0, \langle\rangle, 0, 0, 0, u), ok) \\
op_{enq}((0, \langle\rangle, 0, 0, 0, u), 1) &= ((0, \langle 1\rangle, 0, 1, 1, u), ok) \\
op_{enq}((0, \langle\rangle, 0, 0, 0, u), 2) &= ((0, \langle 2\rangle, 0, 1, 1, u), ok)
\end{aligned}
$$

Figure 3.3: Operation relations in the ADT induced by the ADT-TS of Fig. 3.2.

Let $\mathcal{S}$ be an ADT-TS of type $\mathcal{N}$. Then for each n in N, for each a in I_n, and for each q in Q_c: "$\delta_{n(q,a)}$" denotes the path which begins with the transition label $in(a)$ from the state q in $\mathcal{S}$. The path $\delta_{n(q,a)}$ is deterministic and closed (see Sec. 2.1). The path $\delta_{n(q,a)}$ may be an infinite path (representing a non-terminating loop), a finite path ending in the exception state E (representing exceptions like division by zero) or a finite path ending in a state in Q_c.

For example in Fig. 3.2, $\delta_{\mathtt{enq}((0,\langle\rangle,0,0,0,u),1)}$ is a finite path ending in the complete state $(0, \langle 1\rangle, 0, 1, 1, u)$.

Let $\mathcal{S}$ be an ADT-TS of type $\mathcal{N}$. Then $\delta_{\mathcal{N}}$ is the set of paths: $\{\delta_{n(q,a)} \mid \exists\, q \in Q_c \text{ and } \exists\, n \in N \text{ with } a \in I_n\}$ in $\mathcal{S}$. An ADT-TS can be viewed as a collection of $\delta_{\mathcal{N}}$ paths.

A $\delta_{\mathcal{N}}$ path is called *exception-free*, if it is a finite path ending in a state in Q_c. We note that an exception-free $\delta_{\mathcal{N}}$ path gives a non-exception return value. For example, in the ADT-TS of Fig. 3.2, the $\delta_{QType_{\mathbb{Z}}}$ path $\delta_{\mathtt{enq}((0,\langle\rangle,0,0,0,u),2)}$ is exception-free.

ADT induced by an ADT-TS: An ADT-TS of type $\mathcal{N}$ like $\mathcal{S}$ above *induces* an ADT $\mathcal{A}_{\mathcal{S}}$ of type $\mathcal{N}$ given by $\mathcal{A}_{\mathcal{S}} = (Q_c \cup \{E\}, U, E, \{op_n\}_{n\in N})$, where for each $n \in N$, $p \in Q_c \cup \{E\}$, and $a \in I_n$, we have:

$$
op_n(p, a) = \begin{cases} (q, b) & \text{if the path } \delta_{n(p,a)} \text{ is exception-free and ends in the} \\ & \text{complete state } q \text{ with return label } ret(b). \\ (E, e) & \text{otherwise.} \end{cases}
$$

Thus a path $\delta_{n(p,a)}$ in an ADT-TS contributes an element $((p, a), (q, b))$ for the operation relation op_n in the induced ADT. For example, Fig. 3.3 shows the operation relations in the ADT induced by the ADT-TS of Fig. 3.2.

3.2 Refinement between ADT transition systems

In this section we describe the refinement between ADT transition systems.

Definition 3.2. *Let $\mathcal{S}$ and $\mathcal{S}'$ be two ADT transition systems of type $\mathcal{N}$. Then $\mathcal{S}'$ refines $\mathcal{S}$, iff the induced ADTs $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ are such that $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_{\mathcal{S}}$.*

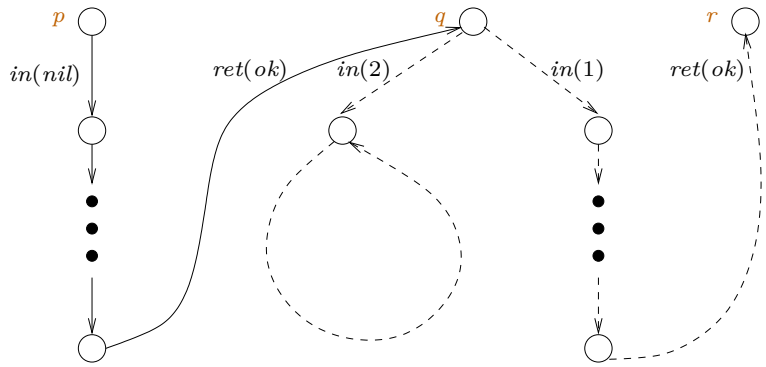


Figure 3.4: A part of an ADT-TS which is assumed to model an implementation of an ADT of type $QType_{\mathbb{Z}}$. Solid edges represent $\delta_{\texttt{init}}$ and dashed edges represent $\delta_{\texttt{enq}}$.

For example, let $\mathcal{S}$ and $\mathcal{S}'$ be the ADT transition systems of Fig. 3.4 and Fig. 3.2 respectively. Then $\mathcal{S}'$ refines $\mathcal{S}$, since $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_{\mathcal{S}}$. But $\mathcal{S}$ cannot refine $\mathcal{S}'$, since $\mathcal{A}_{\mathcal{S}}$ cannot refine $\mathcal{A}_{\mathcal{S}'}$ as it cannot simulate the exception-free initial sequence of operations $(init, nil, ok)(enq, 2, ok)$, which $\mathcal{A}_{\mathcal{S}'}$ allows.

It follows from the above definition that our notion of refinement for ADT transition systems preserves exactly the same set of properties preserved by our refinement notion for ADTs (see Sec. 2.5).

3.3 Equivalent refinement condition

Let $\mathcal{S}$ and $\mathcal{S}'$ be two ADT transition systems of type $\mathcal{N}$. Then we use the notations: "$\delta^{\mathcal{S}}_{\mathcal{N}}$" and "$\delta^{\mathcal{S}'}_{\mathcal{N}}$" to denote the $\delta_{\mathcal{N}}$ paths in $\mathcal{S}$ and $\mathcal{S}'$ respectively. We now lift the refinement condition (RC) presented in Sec. 2.6, to ADT transition systems. Let $\mathcal{S} = (Q_c, Q_l, \Sigma, U, \{\delta_n\}_{n\in N})$ and $\mathcal{S}' = (Q'_c, Q'_l, \Sigma', U', \{\delta'_n\}_{n\in N})$ be two ADT transition systems of type $\mathcal{N}$. We say $\mathcal{S}$ and $\mathcal{S}'$ satisfy the condition (RC-TS), if there exists a relation $\rho \subseteq Q'_c \times Q_c$ such that:

(init) Let $a \in I_{init}$, $p \in Q_c$ and $p' \in Q'_c$. Suppose the path $\delta^{\mathcal{S}}_{init(p,a)}$ is exception-free and it ends in the complete state q_a with the return label $ret(b)$. Then the path $\delta^{\mathcal{S}'}_{init(p',a)}$ must be exception-free and it must end in a complete state q'_a with return label $ret(b)$ such that $(q'_a, q_a) \in \rho$.

(sim) For each $n \in N$, $a \in I_n$, $b \in O_n$, $p, q \in Q_c$, and $p' \in Q'_c$, with $(p', p) \in \rho$. Suppose the path $\delta^{\mathcal{S}}_{n(p,a)}$ is exception-free and ends in a state q with return label $ret(b)$. Then the path $\delta^{\mathcal{S}'}_{n(p',a)}$ must be exception-free and it must end in a complete state q' with return label $ret(b)$ such that $(q', q) \in \rho$.

Fig. 3.5 illustrates the equivalent condition (RC-TS) for refinement between ADT transition systems.

Theorem 3.1. *Let $\mathcal{S}$ and $\mathcal{S}'$ be two ADT transition systems of type $\mathcal{N}$. Then $\mathcal{S}'$ refines $\mathcal{S}$ iff $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS).*

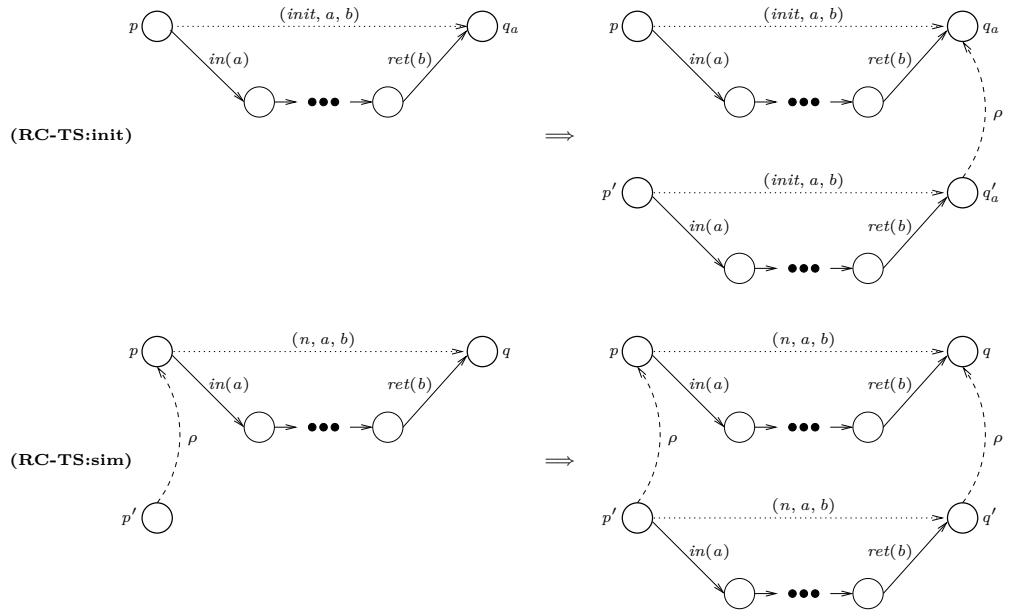


Figure 3.5: Illustrating the equivalent condition (RC-TS) for refinement. Solid arrows represent transitions in the ADT-TS, dashed arrows represent the abstraction relation ρ and dotted arrows represent operation relations in the ADT induced by the $\delta_{\mathcal{N}}$ paths of the ADT-TS.

Proof. Recall that $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ represent the ADTs induced by the ADT transition systems $\mathcal{S}$ and $\mathcal{S}'$ respectively. We fix $\mathcal{S}$, $\mathcal{S}'$, $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ as above for the rest of this section. First we prove the following claim.

Claim 3.1. *The ADT transition systems $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS) with respect to an abstraction relation ρ, iff $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC) of Sec. 2.6, with respect to the same abstraction relation ρ.*

Proof. Let ρ be an abstraction relation similar to the one used in the proof of Claim 2.1, except that here the ADTs are obtained from the ADT transition systems. We know that a state p' of $\mathcal{A}_{\mathcal{S}'}$ is related to a state p of $\mathcal{A}_{\mathcal{S}}$, iff there exists an exception-free initial sequence of operations w such that $U \xrightarrow{w} p$ in $\mathcal{A}_{\mathcal{S}}$ and $U' \xrightarrow{w} p'$ in $\mathcal{A}_{\mathcal{S}'}$.

($\Rightarrow$) Suppose $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS:init) with respect to ρ. Let a be an element in I_{init} and the path $\delta^{\mathcal{S}}_{init(p,a)}$ is exception-free and ends in a complete state q in $\mathcal{S}$ with return label $ret(b)$. Then it follows from the condition (RC-TS:init) that: (i) $\delta^{\mathcal{S}'}_{init(p',a)}$ is an exception-free path ending in a complete state q' in $\mathcal{S}'$ with return label $ret(b)$ and (ii) $(q', q) \in \rho$. Now by the definitions of $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$, we have $p \xrightarrow{(init,a,b)} q$ in $\mathcal{A}_{\mathcal{S}}$ and $p' \xrightarrow{(init,a,b)} q'$ in $\mathcal{A}_{\mathcal{S}'}$. Therefore $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC:init) with respect to the abstraction relation ρ.

Suppose $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS:sim) with respect to ρ. Let n be an operation in N, let a be an input value in I_n, the path $\delta^{\mathcal{S}}_{n(p,a)}$ is exception-free and ends in a complete state q in $\mathcal{S}$ with return label $ret(b)$, and let p' be a complete state of $\mathcal{S}'$ such that $(p', p) \in \rho$. Then it follows from the condition (RC-TS:sim) that: (i) the path $\delta^{\mathcal{S}'}_{n(p',a)}$ is exception-free and ends in a complete state q' in $\mathcal{S}'$ with return label $ret(b)$ and (ii) $(q', q) \in \rho$. Now by the definitions of $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$, we have $p \overset{(n,a,b)}{\rightarrow} q$ in $\mathcal{A}_{\mathcal{S}}$ and $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}_{\mathcal{S}'}$. Therefore $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC:sim) with respect to the abstraction relation ρ.

($\Leftarrow$) Suppose $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC:init) with respect to ρ. Let a be an element in I_{init}, $p \overset{(init,a,b)}{\rightarrow} q$ in $\mathcal{A}_{\mathcal{S}}$ and $p' \overset{(init,a,b')}{\rightarrow} q'$ in $\mathcal{A}_{\mathcal{S}'}$. Then it follows from the condition (RC:init) that $b = b'$ and $(q', q) \in \rho$. Now by the definitions of $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ it follows that: (i) the path $\delta^{\mathcal{S}}_{init(p,a)}$ is exception-free and ends in the complete state q in $\mathcal{S}$ with return label $ret(b)$ and (ii) the path $\delta^{\mathcal{S}'}_{init(p',a)}$ is exception-free and ends in the complete state q' in $\mathcal{S}'$ with return label $ret(b)$. Therefore $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS:init) with respect to ρ, since we have $(q', q) \in \rho$.

Suppose $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC:sim) with respect to ρ. Let n be an operation in N, let a be an input value in I_n, $p \overset{(n,a,b)}{\rightarrow} q$ in $\mathcal{A}_{\mathcal{S}}$ with $b \neq e$, and let p' be a state of $\mathcal{A}_{\mathcal{S}'}$ such that $(p', p) \in \rho$. Then it follows from the condition (RC:sim) that $p' \overset{(n,a,b)}{\rightarrow} q'$ in $\mathcal{A}_{\mathcal{S}'}$ with $(q', q) \in \rho$. Now by the definitions of $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ it follows that: (i) the path $\delta^{\mathcal{S}}_{n(p,a)}$ is exception-free and ends in the complete state q in $\mathcal{S}$ with return label $ret(b)$ and (ii) the path $\delta^{\mathcal{S}'}_{n(p',a)}$ is exception-free and ends in the complete state q' in $\mathcal{S}'$ with return label $ret(b)$. Therefore $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS:sim) with respect to ρ, since we have $(q', q) \in \rho$.

□

Now we prove Theorem 3.1.

($\Rightarrow$) Suppose $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS). Then from Claim 3.1 we know that $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC). Now it follows from Theorem 2.3 that $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_{\mathcal{S}}$. Then we can conclude from Def. 3.2 that $\mathcal{S}'$ refines $\mathcal{S}$.

($\Leftarrow$) Suppose $\mathcal{S}'$ refines $\mathcal{S}$. Then by Def. 3.2 we have $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_{\mathcal{S}}$. Now it follows from Theorem 2.3 that $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}'}$ satisfy condition (RC). Then we can conclude from Claim 3.1 that $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS).

□

3.4 Client ADT transition systems

A client ADT-TS is similar to the client transition system defined in Sec. 2.3, except that it is meant to implement an ADT. Thus a client ADT-TS may contain calls to a sub-ADT. For example, consider the method `resched` of Fig. 1.8. This method is a part of the ADT implementation `c-sched`, which implements the operation *resched* in the scheduler ADT of Fig. 1.7. The transition system modeling the program `c-sched` is called a "$QType_{\mathbb{Z}}$-Client ADT-TS" since it is a client transition system of an ADT of type $QType_{\mathbb{Z}}$ and also implements the functionality of another ADT. After plugging-in a particular sub-ADT, a client ADT-TS becomes an ADT-TS.

Definition 3.3 ($\mathcal{M}$-client ADT-TS)**.** *Let $\mathcal{M} = (M, (I_m)_{m\in M}, (O_m)_{m\in M})$ and $\mathcal{N} = (N, (I_n)_{n\in N}, (O_n)_{n\in N})$ be ADT types. Then an $\mathcal{M}$-client ADT-TS of type $\mathcal{N}$ can be defined as a 6-tuple: $\mathcal{U} = (Q_c, Q_l, \Sigma, U, E, \{\delta_n\}_{n\in N})$ where Q_c, Q_l, and U are as in an ADT-TS. Let $\Sigma = \Sigma_l \cup \Gamma^i_{\mathcal{N}} \cup \Gamma^o_{\mathcal{N}} \cup \Sigma_{\mathcal{M}}$, where Σ_l, $\Gamma^i_{\mathcal{N}}$ and $\Gamma^o_{\mathcal{N}}$ are as in an ADT-TS and $\Sigma_{\mathcal{M}}$ is the set of operation call labels from the ADT type $\mathcal{M}$. The state $E \in Q_l$ is an exceptional state that arises when a call to a sub-ADT returns an exceptional value. Then, for each operation n in N, δ_n is a transition relation of the form $\delta_n \subseteq Q \times \Sigma \times Q$ satisfying similar constraints as in an ADT-TS, except that in addition we require that:*

- *E is a dead state (i.e. δ_n has no transition of the form $(E, -, -)$).*
- *δ_n is "closed" with respect to a given M-operation and input value (thus if $l \overset{(m,a,b)}{\rightarrow} l'$ in δ_n and $l \overset{(m',a',b')}{\rightarrow} l''$ in δ_n, then $m = m'$ and $a = a'$. This is meant to model actions like `enq(cur)` in the program `c-sched` of Fig. 1.8, where the M-operation (`enq`) and the argument (the value of `cur`) are fixed.*
- *The δ_{init} transition relation is assumed to initialize the sub-ADT before going on to make other calls to it.*

For example, Fig. 3.6 shows a part of a $QType_{\mathbb{B}}$-client ADT-TS modeling the method `reshed` in the program of Fig. 1.8, which implements the *resched* operation in the scheduler ADT of Fig. 1.7.

Now we illustrate the structural difference between an ADT-TS and a $\mathcal{M}$-client ADT-TS. Given an $\mathcal{M}$-client ADT-TS $\mathcal{U}$, the sub-transition system of $\mathcal{U}$ induced by δ_n for an operation n in N and for an input a in I_n from a state q in Q_c, denoted "$\delta_{n(q,a)}$" is a set paths as opposed to the unique path in case of an ADT-TS. The sub-transition system $\delta_{n(q,a)}$ branches out when it calls an M-operation say m with an input say a', since an $\mathcal{M}$-client transition system needs to be complete with respect to an M-operation m in M and a' in I_m. For instance, consider the set of paths $\delta_{resched(q,1)}$ of the $QType_{\mathbb{B}}$-client ADT-TS of Fig. 3.6. Some of the paths in $\delta_{resched(q,1)}$ enter the exception state E and others end in one of the complete states: q_1, q_2 or q_3.

Suppose we want to prove that `c-sched`[`c-queue`] refines the abstract ADT `z-queue` of Fig. 1.7. We would like to reason about this in a step-by-step manner to reduce the complexity involved in doing this in a single

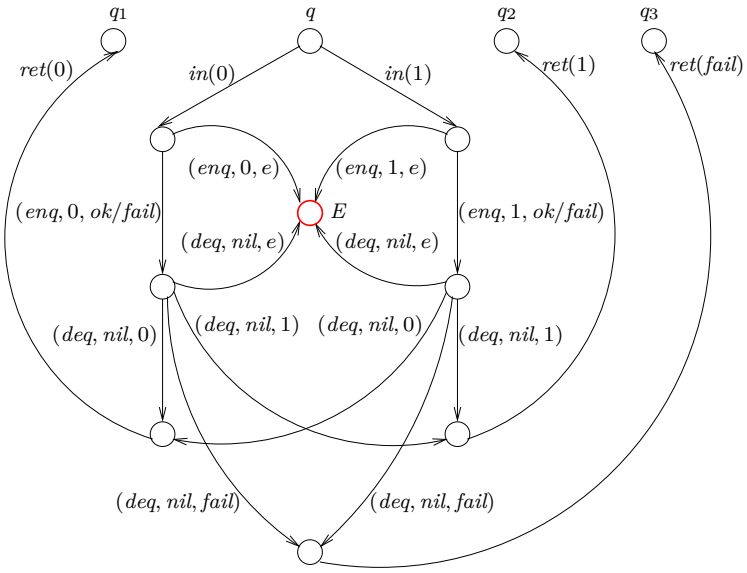


Figure 3.6: A part of a $QType_{\mathbb{B}}$-client ADT-TS modeling the method `reshed` (of Fig. 1.8) in a program implementing the scheduler ADT. We assume that *fail* is a value in the output type $O_{resched}$.

step. As a first step, we could abstract the `c-queue` component and replace it by the simpler and more-abstract `g-queue` (Fig. 1.6) component, and argue that `c-sched`[`g-queue`] refines `z-sched`. As a second step we would need to argue that `c-sched`[`c-queue`] refines `c-sched`[`g-queue`]. We prove in Theorem 3.2 below that our refinement notion is "substitutive" for ADT implementations, which ensures that `c-sched`[`c-queue`] refines `c-sched`[`g-queue`] whenever `c-queue` refines `g-queue`. Thus we could simply prove that `c-queue` refines `g-queue` and then infer using Theorem 3.2 that `c-sched`[`c-queue`] refines `c-sched`[`g-queue`].

We now consider the transition system obtained by plugging-in an ADT of type $\mathcal{M}$ in an $\mathcal{M}$-client ADT-TS. Let $\mathcal{U}$ be an $\mathcal{M}$-client ADT-TS and $\mathcal{A}$ be an ADT of type $\mathcal{M}$. Then we show that the transition system obtained by plugging-in $\mathcal{A}$ in $\mathcal{U}$ is an ADT-TS. We fix $\mathcal{M} = (M, (I_m)_{m\in M}, (O_m)_{m\in M})$ and $\mathcal{N} = (N, (I_n)_{n\in N}, (O_n)_{n\in N})$ for the rest of this section.

Definition 3.4. *Let $\mathcal{U} = (Q_c, Q_l, \Sigma, U, E, \{\delta_n\}_{n\in N})$ be an $\mathcal{M}$-client ADT-TS of type $\mathcal{N}$ and let $\mathcal{A} = (Q', U', E', \{op_m\}_{m\in M})$ be an ADT of type $\mathcal{M}$. Then we can define the ADT-TS obtained by using $\mathcal{A}$ in $\mathcal{U}$, denoted "$\mathcal{U}[\mathcal{A}]$", to be the ADT-TS of the form:*

$$\mathcal{U}[\mathcal{A}] = (Q_c^{\mathcal{U}[\mathcal{A}]}, Q_l^{\mathcal{U}[\mathcal{A}]}, \Sigma, (U, U'), \{\delta_n^{\mathcal{U}[\mathcal{A}]}\}_{n\in N})$$

where $Q_c^{\mathcal{U}[\mathcal{A}]} = Q_c \times Q'$, $Q_l^{\mathcal{U}[\mathcal{A}]} = Q_l \times Q'$ and $\{\delta_n^{\mathcal{U}[\mathcal{A}]}\}_{n\in N}$ is the transition relation such that for each n in N, for each a in I_n and for each p in $Q_c^{\mathcal{U}[\mathcal{A}]}$:

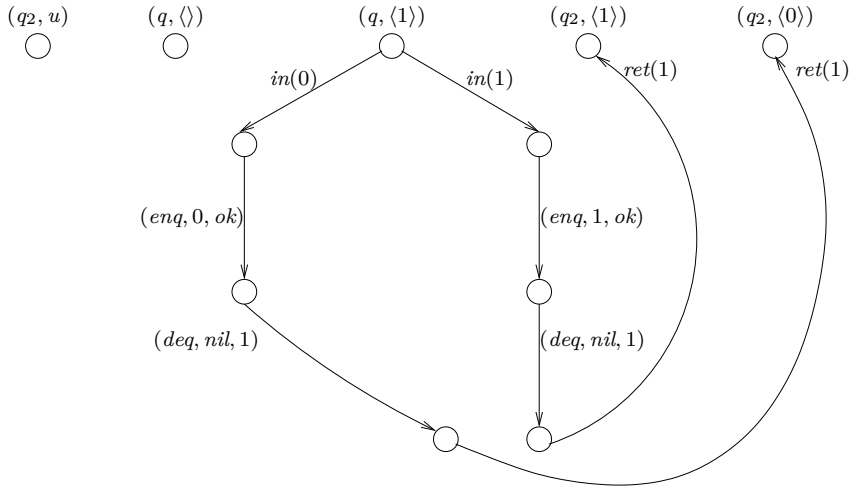


Figure 3.7: A part of the ADT-TS obtained by plugging $QADT_k$ of Fig. 2.3 in the $QType_{\mathbb{B}}$-client ADT-TS of Fig. 3.6.

$\delta^{\mathcal{U}[\mathcal{A}]}_{n(p,a)} \subseteq \delta_{n(p,a)}$ *and the transitions in* $\delta^{\mathcal{U}[\mathcal{A}]}_{n(p,a)}$ *are such that:*

$$\begin{array}{lll} (p, r') \xrightarrow{l} (q, r') & \textit{if} & l \in \Sigma_l \textit{ and } p \xrightarrow{l} q \textit{ in } \mathcal{U}. \\ (p, r') \xrightarrow{(m,a,b)} (q, s') & \textit{if} & (m, a, b) \in \Sigma_{\mathcal{M}} \textit{ and } p \xrightarrow{(m,a,b)} q \textit{ in } \mathcal{U} \textit{ and} \\ & & op_m(r', a) = (s', b). \end{array}$$

For example, let $\mathcal{U}$ be the $QType_{\mathbb{B}}$-client ADT-TS of Fig. 3.6 and let $\mathcal{A}$ be the ADT $QADT_k$ of Fig. 2.3, which is of type $QType_{\mathbb{B}}$. Now the transition system obtained by plugging-in $QADT_k$ in $\mathcal{U}$ (that is $\mathcal{U}[QADT_k]$) is shown in Fig. 3.7. We assume here that the value of the parameter k in $QADT_k$ is greater than 1. This figure shows only some of the complete states in $\mathcal{U}[\mathcal{A}]$.

An $\mathcal{M}$-client ADT-TS $\mathcal{U}$ becomes closed (see Sec. 2.1) when we plug-in an ADT $\mathcal{A}$ of type $\mathcal{M}$ in $\mathcal{U}$ to get $\mathcal{U}[\mathcal{A}]$. Therefore there is no possibility of branching in the sub-transition system $\delta_{n(q,a)}$ in $\mathcal{U}[\mathcal{A}]$ and hence $\delta_{n(q,a)}$ in $\mathcal{U}[\mathcal{A}]$ becomes a path for each n in N, for each a in I_n and for each q in Q_c. Hence $\mathcal{U}[\mathcal{A}]$ is an ADT-TS.

3.5 Compositionality of refinement

In this section we show that our notion of refinement for ADT implementations is compositional.

Theorem 3.2. *Let $\mathcal{U}$ be an $\mathcal{M}$-client ADT-TS of type $\mathcal{N}$, and $\mathcal{B}$ and C be ADTs of type $\mathcal{M}$ such that $\mathcal{C} \preceq \mathcal{B}$. Then $\mathcal{U}[\mathcal{C}]$ refines $\mathcal{U}[\mathcal{B}]$.*

Proof. It is sufficient to define an abstraction relation ρ' from the complete states of $\mathcal{U}[\mathcal{C}]$ to the complete states of $\mathcal{U}[\mathcal{B}]$ satisfying condition (RC-TS). To do this we make use of the necessary and sufficient condition for refinement, (RC) of Theorem 2.3. Since $\mathcal{C}$ refines $\mathcal{B}$, by Theorem 2.3 there must exist a relation ρ from the states of $\mathcal{C}$ to the states of $\mathcal{B}$ satisfying conditions (init)

and (sim) of (RC). We now define a relation ρ' from the complete states of $\mathcal{U}[\mathcal{C}]$ to the complete states of $\mathcal{U}[\mathcal{B}]$ given by $((p, q'), (r, q)) \in \rho'$, iff $p = r$ and $(q', q) \in \rho$.

Now we prove that $\mathcal{U}[\mathcal{C}]$ and $\mathcal{U}[\mathcal{B}]$ satisfy condition (RC-TS) with respect to the abstraction function ρ' defined above. We know that $\mathcal{U}[\mathcal{B}]$ and $\mathcal{U}[\mathcal{C}]$ are ADT transition systems and hence for each operation n in N, for each input a in I_n, for each state u in $\mathcal{U}[\mathcal{B}]$ and for each state u' in $\mathcal{U}[\mathcal{C}]$: $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ and $\delta^{\mathcal{U}[\mathcal{C}]}_{n(u',a)}$ are paths in $\mathcal{U}[\mathcal{B}]$ and $\mathcal{U}[\mathcal{C}]$ respectively.

(init) Suppose the path $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ in $\mathcal{U}[\mathcal{B}]$ is exception-free. Then each operation call to $\mathcal{B}$ in $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ returns a non-exception value, since $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ is exception-free. Therefore each operation call to $\mathcal{C}$ in $\delta^{\mathcal{U}[\mathcal{C}]}_{init(u',a)}$ will produce the same output as produced in the corresponding call to $\mathcal{B}$ in $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ and the resulting state in $\delta^{\mathcal{U}[\mathcal{C}]}_{init(u',a)}$ will be related to the corresponding resulting state in $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ by ρ', since $\mathcal{C}$ refines $\mathcal{B}$. This implies that the paths $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ and $\delta^{\mathcal{U}[\mathcal{C}]}_{init(u',a)}$ terminate in states related by ρ' and produce the same output. Hence $\mathcal{U}[\mathcal{B}]$ and $\mathcal{U}[\mathcal{C}]$ satisfy (RC-TS:init).

(sim) Let u' and u be states in $\mathcal{U}[\mathcal{C}]$ and $\mathcal{U}[\mathcal{B}]$ respectively with $(u', u) \in \rho'$ and the path $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ is exception-free in $\mathcal{U}[\mathcal{B}]$. Then each operation call to the ADT $\mathcal{B}$ in $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ returns a non-exception value, since $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ is exception-free. Therefore each operation call to the ADT $\mathcal{C}$ in the path $\delta^{\mathcal{U}[\mathcal{C}]}_{n(u',a)}$ will produce the same output as produced in the corresponding call to $\mathcal{B}$ in $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ and the resulting state in $\delta^{\mathcal{U}[\mathcal{C}]}_{n(u',a)}$ will be related to the corresponding resulting state in $\delta^{\mathcal{U}[\mathcal{B}]}_{n(u,a)}$ by ρ', since $\mathcal{C}$ refines $\mathcal{B}$ and $(u', u) \in \rho'$. This implies that the paths $\delta^{\mathcal{U}[\mathcal{B}]}_{init(u,a)}$ and $\delta^{\mathcal{U}[\mathcal{C}]}_{init(u',a)}$ terminate in states related by ρ' and produce the same output value. Hence $\mathcal{U}[\mathcal{B}]$ and $\mathcal{U}[\mathcal{C}]$ satisfy (RC-TS:sim).

□

We can extend the definition of client transition systems to allow them to have multiple sub-ADTs. Thus an $(\mathcal{M}_1, \ldots, \mathcal{M}_n)$-client transition system makes calls to ADTs of type $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Theorem 3.2 implies that the congruence property holds for client ADT transition systems with multiple sub-ADTs as well. For example, if $\mathcal{U}$ is an $(\mathcal{M}_1, \mathcal{M}_2)$-client ADT-TS, $\mathcal{A}_1$ and $\mathcal{B}_1$ are ADTs of the same type $\mathcal{M}_1$, and $\mathcal{A}_2$ and $\mathcal{B}_2$ are ADTs of the same type $\mathcal{M}_2$, such that $\mathcal{B}_1 \preceq A_1$ and $\mathcal{B}_2 \preceq \mathcal{A}_2$; then $\mathcal{U}[\mathcal{B}_1, \mathcal{B}_2] \preceq \mathcal{U}[\mathcal{A}_1, \mathcal{A}_2]$. This

follows since:

$$\begin{array}{rcl}
\mathcal{U}[\mathcal{B}_1, \mathcal{B}_2] & = & (\mathcal{U}[\mathcal{B}_1])[\mathcal{B}_2] \\
& \preceq & (\mathcal{U}[\mathcal{B}_1])[\mathcal{A}_2] \text{ (by Theorem 3.2 since } \mathcal{B}_2 \preceq A_2) \\
& = & \mathcal{U}[\mathcal{B}_1, \mathcal{A}_2] \\
& = & (\mathcal{U}[\mathcal{A}_2])[\mathcal{B}_1] \\
& \preceq & (\mathcal{U}[\mathcal{A}_2])[\mathcal{A}_1] \text{ (by Theorem 3.2 since } \mathcal{B}_1 \preceq A_1) \\
& = & \mathcal{U}[\mathcal{A}_1, \mathcal{A}_2].
\end{array}$$

3.6 Related work

Program verification tools like VCC [15], RESOLVE [25] and Dafny [35] allow verification of imperative language implementations of ADT-like systems with respect to specifications in the form of function contracts. But these systems do not talk about a formal model of imperative language programs like the one we propose. Also there is no explicit notion of refinement used in these tools and the technique is not compositional.

A technique is presented in [39] with the ideas from separation logic [43] to ensure that a refined system preserves separation-relations (between the client and library) satisfied by the abstract system. A sound but incomplete theory is presented in [23], where separation logic is used to establish non-interference between a client and a module (or ADT). Non-interference is a desirable property to ensure soundness of a compositional notion of refinement.

Chapter 4

ADTs in Different Modeling Languages

Abstract Data Types can be specified in different modeling languages. They could be specified in a declarative style in a language like Z or in an imperative style in implementation languages like C. In this chapter we describe the mathematical ADTs induced by these different models.

4.1 ADTs in the Z language

In this section we introduce the Z modeling language and illustrate how ADTs are specified. We then describe the mathematical ADTs induced by these specifications.

