Formal modelling of separation kernel components

Andrius Velykis and Leo Freitas

University of York

ICTAC 2010  |  September 1-3, 2010  |  Natal, Brazil
The need for verified kernel

We assume kernel correctness when verifying applications. Wrong assumption — invalid verification.

Verified application

Operating system | OS kernel

Hardware
Different approaches to verification
Different approaches to verification

Program verification

Verify functional and security properties of an existing kernel.

Specification / properties

Implementation

Abstract modelling & refinement

Reason about kernel properties without implementation, then perform data refinement.

Abstract formal model

Concrete formal model

Implementation code

Kernel verification projects

- Microsoft Hyper-V hypervisor
- FreeRTOS
- Verisoft XT
- seL4/L4.verified
  (verification of a Haskell prototype via its abstract specification & refinement)

Separation kernel
Program verification

Verify functional and security properties of an existing kernel.

Specification / properties

verification

Implementation
Kernel verification projects

- Microsoft Hyper-V hypervisor
- FreeRTOS
- Verisoft XT
- seL4/L4.verified

(verification of a Haskell prototype via its abstract specification & refinement)
Different approaches to verification

Program verification
Verify functional and security properties of an existing kernel.

- Specification / properties
- Implementation
- Verification

Abstract modelling & refinement
Reason about kernel properties without implementation, then perform data refinement.

- Abstract formal model
- Concrete formal model
- Prove properties
- Code generation
- Implementation code

Kernel verification projects
- Microsoft Hyper-V hypervisor
- FreeRTOS
- Verisoft XT
- seL4/L4.verified
  (verification of a Haskell prototype via its abstract specification & refinement)
Abstract modelling & refinement

Reason about kernel properties without implementation, then perform data refinement.

**Abstract formal model** ⇔ **Prove properties**

refinement

Concrete formal model

code generation

Implementation code
Project foundations

- Separation kernel
- Craig's formal specification
- Grand Challenge (GC) in Verified Software
**Project foundations**

- Separation kernel
- Craig's formal specification
- Grand Challenge (GC) in Verified Software

---

**Separation kernel**

A secure kernel architecture that simulates a distributed environment— all processes are separated (Rushby, 1981).

- Hypervisors (virtualisation)
- Embedded secure microkernels, etc.

---

**Craig's kernels**

Formal specification and refinement of abstract kernels (Craig, 2007):

- simple kernel & separation kernel;
- main concepts modelled;
- hardware assumptions.

**Specifications**

- Type set by hand in Z notation
- Several manual proofs

---

**Grand Challenge**

**Verified OS kernels** project is part of the Grand Challenge in Verified Software.

Verification of simple kernel components (Freitas, 2009).

**Pilot projects**

- Mondex smartcards;
- POSIX filestore;
- Flash memory;
- OS kernels, etc.

---

**Formal Z specification**

- Data structures
- General lemmas
- Proof tactics
Separation kernel

A secure kernel architecture that simulates a distributed environment — all processes are separated (Rushby, 1981).

Used as:
- Hypervisors (virtualisation)
- Embedded secure microkernels, etc.

Craig's kernels

Formal specification and refinement of abstract kernels (Craig, 2007):

- simple kernel & \textit{separation kernel};
- main concepts modelled;
- hardware assumptions.

\textbf{Specifications}

- Typeset by hand in \textit{Z notation}
- Several manual proofs

\textbf{Scope}

- All necessary kernel components
- Some details specified as natural language (English) comments

Specifications

• Typeset by hand in Z notation
• Several manual proofs
Scope

- All necessary kernel components
- Some details specified as natural language (English) comments
Craig's kernels

Formal specification and refinement of abstract kernels (Craig, 2007):

- simple kernel & separation kernel;
- main concepts modelled;
- hardware assumptions.

Specifications
- Typeset by hand in Z notation
- Several manual proofs

Scope
- All necessary kernel components
- Some details specified as natural language (English) comments

Grand Challenge

Verified OS kernels project is part of the Grand Challenge in Verified Software. Verification of simple kernel components (Freitas, 2009).

