A theory of refinement for ADT's with functional interfaces

Sumesh Divakaran[1], Deepak D'Souza[1], Prahladavaradan Sampath[2], Nigamanth Sridhar[3], and Jim Woodcock[4]

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

Abstract. We propose a theory of refinement for Abstract Data Types (ADTs) that interact with client programs via function calls. Our notion of refinement is in the spirit of Z/VDM. We provide a simulation-based refinement condition similar to that of He et al.'s "upward simulation", and argue that it is both sound and complete for deterministic ADTs. Our theory also facilitates compositional reasoning about complex implementations that may use several layers of sub-ADTs.

1 Introduction

Refinement-based methodologies constitute a powerful and well-developed approach for verifying functional correctness of software systems. In a correct-by-construction approach using step-wise refinement, one begins with an abstract specification of the system's functionality and successively refines it via some intermediate models, to a concrete implementation in an imperative language. Similarly, in a post-facto proof of correctness, one begins with a concrete implementation, specifies its functionality abstractly, and proves via successive refinements, that the implementation refines the abstract specification.

One of the advantages of a refinement-based approach is that it provides a standalone abstract specification (say $\mathcal{A}$) of the implementation (say $\mathcal{C}$), with the guarantee that certain properties proved about a client program P that uses $\mathcal{A}$ as a library (which we refer to as "P with $\mathcal{A}$" and denote by "$P[\mathcal{A}]$") also carry over for P with $\mathcal{C}$ (i.e. $P[\mathcal{C}]$). Thus, to verify that $P[\mathcal{C}]$ satisfies a certain property, it may be sufficient to check that $P[\mathcal{A}]$ satisfies the property. The latter check involves reasoning about a simpler component (namely $\mathcal{A}$) and can reduce the work of a prover by an order of magnitude [11]. A refinement-based proof is also modular and transparent, since it breaks up the task of reasoning about a complex implementation into smaller, more manageable, tasks.

There are a variety of notions of refinement for abstract data types proposed in the literature: VDM [4, 9], Z [3, 16, 7], Event-B [1], and behavioural subtyping [13], to name some representative ones. Each of these notions is characterised by a *definition* of refinement, the *properties* carried over for client programs,

and finally logically phrased *refinement conditions* which are typically based on simulation relations and constitute sufficient conditions for refinement. Some notions are defined directly in terms of simulation relations (Event-B, behavioural subtyping, and program refinement [14]) while others like Z and backward compatibility [15] are defined independent of simulation relations. For instance in the Z notion [16, 7], a client program P interacts with an ADT $\mathcal{A}$ by "initializing" the ADT's state using the client's "global" state, invoking some operations on the ADT, and then "finalizing" the ADT's state back to a global state of the client. An ADT $\mathcal{C}$ is said to refine another ADT $\mathcal{C}$, if the "behaviours" of $P[\mathcal{C}]$ are contained in that of $P[\mathcal{A}]$, for *every* client program P.

In this work we are interested in a setting where a client program interacts with an ADT in a *functional* manner, by periodically calling operations of the ADT, each time supplying an argument and using the value returned by the operation to update its local state. We would also like to reason about ADT implementations that themselves make use of sub-ADT's. We propose a notion of refinement in this setting, which is similar in spirit to Z and backward compatibility. The definition of refinement is independent of simulation relations and is in terms of the sequences of valid operations allowed by an ADT.

We describe the properties carried over for a client program, in this notion of refinement. We also provide sufficient simulation-based refinement conditions.

Finally, we prove a substitutivity or "contextual replacement" result: When reasoning about complex implementations of an ADT library, one often comes across situations where an implementation $\mathcal{C}$ makes use of a sub-library $\mathcal{D}$ and is thus of the form $P[\mathcal{D}]$. Now to argue that $\mathcal{C}$ refines an abstract specification $\mathcal{A}$, it is convenient to be able to abstract the sub-library $\mathcal{D}$ by a simpler abstract version say $\mathcal{B}$, and show that $P[\mathcal{B}]$ refines $\mathcal{A}$. If we can also show that $\mathcal{D}$ refines $\mathcal{B}$, then we would like to conclude that $\mathcal{C}$ refines $\mathcal{A}$. To the best of our knowledge, there are no such results in the setting of imperative programs in the literature.

Our refinement theory specializes nicely to the case of deterministic ADT's. Our refinement condition is now also *necessary*, like in [7], and proofs are more transparent.

We have used this theory of refinement to build a methodology on top of the popular verification tool for C programs called VCC, and used it to carry out verification of the functional correctness of the FreeRTOS scheduler. We refer the reader to [6] for further details on this case study.