4.1.1 About the Z language

The Z specification language was originally proposed by Abrial et al [6] based on the notation used in data semantics [2]. It is a model-oriented specification language based on set theory and mathematical logic [46]. The logic used is a first-order predicate calculus.

We choose to use the Z specification language to specify the abstract versions of ADTs as it gives us a concise yet readable, and mathematically precise specification of ADTs, with tool support for simulation and validation of Z models [36, 44]. The rich set of mathematical objects and operators in the Z language supports easy specification of ADTs.

The *schema* language is the important aspect of Z, which makes it suitable for ADT specification. A user can specify data and predicates governing the properties of data together in a single entity called a *schema*. A schema has a name and its specification comprises the following: (i) a declaration part, where state components can be declared and (ii) a constraint part, where a set of predicates can be included to constrain the values of the fields in the schema. The schema language of Z can be used to model the set of states as well as the set of operations of an ADT.

$$
\begin{aligned}
SchedType &= \{init, create, resched, I_{init}, O_{init}, I_{create}, O_{create}, I_{resched}, O_{resched}\} \\
\text{where: } I_{init} &= \mathbb{N} \\
O_{init} &= \{ok, e\} \\
I_{create} &= \mathbb{B} \\
O_{create} &= \{ok, fail, e\} \\
I_{resched} &= \mathbb{B} \\
O_{resched} &= \mathbb{B} \cup \{fail, e\}
\end{aligned}
$$

Figure 4.1: The ADT type *SchedType*.

4.1.2 Specifying ADTs in Z

In this subsection we discuss the schema language and some of the mathematical objects and operators from the Z language, which we use for specifying an ADT of type *SchedType*, which we define below. We show how we specify the states and operations of an ADT using the schema language of Z.

We define an example ADT type called *SchedType* to illustrate this. Fig. 4.1 shows this ADT type. The scheduler ADT of Fig. 1.7, is of this type except that (i) here the *init* operation takes an argument, which represents the maximum number of tasks in the ready queue and (ii) we do not consider the operation *delay* in this ADT type.

The type of a task is considered as $\mathbb{B}$, the set of bit values $\{0, 1\}$. The *init* operation takes a natural number as an argument, which represents the bound on the number of tasks in the ready queue. The *init* operation is expected to return the dummy value *ok*, when the operation is successful. The operation *create* takes the task to be inserted to the ready list as an argument. The *create* operation is expected to return the value *ok*, when it succeeds in enqueueing the given task into the ready list, and it is expected to return the value *fail*, when it cannot insert the given task to the ready list, due to the unavailability of space in the ready list. The operation *resched* takes the currently running task as an argument. The *resched* operation is expected to return a task as the next task to run, when it succeeds in: (i) enqueueing the given task into the ready list and (ii) extracting a task from the ready list. It can also return the value *fail*, when it cannot successfully complete its operation, due to the unavailability of space in the ready list to insert the given task.

Specifying the states of an ADT in Z

Fig. 4.2 shows a Z schema called *Scheduler* representing the states of an ADT of type *SchedType*. The ready queue is declared as a finite sequence of natural numbers called *ready*. The variable *bound*, which is declared as a natural

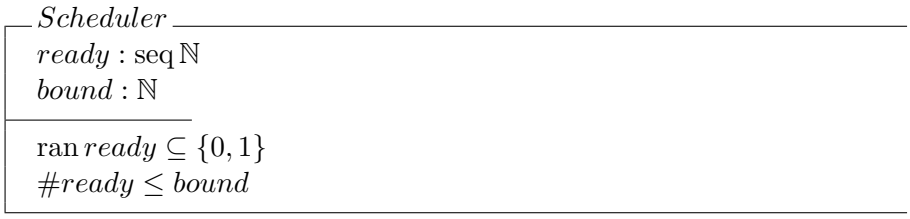


Figure 4.2: A Z schema specifying the states of an ADT of type *SchedType*.

number represents the bound on the length of the ready queue. For a sequence s in Z, the expression "ran s" denotes the set of elements present in s and the expression "# s" denotes the length (number of elements) of s.

We use the term *data-schema* to denote a Z schema describing the states of an ADT. For example, the schema of Fig. 4.2 is a data-schema which describes the states of an ADT of type *SchedType*. The set of states of the ADT described by this schema is: $\{(ready, bound) \mid ready \in \mathbb{N}^* \text{ and } bound \in \mathbb{N}\}$, where "$\mathbb{N}^*$" denotes the set of all finite strings of natural numbers.

The term *state-invariant* is used to denote the formula constraining the valuations for the variables in a data-schema. The state-invariant of a data-schema is the conjunction of the predicates in its constraint part. For example, the state-invariant of the data-schema of Fig. 4.2 is:

$$(\text{ran}\, ready \subseteq \{0, 1\}) \wedge (\#ready \leq bound)$$

where the first predicate restricts the sequence $ready$ to a finite sequence of *bit values* rather than natural numbers and the second predicate bounds the length of the sequence $ready$.

An ADT state is called *legal*, if it satisfies the state-invariant of the data-schema. For example, the set of legal states of the ADT described by the data-schema of Fig. 4.2 is:

$$\{(ready, bound) \mid ready \in \mathbb{B}^* \text{ and } bound \in \mathbb{N} \text{ and } \#ready \leq bound\}$$

.

Specifying the operations of an ADT in Z

Now we focus on specifying the operations of an ADT, using the schema language of Z.

We use the term *operation-schema* to denote a Z schema describing an ADT operation. An operation-schema needs to describe a state change in the ADT with possible input and output. Therefore the declaration part of an operation-schema should declare: (i) a before-state, (ii) input variables, (iii)

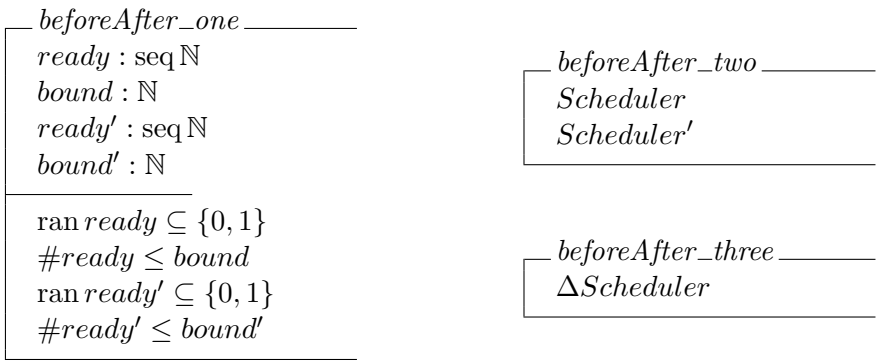


Figure 4.3: Equivalent schemas in Z declaring a before-state and an after-state for the data-schema *Scheduler*.

an after-state and (iv) output variables.

The schemas of Fig. 4.3, shows three different ways for declaring a before-state and an after-state, in terms of the data-schema of Fig. 4.2. By convention, primed variables denote the components in the after-state and unprimed variables denote the components in the before-state of an operation-schema. For example, $ready'$ denotes the ready sequence in an after-state and the predicate "$\#ready' \leq bound'$" denotes a constraint on an after-state. On the other hand, $ready$ denotes the ready sequence in a before-state and the predicate "$\#ready \leq bound$" denotes a constraint on a before-state.

In Fig. 4.3, the schemas: *beforeAfter_two* and *beforeAfter_three* are two other ways in Z, to represent the schema *beforeAfter_one*. Thus, if S is the name of a data-schema, then the symbol S in the declaration part of an operation-schema expands to the definition of the schema S representing the before-state and the symbol S' in the declaration part of an operation-schema expands to the definition of the schema S with primed variables representing the after-state. The expression "ΔS" in the Z language denotes the sequence of declarations: S; S'.

The convention followed in Z for declaring input and output variables is that the name of an input variable ends with a "?" and the name of an output variable ends with an "!". For example, $maxLength?$ is the input variable in the operation-schema of Fig. 4.4, and $status!$ is the output variable in this operation-schema.

Fig. 4.4 shows an operation-schema for the *init* operation in an ADT of type *SchedType*. Recall that the *init* operation in an ADT is independent of the state from which it is invoked and hence there is no need to declare a before-state. This is in-line with the specification of an *init* schema in the Z notation. The type *STATUS* is assumed to be a free data-type with the set of possible values $\{ok, fail\}$, and could be defined as "$STATUS ::= ok \mid fail$" in Z. The predicates in this operation-schema require that, in the after-state, (i)

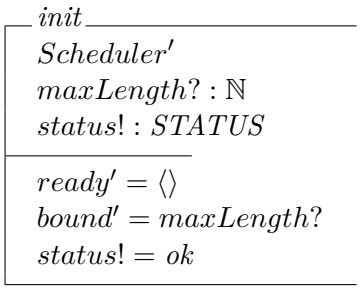


Figure 4.4: An operation-schema in Z for the *init* operation in an ADT of type *SchedType*.

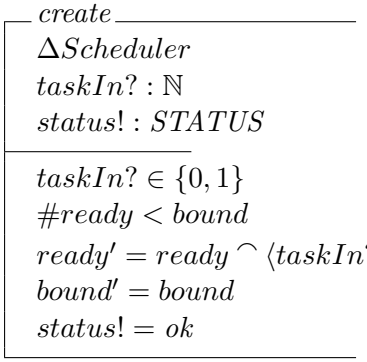


Figure 4.5: An operation-schema in Z for the *create* operation in an ADT of type *SchedType*.

the sequence *ready* is empty, (ii) the variable *bound* has the same value as the input variable *maxLength*? and (iii) the output variable *status*! has the value *ok*.

Fig. 4.5 shows an operation-schema for the *create* operation in an ADT of type *SchedType*. The constraint part includes two predicates to constrain a before-state or an input, which are mandatory to have a consistent update. For instance, without the first predicate, the update may result in an after-state failing to satisfy the first predicate in the data-schema of Fig. 4.2. This operation-schema updates a before-state by appending the input task to the ready sequence in the before-state and defines the output as *ok*.

Fig. 4.6 shows an operation-schema for the *resched* operation of an ADT of type *SchedType*. For a non-empty sequence s in Z, the expression "*head* s" denotes the head (first element) of s and the expression "*tail* s" denotes a sequence which is obtained from s by removing its head.

Let S be an operation-schema. Then the conjunction of predicates in S is called the *Before-After Predicate* of S, written BAP_S. For example, Fig. 4.7 shows the BAP of the operation-schema of Fig. 4.6. Thus the BAP includes the predicates representing declarations. An operation-schema S is said to be *consistent* if there exist a before-state p, an input a, an after-state q and an output b such that BAP_S evaluates to *true*. Each of the above operation schemas is consistent. Fig. 4.8 shows a valuation for the free variables in the BAP of the operation-schema of Fig. 4.6, witnessing its consistency.

4.1.3 Viewing Z models as ADTs

We describe in this subsection how we view a Z model as an ADT as defined in Chap. 2.

Informally, we view the data-schema of a Z model as a representation of the states of the ADT and the BAP of an operation-schema as a realization

resched

$\Delta Scheduler$
$taskIn? : \mathbb{N}$
$taskOut! : \mathbb{N}$

$taskIn? \in \{0, 1\}$
$taskOut! = head\,(ready \frown \langle taskIn? \rangle)$
$ready' = tail\,(ready \frown \langle taskIn? \rangle)$
$bound' = bound$

Figure 4.6: An operation-schema in Z for the *resched* operation of an ADT of type *SchedType*.

$$
\begin{aligned}
BAP_{resched} \;=\; & (ready \in \operatorname{seq} \mathbb{N}) \wedge (bound \in \mathbb{N}) \wedge (ready' \in \operatorname{seq} \mathbb{N}) \wedge (bound' \in \mathbb{N}) \wedge \\
& (taskIn? \in \mathbb{N}) \wedge (\operatorname{ran} ready \subseteq \{0, 1\}) \wedge (\#ready \leq bound) \wedge \\
& (taskIn? \in \{0, 1\}) \wedge (taskOut! = head(ready)) \wedge \\
& (ready' = tail(ready) \frown \langle taskIn? \rangle) \wedge (bound' = bound) \wedge \\
& (\operatorname{ran} ready' \subseteq \{0, 1\}) \wedge (\#ready' \leq bound')
\end{aligned}
$$

Figure 4.7: The Before-After Predicate (BAP) of the schema of Fig. 4.6.

$$
\begin{aligned}
ready &= \langle 0, 1, 1 \rangle & \qquad bound &= 5 \\
taskIn? &= 1 & \qquad ready' &= \langle 1, 1, 1 \rangle \\
bound' &= 5 & \qquad taskOut! &= 0
\end{aligned}
$$

Figure 4.8: A consistent valuation for the free variables in the BAP of the schema of Fig. 4.6.

of an operation in the ADT type.

More formally, a model $\mathcal{M}$ of an ADT in the Z specification language essentially comprises the following:

- A data-schema $Data^{\mathcal{M}}$ with a finite set of variables $Var^{\mathcal{M}}$ and a state-invariant, which constrains the valuations for the variables. Each variable $v \in Var^{\mathcal{M}}$ has an associated data-type T_v, which is the set of possible values for v. A state is a valuation s to these variables with $s(v) \in T_v$ for each $v \in Var^{\mathcal{M}}$. A state is legal if it satisfies the state-invariant in $Data^{\mathcal{M}}$. The state-invariant in the data-schema $Data^{\mathcal{M}}$, is a first-order logic formula with free variables in $Var^{\mathcal{M}}$.

- A finite set $Op^{\mathcal{M}}$ of operation-schemas. Each operation-schema n in $Op^{\mathcal{M}}$ has (for simplicity) a single formal input parameter x_n of type $X_n^{\mathcal{M}}$, and a single output variable y_n of type $Y_n^{\mathcal{M}}$; and a before-after-predicate $BAP_n^{\mathcal{M}}$ with free-variables in $Var^{\mathcal{M}} \cup \{x_n, y_n\} \cup (Var^{\mathcal{M}})'$, where for a set of variables Var we use the convention that Var' denotes the set of variables $\{v' \mid v \in Var\}$. The set of operation-schemas $Op^{\mathcal{M}}$ includes an initialization-schema called *init*, whose BAP predicate is only on the input parameter, the output variable and primed variables (i.e it only constrains the after-state).

We say the Z model is *deterministic*, if for each operation-schema $n \in Op^{\mathcal{M}}$, before-state p and input value $a \in X_n^{\mathcal{M}}$, we have at most one after-state q and output value $b \in Y_n^{\mathcal{M}}$ satisfying $BAP_n^{\mathcal{M}}(p, a, q, b)$.

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

$$\mathcal{A}_{\mathcal{M}} = (Q', U, E, \{op_n\}_{n \in N}) \text{ of type } \mathcal{N} = (N, (I_n)_{n \in N}, (O_n)_{n \in N})$$

where:

- The set of operations N is $Op^{\mathcal{M}}$ with $I_n = X_n^{\mathcal{M}}$ and $O_n = Y_n^{\mathcal{M}} \cup \{e\}$,

- $Q' = Q \cup \{E\}$ where Q is the set of states of $\mathcal{M}$, E is a new exceptional state, and U is an arbitrary state in Q, and

- for each $n \in N$, we have:

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

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

For example, Fig. 4.9 shows the ADT induced by the Z specification discussed in this section.

$$
\begin{array}{lcl}
SchedADT & = & (Q, U, E, \{op_n\}_{n \in SchedType}), \text{ where:} \\
Q & = & (\mathbb{B}^* \times \mathbb{N}) \cup \{E\} \\
op_{init}(s,k) & = & \begin{cases} ((\epsilon, k), ok) & \text{if } s \neq E \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{create}(s,a) & = & \begin{cases} ((w \cdot a, k), ok) & \text{if } s \text{ is of the form } (w,k) \text{ and } |w| < k \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{resched}(s,a) & = & \begin{cases} ((w' \cdot a, k), b) & \text{if } s \text{ is of the form } (b \cdot w', k) \\ ((\epsilon, k), a) & \text{if } s \text{ is of the form } (\epsilon, k) \\ (E, e) & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 4.9: The ADT of type *SchedType*, induced by the Z model discussed in this section.

Refinement between Z models: Given two deterministic Z models $\mathcal{M}$ and $\mathcal{M}'$, we say $\mathcal{M}'$ *refines* $\mathcal{M}$, iff the induced ADTs $\mathcal{A}_{\mathcal{M}}$ and $\mathcal{A}_{\mathcal{M}'}$ are such that $\mathcal{A}_{\mathcal{M}'}$ refines $\mathcal{A}_{\mathcal{M}}$.

4.2 ADTs in the ghost language of VCC

In this section we describe how one can model ADTs in VCC, which is a tool for Verifying Concurrent C programs [15]. We first introduce the *ghost language* of VCC, which can be used to model an ADT. Then we describe how an ADT can be specified in the ghost language.

4.2.1 VCC's ghost language

The tool VCC provides a ghost language for specifying the properties to be verified about a program. For instance, its ghost language provides the `assert` keyword, which can be used to specify a requirement on the program state at a particular point in the program. VCC can automatically check that a program satisfies a given property specified as annotations like `assert` statements.

The ghost language provides mathematical objects, and operators to change the state of these mathematical objects. For instance, it supports natural numbers (via the keyword `\natural`) with operators for performing operations like addition and multiplication.

VCC provides the ghost language in such a manner that the mathematical objects provided by it will not interfere with the executable data objects in the program. Hence a user can safely add ghost statements with mathematical objects in her program without worrying about the unexpected changes to the execution semantics of her program [16].

The ghost language is not as rich as the Z specification language, since it does not support some of the mathematical objects and operators provided in Z. For instance, the ghost language does not implicitly support mathematical

```
struct
{
   _(ghost bool ready[\natural]);
   _(ghost \natural length);
   _(ghost \natural bound);

   _(invariant length <= bound)
} Scheduler;
```

Figure 4.10: A struct in VCC modeling the states of an ADT of type *SchedType*.

objects like sets and sequences, which are provided in the Z language. Nevertheless, one can use existing mathematical objects in the ghost language to model such missing mathematical objects.

The ghost language provides the mathematical object *map*, which is a handy tool for modeling other mathematical objects like sequences and sets. "`_(ghost range_type map_name domain_type)`" is the syntax for declaring a map in VCC. For example, we can declare an infinite sequence of integers, namely `ready` as "`_(ghost int ready \natural)`", where `\natural` represents the set of natural numbers $\mathbb{N}$. An element in such a sequence can be accessed using a syntax similar to the syntax for accessing array elements in the C language. For instance, `ready[i]` gives the i^{th} integer in the above sequence.

4.2.2 Modeling ADTs in the ghost language

In this section we explain how one can use the ghost language to model an ADT. We first describe how the states of an ADT can be modeled in the ghost language and then we describe how the operations of an ADT can be modeled in this langiage.

Modeling the states of an ADT

A "*struct*" in C can be used to model the states of an ADT. A struct allows a programmer to group a set of related data into a single entity. In VCC, a user can add *invariants* in a struct, in addition to the field declarations. An invariant constrains the valuations for the fields in the struct.

The states of an ADT can be modeled using a struct with ghost fields to model the state components of the ADT. The constraints on legal states of the ADT can be specified as invariants in the struct. We use the term *state-structure* to denote a struct in VCC modeling the states of an ADT.

Fig. 4.10 shows a state-structure modeling the states of an ADT of type *SchedType* of Fig. 4.1. The set of ready tasks is modeled as a ghost field `ready`, which is a map from the set of natural numbers to the set of bit values (boolean). This map resembles a sequence of bit values. Recall that the type

```
void init( _(ghost \natural maxLength) )
{
   _(ghost Scheduler.length = 0);
   _(ghost Scheduler.bound = maxLength);
}
```

Figure 4.11: A ghost method, which models the *init* operation of an ADT of type *SchedType*.

```
void create( _(ghost \bool task) )
  _(requires (Scheduler.length < Scheduler.bound))
{
   _(ghost Scheduler.ready[Scheduler.length] = task);
   _(ghost Scheduler.length += 1);
}
```

Figure 4.12: A ghost method, which models the *create* operation of an ADT of type *SchedType*.

of a task is assumed to be the set of bit values in the ADT type *SchedType*. The ghost variable `length` models the number of tasks present in the set of ready tasks and the ghost variable `bound` models the capacity of the ready list.

The invariants in a state-structure specify the legality constraints on the states of the ADT. For example, the invariant in the state-structure of Fig. 4.10 requires that in a legal state, the number of tasks present in the ready list is less than or equal to the value of the variable `bound`.

Modeling the operations of an ADT

We now describe how we model the operations of an ADT, using the ghost language.

We use *ghost methods* to implement the operations of an ADT. A ghost method is similar to a method (function implementation) in C, except that: (i) it is meant to update an instance of the state-structure modeling the states of an ADT, which uses ghost fields rather than executable fields, (ii) there can be a precondition associated with a ghost method, and (iii) a ghost method should restore the invariants in the state-structure.

Fig. 4.11 shows a ghost method, which models the *init* operation of an ADT of type *SchedType*. We assume that the instance of the state-structure (`Scheduler`) is declared as a global variable. The ghost method, `init` does not have a precondition.

A ghost method is a sequence of ghost statements as opposed to a set of predicates in the case of an operation-schema in the Z language. That is, the order of ghost statements is an important factor deciding the state change induced by a ghost method, while the order of the predicates does not affect

```
bool reschedule( _(ghost bool task) )
{
   bool res;
   _(ghost Scheduler.ready[Scheduler.length] = task);
   _(assume res == Scheduler.ready[0])
   _(ghost Scheduler.ready = \lambda \natural i;
      (i < Scheduler.length) ?
         Scheduler.ready[i+1]
      :  Scheduler.ready[i]
   ;
   return res;
}
```

Figure 4.13: A ghost method, which models the *resched* operation of an ADT of type *SchedType*.

the state change represented by an operation-schema.

Fig. 4.12 shows a ghost method, which models the *create* operation of an ADT of type *SchedType*. The ghost keyword `requires` can be used to specify a precondition on a method. The ghost method `create` has a precondition: the number of ready tasks in a valid before-state is at least one less than the maximum number of tasks allowed in the set of ready tasks. Without such a precondition, the method may fail to restore the invariant in the state-structure.

Fig. 4.13 shows a ghost method, which models the *resched* operation of an ADT of type *SchedType*. The ghost language provides the lambda operator to update a map instance. For example, the lambda operator is used in the ghost method `reschedule` to update the map `ready`, which essentially performs a left shift on this sequence. VCC does not allow a ghost method to return a ghost field and hence the ghost keyword `assume` is used in this method to transfer the value of a ghost field into an executable field `res`, which is returned from this method.

We discuss a technique in Sec. 4.3, to view a C implementation of an ADT-like system as an ADT. A similar technique can be used to view a ghost model as an ADT, except that ghost objects are used in a ghost model rather than executable objects. If we apply this technique to the ghost model discussed in this section, then we get the ADT of Fig. 4.9 again.

4.3 Viewing C implementations as ADTs

We present a technique in this section to obtain an ADT-TS and hence an ADT from a C program, which represents an ADT. The same method could be applied to obtain an ADT from a ghost implementation also.

We assume that an ADT implementation in C is a program $\mathcal{P}$ as follows. The program $\mathcal{P}$ comprises a set of global variables *Var*. Each variable $v \in Var$

is associated with a type T_v, which is the set of possible values for v. It has a finite set of function names F with an associated method (function definition) $\texttt{func}_n$ for each $n \in F$, which could contain local variables.

The program $\mathcal{P}$ can be translated to an ADT-TS $\mathcal{S}_\mathcal{P}$ as follows. A state of the ADT-TS comprises the following: (i) a statement number l, (ii) a valuation s to the variables (both global and local), (iii) a heap state h that maps heap locations to values, and (iv) a stack k to model method calls. We assume a special statement number 0 so that states of the form $(0, s, h)$ (with s mapping local variables to an uninitialized value "u") form the complete states Q_c of the required ADT-TS. Other states form the internal states Q_l (which may include values for local variables) of the required ADT-TS (see Def. 3.1).

The local action labels of the required ADT-TS are simply the statements of the program. 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 when any method in F returns.

We would also like to consider C implementations that have a *precondition* for each method. We assume that the precondition for method $\texttt{func}_n$ is a predicate pre_n on the set of complete states and inputs of the method. We view such a C method as an ADT-TS that is defined as before, except that for complete states and inputs that don't satisfy pre_n, the ADT-TS transitions to a "dead" local state.

The above procedure can be used to obtain the ADT-TS $\mathcal{S}_\mathcal{P}$ from a given ADT implementation $\mathcal{P}$ in the C language. Then the technique presented in Sec. 3.1, could be used to obtain the ADT $\mathcal{A}_{\mathcal{S}_\mathcal{P}}$ from the ADT-TS $\mathcal{S}_\mathcal{P}$. For example, Fig. 3.6, shows the ADT-TS induced by the method `reshed` of Fig. 1.8. The ADT induced by this ADT-TS is shown in Fig. 3.3, in the form of operation relations in the ADT.

Chapter 5

Methodology for Proving Functional Correctness

We now present a methodology based on our theory of refinement to prove the functional correctness of a given imperative language implementation of an ADT-like system. We are interested in verifying that a given C implementation of an ADT-like system is a refinement of its abstract ADT specification. We describe a couple of techniques for phrasing the refinement conditions between ADTs in different languages. We also propose techniques for using off-the-shelf tools for verifying these refinement conditions.

5.1 Directed refinement methodology

Here we present a methodology for proving the functional correctness of an imperative language implementation of an ADT-like system. The proposed methodology is based on our theory of refinement. In our methodology, one would need to do a sequence of successive refinements, starting from an abstract ADT model towards the given ADT implementation. Hence we call this a "directed refinement methodology". Our aim is to make use of available tools like VCC/Rodin to carry out the proofs of successive refinements in the proposed methodology. Fig. 5.1 shows the different steps required in our methodology for proving the functional correctness of an ADT implementation.

We describe below the different steps proposed in our methodology for proving the functional correctness of an imperative language implementation of an ADT-like system.

1. To begin with we view the given implementation as implementing an ADT of a certain type $\mathcal{N} = (N, (I_n)_{n \in N}, (O_n)_{n \in N})$. This may require us to elide certain code from the given implementation, or to transform some parts of it to reflect this view. The following are the deviations possible in a given implementation from being an ADT implementation: (i) there may not be a separate *init* operation, but rather the initialization code is set to execute in the first invocation to an operation in N and (ii) interactions (between operations) are not purely *functional*, for example

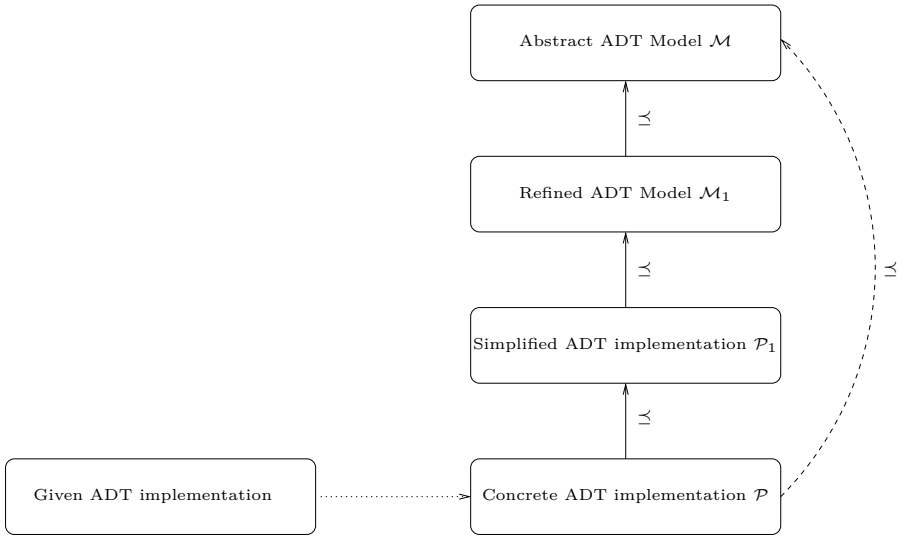


Figure 5.1: Illustrating "directed refinement methodology" for proving the functional correctness of a given ADT implementation $\mathcal{P}$.

an implementation of an ADT of type *SchedType* (of Fig. 4.1) may use a shared global variable to represent the running task, which needs to be updated by different operations in the *SchedType*. In the first case, we need to move such initialization code to a separate *init* operation. In the second case, we have to eliminate such shared updates by adding suitable arguments and return value to the operations in N. Let $\mathcal{P}$ be the concrete implementation of the ADT obtained from the given implementation by applying the required transformations like above.

2. Based on a high-level understanding of the code, and documentation like user manual and comments in the code, construct an abstract ADT model $\mathcal{M}$ in a high-level specification language like Z, that captures the *intended* behavior of the implementation. An example of such an abstract ADT model could be the Z model of the scheduler ADT in Sec. 4.1.2.

3. In general, the concrete implementation $\mathcal{P}$ may make use of several sub-ADTs, 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 an abstract version $\mathcal{A}_i$ of it expressed using the high-level constructs like maps of the ghost language available in tools like VCC. We refer to this simplified implementation $\mathcal{U}[\mathcal{A}_1, \ldots, \mathcal{A}_n]$ of the concrete implementation $\mathcal{P}$, as $\mathcal{P}_1$.

   For example, let $\mathcal{P}$ be the ADT implementation of type *SchedType* (see Fig. 4.1), a part of which is shown in Fig. 1.8. This ADT uses the sub-ADT `c-queue` of Fig. 1.1. Thus $\mathcal{P}$ is of the form $\mathcal{U}[\mathcal{B}]$, or more precisely `c-sched`[`c-queue`]. Now we could replace `c-queue` in `c-sched`[`c-queue`] with `g-queue` (an abstract ghost version in VCC given in Fig. 1.6), to

obtain the simplified ADT implementation $\mathcal{P}_1$. Thus $\mathcal{P}_1$ is of the form $\mathcal{U}[\mathcal{A}]$, or more precisely `c-sched`[`g-queue`].

4. Refine the abstract ADT model $\mathcal{M}$ towards the simplified implementation $\mathcal{P}_1$, via a sequence of successively refined ADT models, that add increasing details of the implementation. Let $\mathcal{M}_1$ be the resulting ADT model that is sufficiently "close" to $\mathcal{P}_1$. The number of intermediate ADT models required and implementation details to be added could be decided based on the gap between $\mathcal{M}$ and $\mathcal{P}_1$.

   For example, suppose the abstract ADT model has a single sequence of ready tasks with length 1024 and $\mathcal{P}_1$ implements it with four ready lists of maximum length 256 each, where 256 is the maximum length supported by the hardware. Then we could refine $\mathcal{M}$ to $\mathcal{M}_1$ by dividing the single sequence of ready tasks to four sequences to reduce the gap between $\mathcal{M}$ and $\mathcal{P}_1$. We discuss in Sec. 5.2.1, a technique for phrasing the refinement conditions between ADT models in Z.

5. Check that $\mathcal{P}_1$ refines $\mathcal{M}_1$. We discuss a technique for doing this in Sec. 5.2.2. At the end of this step, we would have function contracts in the form of `requires` and `ensures` predicates in a tool like VCC, for the ghost implementations of the sub-ADTs, that were used to prove that $\mathcal{P}_1$ refines $\mathcal{M}_1$.

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 $\mathcal{B}_i$. For instance, we need to prove in the above example that `c-queue` refines `g-queue`. We discuss a technique for phrasing the refinement conditions between a ghost model and a C implementation in Sec. 5.2.5.

If these checks are successful, we can conclude using our transitivity result (Proposition 2.1) and substitutivity result (Theorem 3.2) that:

$$\mathcal{P} = \mathcal{U}[\mathcal{B}_1, \ldots, \mathcal{B}_n] \preceq \mathcal{U}[\mathcal{A}_1, \ldots, \mathcal{A}_n] = \mathcal{P}_1 \preceq \mathcal{M}_1 \preceq \mathcal{M}.$$

We note that the given implementation and hence both $\mathcal{P}$ and $\mathcal{P}_1$ may not conform to the abstract ADT model $\mathcal{M}$ and we would need to work with suitably fixed versions for proofs to go through.

5.2 Phrasing refinement conditions

We present in this section, different techniques for phrasing the refinement conditions between ADTs in different languages. In particular, we consider refinement between ADTs in five different contexts: (i) refinement between Z models, (ii) refinement between Z and C models, (iii) refinement between ghost models, (iv) refinement between ghost and C models, and (v) refinement between C models.

Notation	Meaning
$State^{\mathcal{M}}$	*the data-schema of the Z model* $\mathcal{M}$
$Op_n^{\mathcal{M}}$	*the operation-schema of the Z model* $\mathcal{M}$*, which represents the operation* n
$X_n^{\mathcal{M}}$	*the type of the argument to* $Op_n^{\mathcal{M}}$
a	*the value of the argument to an operation-schema*
$Y_n^{\mathcal{M}}$	*the type of the return value from* $Op_n^{\mathcal{M}}$
b	*the value of the output from an operation-schema*
$pre_n^{\mathcal{M}}$	*the precondition of* $Op_n^{\mathcal{M}}$
$BAP_n^{\mathcal{M}}$	*the before-after predicate of* $Op_n^{\mathcal{M}}$
$\texttt{State}^{\mathcal{B}}$	*the state-structure modeling the states of the ADT in a ghost or C model* $\mathcal{B}$
$\texttt{func}_n^{\mathcal{B}}$	*the method which implements the operation* n *in the model* $\mathcal{B}$
$\texttt{Arg}_n^{\mathcal{B}}$	*the type of the argument to* $\texttt{func}_n^{\mathcal{B}}$
$\texttt{Ret}_n^{\mathcal{B}}$	*the type of the return value from* $\texttt{func}_n^{\mathcal{B}}$
$pre_n^{\mathcal{B}}$	*the precondition of* $\texttt{func}_n^{\mathcal{B}}$
s and s'	*the before-state and after-state of an operation in an abstract ADT*
t and t'	*the before-state and after-state of an operation in a concrete ADT*
$u'.y$	*the output variable in the after-state* u' *of a method*
inv_u	*the state-invariant in the ADT state* u *(*u *is called legal when it satisfies* inv_u*)*
inv_ρ	*the gluing invariant, which relates the concrete states to the abstract states*

Table 5.1: Notation used in this chapter.