Pilot projects

- Mondex smartcards;
- POSIX filestore;
- Flash memory;
- OS kernels, etc.

Separation kernel

A secure kernel architecture that simulates a distributed environment — all processes are separated (Rushby, 1981).

Used as:
- Hypervisors (virtualisation)
- Embedded secure microkernels, etc.

Craig's kernels

Formal specification and refinement of abstract kernels (Craig, 2007):
- simple kernel & separation kernel;
- main concepts modelled;
- hardware assumptions.

Specifications
- Typeset by hand in Z notation
- Several manual proofs

Scope
- All necessary kernel components
- Some details specified as natural language (English) comments

Rushby, J.M.: Design and verification of secure systems.

Separation kernel model

The model is based on Craig's specification and includes components to satisfy main separation properties.
Messaging
Process communication via **established channels** only.
Memory management

Spatial separation — no direct cross memory access.
Platform & devices

Trusted (verified) underlying platform and device processes.
Scheduling

Temporal separation — no two processes are executing at the same time.
**Messaging**
Process communication via **established channels** only.

**Scheduling**
**Temporal separation** — no two processes are executing at the same time.

**Memory management**
**Spatial separation** — no direct cross memory access.
Craig's kernels

Formal specification and refinement of abstract kernels (Craig, 2007):

- simple kernel & separation kernel;
- main concepts modelled;
- hardware assumptions.

Specifications

- Typeset by hand in Z notation
- Several manual proofs

Scope

- All necessary kernel components
- Some details specified as natural language (English) comments

Grand Challenge

**Verified OS kernels** project is part of the Grand Challenge in Verified Software. Verification of **simple kernel** components (Freitas, 2009).

**Pilot projects**

- Mondex smartcards;
- POSIX filestore;
- Flash memory;
- OS kernels, etc.

Formal Z specification

- Data structures
- General lemmas
- Proof tactics

Model considerations

Formal model foundations

Verified software repository
Specifications
- Typeset by hand in Z notation
- Several manual proofs

Scope
- All necessary kernel components
- Some details specified as natural language (English) comments


Freitas, L.: Mechanising data-types for kernel design in Z. In: Foundations and Applications. Volume 5902 of LNCS.,

Formal Z specification
- Data structures
- General lemmas
- Proof tactics

Model considerations

Formal model foundations
Z notation

Formal specification language, based on ZF set theory and mathematical logic.

- Schema — labelled record with invariants
- Mathematical logic reasoning & proof
- Refinement techniques

State & operations approach

<table>
<thead>
<tr>
<th>Schema</th>
<th>State</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarations</td>
<td>variables</td>
<td>State; State'</td>
</tr>
<tr>
<td>predicates</td>
<td>invariants</td>
<td>input/output variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state relationship invariants</td>
</tr>
</tbody>
</table>
State & operations and

- Schema
  - declarations
  - predicates

- State
  - variables
  - invocations
State
variables
invariants
Operation

State; State'

input/output variables

state relationship invariants
Z notation

Formal specification language, based on ZF set theory and mathematical logic.

- Schema — labelled record with invariants
- Mathematical logic reasoning & proof
- Refinement techniques

State & operations approach

<table>
<thead>
<tr>
<th>Schema declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarations</td>
</tr>
<tr>
<td>predicates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
</tr>
<tr>
<td>invariants</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>State; State'</td>
</tr>
<tr>
<td>input/output variables</td>
</tr>
<tr>
<td>state relationship invariants</td>
</tr>
</tbody>
</table>
Z/Eves theorem prover

A quantified first-order logic theorem prover for $Z$. Similar to the Boyer-Moore family of provers.

- Low learning threshold
- Restricted number of proof tactics
- Syntax & type checker for $Z$
- Successful use in previous GC projects
Z notation

Formal specification language, based on ZF set theory and mathematical logic.