2 Overview

In this section we illustrate the main ideas in our theory, including our refinement notion and substitutivity result, through a running example.

Fig. 1(a) shows an abstract specification in a Z-like language of a queue ADT. The specification, which we call $\texttt{z-queue}_k$ is parameterized by a constant k representing the maximum length of the queue. The "type" of the queue ADT is its set of operations $\{init, enq, deq\}$ and the associated type of 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.

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 l whose length is less than k, with an argument a, updates the state to append a to l 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.

There is a natural notion of the set of (initialised) sequences of operation calls allowed by an ADT. Each element of the sequence is of the form (n, a, b) where n is an operation name, and a and b are respectively inputs and outputs to the operation. For example, $\texttt{z-queue}_2$ allows the sequence $(\mathit{init}, \mathit{nil}, \mathit{ok}), (\mathit{enq}, 1, \mathit{ok}), (\mathit{deq}, \mathit{nil}, 1)$ (here "*nil*" represents a dummy input value). This sequence of calls is *exception-free*. It also allows the sequence $(\mathit{init}, \mathit{nil}, \mathit{ok}), (\mathit{deq}, \mathit{nil}, e)$ which however contains an exception.

Our notion of when an ADT $\mathcal{C}$ refines another ADT $\mathcal{A}$ of the same type, is that every exception-free sequence of operations permitted 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 sequence $(\mathit{init}, \mathit{nil}, \mathit{ok}), (\mathit{enq}, 1, \mathit{ok}), (\mathit{enq}, 2, \mathit{ok}), (\mathit{enq}, 3, \mathit{ok})$ which $\texttt{z-queue}_2$ does not.

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

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

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

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

(a)

```
1: int A[MAXLEN];
2: unsigned beg,
     end, len;
3:
4: void init() {
5:   beg = 0;
6:   end = 0;
7:   len = 0;
8: }
9:
10: int deq() {...}
```

(b)

```
11: void enq(int t) {
12:   if (len == MAXLEN)
13:     assert(0);
          // exception
14:   A[end] = t;
15:   if (end < MAXLEN-1)
16:     end++;
17:   else
18:     end = 0;
19:   len++;
20: }
```

(c)

```
1: init();
2: enq(0);
3: enq(1);
4: t = deq();
5: while (true) {
6:   if (*) { // tick
7:     enq(t);
8:     t = deq();
9:   }
10: }
```

Fig. 1. (a) An abstract specification $\texttt{z-queue}_k$ of the Queue ADT, parameterized by a constant k denoting the capacity of the queue; (b) `c-queue`: a C implementation of a Queue ADT; and (c) A client program `interp` that interprets two tasks of equal priority.

Consider now a C-like program `c-queue` shown in Fig. 1(b), which gives an efficient implementation of a queue ADT. 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. We can view `c-queue` as an ADT in a natural way, as follows. A program state of `c-queue` is the contents of the variables `A, beg, end` and `len`, together with a location representing the statement number to be executed next. We use a special location "0" to represent the fact that an operation has completed, and the program is not in the middle of executing an operation. The states of the ADT induced by `c-queue` is now the set of *complete* program states. As expected, we view each implementation of an operation as starting in a complete program state, taking an argument, transforming the program state – via a number of intermediate steps – from one complete state to another, and returning a value. If the function does not terminate (due to a buggy loop for example), or causes an exception (due to a null dereference for example), we view the operation as returning the exceptional value e. With this view as an ADT, `c-queue` can be seen to refine `z-queue`$_k$ whenever `MAXLEN` $\geq k$.

Let us now consider a client program of the `c-queue` library, shown in Fig. 1(c), 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. If we want to verify that the program `interp` using the `c-queue` library (we write this as "`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 `assert (t == 0 || t == 1)` at line 5), it is sufficient to check that the program with the abstract `z-queue` library (that is `interp`[`z-queue`]) verifies these properties. This can be done in a prover like VCC for example by using a ghost implementation of the abstract `c-queue` library called `g-queue`, shown in Fig. 2. Since `g-queue` is a simpler program than `c-queue` the latter check is more tractable for a prover than the former.

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

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

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

Fig. 2. A ghost version of `z-queue` in VCC.

