A Chain Datatype in Z

Leo Freitas, Jim Woodcock
Department of Computer Science,
University of York,
{leo,jim}@cs.york.ac.uk
14 August 2009

Abstract

We present results about a general-purpose chain datatype specified in the Z notation and mechanised using the Z/Eves theorem prover. Our particular interest comes from its use in the specification and refinement of operating system kernels for embedded real-time devices as part of a pilot project within the international Grand Challenge in Verified Software, and to contribute to the Verified Software Repository. We show—at a fairly high level—the sort of dogged work that is needed to get a body of results together to form a basis for future projects. Our work discusses important hidden and missing properties in the specification of the chain datatype and its relation to kernel design.

1 Introduction

Formal methods for software development allow the construction of an accurate characterisation of a problem domain that is firmly based on mathematics; by applying standard mathematical analyses, these methods can be used to prove the correctness of systems. The survey presented in [36] describes over 60 industrial projects, and discusses the effect formal methods have on time, cost, and quality. It shows that with tools backed by mature theory, formal methods are becoming more effective, and their use is easier to justify, not as an academic or legal requirement, but as part of a business
case. Despite the initial extra effort, formal methods can give increased reliability, accountability, and precision, and they can save money, particularly when used repeatedly. These recent advances in theory and tool support have inspired industrial and academic researchers to join up in an international Grand Challenge (GC) in Verified Software [19, 33]. Work has started with the creation of a Verified Software Repository (VSR) with two principal aims: (i) the construction of verified software components; and (ii) industrial-scale verification experiments to drive future research in the development of theory and tool support [3].

This paper describes an experiment undertaken as part of a GC pilot project on verifying operating system (OS) kernels. It explores the mechanisation of proofs of correctness of the formal specification and design of several kernels for real-time embedded systems constructed by Craig [7]. In particular, we focus in this paper on one kernel component, the scheduler, and its refinement using a chain datatype, an abstraction of a well-formed singly linked list in a program. We have already mechanised the abstract parts of the scheduler [35]. Our contribution in this paper is in the mechanisation of the chain datatype used in the refinement of the scheduler, where chains are implemented in C as arrays. We particularly emphasise the mechanisation issues encountered while specifying the various parts of the chain specification. We think this is important, since keeping them in mind is the difference between getting to a dead end or successfully completing later proofs. Obviously, different tools will have different detailed mechanisation issues, so we describe general issues. We also propose future work for modelling the other parts of the kernel, such as process queues.

Since Craig’s models are all in the Z notation [30], it naturally follows that we use a Z tool, and for us that is the Z/Eves theorem prover [29, 28]. The choice is based on its ease of use, our long previous experience with the tool, and most importantly for involving students, its gentle learning curve. Although development of Z/Eves has ceased for a while, we are currently discussing with the tool builders the development of a new open-source version. The front-end to the tool has been improved with an experimental plugin for Z/Eves as part of the Community Z Tools Project [8].

In the next two sections, we briefly set the scene. In Section 4, we present the chain datatype, together with a collection of interesting properties. After that, in Section 6, we define the chain’s operations with their main properties and preconditions. In Section 7, we discuss some of the most important lessons learned through this experiment. Finally, in Section 8 we present
conclusions and future work.

2 Related Grand Challenge projects

In 2006, the first VSR pilot project was undertaken on the verification of the Mondex smart card [31] to ITSEC Level 6 (Common Criteria Level 7) [21]. The work is reported in [22], where a summary of Mondex and its original development and certification are described [37]. The experiment mechanised the original manual proofs in Alloy [27], ASM [18], Event-B [4], OCL [24], $\pi$-calculus [23], Raise [17], and Z [14]. A second pilot project on POSIX-compliant flash file stores followed [16]. A clear domain model with widely used terminology and well-understood requirements is needed, and we have based our mechanised domain model based on Craig’s work [7] on the formal refinement of OS kernel designs.

There are two other related GC pilot projects: FreeRTOS [10] and the Microsoft Hypervisor [6]. FreeRTOS is an open-source real-time embedded operating system written in pointer-rich C; it does not have a specification, making it an attractive topic for research in formal analysis and top-down development. The extensive use of pointers offers two complementary challenges: (i) the annotation of the code with suitable assertions and the verification of the code against these assertions; and (ii) the top-down development of the code, starting from a suitable specification of its abstract behaviour. The goal of the Microsoft Hypervisor Verification Project is to develop an industrially viable verification methodology for low-level code, and to use this methodology to verify the functional correctness of the Microsoft Hypervisor [25]. The hypervisor is 60 KLOC of C and assembly language that turns a multi-processor (MP) x64 machine into a number of virtual MP x64 machines.

3 Verified OS kernels pilot project

An OS kernel is a central component of most operating systems, providing an interface to the management of hardware and software resources, including memory, processors, and I/O devices. It offers this interface to application processes through inter-process communication mechanisms and system calls. The most important features are: low-level scheduling of processes; inter-
process communication; process synchronisation; context switching; manipulation of process control blocks; hardware interrupt handling; process creation and destruction; and process suspension and resumption. Kernel development has a reputation for being a very difficult and complex programming task for two prime reasons. First, every computing system requires the OS kernel to provide correct functionality and good performance. Second, there are many reasons why the kernel cannot make use of the abstractions it provides that make higher-level programming of embedded and real-time systems easier.

Our pilot project is inspired by Craig’s book on the formal refinement of OS kernels [7]. Our project objectives are to demonstrate the feasibility of top-down OS kernel development using formal specification and verification, with refinement down to a C implementation. Craig uses the Z notation [30, 34, 20] for specification and refinement, deriving implementations in Dijkstra’s guarded command language [9], and recording correctness arguments in hand-written proofs. Our pilot project investigates the tractability of mechanising all the models in each kernel development, including formalising all proofs. A key principle is to retain Craig’s models as far as possible, making changes only for correctness, not for easing the task of mechanisation.