- Schema — labelled record with invariants
- Mathematical logic reasoning & proof
- Refinement techniques

State & operations approach

<table>
<thead>
<tr>
<th>Schema</th>
<th>State</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarations</td>
<td>variables</td>
<td>State, State'</td>
</tr>
<tr>
<td>predicates</td>
<td>invariants</td>
<td>input/output variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state relationship invariants</td>
</tr>
</tbody>
</table>

Z/Eves theorem prover

A quantified first-order logic theorem prover for Z. Similar to the Boyer-Moore family of provers.

- Low learning threshold
- Restricted number of proof tactics
- Syntax & type checker for Z
- Successful use in previous GC projects

Mechanisation stages

Iterative mechanisation of the formal Z specification.

1. Parsing & typechecking
Mechanisation stages

Iterative mechanisation of the formal Z specification.

1. Parsing & typechecking

Typeset specification as Z LaTeX document and parse using Z/Eves.

A number of syntax and type errors corrected in the original hand-written specification.

2. Domain & axiomatic checking

Domain checks verify that functions are applied within their domain. Generated automatically by Z/Eves.

Verify global consistency in axiom declarations.

\texttt{theorem tPIDConsistency} \\
\exists \text{minpid}, \text{maxpid} : \mathbb{N} \cdot \text{minpid} \leq \text{maxpid} \\
\hline \\
\text{minpid, maxpid} : \mathbb{N} \\
\text{minpid} \leq \text{maxpid} \\
\hline \\
axiomatic declaration

3. Feasibility & preconditions

Check the state invariant to verify model feasibility.

Calculate the operation precondition and verify a valid operation after state.

\texttt{theorem tOperationPre} \\
\forall \text{OperationSig} \cdot \exists \text{State'} \cdot \text{Operation} \downarrow \text{outputs} \\

Verify component API robustness.

4. Model properties

Specify and prove conjectures about the properties of the kernel model.
1. Parsing & typechecking

Typeset specification as Z LaTeX document and parse using Z/Eves.

A number of syntax and type errors corrected in the original hand-written specification.
2. Domain & axiomatic checking

Domain checks verify that functions are applied within their domain.
Generated automatically by Z/Eves.

Verify global consistency in axiom declarations.

\textbf{theorem} \ \texttt{tPIDConsistency}

\[ \exists \ minpid, \ maxpid : \mathbb{N} \bullet minpid \leq maxpid \]
Domain checks verify that functions are applied within their domain. Generated automatically by Z/Eves.

Verify global consistency in axiom declarations.

**theorem** tPIDConsistency

\[ \exists \ minpid, \ maxpid : \mathbb{N} \bullet \ minpid \leq \ maxpid \]

\[
\begin{array}{c}
\minpid, \ maxpid : \mathbb{N} \\
\hline
\minpid \leq \ maxpid
\end{array}
\]

axiomatic declaration
3. Feasibility & preconditions

Check the state invariant to verify model feasibility.

Calculate the operation precondition and verify a valid operation after state.

\textbf{theorem} \ tOperationPre \\
\forall \ OperationSig \ \bullet \ \exists \ State' \ \bullet \ Operation \ \setminus \ \text{outputs}

Verify component API robustness.
Check the state invariant to verify model feasibility.

Calculate the operation precondition and verify a valid operation after state.

\textbf{theorem} \ t\text{OperationPre} \\
\forall \text{OperationSig} \bullet \exists \text{State'} \bullet \text{Operation} \setminus \text{outputs}

Verify component API robustness.
4. Model properties

Specify and prove conjectures about the properties of the kernel model.
Mechanisation stages

Iterative mechanisation of the formal $\mathbf{Z}$ specification.

1. Parsing & typechecking

Typeset specification as $\mathbf{Z}$ LaTeX document and parse using $\mathbf{Z}$/Eves.

A number of syntax and type errors corrected in the original hand-written specification.

2. Domain & axiomatic checking

Domain checks verify that functions are applied within their domain. Generated automatically by $\mathbf{Z}$/Eves.