Finally, we illustrate our "substitutivity" claim. Consider a C implementation `c-sched` of 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 operations *init* (which initializes the lists to empty), *create* (which takes a newly created task and adds it to the end of the ready list), and *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. 3(b) shows an excerpt from `c-sched` of the function implementing the *resched* operation. It uses the `c-queue` library as a sub-ADT.

ready, blocked: seq *tasks*
...

init(): *ready'* = *blocked'* = ⟨⟩

create(*t*: *tasks*):
  ready' = *ready* ⌢ ⟨*t*⟩

resched(*cur*: *tasks*):
  result: *tasks*
  result = *head*(*ready*⌢⟨*cur*⟩)
  ready' = *tail*(*ready*⌢⟨*cur*⟩)

delay(*cur*: *tasks*):
  ...

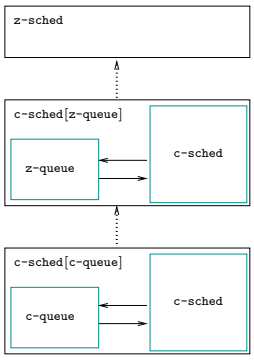
(a)

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

(b)

Fig. 3. (a) `z-sched`: An abstract specification of a scheduler ADT and (b) a part of `c-sched` showing the reschedule operation of a Scheduler ADT, that uses `c-queue` as a sub-ADT. Here `task` is assumed to be of type integer.

Fig. 3(a) shows an abstract specification of the scheduler ADT, called `z-sched`. 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 more-abstract `z-queue` component, and argue that `c-sched`[`z-queue`] refines `z-sched`. As a second step we would need to argue that `c-sched`[`c-queue`] refines `c-sched`[`z-queue`]. This is depicted in the figure alongside. Our substitutivity result tells us that to do this second step, it is sufficient to show that `c-queue` refines `z-queue`. It is in this way that the substitutivity result adds compositionality to our verification task.



3 ADT's and refinement

We begin with some preliminary notions. 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. $\mathcal{S}$ induces a language of (finite) sequences of action labels along execution paths denoted $L(\mathcal{S})$. We say $\mathcal{S}$ 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 $\mathcal{S}$ 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 $l = l'$. We use standard notation to deal with strings over an alphabet, with ϵ denoting the empty string and $u \cdot v$ denoting the concatenation of strings u and v.

3.1 Abstract data types and client transition systems

An *ADT type* is a finite set N of *operation names*. Each operation name n in N has an associated *input type* I_n and an *output type* O_n, each of which is simply a set of values. We require that there is a special *exceptional value* denoted by e, which belongs to each output type O_n; and that the set of operations N includes a designated *initialization operation* called *init*. We fix an ADT type N for the next few sections. In the sequel we will focus on *deterministic* ADTs for clarity of presentation. Most of the theory extends to non-deterministic ADTs as well, and we detail this in Sec. 5.

A (deterministic) *ADT* of type N is a structure of the form $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ where Q is the set of states of the ADT, $U \in Q$ is an arbitrary state in Q used as an *uninitialized* state, and $E \in Q$ is an *exceptional* state. Each op_n is a *realisation* of the operation n given by $op_n : Q \times I_n \to Q \times O_n$ such that $op_n(E, -) = (E, e)$ and $op_n(p, a) = (q, e) \implies q = E$. 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 $a \in I_{init}$.

As an example consider a version of the queue example from the previous section, that stores bits instead of integers. The type of the ADT is $QType = \{init, enq, deq\}$ with $I_{init} = \{nil\}$, $O_{init} = \{ok, e\}$, $I_{enq} = \mathbb{B}$, $O_{enq} = \{ok, fail, e\}$, $I_{deq} = \{nil\}$, and $O_{deq} = \mathbb{B} \cup \{fail, e\}$. Here $\mathbb{B}$ is the set of bit values $\{0, 1\}$, and *nil* is a "dummy" argument for the operations *init* and *deq*. The figure alongside shows an example ADT called $QADT_k$ of type *QType*.

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

An *N-client transition system* is a transition system whose action labels include "calls" to an ADT of type N. It is meant to model a client program like `interp` of Fig. 3(b) that uses an ADT. It is of the form $\mathcal{S} = (Q, \Sigma_l, s, E, \Delta)$ where

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

- $(p, c, E) \in \Delta$ iff $c = (n, a, e)$ for some operation n and input a (thus an exceptional return value leads to the exceptional state and this is the only way to reach it).
- $(p, -, q) \in \Delta$ 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).

Fig. 4(a) shows a *QType*-client transition system corresponding to the `interp` program of Fig. 1(c). In the sequel we will assume that client transitions systems always initialize the ADT they are using before making calls to other operations on it.