In each of the above cases, depending on the types of the abstract and concrete ADT models, we need to consider one of the following: (i) the sufficient condition (RC) of Sec. 2.6, (ii) the sufficient condition (RC-TS) of Sec. 3.3 or (iii) a combination of these.

We show how to formulate the refinement conditions between Z models as a logical formula, which can be checked in a tool like Z/Eves [44], or Rodin [4]. For each of the other cases mentioned above, we show how to phrase the refinement conditions as code level annotations in a tool like VCC [15] or Verifast [29]. We use the `requires` annotation, which specifies a condition on the state and input that is assumed to hold when the method is invoked, and the `ensures` annotation (which asserts the condition expected to hold when the method returns) to specify the refinement conditions as code level annotations.

Table 5.1 shows the notation used in this chapter. The predicate $\mathcal{F}_n^{\mathcal{M}}(x, y)$ is similar to the predicate $\mathcal{F}_n^{\mathcal{M}}$, except that the arguments x and y denote one of the following: (i) free variables in $\mathcal{F}_n^{\mathcal{M}}$, (ii) values for free variables in $\mathcal{F}_n^{\mathcal{M}}$ or (iii) a combination of these.

5.2.1 Refinement between Z models

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two Z models, each of which represents an ADT. Then we can phrase the sufficient condition (RC) of Sec. 2.6, for $\mathcal{M}_1$ and $\mathcal{M}_2$ logically as follows. The Z model $\mathcal{M}_2$ refines the Z model $\mathcal{M}_1$ iff they satisfy all of the following conditions:

- $Op^{\mathcal{M}_1} = Op^{\mathcal{M}_2}$, and input/output types for each $n \in Op^{\mathcal{M}_1}$ match (i.e. $X_n^{\mathcal{M}_1} = X_n^{\mathcal{M}_2}$ and $Y_n^{\mathcal{M}_1} = Y_n^{\mathcal{M}_2}$).

- There exists a predicate ρ on $Var^{\mathcal{M}_1} \cup Var^{\mathcal{M}_2}$ that satisfies the following conditions:

  – For each $a \in X_{init}^{\mathcal{M}_1}$, $q \in Q^{\mathcal{M}_1}$, $b, b' \in Y_{init}^{\mathcal{M}_1}$, and $q' \in Q^{\mathcal{M}_2}$:

  $$[BAP_{init}^{\mathcal{M}_1}(a, q, b) \wedge BAP_{init}^{\mathcal{M}_2}(a, q', b')] \implies b = b' \wedge \rho(q', q),$$

  – and for each $n \in Op^{\mathcal{M}_1}, a \in X_n^{\mathcal{M}_1}, p, q \in Q^{\mathcal{M}_1}$, $b \in Y_n^{\mathcal{M}_1}$, and $p' \in Q^{\mathcal{M}_2}$:

  $$[\rho(p', p) \wedge BAP_n^{\mathcal{M}_1}(p, a, q, b)] \implies \exists\, q' \in Q^{\mathcal{M}_2} \text{ such that } BAP_n^{\mathcal{M}_2}(p', a, q', b) \wedge \rho(q', q).$$

Such a condition can be checked in a theorem prover for Z like Z/Eves [44], or Rodin [4], or even by a suitable translation into a code verifier like VCC [15].

5.2.2 Refinement between Z and C models

Let $\mathcal{M}$ be a Z model of an ADT of type $\mathcal{N}$ and $\mathcal{P}$ be a C implementation of an ADT of type $\mathcal{N}$. Then we describe a technique for phrasing the refinement conditions for $\mathcal{P}$ to refine $\mathcal{M}$. The idea is to directly *import* the requirements from the Z model $\mathcal{M}$ as code level annotations in $\mathcal{P}$. Here we existentially quantify away the abstract state and hence the resulting `requires` and `ensures` annotations are independent of the abstract state. The required annotations in the methods of $\mathcal{P}$ are shown in Fig. 5.2.

We note that by assumption, the *init* operation depends only on the input, and not the before-state. For each input satisfying the abstract precondition ($pre_{init}^{\mathcal{M}}$) of the *init* operation, we need to verify the following: (i) the method $\texttt{func}_{init}^{\mathcal{P}}$ terminates, (ii) there exists an abstract legal after-state s' to which the concrete after-state t' is related by the abstract relation ρ, and (iii) the operation $Op_{init}^{\mathcal{M}}$ and the method $\texttt{func}_{init}^{\mathcal{P}}$ produce the same output. These annotations essentially capture the following: the operation $Op_{init}^{\mathcal{M}}$ and the method $\texttt{func}_{init}^{\mathcal{P}}$ end with states related by ρ and produce the same output, when invoked with an input satisfying the abstract precondition. We note that $Op_{init}^{\mathcal{M}}$ enters the exception state with e as output, when the operation is invoked with an input not satisfying the abstract precondition, and hence there is no demand on the method $\texttt{func}_{init}^{\mathcal{P}}$ to simulate the operation $Op_{init}^{\mathcal{M}}$.

The annotations for a method $\texttt{func}_n^{\mathcal{P}}$, also has three components similar to the case above, except that in addition, these formulas use the before-state as well. The annotations in this case essentially capture the following: the operation $Op_n^{\mathcal{M}}$ and the method $\texttt{func}_n^{\mathcal{P}}$ end with a joint after-state satisfying ρ such that the abstract after-state is legal, and both the abstract and concrete operations produce the same output; when the operation is invoked in a joint before-state satisfying the abstract state-invariant, abstract precondition and the gluing invariant.

(init-a) $\texttt{func}^{\mathcal{P}}_{init}$ must terminate on all inputs x for which $init^{\mathcal{M}}$ is defined (i.e. $pre^{\mathcal{M}}_{init}(x)$ is true).

(init-b)
$\texttt{func}^{\mathcal{P}}_{init}(X^{\mathcal{M}}_{init}\ \texttt{x})$
$\quad$ _(requires $pre^{\mathcal{M}}_{init}(\texttt{x})$)
$\quad$ _(ensures $\exists\, s' : BAP^{\mathcal{M}}_{init}(\texttt{x}, s', s'.\texttt{y}) \wedge\ inv_{s'} \wedge inv_{\rho}(t', s')$)
$\quad$ _(ensures $t'.\texttt{y} = s'.\texttt{y}$))
{
$\quad$ // function body
}

(sim-a) For each operation n, $\texttt{func}^{\mathcal{P}}_{n}$ must terminate on all state-input pairs (t, x) such that there exists a state-input pair (s, x) of $\mathcal{M}$ satisfying: $inv_s \wedge pre^{\mathcal{M}}_{n}(s, x) \wedge inv_{\rho}(t, s)$.

(sim-b) For each operation n:

$\texttt{func}^{\mathcal{P}}_{n}(X^{\mathcal{M}}_{n}\ \texttt{x})$
$\quad$ _(requires $\exists\, s : inv_s \wedge pre^{\mathcal{M}}_{n}(s, \texttt{x}) \wedge inv_{\rho}(t, s)$)
$\quad$ _(ensures $\exists\, s, s' : inv_{\rho}(t, s) \wedge BAP^{\mathcal{M}}_{n}(s, \texttt{x}, s', s'.\texttt{y}) \wedge inv_{s'} \wedge inv_{\rho}(t', s')$))
$\quad$ _(ensures $t'.\texttt{y} = s'.\texttt{y}$)
{
$\quad$ // function body
}

Figure 5.2: Directly importing the requirements from the Z model $\mathcal{M}$ as code level annotations in the C program (see Table 5.1). Here t and t' represent the before-state and after-state respectively of the methods.

Requirement	Possible Z spec.	Equivalent VCC spec.
Set of elements of type T.	A_T : ℙ T	_(ghost \bool A_T[T])
Set membership.	e ∈ 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 ∪ B_T	\forall T t;C_T[t]<==>A_T[t]||B_T[t]

Table 5.2: A table showing the procedure for translating *sets* and operations in Z, to ghost objects and operations in VCC. If X is a set, then the notation "A_X" denotes an arbitrary subset of X.

5.2.3 Z-to-VCC translation

Motivation: The two approaches described so far to check the refinement conditions between two Z models and between Z and C models suffer from the following disadvantages. The first is that performing a refinement proof between abstract models, is challenging because the level of automation in tools such as Z/Eves [44] and Rodin [4] 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 is the difficulty in showing the refinement between a declarative model and an imperative language model. The problem here is that there is no tool which understands *both* these modeling languages. One way of getting around this is to use the technique explained in Sec. 5.2.2, to "import" the before-after-predicates (*BAP*s) from the declarative model to the concrete implementation, by using `requires` and `ensures` clauses that are equivalent to formulas in which the abstract state is existentially quantified away. However there are some disadvantages of 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.

We describe a technique using a VCC-like tool that overcomes some of these difficulties. The idea is to first translate the high-level Z model to a ghost model in VCC and then do the successive refinements completely within VCC. How does this help us to get around the problems mentioned above? The first problem of proving refinement between abstract models is alleviated as VCC is typically able to check the refinement between ghost models efficiently and automatically. The second problem of moving from an abstract model to an imperative implementation is also addressed because we now have both the abstract and the concrete models in a language that VCC understands.

Another advantage of this approach is that the ghost model can be used to verify the local properties of clients of the ADT (see Theorem 2.2).

We propose a mechanizable procedure for translating a Z model to a ghost model in VCC. Our aim is to translate a given model $\mathcal{M}$ in the Z language to

Requirement	Possible Z spec.	Equivalent VCC spec.
Partial function from X to Y	pMap : X ⇸ Y	_(ghost Y pMap[X]) _(ghost \bool pMapDom[X])
Domain restriction for maps	g = A_X ◁ f	\forall gDom x;(g[x]==f[x])∧ \forall X x; gDom[x]<==>(fDom[x] && A_X[x])
Domain subtraction for maps.	g = A_X ⩤ f	\forall gDom x; g[x]==f[x])∧ \forall X x; (gDom[x]<==>(fDom[x] && !A_X[x])
Range restriction for maps.	g = f ▷ B_Y	\forall gDom x; g[x]==f[x])∧ \forall X x; (gDom[x]<==>(fDom[x] && B_Y[f[x]])
Range subtraction for maps.	g = f ⩥ B_Y	\forall gDom x; g[x]==f[x])∧ \forall X x; (gDom[x]<==>(fDom[x] && !B_Y[f[x]])
Relational overriding for maps (f:X⇸Y, g:X⇸Y).	h = f ⊕ g	\forall hDom x;((gDom[x]==>(h[x]==g[x]))∧ (!gDom[x]==> (h[x]==f[x]))∧\forall X x; (hDom[x]<==>(fDom[x]||gDom[x]))
Containment of relational image (f:X⇸Y).	f(⦇ A_X ⦈) ⊆ A_Y	\forall X x;A_X[x]==>A_Y[f[x]]

Table 5.3: A table showing the procedure for translating *maps* and operations in Z, to ghost objects and operations in VCC.

a model $\mathcal{G}$ in the ghost language of VCC, to meet the following objectives: (i) the ghost model $\mathcal{G}$ refines the Z model $\mathcal{M}$ and (ii) the ghost model $\mathcal{G}$ admits an easy proof of the ensuing refinement conditions. We propose a mechanizable translation procedure, which achieves the above objectives.

We present only those Z constructs that we use in the Z models of the case-studies in this thesis. Nevertheless other mathematical objects in Z can be handled in a similar way.

Let $\mathcal{M}$ be a Z specification of an ADT of type $\mathcal{N}$. We translate the data-schema $State^{\mathcal{M}}$ to a state-structure $\texttt{State}^{\mathcal{G}}$ (see Table 5.1). The idea is to make use of: (i) ghost fields in $\texttt{State}^{\mathcal{G}}$ to model the fields of $State^{\mathcal{M}}$ and (ii) an invariant in $\texttt{State}^{\mathcal{G}}$ to model the invariant of $State^{\mathcal{M}}$. For example, our proposed translation procedure will result in the state-structure of Fig. 4.10, when applied to the data-schema of Fig. 4.2.

We translate an operation-schema $Op_n^{\mathcal{M}}$ to a ghost method $\texttt{func}_n^{\mathcal{G}}$. The idea here is to make use of: (i) a ghost parameter in $\texttt{func}_n^{\mathcal{G}}$ to model the input of $Op_n^{\mathcal{M}}$, (ii) `requires` annotation to model the precondition of $Op_n^{\mathcal{M}}$, and (iii) a *sequence* of ghost statements in $\texttt{func}_n^{\mathcal{G}}$ to model the before-after predicate $BAP_n^{\mathcal{M}}$. For example, our proposed translation procedure will result in the ghost method of Fig. 4.12, when applied to the operation-schema of Fig. 4.5.

We propose a table look-up procedure to encode some of the Z objects and operators in VCC, using the ghost language of VCC. Table 5.2, Table 5.3 and Table 5.4, present look-up procedures for encoding the Z objects: *sets*, *maps* and *sequences* respectively, using ghost objects in VCC.

The Z objects are encoded in VCC in a way that facilitates easy proofs

Requirement	Z spec.	Equivalent VCC spec.
An 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 ∈ ran s	sIndex[e] < sLength
Disjoint sequences.	ran s ∩ ran t = ∅	\forall T e; ((sIndex[e]<sLength==> tIndex[e]>=tLength)∧ (tIndex[e]<tLength==>sIndex[e]>=sLength))
Sequence containment.	ran s ⊆ ran t	\forall T e; sIndex[e]<sLength==> tIndex[e]<tLength
Concatenation for injective sequences of the same type (s, t are disjoint sequences of type T).	u = s ⌢ 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 a sequence of type T.	t = s ↾ 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]])
Extraction operation for a sequence of type T.	t = s ⇂ 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 5.4: A table showing the procedure for translating *sequences* and operations in Z, to ghost objects and operations in VCC.

for the ensuing verification conditions. Consider the encoding proposed in Table 5.4, for modeling a *sequence* of elements of an arbitrary type `T`. Let `sElements` be a ghost map modeling a sequence in VCC. Then we use the following two auxiliary objects to encode this sequence to admit an easy proof of the ensuing verification conditions in VCC. The first auxiliary object is a ghost variable called `sLength`, which models the number of elements present in the sequence modeled by the ghost object `sElements`. The second auxiliary object is a ghost map called `sIndex`, which models the positions of elements in the sequence. The map `sIndex` could be defined as follows. The value of an element v under this map (that is, `sIndex[`v`]`) is less than the value of the variable `sLength` iff `sElements[sIndex[`v`]] =` v. Thus VCC requires manual help to make the verification process *efficient* and *tractable* for large programs.

Now we illustrate the use of the above encoding to reduce the complexity of the ensuing verification condition in VCC. Suppose in the Z model, we have the predicate "$v \in \text{ran } s$" which denotes "v is an element of the sequence s". How can we translate this predicate to VCC? A straight forward encoding of this predicate in VCC could be "$\exists\, pos : \mathbb{N} \mid$ (`sElements[`pos`]` $= v$)". But this encoding uses an existential quantifier, which is difficult for a theorem prover to handle. Suppose we want to check the validity of a verification condition φ, which uses existential quantifications. A theorem prover usually checks its validity by checking the *satisfiability* of the negation of the verification condition φ (i.e. $\neg\varphi$). Now it is difficult to check the satisfiability of the formula $\neg\varphi$, since existential quantifications in φ become universal quantifications in $\neg\varphi$.

Our proposed translation procedure encodes the mathematical objects from the Z language without making use of existential quantifications. This helps us in getting a verification condition which is a lot easier for the prover to solve than otherwise. For instance, the use of auxiliary objects such as `sLength` and `sIndex` help us in avoiding the use of existential quantifications in the encoding of sequences in VCC. In particular, the membership query in the above example could be encoded using these auxiliary objects as "`sIndex[`v`] < sLength`".

5.2.4 Refinement between ghost models

We now show how to phrase the refinement conditions when both the abstract and concrete ADTs are ghost models in VCC. Recall that a ghost model comprises a state-structure modeling the states of the ADT and a set of ghost methods implementing the operations of the ADT.

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two ghost models of an ADT of type $\mathcal{N}$. Then we can encode the refinement condition (RC-TS) of Sec. 3.3, for the model $\mathcal{G}_2$ to refine the model $\mathcal{G}_1$ as follows. We construct a "joint" model $\mathcal{G}_{1\&2}$ such that: (i) the state-structure $\texttt{State}^{\mathcal{G}_{1\&2}}$ combines the contents of $\texttt{State}^{\mathcal{G}_1}$ and $\texttt{State}^{\mathcal{G}_2}$, (ii) in $\texttt{State}^{\mathcal{G}_{1\&2}}$ an invariant "inv_ρ" asserts the abstraction relation ρ from the states of $\mathcal{G}_2$ to the states of $\mathcal{G}_1$, and (iii) for each operation n in N, there is a joint ghost method $\texttt{func}_n^{\mathcal{G}_{1\&2}}$ which executes the bodies of the methods $\texttt{func}_n^{\mathcal{G}_1}$ and $\texttt{func}_n^{\mathcal{G}_2}$ (see Table 5.1).

Fig. 5.3 shows how we can phrase the refinement condition (RC-TS) using

```
State^{G1&2}
{
    // contents of State^{G1}
    // contents of State^{G2}
    inv_ρ
};
```

(init)
```
func^{G1&2}_{init}(Arg^{G1}_{init} x)
  _(requires pre^{G1}_n)
  _(ensures inv_{s'} ∧ inv_ρ ∧ s.'y = t'.y)
{
   // body of func^{G1}_{init}
   // body of func^{G2}_{init}
}
```

(sim) For each operation n:

```
func^{G1&2}_{init}(Arg^{G1}_n x)
  _(requires inv_s ∧ pre^{G1}_n ∧ inv_ρ)
  _(ensures inv_{s'} ∧ inv_ρ ∧ s.'y = t'.y)
{
   // body of func^{G1}_n
   // body of func^{G2}_n
}
```

Figure 5.3: Phrasing the refinement between ghost models (see Table 5.1).

the combined model $\mathcal{G}_{1\&2}$. It shows the joint structure and the combined methods for both the *init* operation and an operation n in N. For an operation n in N, the annotations essentially capture the following: if the execution of the combined ghost method starts in a joint state with input, satisfying the gluing (abstraction) relation (inv_ρ), and the abstract state-invariant and precondition; then the execution terminates in a joint state satisfying the gluing relation such that the abstract after-state is legal and both the abstract and concrete ghost methods produce the same output value.

5.2.5 Refinement between ghost and C models

Here we show how to phrase the refinement conditions, when the abstract ADT is a ghost model in VCC and the concrete ADT is a C program.

Let $\mathcal{G}$ be a ghost model of an ADT of type $\mathcal{N}$ and $\mathcal{P}$ be a C program of an ADT of type $\mathcal{N}$. Then we can encode the refinement condition (RC-TS) of Sec. 3.3, for the model $\mathcal{P}$ to refine the model $\mathcal{G}$ as follows. We construct a "joint" model $\mathcal{C}_{\mathcal{G},\mathcal{P}}$ such that: (i) the state-structure $\texttt{State}^{\mathcal{C}_{\mathcal{G},\mathcal{P}}}$ combines the contents of $\texttt{State}^{\mathcal{G}}$ and $\texttt{State}^{\mathcal{P}}$ (see Table 5.1), (ii) in $\mathcal{C}_{\mathcal{G},\mathcal{P}}$ an invariant inv_ρ asserts the abstraction relation ρ from the states of $\mathcal{P}$ to the states of $\mathcal{G}$, and (iii) for each operation n in N, there is a joint method $\texttt{func}_n^{\mathcal{C}_{\mathcal{G},\mathcal{P}}}$ which executes the bodies of the methods $\texttt{func}_n^{\mathcal{G}}$ and $\texttt{func}_n^{\mathcal{P}}$.

Fig. 5.4 shows how we can phrase the refinement condition (RC-TS) using the combined model $\mathcal{C}_{\mathcal{G},\mathcal{P}}$. The technique is similar to the technique explained above for phrasing the refinement conditions between ghost models, except that in addition we need to prove that a concrete method *terminates*, when the joint before-state with input satisfies the abstract state-invariant and precondition.

5.2.6 Refinement between C models

We have presented a technique in Sec. 4.3 to view C implementations as ADT transition systems. We now describe a technique for checking the refinement condition (RC-TS) of Sec. 3.3 between C programs implementing ADTs of a given type. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two C programs implementing ADTs of a type $\mathcal{N}$. Methods in the abstract C program $\mathcal{P}_1$ may have preconditions. We can phrase the sufficient condition (RC-TS) for refinement between $\mathcal{P}_1$ and $\mathcal{P}_2$ in a similar way as explained in Sec. 5.2.5 (C implementation refines ghost implementation), except for the following:

1. Let "$term_n^{\mathcal{P}_1}$" denote a predicate describing the set of state-input pairs on which $\texttt{func}_n^{\mathcal{P}_1}$ terminates. Then the condition $pre_{init}^{\mathcal{G}}$ in (init-a) and (init-b) can be replaced by $pre_{init}^{\mathcal{P}_1} \wedge term_{init}^{\mathcal{P}_1}$, and similarly $inv_s \wedge pre_n^{\mathcal{G}} \wedge inv_\rho$ in (sim-a) and (sim-b) can be replaced by $inv_s \wedge pre_n^{\mathcal{P}_1} \wedge inv_\rho \wedge term_n^{\mathcal{P}_1}$.

2. It must be the case that the function implementations of $\mathcal{P}_1$ and $\mathcal{P}_2$ don't "interfere" with each other. That is, there is no shared data which can

```
State^{C_{G,P}}
{
   // contents of State^G
   // contents of State^P
   inv_ρ
};
```

(init-a) $\texttt{func}^{\mathcal{P}}_{init}$ terminates on all joint state-input pairs satisfying $pre^{\mathcal{G}}_{init}$.

(init-b)
```
func_{init}^{C_{G,P}} (Arg_{init}^G x)
  _(requires pre_{init}^G)
  _(ensures inv_{s'} ∧ inv_ρ ∧ y_{init}^G = y_{init}^P)
{
   // body of func_{init}^G
   // body of func_{init}^P
}
```

(sim-a) For each operation n, $\texttt{func}^{\mathcal{P}}_n$ must terminate on all state-input pairs satisfying $inv_s \wedge pre^{\mathcal{G}}_n \wedge inv_\rho$.

(sim-b) For each operation n:

```
func_n^{C_{G,P}} (Arg_n^G x)
  _(requires inv_s ∧ pre_n^G ∧ inv_ρ)
  _(ensures inv_{s'} ∧ inv_ρ ∧ y_n^G = y_n^P)
{
   // body of  func_n^G
   // body of  func_n^P
}
```

Figure 5.4: Phrasing refinement between ghost model and C implementation (see Table 5.1).

Figure 5.5: A part of the Z model of an ADT of type $QType_{\mathbb{Z}}$.

```
struct                                  void enq(int val)
{                                         _(requires QC.lenC < SIZE)
   unsigned lenC;                         ...
   int arr[SIZE];                         _(ensures lenC <= SIZE)
   _(invariant lenC <= SIZE)            {
} QC;                                      QC.arr[QC.lenC] = val;
                                           QC.lenC++;
                                        }
```

Figure 5.6: A part of the C implementation of an ADT of type $QType_{\mathbb{Z}}$.

be updated by both $\mathcal{P}_1$ and $\mathcal{P}_2$. We present a technique in Sec. 5.5, for handling shared data.

5.3 Proving refinement conditions in VCC

In this section we describe techniques for proving the refinement conditions of Sec. 5.2, in the tool VCC. We consider refinements in two different contexts. The first one considers the refinement between a declarative model in a language like Z and an imperative implementation in a language like C. We describe a technique called the "direct-import" approach for proving refinement conditions in this case. The second one considers the refinement between two ADT models in VCC. To handle this kind of refinements, we describe a technique called the "combined" approach.

We illustrate these techniques using an example ADT of type $QType_{\mathbb{Z}}$, which is discussed in Sec. 3.1. A part of the Z model called `z-queue` of an ADT of type $QType_{\mathbb{Z}}$ is shown in Fig. 5.5. The data-schema in `z-queue` models the contents of the queue as a finite sequence of integers namely *zSeq*. The invariant in the data-schema bounds the length of this sequence by the constant *SIZE*. The figure also shows the operation-schema to insert an element to the queue. This operation-schema assumes that there is at least one vacant space in the queue to insert the new element.

Now consider a C implementation called `c-queue` of an ADT of type $QType_{\mathbb{Z}}$, a part of which is shown in Fig. 5.6. An array of integers namely `arr` represents the elements of the queue. The invariant of the struct avoids the possible array out of bounds error. The method for inserting an element

```
void enq(_(ghost unsigned val))
  _(requires \exists \natural lenZ, \exists int zSeq[\natural];
    ((lenZ <= SIZE) &&
     (lenZ < SIZE)  &&
     ((lenZ == lenC) &&
      (\forall \natural i; (i < lenZ) ==> (zSeq[i] == QC.arr[i])))))
  ...
  _(ensures \exists \natural lenZ, \exists int zSeq[\natural],
            \exists \natural lenZ', \exists int zSeq'[\natural];
    ((lenZ == \old(lenC)) &&
     (\forall \natural i;
       (i < lenZ) ==> (zSeq[i] == \old(QC.arr[i])))) &&
    ((lenZ' == lenZ + 1) &&
      (\forall \natural i; (i < lenZ) ==> (zSeq'[i] == zSeq[i])) &&
      (zSeq'[lenZ] == val)) &&
    (lenZ' <= SIZE) &&
    ((lenZ' == lenC) &&
    (\forall unsigned i; (i < lenZ') ==> (zSeq'[i] == QC.arr[i])))))
{
   // body of the concrete method
}
```

Figure 5.7: Illustrating the "direct-import" approach on the method `enq` of Fig. 5.6.

to the queue is also shown in the figure.

We fix the names `z-queue` and `c-queue` as above for the rest of this section. In the following subsections we describe two techniques in VCC to prove the refinement conditions between ADT models.

5.3.1 Direct-import approach

We now describe a technique in VCC to check the refinement conditions formulated in Sec. 5.2.2, that is the refinement between Z and C models. This approach is called the "direct-import" approach, since we directly import the requirements from the abstract model as code level annotations in VCC.

Fig. 5.7 illustrates this technique for checking the refinement conditions between `z-queue` and `c-queue`, based on the refinement conditions formulated in Sec. 5.2.2. If `x` is a global variable accessible to a method `func`, then the variable `x` in the `requires` and `ensures` annotations of the method `func` respectively denote the value of the variable `x` before and after executing the body of the method `func`. The expression `\old(x)` in the `ensures` annotation of `func` denotes the value of the variable `x` before the execution of the method `func`.

The above is a valid technique for checking refinement conditions in VCC. However it is not a practically feasible approach for large models, since the

```
void enq(int val)
  _(requires QC.lenC < SIZE)
  _(ensures
    (QC.lenC == \old(QC.lenC) + 1) &&
    (\forall \natural i;
     (i < \old(QC.lenC)) ==>
     (QC.arr[i] == \old(QC.arr[i])) &&
    (QC.arr[\old(QC.lenC)] == val))
  _(ensures QC.lenC <= SIZE)
  ...
{
   // body of the concrete method
}
```

Figure 5.8: Illustrating "direct-import approach with quantifier elimination".

underlying theorem prover in VCC is not good in handling existential quantifications.

A possible way to solve the problem with the "direct-import" approach is to manually transform the annotations to remove existential quantifications, and get an equivalent condition that holds over all models of linear arithmetic. The modified technique with manual transformation is called the "direct-import approach with quantifier elimination". For instance, Fig. 5.8 shows how we can do this for the above example. Here existential quantifications are not used in the function contract and hence VCC could efficiently check the function contract.

5.3.2 Combined approach

Now we explain a technique to use VCC for checking the refinement conditions formulated in Sec. 5.2.5, that is the refinement between ghost and C models. We assume here that the abstract ADT is available as a ghost model in VCC. For instance, Fig. 5.9 shows a part of the ghost model called `g-queue` which is obtained from the Z model `z-queue`. Recall that one can use our Z-to-VCC translation procedure of Sec. 5.2.3, to translate a Z model to a ghost model in VCC.

The idea is to use a joint model in VCC, which combines the abstract and concrete models to check the refinement conditions. Therefore this technique is called the "combined" approach. We use a joint state-structure to represent the combined states of the abstract and concrete ADTs and for each operation we use a joint method which executes the statements of both the abstract and the concrete methods.

To illustrate this technique, consider the models `g-queue` and `c-queue` discussed above. Fig. 5.10 shows the joint state-structure which combines the states of the models `g-queue` and `c-queue`, and Fig. 5.11 shows the joint method to check the refinement condition with respect to the operation

```
strcut                              void enq(_(ghost int val))
{                                     _(requires QG.lenG < SIZE)
   _(ghost \natural lenG)             _(ensures QG.lenG <= SIZE)
   _(ghost int gSeq[\natural])      {
   _(invariant lenG <= SIZE)           _(ghost QG.gSeq[QG.lenG] = val)
} QG;                                  _(ghost QG.lenG = QG.lenG + 1)
                                    }
```

Figure 5.9: A part of the ghost implementation of an ADT of type $QType_{\mathbb{Z}}$.

```
struct
{
  _(ghost \natural lenG)
  _(ghost int gSeq[\natural])
  _(invariant lenG <= SIZE)
  unsigned lenC;
  int arr[SIZE];
  _(invariant lenC <= SIZE)
  // gluing invariant
  _(invariant (lenG == lenC) && (\forall \natural i;
    (i < lenG) ==> (gSeq[i] == arr[i])))
} QGC;
```

Figure 5.10: The joint state-structure which combines the states of the models `g-queue` and `c-queue`.

```
void enqCombined(int val)
  _(requires (QGC.lenG <= SIZE) && (QGC.lenG < SIZE) &&
    ((QGC.lenG == QGC.lenC) &&
     (\forall \natural i; (i < QGC.lenG) ==>
                          (QGC.gSeq[i] == QGC.arr[i]))))
  _(ensures QGC.lenG <= SIZE) && ((QGC.lenG == QGC.lenC) &&
    (\forall \natural i; (i < QGC.lenG) ==>
                         (QGC.gSeq[i] == QGC.arr[i]))))
{
   // body of the abstract method
   // body of the concrete method
}
```

Figure 5.11: Joint method to check the refinement between the models `g-queue` and `c-queue` with respect to the operation *enq*.

```
unsigned factorial(unsigned val)
  _(decreases 0)
  ...
{
   unsigned fact, i;
   fact = 1;
   for(i = 1; i <= val; i++)
     _(decreases (val - i))
     ...
   {
      fact = fact * i;
   }
   return fact;
}
```

Figure 5.12: Illustrating the technique in VCC to prove termination.

enq. The function contract shown in this figure is generated from the models `g-queue` and `c-queue` by applying the technique described in Sec. 5.2.5.

This technique can be used to check the refinement conditions when both the abstract and concrete models are in languages that VCC understands. In particular, this technique can be used to check the refinement conditions formulated in Sec. 5.2.4 (between ghost models), Sec. 5.2.5 (between ghost and C models), and Sec. 5.2.6 (between C models).

5.4 Proving termination in VCC

In this section we describe how we use VCC to prove the termination of a method in C. Recall that some of the refinement conditions formulated in Sec. 5.2, require us to prove that a concrete method terminates. This is required when one of the ADT models is an imperative language implementation.

We use the existing technique in VCC to prove the termination of a method and which is to use the `decreases` annotation. In the function contract of a method, the annotation `_(decreases 0)` asserts that the method terminates. To prove this one should again use the `decreases` annotation in each loop in the method to ensure that the value of an expression (of type `unsigned` in C) strictly decreases between successive executions of the loop body. We illustrate this below with a simple example.

Consider a program in C to find the factorial of a number. Fig. 5.12 shows excerpts from a VCC model which illustrates how one can prove the termination of a method which finds the factorial of the given argument.

```
void enq(int t)
  ...
  _(ensures \old(cur) = cur)
{
   //body of the method
}
```

Figure 5.13: Illustrating function contract for verifying the requirements for ensuring the property "effectively functional".

5.5 Handling shared data

We assumed in our theory that a client ADT transition system like the scheduler implementation of Fig. 1.8, interacts in a purely "functional way" with its sub-ADT like the `c-queue` ADT of Fig. 1.1, in the sense that the interaction between the two is via arguments and return value only. In particular, they do not communicate via shared data. However most ADT transition systems violate this assumption. For example, the variable `cur`, which represents the currently running task in the scheduler of Fig. 1.8, could be defined as a global variable and hence violates the above assumption, since `c-queue` can modify this variable.

We can safely weaken the above assumption to require that interactions between a client ADT transition system and its sub-ADT is "*effectively functional*". The interactions between a client ADT transition system and its sub-ADT is said to effectively functional iff each of them does not modify any shared data, which is *owned* by the other. To verify this, one would need to first classify the ownerships of shared data between the ADT transition system and its client. For instance, the scheduler could be defined as the owner of the variable `cur` in the above example.

After classifying ownerships, one could use a suitable function contract to verify that a method in one component does not modify a shared data which is owned by the other component. The annotation `_(ensures \old(x) = x)`, in each method in one component could be used to ensure that this component does not modify the shared variable `x`, which is owned by the other component. For example, the annotation `_(ensures \old(cur) = cur)`, is required in each of the methods in the `c-queue` component, to ensure that the value of the shared variable `cur` is not modified by the `c-queue` component. The function contract for ensuring this in terms of the `enq` method in the program `c-queue` is shown in Fig. 5.13.

Chapter 6

Conformal ADTs and Refinement

This chapter is about a notion of refinement between ADTs whose operations may have different Input/Output (I/O) types. The verification guarantees provided by the proposed notion of refinement is same as that of the refinement notion for ADTs (see Sec. 2.5), but via an "interface" or "wrapper" that translates inputs and outputs between the abstract and the concrete ADTs. We extend the necessary and sufficient refinement condition of Sec. 2.6, to ADTs with different I/O types.

