Science of Computer Programming 74 (2009) 197–218

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Verifying the CICS File Control API with Z/Eves: An experiment in the verified software repository

Leo Freitas *, Jim Woodcock, Yichi Zhang

Department of Computer Science, University of York, YO10 5DD, UK

ARTICLE INFO

Article history:
Received 12 November 2007
Received in revised form 16 September 2008
Accepted 18 September 2008
Available online 30 September 2008

Keywords:
Grand challenge in verified software
IBM CICS
Verified software repository
Z notation
Z/Eves
File API
Mechanical proof
Theorem proving
Verification challenge problems

ABSTRACT

Parts of the CICS transaction processing system were modelled formally in the 1980s in a collaborative project between IBM UK Hursley Park and Oxford University Computing Laboratory. Z was used to capture a precise description of the behaviour of various modules as a means of communicating requirements and design intentions. These descriptions were not mechanically verified in any way: proof tools for Z were not considered mature, and no business case was made for effort in this area. We report a recent experiment in using the Z/Eves theorem prover to construct a machine-checked analysis of one of the CICS modules: the File Control API. This work was carried out as part of the international Grand Challenge in Verified Software, and our results are recorded in the Verified Software Repository. We give a brief description of the other modules, and propose them as challenge problems for the verification community.

1. Introduction

The Grand Challenge in Verified Software is a fifteen-to-twenty-year project that started in 2006 [14,25,1]. Tony Hoare first proposed it in 2004, and its main objective is to create a mature scientific discipline, with the software engineering community setting its own agenda and pursuing ideals of purity, generality, and accuracy far beyond current needs. The challenge is to gather together a significant body of verified programs that have precise and complete external and internal specifications, and machine-checked proofs of correctness with respect to a sound theory of programming. On completion of the project, there will be the following deliverables: a comprehensive theory of programming covering the features needed to build practical and reliable programs; a coherent toolset automating the theory and scaling up to large systems; a collection of verified programs—replacing existing unverified ones, and continuing to evolve as verified code; and a repository curating the results of experiments and giving access to all documentation and tools. This will convincingly demonstrate that we can repeatedly produce dependable software cost effectively.

One of the first steps in the Verification Challenge is to produce an accurate picture of the current capabilities of tools and techniques. A series of pilot projects have been started, including the mechanical verification of the Mondex smart card (using eight different approaches) [26] and the development of a dependable space-flight flash file-store (see the paper by Joshi & Holzmann at [24]). We present the results of an experiment to mechanically verify the correctness of part of the IBM CICS system. We are using formal specifications that were written nearly twenty years ago, and which were at the time

* Corresponding author. Tel.: +44 1904 434753; fax: +44 1904 432708.
E-mail addresses: leo@cs.york.ac.uk (L. Freitas), jim@cs.york.ac.uk (J. Woodcock), yczhang84@googlemail.com (Y. Zhang).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.012

influential in showing that the use of formal methods could add significant value to industrial projects. Our experiment is designed to answer the following questions:

(i) Can we mechanise the analysis of the specifications?
(ii) What degree of automation can be achieved?
(iii) What additional value will be added?
(iv) How much effort is required to carry out the work?
(v) What improvements should be made to the tools?

We chose to prove aspects of the correctness of the *CICS File Control API* [28], using Z/Eves [19–22]. The scope of our mechanisation was threefold:

(i) To check the specification for syntax and type errors.
(ii) To discharge all domain checks (verification conditions guaranteeing freedom from undefined expressions and predicates).
(iii) To calculate the preconditions for each operation.

This project is based on work carried out at IBM UK Laboratories at Hursley Park in 1989. It involved the extension of the CICS File Control interface with a new feature: *the data table*, a kind of VSAM file. A similar piece of work was carried out in 1988, involving the extension of the CICS API with the Common Programming Interface for Communications, a part of IBM's System Application Interface. These projects showed that it is possible to specify large software systems in Z, revealing inaccuracies and omissions in the original informal descriptions [5,17,2]. It was also influential in helping to shape the Z notation itself by establishing what IBM called "the Oxford style", and what Oxford called the "IBM style". This involves the now-familiar description of an abstract data type structured using Z's schema calculus, including Δ and Ξ, "?" and "!", separation of normal and exceptional behaviour, robust interfaces, and the use of promotion for presenting layered system descriptions [23,27,13].

Our Z tool of choice is the Z/Eves theorem prover [22]. The choice is based on its ease of use, long previous experience, and most importantly for involving students, its gentle learning curve. Although development on Z/Eves have ceased for a while, we are currently discussing with the tool builder on an open source version of the tool to be (hopefully) hosted at York. The front-end to the tool has been improved with an experimental plugin for Z/Eves within the *Community Z Tools*[1] framework.

The rest of the paper explains the context and nature of our experimental work. Section 2 explains what a CICS system does, describes its modular architecture, and recalls the history of the use of Z within CICS development. It gives an overview of all the CICS modules and data tables as a background to the use of CICS as a source of challenge problems for the verification community. Section 3 gives more detail about the subject of our case study: *the CICS File Control API*. We describe the module's application and management interfaces and some of the general functional requirements. Section 5 lists the changes that we made to the original Z specifications to make up for omissions and errors that we detected during our analyses. Section 6 discusses the nature of our analysis: domain checks for consistency and preconditions for applicability. Section 7 presents our experimental results: we give facts and figures about the extent of our achievements and how much effort was required. Finally, in Section 8 we draw some conclusions and point to future work.

2. Introduction to CICS

CICS is a family of software systems developed by IBM to assist companies and businesses in managing their day-to-day online business transactions. CICS offers a wide variety of services for transaction processing with high availability, integrity, performance, reliability, and scalability. The most important of these services are: continuous operation, parallel execution from multiple users, connectivity with database management systems, and built-in facilities for ensuring data integrity, failure recovery and transaction back out (see the chapter on CICS and the B method in [8]). The best-known applications are in bank clearing, stock control, airline reservation, and ATM systems and, with many thousands of corporate licenses, CICS must be one of the most successful pieces of software in the world [18,27].

CICS consists of an application programming interface and some control tables. The former contains commands for display access, resource access, communication and transaction control, while the latter contains information on the overall state of the system, terminals connected, resources available, and the state of files.

In the early implementations of CICS, the API involved control blocks and assembly language macro calls, encouraging programmers to know the internal details about the control blocks used in the system implementation. In 1976, with the release of CICS/OS/VS version 1 release 3, a new interface was designed to replace the previous one, providing a cleaner, less error-prone interface for other business applications to request CICS services. CICS commands are similar to operating system calls, but at a higher level, and they can directly provide services such as security checking, transaction logging, and error recovery. Users write a program in an imperative language to create a business application, invoking CICS commands where needed.

[1] http://czt.sourceforge.net.

The API contains more than 90 commands with over 300 options in all; each command returns either a normal response or one of 60 error responses. The API was originally divided into several groups of commands, and these groups were also used in early versions of IBM CICS Application Programming Reference [11]. In the 1980s it was decided to use these groups to structure the formal specifications into fifteen *CICS modules*. The most important of these modules, along with their description can be found in [11]. The modules are:

(i) ***Automatic Transaction Initiation.*** Supports tasks running inside a CICS partition.
(ii) ***Basic mapping support.*** Provides an interface between programs and terminal control, avoiding the need to marshall complicated strings of control characters to send data to and receive data from terminals.
(iii) ***Dump Control.*** Provides transaction dumps to show contents and use of main storage. Can also be used to create dumps on the fly, without program termination.
(iv) ***Exceptional Condition Handling.*** Provides services to handle exceptions raised by calls on CICS commands.
(v) ***File Control.*** An interface between API programs and VSAM disk files.
(vi) ***Interval Control.*** Starts tasks at specified times.
(vii) ***Journal Control.*** Provides a standardised method of creating output files (*journals*), which are used to restore files to recover from system failure.
(viii) ***Program Control.*** Provides an interface between application programs and individual CICS services.
(ix) ***Storage Control.*** Allocates storage space to application programs. Since most programs keep all their data in working storage, which is allocated automatically, storage control commands are not used frequently.
(x) ***Task Control.*** Temporarily suspends task to prevent monopolisation of resources and domination of temporary storage queues.
(xi) ***Temporary Storage Control.*** Stores data in temporary storage queues outside a program's working storage. Temporary storage queues are held either in primary or secondary memory, depending on size.
(xii) ***Terminal Control.*** Provides an interface between application programs and the operating system's telecommunication system. Allows programs to send text to and receive text from the terminal that initiated the task.
(xiii) ***Trace Control.*** Maintains a table to trace the sequence of CICS operations performed within a task.
(xiv) ***Transactions and Principal Facilities.*** Provides communication and control with transaction facilities.
(xv) ***Transient Data Control.*** Allows access to simple sequential files (*destinations*).

There are fifteen control tables in total, each of which defines a part of the CICS environment (a functionality associated with the CICS modules). A complete list of all CICS control tables, with descriptions of their functionality and connectivity, can be found in the official IBM site [11].

- ***File Control Table.*** Registers control information for all files used under CICS. The file control table contains the name and type of each file and lists the file control operations that are valid for each file. It records whether existing records can be read sequentially or randomly, be deleted or modified.
- ***Processing Program Table.*** Registers all CICS application programs and BMS mapsets. Also contains information such as location in memory, library addresses, and language being used.
- ***Program Control Table.*** Registers the control information of all CICS transactions.
- ***Temporary Storage Table.*** Registers the data information of temporary storage being used by application programs. This table is used for later retrieval in case CICS terminates abnormally.
- ***Terminal Control Table.*** Registers all connected terminals, inter-system communication links and multi-region operation links.