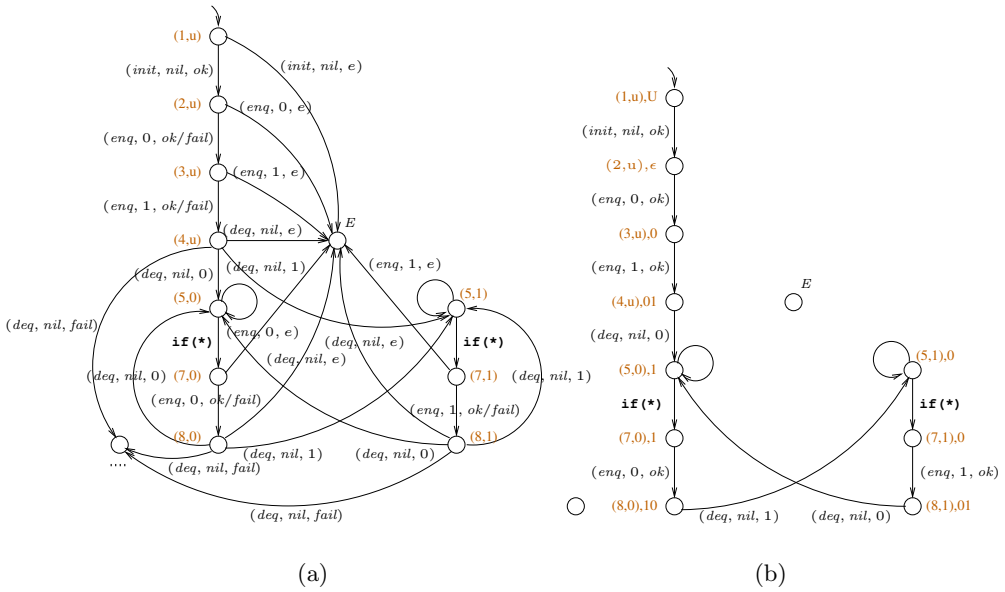


Fig. 4. (a) A *QType*-client transition system corresponding to `interp` of Fig. 1(c), and (b) the resulting transition system `interp`$[QADT_2]$.

Let $\mathcal{S} = (Q, \Sigma_l, s, E, \Delta)$ be an N-client transition system and let $\mathcal{A} = (Q', U', E', \{op_n\}_{n \in N})$ be an ADT of type 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 $\Delta' \subseteq (Q \times Q') \times \Sigma \times (Q \times Q')$ is given by

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

Fig. 4(b) shows the transition system corresponding to $\texttt{interp}[QADT_2]$.

3.2 Refinement between ADT's

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ be an ADT of type N. Then $\mathcal{A}$ induces a (deterministic) transition system $\mathcal{S}_\mathcal{A} = (Q, \Sigma_N, U, \Delta)$ where Δ is given by $(p, (n, a, b), q) \in \Delta$ iff $op_n(p, a) = (q, b)$. We define the language of *initialised sequences of operation calls* of $\mathcal{A}$, denoted $L_{init}(\mathcal{A})$, to be $L(\mathcal{S}_\mathcal{A}) \cap ((init, -, -) \cdot \Sigma_N^*)$. 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 $(-, -, e)$).

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

With reference to the example queue ADT of Fig. 3.1, we could define another ADT say $QADT'_k$ that refines $QADT_k$ by defining the *enq* and *deq* operations to return *fail* (instead of failing with an exception) when the queue is full or empty respectively. Also, $QADT_k$ refines $QADT_l$ whenever $k \geq l$.

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 ADTs of type N such that $\mathcal{A}'$ refines $\mathcal{A}$. and let $\mathcal{S}$ be an N-client transition system. There is a natural relation $\sigma \subseteq Q' \times Q$ which relates a state q' of $\mathcal{A}'$ and q of $\mathcal{A}$ precisely when 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 isomorphism σ' between $\mathcal{S}[\mathcal{A}]$ and $\mathcal{S}[\mathcal{A}']$: a state (p, q) of $\mathcal{S}[\mathcal{A}]$ and (r, q') of $\mathcal{S}[\mathcal{A}']$ are related by σ' iff $p = r$ and $\sigma(q', q)$ holds. Thus when two states are related by σ' the local states of the client program $\mathcal{S}$ in them are the same. This relation σ' can be seen to be an isomorphism or bisimulation in the following sense:

- if $\sigma'(u', u)$, and $u \xrightarrow{l} v$ in $\mathcal{S}[\mathcal{A}]$ with l a non-exception action label, then there exists v' in $\mathcal{S}[\mathcal{A}']$ such that $u' \xrightarrow{l} v'$ and $\sigma(v', v)$.
- Conversely, if $\sigma'(u', u)$, and $u' \xrightarrow{l} v'$ in $\mathcal{S}[\mathcal{A}']$, then either there exists v in $\mathcal{S}[\mathcal{A}]$ such that $u \xrightarrow{l} v$ and $\sigma(v', v)$, or l is of the form (n, a, b) and $u \xrightarrow{(n,a,e)} v$, in $\mathcal{S}[\mathcal{A}]$.