Verify global consistency in axiom declarations.

$\text{theorem } t\text{PIDConsistency}$

$\exists \text{minpid}, \text{maxpid} : \mathbb{N} \bullet \text{minpid} \leq \text{maxpid}$

$\minpid, \maxpid : \mathbb{N}$

$\minpid \leq \maxpid$  \hspace{1cm} \text{axiomatic declaration}

3. Feasibility & preconditions

Check the state invariant to verify model feasibility.

Calculate the operation precondition and verify a valid operation after state.

$\text{theorem } t\text{OperationPre}$

$\forall \text{OperationSig} \bullet \exists \text{State'} \bullet \text{Operation} \setminus \text{outputs}$

Verify component API robustness.

4. Model properties

Specify and prove conjectures about the properties of the kernel model.
Formal specification

Modularisation via Z schema inclusion, each component is robust & reusable.

Process table
Stores all process information in maps (partial functions):
- process identifiers & type;
- process state;
- external identifier translation;
- process memory, messaging, etc.

Process queue
Stores unique process identifiers in a sequence.
References process table for process information.

General lemmas
New general lemmas about injective sequence manipulation.

Device queue
Process queue with renamed variables.

Scheduler
Round-robin scheduler:
schedule, ready, suspend, terminate operations.

Device queue priority execution — simulates synchronous device I/O.

Invariants about executing processes and queue properties.

Model properties
Specified and proved properties about the mechanised components, e.g.:
- \( V.Sched \rightarrow \text{true process state} \rightarrow 0 \)
- \( \forall D \exists ! C. \text{used} \)

Separation proofs require the mechanisation of the complete kernel model.

Composite operation
Operations can be composed.
No operation atomicity if the
New transaction Z pattern
for composite operations was
Not included in this publication.
Formal specification

Modularisation via Z schema inclusion, each component is robust & reusable.

**Process table**

Stores all process information in maps *(partial functions)*:

- process identifiers & type;
- process state;
- external identifier translation;
- process memory, messaging, etc.

**Process**

Stores unique process identifiers.

References properties of processes.

**General lemma**

New general lemma related to injective sequences.
Formal specification

Modularisation via 2 schema inclusion, each component is robust & reusable.

Process table
Stores all process information in maps (partial functions):
- process identifiers & type;
- process state;
- external identifier translation;
- process memory, messaging, etc.

Process queue
Stores unique process identifiers in a sequence.
References process table for process information.

General lemmas
New general lemmas about injective sequence manipulation.

Scheduler
Round-robin scheduler:
schedule, ready, suspend, terminate operations.
Device queue priority execution — simulates synchronous device I/O.
Invariants about executing processes and queue properties.

Device queue
Process queue with renamed variables.

Model properties
Specified and proved properties about the mechanised components, e.g.:
\[ V \text{ \text{Scheduler}} \land \text{process queue done} = \emptyset \]
\[ V \text{ \text{Scheduler}} \land \text{process queue done} = C \text{ \text{used}} \]
Separation proofs require the mechanisation of the complete kernel model.

Composite operation
Operations can be composed.
No operation atomicity if the
New Transaction 2 pattern
for composite operations not
Not included in this publication.
Process table

Stores all process information in maps (partial functions):

• process identifiers & type;
• process state;
• external identifier translation;
• process memory, messaging, etc.
Process queue

Stores unique process identifiers in a sequence.

References process table for process information.

General lemmas

New general lemmas about injective sequence manipulation.
Device queue

Process queue with renamed variables.
Scheduler

Round-robin scheduler: schedule, ready, suspend, terminate operations.

Device queue priority execution — simulates synchronous device I/O.

Invariants about executing processes and queue properties.
Formal specification

Modularisation via Z schema inclusion, each component is robust & reusable.

Process queue
Stores unique process identifiers in a sequence.
References process table for process information.

General lemmas
New general lemmas about injective sequence manipulation.

Device queue
Process queue with renamed variables.