There were two major activities in the 1980s involving the use of Z in IBM, both part of a joint research project between Oxford University Computing Laboratory and IBM UK Laboratories. The first involved the use of the Z notation in the development of a major new release of IBM's CICS, while the second involved the formal specification of the CICS API [5–7,9,16,17]. An evaluation reported that the result was perceived improvement in the quality and reliability of delivered code [3]. In June 1989, the first CICS product was developed using the Z notation: CICS/ESA version 3, and in April 1992, *The Queen's Award for Technological Achievement* was conferred jointly on IBM and Oxford for successfully achieving an innovation: ''applying the Z notation in the production of a transaction processing software'' [18].

The aims of these activities were: (1) to provide a basic command interface for all versions of CICS; (2) to uncover any ''accidental behaviour'' that was not part of the original designer's intention; and (3) to make explicit what behaviours were actually guaranteed. A decision was made to use Z to specify the CICS system, and additionally to provide an explanatory English text so that the specification documents would also be understandable to a wider audience. The only tools available were for preparing, viewing, and printing documents; type checkers and proof assistants were still under construction.

Since the command-level interface was considered too complicated and too big to be formally described as a whole, it was decided to divide the specification into smaller pieces according to the command-level interface modules. Attention was concentrated at the beginning on individual modules in relative isolation, and only later were they composed. The final versions of each specification were published as *IBM Hursley Technical Reports*.

3. CICS file control

CICS File Control API sits between the other modules of the API and the VSAM disk-based file-storage system. When the file interface receives a request, it passes it on to the appropriate VSAM File, which in turn manages the data storage. In addition to the standard file facilities that allow you to read from, write to, and delete a file, you can browse through records in a file.

In our experiment, we worked directly from the *IBM Hursley Technical Report* on File Control, using the Z specification document (written by Houston and Wordsworth) that was used as part of the design process for the data table features of CICS/ESA in the 1980s [10]. It is a precise description of the CICS file control interface as it applies to user-maintained data-tables (UMDTs). Two aspects are specified: (1) access to data table records (read, write, and update); and (2) access to data tables (open, close, enable, and disable). The user-maintained data-table is a kind of CICS file, resembling a database indexed with a unique key.

The operations on a table are of two kinds; the first kind are *application operations*:

- ***Read operation***. Retrieves a record for examination.
- ***Read for update operation***. Retrieves a record for exclusive access.
- ***Write operation***. Adds a new record to the file.
- ***Rewrite operation***. Replaces a record held for update.
- ***Delete operation***. Removes a record or a key from the data table.
- ***Unlock operation***. Unlocks a record that has been read for update, but is no longer required for subsequent updating or deletion.
- ***Syncpoint operation***. Commits recoverable resources.

The second kind are *Management operations*:

- ***Open operation***. Prepares a data table for use by application operations.
- ***Close operation***. Finalises a data table file.
- ***Enable operation***. Allows application operations.
- ***Disable operation***. Prevents application operations.

There are additional user and file requirements on the available operations of a table. These are:

- Each user of a data table can have only one record reserved for subsequent updating, although records can be reserved for other reasons. No user is permitted to read for update, directly delete, or write a record reserved by another user; any attempt to do so will cause the request to be delayed at least until the record is no longer reserved.
- A single user can reserve one or more records in the data table. The consistency of reserved records is under that user's exclusive control, since the system prevents them from being updated by other users (*write-integrity*).
- The level of write-integrity differs between recoverable and non-recoverable data tables (the number of records and reservation periods differ).
- Changes to a recoverable data table—done by rewrite, delete, or write—are provisional until the program chooses to use the syncpoint operation.
- The user of the disable operation can decide to wait for the disabling to become effective.
- The user of the close operation can decide to wait for the closing to become effective.
- When a data table has been closed, it might be still be used by application operations, depending on whether the data table has been disabled.

The table status has two factors. First, an *enablement status*:

- ***Enabled status.*** The table is available for use. Management and application operations are possible.
- ***Disabled status.*** The table is not available for use. The enable operation is necessary to make it available again.
- ***Unenabled status.*** The table is not available for application operations. The enable or the open operation is necessary to make it available again.
- ***Disabling status.*** Only current users of the table can perform application operations. When current users have finished their work the enablement status will become disabled.
- ***Unenabling status.*** Only current users of the table can perform application operations. When current users have finished their work the enablement status will become unenabled.

Second, an *open status*:

- ***Opened status.*** The data is available to users unless the enablement status is disabled.
- ***Closed status.*** The data is not available to users unless the enablement status is enabled.
- ***Willclose status.*** All users have finished their work and open status will become closed.

4. CICS specification

In order to discuss our findings we need to present parts of the original specification [10]. As well as specifying the interfaces, the IBM authors punctuated the Z specification with conjectures, although they did not prove them formally. We have not only mechanically proved these conjectures, but also added some discussion on why they hold. We also add and emphasise the issues that appeared during mechanisation, such as Z idiom suitability, proof difficulty, missing theorems, proofs about state conjectures, and so on. We believe this increases the understanding of some quite intricate issues in the specification.

The specification state is divided into two parts: data-table records kept to control access availability (*i.e.,* integrity and recoverability information) to the underlying contents; and data-table contents themselves. Both are defined using a form of the Z idiom called promotion [27, Ch.14], a technique for addressing the framing problem. Promotion is used to separate concerns in the description of an operation that updates a small fragment of the state, while the rest stays the same. The update of the local fragment is described in isolation, oblivious of the existence of the rest of the state and of the fragment's location. This description is then promoted into an operation on the global state by *framing it*: specifying that nothing else changes and defining where the fragment resides in the global state. The local state (LS) schema is embedded in the global state (GS); this embedding is often performed by a function (f), as in

$$LS \mathrel{\widehat=} [x : \mathbb{N};\ y : \mathbb{P}\ \mathbb{N} \mid x \in y]$$
$$GS \mathrel{\widehat=} [f : \mathbb{N} \pfun LS;\ ids : \mathbb{P}\ \mathbb{N} \mid \mathrm{dom}\ f = ids]$$

The framing schema promotes local state changes into the global state, where connections are made among local and global states.

Frame

$$\Delta LS;\ \Delta GS;\ id? : \mathbb{N}$$
$$id? \in ids \land ids' = ids \cup \{id?\}$$
$$f\ id? = \theta LS$$
$$f' = f \oplus \{(id? \mapsto \theta LS')\}$$

This is then used in conjunction with local operations, with the anonymous local state being hidden.

$$LOp \mathrel{\widehat=} [\Delta LS;\ i? : \mathbb{P}\ \mathbb{N} \mid y' = i?]$$
$$GLOp \mathrel{\widehat=} (\exists \Delta LS;\ i? : \mathbb{P}\ \mathbb{N} \bullet LOp \land Frame)$$

The form of promotion used in the File Control API has no framing schema. Instead, the global operations are defined more directly with various ways of including the local operation within the global one. We believe that this form of promotion complicates readability and mechanisation. In spite of this, we avoided the temptation to make life easier and rewrite the promotions more conventionally.

In what follows, we present: the local state and its application operations; the global state and the promoted application operations; and finally the global state management operations. Due to lack of space, we present only the most interesting of the global application and management operations. This follows the same order as in the original, and is readable by the prover we used [20–22,19].

4.1. Local state—data table records

Data tables are represented by records from *Key*s to *Data*, both of which are sequence of *Byte*, which are abstractly defined.

$$[Byte]$$
$$Key == \mathrm{seq}\ Byte$$
$$Data == \mathrm{seq}\ Byte$$

Records

$$records : Key \pfun Data$$

Thus, each known record *Key* is associated with only one piece of *Data*.

Some of the precondition and initialisation proofs require non-empty witnesses for *Key* and *Data*, hence we need to specify that *Byte* is a non-empty type.

$$someByte : Byte$$

This definition asserts the existence of *someByte*, which we use to prove that at least one byte exists.

theorem tExistsByte
$$\exists b : Byte \bullet true$$

This assertion is perfectly justified if we think of a concrete interpretation of *Byte* as the range $0..255$. The issue of non-empty given sets is in fact a remaining issue in Standard Z [12]. The Z in Z/Eves is prior to standardisation (*i.e.,* Spivey's Z [23]), but the lack of decision about how to model given sets will vary the number of assumptions one needs to have about them.

Availability of records

Before specifying the constraints on available records, we need an auxiliary type representing the set of all singleton or empty sets for a given generic type X.

$$Optional[X] == \{ s : \mathbb{F}\, X \mid \#s \leq 1 \}$$

This is used in the API specification to specify optional values. For instance

$$X = \{0, 1\} \Rightarrow Optional[X] = \{\varnothing, \{0\}, \{1\}\}$$

Available records satisfy integrity and recoverability constraints on each individual user. Reserved records are under exclusive control, and so are denied to other users; *held* records identify at most one *reserved* record for current update.

Integrity

$$held : Optional[Key]; \; reserved : \mathbb{P}\, Key$$

$$held \subseteq reserved$$

Records may be reserved either because they are being held for an update, or because they are already updated recovered records waiting to be committed. *Recovery* records are related to the state of a data table before update in the event of a backout operation.

$$Recovery \mathrel{\widehat=} [\, recovery : Key \pfun Optional[Data] \,]$$

$$InitRecovery \mathrel{\widehat=} [\, Recovery' \mid recovery' = \varnothing \,]$$

Initially, we have no recovery information. The original specification does not mention state initialisation theorems, such as the one for *Recovery* we now state.

theorem tRecoveryExists
$\exists\, Recovery' \bullet InitRecovery$

Recoverability has a further invariant stating that either there is nothing to recover, or else all recoverable records must be reserved to keep the data table's integrity. The equivalence is also stating the strict relationship between integrity and recoverability.