It follows from this characterisation that several properties including some temporal ones, are preserved in going from $\mathcal{S}[\mathcal{A}]$ to $\mathcal{S}[\mathcal{A}']$. In particular if $\mathcal{S}[\mathcal{A}]$ does not see an exception, neither will $\mathcal{S}[\mathcal{A}']$. Also, if $\mathcal{S}[\mathcal{A}]$ satisfies an assertion about the client $\mathcal{S}$'s local state, then either $\mathcal{S}[\mathcal{A}']$ also satisfies this assertion, or the violating execution in $\mathcal{S}[\mathcal{A}']$ is such that the corresponding execution in $\mathcal{S}[\mathcal{A}]$ has a prefix that ends in an exception. □

It follows immediately from the definition of refinement that it is transitive:

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

Let $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ and $\mathcal{A}' = (Q', U', E', \{op_n\}_{n \in N})$ be ADTs of type 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 $p \xrightarrow{(n,a,b)} q$ with $b \neq e$, then there exists $q' \in Q'$ such that $p' \xrightarrow{(n,a,b)} q'$ with $(q', q) \in \rho$.

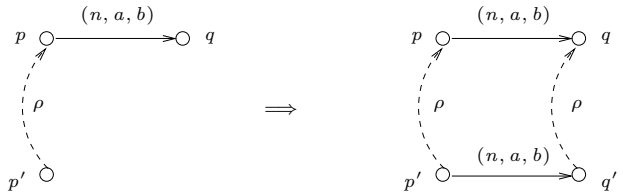


Fig. 5. Illustrating the equivalent condition (RC-sim) for refinement.

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

Proof. 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 N, and $\rho \subseteq Q' \times Q$ an abstraction relation, such that $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (RC) wrt ρ. We prove that for any states $p, q \in Q$ and $p' \in Q'$, if $p \xrightarrow{w} q$ in $\mathcal{A}$ for an initialized error-free sequence of operation calls w, then there exists a state q' in Q' such that $p' \xrightarrow{w} q'$ in $\mathcal{A}'$ and $(q', q) \in \rho$. The proof follows easily by induction on the length of w, and the if direction follows.

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. 3.2. For the (init) part, suppose $p \xrightarrow{(init,a,b)} q$ in $\mathcal{A}$. Then since $\mathcal{A}'$ refines $\mathcal{A}$, we must have $p' \xrightarrow{(init,a,b)} q'$ for some $q' \in Q'$. Also, by definition of ρ, we have $(q', q) \in \rho$. For the (sim) part, suppose $(p', p) \in \rho$, and $p \xrightarrow{(n,a,b)} q$ in $\mathcal{A}$. By definition of ρ, we know that there exists an exception-free initial sequence w such that $U \xrightarrow{w} p$ and $U' \xrightarrow{w} p'$. Since $p \xrightarrow{(n,a,b)} q$ by assumption, we have $U \xrightarrow{w \cdot (n,a,b)} q$. But since $\mathcal{A}' \preceq \mathcal{A}$, we know $U' \xrightarrow{w \cdot (n,a,b)} q'$ for some $q' \in Q'$, and hence also that $p' \xrightarrow{(n,a,b)} q'$. This implies that $(q', q) \in \rho$, and we are done. □

4 ADT transition systems and client ADTs

We are interested in reasoning about imperative language implementations of ADTs and proving a substitutivity result for them. With reference to the running example from Sec. 2, the `c-queue` program of Fig. 1(b) is what we call an "ADT transition system" (ADT TS), and the `c-sched` program of Fig. 3(b) is what we call a "*QType*-Client ADT transition system" since it is a client of a *QType* ADT and itself provides the functionality of a Scheduler type ADT.

An *ADT transition system* of type N is a structure of the form $\mathcal{S} = (Q_c, Q_l, \Sigma_l, U, \{\delta_n\}_{n \in N})$ where:

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

Fig 6(a) shows a part of the ADT transition system induced by `c-queue`, assuming it to be of type *QType*.