Part of this investigation involves constructing prototype tool chains for the development process from specification through design and down to code. For the specification and verification we use Z theorem provers like Z/Eves [29] or ProofPower Z [1]. Data refinement [34] links the abstract specification to a concrete design that is closer to code, and again we use a Z theorem prover to prove correctness. After that, we use the Z refinement calculus (ZRC) [5] to go down to the guarded command language, and we use the ZRC-Refine/Gabriel [26] tool, combined with strategies discussed in [13]. The invariants and pre- and postconditions for each programming statement are then converted to a formal annotation language for C, such as Spec# [2]. Finally, tools like Boogie/PL and the Microsoft Verified C Compiler can be used to perform static and partial correctness analysis. All results, including models, mechanisation lemmas, papers, tools, etc. are being curated in the Verified Software Repository (VSR).

The pilot project is currently in an exploratory phase, mechanising the correctness of the process scheduler [35]. We have found some interesting issues in Craig’s model, including missing and hidden invariants, particularly in the chain datatype [34] that Craig uses extensively in the refinement of his kernels. For example, chains are used to provide efficient representations
of the scheduler’s free, running, and blocked process queues. This paper is
dedicated to our more intimate understanding of the invariants for chains
that are relevant to scheduler design. We take a step back from OS kernel
design and verify the chain datatype to see which invariants are fundamental,
and which can be relaxed and proved as properties instead. In the remainder
of this paper, we present these findings, and how they relate back to Craig’s
original work. We have reports (summarised in this paper) with all definitions
and proofs that can be found in \[11, 12\].

4 Chain datatype

The chain datatype is used to represent finite lists of process identifiers in
the scheduler, such as the list of free identifiers and the queue of processes
ready for scheduling. It can be thought of as a directed, connected, acyclic
graph, whose nodes contain the members of the list. The graph has a single
source and a single sink.

The chain was originally published in an example of the data refinement of
a process scheduler in Woodcock & Davies’s book on Z \[34, Ch. 21\], although
its history precedes this. It started life as part of the formal development of
an industrial real-time operating system, although the complete specification
has never been published for confidentiality reasons. The style of using Z
advocated by Woodcock & Davies separates two concerns in the specification
of datatypes: the successful behaviour of each operation is first specified
as a partial description; then each operation’s precondition is analysed and
appropriate error handling is added. For space reasons, the chain datatype
is left partial without error handling in \[34\]. Craig \[7\] reuses Woodcock &
Davies’s partial specification, with informal analysis and error handling to
make the datatype robust by totalising the operations. In this section, we
repeat Craig’s definition of the totalised chain datatype and formalise its
analysis.

We first define a set of valid process identifiers \(PID\) as a strictly positive
range of integers bounded by \(\text{maxpid} \in \mathbb{N}_1\), where the invalid process identi-
fier \(\text{null}\) is some number outside this range. In \[7, \text{Ch. 3}\], it is given a value
strictly greater than the \(PID\) upper bound \(\text{null} > \text{maxpid}\), whereas in \[34\]
it is defined as zero. The reason for this change is due to Craig’s choice of
refinement towards arrays in C code, which are zero based. A general set of
process identifiers \(GPID\) containing the invalid identifier is also defined.
This is an axiomatic definition: it introduces \textit{maxpid} as a global constant for the specification, constraining its value to be strictly positive. It is said to be \textit{loosely specified}, since although it has a single value, we do not say precisely which one it is. To aid automation in Z/Eves, we prove some obvious facts about such global constants. Although these are often rather trivial, they are useful in increasing the automation of subsequent proofs. For instance, \textit{maxpid} is at least 1. This is blatantly obvious, since it must be a strictly positive integer, but its proof requires facts about integer ranges and finite sets. Having proved it once and for all, the fact is then added to the armoury of lemmas used by the theorem prover.

\begin{verbatim}
theorem rule lMaxPIDPositive
1 \leq \textit{maxpid}
\end{verbatim}

We also prove that \textit{maxpid} is an integer. Once more, this is blatantly obvious, since it is a non-zero natural number, and all non-zero natural numbers are themselves integers.

\begin{verbatim}
theorem grule gMaxpidMaxType
\textit{maxpid} \in \mathbb{Z}
\end{verbatim}

In this paper, theorems such as this have all been proved using Z/Eves version 2.3.1. Next, \textit{PID} is declared as an abbreviation for the set of integers between 1 and \textit{maxpid}.

\begin{verbatim}
PID == 1..\textit{maxpid}
\end{verbatim}

A useful automation fact is that \textit{PID} is non-empty.

\begin{verbatim}
theorem rule lPIDNotEmpty
\neg \textit{PID} = \{\}
\end{verbatim}

The null \textit{PID} is a natural number greater than every other \textit{PID}.

\begin{verbatim}
null : \mathbb{N}
\forall p : \textit{PID} \bullet p < \textit{null}
\end{verbatim}
The label \( \langle \text{disabled rule dnull} \rangle \) gives a name to the fact that \( \forall p : \text{PID} \bullet p < \text{null} \), and identifies it as a kind of rewrite rule that needs to be explicitly invoked. We prove a few obvious theorems.