6.1 Conformal ADTs

In this section we introduce the concept of "conformal" ADTs. We also introduce a running example for this chapter to illustrate the proposed notion of refinement between conformal ADTs.

A notion of refinement between ADTs with operations supporting different I/O types is typically required when one refines an abstract ADT model to a concrete ADT implementation. For example, consider the scheduler ADT discussed in Sec. 4.1. A natural number is used as a unique task-id to represent a task in the Z model of the scheduler ADT of Fig. 4.2. Suppose a task is represented by a struct say TCB in a C implementation of this ADT. Now an ADT operation like *resched*, which takes a task as an input and returns a task as an output has the set of all natural numbers as the I/O type in the Z model, and the set of all TCBs as the I/O type in the C implementation.

Let $\mathcal{N}_1 = (N_1, (I^1_n)_{n \in N_1}, (O^1_n)_{n \in N_1})$ and $\mathcal{N}_2 = (N_2, (I^2_n)_{n \in N_2}, (O^2_n)_{n \in N_2})$ be two ADT types. Then $\mathcal{N}_1$ and $\mathcal{N}_2$ are said to be *conformal* iff $N_1 = N_2$. That is, conformal ADT types have the same set of operations with possibly different I/O types. For example, the ADT type $SchedType_1$ of Fig. 6.1 and the ADT type $SchedType_2$ of Fig. 6.2 are conformal, since they both have the set of operations $\{init, create, resched\}$. We fix the ADT types $\mathcal{N}_1$ and $\mathcal{N}_2$ as above for the rest of this chapter.

Let $\mathcal{A}_1$ be an ADT of type $\mathcal{N}_1$ and let $\mathcal{A}_2$ be an ADT of type $\mathcal{N}_2$. Then $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be *conformal*, if their types $\mathcal{N}_1$ and $\mathcal{N}_2$ are conformal.

$$
\begin{aligned}
SchedType_1 &= \{init, create, resched, I^1_{init}, O^1_{init}, I^1_{create}, O^1_{create}, I^1_{resched}, O^1_{resched}\} \\
\text{where: } I^1_{init} &= \{nil\} \\
O^1_{init} &= \{ok, e\} \\
I^1_{create} &= \mathbb{N} \\
O^1_{create} &= \{ok, fail, e\} \\
I^1_{resched} &= \mathbb{N} \\
O^1_{resched} &= \mathbb{N} \cup \{fail, e\}
\end{aligned}
$$

Figure 6.1: The ADT type $SchedType_1$.

$$
\begin{aligned}
SchedType_2 &= \{init, create, resched, I^2_{init}, O^2_{init}, I^2_{create}, O^2_{create}, I^2_{resched}, O^2_{resched}\} \\
\text{where: } I^2_{init} &= \{nil\} \\
O^2_{init} &= \{ok, e\} \\
I^2_{create} &= \text{TCB} \\
O^2_{create} &= \{ok, fail, e\} \\
I^2_{resched} &= \text{TCB} \\
O^2_{resched} &= \text{TCB} \cup \{fail, e\}
\end{aligned}
$$

Figure 6.2: The ADT type $SchedType_2$.

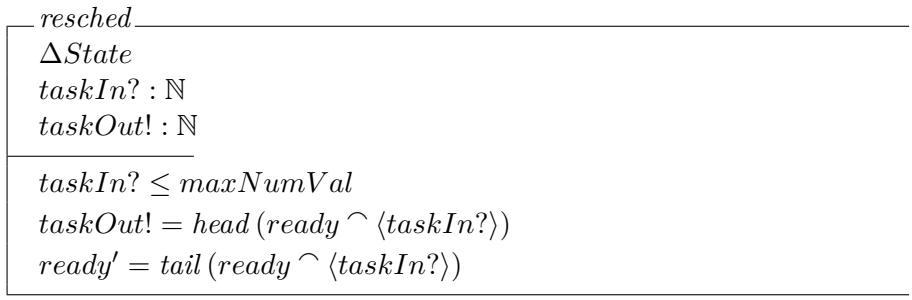


Figure 6.3: A part of a Z model which represents an ADT of type $SchedType_1$.

```
typedef struct
{
   unsigned id;
   // extra fields
} TCB;

TCB* ready[MAXSIZE];
unsigned len;
```

```
TCB * cResched(TCB *in)
{
   TCB *out; unsigned i;
   ready[len] = in;
   out = ready[0];
   for(i = 0; i < len; i++)
   ready[i] = ready[i + 1];
   return out;
}
```

Figure 6.4: A part of a C program which implements an ADT of type $SchedType_2$.

A part of a Z model (say $\mathcal{M}$) which represents an ADT of type $SchedType_1$ is shown in Fig. 6.3. It shows the operation-schema for the *resched* operation. This schema assumes that the data-schema *State* contains a field *ready*, which is a sequence of natural numbers modeling the set of ready tasks. It also assumes that each element in the sequence *ready* is bounded by the value of the variable "$maxNumVal$", which denotes the maximum value allowed by the data type `unsigned` in the C language. A task is represented by a unique task-identifier (natural number).

Now consider the C program of Fig. 6.4. It shows a part of a C program (say $\mathcal{P}$) which implements an ADT of type $SchedType_2$. An array called `ready` represents the set of ready tasks in the system. In this implementation, a task is represented by a pointer to the struct called `TCB`, which contains a field to store the abstract id of a task and extra fields to store more implementation-related details.

The ADTs $\mathcal{M}$ and $\mathcal{P}$ are conformal, since their ADT types are conformal. We note that the operations in these ADTs have different I/O types. For instance, the set of natural numbers is the I/O type for the *resched* operation

$$f_{resched} \subseteq \mathbb{N} \times \{\texttt{TCB *}\},\ \text{such that}\ tcb \in f_{resched}(k)\ \text{iff}\ k = \texttt{tcb->id}\ \text{in}\ M.$$

$$g_{resched} : \{\texttt{TCB *}\} \to \mathbb{N},\ \text{such that}\ g_{resched}(tcb) = k,\ \text{iff}\ \texttt{tcb->id} = k\ \text{in}\ M.$$

Figure 6.5: A pair of relations defining the type translations for the *resched* operation.

in the Z model $\mathcal{M}$, but the I/O type for this operation in the C implementation $\mathcal{P}$ is the set of all pointers to the struct `TCB`. We would like to consider the C method, which implements the *resched* operation as a valid refinement of the operation-schema modeling the same operation. To do this, we need to extend our theory of refinement to allow a refined operation to have different I/O types than that of the corresponding abstract operation.

6.2 Refinement between conformal ADTs

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be conformal ADTs of types $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively. Then we know that $N_1 = N_2$, since $\mathcal{N}_1$ and $\mathcal{N}_2$ are conformal. We define a notion of refinement between the conformal ADTs $\mathcal{A}_1$ and $\mathcal{A}_2$, in terms of a pair of relations: $f_n \subseteq I_n^1 \times I_n^2$ and $g_n : O_n^2 \to O_n^1$, for each operation n in the ADT type $\mathcal{N}_1$.

For example, the relations $f_{resched}$ and $g_{resched}$ for the operation *resched* in the above example could be defined as shown in Fig. 6.5. These relations are assumed to be defined with respect to a particular memory map M which maps memory addresses to TCB pointers.

The pair of relations (f_n, g_n) for each operation n in N_1 is called a *translation pair* for n, since these relations translate an input (or output) from one type to another.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be as above and let $(f_n, g_n)_{n \in N_1}$ be a family of translation pairs for the operations in N_1. Then an initial sequence of operations allowed in $\mathcal{A}_1$, $w_1 = (n_0^1, a_0^1, b_0^1) \cdots (n_k^1, a_k^1, b_k^1)$ is said to be "(f_n, g_n)-equivalent" to an initial sequence of operations, $w_2 = (n_0^2, a_0^2, b_0^2) \cdots (n_l^2, a_l^2, b_l^2)$ allowed in $\mathcal{A}_2$ (denoted $w_1 \overset{(f_n, g_n)}{\equiv} w_2$), iff $|w_1| = |w_2|$; $\forall\, 0 \leq i < |w_1|, n_i^1 = n_i^2, a_i^2 \in f_{n_i}(a_i^1)$ and $g_{n_i}(b_i^2) = b_i^1$.

For example, suppose the translation pair for the operation *resched* is such that $f_{resched}(5) = \{(5, tcb_1), (5, tcb_2)\}$ and $g_{resched}(2, tcb_3) = g_{resched}(2, tcb_4) = 2$. Let σ_1 be an initial sequence of operations allowed in $\mathcal{A}_1$ and let σ_2 be an initial sequence of operations allowed in $\mathcal{A}_2$ such that $\sigma_1 \overset{(f_n, g_n)}{\equiv} \sigma_2$. Now consider the initial sequence of operations, $\gamma_1 = \sigma_1 \cdot (resched, 5, 2)$ allowed in $\mathcal{A}_1$. Then the initial sequence of operations, $\gamma_2 = \sigma_2 \cdot (resched, (5, tcb_2), (2, tcb_3))$ allowed in $\mathcal{A}_2$ is $(f_{resched}, g_{resched})$-equivalent to γ_1 (i.e $\gamma_1 \overset{(f_{resched}, g_{resched})}{\equiv} \gamma_2$).

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be conformal ADTs as above. Then $\mathcal{A}_2$ is said to be a refinement of $\mathcal{A}_1$, iff there exists a family of translation pairs $(f_n, g_n)_{n \in N_1}$, such

```
struct
{
   _(ghost unsigned seq[\natural])
   _(ghost unsigned gLen)
} GS;
```

```
unsigned resched(unsigned t)
{
   unsigned res;
   _(ghost GS.seq[gLen] = t)
   _(assume res == GS.seq[0])
   _(ghost GS.seq =
      \lambda \natural i;
         GS.seq[i + 1]
   return res;
}
```

Figure 6.6: A part of a ghost implementation of an ADT of type $SchedType_1$.

that $w_1 \in L_{init}(\mathcal{A}_1)$ and $w_1 \overset{(f_n,g_n)}{\equiv} w_2$ implies $w_2 \in L_{init}(\mathcal{A}_2)$. Recall that the notation $L_{init}(\mathcal{A})$ denotes the language of initialized sequences of operation calls allowed by an ADT $\mathcal{A}$. We use the notation "$\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$" to denote the refinement defined here. It is not difficult to see that the C program of Fig. 6.4 refines the Z specification of Fig. 6.3.

6.3 Clients with conformal ADTs

We explain in this section a technique, which enables an existing client of an abstract ADT $\mathcal{A}_1$ to interact with a conformal ADT $\mathcal{A}_2$ refining $\mathcal{A}_1$, without modifying its code. Let $\mathcal{P}$ be a client program to an ADT $\mathcal{A}_1$ of type $\mathcal{N}_1$ and $\mathcal{G}$ be a ghost implementation of $\mathcal{A}_1$ in VCC. For example, let $\mathcal{G}$ be the ghost implementation of an ADT of type $SchedType_1$, a part of which is shown in Fig. 6.6. This is in fact a ghost implementation of the ADT of Fig. 6.3. We fix the symbol $\mathcal{G}$ to denote this ghost implementation for the rest of this chapter and we will use this as the abstract ADT $\mathcal{A}_1$ to illustrate refiement.

We now consider a conformal ADT $\mathcal{A}_2$ refining the abstract ADT $\mathcal{A}_1$. For example, consider the C implementation of Fig. 6.4. This program implements an ADT of type $SchedType_2$. We fix the symbol $\mathcal{C}$ to denote this C implementation for the rest of this chapter. Thus the ADTs $\mathcal{G}$ and $\mathcal{C}$ are conformal, since their types are conformal. Let $(f_n, g_n)_{n\in\{init,create,resched\}}$ be the family of translation pairs, a part of which is shown in Fig. 6.5 such that $\mathcal{C} \overset{(f_n,g_n)}{\preceq} \mathcal{G}$.

We now consider how a client program $\mathcal{P}$ of an abstract ADT $\mathcal{A}_1$, can call the operations in a conformal ADT $\mathcal{A}_2$ refining $\mathcal{A}_1$, rather than calling the operations in $\mathcal{A}_1$. For example, let $\mathcal{P}[\mathcal{G}]$ be a client program of $\mathcal{G}$, which calls the operations from $\mathcal{G}$. Can we replace $\mathcal{G}$ in $\mathcal{P}[\mathcal{G}]$ with $\mathcal{C}$, to obtain $\mathcal{P}[\mathcal{C}]$ without modifying the client program $\mathcal{P}$? The answer is "no", since the I/O types in $\mathcal{C}$ are different from that of $\mathcal{G}$.

We can solve the above problem by adding an interface of wrapper functions with the same I/O types as $\mathcal{G}$ to the refined ADT $\mathcal{C}$. A wrapper function for

```
unsigned resched(unsigned id)
{
   TCB *in, *out;
   in = select();
     // this operation selects a TCB pointer from the
     // memory map M such that it points to a TCB with
     // id as its task identifier
   out = cResched(in);
   return out.id;
}
```

Figure 6.7: Wrapper function for the *insert* operation.

each operation n in the ADT type does the following sequence of operations: (i) obtains $f_n(a)$, for the given input a in I_n^1, (ii) calls the refined operation with $f_n(a)$ as input and stores the output to a temporary variable say `out`, and (iii) returns $g_n(\texttt{out})$.

For example, Fig. 6.7 shows an example wrapper function for the *resched* operation in $\mathcal{C}$. The method `select`, which is invoked from the the wrapper is assumed to select a TCB pointer, which points to a TCB instance `tcb` such that in the memory map M, `tcb.id = id`. The memory map M is assumed to represent the set of all TCBs that will be created in the lifetime of the system. This wrapper uses a fixed pair of functions which represents the translation pair of Fig. 6.5.

In general, if $\mathcal{C} \overset{(f_n,g_n)}{\preceq} \mathcal{G}$ and if $\mathcal{W}(\mathcal{C})$ is obtained from $\mathcal{C}$ by adding an interface of wrapper functions as above, then we can just replace the ADT $\mathcal{G}$ in $\mathcal{P}[\mathcal{G}]$ with $\mathcal{W}(\mathcal{C})$ to obtain $\mathcal{P}[\mathcal{W}(\mathcal{C})]$.

6.4 Verification guarantee

We now describe the verification guarantees given by our notion of refinement for conformal ADTs. In fact we extend the verification guarantees for ADTs presented in Sec. 2.5, to conformal ADTs as well.

Let $\mathcal{A}_1 = (Q_1, U_1, E_1, \{op_n^1\}_{n\in N_1})$ and $\mathcal{A}_2 = (Q_2, U_2, E_2, \{op_n^2\}_{n\in N_2})$ be conformal ADTs of types $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively. Let $(f_n, g_n)_{n\in N_1}$ be a family of translation pairs such that $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$. We fix the ADTs $\mathcal{A}_1$ and $\mathcal{A}_2$ as above for the rest of this section. We also fix the notation "$\mathcal{W}(\mathcal{A}_2)$" to denote the concrete ADT $\mathcal{A}_2$ with a wrapper $\mathcal{W}$ as discussed in Sec. 6.3.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be as above and let $\mathcal{T} = (R, \Sigma_l \cup \Sigma_{N_1}, s, E, \Delta)$ be an $\mathcal{N}_1$-client transition system. There is a natural relation σ between the states of $\mathcal{A}_2$ and $\mathcal{A}_1$ such that $(q_2, q_1) \in \sigma$ iff there exist exception-free initial sequences of operations w_1 and w_2 such that $w_1 \overset{(f_n,g_n)}{\equiv} w_2$, $U_1 \xrightarrow{w_1} q_1$ in $\mathcal{A}_1$ and $U_2 \xrightarrow{w_2} q_2$ in $\mathcal{A}_2$. We can use the relation σ to define a kind of bisimulation relation σ'

between $\mathcal{T}[\mathcal{A}_1]$ and $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$: a set of states (s, X_2) of $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ and a state (r, q_1) of $\mathcal{T}[\mathcal{A}_1]$ are related by σ' iff $s = r$ and $\sigma(q_2, q_1)$ holds for each $q_2 \in X_2$. We note that for an initial sequence of operations w_1 allowed in $\mathcal{A}_1$, there can be a set of initial sequences W_2 allowed in $\mathcal{A}_2$ such that $w_2 \overset{(f_n,g_n)}{\equiv} w_1$ for each w_2 in W_2. It follows from the definition of σ' that, when two states are related by σ', the local states of the client program $\mathcal{T}$ in them are the same.

Let R be a binary relation. Then we use the notations $(a, b) \in R$ or the notation $b \in R(a)$ to denote that a is related to b under the relation R.

Theorem 6.1. *Let $\mathcal{A}_1$ $\mathcal{A}_2$, $\mathcal{T}$, $\mathcal{W}(\mathcal{A}_2)$ and σ' be as above. Then σ' is a bisimulation in the following sense:*

1. *if $((r, p_2), (r, p_1)) \in \sigma'$, and $(r, p_1) \xrightarrow{l} (s, q_1)$ in $\mathcal{T}[\mathcal{A}_1]$ with l a non-exception action label, then there exists a non-empty set of states (s, X_2) in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ such that $(r, p_2) \xrightarrow{l} (s, q_2)$ and $((s, q_2), (s, q_1)) \in \sigma'$ for each $q_2 \in X_2$.*

2. *Conversely, if $((r, p_2), (r, p_1)) \in \sigma'$, and for a non-empty set of states X_2 in Q_2, $(r, p_2) \xrightarrow{l} (s, q_2)$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ for each $q_2 \in X_2$. Then either there exists a state (s, q_1) in $\mathcal{T}[\mathcal{A}_1]$ such that $(r, p_1) \xrightarrow{l} (s, q_1)$ and $((s, q_2), (s, q_1)) \in \sigma'$ for each $q_2 \in X_2$, or l is of the form (n, a_1, b_1) and $(r, p_1) \overset{(n,a_1,e)}{\rightarrow} (-, E_1)$ in $\mathcal{T}[\mathcal{A}_1]$.*

Proof. We give a proof similar to the proof given for Theorem 2.1 of Sec. 2.5.

1. Suppose $((r, p_2), (r, p_1)) \in \sigma'$, and $(r, p_1) \xrightarrow{l} (s, q_1)$ in $\mathcal{T}[\mathcal{A}_1]$ with l a non-exception action label. There are two possibilities to consider here:

   (a) l is a local action label in $\mathcal{T}$. This means that $r \xrightarrow{l} s$ in $\mathcal{T}$, and then it follows from the definition of $\mathcal{T}[\mathcal{A}_1]$ that $p_1 = q_1$. Now it follows from the definition of $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ that $(r, p_2) \xrightarrow{l} (s, p_2)$. Also we have $((s, p_2), (s, p_1)) \in \sigma'$, since $(p_2, p_1) \in \sigma$ by assumption. Hence we are done.

   (b) l is of the form (n, a_1, b_1). Thus we have $(r, p_1) \overset{(n,a_1,b_1)}{\rightarrow} (s, q_1)$ in $\mathcal{T}[\mathcal{A}_1]$. Then by definition of $\mathcal{T}[\mathcal{A}_1]$, we have $r \overset{(n,a_1,b_1)}{\rightarrow} s$ in $\mathcal{T}$ and $p_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$. It follows from the definition of σ that there exist exception-free initial sequences of operations w_1 and w_2 such that $w_2 \overset{(f_n,g_n)}{\equiv} w_1$, $U_1 \xrightarrow{w_1} p_1$ in $\mathcal{A}_1$ and $U_2 \xrightarrow{w_2} p_2$ in $\mathcal{A}_2$, since we have $(p_2, p_1) \in \sigma$. Thus we have $U_1 \overset{w_1 \cdot (n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$. This implies that for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, there exists a q_2 in Q_2 such that $U_2 \overset{w_2 \cdot (n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$, since $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$ by assumption. Therefore it follows that for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, $p_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$, since $\mathcal{A}_2$ is deterministic and $U_2 \xrightarrow{w_2} p_2$ in $\mathcal{A}_2$. Therefore it follows

from the definition of $\mathcal{W}(\mathcal{A}_2)$ that there exists a non-empty set of states X_2 in Q_2 with $p_2 \overset{(n,a_1,b_1)}{\rightarrow} q_2$ in $\mathcal{W}(\mathcal{A}_2)$ for each q_2 in X_2. Now it follows from the definition of $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ that, there exists a non-empty set of states X_2 in Q_2 with $(r, p_2) \overset{(n,a_1,b_1)}{\rightarrow} (s, q_2)$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ for each q_2 in X_2, since we have $r \overset{(n,a_1,b_1)}{\rightarrow} s$ in $\mathcal{T}$. Also it follows from the definition of σ that $(q_2, q_1) \in \sigma$ for each q_2 in X_2, since we have $U_1 \overset{w_1\cdot(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$ and $U_2 \overset{w_2\cdot(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ with $w_1 \cdot (n, a_1, b_1) \overset{(f_n,g_n)}{\equiv} w_2 \cdot (n, a_2, b_2)$. This implies that $((s, q_2), (s, q_1)) \in \sigma'$ for each q_2 in X_2 and hence we are done.

2. Conversely, suppose $((r, p_2), (r, p_1)) \in \sigma'$, and there exists a non-empty set of states X_2 in Q_2 such that $(r, p_2) \overset{l}{\rightarrow} (s, q_2)$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ for each q_2 in X_2. Here also we need to consider two cases:

   (a) l is a local action label in $\mathcal{T}$. This implies that $r \overset{l}{\rightarrow} s$ in $\mathcal{T}$ and then it follows from the definition of $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ that $p_2 = q_2$ for each q_2 in X_2. Now it follows from the definition of $\mathcal{T}[\mathcal{A}_1]$ that $(r, p_1) \overset{l}{\rightarrow} (s, p_1)$. Also we have $((s, p_2), (s, p_1)) \in \sigma'$ for each q_2 in X_2, since $p_2 = q_2$ for each q_2 in X_2, and $(p_2, p_1) \in \sigma$ by assumption. Hence we are done.

   (b) l is of the form (n, a_1, b_1). Thus we have $(r, p_2) \overset{(n,a_1,b_1)}{\rightarrow} (s, q_2)$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ for each q_2 in X_2. Then by definition of $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$, we have $r \overset{(n,a_1,b_1)}{\rightarrow} s$ in $\mathcal{T}$ and $p_2 \overset{(n,a_1,b_1)}{\rightarrow} q_2$ in $\mathcal{W}(\mathcal{A}_2)$ for each q_2 in X_2. Therefore there exists an input $a_2 \in f_n(a_1)$ and there exists an output b_2 with $g_n(b_2) = b_1$ such that $p_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ for each q_2 in X_2. It follows from the definition of σ that there exist exception-free initial sequences of operations w_1 and w_2 such that $w_1 \overset{(f_n,g_n)}{\equiv} w_2$, $U_1 \overset{w_1}{\rightarrow} p_1$ in $\mathcal{A}_1$ and $U_2 \overset{w_2}{\rightarrow} p_2$ in $\mathcal{A}_2$, since we have $(p_2, p_1) \in \sigma$. Therefore there exists an input $a_2 \in f_n(a_1)$ and there exists an output b_2 with $g_n(b_2) = b_1$ such that $U_2 \overset{w_2\cdot(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ for each q_2 in X_2. This implies that for an input a_1 with $a_2 \in f_n(a_1)$: either there exists a q_1 in Q_1 and $b_1 = g_n(b_2)$ such that $U_1 \overset{w_1\cdot(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$, or $U_1 \overset{w_1\cdot(n,a_1,e)}{\rightarrow} E_1$ in $\mathcal{A}_1$, since $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$ by assumption. The latter implies that $p_1 \overset{(n,a_1,e)}{\rightarrow} E_1$ in $\mathcal{A}_1$ since $\mathcal{A}_1$ is deterministic and we have $U_1 \overset{w_1}{\rightarrow} p_1$ in $\mathcal{A}_1$, and hence the result immediately follows from the definition of $\mathcal{T}[\mathcal{A}_1]$. The former implies that $p_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$, since $\mathcal{A}_1$ is deterministic and we have $U_1 \overset{w_1}{\rightarrow} p_1$ in $\mathcal{A}_1$, and then it follows from the definition of $\mathcal{T}[\mathcal{A}_1]$ that $(r, p_1) \overset{(n,a_1,b_1)}{\rightarrow} (s, q_1)$ in $\mathcal{T}[\mathcal{A}_1]$, since we have $r \overset{(n,a_1,b_1)}{\rightarrow} s$ in $\mathcal{T}$. Also it follows from the definition of σ that $(q_2, q_1) \in \sigma$ for each q_2 in X_2, since we have $U_1 \overset{w_1\cdot(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$ and $U_2 \overset{w_2\cdot(n,a_2,b_2)}{\rightarrow} q_2$

in $\mathcal{A}_2$ with $w_1 \cdot (n, a_1, b_1) \overset{(f_n, g_n)}{\equiv} w_2 \cdot (n, a_2, b_2)$. Therefore we have $((s, q_2), (s, q_1)) \in \sigma'$ for each q_2 in X_2. Hence we are done

□

Let $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{T}$, $\mathcal{W}(\mathcal{A}_2)$ and σ' be as above. Then a path p_2 of the form $p_2 = v_0^2 \overset{a_1^2}{\to} v_1^2 \cdots \overset{a_m^2}{\to} v_m^2$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ is said to be σ'-*equivalent* to a path p_1 of the form $p_1 = v_0^1 \overset{a_1^1}{\to} v_1^1 \cdots \overset{a_n^1}{\to} v_n^1$ in $\mathcal{T}[\mathcal{A}_1]$, written "$p_2 \overset{\sigma'}{\equiv} p_1$", iff $word(p_2) = word(p_1)$ (see Sec. 2.1) and $(v_i', v_i) \in \sigma'$. We say that two traces: $t_2 = \langle v_0^2, v_1^2, \cdots, v_m^2 \rangle$ and $t_1 = \langle v_0^1, v_1^1, \cdots, v_n^1 \rangle$ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ and $\mathcal{T}[\mathcal{A}_1]$ respectively are σ'-equivalent, written "$t_2 \overset{\sigma'}{\equiv} t_1$", iff $m = n$ and $(v_i', v_i) \in \sigma'$.

Corollary 6.1. *Let $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{T}$ and σ' be as above. Then Theorem 6.1 implies the following:*

1. *For every* path p_1 *in* $\mathcal{T}[\mathcal{A}_1]$*, there exists a* path p_2 *in* $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ *such that* $p_2 \overset{\sigma'}{\equiv} p_1$.
2. *For every* path p_2 *in* $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$*, either there exists a* path p_1 *in* $\mathcal{T}[\mathcal{A}_1]$ *such that* $p_2 \overset{\sigma'}{\equiv} p_1$ *or there exists a* path p_1^{pref} *in* $\mathcal{T}[\mathcal{A}_1]$ *such that* p_2 *is of the form* $p_2^{pref} \overset{(n,a,b)}{\to} p_2^{suf}$ *with* $p_2^{pref} \overset{\sigma'}{\equiv} p_1^{pref}$ *and* $p_1^{pref} \overset{(n,a,e)}{\to} E \in \mathcal{T}[\mathcal{A}_1]$.

A *path* in the above conditions is assumed to not contain an exception (a transition label of the form $(-, -, e)$). The condition 1 above follows from condition 1 of Theorem 6.1 and the condition 2 follows from condition 2 of Theorem 6.1, since the start states of $\mathcal{T}[\mathcal{A}_1]$ and $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ are related by σ'.

We now state and prove a theorem below, similar to Theorem 2.2 of Sec. 2.5, which shows the properties preserved by our notion of refinement for conformal ADTs. We use the definitions of *LT properties*, *locally-equivalent traces* ($\overset{l}{\equiv}$) and *local LT properties* of Sec. 2.5, with the modification that we use the bisimulation relation σ' defined in the beginning of this section instead of the definition (of σ') given in Sec. 2.5.

Theorem 6.2. *Let $\mathcal{A}_1$ $\mathcal{A}_2$, $\mathcal{T}$ and $\mathcal{W}(\mathcal{A}_2)$ be as above. Let φ be a local LT property over the vocabulary $(R \times (Q_1 \cup Q_2))$. Then if $\mathcal{T}[\mathcal{A}_1]$ satisfy φ, either $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ will also satisfy φ or each trace violating φ in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ contains a prefix with a locally-equivalent trace leading to the exception state in $\mathcal{T}[\mathcal{A}_1]$. In particular, if the client $\mathcal{T}$ does not see an exception with the abstract ADT $\mathcal{A}_1$, then both $\mathcal{T}[\mathcal{A}_1]$ and $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ satisfy exactly the same set of local LT properties.*

Proof. Suppose t_2 is a trace in $\mathcal{T}[\mathcal{W}(\mathcal{A}_2)]$ violating φ. It follows from Corollary 6.1 that one of the following conditions is true:

1. there exists a trace t_1 in $\mathcal{T}[\mathcal{A}_1]$ such that $t_2 \overset{l}{\equiv} t_1$.
2. there exists a trace t_1^{pref} in $\mathcal{T}[\mathcal{A}_1]$ such that t_2 is of the form $t_2^{pref} \cdot t_2^{suf}$ with $t_2^{pref} \overset{l}{\equiv} t_1^{pref}$ and $t_1^{pref} \cdot \langle E \rangle \in \mathcal{T}[\mathcal{A}_1]$.

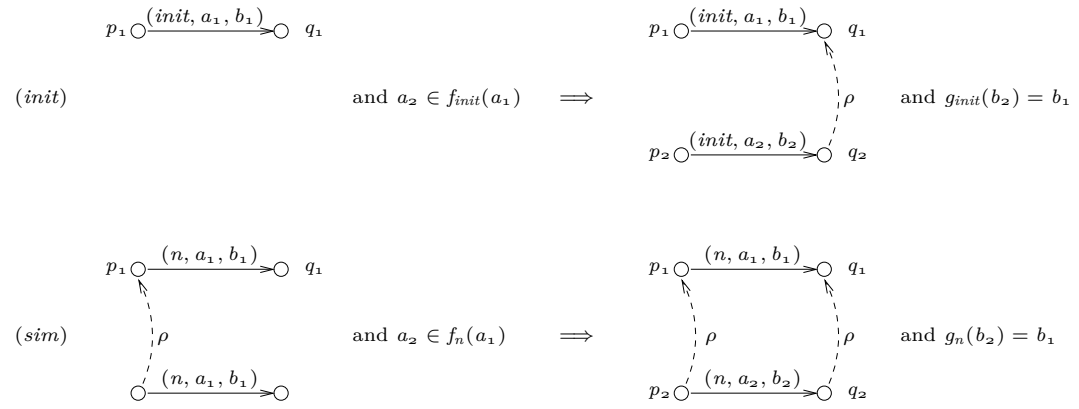


Figure 6.8: Illustrating the equivalent condition (CRC) for refinement.

Now condition 1 above cannot be true since it contradicts the assumption that $\mathcal{T}[\mathcal{A}_1]$ satisfies φ. Hence the condition 2 must be true and which gives a prefix for t_2 as required in the theorem and hence we are done. □

Thus our notion of refinement for conformal ADTs preserves exactly the same set of properties preserved by our notion of refinement for ADTs.

6.5 Equivalent refinement condition

Let $\mathcal{A}_1 = (Q_1, U_1, E_1, \{op_n^1\}_{n \in N_1})$ and $\mathcal{A}_2 = (Q_2, U_2, E_2, \{op_n^2\}_{n \in N_2})$ be conformal ADTs of types $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively. We formulate an *equivalent* condition for $\mathcal{A}_2$ to refine $\mathcal{A}_1$, based on: (i) an abstraction relation that relates states of $\mathcal{A}_2$ to states of $\mathcal{A}_1$, and (ii) a family of translation pairs $(f_n, g_n)_{n \in N_1}$. We say $\mathcal{A}_1$ and $\mathcal{A}_2$ satisfy condition (CRC) if there exist, a relation $\rho \subseteq Q_2 \times Q_1$ and a family of translation pairs $(f_n, g_n)_{n \in N_1}$ such that:

(init) For each $a_1 \in I^1_{init}$, if the *init* operation in $\mathcal{A}_1$ transitions to a state q_1 with b_1 as output, then for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, there exists a q_2 in Q_2 such that the *init* operation in $\mathcal{A}_2$ transitions to q_2 with output b_2 and $(q_2, q_1) \in \rho$.

(sim) For each $n \in N_1$, $a_1 \in I^1_n$, $b_1 \in O^1_n$, $p_1 \in Q_1$ and $p_2 \in Q_2$, with $(p_2, p_1) \in \rho$, whenever $p_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$ with $b_1 \neq e$, then for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, there exists a q_2 in Q_2 such that $p_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ with $(q_2, q_1) \in \rho$.

Fig. 6.8 illustrates the equivalent refinement condition (CRC) between conformal ADTs.

Theorem 6.3. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be as above. Then $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$, iff they satisfy condition (CRC).*

Proof. We give a proof similar to the proof given for Theorem 2.3 of Sec. 2.6.

($\Leftarrow$) Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be as above. Let $(f_n, g_n)_{n\in N_1}$ be a family of translation pairs and let $\rho \subseteq Q_2 \times Q_1$ be an abstraction relation, such that $\mathcal{A}_1$ and $\mathcal{A}_2$ satisfy condition (CRC) with respect to ρ and $(f_n, g_n)_{n\in N_1}$.

We first prove the following claim.

Claim 6.1. *For any states $p_1, q_1 \in Q_1$, $p_2 \in Q_2$ and an exception-free initial sequence of operations w_1 of the form $w_1 = (init, a_1, b_1) \cdot u_1$, if $p_1 \xrightarrow{w_1} q_1$ in $\mathcal{A}_1$, then for each w_2 with $w_2 \overset{(f_n,g_n)}{\equiv} w_1$, there exists a state q_2 in Q_2 such that $p_2 \xrightarrow{w_2} q_2$ in $\mathcal{A}_2$ and $(q_2, q_1) \in \rho$.*

Proof. We prove this claim by induction on length of u_1.