An ADT transition system like $\mathcal{S}$ above *induces* an ADT $\mathcal{A}_\mathcal{S}$ of type 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:

$$op_n(p, a) = \begin{cases} (q, b) & \text{if there exists a path of the form} \\ & p \xrightarrow{in(a)} r_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} r_k \xrightarrow{ret(b)} q \text{ in } \mathcal{S} \\ (E, e) & \text{otherwise.} \end{cases}$$

We say that an ADT transition system $\mathcal{S}'$ *refines* another ADT transition system $\mathcal{S}$ of the same type iff $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_\mathcal{S}$. We can also lift the sufficient condition for refinement to ADT transition systems as well. Let $\mathcal{S} = (Q_c, Q_l, \Sigma_l, U, \{\delta_n\}_{n \in N})$ and $\mathcal{S}' = (Q_c', Q_l', \Sigma_l', U', \{\delta_n'\}_{n \in N})$ be two ADT transition systems of type 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:

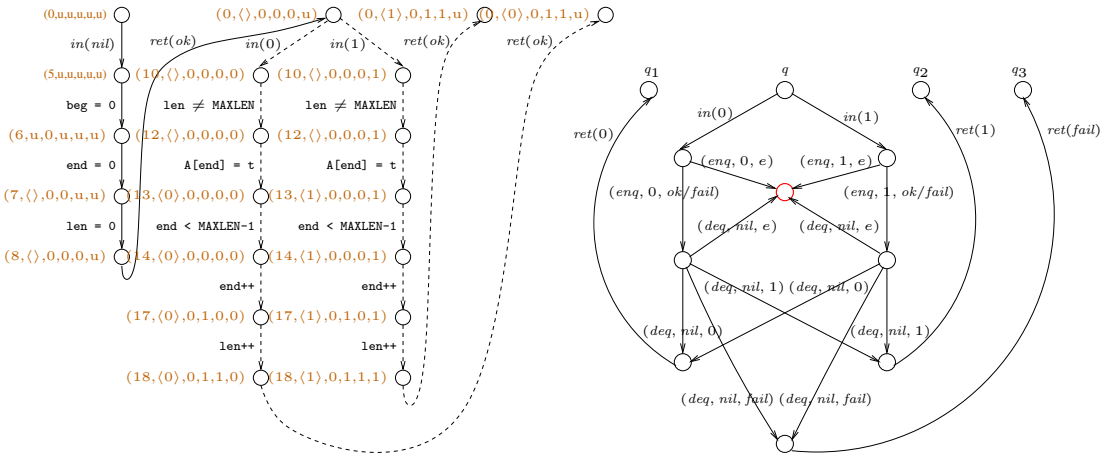


Fig. 6. (a) Part of an ADT TS representing a queue implementation from Fig.1(b), with solid edges representing δ_{init} and dashed edges representing δ_{enq}; and (b) part of a *QType*-client ADT transition system representing the *resched* implementation of Fig. 3(b).

(init) Let $a \in I_{init}$ and let (q_a, b), with $b \neq e$, be the resultant complete state and output after an $init(a)$ operation in $\mathcal{S}$ (thus, starting from an arbitrary complete state q, there is a sequence of transitions starting with $in(a)$ and ending with a $ret(b)$ in state q_a). Then, on doing an *init* operation with input a from any complete state in $\mathcal{S}'$, (1) the run in $\mathcal{S}'$ must terminate, (2) the output should be b, and (3) the resultant complete state q'_a must be such that $(q'_a, q_a) \in \rho$.

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

It is not difficult to see that $\mathcal{S}$ and $\mathcal{S}'$ satisfy condition (RC-TS) iff the ADT $\mathcal{A}_{S'}$ induced by $\mathcal{S}'$ refines the ADT $\mathcal{A}_S$ induced by $\mathcal{S}$.

Let M and N be ADT types. Then an *M-client ADT transition system* of type N (recall that this is meant to capture an ADT implementation like `c-sched`) is similar to a ADT transition system of type N, except that it makes calls to a sub-ADT of type M. It is a structure of the form $\mathcal{U} = (Q_c, Q_l, \Sigma_l, U, E, \{\delta_n\}_{n \in N})$ where Q_c, Q_l, Σ_l, and U are as in an ADT transition system. $E \in Q_l$ is an exceptional state that arises when a call to a

sub-ADT returns an exceptional value. Let Σ be the disjoint union of Σ_l, Γ_N^i,Γ_N^o and Σ_M (recall that Σ_M is the set of operation calls of type M). 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 transition system, 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 $l \xrightarrow{(m,a,b)} l' \in \delta_n$ and $l \xrightarrow{(m',a',b')} l'' \in \delta_n$, then $m = m'$ and $a = a'$).
- The δ_{init} transition relation is assumed to initialize the sub-ADT before going on to make other calls to it.