Scheduler
Round-robin scheduler:
schedule, ready, suspend, terminate operations.

Device queue priority execution — simulates synchronous device I/O.

Invariants about executing processes and queue properties.

Process table
Stores all process information in maps (partial functions):
- process identifiers & type;
- process state;
- external identifier translation;
- process memory, messaging, etc.

Model properties
Specified and proved properties about the mechanised components, e.g.:
V Schedule
 slipped process done = 0
V Release
 slipped C used

Separation proofs require the mechanisation of the complete kernel model.

Composite operations
Operations can be composed
No operation atomicity if the
New Transaction Z pattern
for composite operations was
Not included in this publication.
Model properties

Specified and proved properties about the mechanised components, e.g.:

∀ Sched • ran proc ∩ ran devs = ∅

∀ Sched • queued ⊆ used

Separation proofs require the mechanisation of the complete kernel model.
Composite operations

Operations can be composed for sequential execution.

No operation atomicity if the second operation can fail:
Operations can be composed for sequential execution.

No operation atomicity if the second operation can fail:

New **Transaction Z** pattern to ensure atomicity for composite operations with failures.

*Not included in this publication.*
Mechanisation & verification
Mechanised & corrected model

Significant upgrades to the original separation kernel component specifications (Craig, 2007):

- eliminated syntax errors;
- verified model feasibility & API robustness;
- added missing invariants;
- replaced verbal requirements with invariants;
- improved specification design.
Abstract data structures in Z

Reusable abstract data structures for kernel design. *Contributed to the Verified Software Repository (VSR).*

- **Process table**
  process types, external identifiers

- **Process queue**
  robust operations, to be used in *semaphores*

- **Scheduler**
  separate queues, queue priority, robust API operations
All theorems and verification conditions proved with $\mathbb{Z}/\text{Eves}$.

New general (model-independent) lemmas extracted for reuse (*contributed to the VSR*), about:

- injective sequence updates;
- partial functions;
- injective functions, etc.
Composite operations & atomicity

Demonstrated specification and proofs of complex composite operations.

Analysed how to specify composite atomic operations in Z notation; developed the Transaction Z pattern.
Benchmarks

The verification report, experiences and benchmarks contribute to the VSR.

<table>
<thead>
<tr>
<th>Mechanisation:</th>
<th>5 out of 12 sections in (Craig, 2007)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paragraphs:</td>
<td>263 (schemas, types, theorems, etc.)</td>
</tr>
<tr>
<td>Proofs:</td>
<td>254 (50% model feasibility, 35% automation lemmas)</td>
</tr>
<tr>
<td>Proof commands:</td>
<td>1300 (40% push-button steps)</td>
</tr>
<tr>
<td>Effort:</td>
<td>900 man-hours (incl. training &amp; write-up)</td>
</tr>
</tbody>
</table>
Proofs & general lemmas

All theorems and verification conditions proved with Z/Eves.
New general (model independent) lemmas extracted for reuse (contributed to the VS9), about:
• injective sequence updates;
• partial functions;
• injective functions, etc.

Composite operations & atomicity

Demonstrated specification and proofs of complex composite operations.
Analysed how to specify composite atomic operations in Z notation; developed the Transaction Z pattern.

Benchmarks

The verification report, experiences and benchmarks contribute to the VS9.

| Mechanisation: | 5 out of 12 sections in (Craig, 2007) |
| Paragraphs: | 261 (schemas, types, theorems, etc.) |
| Proofs: | 254 (50% model feasibility, 35% automation lemmas) |
| Proof commands: | 1300 (40% push-button steps) |
| Effort: | 900 man-hours (incl. training & write-up) |

Future work

• Complete abstract model
  remaining separation kernel components, prove process separation
• Data refinement
  perform refinement of the abstract model to the existing C implementation code
Future work

- Complete abstract model remaining separation kernel components, prove process separation

- Data refinement perform refinement of the abstract model to the existing C implementation code
Thank you for your attention.

Do you have any questions?