UserAvailState

$$Integrity; \; Recovery$$

$$recovery \neq \varnothing \Rightarrow \mathrm{dom}\; recovery = reserved$$

The *UserAvailState* schema represents an individual user's availability state. It represents the local state for the global operations defined later and it includes both recoverability and integrity constraints, where initialisation is again trivial.

InitialUserAvailState

$$UserAvailState'$$

$$reserved' = \varnothing$$

That is, since $reserved'$ is empty, then so is $held'$ and dom $recovery'$, hence $recovery'$ itself.

theorem InitialUserAvailStatePrecondition
$\exists\, UserAvailState' \bullet InitialUserAvailState$

With this in place, the specification defines three properties of the initial user's available state: (i) both *recovery* and *held* are empty; (ii) the user's initial availability state implies the initial recoverability state; and (iii) there is only one such initial state.

theorem tInitialUserAvailStateEmptyness
$\forall\, InitialUserAvailState \bullet$
$\quad recovery' = \varnothing \land held' = \varnothing$

theorem InitialUserAvailStateRecoveryInit
$\forall\, InitialUserAvailState \bullet InitRecovery$

theorem InitialUserAvailStateUniqueness
$\exists_1 \mathit{UserAvailState}' \bullet \mathit{InitialUserAvailState}$

The proofs in Z/Eves for these three properties are relatively simple: splitting on the case where $\mathit{recovery} = \varnothing$ is enough to finish the first two, whereas the third requires us to explicitly restate the conclusions from the first that both remaining state components are also empty.

4.2. Local operations—records availability

With the local state in place, we can now define its basic operations. When defining update operations over the data-table contents, the specification adds four extra requirements on how state availability constraints apply. These constraints are related to recoverable and non-recoverable data tables for read update and completion update (*e.g.,* (re-)write, delete, and unlock) operations over the availability information for the data table *UserAvailState*. It also includes deleting and adding availability information directly. They are defined below.

While reading a recoverable data table for update, we need to input a key (*ridfld*?) and corresponding data (*data*?) for the record being read.

AvailRecovReadUpdate

$$\Delta \mathit{UserAvailState};\ \mathit{ridfld}? : \mathit{Key};\ \mathit{data}? : \mathit{Data}$$

$$\mathit{held} = \varnothing \land \mathit{held}' = \{ \mathit{ridfld}? \}$$
$$\mathit{recovery}' = \mathit{recovery} \cup \mathit{reserved} \lhd \{ (\mathit{ridfld}? \mapsto \{ \mathit{data}? \}) \}$$
$$\mathit{reserved}' = \mathit{reserved} \cup \mathit{held}'$$

As a recoverable read update sets the given key for update ($\mathit{held}' = \{ \mathit{ridfld}? \}$), it is not allowed for any originally held read update ($\mathit{held} = \varnothing$). All held keys must be reserved, so the operation makes sure that the input is in *reserved*. If the record was not already *reserved*, then the operation updates the user's *recovery* information.

The behaviour for reading non-recoverable data-tables for update is similar, except that we do not need any data, no recovery information changes, and the reserved record is the one input to be held.

AvailNonRecovReadUpdate

$$\Delta \mathit{UserAvailState};\ \Xi \mathit{Recovery};\ \mathit{ridfld}? : \mathit{Key}$$

$$\mathit{held} = \varnothing \land \mathit{held}' = \{\mathit{ridfld}?\}$$
$$\mathit{reserved}' = \mathit{held}'$$

For completing an update, usually after a rewrite, delete, or unlock operation, a recoverable record must have already been held ($\mathit{held} \neq \varnothing$), nothing is held afterwards ($\mathit{held}' = \varnothing$), and no availability information changes.

AvailRecovCompletedUpdate

$$\Delta \mathit{UserAvailState}$$

$$\mathit{held} \neq \varnothing \land \mathit{held}' = \varnothing$$
$$\mathit{recovery}' = \mathit{recovery} \land \mathit{reserved}' = \mathit{reserved}$$

Similarly, in the case of non-recoverable data-tables, a record must be held for update, nothing is held afterwards, no recovery information changes, and nothing else is still reserved for recovery update ($\mathit{reserved}' = \varnothing$).

AvailNonRecovCompletedUpdate

$$\Delta \mathit{UserAvailState}$$

$$\mathit{held} \neq \varnothing \land \mathit{held}' = \varnothing$$
$$\mathit{recovery}' = \mathit{recovery} \land \mathit{reserved}' = \varnothing$$

For non-recoverable data tables, update operations serve only to cancel the availability effects of the previous read update, as the reserved records are cleared. Note that, although no record is held ($\mathit{held}' = \varnothing$), the previous updated record is under exclusive control ($\mathit{held} = \varnothing$). That keeps the held record until the next synch-point operation.

We proved the trivial property that the resulting reserved records after a recoverable read update are amongst those originally held for update. This is trivial because $\mathit{reserved}'$ does not change.

theorem AvailRecovCompletedUpdate_first
$\forall \mathit{AvailRecovCompletedUpdate} \bullet \mathit{held} \subseteq \mathit{reserved}'$

That means, even if the record is rewritten or deleted, only the user holding the record can read it for update, delete, or write, since the record is locked by that user. As no recovery information changes, some data may become redundant until the next synch-point operation.

Table 1
Local state operations precondition table.

AvailRecovReadUpdate *AvailRecovDeleteDirect* *AvailRecovWrite*	$held = \varnothing \land$ $recovery = \varnothing \Rightarrow ridfId? \in reserved$
AvailNonRecovReadUpdate	$held = \varnothing \land$ $recovery \neq \varnothing \Rightarrow reserved = \{ridfId?\}$
AvailRecovCompletedUpdate *AvailNonRecovUpdate*	$held \neq \varnothing$
AvailNonRecovCompletedUpdate	$held \neq \varnothing \land$ $recovery \neq \varnothing \Rightarrow reserved = \varnothing$

After considering read updates on data-tables, let us define the same for the direct delete and write operations. Recording the direct deletion of a recoverable record is just like *AvailRecovReadUpdate*, except that no key is held afterwards. Note that the key is still reserved for exclusive control, though.

AvailRecovDeleteDirect

$$\Delta UserAvailState;\ ridfId? : Key;\ data? : Data$$

$$held = \varnothing \land held' = \varnothing$$
$$recovery' = recovery \cup reserved \lhd \{ (ridfId? \mapsto \{ data? \}) \}$$
$$reserved' = reserved \cup \{ ridfId? \}$$

Also, the availability of non-recoverable records is not affected by a delete operation

AvailNonRecovUpdate

$$\Xi UserAvailState$$

$$held = \varnothing$$

That is, records being held cannot be set to be deleted.

Finally, adding availability information about a given record key is just like read update (*AvailRecovDeleteDirect*), but the recovery entry key is added with empty data, provided it is not already reserved.

AvailRecovWrite

$$\Delta UserAvailState;\ ridfId? : Key$$

$$held = \varnothing \land held' = \varnothing$$
$$recovery' = recovery \cup reserved \lhd \{ (ridfId? \mapsto \varnothing) \}$$
$$reserved' = reserved \cup \{ ridfId? \}$$

Adding availability information to a non-recoverable data-table is just like direct delete (*AvailNonRecovUpdate*): nothing changes provided nothing is currently being held. A complete example on how these availability operations take place is present in [10, Table 1].

Local state operations precondition

The local operation preconditions are summarised in Table 1. The proofs for these preconditions are quite trivial, except for the first three. The first three proofs are mostly identical and follow the same proof plan: the case where $recovery = \{\}$ and otherwise. The first case is trivial, whereas the second relies on the right instantiation for $recovery'$ and the fact that $ridfId? \in recovery$.

4.3. Global state—data table content

Several operations are allowed over the global state, where different types of recoverability apply. These are defined by the next two free types.

$$ServiceRequest ::= add \mid delete \mid read \mid update$$
$$RecoverType ::= recoverable \mid nonRecoverable$$

The data-table contents store information about: (i) the strictly positive length of the record keys (*keylen*) and the longest allowable record (*maxlen*); (ii) the possible operations for this table (*servreq*); (iii) the recoverability category the data table belongs to (*recovStatus*); and (iv) the actual records (*Records*) from keys to data.

ContentDefn

$keylen, maxlen : \mathbb{N}$
$servreq : \mathbb{P}\ ServiceRequest$
$recovStatus : RecoverType$

$keylen \neq 0 \land maxlen \neq 0$

As the keys and data in records are sequences, they are both finite and we can apply the cardinality operator safely.

DataContent

Records; *ContentDefn*

$\forall k : \mathrm{dom}\ records \bullet \# k = keylen$
$\forall d : \mathrm{ran}\ records \bullet \# d \leq maxlen$

Given the data contents, some global operations project out certain information over all user availability states (*UserAvailState*), such as all reserved or held keys. Thus, before specifying the global state, auxiliary functions over *UserAvailState* are axiomatically defined as

$projReserved : UserAvailState \rightarrow \mathbb{P}\ Key$
$projHeld : UserAvailState \rightarrow Optional[Key]$

$\langle\!\langle$ disabled rule dProjReserved $\rangle\!\rangle$
$projReserved = (\lambda\ UserAvailState \bullet reserved)$

$\langle\!\langle$ disabled rule dProjHeld $\rangle\!\rangle$
$projHeld = (\lambda\ UserAvailState \bullet held)$

We must prove a consistency theorem every time we introduce an axiomatic definition [22, Sect.3.1.3]. So, before introducing the definitions of *projReserved* and *projHeld*, we proved their consistency with the following two theorems.

theorem tProjReservedConsistency
$\exists\ projReserved : UserAvailState \rightarrow \mathbb{P}\ Key \bullet projReserved = (\lambda\ UserAvailState \bullet reserved)$

theorem tProjHeldConsistency
$\exists\ projHeld : UserAvailState \rightarrow Optional[Key] \bullet projHeld = (\lambda\ UserAvailState \bullet held)$

These theorems have easy proofs, since by the one-point-rule it is enough to show that the λ-expressions are total functions over the corresponding types. One minor modification we need in order to get greater levels of automation with Z/Eves is to specify λ-expressions that pattern-match available mechanisation rules. For instance, consider the Z/Eves toolkit law

theorem rule lambdaConstFnIsFun $[X, Y]$
$\forall y : Y \bullet (\lambda x : X \bullet y) \in X \rightarrow Y$

The shape of the this law dictates the shape of more efficient λ-expressions. An equivalent definition for *projReserved* like

$(\lambda x : UserAvailState \bullet x.reserved)$

is better, as more toolkit laws are available. Therefore, we define two equivalence theorems below that are easily proved.

theorem rule lProjReservedEquiv
$projReserved = (\lambda x : UserAvailState \bullet x.reserved)$

theorem rule lProjHeldEquiv
$projHeld = (\lambda x : UserAvailState \bullet x.held)$

A `rule` is a theorem that can be used as a tautology by Z/Eves' underlying proof engine. By default, the `rule`'s *ability* is `enabled`: the proof engine will always apply the rule whenever needed. If we wanted to prevent a rule being applied, perhaps to avoid expansion to the lowest level, one could define a `disabled` rule, in which case the prover will never automatically use the theorem, and it is available only through direct user interaction.

