Verifying the CICS File Control API with Z/Eves: An Experiment in the Verified Software Repository

Leo Freitas, Konstantinos Mokos, Jim Woodcock
Department of Computer Science
University of York, UK
{leo, jim}@cs.york.ac.uk

Abstract

Parts of the CICS transaction processing system were modelled formally in the 1980s in a collaborative project between IBM 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 on using the Z/Eves mechanical 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 [18, 27]. 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 be a convincing demonstration 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) [28] and the development of a dependable space-flight file-store (see the paper by Joshi & Holzmann at [26]). 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 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:

1. Can we mechanise the analysis of the specifications?
2. What degree of automation can be achieved?
3. What additional value will be added?
4. How much effort is required to carry out the work?
5. What improvements should be made to the tools?

We chose to prove aspects of the correctness of the *CICS File Control API* [30], using Z/Eves [22, 23, 24], since it is directly related to another grand challenge activity on verifying flash-memory POSIX file stores [6]. The scope of our mechanisation was threefold:

1. To check the specification for syntax and type errors.
2. To discharge all Z consistency checks.
3. 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 [7, 20, 4]. 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 [25, 29].

The rest of the paper explains the context and nature of our experimental work. Sect. 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. Sect. 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. Sect. 4 lists the changes that we made to the original Z specifications to make up for omissions and errors that we detected during our analyses. Sect. 5 discusses the nature of our analysis: domain checks for consistency and preconditions for applicability. Sect. 6 presents our experimental results: we give facts and figures about the extent of our achievements and how much effort was required. Finally, in Sect. 7 we draw 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 [10]). The best-known applications are in bank clearing, stock control, airline reservation, and ATM systems, and with many thousands of corporate licenses, it must be one of the most successful pieces of software in the world [21].

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.

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 [15]. 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 [14]. The modules are:

1. ***Automatic Transaction Initiation.*** To support tasks runnign inside a CICS partition.
2. ***Basic mapping support.*** 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.
3. ***Dump Control.*** Provides transaction dumps to show contetns and use of main storage. Can also be used to create dumps on the fly, without program termination.
4. ***Exceptional Condition Handling.*** Provides services to handle exceptions raised by CICS commands.
5. ***File Control.*** An interface between API programs and VSAM disk files.
6. ***Interval Control.*** For starting tasks at specified times.
7. ***Journal Control.*** Provides a standardised method of creating output files (*journals*), which are used to restore files to recover from system failure.
8. ***Program Control.*** An interface between application programs and individual CICS services.

9. ***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.

10. ***Task Control.*** For temporary suspension of a task to prevent monopolisation of resources and domination of temporary storage queues.

11. ***Temporary Storage Control.*** For storing 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.

12. ***Terminal Control.*** 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.

13. ***Trace Control.*** Maintains a table to trace the sequence of CICS operations performed within a task.

14. ***Transactions and Principal Facilities.*** Provides communication and control with transaction facilities.

15. ***Transient Data Control.*** For 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 [13].

- ***File Control Table.*** To register 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 [7, 8, 11, 19, 20]. An evaluation reported that the result was perceived improvement in the quality and reliability of delivered code [5]. 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" [21]. Thus, we see mechanisation of such an achievement with Z provers as an interesting and important exercise that might bring light not only to old mistakes, but also to new directions for Z itself.

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*.

We see a possible interesting link between this well know work on transaction processing and fault-tolerance and functionality coverage in formalised the POSIX flash-memory file-stores grand challenge pilot project [6]. It should open new opportunities for researchers in this field to contribute to the POSIX challenge.

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 [12]. 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*** retrieve 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*** used when a record has been read for update, but is no longer required for subsequent updating or deletion.
- ***Syncpoint operation*** used by transactions to commit recoverable resources.

The second kind are *Management operations*:

- ***Open operation*** used to prepare a data table for use by application operations.
- ***Close operation*** used to finalise the data table file.
- ***Enable operation*** to allow application operations.
- ***Disable operation*** to prevent 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.*** where the table is available for use. Management and application operations are possible.
- ***Disabled status.*** where the table is not available for use. The enable operation is necessary to make it available again.
- ***Unenabled status.*** where the table is not available for application operations. The enable or the open operation is necessary to make it available again.
- ***Disabling status.*** where 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.*** where 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.*** where the data is available to users unless the enablement status is disabled.
- ***Closed status.*** where the data is not available to users unless the enablement status is enabled.
- ***Willclose status.*** where all users have finished their work and open status will become closed.

4 Changes made to specification

The Z notation has evolved over the last twenty years. It stabilised considerably as a result of the CICS work. Spivey's *Reference Manual* [25] became a *de facto* standard for hand-written use, and Z/Eves and ProoPowerZ [1] enforced their own dialects. An ISO standard was eventually produced [16]. 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.