\textbf{theorem} rule lnullBound
\[ \text{maxpid} < \text{null} \]

\textbf{theorem} rule lnullDisjoint
\[ \forall p : \text{PID} \bullet \neg p = \text{null} \]

\textbf{theorem} rule lNullIsNotPID
\[ \neg \text{null} \in \text{PID} \]

Finally, we introduce \( \text{GPID} \) as including the set \( \text{PID} \) and the \text{null} PID.

\[ \text{GPID} \equiv \text{PID} \cup \{ \text{null} \} \]

And add three small automation theorems.

\textbf{theorem} grule gPIDMaxType
\[ \text{PID} \in \mathbb{P} \mathbb{Z} \]

\textbf{theorem} grule gGPIDMaxType
\[ \text{GPID} \in \mathbb{P} \mathbb{Z} \]

\textbf{theorem} rule lNullIsGPID
\[ \text{null} \in \text{GPID} \]

And so to the details of the chain datatype itself. It has four components: a set of process identifiers arranged in links, with start and end identifiers. By declaration, the start and end identifiers can be \text{null}, the links of the chain are represented by an injective function, and the set contains only non-\text{null} identifiers. The state space of the datatype is specified using one of the most characteristic elements of \( \mathbb{Z} \): the schema. This is a named mathematical structure describing an arbitrary instance of the \text{Chain} with an invariant constraining the relationship between the \text{Chain’s} components. The \text{Chain} schema is declared in Table 1. Six invariants constrain and relate the state components.
1. The \textit{links} function is finite.

2. The \textit{PID} set is finite.

3. The \textit{set} contains all linked \textit{PID}s, as well as the \textit{start} identifier, when it is not representing the empty \textit{Chain} (\textit{i.e.}, $\text{start} = \text{null}$).

4. If there are no links, then the \textit{start} and \textit{end} are the same (either both \textit{null}, or both the same non-\textit{null} identifier).

5. If there are some links, then the \textit{start} and the \textit{end} are the only source and sink (respectively).

6. Finally, every non-\textit{start} identifier can be reached from the \textit{start} by following \textit{links}.

These invariants guarantee that the \textit{Chain} has neither cycles nor repeated links. So, if this list were to be implemented using pointers, no aliasing or self-reference would be allowed.

Note that the only difference from [34, Ch. 21] is the way finiteness is encoded as an invariant, rather than directly on the declared type. This is a technicality that makes mechanisation with Z/Eves easier, due to the way types are handled, although it makes no semantic difference.
Next, we prove some useful properties about Chain: some are quite obvious (i.e., they come directly from the definition); some are stated in [34, Ch. 21] and proved here; others are new. All of them are useful in discharging proofs about a data refinement to Chain made in Craig’s work. Each property has been proved as a theorem and summarised in the Table 2. When links are

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| T1 | \(
\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{start} \not=\) \(\text{null}\) |
| T2 | \(\not\vdash\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{start} \in\) dom links |
| T3 | \(\not\vdash\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{start} \notin\) ran links |
| T4 | \(\not\vdash\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{end} \not=\) \(\text{null}\) |
| T5 | \(\not\vdash\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{end} \notin\) dom links |
| T6 | \(\not\vdash\) links \(\not=\) \(\emptyset\) \(\Rightarrow\) \(\text{end} \in\) ran links |
| T7 | \(\not\vdash\) links \(=\) \(\emptyset\) \(\Rightarrow\) \(\text{start} =\) \(\text{end}\) |
| T8 | \(\not\vdash\) set \(=\) \(\emptyset\) \(\Rightarrow\) \(\text{start} =\) \(\text{end} =\) \(\text{null}\) |
| T9 | \(\not\vdash\) set \(=\) \{ \(\text{start}\) \} \(\Rightarrow\) \(\text{start} =\) \(\text{end} \not=\) \(\text{null}\) |
| T10 | \(\not\vdash\) (\(\text{start},\) \(\text{end}\)) \(\in\) links \(\Rightarrow\) links \(=\) \{(\(\text{start},\) \(\text{end}\))\} |

Table 2: Properties about Chain elements: start and end points

not empty, start is a valid PID (T1), and has an outgoing link (T2) but no incoming links (T3), as the Chain is acyclic and with a single starting point. Conversely, when links is not empty, end is valid (T4), has no outgoing link (T5), and has an incoming link (T6). When links is empty, we have either a singleton or an empty Chain (T7). In the former, we have that start = end \(\not=\) null, whereas in the latter we have that start = end = null. When the set of identifiers in the Chain is empty, we know that the Chain is empty, hence both start and end are the invalid identifier (T8). Similarly, if set contains only a valid start point, it represents a singleton Chain, and this start point is equal to the valid end point T9. Finally, when start is linked to end, they are both valid, as enforced by the type of links, and we have a Chain with two elements and one link (T10). It cannot be the singleton Chain because if start = end, the invariant about the uniqueness of start and end would not be satisfied. This theorem could be strengthened to an equivalence.

The properties in Table 2 are useful in proving properties about the Chain datatype, such as operation preconditions, characterising mid-point
The proofs for T1–T9 are relatively simple: by extensionality on the properties of the Chain datatype for start and end points when \( \text{links} \neq \emptyset \). In particular, Theorem T10 about a Chain with exactly two elements was useful when analysing all possible cases for some of the Chain datatype’s operations, such as deleting a valid point at the end (i.e., \( p? \in \text{PID} \land p? \neq \text{start} \land p? = \text{end} \)). It was the most complex to prove of those in the table, and relied on some theorems listed in Table 3, which are needed for the consistency checks for the initialisation, push, pop, and delete operations on chains, described in the next section.