(Base) Let $|u_1| = 0$. Then $w_1 = (init, a_1, b_1)$, where $a_1 \in I^1_{init}$ and $b_1 \in O^1_{init}$. Then it follows from (*init*) of condition (CRC) that for each $a_2 \in f_{init}(a_1)$ and for each b_2 with $g_{init}(b_2) = b_1$, there exists a q_2 in Q_2 such that $p_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ and $(q_2, q_1) \in \rho$. Hence we are done.

(Step) Let $|u_1| = k+1$ and $w_1 = (init, a, b)\cdot v_1 \cdot (n, a_1, b_1)$, where $|v_1| = k$, $n \in N_1$, $a_1 \in I^1_n$, $b_1 \in O^1_n$; and let $p_1 \xrightarrow{(init,a,b)\cdot v_1} r_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ be the path corresponding to w_1 in $\mathcal{A}_1$. It follows from the induction hypothesis that for each $(init, a', b') \cdot v_2$ with $(init, a', b') \cdot v_2 \overset{(f_n,g_n)}{\equiv} (init, a, b) \cdot v_1$, there exists a state $r_2 \in Q_2$ such that $p_2 \xrightarrow{(init,a',b')\cdot v_2} r_2$ in $\mathcal{A}_2$ and $(r_2, r_1) \in \rho$. Now it follows from (*sim*) of condition (CRC) that for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, there exists a q_2 in Q_2 such that $r_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$ and $(q_2, q_1) \in \rho$. Therefore for each w_2 with $w_2 \overset{(f_n,g_n)}{\equiv} w_1$, $p_2 \xrightarrow{w_2} q_2$ and $(q_2, q_1) \in \rho$ and hence we are done.

□

Now it follows from Claim 6.1 that whenever $w_1 \in L_{init}(\mathcal{A}_1)$ for any exception-free initial sequence of operations w_1, we also have $w_2 \in L_{init}(\mathcal{A}_2)$, for each w_2 with $w_2 \overset{(f_n,g_n)}{\equiv} w_1$. This proves that $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$.

($\Rightarrow$) Conversely suppose $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$. Let ρ be the relation $\sigma \subseteq Q_2 \times Q_2$ defined for the verification guarantee of Sec. 6.4. Recall that $(q_2, q_1) \in \rho$, iff there exist exception-free initial sequences of operation calls w_1 and w_2 such that $w_2 \overset{(f_n,g_n)}{\equiv} w_1$, $U_1 \xrightarrow{w_1} q_1$ in $\mathcal{A}_1$ and $U_2 \xrightarrow{w_2} q_2$ in $\mathcal{A}_2$.

To show that ρ satisfies (CRC-init), suppose $p_1 \overset{(init,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$. Then since $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$, we must have $p_2 \overset{(init,a_2,b_2)}{\rightarrow} q_2$ for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, for a $q_2 \in Q_2$. Also, by definition of ρ,

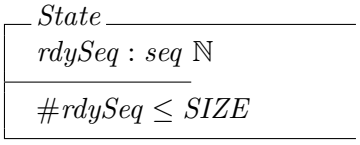


Figure 6.9: The data-schema in the ADT `z-queue`.

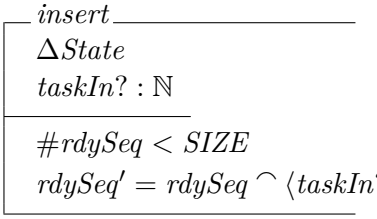


Figure 6.10: The operation-schema for the *insert* operation in the ADT `z-queue`.

we have $(q_2, q_1) \in \rho$. Hence ρ satisfies (CRC-init). Recall that the *init* operation is independent of the state on which it is invoked.

We now show that ρ satisfies the condition (CRC-sim). Suppose $(p_2, p_1) \in \rho$, and $p_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$ with $b_1 \neq e$. By definition of ρ, we know that there exist exception-free initial sequences of operations w_1 and w_2 such that $U_1 \xrightarrow{w_1} p_1$ in $\mathcal{A}_1$, $U_2 \xrightarrow{w_2} p_2$ in $\mathcal{A}_2$ and $w_2 \overset{(f_n,g_n)}{\equiv} w_1$. Since $p_1 \overset{(n,a_1,b_1)}{\rightarrow} q_1$ in $\mathcal{A}_1$ by assumption, we have $U_1 \xrightarrow{w_1 \cdot (n,a_1,b_1)} q_1$ in $\mathcal{A}_1$. But since $\mathcal{A}_2 \overset{(f_n,g_n)}{\preceq} \mathcal{A}_1$, we know that for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, $U_2 \xrightarrow{w_2 \cdot (n,a_2,b_2)} q_2$ in $\mathcal{A}_2$ for a $q_2 \in Q_2$. Hence it follows that $p_2 \overset{(n,a_2,b_2)}{\rightarrow} q_2$ in $\mathcal{A}_2$, for each $a_2 \in f_n(a_1)$ and for each b_2 with $g_n(b_2) = b_1$, since $\mathcal{A}_2$ is deterministic and we have $U_2 \xrightarrow{w_2} p_2$ in $\mathcal{A}_2$. Also by definition of ρ we have $(q_2, q_1) \in \rho$. Hence ρ satisfies condition (CRC-sim).

□

6.6 Phrasing and verifying refinement conditions

In this section we describe techniques for phrasing the refinement condition (CRC) between conformal ADTs. In fact we extend the techniques proposed for ADTs (in Sec. 5.2) to conformal ADTs. We also show how one can use the tool VCC, to check these refinement conditions.

The conditions formulated in Sec. 5.2, for ADTs can be easily extended to conformal ADTs. The extension required is the technique to handle different I/O types for the operations in the abstract and concrete ADTs. We do this by using the family of translation pairs from the necessary condition (CRC). We explain our technique for doing this for handling the refinement between a Z model and a C model. Similar techniques could be used for other cases.

We use a simple example to illustrate our technique for checking refinement between conformal ADTs in the tool VCC. Consider the ADT type $QType_{\mathbb{Z}}$

```
typedef struct
{
   unsigned id;
   struct TCB *tcb;
} TaskType;

TaskType ready[MAXSIZE];
unsigned len;
```

```
void insert(TaskType tt)
{
   ready[len] = tt;
   len++;
   return;
}
```

Figure 6.11: A part of a C program, which implements an ADT of type $QType^2_{\mathbb{Z}}$.

Notation	Meaning
$\mathtt{Arg}^2_n$	*Type of an argument (or input) to a concrete operation n*
x_1	*Argument (or input) to an abstract operation*
x_2	*Argument (or input) to a concrete operation*
$pre^{\mathcal{M}}_n$	*Precondition of an abstract operation n*
$BAP^{\mathcal{M}}_n$	*Before-After Predicate of an abstract operation n*
inv_{s_1}	*State-invariant in an abstract state* s_1
inv_ρ	*Gluing invariant between an abstract state and a concrete state*
p_1	*Before-state of an abstract operation*
q_1	*After-state of an abstract operation*
p_2	*Before-state of a concrete operation*
q_2	*After-state of a concrete operation*
$q_1.y$	*Output from an abstract operation*
$q_2.y$	*Output from a concrete operation*

Table 6.1: Notations used in Fig. 6.12.

discussed in Sec. 3.1. Fig. 6.9 shows the data-schema and Fig. 6.10 shows the operation-schema in a Z model called `z-queue`, which models an ADT of type $QType_{\mathbb{Z}}$. This ADT is assumed to model a FIFO queue of ready tasks in an operating system. Thus a task is represented by a natural number (task-ID).

Now consider a C program `c-queue`, a part of which is shown in Fig. 6.11. This program implements an ADT of type $QType^2_{\mathbb{Z}}$, which is similar to the ADT type $QType_{\mathbb{Z}}$, except that a task is represented by an instance of the struct called `TaskType`, rather than a natural number. We note that a task is an input or output for most of the operations in these ADT types. The ADT types $QType_{\mathbb{Z}}$ and $QType^2_{\mathbb{Z}}$ are conformal and hence so are the ADTs `z-queue` and `c-queue`.

Phrasing refinement between Z and C models

Now we explain a technique for phrasing the condition (CRC) between a Z model and a C model. This is in fact an extension of the technique explained

(init-a) $\texttt{func}_{init}$ must terminate on all inputs $x_2 \in f_{init}(x_1)$ for which $init^{\mathcal{M}}(x_1)$ is defined (i.e. $pre^{\mathcal{M}}_{init}(x_1)$ is true).

(init-b)
$\texttt{func}_{init}(\texttt{Arg}^2_{init}\ \texttt{x}_2)$
$\_(\texttt{requires}\ \exists\, x_1 \mid x_2 \in f_{init}(x_1) \wedge pre^{\mathcal{M}}_{init}(x_1))$
$\_(\texttt{ensures}\ \exists\, x_1, q_1 \mid x_2 \in f_{init}(x_1) \wedge BAP^{\mathcal{M}}_{init}(x_1, q_1, q_1.y) \wedge inv_{q_1} \wedge inv_\rho(q_2, q_1) \wedge q_1.y = g_{init}(q_2.y))$
{
// function body
}

(sim-a) For each operation n, $\texttt{func}_n$ must terminate on all state-input pairs (p_2, x_2) such that there exists a state-input pair (p_1, x_1) of $\mathcal{M}$ satisfying: $x_2 \in f_n(x_1) \wedge inv_{p_1} \wedge pre^{\mathcal{M}}_n(p_1, x_1) \wedge inv_\rho(p_2, p_1)$.

(sim-b) For each operation n:

$\texttt{func}_n(\texttt{Arg}^2_{init}\ \texttt{x}_2)$
$\_(\texttt{requires}\ \exists\, x_1, p_1 \mid x_2 \in f_n(x_1) \wedge inv_{p_1} \wedge pre^{\mathcal{M}}_n(p_1, x_1) \wedge inv_\rho(p_2, p_1))$
$\_(\texttt{ensures}\ \exists\, x_1, p_1, q_1 \mid x_2 \in f_n(x_1) \wedge inv_\rho(p_2, p_1) \wedge BAP^{\mathcal{M}}_n(p_1, x_1, q_1, q_1.y) \wedge inv_{q_1} \wedge inv_\rho(q_2, q_1) \wedge (q_1.y = g_n(q_2.y)))$
{
// function body
}

Figure 6.12: Phrasing the refinement condition (CRC) by directly importing the requirements from the Z model as function contracts in VCC.

in Sec. 5.2.2.

Fig. 6.12 shows how one can directly import the requirements corresponding to the condition (CRC), from a Z model as function contracts in VCC. Table 6.1 shows different notation used in Fig. 6.12. This technique is similar to the technique explained in Sec. 5.2.2, except that here we use the family $(f_n, g_n)_{n \in N_1}$ of translation pairs from the necessary condition (CRC) to translate the input or output between the abstract and concrete ADTs.

Checking the refinement condition (CRC) in VCC.

We now explain two techniques in VCC to check the refinement condition formulated above. These techniques are similar to the techniques presented in Sec. 5.3.1, except that here we need to deal with different I/O types. The first technique uses the ghost language of VCC to specify the function contract formulated in Fig. 6.12. However in this technique, function contracts make use of existential quantifications. The second technique avoids the use of existential quantifications by manually translating a formula which represents a function contract to a logically equivalent formula without existential quantifications.

To illustrate the first technique, consider the ADTs `z-queue` and `c-queue`

```
void insert( TaskType tt )
  _(requires \exists \natural taskIn?,
    \exists \natural readySeq1[\natural], \natural seqLen1;
     (taskIn? == tt.id) && (seqLen1 <= SIZE)  &&
     (seqLen1 < SIZE) && ((len == seqLen1) &&
     (\forall \natural i; (i < seqLen1) ==>
                          (readySeq1[i] == ready[i].id))))

  _(ensures \exists \natural taskIn?,
    \exists \natural readySeq1[\natural], \natural seqLen1;
    \exists \natural readySeq2[\natural], \natural seqLen2;
    (taskIn? == tt.id)  && ((\old(len) == seqLen1) &&
    (\forall \natural i; (i < seqLen1) ==>
                         (readySeq1[i] == \old(ready[i].id))) &&
    ((seqLen2 == seqLen1 + 1) &&
     (\forall \natural i; (i < seqLen1) ==>
                          (readySeq2[i] == readySeq1[i]) &&
     (readySeq2[seqLen1] == taskIn?)) &&
    (seqLen2 <= SIZE) && ((len == seqLen2) &&
    (\forall \natural i; (i < seqLen2) ==>
                         (readySeq2[i] == ready[i].id)))
{
   //body of insert method
}
```

Figure 6.13: Illustrating a technique in VCC to check the refinement condition (CRC) shown in Fig. 6.12.

```
void insert( TaskType tt )
  _(requires len < SIZE)

  _(ensures (len == (\old(len) + 1)) &&
    (\forall \natural i; (i < \old(len) ==>
                          (ready[i].id == \old(ready[i].id)) &&
    (ready[\old(len)].id == tt.id))
{
   //body of insert method
}
```

Figure 6.14: Illustrating a technique in VCC to check the refinement condition (CRC) shown in Fig. 6.12 by manually transforming the requirements from the Z model.

discussed in the beginning of this section. Fig. 6.13 shows the function contract in VCC, which is generated based on the conditions shown in Fig. 6.12. Here the requirement from the Z model of Fig. 6.10 is directly imported as function contract on the `insert` method of Fig. 6.11. The problem with this technique is that it uses existential quantifications which are difficult to handle for the underlying theorem prover in VCC.

We now extend the "direct-import approach with quantifier elimination" of Sec. 5.3.1, to conformal ADTs. We illustrate this technique with the running example in this section. Fig. 6.14 shows how one can manually transform the requirements from the Z model of Fig. 6.10 as function contract on the `insert` method of Fig. 6.11.

Chapter 7

FreeRTOS Case-Study

This chapter presents a case-study in which we apply our verification methodology to reason about the functional correctness of the scheduler-related functionality of a real-time operating system called FreeRTOS. During the process of verifying FreeRTOS we found a number of subtle bugs. These bugs were fixed and the verification was completed.

7.1 About FreeRTOS

FreeRTOS [38] is a real-time kernel meant for use in embedded applications that run on microcontrollers with small to mid-sized memory. FreeRTOS has a large community of users. There are more than 100,000 downloads from SourceForge each year, putting it in the top 100 most-downloaded SourceForge codes.

FreeRTOS allows an application running on top of it to organize 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 her code and invoke as method calls. These APIs provide the programmer ways to create tasks, schedule tasks based on priority-based preemption, communicate between tasks (via message queues, semaphores, etc), and carry out time-constrained blocking of tasks.

FreeRTOS is architected in a modular fashion. The implementation of the FreeRTOS kernel comprises the following: (i) a port-independent layer written in C which implements the part of the kernel common to all hardware ports (or architectures) supported by FreeRTOS and (ii) a port-specific layer which implements the part of the kernel specific to a hardware architecture. Fig. 7.1 shows the layered architecture of a FreeRTOS application. The arrows represent interactions between different layers.

The port-independent layer of FreeRTOS comprises about 2,500 lines of C code. The functionality provided in this layer is implemented in 3 C files: `task.c`, `queue.c`, and `list.c`. The scheduler-related operations in the API like operations for creating and maintaining tasks, are implemented in the C file `task.c`. FreeRTOS provides a set of operations in its API for interprocess

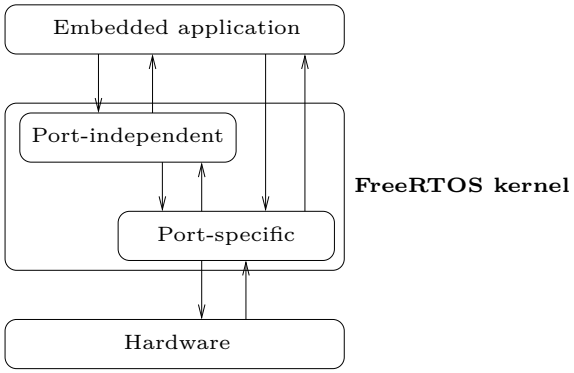


Figure 7.1: Layered architecture of a FreeRTOS application.

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

```
void foo(void* params)
{
   for(;;) {}
}
void bar(void* params)
{
   for(;;) { vTaskDelay(2); }
}
```

Figure 7.2: An example FreeRTOS application.

communication and synchronization, and such operations are implemented in the C file `queue.c`. The C file `list.c`, implements the operations for creating and maintaining task lists used in the FreeRTOS kernel.

The port-specific layer is present in a separate sub-directory. This sub-directory contains implementations of the port-specific functionality for each hardware port (processor/compiler pair) supported by the FreeRTOS kernel. The port-specific functionality includes: memory management, interrupt management and operations to store (and load) a process context to (and from) a hardware architecture. These operations are implemented using the C language, and the assembly language of the respective hardware architecture.

7.2 How FreeRTOS works

A simple application that uses FreeRTOS is shown in Fig. 7.2. The application creates two tasks "`A1`" and "`B2`" with priorities 1 and 2 respectively (a higher number indicates a higher priority). A task is like a "thread" in an operating system. After creating the above tasks, the application starts the FreeRTOS scheduler by invoking the required API: `vTaskStartScheduler`. We use a naming convention that indicates the task's priority in its name. The task `A1`'s code is the function "foo" and the task `B2`'s code is the function

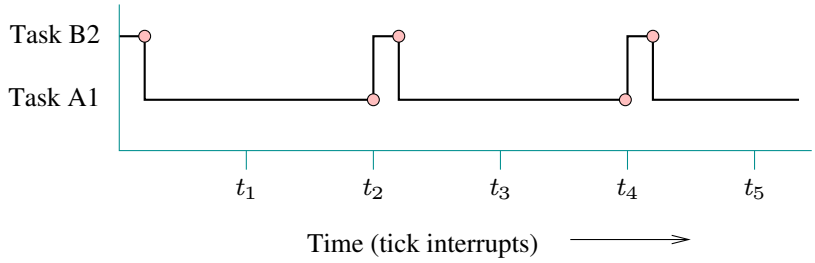


Figure 7.3: A timing diagram of the FreeRTOS application of Fig. 7.2.

"bar". After performing some initialization work like creation of the "idle" task and initialization of the system clock, the scheduler runs task `B2`, which is the highest priority task ready to run. The task `B2` requests to delay it for 2 time units by invoking the required API: `vTaskDelay`. `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 behavior continues forever. A timing diagram showing the execution of this application is shown in Fig. 7.3.

We now take a look under the hood to get a closer look at what exactly happens when our example application executes. Fig. 7.4 shows (on the left) the main checkpoints in the execution of the application, and (on the right) the layout of code and data on the processor.

To begin with, the application code (i.e. the code for `main`, `foo`, and `bar`) is compiled along with the FreeRTOS code (for the scheduler, and the API calls including `xTaskCreate`), and loaded into memory as shown in the figure. The scheduler code is loaded into the Interrupt Service Routine (ISR) code area so that it services a S/W Interrupt (SWI). This is done by a direction to the compiler, contained in the FreeRTOS code.

Execution begins with first instruction in `main` which happens to be a call to the `xTaskCreate` API. This code, which is provided by FreeRTOS, allocates space in the heap for a stack (of size say 1000 bytes) for the task, as well as space to store its "Task Control Block" or TCB. The TCB contains all vital information about the task: where its code (`foo` in this case) is located, where its stack begins, where its current top of stack pointer is, what its priority is, etc. The API call duly initializes the TCB entries for `A1`. Being the first invocation to the `xTaskCreate` API, it also creates and initializes the task queues that the OS maintains: the ready queue which is an array of FIFO queues, one for each priority; and the delayed and suspended queues. It finally adds `A1` to the ready queue and returns. Next, `main` calls `xTaskCreate` for `B2` and the API call sets up the stack and TCB for `B2` and adds it to the ready queue, in a similar way. The next instruction in `main` is a call to the `vTaskStartScheduler` API. This call creates the "idle" task with priority 0, and adds it to the ready queue. It also initializes the system clock (`xTickCount`) and sets the timer tick interrupt to occur at the required frequency. Finally, it does a context-switch to the highest priority ready task. That is it restores its execution state, namely the contents of its registers, from the task's stack where they were stored previously. Thus the processor will next execute the instruction in the task that

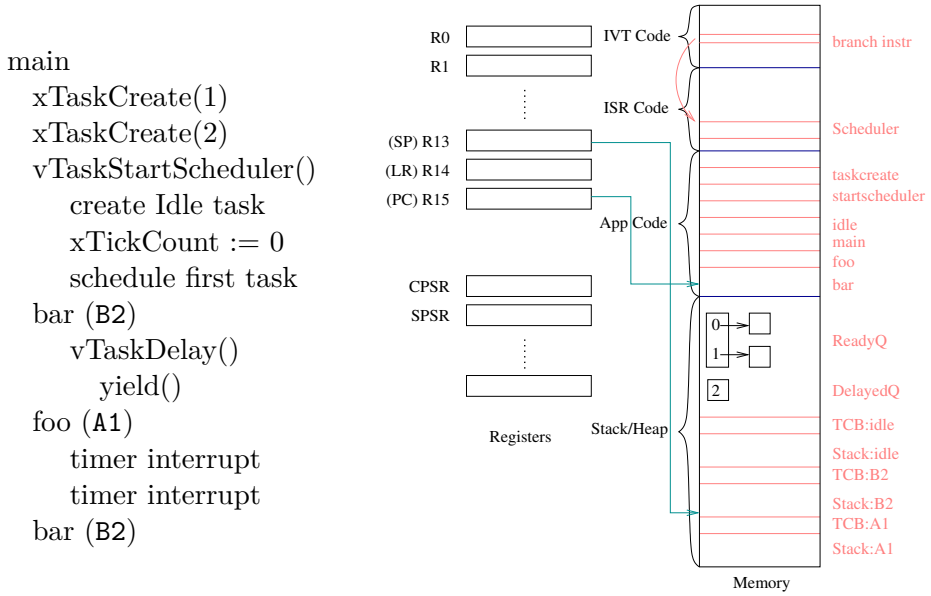


Figure 7.4: Order of statements executed in example application (left) and the memory contents during the execution (right).

is resumed. In our example, this means that `B2` will now begin execution.

When `B2` begins execution it makes a call to the `vTaskDelay` API. The code for this API call will add `B2` to a "delayed queue" which is a priority queue of delayed tasks, with a key value equal to the current tick-count plus 2, where 2 is the number of clock ticks for which the task needs to be delayed and that is passed as an argument to the `vTaskDelay` API. The API code then does a `yield` (a software interrupt), which is trapped by the scheduler. The scheduler picks the (longest waiting) highest priority ready task, which in this case is `A1`, and makes it the running task. Before this the scheduler saves the register context of `B2` on its stack, and restores the register context of `A1` from its stack.

`A1` now executes it's trivial `for` loop, till an IRQ (Interrupt Request) for the next timer tick arrives from the hardware clock. This interrupt is again trapped by the scheduler, and it increments its clock value (or tick-count). The scheduler then checks if any of the delayed tasks have a time-to-awake value that equals the current tick-count. There are none, and the scheduler hands back control to `A1`. However when the next timer interrupt takes place, the scheduler finds that `B2`'s time-to-awake equals the current tick-count, and moves it to the ready queue. Since there is now a higher priority ready task, `A1` is switched out (by saving its register context) and `B2`'s context is restored and made to execute. The execution continues in this way, ad infinitum.

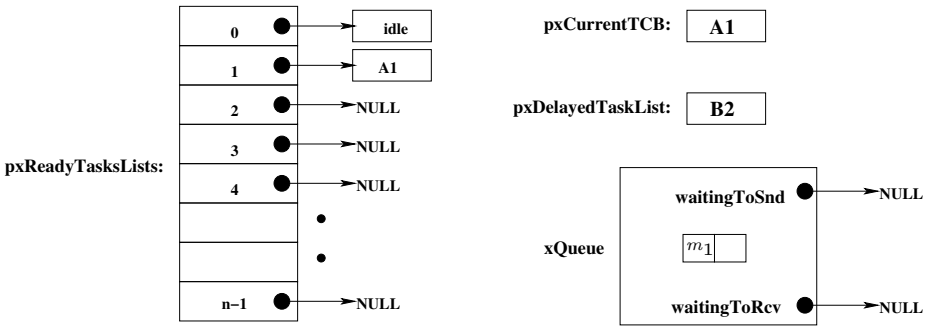


Figure 7.5: Illustrating some of the data-structures maintained by FreeRTOS.

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

Figure 7.6: Excerpts from the `vTaskDelay` API.

7.3 Data-structures maintained by FreeRTOS

FreeRTOS maintains a number of data-structures which are accessed and updated by the scheduler and the various API calls.

Fig. 7.5 shows a snapshot of some of the main scheduling related data-structures during the execution of a hypothetical application. The ready queue is maintained as an array of FIFO queues (one for each priority). The ready queue in the figure has the task `A1` with priority 1, and also has the idle task which has priority 0. The pointer variable `pxCurrentTCB` represents the running task and in this case it is the task `A1`. The delayed list in the snapshot contains the task `B2`. The figure also shows a message queue of capacity 2 bytes, with one free slot. Associated with the message queue are two priority queues (called "event" queues) which contain tasks that are blocked on sending to (respectively receiving from) the message queue. Since the message queue in the snapshot is neither full nor empty, there are no tasks in the event queues.

Fig. 7.6 shows excerpts from the code of the `vTaskDelay` method. 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,

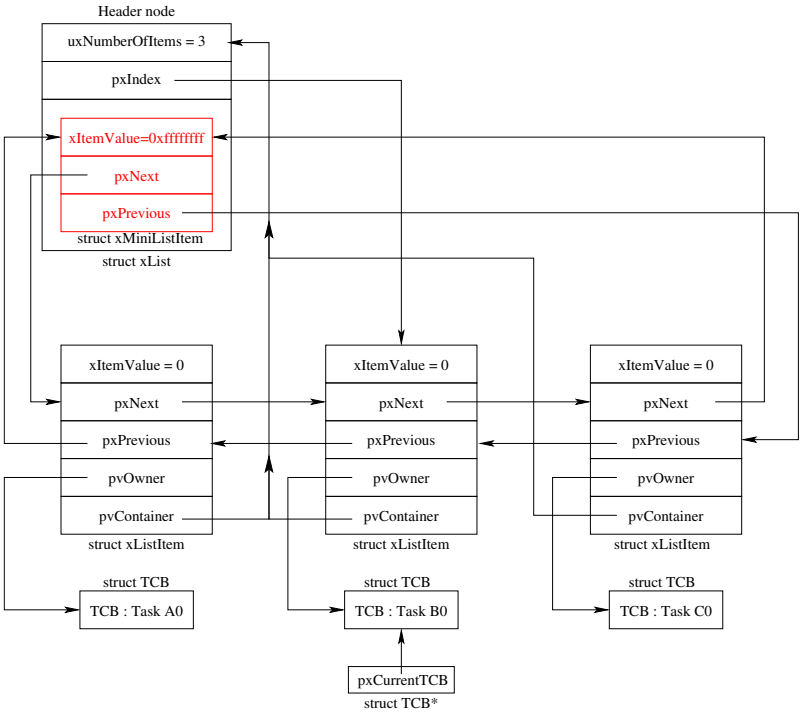


Figure 7.7: An example `xList` instance of a ready (FIFO) queue of priority 0.

when this method is invoked with an argument greater than zero. The last 3 steps are done using calls to a list data-structure called `xList`, which we describe in the subsequent paragraphs. The list from which a node to be deleted need not be passed as an argument to the `vListRemove` API, as one of the field (`pvContainer`) in the node gives the list in which it is present.

The core data-structure used in FreeRTOS is `xList`, which is a circular doubly linked-list. Each node in it is of type `xListItem` 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`), get the task that owns the node at the head of a priority queue (`listGET_OWNER_OF_HEAD_ENTRY`), and rotate left and then get the task that owns the next node of a FIFO queue (`listGET_OWNER_OF_NEXT_ENTRY`).

The `xList` data-structure is used to represent each of the task lists maintained by the FreeRTOS kernel. Fig. 7.7 shows an example instance of the `xList` data-structure. Each of the `xList` instances comprises a header node and a circular doubly linked-list of task-nodes. The header node contains the number of task-nodes present in the linked-list (`uxNumberOfItems`), a pointer to one of the task-nodes in the linked-list (`pxIndex`), and a special node namely `xListEnd` which is used to mark the end of the circular linked-list for a priority queue. A task-node contains the following information, in addition to the pointers to the successor and predecessor nodes: (i) a key value (`xItemValue`), (ii) a pointer to the TCB of the task which owns this node (`pvOwner`) and

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

Figure 7.8: Excerpts from the `xList` method `vListInsert`.

(iii) a pointer to the header node of the list in which this node is present (`pvContainer`).

The following are some of the example task lists maintained in the FreeRTOS kernel. The set of ready tasks is maintained as an array of `xList` instances, each of which represents a FIFO queue. Here tasks are arranged in the order of their insertions into the respective ready queues. The set of delayed tasks is maintained as two `xList` instances, each of which represents a priority queue. Here tasks are arranged in the increasing (or more precisely, non-decreasing) order of their time-to-awake values. In addition to the above, the waiting lists (of tasks) associated with the message queues are also represented using the `xList` data-structure.

The `xList` instance shown in Fig. 7.7, represents a ready (FIFO) queue corresponding to the priority value 0. The first node of a FIFO queue is the node pointed to by the `pxNext` field of the node pointed to by the `pxIndex` field of the header node of the list. Thus the instance shown in Fig. 7.7, represents the sequence of tasks $\langle$`C0`, `A0`, `B0`$\rangle$.

Fig. 7.8 shows a part of the `vListInsert` method of `xList`. This operation is used to insert a node into a priority queue. It first finds the required position of the node to be inserted (given by the second argument), and then inserts it so that the resulting `xList` instance maintains the nodes in the increasing (or more precisely, non-decreasing) order of key values. We note that the value of the key field (`xItemValue`) of the end-marking node (`xListEnd`) is the maximum possible value allowed by its data type. A task node in a priority queue is expected to have a key value which is strictly less than the maximum value allowed by its data type.

Invoking the correct API operation is important to get the intended functionality of the three kinds of task lists supported by the `xList` data-structure. For example, the method `vListInsert` should be used for inserting a task to a priority queue, while `vListInsertEnd` is the operation to be used for inserting a task to a FIFO queue.

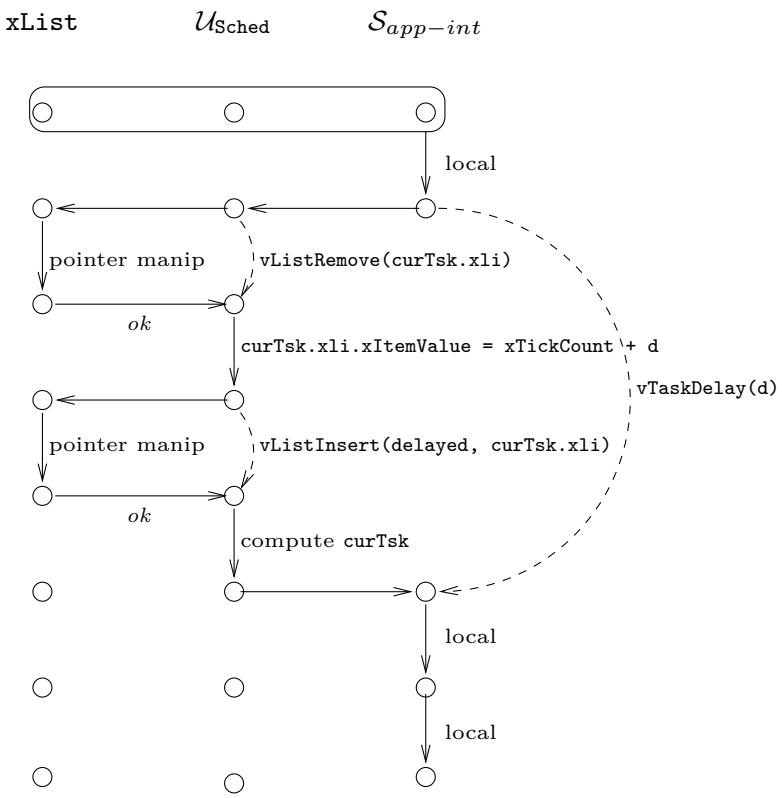


Figure 7.9: An application running with the FreeRTOS scheduler. Solid downward-arrows represent local transitions and dashed arrows represent operation calls to sub-ADTs.

7.4 Overview of FreeRTOS verification

In this section we give an overview of different steps involved in the verification of the FreeRTOS scheduler. We apply our "directed refinement methodology" of Sec. 5.1, to verify the functional correctness of the FreeRTOS scheduler.

In an actual execution of a FreeRTOS application, API calls could be interleaved in a non-atomic fashion. For example, while the `vTaskDelay` method is running, a tick interrupt might arrive causing the kernel method `vTaskIncrementTick` to execute, before the call to `vTaskDelay` finishes. But in this verification exercise, we assume a limited form of preemption in which interleavings are assumed to happen only at API boundaries.

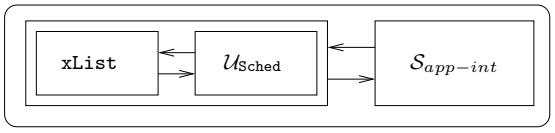


Figure 7.10: State of an application running with the FreeRTOS scheduler.

We view the system corresponding to a FreeRTOS application as conceptually having the following two components. The first 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 second one is a component which we call the *scheduler*, whose job 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 (clock value), etc. The interpreter component makes calls to the operations (APIs) 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. 2.3, the application-interpreter is a FreeRTOS-client transition system (say $\mathcal{S}_{app-int}$), that uses the scheduler component as an ADT.

The scheduler itself is an ADT that comprises two components: its own local state (keeping track of the tick-count, array of ready lists, etc), and the state of `xList` that maintains a bunch of lists. Thus, in the terminology of Sec. 3.4, the scheduler is an `xList`-client ADT transition system (say $\mathcal{U}_{\texttt{Sched}}$) and hence is of the form $\mathcal{U}_{\texttt{Sched}}[\texttt{xList}]$. The scheduler component $\mathcal{U}_{\texttt{Sched}}$ communicates with the `xList` sub-ADT via interface operations like `vListInsert`, `vListRemove`, `list_GET_OWNER_OF_HEAD_ENTRY`, etc. In Fig. 7.10, we show the components of a state of an application running with FreeRTOS.