Although tedious to accomplish, there were no profound discoveries. The revised specification failed to type check, uncovering some small but interesting errors that exposed some commonalities with other challenges [28, 6]. Nevertheless, considered the proof work was undertaken by a MSc. student (Konstantinos Mokos) who had no previous knowledge of Z or theorem proving, the achievements show how powerful Z/Eves can be. The tool more than once have shown its capability in deskilling the theorem proving process for naive attempts, and, at the same time, provide a robust framework for more complex theorems.

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 (VCs) to guarantee soundness. These VCs require 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 from [12]:

1. *ReadNoTrunc*: there were missing invariants:

   $$ridfId? \in \operatorname{dom} records \wedge records \in Key \mathrel{+\!\!\!\!\!\twoheadrightarrow} Data$$

   The domain containment is somewhat self-evident, since *records* is a partial function over *Key*, whereas the finiteness of *records* is less counter-intuitive. Yet, this finiteness restriction is very important, otherwise the set cardinality operator ($\#$) could not have been used for $\#(records\ ridfid?)$, since it requires finite sets.

2. *ReadTrunc*: the same missing constraints as *ReadNoTrunc*.

3. *ReadUpdateOk*2. The declaration of *data*? is missing.

4. *WriteBase*. The declaration of *user*? is missing.

5. *WriteNoTrunc*. The declaration of *length*? is missing.

6. *WriteTrunc*. The declaration of *length*? is missing.

7. *CommitOk*. The binding $\theta InitialUserAvailState$ is not the type expected: it should be $\theta UserAvailState'$. As *InitialUserAvailState* is included in the declarations, the $\theta UserAvailState'$ bindings have its (primed) variables initialised as in *InitialUserAvailState*.

8. *BackoutOk*. The state variable *recovery* is not in scope (it should be $recovery'$). A tuple of variable names is used in a declaration, when each variable should be introduced separately. Also, the binding $\theta InitialUserAvailState$ is not the type expected: it should be $\theta UserAvailState'$ as above.

9. *CloseNoUsers*. The variable *users* is not defined. After the close operation, the table has no users, so this should be the empty set.

10. *CloseUsersWait*. The variable *users* is not defined. After the close operation, the set of current users waiting does not change, so this should be *currentUsers*.

11. *OpenDisabling*. There is a type error: *disabling* is not a value of *ostatus* but value of *estatus*. So, we assigned $estatus = disabling$, instead.

Finally, omissions were also found in free type definitions. The constant *ioerrResp* is used in *Ioerr*, *ClosedIoErr*, and *OpenIoErr*, but is not defined as a value of the *Response* type. We corrected this by adding *ioerrResp* to *Response*.

Furthermore, a suggested change for higher-levels of automation sake (that we have not done for the sake of keeping to the original) is that the pattern of projecting schemas elements using λ functions should be avoided. That is, we should redefine, say *ContentAvail*, as

$$\begin{array}{l}
\textit{ContentAvailNew} \\
\hline
currentUsers : \mathbb{P}\, UOWid \\
userAvail : UOWid \rightarrow UserAvailState \\
reservedToUsers : \mathbb{P}\, Key \\
reservedBy : UOWid \rightarrow \mathbb{P}\, Key \\
\hline
\forall\, \mathbf{u} : \mathbf{reservedToUsers} \bullet \\
\qquad \mathbf{reservedBy\,u} = (\mathbf{userAvail\,u}).\mathbf{reserved} \\
reservedBy \ \mathsf{partition}\ reservedToUsers \\
\operatorname{dom}(reservedBy \rhd \{\emptyset\}) \subseteq currentUsers \\
\hline
\end{array}$$

hence avoid *projReserved* altogether. This avoids unnecessarily complicated predicates involving functional composition ($userAvail \mathbin{⨾} projReserved$), for example. Obviously, to ensure this change indeed does not incur any hidden side effect, we proved a equivalence theorem for *ContentAvail* as a rewriting rule

$$\begin{array}{l}
\textbf{theorem}\ \text{rule tOldContenAvailEquivalence} \\
\quad \forall\, ContentAvail \bullet \forall\, u? : UOWid \bullet \\
\qquad reservedBy\, u? = (userAvail\, u?).reserved
\end{array}$$

and kept it to like the original. Similarly, this change should also be done for the *projHeld* schema projection used in *ReadUpdateBase* as

$$user? \in \bigcup \{\, u : \operatorname{dom} userAvail \bullet (userAvail\, u).held \,\}$$

since Z/Eves has more automation for $\bigcup$ than λ expressions.