\[
\begin{align*}
\text{T11} & \vdash \text{null} \notin \text{dom links} \cup \text{ran links} \\
\text{T12} & \vdash p? \in \text{set} \setminus \{ \text{start} \} \Rightarrow \text{links} \neq \emptyset \\
\text{T13} & \vdash \text{links} \neq \emptyset \land p? \in \text{set} \setminus \text{ran links} \Rightarrow p? \in \text{dom links} \\
\text{T14} & \vdash p? \in \text{set} \setminus \{ \text{start}, \text{end} \} \Rightarrow p? \in \text{dom links} \cap \text{ran links} \\
\text{T15} & \vdash \text{end} \neq \text{null} \land p? \notin \text{set} \Rightarrow \text{links} \cup \{(\text{end}, p?)\} \in \text{PID} \mapsto \text{PID}
\end{align*}
\]

Table 3: Properties about Chain operations over a given valid identifier

Each theorem in Table 3 involves a Chain and a valid input identifier \( (p? \in \text{PID}) \), except the first one, which just states the fact that invalid identifiers are never linked (T11). Whenever we have any valid linked element \( p? \) that is not the start point, we have something linked (T12). Whenever we have identifiers without incoming links \( (p? \notin \text{ran links}) \), there are outgoing links \( (p? \in \text{dom links}) \) (T13). Although this last statement seems obvious, its proof is not as straightforward as one might expect, yet this fact is crucial for future proofs related to linked elements. For some operations, such as deletion, it is important to characterise properties about a mid-point in the Chain. These properties, as well as lemmas about stepwise traversal (see Table 6), were needed, despite the fact an incremental search operation has not yet been modelled here. One such property is that a valid mid-point element must be neither the start nor the end points, and must have incoming and outgoing links (T14). Other interesting properties about Chain mid-points are defined in Section 6.2 below. Finally, the operation that appends a new valid element \( (p? \in \text{PID} \land p? \notin \text{set}) \) to the end of a non-empty Chain (one with a valid end-point) requires that injectivity is maintained after the update (T15). That means we need to ensure such an update keeps the links injective, as the theorem’s conclusion states.
The proofs of Theorems T11–T15 are more elaborate and require some intuition about case splitting and the identity of witnesses in the last theorem. Also, the mid-point theorem requires information about a general transitive closure property that some element in the transitive closure of a homogeneous relation is in the range of the relation before the closure operation (see Theorem T20 in Table 5 below). More details on general lemmas about transitive closure proofs required for the \textit{Chain} datatype can be found in [11, 12].

6 Chain operations

Some minor syntactic changes from the original report [11] are used in this section. Firstly, we define a free type containing the report on successful and error cases.

\[\text{ChainErrMsg} ::= \text{chain\_ok} | \text{push\_known} | \text{pop\_empty} | \text{del\_unknown}\]

The \textit{Chain} is initialised by setting the \textit{start} and \textit{end} points to be \textit{null}. This under-specification of initialisation says nothing about \textit{links'}, enabling an implementation to choose any initial value for \textit{links'} that satisfies the \textit{Chain} invariant. Obviously, this requires that \textit{links'} = \emptyset, which is used in the initialisation proof that establishes the \textit{Chain} invariant. Most \textit{Chain} operations share the same signature containing the before and after state (\(\Delta \text{Chain}\)), together with a valid identifier given as input (\(p? \in \text{PID}\)). The first operation after initialisation is to \textit{Push} an element onto the \textit{Chain}, where we have two cases. When the \textit{Chain} is empty and the original \textit{end} point is invalid, the operation updates the final \textit{end}' point as the given identifier \(p?\). Otherwise, when the \textit{Chain} is not empty, the \textit{links} get updated by mapping the previous (valid) \textit{end} point to the new given input \(p?\) that becomes \textit{end}'. Note that this operation will require a proof obligation that \textit{links'} remains injective, hence the need for Theorem T16 from Table 3 above. The successful behaviour of the \textit{Push} operation, \textit{PushOkay}, is the disjunction of these two cases. We follow for all operations the style for Z specifications advocated in [34, 32].

\[
\begin{align*}
\text{ChainOp} & \equiv [\Delta \text{Chain}; p? : \text{PID}] \\
\text{ChainInit} & \equiv [\text{Chain'} | \text{start}' = \text{end}' = \text{null}] \\
\text{PushEmpty} & \equiv [\text{ChainOp} | \text{end} = \text{null} \land \text{end}' = p? \land \text{links}' = \text{links}] \\
\end{align*}
\]
Next, we define the error case for Push, requiring a general signature for reporting success or error messages. When successful, an operation signals a chain_ok message, whereas upon failure, nothing changes (\exists Chain) and some error message (msg!) is output. Since the Chain is injective, whenever the given input p? is already present, an error message is raised. The total Push operation is the disjunction between the successful cases and the success report, and the single error case on known elements. The precondition is given in Table 4.

\[
\begin{align*}
\text{PushNonEmpty} & \triangleq [\text{ChainOp} \mid \text{end} \neq \text{null} \land \text{links}' = \text{links} \cup \{(\text{end} \mapsto p?)\}] \\
\text{PushOkay} & \triangleq \text{PushEmpty} \lor \text{PushNonEmpty}
\end{align*}
\]

The next operation pops an element from the Chain. The original Chain defines two cases: popping the singleton Chain, and popping the Chain with multiple elements. In the singleton case, we expect at least the start point to be valid, where the links are empty. The result of the operation is to keep the links constant, invalidate the start point, and return on p! the original starting point being popped.

\[
\begin{align*}
\text{PopSingleton} \\
\begin{array}{l}
\Delta \text{Chain};
p! : \text{PID} \\
\text{start} \neq \text{null} \land \text{links} = \emptyset \land \text{links}' = \text{links} \land \text{start}' = \text{null} \land p! = \text{start}
\end{array}
\end{align*}
\]