A typical execution of an interpreted application with its scheduler ADT and the `xList` sub-ADT is shown in Fig. 7.9. The application interpreter ($\mathcal{S}_{app-int}$) invokes interface operations in the scheduler ADT ($\mathcal{U}_{\texttt{Sched}}$). For instance $\mathcal{S}_{app-int}$ in the figure invokes the `vTaskDelay` API with the ticks-to-delay as argument. The variable `curTsk` is assumed to be a global variable representing the currently running task (`pxCurrentTCB`). The scheduler ADT $\mathcal{U}_{\texttt{Sched}}$ in turn makes calls to the `xList` sub-ADT. For instance the body of the method `vTaskDelay` in $\mathcal{U}_{\texttt{Sched}}$ of the figure does the following. Firstly it calls the `vListRemove` API to remove the current task from the ready queue. Then it computes the time-to-awake value and assigns it to the `xItemValue` field of the node which represents the current task. Then it invokes the `xList` API `vListInsert` to insert this node to the delayed queue. Finally it computes and returns (via the global variable `curTsk`) a new ready task as the current task to run. One of the invariants maintained by the FreeRTOS scheduler is that there is always a task which is ready to run. The "idle" task is used to maintain this invariant and it is assumed that FreeRTOS applications will never try to delay or block the idle task.

In this work, our interest lies in the verification of the conceptual scheduler component. We restrict ourselves to the task-related APIs in the file `task.c`, and the `xList` APIs in the file `list.c` of the FreeRTOS code. We consider the relevant parts of this code to be the implementation $\mathcal{P}$ of the scheduler component. The list of task-related APIs verified in this work is shown in Table 7.1. In addition to the task-related APIs, we verified the set of all list-related APIs in the `xList` sub-ADT. Our aim is to specify and verify this ADT implementation ($\mathcal{U}_{\texttt{Sched}}[\texttt{xList}]$) using our theory of refinement explained in Chap. 2 and Chap. 3, and by using the methodology outlined in Sec. 5.1.

FreeRTOS API	Basic Functionality
`vTaskStartScheduler`	Schedule the longest waiting highest priority task.
`xTaskCreate`	Create the given task and add it to the ready queue.
`vTaskDelete`	Delete the given task (move it to the deleted queue).
`vTaskDelay`	Delay the current task for a given period of time.
`vTaskDelayUntil`	Delay the current task relative to its previous wake-time.
`vTaskIncrementTick`	Increment the tick-count and awaken delayed tasks.
`vTaskPrioritySet`	Change the priority of the given task.
`vTaskSuspend`	Suspend the given task.
`vTaskResume`	Resume the given task which is in the *suspended* state.
`xTaskGetTickCount`	Get the value of the current tick-count.
`vTaskPriorityInherit`	Implements the priority inheritance scheme in FreeRTOS.

Table 7.1: Scheduler APIs considered in this verification

The strategy used to verify the functional correctness of the FreeRTOS scheduler is shown in Fig. 7.11. Following the methodology of Sec. 5.1, we first build a high-level deterministic model $\mathcal{M}$ of the scheduler in the Z specification language.

Next we observe that the existing implementation of the FreeRTOS scheduler $\mathcal{P}$ (or $\mathcal{U}_{\texttt{Sched}}$) uses a sub-ADT, namely `xList`, and thus is of the form $\mathcal{U}_{\texttt{Sched}}[\texttt{xList}]$. We now obtain a simplified FreeRTOS implementation say $\mathcal{P}_1$ from $\mathcal{P}$ by replacing the sub-ADT, `xList` in $\mathcal{U}_{\texttt{Sched}}$ by a ghost implementation in VCC which we call `xListMap`. Thus the simplified FreeRTOS implementation $\mathcal{P}_1$ is of the form $\mathcal{U}_{\texttt{Sched}}[\texttt{xListMap}]$.

The set of delayed tasks is modeled as a single sequence of tasks in the Z model $\mathcal{M}$. But this set of tasks is implemented as two different lists in FreeRTOS namely `delayed-list` and `overflow-delayed-list`. The following is the reason for using two delayed lists in $\mathcal{P}_1$. The value of tick-count in FreeRTOS cycles in the interval `[0,maxNumVal]`, where `maxNumVal` denotes the maximum value that an `unsigned` variable can take in C. Now a task to be delayed is added to `delayed-list`, when its value of time-to-awake is less than or equal to `maxNumVal`. Otherwise the task is added to `overflow-delayed-list`, with the value of time-to-awake computed modulo "`(maxNumVal+1)`". The FreeRTOS scheduler interchanges the roles of these delayed lists when the clock value cycles back to `0` from `maxNumVal`.

We reduce the gap between the models $\mathcal{M}$ and $\mathcal{P}_1$ by refining the high-level Z model $\mathcal{M}$ to a low-level Z model called $\mathcal{M}_1$, by adding a separate "overflow-delayed-list" list to store tasks whose time-to-awake values are beyond `maxNumVal`. The Z models $\mathcal{M}$ and $\mathcal{M}_1$ differ only in the way they maintain the set of delayed tasks. Therefore $vTaskDelay$ and $vTaskDelayUntil$ are the only operation-schemas that differ between the models $\mathcal{M}$ and $\mathcal{M}_1$. We verified the refinement between $\mathcal{M}$ and $\mathcal{M}_1$ as follows. Firstly we applied our Z-to-VCC translation technique explained in Sec. 5.2.3, to obtain the ghost implementations $\mathcal{G}$ and $\mathcal{G}_1$ respectively from $\mathcal{M}$ and $\mathcal{M}_1$. In fact

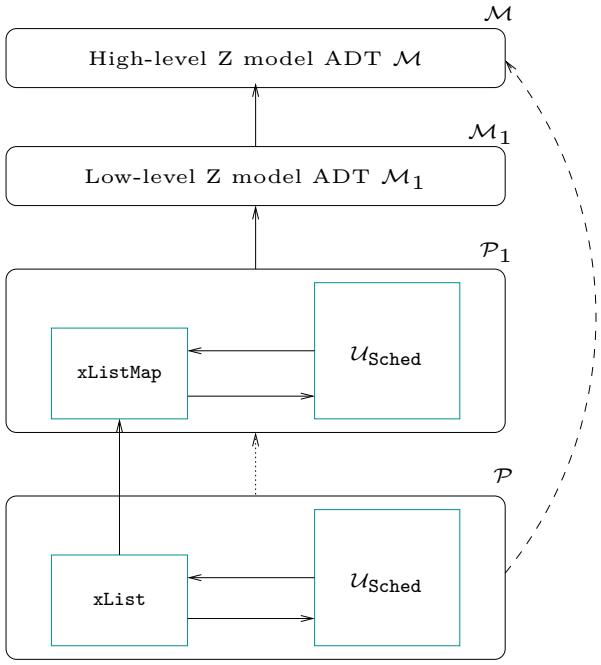


Figure 7.11: Illustrating the correctness proof of the FreeRTOS scheduler. Bold upward-arrows represent directly proved refinements, the dotted arrow represents the refinement inferred using our substitutivity result, and the dashed arrow represents the refinement inferred using our transitivity result.

we obtained ghost methods corresponding to the operations $vTaskDelay$ and $vTaskDelayUntil$. We then applied the technique explained in Sec. 5.2.4 to check the refinement between the ghost models $\mathcal{G}$ and $\mathcal{G}_1$.

The models $\mathcal{M}_1$ and $\mathcal{P}_1$ are very similar. Hence we applied our "direct import approach with quantifier elimination" of Sec. 5.3.1 to verify the refinement between the models $\mathcal{M}_1$ and $\mathcal{P}_1$. We found a number of bugs in this step of FreeRTOS verification. We fixed all these bugs and the fixed implementation is proved to be a refinement of the model $\mathcal{M}_1$.

The final step in the FreeRTOS verification is the proof of refinement between the list models `xListMap` and `xList`. We applied our "combined" approach of Sec. 5.3.2 to prove that `xList` refines `xListMap`. Now it follows from our substitutivity result that $\mathcal{P} = \mathcal{U}_{\texttt{Sched}}[\texttt{xList}]$ refines $\mathcal{P}_1 = \mathcal{U}_{\texttt{Sched}}[\texttt{xListMap}]$. Also it follows from transitivity of refinement that the C implementation, $\mathcal{P}$ of FreeRTOS refines the high-level Z model $\mathcal{M}$.

The sequence of steps in our verification process is summarized below:

$$
\begin{aligned}
\mathcal{M} &\succeq \mathcal{M}_1 \\
&\quad \text{(proved via the ghost models } \mathcal{G} \text{ and } \mathcal{G}_1, \text{ using our} \\
&\quad \text{“combined” approach of Sec. 5.3.2)} \\
&\succeq \mathcal{P}_1 = \mathcal{U}_{\texttt{Sched}}[\texttt{xListMap}] \\
&\quad \text{(proved using Theorem 6.2 and our “direct import} \\
&\quad \text{approach with quantifier elimination” of Sec. 5.3.1)} \\
\texttt{xListMap} &\succeq \texttt{xList} \\
&\quad \text{(proved using our “combined” approach of Sec. 5.3.2)} \\
\mathcal{P}_1 = \mathcal{U}_{\texttt{Sched}}[\texttt{xListMap}] &\succeq \mathcal{U}_{\texttt{Sched}}[\texttt{xList}] = \mathcal{P} \\
&\quad \text{(follows from our substitutivity result (Theorem 3.2),} \\
&\quad \text{since } \texttt{xListMap} \succeq \texttt{xList}) \\
\mathcal{M} &\succeq \mathcal{P} \\
&\quad \text{(follows from transitivity of refinement (Proposition 2.1))}
\end{aligned}
$$

In this verification of FreeRTOS we assumed a limited form of concurrent/interleaved execution. In particular we assumed that if an API starts executing, it is allowed to continue its execution until it finishes. Thus the properties guaranteed about FreeRTOS by our verification hold only if this assumption is satisfied. But such an assumption is not typically satisfied by a FreeRTOS application. For example an interrupt can occur during the execution of an API which may cause another API of a different task to execute before the currently running API completes. We suggest below some ideas to extend our verification guarantees when concurrent execution is allowed.

One could use a method in VCC like the one proposed by Cohen et al [17] to prove refinement when the APIs are allowed to execute in a concurrent fashion. One could also utilize correctness requirements for shared data objects, like linearizability proposed by Herlihy and Wing [27] to extend our verification to ensure correctness when concurrent execution is allowed.

In subsequent sections we describe the details of the steps involved in the verification of the FreeRTOS scheduler.

7.5 Details of steps in FreeRTOS verification

In this section we report the details of the steps involved in the verification of the FreeRTOS scheduler. We have verified the functional correctness of the scheduler-related functionality of the FreeRTOS version 6.1.1. All artifacts of our FreeRTOS verification case-study are available in [24].

We did some transformations (or changes) in the FreeRTOS scheduler code to view it as an ADT implementation. Firstly we added a separate initialization operation to the FreeRTOS API. In the existing implementation of the

FreeRTOS scheduler the `vTaskStartScheduler` API includes a part of the initialization code, and rest is included in the `xTaskCreate` API such that this part of the code will be executed when a FreeRTOS application calls the `xTaskCreate` API for the first time. We moved these initialization codes into a separate initialization API and thus require a FreeRTOS application to invoke the initialization API before invoking any other FreeRTOS APIs. Recall that there is a designated initialization operation in an ADT type.

Secondly we replaced two configurable constants in the FreeRTOS scheduler with corresponding variables. In the FreeRTOS scheduler, the maximum priority of a task is represented by a user-configurable macro in the file "`FreeRTOSConfig.h`". The maximum numeric value that a kernel variable can take is also represented as a macro, but it is fixed for a compiler/hardware combination. Our aim is to do a parameterized verification of the FreeRTOS scheduler such that the verification guarantee provided is independent of the values for these configurable macros. To achieve this, we replaced these macros with corresponding variables such that their values can be initialized via arguments to the initialization API discussed above. All other APIs are specified to not modify the values for these variables.

7.5.1 Z models

In this section we describe the Z models used in the verification of the FreeRTOS scheduler. We use two Z models in the sequence of refinements to prove the functional correctness of the FreeRTOS scheduler. We started with a high-level Z model capturing the intended functionality of the FreeRTOS scheduler and then we refined it to a low-level Z model to capture some implementation details. Some of the important aspects of these models are described in the following subsections.

High-level Z model

We tried to understand the "intended" behavior of the FreeRTOS scheduler based on the details given in the FreeRTOS user guide [10]. For some of the APIs, we had to look at the code and the comments therein to infer the meaning.

Next we specified this behavior in a high-level Z model which we call $\mathcal{M}$. 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 priority-indexed array of FIFO queues used in FreeRTOS.

The data-schema of Fig. 7.12, shows the main state components in the high-level model $\mathcal{M}$. The variables *maxPrio* and *maxNumVal* denote the variables that we added in places of the configurable macros in FreeRTOS as described in the preamble of Sec. 7.5.

The type *TASK* is a free data-type in Z, which represents the set of all tasks which will be created in the system and the variable *tasks* represents the set of tasks already being created.

$$
\begin{array}{|l}
\hline
\textit{Scheduler}^{\mathcal{M}} \\
\textit{maxPrio}, \textit{maxNumVal}, \textit{tickCount}, \textit{topReadyPriority} : \mathbb{N} \\
\textit{tasks} : \mathbb{P}\ \textit{TASK} \\
\textit{priority} : \textit{TASK} \pfun \mathbb{N} \\
\textit{running}, \textit{idle} : \textit{TASK} \\
\textit{ready} : \text{seq (iseq } \textit{TASK}) \\
\textit{delayed} : \text{seq } (\textit{TASK} \times \mathbb{N}) \\
\textit{blocked} : \text{seq } \textit{TASK} \\
\ldots \\
\hline
\textit{idle} \in \textit{tasks} \land \textit{idle} \in \text{ran} \frown\!/ (\text{ran } \textit{ready}) \\
\textit{running\_task} \in \textit{tasks} \land \textit{topReadyPriority} \in \text{dom } \textit{ready} \\
\textit{running\_task} = \textit{head ready}(\textit{topReadyPriority}) \\
\forall\, i, j : \text{dom } \textit{delayed} \mid (i < j) \implies (\textit{delayed}(i).2 \leq \textit{delayed}(j).2) \\
\forall\, \textit{tcn} : \text{ran } \textit{delayed} \mid \textit{tcn}.2 > \textit{tickCount} \\
\textit{tickCount} \leq \textit{maxNumVal} \\
\forall\, i, j : \text{dom } \textit{blocked} \mid (i < j) \implies (\textit{priority}(\textit{blocked}(i)) \geq \textit{priority}(\textit{blocked}(j))) \\
\ldots \\
\hline
\end{array}
$$

Figure 7.12: The data-schema modeling the states of the FreeRTOS scheduler.

The priorities of tasks in the system is modeled as a partial function called *priority* from the set of tasks in the system to the set of natural numbers. Now the priority of a task t is obtained by $\textit{priority}(t)$.

The set of ready tasks is modeled as a sequence of injective sequences (sequences with distinct elements) called *ready*. A finite sequence of length n in Z is a map from the set $\{1, 2, \ldots, n\}$ to the set of elements in the sequence. Now the expression "dom s" denotes the set $\{1, 2, \ldots, n\}$ and the expression "ran s" represents the set of elements present in s, for a finite sequence s of length n. The expression $s.i$ can be used to access the ith element in the sequence s. Now the sequence of ready tasks of the same priority i can be accessed by the expression $\textit{ready}(i)$ and the j^{th} task in this sequence can be accessed by the expression $\textit{ready}(i).j$. In Z, $\frown\!/$ is a "flatten" operator that takes a list of lists and flatten it by concatenating the lists in it. Thus the expression $\frown\!/\{\langle e_1, e_2\rangle, \langle e_3, e_4\rangle\}$ gives the sequence $\langle e_1, e_2, e_3, e_4\rangle$.

The set of delayed tasks is modeled as a sequence of pairs called *delayed*. An element "(t, n)" in this sequence represents the delayed task t with time-to-awake value n. The expression "$(t, n).2$" represents the second element (that is n) in this pair. There are two invariants specified on this sequence. The first specifies that the pairs are arranged in the increasing (or more precisely, non-decreasing) order of time-to-awake values. The second invariant specifies that the time-to-awake value of each of the delayed tasks is strictly greater than the current clock value (*tickCount*).

We model an event queue in FreeRTOS as a sequence of tasks called *blocked*.

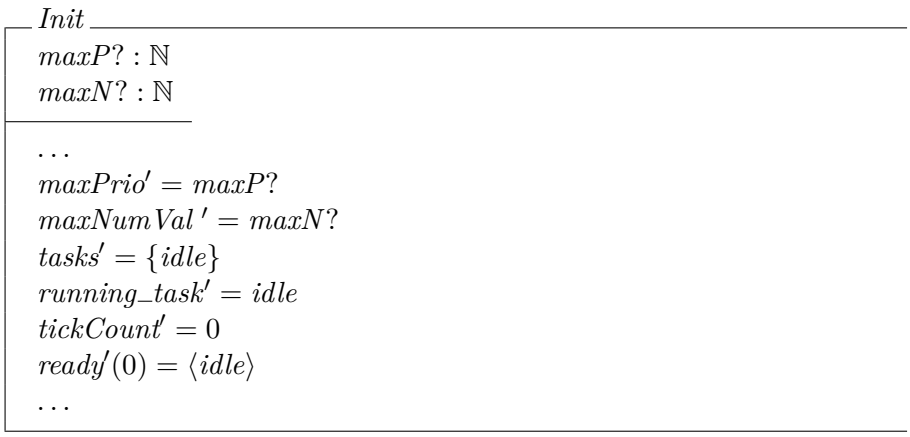


Figure 7.13: The operation-schema for the *init* operation in the Z model $\mathcal{M}$.

.

This represents an event queue like the set of tasks waiting to receive a message from a message queue. There is an invariant specified on this sequence to maintain the tasks in this sequence in the decreasing (or more precisely, non-increasing) order of their priority values.

We now describe two important operation-schemas in the Z model $\mathcal{M}$. The first one models the initialization operation discussed above and the second one models the `vTaskDelay` API.

The operation-schema of Fig. 7.13, models the initialization operation in the FreeRTOS scheduler. It takes two arguments *maxN*? and *maxP*? which give the values to initialize the state variables *maxNumVal* and *maxPrio* respectively. This schema initializes the system by defining a valuation for the fields in the data-schema. For instance, the initialization results in a state where the clock value is zero and the task "*idle*" is running.

The operation-schema of Fig. 7.14, models the `vTaskDelay` API. Recall that this API is for delaying the running task for a given number of clock ticks. This schema includes an input variable called *delay*?, which represents the number of clock ticks for which the current task needs to be delayed. This schema assumes that the requested delay period is greater than zero and also that the ready sequence corresponding to the priority of the running task (represented by *topReadyPriority*) contains at least one task other than the currently running task, which is at the head of this sequence. The temporary sequences *delayedPrefix* and *delayedSuffix* are constrained to be the prefix and the suffix which divides the sequence *delayed* such that the new pair representing the currently running task should be inserted in-between these two. The symbol "$\oplus$" represents the function overriding operator, which is used in this schema to replace the sequence of ready tasks indexed by the priority of running task with the tail of this sequence.

TaskDelay

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

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

Figure 7.14: Operation schema for the *vTaskDelay* API.

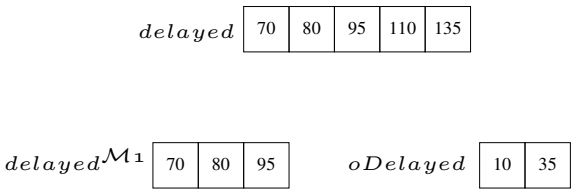


Figure 7.15: An instance of the list *delayed* in $\mathcal{M}$ and the corresponding instances of the lists $delayed^{\mathcal{M}_1}$ *oDelayed* in $\mathcal{M}_1$.

Low-level Z model

In this step of the verification, our aim was to reduce the gap between the models $\mathcal{M}$ and $\mathcal{P}_1$ of Fig. 7.11. There are two mismatches between these two models, which we describe in the following paragraphs.

The first mismatch is the way the set of delayed tasks are maintained in these models. In the model $\mathcal{P}_1$ these tasks are represented by two delayed lists called `delayed-list` and `overflow-delayed-list`, while these tasks are represented by a single sequence called *delayed* in the model $\mathcal{M}$. The following is the reason for using two lists in the model $\mathcal{P}_1$. The clock value in $\mathcal{P}_1$ cycles in the interval $[0, maxNumVal]$. Recall that "*maxNumVal*" represents the maximum value that an unsigned int in C can take. FreeRTOS allows to delay the running task for a maximum period of "*maxNumVal*" clock ticks. Therefore a delayed task may have its time-to-awake value in the interval $[0, 2 \times maxNumVal]$. This is managed in the model $\mathcal{P}_1$ by maintaining the above two delayed lists such that `delayed-list` stores the delayed tasks with time-to-awake values in the interval $[0, maxNumVal]$ and `overflow-delayed-list` stores the delayed tasks with time-to-awake values

$Scheduler^{\mathcal{M}_1}$

$\ldots$

$delayed^{\mathcal{M}_1}, oDelayed : (\text{seq } TASK \times \mathbb{N})$

$\ldots$

$\forall\, tcn : \text{ran } delayed^{\mathcal{M}_1} \mid tcn.2 \leq maxNumVal$

$\forall\, tcn : \text{ran } delayed^{\mathcal{M}_1} \mid tcn.2 > tickCount$

$\forall\, tcn : \text{ran } oDelayed \mid tcn.2 \leq maxNumVal$

$//$ (modulo "$(maxNumVal + 1)$" value.)

$\forall\, tcn : \text{ran } oDelayed \mid tcn.2 \leq tickCount$

$\forall\, i, j : \text{dom } oDelayed \mid (i < j) \implies oDelayed(i).2 \leq oDelayed(j).2$

$\forall\, i, j : \text{dom } blocked \mid (i < j) \implies$

$(\ maxPrio - priority(blocked(i))\ ) \leq (\ maxPrio - priority(blocked(j))\ )$

Figure 7.16: The data-schema of the low-level Z model $\mathcal{M}_1$.

in the interval $[maxNumVal+1, 2 \times maxNumVal]$. The latter contains the tasks which have to be awakened after the clock value overflows (cycles back to 0 from $maxNumVal$), and hence the name `overflow-delayed-list`. In fact the time-to-awake values in the latter is maintained modulo "$(maxNumVal+1)$", which is necessary as $maxNumVal$ represents the maximum value that a numeric variable can store. That is if "val" is the required time-to-awake value in the interval $[maxNumVal+1, 2 \times maxNumVal]$, then the key value stored in `overflow-delayed-list` is "$val-(maxNumVal+1)$" rather than "val". FreeRTOS interchanges the roles of these two lists when the clock value overflows.

Fig. 7.15 shows an instance of the single delayed sequence in the high-level Z model $\mathcal{M}$ and instances of the corresponding sequences used in the low-level Z model $\mathcal{M}_1$. In this figure we assume that $maxNumVal = 99$ and $tickCount = 65$.

The second mismatch between the models $\mathcal{M}$ and $\mathcal{P}_1$ is the order in which the set of blocked tasks is maintained in these models. In the model $\mathcal{P}_1$, these tasks are maintained in the increasing order of their *complemented* priority values, while these tasks are maintained in the decreasing order of their priority values in the model $\mathcal{M}$. For a task t with priority p, the complemented priority is "$maxPrio - p$". The reason for using complemented priority to order the set of blocked tasks in FreeRTOS is as follows. The task lists which represent the set of delayed tasks are maintained in the increasing order of key (time-to-awake in this case) values. Therefore maintaining the set of blocked tasks also in the increasing order of key (complemented priority in this case) values enables FreeRTOS to use the same data-structure namely `xList` to represent each of these priority queues thereby reducing the footprint of the code.

We bridge the above gap between the models $\mathcal{M}$ and $\mathcal{P}_1$ by refining the high-level Z model $\mathcal{M}$ to a low-level Z model called $\mathcal{M}_1$. The data-schema of Fig. 7.16, shows the important components which are added (or changed) to obtain this schema from the data-schema of the model $\mathcal{M}$. To rectify the first mismatch discussed above, we divided the single sequence *delayed*,

$TaskDelay^{\mathcal{M}_1}$

$delay? : \mathbb{N}$
$oDelayedPrefix : (\text{seq } TASK \times \mathbb{N})$
$oDelayedSuffix : (\text{seq } TASK \times \mathbb{N})$
$\ldots$

$delay? + tickCount > maxNumVal$
$\ldots$
$oDelayed' = oDelayedPrefix^\frown$
$\quad \langle (running\_task, (tickCount + delay?) - (maxNumVal + 1)) \rangle^\frown$
$\quad oDelayedSuffix$
$delayed^{\mathcal{M}_1\prime} = delayed^{\mathcal{M}_1}$
$\ldots$

Figure 7.17: An operation-schema for the *vTaskDelay* operation in the model $\mathcal{M}_1$.

which represents the set of delayed tasks in the model $\mathcal{M}$ to two sequences namely $delayed^{\mathcal{M}_1}$ and *oDelayed* in the model $\mathcal{M}_1$, which respectively store the delayed tasks with time-to-awake values in the intervals: $[0, maxNumVal]$ and $[(maxNumVal + 1), 2 * maxNumVal]$. The latter stores the time-to-awake values modulo "$(maxNumVal + 1)$". The second mismatch discussed above is rectified by replacing the invariant on the sequence of blocked tasks with a *logically equivalent* invariant in terms of complemented priority values.

The operation-schemas in the model $\mathcal{M}_1$ for the FreeRTOS scheduler APIs *init*, *vTaskDelay*, *vTaskDelayUntil* and *vTaskIncrementTick* are obtained from the model $\mathcal{M}$ by making necessary changes in accordance with the changes made for the data fields discussed above. The rest of the operation-schemas remain unchanged in the refined model.

Fig. 7.17 shows an operation-schema in the model $\mathcal{M}_1$ for the *vTaskDelay* operation in FreeRTOS. This schema assumes that the required value of time-to-awake for running task is in the interval $[(maxNumVal+1), 2 \times maxNumVal]$, and hence the running task needs to be added to the sequence *oDelayed*. This schema applies a similar technique as in the schema of Fig. 7.14 to update the state of the data-schema, except that here the time-to-awake value is computed modulo $(maxNumVal + 1)$.

7.5.2 Checking refinement between Z models

Now we explain the technique used to verify the refinement between the Z models $\mathcal{M}$ and $\mathcal{M}_1$. To do this we first applied our Z-to-VCC translation technique explained in Sec. 5.2.3, to obtain the ghost models $\mathcal{G}$ and $\mathcal{G}_1$ respectively from the Z models $\mathcal{M}$ and $\mathcal{M}_1$. In fact we translated only those schemas which differ between these models. Then we applied the "combined" approach

```
struct
{
   ...
   _(ghost \bool tasks[TaskType])
   _(ghost TaskType delayed-one[\natural])
   _(ghost \natural delayed-two[\natural])
   _(ghost \natural delayedLength)
   ...
   // invariants
   ...
   _(invariant \forall \natural i,j;
     ((i < j) && (j < delayedLength)) ==>
                    (delayed-two[i] <= delayed-two[j]))
   ...
} SchedulerM;
```

Figure 7.18: A part of the state-structure modeling the data-schema of the high-level Z model $\mathcal{M}$.

explained in Sec. 5.3.2 to check the refinement between the ghost models $\mathcal{G}$ and $\mathcal{G}_1$.

The data-structure of Fig. 7.18, shows a part of the state-structure obtained from the data-schema of the high-level Z model $\mathcal{M}$. We applied our table lookup procedure explained in Sec. 5.2.3, to translate the data-schema into this state-structure. This figure mainly shows the components of the structure which model the set of delayed tasks in the system. It also shows how the set of tasks in the system is represented in this ghost model $\mathcal{G}$.

The set of tasks in the system is represented by a ghost field called `tasks`, which is a map from the set of all tasks in the system to the set of boolean values $\{true, false\}$. A task in the system is represented by a unique task-identifier of type `TaskType`, which is simply the type `unsigned` in C. The value of a task-identifier t under this map is *true*, iff t is a valid (already created) task in the system.

The set of delayed tasks in the system, which is represented as a sequence of pairs in the Z model $\mathcal{M}$, is represented by two ghost maps namely `delayed-one` and `delayed-two`. The first one represents the sequence of delayed tasks and the second one represents the sequence of time-to-awake values such that the i^{th} value in the second sequence represents the time-to-awake value for the i^{th} task in the first sequence. That is, `delayed-one[i]` and `delayed-two[i]` respectively represent the i^{th} delayed task and its time-to-awake value. The figure also shows the translated version of the first invariant on the delayed list in the data-schema of the Z model $\mathcal{M}$.

The state-structure of the ghost model $\mathcal{G}_1$ and the methods of the ghost models $\mathcal{G}$ and $\mathcal{G}_1$ are obtained in a similar way from the respective Z models.

A part of the combined state-structure representing the joint-state of the ghost models $\mathcal{G}$ and $\mathcal{G}_1$ is shown in Fig. 7.19. This figure shows the gluing

```
struct
{
   ...
   //gluing invariants
   _(invariant
     (delayedLength = delayedM1Length + oDelayedLength) &&
     (\forall \natural i, (i < delayedM1Length) ==>
       ((delayedM1-one[i] == delayed-one[i]) &&
        (delayedM1-two[i] == delayed-two[i]))) &&
     (\forall \natural i, (i < oDelayedLength) ==>
      ((oDelayed-one[i] ==
        (delayed-one[delayedM1Length + i] - (maxNumVal + 1))) &&
       (oDelayed-two[i] ==
        (delayed-two[delayedM1Length + i] - (maxNumVal + 1))))))
} SchedulerMandM1;
```

Figure 7.19: A part of the structure representing the combined state of the ghost models $\mathcal{G}$ and $\mathcal{G}_1$.

invariant which relates the delayed lists of the models $\mathcal{G}$ and $\mathcal{G}_1$. Given this invariant the number of tasks in the single delayed sequence in the model $\mathcal{G}$ is the same as the sum of the number of tasks present in the two delayed sequences of the model $\mathcal{G}_1$, also the delayed sequence of the model $\mathcal{G}$ is the concatenation of the sequences of the model $\mathcal{G}_1$, except that `(maxNumVal+1)` is subtracted from the time-to-awake values of the tasks from the list `oDelayed` (see Fig. 7.15).

Now the refinement between the ghost models $\mathcal{G}$ and $\mathcal{G}_1$ is proved by using our "combined" approach of Sec. 5.3.2 as follows. For each of the ghost methods for the FreeRTOS scheduler APIs *init*, *vTaskDelay*, *vTaskDelayUntil* and *vTaskIncrementTick* we proved that the joint method results in a joint after-state satisfying the state-invariant in $\mathcal{G}$ and the gluing invariant, and also that both the abstract and concrete methods produce the same output, when the joint before-state and input satisfy the abstract state-invariant, abstract precondition and the gluing relation.

7.5.3 Verifying that $\mathcal{P}_1$ refines $\mathcal{M}_1$

We now address the task of showing that the simplified FreeRTOS implementation $\mathcal{P}_1$ refines the low-level Z model $\mathcal{M}_1$. Recall that $\mathcal{P}_1$ uses the ghost version `xListMap` rather than the linked-list `xList` for the task lists in the FreeRTOS scheduler. Fig. 7.20 shows a part of the definition of `xListMap`.

Like `xList`, the state-structure of the ghost model `xListMap` maintains a sequence of pointers to `xListItem` nodes. This sequence is represented as a ghost map called `list`. The field `length` records the number of items present in this sequence. The field `type` keeps track of whether the list is meant to be a FIFO queue or a priority queue. The invariant on the respective list is

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

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

Figure 7.20: Excerpts from the ghost implementation: `xListMap`.

```
void vTaskDelay(unsigned delay _(out unsigned oDIndex))
  _(requires (delay > (SchedP1.maxNumVal - SchedP1.tickCount)))
  ...
  _(ensures \old(pxCurrentTCB)->xGLI.xItemValue ==
            (delay - (SchedP1.maxNumVal - SchedP1.tickCount) - 1))
  _(ensures ((oDIndex <= \old(SchedP1.oDLength)) &&
     (\forall unsigned i;
       (i < oDIndex) ==> (\old(SchedP1.oD[i].xli.xItemValue) <=
                          \old(pxCurrentTCB)->xGLI.xItemValue)) &&
     (\forall unsigned i;
        ((i >= oDIndex) && (i < \old(SchedP1.oDLength))) ==>
         (\old(SchedP1.oD[i].xli.xItemValue) >
          \old(pxCurrentTCB)->xGLI.xItemValue))))
  _(ensures \forall unsigned i; (i < oDIndex) ==>
     (SchedP1.oD[i] == \old(SchedP1.oD[i]))
  _(ensures SchedP1.oD[oDIndex] == \old(pxCurrentTCB)->xGLI)
  _(ensures \forall unsigned i;
     ((i > oDIndex) && (i <= \old(SchedP1.oDLength))) ==>
      (SchedP1.oD[i] == \old(SchedP1.oD[i - 1]))
  _(ensures SchedP1.oDLength == (\old(SchedP1.oDLength) + 1))
  ...
{
     //body of the method
}
```

Figure 7.21: Excerpts from verification of the method `vTaskDelay` in the model $\mathcal{P}_1$.

defined in terms of the list-type. For instance, the invariant shown in the figure specifies that the items are stored in the increasing order of item-values, when the list is a priority queue.

The figure also shows a part of the definition of the ghost method called `vListInsert`. The annotation "`\wrapped(mList)`" essentially asserts that the instance `mList` satisfies the state-invariant in the struct `xListMap`. We use an `assume` annotation, to guess the required position of the node to be inserted in the sequence represented by the map `list`. The lambda operator in VCC is used to update the ghost map `list`.