5 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 cannot 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 by 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:

$$held = \emptyset \land recovery\ ridfId? = \{data?\} \qquad [a]$$
$$\land\ \mathrm{dom}\ recovery = reserved \cup \{ridfId?\}$$

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

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

If the conjunction $[a]$ also appears in the schema *AvailRecovReadUpdate*, then $[a]$ is not only sufficient, it is also necessary. Operations are specified in the IBM style of using Z 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.

The rectification of *OpenDisabling* have a disturbing side-effect: the operation *OpenNoRefresh* is not total (*i.e.,* $\mathrm{pre}\ OpenNoRefresh \Leftrightarrow true$). In fairness, the original (erroneous) operation precondition would not be *true* either. The rectified version precondition requires that whenever $ostatus = closed$, $currentUsers \neq \emptyset$ regardless of *estatus* value; and that whenever $ostatus = willclose$ and $estatus = unenabling$, $currentUsers = \emptyset$.

6 Experimental results

We successfully entered the entire File Control specification document into Z/EVES. We checked syntax and types, and proved every domain check (after the changes listed above) in the 223 paragraphs of the specification. The work was carried out as a Masters dissertation, and time limitations prevented the completion of the work: we calculated the preconditions of 73 partial operations; 11 preconditions remain to be calculated.

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.

Our experiment was conducted over four months. We spent almost three weeks studying the existing documentation. We spent twelve weeks with precondition calculation, type, syntax and domain checking, and two weeks writing up the results. We estimate that no more than a month is required to complete the proofs.

We proved 118 theorems, using 1,213 interactive commands. The breakdown into individual Z/Eves commands is shown in the following table.

Command	
`apply`	100
`cases`	16
`equality`	45
`instantiate`	59
`invoke`	127
`next`	32
`prenex`	11
`prove`	44
`rearrange`	110
`reduce`	112
`rewrite`	295
`simplify`	57
`split`	146
`substitute`	45
`use`	14
Total	1213

More than half of these commands (677) require no creativity: these are the commands that take no parameters. Another quarter (318) take their parameters by pointing and clicking the current goal or assumptions. Half the proofs require six or fewer commands to complete. These results we possible from an user (Konstantinos Mokos) with previous experience with neither Z nor Z/Eves. In retrospect, we believe this proof effort could be refactored in a much smaller (and more general) fashion, given more experience with the tool. Knowing that that the learning curve for such effective results on other theorem provers, such as ProofPowerZ [1], can be from six months up to an year, it is quite remarkable the combination of power and simplicity Z/Eves offers. In fairness, the drawback in Z/Eves is that no tactic language is directly available, hence encoding tactics is not an easy exercise, something ProoPowerZ masters. We believe that, with more care and experience, the proofs could be simplified and the level of automation raised considerably.

Furthermore, an important aspect to this effort is that some techniques used to discharge certain kinds of predicates/expressions were reused from previous experience gathered through other grand challenge projects [28, 6]. And that is in accord with the intended goals for a verified software repository containing general theories to be reused across different domains. For instance, this involves the precondition calculation style mentioned above in Sect. 5. In [28], we present more details on other techniques, such as: proof strategies for finite and injective functions; surgical schema expansion through specialised rewriting rules for higher levels of automation; new one-point law between definite-description (μ-expression) and schema binding (θ-expression), which in here were generalised for set comprehension rather then equality; weakening type rules that simplifies domain checking and precondition calculation for complex data types, such as free-types and injections; and so on. The one-point law was used for the expression

$$\begin{aligned}&userAvail' = UOWid \times \\ &\quad \{\, (\mu\, UserAvailState' \mid InitialUserAvailState)\,\}\end{aligned}$$

from the *InitialTableContent* schema, where the theorem to proved to avoid μ and opt for θ is given as

$$\begin{aligned}&\textbf{theorem}\ \text{muEquality}\\ &\quad \{\, (\mu\, UserAvailState' \mid InitialUserAvailState)\,\} =\\ &\quad \{\, UserAvailState' \mid InitialUserAvailState\,\}\end{aligned}$$

and proved true with similar techniques originally developed in [28]. It is interesting to see how this strategy was reused on high-quality Z specifications, where both (Mondex and CICS) were developed in Oxford. We think the motivation behind the Z/Eves implementors for preferring θ-expressions is that it is heavily used in mechanisation for the schema calculus, which is already highly automated. Similarly, θ-expressions are also preferred for ProofPowerZ, as far as we know.

7 Conclusion and future work

1. ***Can we mechanise the analysis of the specifications?*** Our experiment has not revealed any impediment to mechanising the IBM CICS specifications. The File Control module has been almost entirely mechanised (several precondition theorems remain to be proved, due to lack of time).