The global state is defined next, after we introduce a given set for the units of work, which identify each user.

$[UOWid]$

As for *Byte*, we assume this type is non-empty.

$someUOWid : UOWid$

and prove a simple witness introduction theorem.

theorem tExistsUOWid
$\exists\, uid : UOWid \bullet true$

The global state is formed by the data-table contents together with the user availability information from the local state. It contains: (i) the current users of a data table (*currentUsers*); (ii) user availability information for all users; (iii) a set of all reserved records; and (iv) the set of keys of all the records reserved for the exclusive control of each user, for all users.

ContentAvail

$currentUsers : \mathbb{P}\ UOWid$
$userAvail : UOWid \rightarrow UserAvailState$
$reservedToUsers : \mathbb{P}\ Key$
$reservedBy : UOWid \rightarrow \mathbb{P}\ Key$

$reservedBy = (userAvail \fatsemi projReserved)$
$reservedBy$ partition $reservedToUsers$
$\mathrm{dom}\,(reservedBy \rhd \{\varnothing\}) \subseteq currentUsers$

The fact that record reservation is for exclusive control of any user is captured by the partition between the keys reserved for each user, for all reserved keys. A partition represents a disjoint set of sets that encompass the whole partitioned type, as defined by the Z toolkit operator [19, p.80]:
syntax disjoint *prerel* `\disjoint`
syntax partition *inrel* `\partition`

$[I, X]$

disjoint $\_ : \mathbb{P}\,(I \pfun \mathbb{P}\ X)$
$\_$ partition $\_ : (I \pfun \mathbb{P}\ X) \rightarrow \mathbb{P}\ X$

$\langle\!\langle$ disabled rule disjointDef $\rangle\!\rangle$
$\forall S : I \pfun \mathbb{P}\ X \bullet$ disjoint $S \Leftrightarrow (\forall i, j : \mathrm{dom}\, S \mid \neg\, i = j \bullet S(i) \cap S(j) = \{\})$

$\langle\!\langle$ rule partitionDef $\rangle\!\rangle$
$\forall S : I \pfun \mathbb{P} X;\ T : \mathbb{P}\ X \bullet S$ partition $T \Leftrightarrow$ disjoint $S \wedge \bigcup(\mathrm{ran}\ S) = T$

Note that by partitioning all reserved records from the user availability information, *reservedToUsers* represents the set of all records not available to any potential user. Potential users are all those outside *currentUsers*, yet with user availability information, since *userAvail* is a total function. In other words, *reservedToUsers* represents the union of all reserved records among *currentUsers*. Finally, as the domain of *reservedBy* is the same as the domain of *userAvail*

theorem disabled rule lContentAvailDomEquiv
$\forall\, ContentAvail \bullet \mathrm{dom}\ reservedBy = \mathrm{dom}\ userAvail$

and the *currentUsers* encompass all non-empty *reservedBy* record keys, *currentUsers* contains all unit-of-work identifiers for all those users with reserved records.

The specification adds a theorem to ensure that with the partitioning of *reservedToUsers*, and hence the disjointness of *reservedBy*, no two users can reserve the same record for their exclusive control.

theorem tContentAvailPartitionsExclusiveControl
$\forall\, ContentAvail \bullet \forall\, ua, ub : UOWid \mid ua \neq ub \bullet reservedBy\, ua \cap reservedBy\, ub = \varnothing \wedge$
$\quad \bigcup\{\, u : UOWid \bullet (reservedBy\, u)\,\} = reservedToUsers$

The actual proof is quite laborious, requiring forty-five proof steps, but not terribly insightful, since the conclusion is nearly a direct match to the definitions of partition and disjointness above.

Mechanisation issues

After analysing some of the definitions involving *ContentAvail*, we noticed that proofs became increasingly and unnecessarily complicated. The major problem came when trying to extract the actual reserved record keys from a given *UOWid*. That is, when applying

$\forall\, ContentAvail;\ uid? : UOWid \bullet reservedBy\, u = \ldots$

Because of the equation for *reservedBy* and the λ-expression for *projReserved*, we end up with the function application

$(\, userAvail \fatsemi (\, \lambda\, UserAvailState \bullet reserved\,)\,)\, ui = \ldots a_1$

Relational composition $(\_ \fatsemi \_)$ is defined for relations in the Z toolkit as

syntax $\fatsemi$ *infun*4 `\comp`

$[X, Y, Z]$

$$\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z)$$

⟪ disabled rule compDef ⟫

$$\forall Q : X \leftrightarrow Y;\ R : Y \leftrightarrow Z \bullet Q \fatsemi R = \{ x : X;\ y : Y;\ z : Z \mid x \underline{Q}\ y\ \underline{R}\ z \bullet (x, z) \}$$

with a rule for composition application as

theorem rule applyComp $[X, Y, Z]$

$$f \in X \pfun Y \land g \in Y \pfun Z \land x \in \mathrm{dom} f \land f(x) \in \mathrm{dom} g \Rightarrow (f \fatsemi g)(x) = g(f(x))$$

To mechanise the expression above, we require that the *ui* being applied is within the domain of *userAvail*, seen as a partial function ($\pfun$), and that the result of *userAvail ui* is also within the domain of *projReserved*, again seen as a partial function. Although all these requirements hold because both *userAvail* and *projReserved* are total, whenever state updates take place the complexity goes to another level.

For instance, in the quite simple *ReadUpdateOk*1 defined below

$$userAvail' = userAvail \oplus \{ user? \mapsto \theta UserAvailState' \}$$

the resulting expression (a_1) above for *userAvail*′ would be

$$((userAvail \oplus \{ user? \mapsto \theta UserAvailState' \}) \fatsemi (\lambda\, UserAvailState \bullet reserved)\ )\ ui = \ldots \quad (a_2)$$

This is clearly unscalable for mechanisation. Therefore, we came up with an equivalent definition for *reservedBy* that is much easier to mechanise as

$$reservedBy = (\lambda\, id : UOWid \bullet (userAvail\, id).reserved)$$

This is the case because λ-expressions are total functions, and since *userAvail* is itself a total function, applying it to $id \in UOWid$ is never a problem. Finally, we proved that the schema with this version of *reservedBy*

ContentAvailNew

$$currentUsers : \mathbb{P}\ UOWid$$
$$userAvail : UOWid \rightarrow UserAvailState$$
$$reservedToUsers : \mathbb{P}\ Key$$
$$reservedBy : UOWid \rightarrow \mathbb{P}\ Key$$

$$reservedBy = (\lambda\, id : UOWid \bullet (userAvail\, id).reserved)$$
$$reservedBy \text{ partition } reservedToUsers$$
$$\mathrm{dom}\,(reservedBy \rhd \{ \varnothing \}) \subseteq currentUsers$$

is equivalent to the original,

theorem tContentAvailNewEquiv

$$ContentAvailNew \Leftrightarrow ContentAvail$$

hence it is harmless to modify the original schema everywhere. The benefits of doing so are great, since it considerably simplifies mechanisation and precondition proofs, yet we kept faithful to the original.

Adding recoverability to the global state

Next, local state availability properties of recoverability and integrity are linked to the global state, hence dividing it into two parts. Recoverable data-table contents associated with $u \in currentUsers$ are *reservedBy* the same user (u),

RecovTableContent

$$DataContent;\ ContentAvailNew$$

$$recovStatus = recoverable$$
$$\forall u : currentUsers \bullet \mathrm{dom}\,((userAvail\, u).recovery) = reservedBy\, u$$

whereas non-recoverable contents have no recoverability data associated with $u \in currentUsers$, and the key reserved by that user is just the one held for update for exclusive control of that user.