Now we describe the technique used to verify the refinement between the models $\mathcal{M}_1$ and $\mathcal{P}_1$ in verification of the FreeRTOS scheduler. To do this we applied our "direct-import approach with quantifier elimination" described in Sec. 5.3.1. We illustrate this step below, in terms of the *vTaskDelay* API.

Fig. 7.21 shows excerpts from verification of the method `vTaskDelay` in the model $\mathcal{P}_1$. We use the "direct-import approach with quantifier elimination" to import the requirements from the Z model of Fig. 7.17 to the `vTaskDelay` API in the model $\mathcal{P}_1$. A ghost output parameter called `oDIndex` is used in this method to simplify the formulas in the function contract. This variable

represents the position at which the running task is inserted in the sequence `oD`. The expressions like "delay? + *tickCount* > *maxNumVal*" in the model $\mathcal{M}_1$ is rephrased to avoid the possible overflows in the computation. The field `xGLI` of the TCB struct represents the `xListItem` node, which in this case represents the running task in a delayed list. The field `oD` of the scheduler struct is the map which represents `overflow-delayed-list` in the model $\mathcal{P}_1$.

We applied our theory explained in Chap. 6 together with the "direct-import approach with quantifier elimination" to handle the change of data types between operations in the models $\mathcal{M}_1$ and $\mathcal{P}_1$. We assumed a particular memory map which represents the set of all task TCBs that will be created in the lifetime of the scheduler to define the family of translation pairs for checking the refinement between $\mathcal{M}_1$ and $\mathcal{P}_1$. In particular we used translation pairs similar to the one shown in Fig. 6.5.

VCC was able to check most of the resulting annotations in the APIs in $\mathcal{P}_1$, except for the `xTaskCreate` API, and a couple of other APIs we mention in Sec. 7.6. The problem with `xTaskCreate` was as follows. FreeRTOS follows a convention of keeping the running task at the *end* of the ready queue corresponding to its priority. However this convention leads to inconsistencies like the following. Consider the scenario where the tasks A1, B1 (both of priority 1) are ready, with A1 currently executing. By the FreeRTOS convention, the ready queue is the list $\langle \text{B1}, \text{A1} \rangle$. Now suppose A1 creates a task C1. The `xTaskCreate` API uses the `xList` operation `vListInsertEnd` to add C1 to the *end* of the queue, to get $\langle \text{B1}, \text{A1}, \text{C1} \rangle$. Thus the running task A1 is no longer at the end of the queue. If a couple of tick interrupts now arrive, causing A1 and then B1 to be preempted, it will be A1 that runs again (instead of C1!).

We chose to fix this problem in the design of FreeRTOS by following the convention of our Z models to keep the running task at the *head* of its ready queue. However to do this we needed to add two new functions to the `xList` (and `xListMap`) library: `listRotateLeft` and `list_GET_FIRST_ENTRY` that respectively rotate a FIFO queue by one position to the left, and return the task that owns the node at the head of the list. The method `listRotateLeft` is used in the case of preemption (time slicing within tasks of the top priority), while `list_GET_FIRST_ENTRY` is used to find the next task to run. With these changes and other fixes we mention in Sec. 7.6, VCC verifies all the API operations of $\mathcal{P}_1$.

7.5.4 Verifying that `xList` refines `xListMap`

Now we describe the method used to verify the refinement between the list models `xList` and `xListMap`. This part of the FreeRTOS scheduler verification is done jointly with Anirudh Kushwah, a Masters student of 2011-2013 batch.

The `xList` data-structure and hence the `xListMap` data-structure represent a bunch of lists in the models $\mathcal{P}$ and $\mathcal{P}_1$ respectively. However it is sufficient to consider a single pair of instances of `xList` and `xListMap`, since we are verifying the *functionality* of these data-structures which is the same for all instances.

```
struct
{
   // contents of the struct xListMap
   // contents of the struct xList
   // Gluing invaraints
   ...
   _(invariant length == uxNumberOfItems)
   _(invariant (type == PQ) ==> (list[0] == xListEnd.pxNext))
   _(invariant (type == PQ) ==> (\forall unsigned i;
     ((length > 0) && (i >= 0) && (i < (length - 1))) ==>
      ((list[i]->pxNext == list[i + 1]) &&
      (list[i + 1]->pxPrevious == list[i]))))
   _(invariant (length > 0) ==>
     ((list[length - 1]->pxNext == list[0]) &&
      (list[0]->pxPrevious == list[length - 1])))
   ...
} xListJoint;

void vListInsertJoint(xListItem *xli)
  _(requires \wrapped(xListJoint))
  _(requires xListJoint.length < MAXSIZE)
  ...
  _(ensures \wrapped(xListJoint))
{
   body of the abstract method
   body of the concrete method
}
```

Figure 7.22: Excerpts from `xList` - `xListMap` refinement check.

We applied the "combined" approach explained in Sec. 5.3.2, to verify the refinement conditions between the models `xList` and `xListMap`. Fig. 7.22 shows excerpts from the VCC model used to verify this refinement. The gluing invariants in the joint structure of the figure shows how the `xList` state is related to the `xListMap` state when the list is of type `PQ`. The figure also shows the skeleton of the joint method used to verify the refinement condition for the list API *vListInsert*.

VCC verifies that each of the methods in the `xList` data-structure refines the corresponding method in the `xListMap` data-structure.

7.5.5 Handling shared data and proving termination

The kernel data in the FreeRTOS scheduler, like the task lists, currently running task etc are defined as global variables and hence both the scheduler and the `xList` data-structure can modify these data. Hence we need to prove that the communication between the scheduler and the `xList` data-structure is *effectively functional* (see Sec. 5.5). We applied the technique explained in Sec. 5.5 to prove this.

We classified the ownerships of the shared data maintained by FreeRTOS, between the scheduler and the `xList` data-structure. Then by using an `ensures` annotation in each of the method in the FreeRTOS implementation, we proved that the communication between the scheduler and the `xList` data-structure is effectively functional. In particular, we used the annotation `_(ensures \old(x) = x)`, in each method, where the variable `x` of this annotation in component is assumed to be a shared data owned by the other component.

We applied the technique explained in Sec. 5.4, to prove terminations for the methods with loops in the scheduler and the `xList` data-structure.

7.5.6 Verification effort involved

The verification effort involved in this case-study is shown in Fig. 7.23. It shows the number of lines of code involved at the different layers of the strategy used to verify the scheduler-related functionality of FreeRTOS. The numbers reported exclude comments and blank lines.

The high-level Z model $\mathcal{M}$ comprises 50 schemas in the Z language, some of them are data-schemas and others are operation-schemas. We followed the Z convention of modeling the states of a system using a number of sub-schemas to make the model simple and readable. We also use a number of operation-schemas to model an API such that two operation-schemas modeling an operation differ in their preconditions. For instance, the operation-schema for delaying a task when the task needs to be added to `overflow-delayed-list` has a different precondition than the operation-schema which needs to add the running task to the `delayed-list`. There are 766 lines of code (or predicates) in the Z model $\mathcal{M}$.

The low-level Z model $\mathcal{M}_1$ includes 60 schemas and there are 1239 lines

Z Model $\mathcal{M}$		Z Model $\mathcal{M}_1$		API functions in $\mathcal{P}_1$		
Schemas	LOC	Schemas	LOC	Funcs.	LOC	LOA
50	766	60	1239	11	377	2347

`xListMap`	`xList`		
lines of ghost code	Funcs.	LOC	LOA (`xListJoint`)
1339	15	121	1450

Figure 7.23: Size of artifacts in the verification of the FreeRTOS scheduler (LOC stands for Lines of Codes and LOA for Lines of Annotation).

of predicates in it. As we already explained in Sec. 7.5.2, we verified the refinement between the models $\mathcal{M}$ and $\mathcal{M}_1$ by translating these models in to ghost models in VCC. There is an inevitable blow-up of around 10x in the number of specification lines while going from Z to VCC. The reason for this blow-up is that VCC does not support many data-types such as sequences and operators that Z supports and hence one would need to use many lines of specifications to encode a Z object like a sequence.

The port-independent layer of FreeRTOS comprises 2514 lines of code, which include the scheduler-related functionality and the APIs for inter-process communication and synchronization. The scheduler-related functionality is included in the C file `task.c` which contains 30 APIs, many of which deal with tracing and other non-core functionality. The core functionality of the scheduler is implemented by 11 APIs and it comprises 377 lines of C code. This is the lines of code in the model $\mathcal{P}_1$. We used 2347 lines of annotations in VCC to verify these 11 APIs.

The `xList` data-structure includes 15 APIs and comprises 121 lines of C code. As we already explained, a ghost version of this data-structure namely `xListMap` is used to verify the correctness of the model $\mathcal{P}_1$. The `xListMap` data-structure comprises 1339 lines of ghost code. We used our "combined" approach of Sec. 5.3.2 to verify that `xList` refines `xListMap` and there we used 1450 lines of ghost code in the joint model.

7.6 Bugs found

We now report the bugs found in the course of our verification exercise in addition to the previously mentioned problem with `xTaskCreate`. Another related problem is that if the application creates tasks "`A1`" followed by "`B1`", and then starts the scheduler, the task that runs is "`B1`" (instead of "`A1`"). This is due to a problem with the way the `pxCurrentTCB` is updated. In particular, the comparison operator "`<=`" is used in the guard to update the value of `pxCurrentTCB` rather than the required comparison operator "`<`". We fixed this problem by appropriately changing the comparison operator in this guard.

A more serious bug was in the `vTaskPrioritySet` method which changes the priority of a given task. The issue here is that when the given task is in a blocked queue (say waiting to receive a message from a message queue), its priority is updated but its *position* in the waiting queue (which is a priority

queue) is *not* adjusted. This has the following consequence. Let "A2" and "B1" be two tasks waiting to receive a message from a message queue "mq". By our naming convention, the tasks "A2" and "B1" have priorities 2 and 1 respectively. Thus the waiting queue is $\langle \texttt{A2}, \texttt{B1} \rangle$ with "A2" being the higher priority task, at the front of this queue. Now suppose the running task performs the following sequence of operations. Firstly it increases the priority of the waiting task "B1" to 3 by invoking the `vTaskPrioritySet` API. Secondly it sends a message to the message queue `mq`. The first operation, changes the priority of the task "B1" to 3 and thus the waiting queue becomes $\langle \texttt{A2}, \texttt{B3} \rangle$. The second operation causes the scheduler to unblock the waiting task at the front of the waiting queue which is "A2", instead of the higher priority waiting task "B3". This is a violation of a guarantee given by FreeRTOS that "the longest waiting highest priority task will be unblocked when a message arrives in the queue" (see page 50 in the FreeRTOS user guide [10]).

Mutexes are provided in FreeRTOS for inter-process synchronization. A situation where a higher priority task is made to wait for a mutex which is held by a lower priority task is called *priority inversion*. FreeRTOS implements a scheme called *priority inheritance* to minimize the negative effects of priority inversion. The idea is to temporarily increase the priority of the mutex holder to the priority of the task requesting the mutex so that the mutex holder can complete sooner and hence can minimize the waiting time for the higher priority task to get the mutex.

The method `vTaskPriorityInherit` implements the priority inheritance scheme in FreeRTOS. In our verification process, VCC failed to prove that this method is a refinement of the corresponding abstract method. The problem here is similar to the problem mentioned above for the `vTaskPrioritySet` API. Consider the example scenario explained in the case of the `vTaskPrioritySet` API. Suppose the task "B2" is holding a mutex while it is waiting for a message to arrive in the message queue `mq`. Now suppose the running task "C3" of priority 3 requests the mutex held by the task "B2". This causes the scheduler to increase the priority of the mutex holder to 3 and hence the waiting queue becomes $\langle \texttt{A2}, \texttt{B3} \rangle$. Thus when a message arrives in the message queue `mq`, the scheduler unblocks the task "A2" instead of the task "B3" and hence it nullifies the intended purpose of the priority inheritance scheme.

The APIs `vTaskPrioritySet` and `vTaskPriorityInherit` in turn call the `xList` method `list_SET_ITEM_VALUE`, which however does not have the desired effect, when the lower priority task is in a blocked queue. A simple fix is to modify these APIs by first removing the concerned node from the blocked queue, update its priority by invoking the method `list_SET_ITEM_VALUE`, and then insert it back in the queue by invoking the list method `vListInsert`.

Another problem we found is the issue with the `vTaskSuspend` API. In our Z models of FreeRTOS, this operation is assumed to suspend a task only when it is ready to execute. We note that the running task is also a "ready" task in FreeRTOS. During the verification process, we found that the implementation does not have such an assumption. This API and the `vTaskResume` API are used to temporarily suspend or resume a task. Consider a situation

Figure 7.24: Example FreeRTOS application.

```
1 void interp( void )
2 {
3    unsigned t;
4    init();
5    xTaskCreate(Task1,"A1",1,...);
6    t = vTaskStartScheduler();
7    _(assert t == 1)
8    t = vTaskDelay(1);
9    _(assert t == 0)
10 }
```

Figure 7.25: Example interpreter for the FreeRTOS application in Fig. 7.24.

where `vTaskSuspend` is used to suspend a blocked or a delayed task. The `vTaskResume` API when invoked to resume this task will take it back to the ready list rather than to the blocked or delayed list from which it was taken out. In particular, a task which is made to delay for a certain time period may execute before the delay period is over if some other task suspends and resumes it in-between. A fix is to allow the suspend operation only when the task to be suspended is *ready*.

We communicated these issues to the developers of FreeRTOS who acknowledged that our understanding of the intended behavior was correct and that the said behaviors were indeed deviations from what they had intended [9]. 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 okay 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 APIs involved only a small part of the code: 19 lines in the API code were modified and 7 lines added to `xList`.

The modified version of the FreeRTOS scheduler can be viewed as a piece of software in which the sequential behaviour of the task-related API's has been formally specified and verified. By "sequential behaviour" we mean that each API behaves correctly in the absence of interleaving with other API's. The verified version of FreeRTOS and other artifacts of our FreeRTOS verification case-study are available in [24].

7.7 Verifying FreeRTOS application

In this section we describe the verification guarantees provided to the FreeRTOS clients. We illustrate this with an example program which interprets a FreeRTOS application.

Fig. 7.24 shows an example FreeRTOS application and Fig. 7.24 shows an interpreter for this application. Consider what happens when this interpreter program executes. Firstly the interpreter invokes the `init` API, which initializes the scheduler and creates the "*idle*" task with task-id `0`. After this the interpreter creates the task "`A1`" by invoking the `xTaskCreate` API and then it invokes the `vTaskStartScheduler` API, which now returns the first task to run which is "`A1`" with task-id `1` (task-ids for user created tasks are assigned in the order of their creations starting from `1`). Now the interpreter invokes the `vTaskDelay` API with argument `1`, which causes the scheduler to return the "*idle*" task with task-id `0` as the next task to run. Thus the interpreter program satisfies the assert statements in it.

Using our ghost implementation $\mathcal{G}$ of FreeRTOS, we proved in VCC that the interpreter program satisfies the assert statements in it. These are local LT properties in the terminology of Sec. 2.5. Now it follows from our notion of refinement, Theorem 2.2 and Theorem 6.2 that the interpreter application with a suitable wrapper (see Sec. 6.3) continues to satisfy this assertion when it calls these operations in the verified version of the FreeRTOS implementation.

Can we infer this property if we use the notion of refinement which allows the concrete ADT to strengthen the precondition of an ADT operation? The answer is "no", because the concrete may strengthen the precondition of some of these APIs and hence can lead to unpredictable behaviors. For example, suppose the concrete implementation strengthens the precondition of the `vTaskDelay` API to require the argument at least `2` units of time. Then anything can happen when `vTaskDelay` API is invoked with argument `1` and hence the second assertion in the interpreter will fail. For instance, suppose the concrete `vTaskDelay` method returns the running task itself as the next task to run when the argument value is less than `2`, which in this case is `1` and hence violates the assertion at line `9` of the interpreter program.

7.8 Related work

We first consider the design-for-verification projects. The most prominent work here is the seL4 project [33], 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 use a refinement-based approach to prove functional correctness of the C implementation with respect to a high-level specification in Haskell. The main difference in our verification is that we did a post-facto verification of an existing software which was designed for efficiency rather than verifiability. For instance the authors of FreeRTOS may not have used a complex data structure like a circular doubly linked-list if they had verification in mind.

Among the work on post-facto verification, the most related is the Verisoft XT project [12] 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 [31],[34]) PikeOS [11] is an embedded OS, similar in nature to FreeRTOS though with a few

more features like virtualization. The verification uses VCC where specifications are annotations and correctness is in terms of conformance to ghost code in a two-tier structure which contains both the concrete implementation and a ghost specification which are related by a coupling relation. However there no explicit notion of refinement and hence leads to some disadvantages that we discussed in Sec. 1.6. In contrast, we have an abstract specification (or model) and give a clear guarantee in terms of conforming to the abstract model.

In [42] the authors verify properties like data-race freedom of a Linux USB keyboard driver using Verifast, but do not address functional correctness.

An open framework called OCAP is proposed by Feng et al in [21], which supports domain-specific verification systems for verifying individual components and also allows inter-operation of different verification systems. Reuse of individual proofs is an important aspect of this framework. This framework is applied in a case-study [22] in which a complete OS kernel is considered in the sense that they consider both the port-specific and port-independent layers of the OS, to verify the implementation of preemptive threads and some synchronization primitives. In contrast we verified the complete functional correctness of the port-independent layer of the FreeRTOS scheduler.

A framework is proposed by Zhaozhong et al in [40] to verify the correctness of three context management functions for the x86 architecture. The requirement is specified as properties in terms of a context structure and the assembly verification framework XCAP is used to verify the correctness. The Coq proof assistant is used to check the verification condition by encoding the XCAP theory, machine model and the functions together with their specifications; in Coq. In our verification exercise we have proved the functional correctness of the scheduler-related functionality of FreeRTOS with respect to an abstract mathematical model of the FreeRTOS scheduler.

Chapter 8

Checking Refinement Conditions Efficiently

In this chapter we propose an efficient way to phrase the refinement conditions in VCC, which considerably improves the performance of VCC. We illustrate this technique with a case-study in which we verify a simplified C implementation of the FreeRTOS scheduler, with respect to its abstract Z specification.

8.1 Motivation

Our aim is to propose a practically efficient technique for carrying out the proofs of successive refinements proposed in our "directed refinement methodology".

We presented two techniques namely "direct-import" approach (of Sec. 5.3.1) and "combined" approach (of Sec. 5.3.2) for checking refinements in VCC. However these approaches suffer from some disadvantages, which we describe in the subsequent paragraphs.

The following are the problems with the "direct-import" approach. 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 requirements need to be specified *directly* on the concrete state. This can be quite complex for both a human and a tool especially when dealing with complex objects like self-referential data-structures.

The problem with the "combined" approach is that it leads to excessive time requirements for discharging the proof obligations when one of the ADT models is an imperative language implementation. This approach may cause the prover to take a lot of time or may even cause the prover to run out of memory. 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. Each extra annotation is required as a loop invariant (or as a function contract), to specify that each ghost (or abstract) object in the system is kept unmodified by a loop (or a function) that modifies a concrete data object. We illustrate this problem with the following example.

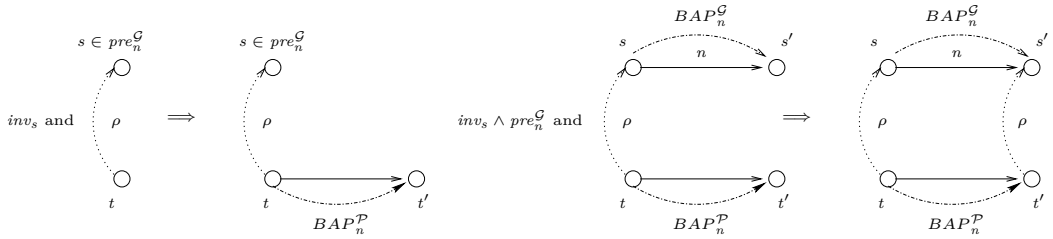
```
strcut                              int deq()
{                                     _(requires QG.lenG > 0)
   _(ghost \natural lenG)             _(ensures lenG <= SIZE)
   _(ghost int seq[\natural])       {
   _(invariant lenG <= SIZE)           int resG,
} QG;                                  _(assume resG == QG.seq[0])
                                       _(ghost QG.seq = \lambda \natural i;
                                          QG.seq[i + 1])
                                       _(ghost QG.lenG = QG.lenG - 1)
                                       return resG;
                                    }
```

Figure 8.1: A part of the ghost implementation of an ADT of type $QType_{\mathbb{Z}}$.

```
strcut                          int deq()
{                                 ...
   unsigned lenC;                 _(ensures lenC <= SIZE)
   int arr[SIZE];               {
   _(invariant lenC <= SIZE)       int resC,i;
} QC;                              resC = QC.arr[0];
                                   for(i = 0; i < (QC.lenC - 1); i++)
                                    ...
                                   {
                                      QC.arr[i] = QC.arr[i + 1];
                                   }
                                   QC.lenC--;
                                   return resC;
                                }
```

Figure 8.2: A part of the C implementation of an ADT of type $QType_{\mathbb{Z}}$.

Consider an ADT of type $QType_{\mathbb{Z}}$, which we discussed in Sec. 3.1. Let `g-queue` be a ghost implementation of an ADT of type $QType_{\mathbb{Z}}$, a part of which is shown in Fig. 8.1 and let `c-queue` be a C implementation of an ADT of this type, a part of which is shown in Fig. 8.2. The method `deq` in `c-queue` uses a for loop to update the state of the array. Verifying methods with loops is a difficult task in VCC. This is because of the fact that VCC forgets its knowledge about the state of the method (or program) when it enters a loop. Hence it is the responsibility of the user to provide suitable loop invariants, which are necessary to prove the postcondition of a method. For instance, in the "combined" approach for proving the refinement between `g-queue` and `c-queue`, in terms of the *deq* operation, the user should provide the following loop invariant to ensure that the ghost components are not modified by the loop body.



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

Figure 8.3: Illustrating the conditions checked in the two-step approach.

```
(lenG == \old(lenG)) &&
(\forall \natural i; (i < lenG) ==> (seq[i] == \old(seq[i])))
```

The above loop invariant is required regardless of the order in which the combined method executes the bodies of the abstract and concrete methods. In one case, this invariant is required to ensure that the modified state of the abstract model is preserved by the loop. In the other case, it is required to ensure that the initial state of the abstract model is preserved by the loop.

A second possibility in the "combined" approach is to use method invocations in the joint method rather than directly executing the abstract and concrete method bodies. However one would need to use more annotations in this technique, than the above, since the function contract of the method in one model should include a similar predicate to ensure the preservation of the state of the other model.

We fix the models `g-queue` and `c-queue` as above for the rest of this chapter.

8.2 Proposed efficient approach

We now propose an efficient approach called the "two-step" approach, which overcomes the difficulties mentioned above. The idea is to divide the refinement checking task into two steps. The first step is to prove the BAPs for the abstract and concrete methods separately, by manually supplying the BAPs. Recall that the BAP of an operation is a predicate representing the state change induced by the operation. The second step is to prove the following: (i) the after-states represented by the abstract and concrete BAPs satisfy the gluing invariant and (ii) the abstract and concrete BAPs represent the same output value.

Fig. 8.3 illustrates the two steps required in our approach to prove the refinement between an abstract ADT model $\mathcal{G}$ and a concrete ADT model $\mathcal{P}$. It shows Step 1 of the "two-step" approach only for the concrete method. A similar BAP check is required to be performed on the abstract method.

$op^{\mathcal{G}}$( )
  _(requires $inv_s^{\mathcal{G}} \wedge pre_{op}^{\mathcal{G}}$)
  _(ensures $BAP_{op}^{\mathcal{G}}$)

{
  // body of $op^{\mathcal{G}}$
}

(a)

$op^{\mathcal{P}}$( )
  _(requires $inv_s^{\mathcal{G}} \wedge pre_{op}^{\mathcal{G}} \wedge inv_\rho$)
  _(decreases 0)
  _(ensures $BAP_{op}^{\mathcal{P}}$)
{
  // body of $op^{\mathcal{P}}$
}

(b)

Figure 8.4: (a): Step 1 of the "two-step" approach for proving the abstract BAP and (b): Step 1 of the "two-step" approach for proving the concrete BAP.

```
int deqStep1Abstract()
  _(requires (QGC.lenG <= SIZE) && (QGC.lenG > 0))
  _(ensures
    (QGC.lenG == (\old(QGC.lenG) - 1)) &&
    (\forall \natural i;
       (i < QGC.lenG) ==> (QGC.seq[i] == \old(QGC.seq[i + 1]))))
{
   // body of the abstract method
}
```

Figure 8.5: Illustrating the Step 1 of the "two-step" approach for proving the abstract BAP $BAP_{deq}^{\texttt{g-queue}}$.

Fig. 8.4 shows the skeletons of the methods in VCC to prove the abstract and concrete BAPs for an ADT operation op. Here the notations: "$inv_s^{\mathcal{G}}$", "$pre_{op}^{\mathcal{G}}$", "$BAP_{op}^{\mathcal{B}}$" and "inv_ρ" respectively denote - the invariant on the state-structure of the abstract ADT, the precondition of the method in the abstract ADT corresponding to the operation op, the BAP of the operation op in the model $\mathcal{B}$, and the gluing invariant, which relates the abstract and concrete states.

For example, consider the ADT models `g-queue` and `c-queue` discussed above. Step 1 of the "two-step" approach for proving the abstract BAP, $BAP_{deq}^{\texttt{g-queue}}$ for the deq method in the model `g-queue` is shown in Fig. 8.5. Fig. 8.6 shows Step 1 of the "two-step" approach for proving the concrete BAP, $BAP_{deq}^{\texttt{c-queue}}$ for this method in the model `c-queue`. The `decreases` annotation ensures that the method terminates (see Sec. 5.4).

The problem with the "direct-import approach with quantifier elimination" is solved, since manually supplying the BAPs for the abstract and concrete methods is more transparent and easy to do than specifying the requirements completely on the concrete data-structures, by manually translating the refinement conditions to eliminate existential quantifications. The problem with

```
int deqStep1Concrete()
  _(requires (QGC.lenG <= SIZE) && (QGC.lenG > 0) &&
     ((QGC.lenG == QGC.lenC) && (\forall \natural i;
         (i < QGC.lenG) ==> (QGC.seq[i] == QGC.arr[i]))))
  _(decreases 0)
  _(ensures
    (QGC.lenC == \old(QGC.lenC) - 1) &&
    (\forall \natural i;
      (i < QGC.lenC) ==> (QGC.arr[i] == \old(QGC.arr[i + 1])))
{
   // body of the concreet method
}
```

Figure 8.6: Illustrating the Step 1 of the "two-step" approach for proving the concrete BAP $BAP^{\texttt{c-queue}}_{deq}$.

the "combined" approach is also solved, since 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 use the set of predicates in a loop (or a concrete function contract), which is otherwise required to ensure that the abstract state is not modified by a loop (or a method).

The second step of the "two-step" approach is to check the validity of the following implication, whose RHS specifies that the after-states defined by the abstract and the concrete BAPs are related by the gluing relation and also that the abstract and the concrete operations produce the same output.

$$inv^{\mathcal{G}}_{s} \wedge pre^{\mathcal{G}}_{op} \wedge inv_{\rho} \wedge BAP^{\mathcal{G}}_{op} \wedge BAP^{\mathcal{P}}_{op} \implies inv'_{\rho} \wedge res^{\mathcal{G}} = res^{\mathcal{P}}.$$

In the above implication, the notation "inv'_{ρ}" denotes the gluing invariant in the joint after-state. One can use a method with suitable function contract in VCC to check the validity of a predicate like this. For instance, Fig. 8.7 shows the method which proves Step 2 of the "two-step" approach for proving the refinement between the models `g-queue` and `c-queue` in terms of the *deq* operation in the ADT type $QType_{\mathbb{Z}}$. Firstly the method assumes the part of the LHS of the implication, which is about the joint before-state and input. Then `havoc` is invoked to inform VCC that the abstract and concrete states are changed in some way. This is followed by an annotation which assumes the part of the LHS of the implication which represents the BAPs of the abstract and concrete methods. The RHS of the implication is specified as the `ensures` annotation in the method.

```
int deqStep2()
  _(ensures
    ((QGC.lenG == QGC.lenC) && (\forall \natural i;
         (i < QGC.lenG) ==> (QGC.seq[i] == QGC.arr[i]))) &&
    (resG == resC))
{
   _(assume
     (QGC.lenG <= SIZE) && (QGC.lenG > 0) &&
       ((QGC.lenG == QGC.lenC) && (\forall \natural i;
        (i < QGC.lenG) ==> (QGC.seq[i] == QGC.arr[i]))))
   havoc();
   _(assume
     ((QGC.lenG == (\old(QGC.lenG) - 1)) &&
       (\forall \natural i;
        (i < QGC.lenG) ==> (QGC.seq[i] == \old(QGC.seq[i + 1])))) &&
       ((QGC.lenC == \old(QGC.lenC) - 1) &&
        (\forall \natural i;
        (i < QGC.lenC) ==> (QGC.arr[i] == \old(QGC.arr[i + 1])))))
}
```

Figure 8.7: Illustrating the Step 2 of the "two-step" approach for the operation *deq*.

8.3 Case-study: `Simp-Sched`

In this section, we describe a case-study in which we apply our efficient refinement checking technique, to prove the functional correctness of an existing ADT implementation. We also report a comparison of the relative performance of the approaches: "direct-import with quantifier elimination", "combined" and "two-step"; in terms of this case-study.

FreeRTOS implementation uses a linked-list data-structure to implement the task lists maintained in the kernel, also it uses a pointer to a TCB struct to represent a task. We wanted to avoid the complexity of dealing with heap data-structures and pointers in this case-study, since here we aim to evaluate the performance improvement of the "two-step" approach over the other two approaches. Hence we decided to work with a simplified version of the FreeRTOS scheduler called `Simp-Sched`, which we constructed for this verification exercise. The operations provided by the `Simp-Sched` API and the operations in its sub-ADT called *list* are shown in Fig. 8.8. `Simp-Sched` maintains all key aspects of timing and scheduling in FreeRTOS. The simplification is with respect to the following two aspects:

1. A task in the FreeRTOS scheduler is maintained in a struct called a TCB, which includes pointers to function behavior. In `Simp-Sched`, we simply use an integer *task-ID* to represent the TCB of a task.

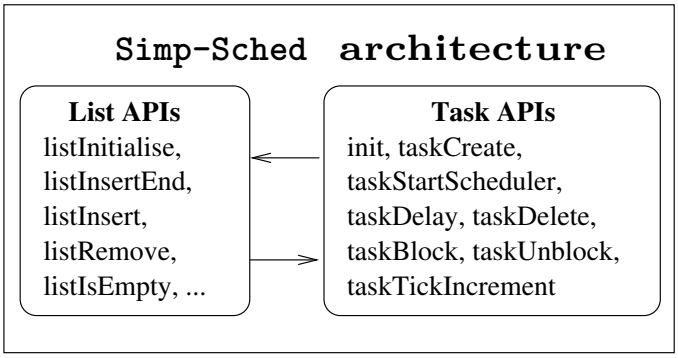


Figure 8.8: Components in the scheduler implementation

2. Each task list like *ready* and *delayed* is maintained using the `xList` data-structure, which is implemented as a circular doubly linked-list in FreeRTOS. In `Simp-Sched`, we replace this data-structure with an array-based list implementation.

All other aspects of the FreeRTOS scheduler implementation are maintained. We use the high-level Z model, which is used for the verification of the FreeRTOS scheduler for verifying `Simp-Sched`. Given this, one of the key uses of our `Simp-Sched` implementation is the use in a run time monitor that can be used to identify potential scheduling inconsistencies and errors in the FreeRTOS scheduler. Each method in FreeRTOS can be instrumented to include a call to the corresponding method 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 [20]. The task lists are implemented as a separate library in which lists are implemented using arrays in C.

8.3.1 Proving the functional correctness of `Simp-Sched`

We now describe the technique used for proving the the functional correctness of the scheduler: `Simp-Sched`. We started with the high-level Z model that we used to prove the functional correctness of the FreeRTOS scheduler (Sec. 7.5.1). Then we proved that the C implementation of `Simp-Sched` is a refinement of the high-level Z model.

The strategy used here is similar to the strategy of Sec. 7.4, which is used to verify the functional correctness of the FreeRTOS scheduler, except that here we use our "two-step" approach when one of the ADT models is a C implementation. To avoid repetition, we do not give the details of this case-study. Nonetheless we describe the technique in terms of the methodology used for verifying `Simp-Sched`. The methodology is shown in Fig. 8.9, which involves six stages:

1. We start with the high-level Z model $\mathcal{M}$, which is used to prove the functional correctness of the FreeRTOS scheduler.

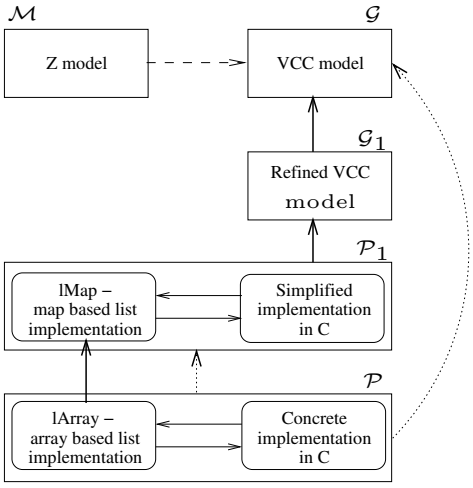


Figure 8.9: Overview of the verification of `Simp-Sched`. Dashed arrow denotes the Z-to-VCC translation, bold upward-arrows denote the refinements to be proved and dotted arrows denote the refinements which follows from our transitivity or substitutivity results.

2. The next step in the methodology is to apply our mechanizable translation procedure explained in Sec. 5.2.3, to translate $\mathcal{M}$ to a ghost model in VCC, which we call $\mathcal{G}$. Recall that our translation guarantees that $\mathcal{G}$ refines $\mathcal{M}$.