The other case where the Chain has multiple elements, expects links to be non-empty, and the start point is output on p!. The operation updates the new starting point (\text{start}') to be the identifier linked to the original starting point (\text{links start}), and the resulting \text{links}' has the original starting point removed from its domain (\triangle). Since links is not empty, then start must be valid and with outgoing links, as Theorems T1 and T2 in Table 2 show. This is important to discharge the proof obligations generated by this operation that \text{start} \neq \text{null} since p? \in \text{PID}, and that \text{start} is linked
(start ∈ dom links). This also covers the case where a Chain of two elements becomes a singleton Chain.

\[
\text{PopMult} \\
\Delta \text{Chain}; \ p! : \text{PID} \\
\text{links} \neq \emptyset \land p! = \text{start} \land \text{start}' = \text{links start} \land \text{links}' = \{ \text{start} \} \triangleleft \text{links}
\]

Similarly, the successful behaviour of the Pop operation, PopOkay, is the disjunction of these two cases.

\[
\text{PopOkay} \equiv \text{PopSingleton} \lor \text{PopMult}
\]

Now, what happens when one tries to pop an element from an empty Chain? The Chain is empty when the starting point is invalid, in which case nothing changes and we return an error. The Chain has only one link when the element linked with the starting point has no outgoing links (\text{links start} \notin \text{dom links}). Theorem T2 from Table 2 says that when \text{links} is not empty and \text{start} has outgoing links we can apply \text{links} to \text{start}.

\[
\text{PopEmptyErr} \equiv [ \text{ChainErr} \mid \text{start} = \text{null} \land \text{msg}! = \text{pop_empty} ]
\]

Now we can assemble the pop operation.

\[
\text{Pop} \equiv (\text{PopOkay} \land \text{ChainSuccess}) \lor \text{PopEmptyErr}
\]

Our final operation deletes elements at the start, end, and middle of the Chain. Deleting at the start is just like popping, hence it is defined in terms of PopOkay, but with a different signature: the input given (p?) must be the start point, and PopOkay takes place. That is, either \(p? = p! = \text{start}\) when the Pop operation succeeds on either PopSingleton and PopMult (msg! = chain_ok), or \(p? = \text{start}\) and some error case is returned on msg!.

\[
\text{DelStart} \\
\text{ChainOp} \\
p? = \text{start} \land (\exists p! : \text{PID} \bullet \text{PopOkay})
\]

Deleting at the end means the Chain cannot be empty. And through Theorem T4 from Table 2, we have that end must not be null, which is necessary
since \( p? \in PID \), hence \( \text{links} \) is not empty. The result of the operation is to remove the link leading to the \( \text{end} \) from the range of \( \text{links} \) (\( \triangleright \)). This makes \( \text{end}' = \text{links}^{-} \text{end} \), which is okay, since from Theorem T6, \( \text{end} \) is in the range of \( \text{links} \), whenever \( \text{links} \) is not empty, which is true since \( \text{end} \) is valid.

\[
\begin{array}{l}
\text{DelEnd} \\
\text{ChainOp} \\
p? \neq \text{start} \land p? = \text{end} \land \text{links}' = \text{links} \triangleright \{ \text{end} \}
\end{array}
\]

The most complicated case is deleting a mid-point element. Mid-points are all those known elements (within \( \text{set} \)) that are neither the \( \text{start} \) nor the \( \text{end} \), as characterised by Theorem T14 in Table 3 above. The update on \( \text{links}' \) is defined by first removing (\( \triangleleft \)) the mid-point (\( p? \)) from the domain of \( \text{links} \), and then updating the resulting function with a new pair using relational overriding (\( \oplus \)). The new pair makes the thing that pointed to \( p? \) (i.e., \( \text{links}^{-} p? \)) point to the thing that \( p? \) pointed to (i.e., \( \text{links} p? \)).

\[
\begin{array}{l}
\text{DelMiddle} \\
\text{ChainOp} \\
p? \in \text{set} \setminus \{ \text{start}, \text{end} \} \\
\text{links}' = (\{ p? \} \triangleleft \text{links}) \oplus \{ (\text{links}^{-} p? \mapsto \text{links} p?) \}
\end{array}
\]

This operation relies on a series of interesting properties about \( \text{Chain} \): \( \text{links} p? \) must be unique within \( \text{links} \), otherwise the update will breach the injectivity of \( \text{links}' \); the mid-point \( p? \) must be both in the domain and range of \( \text{links} \), otherwise we could not apply it to \( \text{links} \) and its inverse present on the new pair; and so on. This comes from theorem T15 in Table 3. We totalise the \( \text{Delete} \) operation with an error case when an element to be deleted is unknown (\( p? \notin \text{set} \)). This is defined next with the appropriate error message, where the total \( \text{Delete} \) operation is defined as the disjunction of all cases

\[
\text{DeleteOkay} \equiv (\text{DelStart} \lor \text{DelEnd} \lor \text{DelMiddle}) \land \text{ChainSuccess}
\]

\[
\text{DelUnknownErr} \equiv [\text{ChainErr} \mid p? \notin \text{set} \land \text{msg}! = \text{del}_\text{unknown}]
\]

\[
\text{Delete} \equiv \text{DeleteOkay} \lor \text{DelUnknownErr}
\]

And this completes the description of the operations.
6.1 Chain operations preconditions

In addition to the signature containing the before state for Chain and a valid identifier input \((p? \in PID)\), a precondition is required for each operation, and these are summarised in Table 4. We also prove that the Chain can be