Fig 6(b) shows part of a *QType*-client ADT TS corresponding to $\delta_{resched}$ for the *resched* operation of the `c-sched` Scheduler ADT implementation of Fig. 3(b).

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

Theorem 2. *Let $\mathcal{U}$ be an M-client ADT transition system of type N, and $\mathcal{B}$ and C be ADTs of type M such that $\mathcal{C} \preceq \mathcal{B}$. Then $\mathcal{U}[\mathcal{C}]$ refines $\mathcal{U}[\mathcal{B}]$.*

Proof. It is sufficient to define a relation ρ' between the complete states of $\mathcal{U}[\mathcal{C}]$ and $\mathcal{U}[\mathcal{B}]$ satisfying condition (RC-TS). To do this we make use of the necessary and sufficient condition for refinement (RC) of Thm. 1. Since $\mathcal{C}$ refines $\mathcal{B}$, by Thm. 1 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 states of $\mathcal{U}[\mathcal{C}]$ to $\mathcal{U}[\mathcal{B}]$ given by $((p, q'), (r, q)) \in \rho'$ iff $p = r$ and $(q', q) \in \rho$. It is easy to check that ρ' satisfies condition (RC-TS) between $\mathcal{U}[\mathcal{B}]$ and $\mathcal{U}[\mathcal{C}]$, and it follows that $\mathcal{U}[\mathcal{C}]$ refines $\mathcal{U}[\mathcal{B}]$. □

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

5 Non-Deterministic ADTs

In this section we describe the version of our theory for non-determnistic ADTs. A *non-deterministic ADT* (NADT) of type N is a structure of the form $\mathcal{A} = (Q, U, E, \{op_n\}_{n \in N})$ similar to a deterministic ADT, except that each op_n is now a non-deterministic realization of the operation n. Thus, for each $n \in N$, $op_n \subseteq (Q \times I_n) \times (Q \times O_n)$ 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.

Let $\mathcal{A}$ and $\mathcal{B}$ be NADTs of type N. We say $\mathcal{B}$ *refines* $\mathcal{A}$, written $\mathcal{B} \preceq \mathcal{A}$, iff they satisfy the following conditions:

1. For each exception-free sequence of operation calls, w in $L_{init}(\mathcal{A})$:
   (a) w is in $L_{init}(\mathcal{B})$ or
   (b) w is of the form $u \cdot (n, a, b) \cdot v$ such that $u \cdot (n, a, b)$ not in $L_{init}(\mathcal{B})$, there exists a b' in O_n such that $u \cdot (n, a, b')$ in $L_{init}(\mathcal{A})$ and $u \cdot (n, a, b') \cdot v'$ in $L_{init}(\mathcal{B})$. That is, after the prefix u, $\mathcal{B}$ decided to reduce non-determinism by discarding the transition corresponding to output b and allowing a transition corresponding to output b' which is also allowed by $\mathcal{A}$.
2. For each exception-free sequence of operation calls, 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 $\mathcal{A}$.

We can give a version of the refinement conditions (RC) for NADTs which is however sufficient but *not* necessary. 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 N. We say $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (NRC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

(init) Let p and p' be arbitrary states in $\mathcal{A}$ and $\mathcal{A}'$ respectively. For each $a \in I_{init}$, $b \in O_{init}$, if $init(p, a) \neq (E, e)$ in $\mathcal{A}$ then $init(p', a) \neq (E, e)$ in $\mathcal{A}'$ 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 $\mathcal{A}$ 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 $\mathcal{A}'$ either there exists a $q \in Q$ such that $(q, b) \in n(p, a)$ in $\mathcal{A}$ and $(q', q) \in \rho$ or $n(p, a) = (E, e)$.

Fig. 7 illustrates sufficient condition for refinement between NADTs. This condition essentially captures the following: (i) the concrete cannot introduce a new transition when the abstract transition is not an exception and (ii) the concrete ADT must allow at least one of the non-exception transitions allowed in the abstract.

Theorem 3. *Let $\mathcal{A}$ and $\mathcal{A}'$ be two NADTs of type N. Then $\mathcal{A}' \preceq \mathcal{A}$ if they satisfy condition (NRC).*

This notion of refinement gives similar verification guarantees for clients of the NADTs, as for the deterministic case.

The definitions of ADT transition systems can be extended to allow non-determinism, and the substitutivity result (Thm 2) continues to hold in this setting as well.