3. We know that the implementation, $\mathcal{P}$ of `Simp-Sched` uses the operations from the `lArray` sub-ADT to implement the task lists maintained in `Simp-Sched`. Thus the existing implementation is of the form $\mathcal{P}$[`lArray`]. We now obtain a simplified implementation $\mathcal{P}_1$ of `Simp-Sched` from $\mathcal{P}$ by replacing the sub-ADT `lArray` in $\mathcal{P}$ with a map-based sub-ADT called `lMap`. Thus the simplified implementation is of the form $\mathcal{P}_1$[`lMap`].

4. We then refine the high-level ghost model $\mathcal{G}$ to a low-level ghost model called $\mathcal{G}_1$ to capture some implementation details in $\mathcal{P}_1$. For instance, the system clock is unbounded in $\mathcal{G}$. On the other hand, the system clock is implemented as a bounded variable in $\mathcal{P}_1$ whose 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 set of delayed tasks, which is maintained in a single delayed list in $\mathcal{G}$, is implemented as two task lists in $\mathcal{P}_1$ to cope with the bounding of the clock value. We refine the model $\mathcal{G}$ to the low-level model $\mathcal{G}_1$ to capture these changes. We use the "combined" approach explained in Sec. 5.3.2, to prove that the low-level ghost model $\mathcal{G}_1$ refines the high-level ghost model $\mathcal{G}$.

5. The next step in the methodology is to prove the refinement between $\mathcal{G}_1$ and $\mathcal{P}_1$. We prove this refinement by applying our "two-step" approach explained in Sec. 8.2.

Model	Lines of code	Lines of annotation
$\mathcal{M}$	-	222
$\mathcal{G}$	1317	1580
$\mathcal{G}_1$	1954	2287
$\mathcal{G}_1 \preceq \mathcal{G}$	3271	741
$\mathcal{P}_1$	609	293
$\mathcal{P}_1 \preceq \mathcal{G}_1$	24	7639
`lMap`	240	602
`lArray`	104	56
`lArray` $\preceq$ `lMap`	20	837
$\mathcal{P}$	769	-

Table 8.1: Code metrics and annotation effort involved in the verification of `Simp-Sched`.

6. The final step in the methodology is to prove that $\mathcal{P}$ refines $\mathcal{P}_1$. For this we first prove by using our "two-step" approach that `lArray` refines `lMap`. Then we infer using Theorem 3.2 (refinement is substitutive), that $\mathcal{P}$ refines $\mathcal{P}_1$.

Now it follows from Proposition 2.1 (transitivity of refinements), that $\mathcal{P}$ refines $\mathcal{G}$. Recall that our mechanizable translation procedure of Sec. 5.2.3, translates the Z model $\mathcal{M}$ to the ghost model $\mathcal{G}$ such that $\mathcal{G}$ refines $\mathcal{M}$. Hence it follows that the C implementation of `Simp-Sched`, $\mathcal{P}$ refines the Z model $\mathcal{M}$. The verification artifacts from this case-study are available at [20].

8.3.2 Code metrics and human effort involved

We spent two human-months to obtain the implementation `Simp-Sched` from a FreeRTOS implementation and to verify the functional correctness of this simplified implementation with respect to the high-level Z model $\mathcal{M}$ for FreeRTOS. The code metrics are given in Table 8.1. Even though there are around 22500 lines of code/annotations, there are only a few lines of modifications required in successive refinements and hence the size of the high-level ghost model $\mathcal{G}$ and the `lMap` model, which is extracted from $\mathcal{G}$, are the important parts deciding the human effort required. The size of $\mathcal{G}$ and `lMap` comes to 2422 lines of annotations is VCC and that is about 3 times the size of the executable code $\mathcal{P}$.

8.3.3 Performance comparison

In this section we report the time taken by VCC to prove the refinement conditions between the different models in this case-study. Table 8.2 shows the time taken under three different approaches namely "direct-import with quantifier elimination", "combined" and "two-step" approaches described earlier. On an average, 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

Sl.No.	API	Time taken by VCC (in seconds)				
		"direct-import"	"combined"	"two-step"		
				Step 1	Step 2	Total
$\mathcal{P}_1 \preceq \mathcal{G}_1$	*init*	**257**	**89**	231	4	**235**
	taskCreate	**357**	**781**	9	4	**13**
	taskStartScheduler	**10**	**14**	5	4	**9**
	taskDelay	**285**	**18773**	22	8	**30**
	taskDelete	**436**	**18391**	68	8	**76**
	taskBlock	**423**	**20699**	22	5	**27**
	taskUnblock	**227**	**16838**	27	6	**33**
`lArray` $\preceq$ `lMap`	*listInitialise*	**2**	**3**	2	2	**4**
	listGetNumberOfElements	**2**	**2**	2	2	**4**
	listIsEmpty	**2**	**2**	4	2	**6**
	listIsContainedIn	**2**	**2**	3	4	**7**
	listGetIDofFirstFIFOtask	**3**	**2**	2	3	**5**
	listGetIDofFirstPQtask	**2**	**3**	3	4	**7**
	listGetKeyOfFirstPQtask	**2**	**3**	2	2	**4**
	listInsertEnd	**2**	**3**	2	5	**7**
	listInsert	**32**	**9**	3	2	**5**
	listRemove	**4448**	**43**	4	2	**6**
Total time taken by each technique		**6493**	**75658**			**490**

Table 8.2: Time taken by VCC to prove refinement conditions under the different techniques.

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 the statements in the abstract methods, in addition to the overhead involved in the "direct-import" approach.

This case-study shows that our "two-step" approach consistently improves the performance of VCC for checking refinement between complex models. This is evident from the time savings that we achieved for checking the refinement between the task APIs. However when the concrete model is very short like the `lArray` model which contains less than 10 lines of C code in each method, the other approaches performs better than the "two-step" approach. This is because of the fact that in such a case the time savings possible in Step 2 is less than the time requirement for Step 1.

Chapter 9

Conclusion and Future Work

In this thesis we have proposed a methodology to carry out mechanized proofs of functional correctness of Abstract Data Type (ADT) implementations.

The methodology is based on a natural notion of refinement in which we require the concrete ADT to support all the exception-free sequences of operations that the abstract ADT allows. A client program of an ADT is viewed as interacting with the ADT in a "purely functional" way by making operation calls and using the return values. In this setting we spell out the properties preserved in a client when an abstract ADT is replaced by a refined version. We prove a substitutivity result that is convenient while reasoning about complex implementations of ADTs that use sub-ADTs to realize their functionality.

We explore this methodology using mainly the VCC tool as a vehicle. We show how to represent abstract versions of ADTs using auxiliary ghost constructs in VCC and reason about refinement within VCC. Though we have to deal with issues like ownership in VCC, on the whole VCC provides a powerful and convenient environment for executing our methodology.

We have evaluated our methodology using a couple of case-studies centered around a popular open-source embedded operating system, FreeRTOS. In the first, more elaborate, case-study we use our methodology to prove the functional correctness of the FreeRTOS scheduler. In this process we uncovered significant errors in the implementation, which we had to fix for the verification to go through. As an end product we have a version of the FreeRTOS scheduler with a mathematically precise specification for its sequential behaviour and a machine checked proof of conformance to that specification. As a proof of concept we also show how we can efficiently prove desirable properties of FreeRTOS applications using the abstract specification.

In the second case-study we investigate different ways of phrasing the refinement conditions of our theory in a simplified version of FreeRTOS. We show that a certain two-step approach to phrase these conditions leads to significant saving in time and memory that VCC spends to check these conditions.

9.1 Future work

There are several interesting directions we would like to pursue as future work.

In our refinement theory we assumed that the execution of a program which implements an ADT is sequential, in the sense that the execution of each method when started will run to completion before another method can start execution. However this assumption is not satisfied by most of the programs even in a single processor architecture. For instance, an interrupt can start execution causing another API to run before the currently executing method completes. We would like to extend our theory to handle interleaved and parallel executions of the methods in a program, which implements an ADT. Here one could use a method in VCC like the one proposed by Cohen et al [17] or the notion of linearizability proposed by Herlihy and Wing [27] to achieve this.

The verification in the FreeRTOS case-study is completed with the assumption that the execution is sequential. However FreeRTOS supports a limited form of interleaved execution. For instance, a tick interrupt can cause the scheduler to execute immediately by preempting the currently running task while it is executing in the middle of a FreeRTOS API. Some synchronization primitives like disabling interrupts are used in FreeRTOS to ensure that shared data is accessed/updated in a safe way in an interleaved execution. We would like to make use of the built-in support of VCC to verify the FreeRTOS implementation by considering the restricted form of interleaved executions allowed. To do this one would need to see how the synchronization primitives in FreeRTOS can be encoded in VCC to use its built-in support for reasoning about concurrency.

One aspect of the scheduler functionality in FreeRTOS that we have not addressed in this thesis inter-process communication and synchronization through message queues and semaphores. An interesting future direction is to verify the operations for inter-process communication and synchronization. One could use the VCC tool to carry out the steps proposed in our methodology to verify this, in a similar way as we did for verifying the scheduler-related functionality. Another direction is to address the correctness of the port-specific layer of FreeRTOS.

The stand-alone ghost model of the FreeRTOS scheduler that we developed for proving the scheduler-related functionality can be used together with our theory of refinement to prove local linear time properties about FreeRTOS applications. We have proved such properties about few simple FreeRTOS applications which we developed. A future direction is verifying interesting local linear time properties about real FreeRTOS applications using the ghost model of FreeRTOS from our case-study.

Our translation procedure from the Z language to VCC's ghost language is not automated. Also it covers only a part of the Z language, that we used in our case-studies. This basically uses *relations* and *finite sequences* with operations on them. A possible future direction is to consider more Z objects in the translation procedure and to automate the translation procedure.

VCC's ghost language does not support abstract mathematical objects like sets and sequences, which are provided by specification languages like Z. Such mathematical objects are essential to ease the task of specification. A future

work along this line is to extend VCC's ghost language to support the essential mathematical objects required in a specification language. One could use a technique similar to the one in our Z-to-VCC translation procedure to achieve this.

In our opinion, a simulator in VCC may help the user to make the modeling phase easy. Hence we would like to have an option in VCC to simulate the ghost model, like the ProZ animator for the Z language, which is supported by the tool ProB. One could do this by translating the ghost objects into equivalent objects in C and hence by generating an equivalent C program which could be executed in the background to simulate the ghost model.

Bibliography

[1] C.A.R. Hoare, I.J. Hayes, Jifeng He, C.C. Morgan, J.W. Sanders, I.H. Sorensen, J. M. Spivey and B.A. Sufrin. Data refinement refined. Typescript, Programming Research Group, Oxford University., May 1985.

[2] Jean-Raymond Abrial. Data semantics. In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010. `http://dx.doi.org/10.1007/s10009-010-0145-y`.

[5] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. 1980.

[6] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. 1980.

[7] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In *3rd Intl Conf on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 40–54, Edinburgh, UK, 2010. Springer.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[9] Richard Barry. personal communication.

[10] Richard Barry. **U**sing the FreeRTOS Real Time Kernel – A Practical Guide. 2010.

[11] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *SSV*, volume 102 of *EPTCS*, pages 18–32, 2012.

[12] Bernhard Beckert and Michal Moskal. Deductive Verification of System Software in the Verisoft XT Project. *KI*, 24(1):57–61, 2010.

[13] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[14] Ernie Cohen. Data abstraction in vcc. pages 79–114.

[15] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, pages 23–42, 2009.

[16] Ernie Cohen, Mark A. Hillebr, Stephan Tobies, MichaÅĆ Moskal, and Wolfram Schulte. Verifying c programs: A vcc tutorial, 2011.

[17] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV'10*, pages 480–494, 2010.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[19] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, LNCS, Boston, July 2004. Springer Verlag. Tool descript.

[20] Efficient refinement check in VCC. Project artifacts. `www.csa.iisc.ernet.in/~deepakd/SimpSched`, 2014.

[21] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In FranÃğois Pottier and George C. Necula, editors, *TLDI*, pages 67–78. ACM, 2007.

[22] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Combining domain-specific and foundational logics to verify complete software systems, 2008.

[23] Ivana Filipovic, Peter W. O'Hearn, Noah Torp-Smith, and Hongseok Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Asp. Comput.*, 22(5):547–583, 2010.

[24] FreeRTOS verification project. Project artifacts. `www.csa.iisc.ernet.in/~deepakd/FreeRTOS`, 2014.

[25] RESOLVE Software Research Group. The FreeRTOS Project. http://resolve.cs.clemson.edu/interface.

[26] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 1986.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[28] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[29] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 08 2008.

[30] Clifford B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[31] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[32] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.

[33] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.

[34] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *LNCS*, pages 806–809, Eindhoven, 2009. Springer.

[35] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[36] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[37] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[38] Real-Time Engineers Pvt Ltd. The FreeRTOS Project. `www.freertos.org`, Cited 10 April 2012.

[39] Ivana Mijajlovic, Noah Torp-Smith, and Peter W. O'Hearn. Refinement and separation contexts. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 421–433. Springer, 2004.

[40] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 189–206, 2007.

[41] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[42] Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. Sound formal verification of linux's usb bp keyboard driver. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 210–215. Springer, 2012.

[43] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[44] Mark Saaltink. The Z/Eves System. In *ZUMâĂŹ97: Z Formal Specification Notation*, pages 72–85. Springer-Verlag, 1997.

[45] Yannick Welsch and Arnd Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Sci. Comput. Program.*, 92:129–161, 2014.

[46] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, 1996.

Appendix A

Non-deterministic ADTs and Refinement

In this appendix we propose an extension to our refinement theory to handle non-deterministic ADTs. Non-determinism gives more flexibility in abstract specification. For example the scheduler of a simple operating system, where tasks are of the same priority can be specified to schedule any one of the set of ready tasks and thus have the freedom of resolving this choice at a later stage in refinement. We propose a notion of refinement for non-deterministic ADTs which preserves the same set of properties preserved by our refinement notion for deterministic ADTs.

A.1 Non-deterministic ADTs

We consider an ADT type called *SchedType2* and an ADT of this type to explain the notion of refinement for non-deterministic ADTs. Fig. A.1, shows the ADT type *SchedType2*. The type of a task is assumed as $\mathbb{N}$, the set of natural numbers. There is no argument to the *init* operation and hence is assumed to take the dummy argument *nil*. The *init* operation is expected to return *ok* when the operation is successful. The operation *create* takes the task to be inserted to the ready list as an argument. The *create* operation is expected to return *ok*, when it succeeds in inserting the given task to the ready list and it is expected to return the value *fail*, when it cannot insert the given task to the ready list, may be due to the unavailability of a vacant space in the ready list. The operation *resched* takes the currently running task as an argument. The *resched* operation is expected to return a task as the next task to run, when it succeeds in: (i) inserting the given task to the ready list and (ii) extracting a task from the ready list. It can also return the value *fail*, when it cannot successfully complete its operation, may be due to the unavailability of a vacant space in the ready list to insert the given task or due to the unavailability of a task to extract from the ready list.

Definition A.1 (NADT)**.** *A (non-deterministic)* ADT *of type* $\mathcal{N}$ *can be defined as a 4-tuple:* $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ *similar to a deterministic ADT,*

$SchedType2 = \{init, create, resched\}$ where:

$$I_{init} = \{nil\} \qquad O_{init} = \{ok, e\}$$
$$I_{create} = \mathbb{N} \qquad O_{create} = \{ok, fail, e\}$$
$$I_{resched} = \mathbb{N} \qquad O_{resched} = \mathbb{N} \cup \{fail, e\}$$

Figure A.1: The ADT type *SchedType2*.

$$\begin{array}{lcl}
SchedADT2 & = & (Q, U, E, \{op_n\}_{n \in SchedType2}) \text{ where} \\
Q & \subseteq & 2^{\mathbb{N}} \cup \{E\} \\
op_{init}(q, nil) & = & \begin{cases} \{(\epsilon, ok)\} & \text{if } q \neq E \\ \{(E, e)\} & \text{otherwise.} \end{cases} \\
op_{create}(q, a) & = & \begin{cases} \{(q \cup \{a\}, ok)\} & \text{if } q \neq E \\ \{(E, e)\} & \text{otherwise.} \end{cases} \\
op_{resched}(q, a) & = & \begin{cases} \{(q' \cup \{a\}, b)\} & \text{if } q \neq e \text{ where } b \in q \text{ and } q' = q \setminus \{b\}. \\ \{(E, e)\} & \text{otherwise.} \end{cases}
\end{array}$$

Figure A.2: An NADT $SchedADT2$ of type *SchedType2*.

except that for each operation n in N, op_n is now a non-deterministic realization of the operation n: $op_n \subseteq (Q \times I_n) \times (Q \times O_n)$ satisfying the following conditions:

1. *if $(q, e) \in op_n(p, a)$ then $q = E$,*
2. *if $(E, e) \in op_n(p, a)$ and $(q, b) \in op_n(p, a)$ then $q = E$ and $b = e$, and*
3. *$op_n(E, -) = \{(E, e)\}$.*

Thus if an operation returns the exceptional value the ADT moves to the exceptional state E, and all operations must keep it in E thereafter. Also if an operation can return the exception value, then it cannot return any other value.

Fig. A.2 shows an example ADT called $SchedADT2$ of type *SchedType2*. The state set Q models the set of ready tasks in the operating system. It also contains the exception state E. The *resched* operation non-deterministically removes a task, inserts the given (currently running) task into the set of ready tasks and returns the removed task as the next task to run .

A.2 Refinement between NADTs

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ be an NADT of type $\mathcal{N} = (N, (I_n)_{n \in N}, (O_n)_{n \in N})$. We note that $L_{init}(\mathcal{A})$ represents the language of initialized sequences of operation calls allowed by an ADT $\mathcal{A}$.

$$
\begin{array}{lcl}
SchedADT2' & = & (Q, U, E, \{op_n\}_{n \in SchedType2}) \text{ where} \\
Q & = & \{\epsilon\} \cup \bigcup_{i=1}^{\infty} \mathbb{N}^i \cup \{E\} \\
op_{init}(q, nil) & = & \begin{cases} \{(\epsilon, ok)\} & \text{if } q \neq E \\ \{(E, e)\} & \text{otherwise.} \end{cases} \\
op_{create}(q, a) & = & \begin{cases} \{(q \cdot a, ok)\} & \text{if } q \neq E \\ \{(E, e)\} & \text{otherwise.} \end{cases} \\
op_{resched}(q, a) & = & \begin{cases} \{(q' \cdot a, b)\} & \text{if } q \neq E \text{ and } q = b \cdot q' \\ \{(E, e)\} & \text{otherwise.} \end{cases}
\end{array}
$$

Figure A.3: An ADT $SchedADT2'$ refining the $SchedADT$ of Fig. A.2.

Definition A.2 (Refinement)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be NADTs of type $\mathcal{N}$. We say $\mathcal{B}$* refines *$\mathcal{A}$, written $\mathcal{B} \preceq \mathcal{A}$, iff they satisfy each of the following conditions:*

1. *For each exception-free sequence w in $L_{init}(\mathcal{A})$:*
   (a) *w is in $L_{init}(\mathcal{B})$ or*
   (b) *$w = u \cdot (n, a, b) \cdot v$ such that $u \cdot (n, a, b)$ not in $L_{init}(\mathcal{B})$ and there exists a "b'" in O_n such that $u \cdot (n, a, b')$ in $L_{init}(\mathcal{A})$ and $u \cdot (n, a, b')$ in $L_{init}(\mathcal{B})$. That is, after the prefix u, $\mathcal{B}$ decided to reduce nondeterminism by avoiding the transition corresponding to output b, and allowing the transition corresponding to the output b', which is also allowed by $\mathcal{A}$.*
2. *For each exception-free sequence w in $L_{init}(\mathcal{B})$:*
   (a) *w is in $L_{init}(\mathcal{A})$ or*
   (b) *$w = u \cdot (n, a, b) \cdot v$ and $u \cdot (n, a, e)$ in $L_{init}(\mathcal{A})$. That is a prefix of w leads to exception in $\mathcal{A}$.*

The ADT, $SchedADT2'$ of Fig. A.3, is a refinement of the NADT of Fig. A.2. $SchedADT2'$ refines $SchedADT2$ by determinizing its only nondeterministic operation *resched*. To achieve this, $SchedADT2'$ uses a *sequence* instead of set for modeling the set of ready tasks and the operation *resched* returns the longest waiting task as the next task to run.

A.3 Verification guarantee

Let us consider now the verification guarantee given by this definition of refinement. Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ and $\mathcal{A}' = (Q', U', E', \{op'_n\}_{n \in N})$ be two NADTs of type $\mathcal{N}$ such that $\mathcal{A}'$ refines $\mathcal{A}$ and let $\mathcal{S} = (R, \Sigma_l \cup \Sigma_{\mathcal{N}}, s, E_{\mathcal{S}}, \Delta)$ be an $\mathcal{N}$-client transition system. There is a natural relation σ between the states of the NADTs $\mathcal{A}'$ and $\mathcal{A}$ such that $(q', q) \in \sigma$ 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 can use this relation to define a kind of homomorphism σ' from the states of

$\mathcal{S}[\mathcal{A}']$ to the states of $\mathcal{S}[\mathcal{A}]$ such that (s, q') of $\mathcal{S}[\mathcal{A}']$ is related to (t, q) of $\mathcal{S}[\mathcal{A}]$ under σ' iff $s = t$ and $\sigma(q', q)$ holds. Thus when two states are related by σ', the local states of the client program in them are the same. This relation σ' can be seen to be a homomorphism in the following sense:

> Let (s, q') be a state in $\mathcal{S}[\mathcal{A}']$, then for each $(s, q') \xrightarrow{l} (t, r')$ in $\mathcal{S}[\mathcal{A}']$, either there exists a state (s, q) in $\mathcal{S}[\mathcal{A}]$ such that $((s, q'), (s, q)) \in \sigma'$ and $(s, q) \xrightarrow{l} (t, r)$ in $\mathcal{S}[\mathcal{A}]$ with $((t, r'), (t, r)) \in \sigma'$ or l is of the form (n, a, b) and there exists a state (s, q) in $\mathcal{S}[\mathcal{A}]$ such that $((s, q'), (s, q)) \in \sigma'$ and $(s, q) \xrightarrow{(n,a,e)} (-, E)$ in $\mathcal{S}[\mathcal{A}]$.

It is not difficult to see that Theorem 2.2 is valid for refinement between NADTs also. In fact here the proof follows more easily from condition 2 of the definition of refinement. Hence our notion of refinement for NADTs preserves the same set of properties preserved by the notion of refinement for ADTs.

A.4 Sufficient refinement condition

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n\in N})$ and $\mathcal{A}' = (Q', U', E', \{op_n\}_{n\in N})$ be NADTs of type $\mathcal{N} = (N, (I_n)_{n\in N}, (O_n)_{n\in N})$. We formulate a *sufficient* condition for $\mathcal{A}'$ to refine $\mathcal{A}$, based on an abstraction relation that relates states of $\mathcal{A}'$ to states of $\mathcal{A}$. We say $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (NRC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

(init) For each $a \in I_{init}$ and $b \in O_{init}$, if $init(p, a) \neq \{(E, e)\}$ in $\mathcal{A}$ for some state p in Q, then $init(p', a) \neq \{(E, e)\}$ in $\mathcal{A}'$ for any state $p' \in Q'$ and for each $(q', b) \in init(p', a)$ in $\mathcal{A}'$ there exists a $q \in Q$ such that $(q, b) \in init(p, a)$ in $\mathcal{A}$ for some state p in Q, and $(q', q) \in \rho$.

(sim) For each $n \in N$, $a \in I_n$, $b \in O_n$, and $p' \in Q'$, with $(p', p) \in \rho$, if $n(p, a) \neq \{(E, e)\}$ in $\mathcal{A}$, then $n(p', a) \neq \{(E, e)\}$ in $\mathcal{A}'$ and for each $(q', b) \in n(p', a)$ in $\mathcal{A}'$, there exists a $q \in Q$ such that $(q, b) \in n(p, a)$ in $\mathcal{A}$, and $(q', q) \in \rho$.

Fig. A.4 illustrates the sufficient condition (NRC) for refinement between NADTs. This condition essentially captures the following: (i) concrete cannot introduce a new transition when the abstract transition is not an exception and (ii) concrete ADT must allow at least one of the non-exception transitions allowed in the abstract.

Theorem A.1. *Let $\mathcal{A}$ and $\mathcal{A}'$ be two NADTs of type $\mathcal{N}$. Then $\mathcal{A}' \preceq \mathcal{A}$ if they satisfy condition (NRC).*

Proof. We first prove the following claim.

Claim A.1. *Let $\mathcal{A}$ abd $\mathcal{A}'$ be as above and $\rho \subseteq Q' \times Q$ an abstraction relation, such that $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (NRC) wrt ρ. Then for any states $p, q \in Q$, $p' \in Q'$ and an exception-free sequence of operations w of the form $w = (init, a, b) \cdot u$, if $p \xrightarrow{w} q$ in $\mathcal{A}$, then*

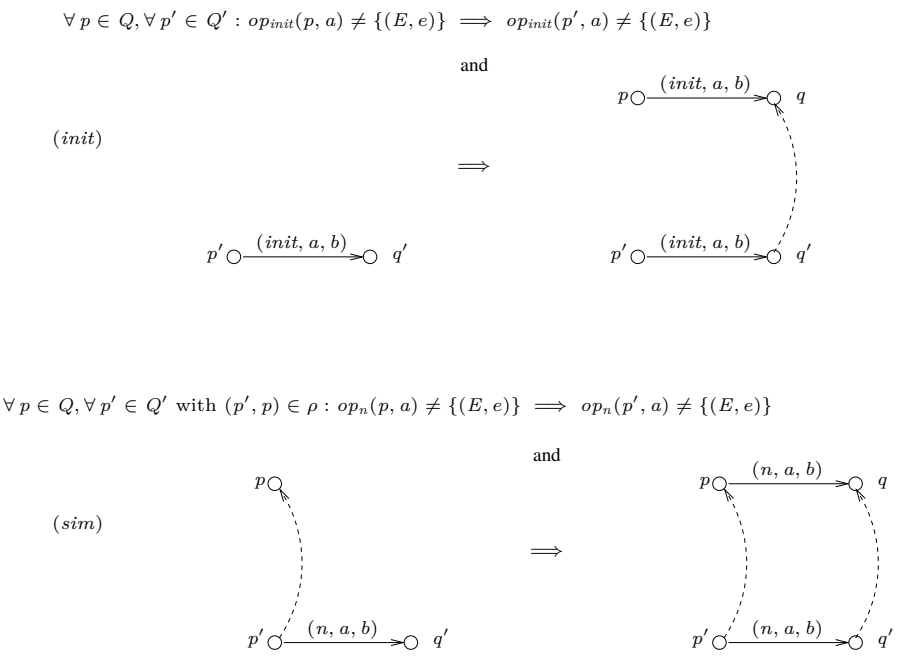


Figure A.4: Illustrating the sufficient condition (NRC) for refinement.

1. *there exists a state* q' *in* Q' *such that* $p' \xrightarrow{w} q'$ *in* $\mathcal{A}'$ *and* $(q', q) \in \rho$ *or*
2. $w = v \cdot (n, a, b) \cdot x$ *such that* $v \cdot (n, a, b)$ *not in* $L(\mathcal{A}')$, $p \xrightarrow{v} r$ *in* $\mathcal{A}$, $p \xrightarrow{v} r'$ *in* $\mathcal{A}'$, $(r', r) \in \rho$, *and there exists a* b' *in* O_n *such that* $r \overset{(n,a,b')}{\rightarrow} q$ *in* $\mathcal{A}$, $r' \overset{(n,a,b')}{\rightarrow} q'$ *in* $\mathcal{A}'$ *and* $(q', q) \in \rho$.

Proof. We prove this claim by induction on the length of u.

(Base) Let $|u| = 0$. Then $w = (init, a, b)$, where $a \in I_{init}$ and $b \in O_{init}$. Now by $(init)$ of condition (NRC), there exists a b' in O_{init} such that: $p' \overset{(init,a,b')}{\rightarrow} q'$ in $\mathcal{A}'$, $p \overset{(init,a,b')}{\rightarrow} q$ in $\mathcal{A}$ and $(q', q) \in \rho$. Thus either condition 1 or condition 2 of the claim is true and hence we are done.

(Step) Let $|u| = k + 1$. Then w is of the form: $(init, a, b) \cdot v \cdot (n, a_n, b_n)$, where $|v| = k$, $n \in N$, $a_n \in I_n$, $b_n \in O_n$ and let $p \xrightarrow{(init,a,b)\cdot v} r \overset{(n,a_n,b_n)}{\rightarrow} q$ be a trace corresponding to w in $\mathcal{A}$. It follows from the induction hypothesis that:

(a) there exists a state $r' \in Q'$ such that $p' \xrightarrow{(init,a,b)\cdot v} r'$ and $(r', r) \in \rho$ or

(b) $w = x \cdot (n, a, b) \cdot y$ such that $x \cdot (n, a, b)$ not in $L(\mathcal{A}')$, $p \xrightarrow{x} t$ in $\mathcal{A}$, $p \xrightarrow{x} t'$ in $\mathcal{A}'$, $(t', t) \in \rho$, and there exists a b' in O_n such that $t \overset{(n,a,b')}{\rightarrow} q$ in $\mathcal{A}$, $t' \overset{(n,a,b')}{\rightarrow} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$.

Either the condition 1 or the condition 2 of Claim A.1 follows from the condition (a) above and from the condition (sim) of (NRC). Also the

condition 2 of Claim A.1 follows immediately from the condition (b) above. Thus one of the conditions in Claim A.1 is always satisfied and hence we are done.

□

Let $\mathcal{A}$ and $\mathcal{A}'$ be two ADTs satisfying condition (NRC) wrt an abstraction relation ρ and w is an exception free sequence of operations in $L_{init}(\mathcal{A})$. Then it follows from Claim A.1 that either w is in $L_{init}(\mathcal{A}')$ or $w = u \cdot (n, a, b) \cdot v$ such that $u \cdot (n, a, b)$ not in $L_{init}(\mathcal{A}')$, and there exists a b' in O_n with $u \cdot (n, a, b')$ in $L_{init}(\mathcal{A})$ and $u \cdot (n, a, b')$ in $L_{init}(\mathcal{A}')$. Thus we proved that $\mathcal{A}'$ refines $\mathcal{A}$, when $\mathcal{A}'$ and $\mathcal{A}$ satisfies the condition (NRC).

□

A.5 Checking refinement condition

We show in this section a way of writing the sufficient condition (NRC) for refinement between Z models of NADTs as a logical formula that can be checked using off-the-shelf theorem provers.

Let $\mathcal{A}$ and $\mathcal{B}$ be two Z models specifying NADTs and $Var^{\mathcal{A}}$ and $Var^{\mathcal{B}}$ respectively represents the set of variables in the models $\mathcal{A}$ and $\mathcal{B}$ (see Sec. 4.1.3 for details). Then we can phrase the sufficient condition (NRC) of Sec. A.4 for $\mathcal{A}$ and $\mathcal{B}$ logically as follows:

- $Op^{\mathcal{A}} = Op^{\mathcal{B}}$, and input/output types for each $n \in Op^{\mathcal{A}}$ match (i.e. $X_n^{\mathcal{B}} = X_n^{\mathcal{A}}$ and $Y_n^{\mathcal{B}} = Y_n^{\mathcal{A}}$.)
- There exists a predicate ρ on $Var^{\mathcal{B}} \cup Var^{\mathcal{A}}$ that satisfies the following conditions:

(init)
- For each $a \in X_{init}^{\mathcal{A}}$, for each $q \in Q^{\mathcal{A}}$ and $b \in Y_{init}^{\mathcal{A}}$ with $b \neq e$:
  $BAP_{init}^{\mathcal{A}}(a, q, b) \implies \exists\, q' \in Q^{\mathcal{B}}, b' \in Y_{init}^{\mathcal{A}} \mid BAP_{init}^{\mathcal{B}}(a, q', b') \wedge \rho(q', q)$,
- For each $a \in X_{init}^{\mathcal{A}}$, for each $q' \in Q^{\mathcal{B}}$ and $b \in Y_{init}^{\mathcal{A}}$ with $b \neq e$:
  $BAP_{init}^{\mathcal{B}}(a, q', b) \implies \exists\, q \in Q^{\mathcal{A}} \mid BAP_{init}^{\mathcal{B}}(a, q, b) \wedge \rho(q', q)$,

(sim)
- for each $n \in Op^{\mathcal{A}}$, for each $a \in X_n^{\mathcal{A}}$, for each $p, q \in Q^{\mathcal{A}}$, $b \in Y_n^{\mathcal{M}}$, and $p' \in Q^{\mathcal{B}}$:
  $BAP_n^{\mathcal{A}}(p, a, q, b) \wedge \rho(p', p) \implies \exists\, q' \in Q^{\mathcal{B}}, b' \in Y_n^{\mathcal{A}} \mid BAP_n^{\mathcal{A}}(p', a, q', b') \wedge \rho(q', q)$,
- for each $n \in Op^{\mathcal{A}}$, for each $a \in X_n^{\mathcal{A}}$, for each $p', q' \in Q^{\mathcal{B}}$, $b \in Y_n^{\mathcal{M}}$, and $p \in Q^{\mathcal{A}}$:
  $BAP_n^{\mathcal{B}}(p', a, q', b) \wedge \rho(p', p) \implies \exists\, q \in Q^{\mathcal{A}} \mid BAP_n^{\mathcal{A}}(p, a, q, b) \wedge \rho(q', q)$,

We proposed a theory and a framework for proving the functional correctness of an imperative language implementation of an ADT implementation. Since we consider only deterministic implementations of ADTs, we assume that the user will determinize the Z model using a sequence of successive refinements before she proceeds to prove that the imperative implementation

is a refinement of a declarative model in Z. Hence we consider checking the condition (NRC) only between Z models. We have given above a technique for formulating the refinement condition (NRC) between two Z models. Such a condition can be checked in a theorem prover for Z like Z/Eves [44], or Rodin [4], or even by a suitable translation into a code verifier like VCC [15].

Index