NonRecovTableContent

$$DataContent;\ ContentAvailNew$$

$$recovStatus = nonRecoverable$$
$$\forall u : currentUsers \bullet (userAvail\, u).recovery = \varnothing \land reservedBy\, u = (userAvail\, u).held$$

This implies that the number of record contents available to any current user of a non-recoverable data table is at most the number of reserved users.

theorem tNonRecovTableContentMostUsers

$$\forall NonRecovTableContent \bullet \forall u : currentUsers \bullet \#(reservedToUsers \setminus reservedBy\, u) \leq \# currentUsers$$

This theorem is difficult to prove, mainly because proofs involving cardinality are difficult and usually involve finiteness arguments. The main argument is that from *ContentAvail* we know the hypotheses that

$$reservedBy \text{ partition } reservedToUsers \land \qquad (h_1)$$
$$\text{dom}\,(reservedBy \rhd \{\varnothing\}) \subseteq currentUsers \quad (h_2)$$

which implies the conclusion that

$$\#\left(\bigcup(\text{ran } reservedBy) \setminus reservedBy\, u\right) \leq \# currentUsers \qquad (c_1)$$

From a toolkit theorem about cardinality of set difference applied to (c_1) we have that

$$\#\left(\bigcup(\text{ran } reservedBy)\right) - \#\left(\bigcup(\text{ran } reservedBy) \cap reservedBy\, u\right) \leq \# currentUsers \qquad (c_2)$$

The invariant from *Integrity* that $held \in Optional[Key]$ entails the conclusion that $\# held \leq 1$ by the definition of $Optional[X]$. From *NonRecovTableContent* we know that recovery information for a user (u) is empty and that such users have records held for update $(reserveBy\, u = (userAvail\, u).held)$. Thus, we know that $\# reservedBy\, u \leq 1$. From (1) and the definition of $(\_ \text{ partition } \_)$ we know that $reservedToUsers = \bigcup(\text{ran } reservedBy)$. Since each element in the range of *reservedBy* is at most 1 in size, then $\#(\bigcup(\text{ran } reservedBy)) \leq \text{dom } reservedBy$. From (2) we know that $\#\,\text{dom}\,(reservedBy \rhd \{\varnothing\}) \leq \# currentUsers$. As the case where $reservedBy\, u = \varnothing$ is removed by both $\bigcup$ and (2), we can add to our hypothesis that

$$\#\left(\bigcup(\text{ran } reservedBy)\right) \leq \#\,\text{dom}\,(reservedBy \rhd \{\varnothing\}) \leq \# currentUsers$$

With all this information, it is now possible to discharge the goal (c_2). Apart from this intricate flow of related properties, handling cardinality is particularly difficult because its definition involves finding a bijection over the set being counted.

$[X]$

$$\# : \mathbb{F} X \to \mathbb{N}$$

$$\forall S : \mathbb{F} X \bullet \exists f : 1\,..\,(\#S) \rightarrowtail\!\!\!\!\!\!\rightarrow S \bullet true$$

The best strategy for such proofs is to try to establish relationships between the sets being counted, so that there is no need to actually refer to and expand this definition. That was exactly our approach explained above. Yet, the level of automation of such proofs tends to be quite low.

Finally, the complete global state of data table contents is the disjunction of each kind of data table above.

$$TableContent \mathrel{\widehat=} (RecovTableContent \lor NonRecovTableContent)$$

This is needed because some operations are defined regardless of their recoverability nature.

Global state initialisation

The initial state of the data-table contents is divided according to their recoverability status, yet some general invariants hold, as defined by the next schema. Most values are loosely defined, since they depend either on the attributes of the source data set, or on the choices determined by the CICS system programmer. Only two components are explicitly defined: initially there are no current users; and every potential user is associated with availability data in its initial state.

InitialTableContent

$$DataContent';\ ContentAvail'$$

$$currentUsers' = \varnothing$$
$$userAvail' = UOWid \times \{(\mu\, UserAvailState' \mid InitialUserAvailState)\}$$

Here we found the first instance of a mistake in the original specification, since the definite description (μ) above does not return a set of *UserAvailState* as required by the type of *userAvail*, hence we added the set braces to it. If the definite description is well defined, then the cross product above is between *UOWid* and a singleton set. Well-definedness is guaranteed by proving the domain check for *InitialTableContent* [21].

The schematic verification condition for state initialisation is [27]:

$$\forall\, in? : TIn \bullet \exists\, State'; \; out! : TOut \bullet InitState$$

which requires us to find witnesses for existentially quantified variables. In our case, the verification condition to prove is

$$\exists\, DataContent'; \; ContentAvail' \bullet InitialTableContent\,(G1)$$

From *InitialTableContent*, have nine components and two equations, so we will require seven instantiation witnesses, unless hidden equations follow from other invariants. Note that we have not yet included our equivalent version of *ContentAvailNew* in the definition of *InitialTableContent*; the reason is to show an example of the impact it will cause in the mechanisation effort.

A naive attempt at proving this conjecture leads to the following goal

$$\begin{array}{l}\exists\, DataContent'; \; ContentAvail'[currentUsers' := \{\},\\ \quad userAvail' := UOWid \times \{\, (\mu\, m : \{\, UserAvailState' \mid InitialUserAvailState \,\}) \,\}] \bullet true(G1)\end{array}$$

This means that the initialisation affects only components from *ContentAvail'*, and we still need to provide seven witnesses, two of which belong to *DataContent'*. Note that Z/Eves has an extended expression substitution operator for schemas, which is syntactic sugar for Standard Z name-substitution for a schema S defining $x \in \mathbb{N}$ and an expression containing free variable y.

$$S[x := y + 1] \Leftrightarrow (S \wedge (\exists\, i : \mathbb{N} \bullet i = y + 1 \bullet S[i/x]))$$

As we have proved the existence of *DataContent'* with an arbitrary initialisation, we already know the natural choices for witnesses in our goal for the components of *DataContent'*, which is performed with the following proof command

$$\begin{array}{l}instantiatekeylen' == 1, maxlen' == 1,\\ \quad records' == \{\, (\langle someByte\rangle, \langle someByte\rangle)\,\},\\ \quad recovStatus' == recoverable,\\ \quad servreq' == \{\, add\,\};\end{array}$$

We assign arbitrary witnesses for *DataContent'* that satisfy all its invariants, which leads our goal ($G1$) to

$$\begin{array}{l}\exists\, ContentAvail'[currentUsers' := \{\},\\ \quad userAvail' := UOWid \times \{\, (\mu\, m : \{\, UserAvailState' \mid InitialUserAvailState \,\}) \,\}] \bullet true(G1)\end{array}$$

We have only two witnesses left out of the four components for *ContentAvail'*. The trouble is that the remaining components are defined in terms of *userAvail* in quite a complicated fashion which, after expanding the definition of ($\_$ *partition* $\_$) and substituting *reservedBy* and *projReserved*, leads to the following witnesses for *reservedBy* and *reservedToUsers* respectively

$$(userAvail \fatsemi (\lambda\, UserAvailState \bullet reserved)) \quad (W1)$$

$$\bigcup (\mathrm{ran}\, (userAvail \fatsemi (\lambda\, UserAvailState \bullet reserved))) \; (W2)$$

It is easy now to see that our witnesses for the remaining components, taking into account the actual value of *userAvail'*, will be very complicated expressions indeed. And that is not to mention how all these equation witnesses will substitute into the invariants of the original *ContentAvail'*: a grizzling fifty lines and quite unreadable goal.

To make progress in such a situation it is vital to find simpler idioms to mechanise, as well as the finding the right "accent" (*i.e.,* shape and placement) for the tool in use. We have already made some effort in this direction by introducing *ContentAvailNew* and substituting it into *InitialTableContent*; but that is not enough. From previous experience on the Mondex grand challenge pilot project [26], we know how hard it is to reason about μ-expressions. As they denote a single value, when involving schemas, we devised a general law we call the one-point-μ rule, which transforms μ to θ-expressions. This is very useful because the schema automation rules in Z/Eves are much richer than those for μ-expressions. To pursue this direction, we need to establish some lemmas about μ-expressions for *UserAvailState*.

First, as the μ-expression is well defined, we can replace it by something simpler, as the next lemma shows. Its proof is not easy, but it follows a strategy used before for Mondex.

theorem rule lMuEquality1
$\{ (\mu\, UserAvailState' \mid InitialUserAvailState) \} = \{ UserAvailState' \mid InitialUserAvailState \}$

The next step is to ensure this simpler set is a singleton set. Provided we can initialise *UserAvailState*, as the next easy lemma establishes,

theorem lExistsUserAvailState
$\exists\, UserAvailState' \bullet true$

we can now define singleton set equality in the next lemma, which widens the scope of the restriction imposed upon *UserAvailState*′ by its initialisation schema.

theorem rule lMuEquality2
$\forall\, UserAvailState' \mid reserved' = \{\} \bullet \{ UserAvailState' \mid InitialUserAvailState \} = \{ \theta UserAvailState' \}$

The proof of *lExistsUserAvailState* is pretty easy, as instantiating all components to the empty set is enough. As *InitialUserAvailState* is exactly the lemma's side condition, the proof of *lMuEquality*2 is quite trivial through set extensionality. That is the most appropriate Z idiom for the given definite description in CICS. We have paved the way for an equivalent (and just as easy to prove) version with expression substitution on θ-expressions added to it:

theorem rule lMuEquality3
$\{ UserAvailState' \mid InitialUserAvailState \} =$
$\{ \theta UserAvailState'[reserved' := \{\}, held' := \{\}, recovery' := \{\}] \}$

As all these equivalence lemmas are declared as enabled rewriting `rules`, the prover will automatically reduce expressions to the simpler definitions that we want. A last modification on the original expression is still needed. Since *userAvail*′ is a total function, the cross product assigned to it, even with the μ-expression removed, is not the best choice. From our positive experience from *ContentAvailNew* using λ-expressions for total functions, we add a further lemma translating the original equation for *userAvail*′ into an equivalent λ-expression, which is again easily proved by extensionality.

theorem rule lUserAvailInitLambdaEquiv
$UOWid \times \{ UserAvailState' \mid$
$\quad InitialUserAvailState \} = (\lambda\, id : UOWid \bullet \theta UserAvailState'[reserved' := \{\}, recovery' := \{\}, held' := \{\}])$

Finally, we can introduce *InitialTableContentNew*, which includes not only the easier-to-mechanise λ-expression above, but also the equivalent *ContentAvailNew*′ that is also better suited for mechanisation. Also, because of the absence of definite description, this new version has no domain check to prove.