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushEmpty</td>
<td>(\text{end} = \text{null})</td>
</tr>
<tr>
<td>PushNonEmpty</td>
<td>(\text{end} \neq \text{null} \land p? \notin \text{set})</td>
</tr>
<tr>
<td>PushOkay</td>
<td>(p? \notin \text{set})</td>
</tr>
<tr>
<td>PushKnownErr</td>
<td>(p? \in \text{set})</td>
</tr>
<tr>
<td>Push</td>
<td>(\text{true})</td>
</tr>
<tr>
<td>PopSingleton</td>
<td>(\text{start} \neq \text{null} \land \text{links} = \emptyset)</td>
</tr>
<tr>
<td>PopMult</td>
<td>(\text{links} \neq \emptyset)</td>
</tr>
<tr>
<td>PopOkay</td>
<td>(\text{links} = \emptyset \Rightarrow \text{start} \neq \text{null})</td>
</tr>
<tr>
<td>PopEmptyErr</td>
<td>(\text{start} = \text{null})</td>
</tr>
<tr>
<td>Pop</td>
<td>(\text{true})</td>
</tr>
<tr>
<td>DelStart</td>
<td>(p? = \text{start})</td>
</tr>
<tr>
<td>DelEnd</td>
<td>(p? \neq \text{start} \land p? = \text{end})</td>
</tr>
<tr>
<td>DelMiddle</td>
<td>(p? \in \text{set} \setminus { \text{start, end} })</td>
</tr>
<tr>
<td>DelUnknownErr</td>
<td>(p? \notin \text{set})</td>
</tr>
<tr>
<td>DeleteOkay</td>
<td>(p? \in \text{set})</td>
</tr>
<tr>
<td>Delete</td>
<td>(\text{true})</td>
</tr>
</tbody>
</table>

Table 4: Chain operations preconditions

initialised: \(\exists \text{Chain}' \odot \text{ChainInit}\), hence we can establish the state invariant. This proof is easily discharged with the empty set for both \(\text{links}'\) and \(\text{set}'\). These precondition proofs were the most rewarding part in increasing our understanding of the Chain datatype, and its role in the refinement path from an abstract specification down to C code for Craig’s operating system kernels [7].

6.2 Properties of chain operations

Many of the theorems from Tables 2 and 3 come from the consistency checks for the Chain operations and some of the easier precondition proofs. Other more complex precondition calculations gave rise to even more interesting
properties about the *Chain* datatype, as well as general properties for injective functions (_ → _ ) and transitive closure ( _ + _ ). Table 5 summarises some

Table 5: Chain and general mathematical properties

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T16</td>
<td>s ∈ dom f ( f s \neq e \land (s, e) \in f^+ \Rightarrow (f s, e) \in f^+ )</td>
</tr>
<tr>
<td>T17</td>
<td>s ∈ dom f ( f s \neq e \land (s, e) \in f^+ \Rightarrow (f s, e) \in ({ s } \triangleleft f)^+ )</td>
</tr>
<tr>
<td>T18</td>
<td>x ∈ dom f ( \Rightarrow \ran ({ x } \triangleleft f) = \ran f \setminus { f x } )</td>
</tr>
<tr>
<td>T19</td>
<td>y ∈ ran f ( \Rightarrow \dom (f \triangleright { y }) = \dom f \setminus { f y } )</td>
</tr>
<tr>
<td>T20</td>
<td>(x, y) ∈ R^+ ( \Rightarrow y \in \ran R )</td>
</tr>
<tr>
<td>T21</td>
<td>(x, y) ∈ R ( \Rightarrow (x, y) \in R^+ )</td>
</tr>
<tr>
<td>T22</td>
<td>s \notin \ran R ( \Rightarrow ({ s } \triangleleft R)^+ = ({ s } \triangleleft R) )</td>
</tr>
<tr>
<td>T23</td>
<td>e \notin \dom R ( \Rightarrow (R \triangleright { e })^+ = R^+ \triangleright { e } )</td>
</tr>
<tr>
<td>T24</td>
<td>(x, z) \notin R^+ \land (x, y) \in R^+ ( \Rightarrow (x, z) \in (R \cup { (y, z) })^+ )</td>
</tr>
</tbody>
</table>

of them. In Theorem T16, f is a successor function. If e is a successor of s, but not the immediate successor, then e is also a successor of f s, maybe even the immediate one. Theorem T17 is a similar property, but involves removing a starting point. These are useful when proving properties about operations that might need to traverse the *Chain*. The next seven theorems are about mathematical properties for injective functions and transitive closure in general. Theorems T18 and T19 concern the range of domain anti-restriction and the domain of range anti-restriction, two laws absent from the mathematical toolkit. This is useful because there are more proved lemmas in the Standard Z and Z/Eves mathematical toolkits [20, 28] about domain, range, set difference, and singleton sets. In fact, all the theorems in this table reduce to goals with more known lemmas, or else are lemmas weakening the original goal. Theorem T20 is the first on transitive closure, and it states that an element of a transitively closed relation is within the original relation’s range. Lemmas about transitive closure are difficult to prove for several reasons. Transitive closure is defined as the smallest set containing a transitive extension of the original relation, and this does not provide a natural automation in Z/Eves. Perhaps because of this, there is a general lack of good lemmas about transitive closure. In particular, there are no induction principles. Theorem T21 is quite simple, yet very useful: transitive closure contains the original relation. Theorems T22 and T23 are about distributing
transitive closure through domain and range anti-restriction over a singleton set. Finally, Theorem T24 is an induction principle.

Next, we provide an overview of the complexity involved in the proof of each precondition calculation. The calculation for the Push operations were the smallest, although even these were non-trivial. PushEmpty relied on: Theorem T11 about Chain, and PushNonEmpty relied on Theorems T1–T6, T11, and T15 about Chain, and Theorems T21 and T24 about transitive closure.