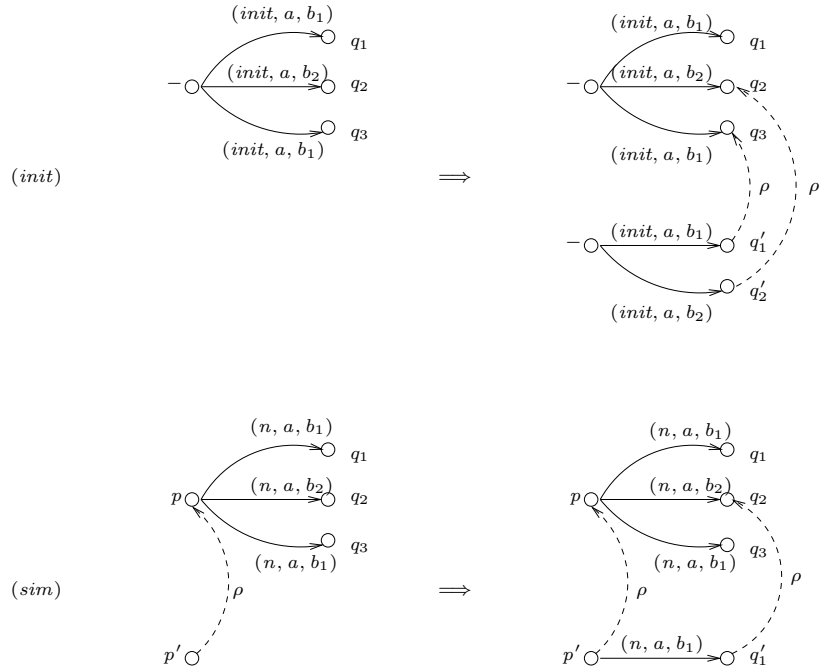


Fig. 7. Illustrating the sufficient condition (NRC) for refinement.

6 Related Work and Conclusion

We discuss related work in relation to following aspects of our work: the notion of refinement and the compositionality result.

The notion of refinement in Event-B [2], and proof environments like Dafny [12] and Resolve [5] is that the abstract simulates the concrete. This notion is not strong enough to show that the concrete provides the same functionality as the abstract, as it allows the concrete to leave out some functionality that is present in the abstract. Moreover it does not allow what we consider to be valid refinements like $QADT_3$ refining $QADT_2$. Liskov and Wing give a well-known notion of refinement in the form of behavioural subtyping, which in a deterministic setting essentially asks for both the abstract and concrete to be able to simulate each other. Once again this notion is too strong and disallows $QADT_3$ from refining $QADT_2$. Furthermore, they do not seem to have any requirements on termination, which is crucial for the verification guarantees we are interested in. Finally, as already mentioned, the notion of refinement in VDM [4, 9] and Z [3, 16] is closest to ours, when specialized to deterministic ADTs. However our definition is unique in that it is trace-based, and we extend our theory to programs that implement ADTs. We should also mention that Kapur [10]

proposes a behavioural and algebraic notion of ADTs, but the emphasis is on proving properties about them rather than refinement.

He et al.[7] give a notion of refinement which is similar to the Z notion of refinement. In this notion, the communication between the client and the ADT is not functional, since in this notion the client maintains the ADT state in the form of "global state" and communicates the state to the ADT by "initialization" and "finalization" operations. This notion is not compositional, in particular an ADT operation is not allowed to make a call to a sub-ADT, since a sub-ADT may have a different data type.

Welsch at el.[15] proposed a notion of refinement in the form of "backward compatibilty" for proving that a new library implementation is backward compatible with an existing implementation. The concrete implementation simulates the abstract implementation in this notion of reifnement. This notion uses a bijective relation between the abstact and concrete states, which is too strong and disallows certain valid reifinements that we allow. There is no abstract mathematical model or specification in this notion of refinement.

In terms of our compositionality result, the theory of CSP [8] provides notions of refinement based on traces and failures/refusals, for which a variety of compositionality (also called monotonicity or congruence) results are proved. However none of these results imply our result which is in the specific setting of transition system implementations of ADTs.

References

1. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.
2. 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`.
3. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. 1980.
4. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
5. Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part ii: specifying components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, October 1994.
6. FreeRTOS verification project. Project artifacts. `www.csa.iisc.ernet.in/~deepakd/FreeRTOS`, 2014.
7. Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *LNCS*, pages 187–196. Springer, 1986.
8. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
9. Clifford B. Jones. *Systematic software development using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, 1986.
10. Deepak Kapur. *Towards a theory of Abstract Data Types.* PhD thesis, MIT, May 1980.

11. 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.
12. 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 *LNCS*, pages 348–370. Springer, 2010.
13. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
14. Carroll C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall, 1994.
15. 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.
16. Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, 1996.