InitialTableContentNew
$DataContent'$; $ContentAvailNew'$

$currentUsers' = \varnothing$
$userAvail' = (\lambda\, id : UOWid \bullet \theta UserAvailState'[reserved' := \{\}, recovery' := \{\}, held' := \{\}])$

As before, to make sure these modifications keep faithful to the original CICS, and for the sake of proper mechanisation, we prove the following equivalence, which is again quite easy.

theorem InitialTableContentEquivalence
$InitialTableContent \Leftrightarrow InitialTableContentNew$

We think this μ to θ to λ transformation is quite a general rule for any specification making use of μ and cross product to define values of total functions. We call this the *one-point-lambda-mu* rule.

In summary, if we had kept to the original formulation involving definite description, proofs would have been difficult and time consuming, whereas the domain check for *InitialTableContent* now has twelve easy proof steps. Apart from the proof of *lMuEquality*1 (twenty-three steps), which we inherited from Mondex, all other proofs are quite straightforward: the above four lemmas have just two repeated proof steps each; *InitialTableContentNew* has no domain check; and the proof of *InitialTableContentEquivalence* took fourteen quite trivial steps. Good news for mechanisation.

Initialisation existential proof

With the right global state initialisation definition, we can now move on to its existence proof, which we have shown was originally hard and unnecessarily complex. The initialisation theorem is

theorem InitialTableContentNewExists
$\exists\, DataContent'$; $ContentAvail'$ $\bullet$ $InitialTableContentNew$

Although the proof is still not easy, it is at least amenable to our original proof plan. If we follow the naïve proof, which instantiates *DataContent′* appropriately, the new version of the goal after elimination of *DataContent′* is

$$\exists\, ContentAvailNew' [currentUsers' := \{\},$$
$$userAvail' := (\lambda\, x : UOWid \bullet \theta UserAvailState[held := \{\}, recovery := \{\}, reserved := \{\}])\}]$$
$$\bullet\ true \qquad (G1_{v2})$$

The new version of witnesses remaining for *reservedBy* and *reservedToUsers* are given as

$$reservedBy = (\lambda\, id : UOWid \bullet \varnothing[Key]) \quad (W1_{v2})$$
$$reservedToUsers = \varnothing[Key] \qquad (W2_{v2})$$

Now we know what the values for *UserAvailState* are after its initialisation, and that they correspond to a unique singleton set, it is much easier to define the corresponding total function for *reservedBy* and set for *reservedToUsers* that will satisfy the invariants of *ContentAvailNew′*. These are the function mapping all *UOWid* to the empty set of reserved record keys, and the empty set of keys itself, since we know through the same reasoning as before that $reservedToUsers = \bigcup (\text{ran}\ reservedBy)$.

As before, after this instantiation, we need to show that our witnesses satisfy the invariant of *ContentAvailNew′*. After expansion and simplification the third version of the goal now has four subgoals.

$$UserAvailState' \wedge InitialUserAvailState$$
$$\Rightarrow$$
$$(\lambda\, id : UOWid \bullet \varnothing[Key]) = \qquad (G1.1_{v3})$$
$$(\lambda\, id : UOWid \bullet ((\lambda\, x : UOWid \bullet$$
$$\theta UserAvailState')\, id).reserved \wedge$$
$$(\text{disjoint}\ (\lambda\, id : UOWid \bullet \varnothing[Key])) \wedge \qquad (G1.2_{v3})$$
$$\bigcup (\text{ran}\ (\lambda\, id : UOWid \bullet \varnothing[Key])) = \varnothing[Key] \quad (G1.3_{v3})$$
$$\wedge\ \text{dom}\ ((\lambda\, id : UOWid \bullet \varnothing[Key]))\}$$
$$\rhd\{\varnothing[Key]\}) = \{\} \qquad (G1.4_{v3})$$

The first sub goal ($G1.1_{v3}$) is easily proved using lemma *lMuEquality2*, and so is the next ($G1.2_{v3}$) by using the definition of (disjoint _). The third sub goal ($G1.3_{v3}$) is still easy, in spite of being the most complicated, as it involves set extensionality, and the definitions of generalised set union and range. The final subgoal ($G1.4_{v3}$) is again easy, and is proved by extensionality, the definition domain, and some properties of range anti-restriction ($\rhd$).

Global state initialisation properties

The proofs of the properties about global state initialisation are easier. The first one states that every potential user (*i.e.,* those users outside *currentUsers*) has its availability information in its initial state.

theorem tInitialTableContentAvailabilityInit
$$\forall\, InitialTableContentNew;\ u : UOWid \bullet$$
$$(userAvail'\ u).held = \{\} \wedge$$
$$(userAvail'\ u).reserved = \{\} \wedge$$
$$(userAvail'\ u).recovery = \{\}$$

No user holds any records for update, no records are reserved for exclusive control, and no recovery data is known. The proof is trivial and follows directly from the definitions.

Finally, we prove that there is a valid initial global state for recoverable and non-recoverable data table contents as well as their disjoined composition.

theorem InitialTableContent_second
$$\exists\, RecovTableContent' \bullet InitialTableContentNew$$

theorem InitialTableContent_third
$$\exists\, NonRecovTableContent' \bullet InitialTableContentNew$$

theorem InitialTableContentInitialisation
$$\exists\, TableContent' \bullet InitialTableContentNew$$

The proofs of these three theorems are identical to the one detailed above (*InitialTableContentNewExists*), apart from some extra expansion and simplification.

4.4. Global application operations

Although the application operations affect the data content in various ways, all information within *ContentDefn* remains constant, because $\Xi ContentDefn$ is included in all application operations. There are six application operations, each of which is specified in several parts in order to capture separate aspects in a modular fashion. The schema calculus is a very good tool for this type of task and is used to a great extent.

In general, all data tables can be characterised by the level of write-integrity they provide. Thus, there are two distinct meanings for each of the six CICS application operations that can have an effect of data availability. At the level of unit-of-work, write integrity applies to operations on recoverable data tables as defined by the next schema.

UOWLevelWriteIntegrity

$\Delta RecovTableContent;\ \Xi ContentDefn;$
$user? : UOWid$

Recoverable data-tables' reserved records are available to users for reading, but not for updating, until they have been backed out or committed by the user who reserved them. Similarly, for non-recoverable data-tables, write integrity is set at the request level.

RequestLevelWriteIntegrity

$\Delta NonRecovTableContent;\ \Xi ContentDefn;$
$user? : UOWid$

Available records in a valid state are available for reading, but are not available for update until they have been rewritten, deleted, or unlocked after having been read for update.

These two levels of write integrity are found in all six operations. Due to space restrictions we will show one such operation: to read from a data table. The effect of a read operation depends on various factors, such as the correct key length, whether the requested record is present, whether the requested record data will fit the output area provided, and so on. Also, as data tables can always be read, no reference to *servreq* is needed.

Read without update

First, we specify common aspects to all read requests that do not affect the availability contents table, hence perform no updates. The user must supply an input key and may supply a key length.

ReadBase

$\Xi DataContent;\ \Xi ContentAvail$
$ridfld? : Key;\ keylength? : Optional[\mathbb{N}]$

$ridfld? \in \mathrm{dom}\ records \land keylength? \in \mathbb{P}\{keylen\}$

This ensures that the requested record key must be in the table records, and that if the user specifies a key length, then it must an acceptable one with respect to *keylen*. Neither data nor its availability information change.

The part of a read operation that produces a normal response is given next. If the output of *into*! is long enough for the requested *length*?, then the corresponding data is output.

ReadNoTrunc

$\Xi DataContent;\ ridfld? : Key$
$length?, length! : \mathbb{N};$
$into! : Data;\ response! : Response$

$ridfld? \in \mathrm{dom}\ records$
$length? \geq \#(records\ ridfld?)$
$length! = \#(records\ ridfld?)$
$into! = records\ ridfld?$
$response! = normalResp$

Whilst proving the well-definedness (*i.e.*, domain check) of this operation we found a missing invariant: *ridfld*? must belong to the domain of the partial function *records*. Otherwise, applying *ridfld*? to it could be undefined.

The next operation is a corresponding version, where the data read is truncated to the requested input, which is smaller than the actual stored data.

ReadTrunc

$$\Xi DataContent;\ ridfId? : Key$$
$$length?, length! : \mathbb{N}$$
$$into! : Data;\ response! : Response$$

$$ridfId? \in \mathrm{dom}\ records$$
$$length? < \#\,(records\ ridfId?)$$
$$length! = \#\,(records\ ridfId?)$$
$$into! = (1\,..\,length?) \lhd (records\ ridfId?)$$
$$response! = lengerrResp$$

Again, the same domain check condition on *ridfId*? was originally missing and added here. Finally, the two operations are disjoined together to represent extracting data considering both cases.

$$ReadLength \mathrel{\widehat{=}} ReadNoTrunc \lor ReadTrunc$$

The complete read operation without table update is defined next. Together with *ReadBase*, this forms the basis for most of the successful read requests. That is, *Read*0 represents the base specification for a successful read operation.

$$Read0 \mathrel{\widehat{=}} ReadBase \land ReadLength$$

Originally it was specified as $Read_0$, which in Z has a particular meaning: it is an operator over schemas that subscribes (renames) all declared components with a 0 subscript. This would generate a type error since there is no schema named *Read* previously declared. To fix that, we simply declare the schema as *Read*0, instead.

Read update

Next, we specify read requests that update the availability contents. The user must supply an input key and may supply a key length. As an update will take place, the user identifier for this unit-of-work must also be given. For such an update to take place, several requirements are necessary: *update* must belong to the allowed operations for this table; there must be no records held by (*reservedBy*) the given *user*?; the user must supply an input key and may supply a key length; and so on.