Proofs for pop operations were an order of magnitude more difficult than push operations, mainly due to PopMult. This operation relied on Theorems T1–T6, T11, and T15 about Chain, Theorems T16 and T17 about the properties on traversing a Chain, Theorem T18 about injections, and Theorems T20 and T22 about transitive closure.

The proof of operation DelStart is lengthy but not difficult, as it mostly relies on the proof for Pop. Like PopMult, DelEnd was also complex and relied on another series of facts: Theorems T1–T6, and T10 about Chain, Theorem T16 about Chain traversal, Theorem T19 about injections, and Theorems T20 and T23 about transitive closure. The DelMiddle operation is a different and much more complex story. It relies on more elaborate properties and proof strategy, and we devote a entire section to it below.

The remaining operations are laborious but straightforward. Error cases are easy, as the state remains constant (ΞChain). All the transitive closure properties above, as well as a plethora of general lemmas, were also proved and became a byproduct of this exercise and are available elsewhere [15, 11].

6.3 Characterising Chain mid-points

We use the next schema to characterise what a mid-point is in the deletion operation. A process identifier p? is a mid-point (more properly, an interior element) of a chain providing it has an incoming and an outgoing link. This requires the chain to have at least two links and three elements. ChainMidPoint asserts that the incoming link comes from x and the outgoing link goes to y.
Note that it follows from \texttt{ChainMidPoint} that \(x = \text{links} \preceq p?\) and \(y = \text{links} \preceq p?\). This characterisation was originally crafted in order to calculate the precondition for \texttt{DelMiddle}, yet it can be reused to prove properties about any operation relying on mid-points, such as incremental searching or sorting. To prove the precondition of \texttt{DelMiddle}, we still need extra properties about injections and \texttt{Chain} mid-point manipulation, and they are given in Table 6. Theorem T25 is a general lemma on the conditions for associativity

\begin{table}[h]
\centering
\begin{tabular}{c}
\hline
\texttt{ChainMidPoint} \hfill \texttt{Chain: } x, y, p? : PID \\
\hline
\texttt{links} \neq \emptyset \land p? \in (\text{dom links} \cap \text{ran links}) \setminus \{\text{start, end, } x, y\} \\
(x, p?) \in \text{links} \land (p?, y) \in \text{links} \land (x, y) \notin \text{links} \\
\hline
\end{tabular}
\end{table}

Table 6: Properties useful for \texttt{Chain} mid-points

between domain anti-restriction (\(\preceq\)) and relational overriding (\(\oplus\)) on singleton sets: we can remove \(a\) from the domain and then update \(b\) with \(c\) or vice-versa, provided the domain elements are different. Theorem T26 states that the mid-point update performed by \texttt{DelMiddle} (or any similar mid-point operation) preserves injectivity. Theorem T27 provides a weakening lemma about the resulting range from the mid-point update on a given injective function. It is useful as it simplifies reasoning from union, range, domain anti-restriction, and overriding, to union and range only. In fact, we needed some extra lemmas to show that the mid-point partitions the \texttt{Chain} from the start up to (but not including) the mid-point \((p?)\), the two links related to the mid-point in Theorem T28, and the remaining links from \(y\) to the \texttt{end}. In this characterisation, various cases arise, such as the mid-point just after
start or right before the end, as well as a mid-point away from the extremities of the Chain. All this can be found in [11, 12].

7 Interesting lessons

Mechanising the verification of the Chain datatype has led to a deeper understanding of the datatype and its properties. We attempted simplifications by weakening some of the invariants that could have been derived as properties, in order to make our proofs simpler, but without compromising the specification. Something similar has been attempted by Craig [7, Ch. 3], but the lack of mechanisation has resulted in his work missing some important invariants.

We start by analysing each invariant related to a Chain with multiple elements (links ≠ ∅), which we summarise in Table 7. After a little thought,

<table>
<thead>
<tr>
<th>Description</th>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>start ∈ dom l \ ran l</td>
<td>no incoming points</td>
</tr>
<tr>
<td>I2</td>
<td>end ∈ ran l \ dom l</td>
<td>no outgoing points</td>
</tr>
<tr>
<td>I3</td>
<td>l ∈ PID ↦ PID</td>
<td>unique outgoing points</td>
</tr>
<tr>
<td>I4</td>
<td>set = dom l ∪ ran l ∪ ({start} \ {null})</td>
<td>mult., single, empty</td>
</tr>
<tr>
<td>I5</td>
<td>∀ e : set • e ≠ start ⇒ start ↦ e ∈ l⁺</td>
<td>connected elements</td>
</tr>
<tr>
<td>I6</td>
<td>{start} = dom l \ ran l</td>
<td>unique outgoing only</td>
</tr>
<tr>
<td>I7</td>
<td>{end} = ran l \ dom l</td>
<td>unique incoming only</td>
</tr>
<tr>
<td>I8</td>
<td>l ∈ PID ↦ PID</td>
<td>unique incoming points</td>
</tr>
</tbody>
</table>

Table 7: Chain invariants analysis summary

it is clear that we do not need all of the invariants for the chain datatype. For example, if we have an injection whose graph is connected, and for which there is at least one source and at least one sink node, then there will be exactly one source and exactly one sink node. The invariants for the original datatype were not intended to be minimal. But if we were to find a smaller collection of invariants, then we would simplify our proofs, at the expense of proving that some invariants follow from the others.