2. ***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 [28]. We believe that this is due to the relative inexperience of the Masters student (Konstantinos Mokos) 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.

3. ***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 a way that others really benefit from reusing them.

4. ***How much effort is required to carry out the work?*** The total amount of time required for the analysis of File Control is around sixteen weeks (this includes the twelve weeks already expended, and the upper-bound estimate of the amount of time required to complete the job). 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 [29] 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 (based on JCPW's memory of events seventeen years ago) it is likely to have been much longer than sixteen weeks.

5. ***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 collections 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, it is not essential). Together with efforts from [28, 6], we believe a general theory for some Z constructs, such as free-types, finite sets, and injections, could be added to the Z/Eves mathematical toolkit.

Our plan for further work is clear:

1. Complete the verification of File Control.
2. Improve the levels of automation.
3. Refine the specification to working, verified code.
4. Extract reusable theories and tactics.
5. 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, and tests their results with our given benchmark.

8 Acknowledgements

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

> 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, dessireth 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 their expectations.

References

[1] Rob Artan. *ProofPower Z Reference Manual*. Lemma 1 Ltd., 2000. File USR029.DOC.

[2] Rosalind Barden, Susan Stepney, David Cooper. *Z in Practice*. BCS practitioner series. Prentice Hall, 1994.

[3] 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):143–151 (2006).

[4] S. Croxall, P. J. Lupton and J. B. Wordsworth. *A formal specification of the CPI Communications*. Technical Report TR12.277, IBM Hursley Park, December 1990.

[5] B. P. Collins, J. E. Nicholls and I. H. Sørensen. Introducing Formal Methods: The CICS experience with Z. In B. Neumann *et al.*, editors. *Mathematical Structures for Software Engineering*. Oxford University Press, 1991, pp.153–164.

[6] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In 12^{th} *ICECCS*, Auckland New Zealand, Jul. 2007. IEEE.

[7] I. J. Hayes. *Specifying the CICS Application Programmer's interface*. Technical Monograph PRG-47, Oxford University Computing Laboratory. January 1987.

[8] I. J. Hayes. Applying formal specification to software development in industry. In [9].

[9] I. J. Hayes. editor. *Specification Case Studies*. Prentice Hall, 2nd edition. 1993.

[10] Michael G. Hinchey and Jonathan P. Bowen. *Applications of formal methods*. Prentice Hall. 1995.

[11] I. Houston and S. King. CICS Project Report: Experiences and Results from the use of Z. *Proceedings of VDM'91*, LNCS **551**:588–596. Springer Verlag, 1991.

[12] Iain S. C. Houston and 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.

[13] IBM Official Website. `www.ibm.com/us/`.

[14] CICS Official Website. `www-306.ibm.com/software/htp/cics`

[15] CICS Application Programming Interface Release 3. Technical Report SC33-1688-01, IBM UK, Hursley Park, March 1999.

[16] ISO SC22 Working Group 19. *Z notation.* Technical Report, ISO/IEC JTC1/SC22 N1970, 1995. ISO CD 13568; Committee Draft of the proposed Z Standard.

[17] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press. 1997.

[18] Cliff B. Jones, Peter W. O'Hearn, Jim Woodcock. Verified software: a grand challenge. *IEEE Computer* **39**(4):93–95 (2006).

[19] S. King. The use of Z in the restructure of IBM CICS. In [9].

[20] S. King. Specifying the IBM CICS Application Programming Interface. In [9].

[21] The Queen's Award for Technological Achievement in 1992. `web.comlab.ox.ac.uk/oucl/about/qata92.html`.

[22] Mark Saaltink. *The Z/Eves 2.2 Mathematical Toolkit*. TR-03-5493-05c. Ora Canada, June 2003.

[23] Mark Saaltink. *The Z/Eves Reference Manual*. TR-97-5493-03d. Ora Canada, September 1997.

[24] Mark Saaltink. *The Z/Eves 2.0 User's Guide*. TR-99-5493-06a. Ora Canada, October 1999.

[25] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[26] *Verified Software: Theories, Tools, Experiments*. Conference held during Oct. 10th-14th, 2005 at the ETH Zurich. `vstte.ethz.ch/`

[27] Jim Woodcock: First Steps in the Verified Software Grand Challenge. *IEEE Computer* **39**(10):57–64 (2006).

[28] Jim Woodcock, Leo Freitas. Z/Eves and the Mondex Electronic Purse. *Theoretical Aspects of Computing—ICTAC 2006*. Lecture Notes in Computer Science **4281**:15–64. Springer 2006.

[29] Jim Woodcock, Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[30] John Wordsworth. The CICS Application Programming Interface Definition. Workshops in Computing Archive. *Proceedings of the Fifth Annual Z User Meeting*, 1990, pp.285–294.