ReadUpdateBase

$$\Xi DataContent;\ \Delta ContentAvailNew$$
$$ridfId? : Key;\ keylength? : Optional[\mathbb{N}]\ user? : UOWid$$

$$update \in servreq$$
$$keylength? \in \mathbb{P}\,\{keylen\}$$
$$reservedBy\ user? = \varnothing$$
$$\#\,ridfId? = keylen$$
$$records' = records$$
$$currentUsers' = currentUsers \cup \{\, user? \,\}$$

Although availability information may be updated, the actual data table records remain unchanged. As the record may be held by some other user, the next operation specifies the case where the calling *user*? need to wait. That is, whenever the same conditions for *ReadUpdateBase* hold, and the requested key (*ridfId*?) is reserved to another user.

ReadUpdateNote

$$ReadUpdateBase;\ \Xi ContentAvailNew$$

$$ridfId? \in reservedToUsers \setminus reservedBy\ user?$$

An effect of this operation is to include the given *user*? into the set of *currentUsers* awaiting to be served, even though this user may not have any other record under his exclusive control.

Otherwise, the call might succeed without waiting, in which case the availability update schema for recoverable tables for read update is included (*AvailRecovReadUpdate*). The input key (*ridfId*?) is used for both *ReadUpdateBase* and the availability schema.

*ReadUpdateOk*1

UOWLevelWriteIntegrity; *ReadUpdateBase*
ReadLength; *AvailRecovReadUpdate*

$ridfId? \notin reservedToUsers \setminus reservedBy\, user?$
$\theta UserAvailState = userAvail\, user?$
$userAvail' =$
$\quad userAvail \oplus \{ (user? \mapsto \theta UserAvailState') \}$
$data? = records\, ridfId?$

For this case, the record key of a recoverable table must not be under exclusive control of any other *user*?, and user availability (*userAvail*) is changed according to the result of *AvailRecovReadUpdate*. That is, the local state *UserAvailState*′ resulting from this availability update operation is promoted into the global state for the given *user*?. Also, as the operation is occurring at the level of unit-of-work, the right data integrity schema is added so that reserved records are available to users for reading.

Read update for non-recoverable data tables is defined similarly, yet this time *AvailNonRecovReadUpdate* is included instead.

*ReadUpdateOk*2

RequestLevelWriteIntegrity; *ReadUpdateBase*
ReadLength; *data*? : *Data*
AvailNonRecovReadUpdate

$ridfId? \notin reservedToUsers \setminus reservedBy\, user?$
$\theta UserAvailState = userAvail\, user?$
$userAvail' =$
$\quad userAvail \oplus \{ (user? \mapsto \theta UserAvailState') \}$
$data? = records\, ridfId?$

For this case, the record key of a non-recoverable table must not be under exclusive control of any other *user*?, and user availability (*userAvail*) is changed according to the result of *AvailNonRecovReadUpdate*. Just as before, the resulting local state is promoted into the global state for the given *user*?. The data integrity level for non-recoverable data tables is set to request level, since no records are actually being updated until they have been rewritten, deleted, or unlocked.

Finally, the complete successful operation for read update is specified next .

$$ReadUpdate0 \mathrel{\widehat{=}} ReadUpdateOk1 \lor ReadUpdateOk2 \setminus (held, held', recovery, recovery', reserved, reserved', data?)$$

It is either the behaviour of a recoverable (*Ok*1) or non-recoverable (*Ok*2) data table read for update, where the before and after local state (*ΔUserAvailState*) is hidden. Similar to *Read*0, *ReadUpdate*0 was originally specified with a mistaken subscript 0 and corrected here.

In the usual pattern for promotion, this operation would have been specified with the equivalent schema below.

$$ReadUpdate0 \mathrel{\widehat{=}} (\exists \Delta UserAvailState \bullet ReadUpdateOk1 \lor ReadUpdateOk2)$$

This is the case of successful execution for read update. Other (error) cases are specified later on in the original specification and omitted here due to lack of space.

The other (5) application operations are defined in a similar style, and we omitted them here. We also could not include management operations due to lack of space.

Global read operation precondition

The global operation preconditions are summarised in Table 2. The proofs for these precondition are not so trivial as before, but they are quite manageable. That is thanks to the good levels of automation we achieved, as well as the schema modifications for using λ-expressions explained above. One interesting remark is that due to the absence of the domain check for *ridfId*? ∈ dom *records* in a couple of schemas above and, consequently, in many other schemas including them, error handling must also change. That is because since CICS was specified as a robust system (*i.e.,* one where operations are always available, hence total), there must be an extra error case when the requested key *ridfId*? does not belong to the available *records*. We have added these error cases, and omitted them here as their specification is trivial. The interesting fact is to have found them through proof, in spite of the fact the written document expects it to be taking care of all cases (*i.e.,* having specified a total operation).

Table 2
Global state operations precondition table.

UOWLevelWrite *RequestLevelWrite*	*true*
ReadBase *Read*0	$ridfld? \in \mathrm{dom}\ records \land$ $keylength? \in \mathbb{P}\{keylen\}$
ReadNoTrunc	$ridfld? \in \mathrm{dom}\ records \land$ $length? \geq \#(records\ ridfld?)$
ReadTrunc	$ridfld? \in \mathrm{dom}\ records \land$ $length? < \#(records\ ridfld?)$
ReadLength	$ridfld? \in \mathrm{dom}\ records$
ReadUpdateBase	$reservedBy\ user? = \varnothing \land$ $update \in servreq \land$ $\#\ ridfld? = keylen \land$ $keylength? \in \mathbb{P}\{keylen\}$
ReadUpdateNote	$user? \in currentUsers \land$ $ridfld? \in reservedToUsers \setminus reservedBy\ user?$

5. Changes made to specification

The joint IBM–Oxford project contributed greatly to the development of the Z notation, but it has evolved over the last twenty years. Spivey's *Reference Manual* [23] became a *de facto* standard for hand-written use, and Z/Eves [20] extended and changed this to produce its own machine-readable dialect. So the first task that we faced in our experiment was to update the notation used in the original specification of File Control to make it acceptable to Z/Eves. The resulting specification failed to type check, uncovering some small but interesting errors.

The logic used in the proof engine underlying Z/Eves is classical, in the sense that undefined values are never manufactured. On the other hand, the logic of Z is semi-classical, in the sense that terms can fail to denote, but predicates are classical. Z/Eves uses its classical logic to prove facts about Z by generating verification conditions to guarantee soundness; essentially that partial functions are applied within their domains, and that definite descriptions denote unique terms. The Z/Eves terminology for such a VC is a *domain check*.

We found type errors and failed domain checks in the following schemas:

(i) *ReadNoTrunc*. Missing domain constraint: $ridfld? \in \mathrm{dom}\ records$.
(ii) *ReadTrunc*. Missing domain constraint: $ridfld? \in \mathrm{dom}\ records$.
(iii) *ReadUpdateBase*. Missing domain constraint: $user? \in \mathrm{dom}\ userAvail \fatsemi projHeld$.
(iv) *ReadUpdateOk*2. Missing declaration: *data*?.

Other minor problems with missing free-type elements, and schema names with zero subscripts were also found and fixed.

6. Preconditions

The operations in the CICS API are required to be *robust*. That is, there must be no circumstances in which the operation might fail: every operation call must result in successful termination, or else it must return an error code; the operation must never abort. An operation is specified in Z as a relation, and the domain of the relation—its *precondition*—specifies the situations in which an abort must not occur. The operator "pre" extracts the precondition from an operation schema by existentially quantifying the after-state and output variables. This precondition can be investigated using Z/Eves, collecting irreducible predicates. Once this is complete, a theorem can then be constructed as follows. Suppose that we believe that the precondition of the operation *AvailRecovReadUpdate* is the conjunction:

$$held = \varnothing \land recovery\ ridfld? = \{data?\} \land \mathrm{dom}\ recovery = reserved \cup \{ridfld?\}$$

Then we can show that this is a sufficient precondition by proving the following theorem:

theorem AvailRecovReadUpdatePrecondition
$\forall \Delta UserAvailState;\ ridfld? : Key;\ data? : Data$
$\quad \mid held = \varnothing \land recovery\ ridfld? = \{data?\} \land \mathrm{dom}\ recovery = reserved \cup \{ridfld?\} \bullet \mathrm{pre}\ AvailRecovReadUpdate$

If the conjunction also appears in the schema, then it is not merely sufficient, it is also necessary. Operations are specified in the IBM style by treating each case separately as a partial operation, and then composing the pieces. The precondition of an operation in the interface can then be calculated by composing the preconditions of its parts. Although this is not true in general (*e.g.,* the existential quantifier in pre *Op* does not distribute through conjunction in general), because CICS follows the Oxford Style of modelling, preconditions can almost always be calculated in parts and then composed together.

7. Experimental results

We successfully entered the File Control specification presented here into Z/EVES. We checked syntax and types, and proved every domain check (after the changes listed above) in the 148 paragraphs of the specification shown here. The work presented here started from scratch from the original specification and was carried out by a MSc student, Yichi Zhang, who conducted some of the precondition proofs. We refactored most of the specification automation rules, and proposed and proved the new alternative definitions (using λ) equivalent to the original. These steps improved the mechanisation effort considerably, and led to a much smoother and simpler experiment.

In order to get a better idea of the project we provide some statistics for our work. The main aspects of these statistics are:

- How long did we actually spend doing proofs?
- Classification of the proofs in terms of difficulty and amount of interaction required.
- How long we estimate it would take to complete the rest of the proofs.