We would like every element to lie on a straight-line path between start and end in the directed graph represented by links. For this to be true, every element must be reachable from the start (I5). I1 guarantees that
start is actually a source node, although not necessarily the unique one. I2 guarantees that end is actually a sink node, although also not necessarily the unique one. Now suppose that there is a divergence somewhere along the path: alternative different paths may lead to cycles or to sinks other than end. Divergence is forbidden by links being a function (I3). This makes links a tree, rooted at the end; thus, it makes end the unique sink for links. Now suppose that there is a convergence somewhere along the path. This must arise from a source other than start, rather than from a cycle, for otherwise there would have to be a divergence somewhere, and this has been ruled out. So we can prevent convergence of paths by one of two means: either require start to be the unique source (I6), or require links to be injective (I8).

It would have been very attractive to have eliminated the invariant requiring links to be injective, as this has provided much of the complication in the proofs.

7.1 Going back to the scheduler design

The Chain datatype presented here is part of the process table within the model for an OS separation microkernel. Craig starts by designing a simple kernel, and later develops this into a separation kernel. In a separation kernel all user process address spaces are disjoint, and all process execution times are disjoint. Following the model for a process table using our Chain datatype, a following priority queue is built using the process table operations in order to form the microkernel scheduler. Other familiar OS components are also modelled, such as a global semaphore table, a synchronous message passing system, a process sleeping mechanism, and so on. The model is for an embedded or real-time system application, where unique identifiers are sequentially allocated for every process, where an idle process is allocated first, and an initial (system) process is allocated second.

The process table is used to represent both these two initial processes, and the user processes in a uniform representation. It is implemented as an array-based structure similar to Linux, with the following structure: a stack pointer is used for context switching; priority is used for sorting the scheduler’s ready queue; a state component records whether the process is running, waiting, has terminated, and so on; an incoming message queue records all pending messages; and a waking time component records how long should a process sleep until it becomes ready for scheduling again. The scheduler itself is a simple priority ready queue, which is refined to a chain
of process identifiers with injective mapping encoded in the process table. Rescheduling occurs when a non-empty ready queue is present, where the current process’s priority is lower than ready queue head, or the current process is neither ready nor running. Again, our \textit{Chain} datatype is present in the refinement given by Craig [7].

Other services like semaphores, message passing, and system calls are also available. Processes can synchronise using counting semaphores, FIFO queues, and so on. They again are defined as separate mathematical datatypes, later refined to a chain of process identifiers. Message passing enables processes to exchange messages, where the discipline that receivers wait and senders retry is observed. Finally, system calls can be used to: create or terminate processes; retrieve process identifiers; send or receive synchronous messages; allocate and release semaphores; put processes to wait or to sleep, as well as signal them; and so forth.

8 Conclusions

The Grand Challenge’s pilot projects [19, 33] help us to learn the best ways to model various application domains and how to verify those models. The intention is to make it easier for the next team who want to work in the application domain.

The experiments that started with mechanising Craig’s refinement of simple OS kernel schedulers [7] led to the mechanisation of a more general \textit{Chain} datatype. This in itself instigated thinking about general properties for injective functions and transitive closure, and started two reports [11, 12] that are good candidates to become part of the VSR as reusable mathematical datatypes. Our formalisation of [7, Ch. 3] is discussed in [35]. It contains a series of declarations from Craig’s book, and 38 mechanically verified theorems. For the \textit{Chain} datatype, we have mechanised the whole Section 21.3 from [34], with some 59 extra theorems as properties and mechanisation lemmas [12]. The general theories [11] contain 124 theorems about various mathematical datatypes, including injections and transitive closure.

We improved the specification of the \textit{Chain} datatype [34, Ch. 21], which is used in [7, Ch. 3], by identifying useful properties and verifying consistency by calculating the preconditions for each operation. This mechanisation enabled both a better understanding of the datatype, and a clearer definition of the separation kernel’s scheduler specification use of it. As its use in Craig’s
book had some mistaken simplifications of this central datatype, as well as the missing error cases from the original specification here uncovered, we believe this to be an important contribution in building general theories for formal modelling of OS kernels.

Acknowledgements We are grateful to QinetiQ Malvern for its long term support for our research work. An early version of the chain was discussed with Roger Shaw during a VDM course we gave together at St Hughes College Cambridge in the 1980s. It was discussed extensively with Jason Hulance and Michael Goldsmith at Formal Systems (Europe) Ltd as part of our work for an external client. We are also thankful to Juan Perna for many fruitful discussions about properties of transitive closure and the Chain datatype. Finally, we are grateful to Iain Craig for producing such a useful account of the formal specification and refinement of OS kernels.

An early version of this paper was presented at the Brazilian Formal Methods Symposium in Ouro Preto in 2007.

References


[26] M. V.M. Oliveira, M. A. Xavier, and A. L. C. Cavalcanti. Refine and
Gabriel: Support for Refinement and Tactics. In J. R. Cuellar and Z. Liu,
editors, 2nd IEEE International Conference on Software Engineering
and Formal Methods, pages 310–319. IEEE Computer Society Press,
September 2004.

[27] Tahina Ramananandro. Mondex, an electronic purse: Specification and
refinement checks with the Alloy model-finding method. Formal Aspects

[28] Mark Saaltink. Z/Eves 2.0 Mathematical Toolkit. ORA Canada, October

5493-06a.


[31] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse:
Specification, refinement, and proof. Technical Monograph PRG-126,
Oxford University Computing Laboratory, July 2000.


[34] Jim Woodcock and Jim Davies. Using Z: Specification, Refinement, and

[35] Jim Woodcock, Leo Freitas, and Iain Craig. A Verified Simple Operating
System Kernel. In Workshop on the Verified Software Repository as part
of FM Symposium, Turku, Finland, 2008.

[36] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John FitzGER-
ald. Formal methods: Practice and experience. ACM Computing Sur-