At first, a MSc. student (Konstantinos Mokos) typeset the original CICS report into the theorem prover. This took him around two months (in 2006) and produced the preliminary results discussed in [4]. From there, another MSc. student (Yichi Zhang) helped with some of the proof work. This second (proof) experiment presented here was conducted over another couple of months (in 2007), where Zhang got in touch with the original specification. Zhang, being a more experienced Z and Z/Eves user, we wanted him to examine how the results of the experiment would turn out, in comparison with the first preliminary work of Mokos. Apart from the MSc student, we already knew about the existing documentation from the previous work which we did on it at the beginning of the year.

We proved 73 theorems, using 746 interactive commands. The breakdown into individual Z/Eves commands is shown in the following table.

Command		Command	
`apply`	77	`cases`	15
`equality`	9	`instantiate`	48
`invoke`	210	`next`	22
`prenex`	23	`prove`	118
`rearrange`	39	`reduce`	27
`rewrite`	89	`simplify`	12
`split`	18	`use`	39
Grand Total			**746**

More than half of these commands (385) require no creativity: these are the commands that take no parameters. That is, one just needs to (mostly) blindly attempt at rewriting the goal with a few rewriting commands. Another third (237) take their parameters by pointing and clicking on the current goal or assumptions. Half the proofs require six or fewer commands to complete. The added equivalence schemas and some fine-tuned automation rules lead to much greater levels of automation than in the first exercise.

8. Conclusion and future work

(i) ***Can we mechanise the analysis of the specifications?*** That is, after the preliminary work in typesetting and consistency checking [4], our current experiment has not revealed any impediment to mechanising the IBM CICS specifications. The File Control module has been entirely mechanised.

(ii) ***What degree of automation can be achieved?*** The level of automation is below that achieved for the verification of the specification and refinement of the Mondex smart card [26]. We believe that this is due to the relative inexperience of the Masters student who carried out the bulk of the proof work. Interestingly, the File Control proofs were able to reuse some of the proof tactics and theorems about basic operators that were developed for Mondex. A considerable increase in automation can be achieved with sufficient additional effort.

(iii) ***What additional value will be added?*** One of the most obvious benefits of carrying out the proof work has been to gain a deep understanding of this specification. To paraphrase the management gurus, mathematics is like a contact sport: you have to get involved to get the most out of it. Z/Eves certainly gets you involved.

Our most important findings were missing constraints, mostly to do with preconditions guarding the application of partial functions. Documenting the precise precondition for an operation is important in understanding it fully. It is also important for the correct derivation of code from the specification. We have no access to the code that was developed from the File Control specification, so we cannot tell what happened during its development.

The additional effort required for a high degree of automation may well produce benefits beyond this experiment. Our experience is that each large-scale verification with Z/Eves brings collections of theories and proof tactics that are reusable. The challenge is to record these in such as way that others really can reuse them.

(iv) ***How much effort is required to carry out the work?*** The total amount of time required for the analysis of File Control is eighty working days. Most of the effort in driving Z/Eves was carried out by a Masters student who had recently completed a course in *Formal Methods for Specification*, essentially the chapters in [27] on Z's mathematical and schema languages. He had to learn how to use Z/Eves while he was constructing the File Control proofs. There is no record of the amount of time that it took to produce the original Z specification of File Control, but it is likely to have been much longer than sixteen weeks. This comes from evidence of email archives.

(v) ***What improvements should be made to the tools?*** Z/Eves continues to impress, both as a verification tool for the determined novice, and as a robust tool for large specifications (the Mondex experiment is much larger than File Control). Obviously, we would like to have a version of Z/Eves that has even more automation, but increasing the collection of useful theorems about the mathematical language is a good way to achieve higher levels of automation. We would like to derive code from the specification of File Control, and the automatic generation of appropriate verification conditions would be helpful (although, of course, not essential).

Our plan for further work is clear:

(i) Complete the verification of File Control.
(ii) Improve the levels of automation.
(iii) Refine the specification to working, verified code.
(iv) Extract reusable theories and tactics.
(v) Analyse other CICS modules.

But our experience with the Mondex experiment is that there is a lot to be learnt by repeating our work using different tools and techniques. We've set a benchmark to act as a point of reference for measuring the automation of verification tools. We challenge the verification community to do better. Further experiments need not be carried out in the Z notation. For example, in [15], the behaviour of a system specified in a legacy Z specification was remodelled and verified using a range of different notations and tools. In spite of their differences, some meaningful comparisons were made, such as the discovery of residual errors, the number of verification conditions, the level of automation and the amount of effort required.

Acknowledgments

We thank Konstantinos Mokos for his contributions to the preliminary File Control API typesetting and consistency checking during his time with us as an MSc student (2006–7). In writing this paper, we needed to start the proofs afresh (*i.e.*, the original proof attempt was neither complete, nor accurate).

We would like to thank the original authors of the CICS File Control specification for providing – all these years later – an excellent subject for a case study. We quote their Preface in full.

> Francis Bacon, in a survey of the method by which knowledge was communicated in his time, wrote
>
> ''For as knowledges are now delivered, there is a kind of contract of error between the deliverer and the receiver. For he that delivereth knowledge, desireth to deliver it in such a form as may best be believed, and not as best be examined, and he that receiveth knowledge, desireth, rather present satisfaction, than expectant enquiry; and so rather not to doubt, than not to err: glory making the author not to lay open his weakness, and sloth making the disciple not to know his strength.''
>
> We would like this report to contribute to the overthrowing of this conspiracy, and we present it as the basis of a clearly understood contract between the deliverer and the receiver of the file control interface. In this spirit we urge you, the reader, to adopt the required approach of expectant and critical enquiry, and to tell the authors about any errors that you may find or any changes that would make their meaning clearer.

We hope that we have lived up to the CICS authors' expectations.

References

[1] Juan Bicarregui, C.A.R. Hoare, J.C.P. Woodcock, The verified software repository: A step towards the verifying compiler, Formal Aspects of Computing 18 (2) (2006) 143–151.
[2] S. Croxall, P.J. Lupton, J.B. Wordsworth, A formal specification of the CPI communications, Technical Report TR12.277, IBM Hursley Park, December 1990.
[3] B.P. Collins, J.E. Nicholls, I.H. Sørensen, Introducing formal methods: The CICS experience with Z, in: B. Neumann, et al. (Eds.), Mathematical Structures for Software Engineering, Oxford University Press, 1991, pp. 153–164.
[4] Leo Freitas, Konstantinos Mokos, Jim Woodcock, Verifying the CICS File Control API with Z/Eves: An experiment in the verified software repository, in: 12th International Conference on Engineering of Complex Computer Systems, ICECCS 2007, July 2007, New Zealand, IEEE Computer Society, 2007, pp. 290–298.
[5] I.J. Hayes, Specifying the CICS application programmers interface, Technical Monograph PRG-47, Oxford University Computing Laboratory, January 1987.
[6] I.J. Hayes, Applying formal specification to software development in industry, In [7].
[7] I.J. Hayes (Ed.), Specification Case Studies, 2nd ed., Prentice Hall, 1993.
[8] Michael G. Hinchey, Jonathan P. Bowen, Applications of Formal Methods, Prentice Hall, 1995.

[9] I. Houston, S. King, CICS project report: Experiences and results from the use of Z, in: Proceedings of VDM'91, in: LNCS, vol. 551, Springer Verlag, 1991, pp. 588–596.
[10] Iain S.C. Houston, John B. Wordsworth, A Z specification of part of the CICS File Control API, IBM Technical Report TR12.272, IBM UK, Hursley Park, February 1990.
[11] IBM Official Website. www.ibm.com/us/.
[12] Information Technology Z Formal Specification NotationSyntax, Type System and Semantics, ISO/IEC 13568:2002, first edition.
[13] Jonathan Jacky, The Way of Z: Practical Programming with Formal Methods, Cambridge University Press, 1997.
[14] Cliff B. Jones, Peter W. O'Hearn, Jim Woodcock, Verified software: A grand challenge, IEEE Computer 39 (4) (2006) 93–95.
[15] Cliff Jones (Ed.), The Mondex Electronic Purse, Formal Aspects of Computing Journals 20 (1) (2008) (special issue).
[16] S. King, The use of Z in the restructure of IBM CICS, In [7].
[17] S. King, Specifying the IBM CICS application programming interface, In [7].
[18] The Queens Award for Technological Achievement in 1992. web.comlab.ox.ac.uk/oucl/about/qata92.html.
[19] Mark Saaltink, The Z/Eves 2.2 Mathematical Toolkit. TR-03-5493-05c. Ora Canada, June 2003.
[20] Mark Saaltink, The Z/EVES system. ZUM 97: The Z formal specification notation, in: Jonathan P. Bowen, Michael G. Hinchey, David Till (Eds.), 10th International Conference of Z Users, pp. 72–85.
[21] Mark Saaltink, The Z/Eves Reference Manual, TR-97-5493-03d. Ora Canada, September 1997.
[22] Mark Saaltink, The Z/Eves 2.0 users guide, TR-99-5493-06a. Ora Canada, October 1999.
[23] J.M. Spivey, The Z Notation: A Reference Manual, 2nd ed., Prentice Hall, 1992.
[24] Verified software: Theories, tools, experiments, conference held during October 10th–14th, 2005 at the ETH Zurich. vstte.ethz.ch/.
[25] Jim Woodcock, First steps in the verified software grand challenge, IEEE Computer 39 (10) (2006) 57–64.
[26] Jim Woodcock, Leo Freitas, Z/Eves and the Mondex electronic purse, in: Theoretical Aspects of Computing—ICTAC 2006, in: Lecture Notes in Computer Science, vol. 4281, Springer, 2006, pp. 15–64.
[27] Jim Woodcock, Jim Davies, Using Z: Specification, Refinement, and Proof, Prentice Hall, 1996.
[28] John Wordsworth, The CICS application programming interface definition. Workshops in computing archive. in: Proceedings of the Fifth Annual Z User Meeting, 1990, pp. 285–294.