Unifying Theories of Programming

Andrew Butterfield

Trinity College Dublin

Hamilton Institute, Maynooth, 20th December 2023
What are Formal Methods?

Using mathematics to reason about computing artefacts

Proving programs correct rather than just testing

Reasoning about relationships between different development levels

Requirements | Specifications | High Level Design | Low Level Design | Code
Why doesn’t everyone use them?

Too hard!

Proof reviewing way harder than code review!

Unrealistic!

Testing is good enough!

“Program testing can be used to show the presence of bugs, but never to show their absence!”
(Edsger W. Dijkstra)
Even a key player got pessimistic...

“Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalisation to solve the problems of reliability that arise when programs get large and more safety-critical. Programs have now got very large and very critical – well beyond the scale which can be comfortably tackled by formal methods. There have been many problems and failures, but these have nearly always been attributable to inadequate analysis of requirements or inadequate management control. It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve.” (Tony Hoare 1995)
Does anyone in industry use Formal Methods?

https://fbinfer.com/

https://aws.amazon.com/

https://www.msdn.microsoft.com/

https://www.adacore.com/

https://www.fmeurope.org/

https://fme-industry.github.io/
What to we do with formal methods?

Specification

Abstract Input  -->  Abstract Output

Concrete Input  -->  Concrete Output

"refinement" or "reification"

Implementation
A “Formal Notation” is a language with a mathematical meaning

We need Formal Notations for:

• Abstract and Concrete Inputs and Outputs
• Descriptions of Specifications and Implementations
• Descriptions of the Refinement relationship
Formal Semantics
Giving mathematical meaning to modelling, specification, and program languages

Three key approaches:

Algebraic

Denotational

Operational
Operational Semantics

- Defined as some kind of State Machine ($S$)
  The state typically includes variable values and current line of code being executed

- Programming language elements define State Transitions.
  These are usually defined using Inference Rules similar to those used in Logic.

- If $s_1, s_2, s_3 : S$ are states, then the semantics for $P;Q$ might be written:
  \[
  \begin{align*}
  (s_1, P) &\rightarrow s_2 \\
  (s_2, Q) &\rightarrow s_3 \\
  \hline
  (s_1, P;Q) &\rightarrow s_3 
  \end{align*}
  \]
  “If $P$ changes $s_1$ to $s_2$ and $Q$ changes $s_2$ to $s_1$, then $P;Q$ changes $s_1$ to $s_3$.”
Denotational Semantics

- Defined as a function $\mathcal{M}$ from program text $Prog$ to a mathematical object $Domain$:

  $$\mathcal{M} : \text{Prog} \rightarrow \text{Domain}$$

- A $Domain$ is a (often complex) mathematical structure

- A key principle is compositionality:
  The meaning of a composite (e.g. $P ; Q$) is defined in terms of the meanings of its sub-components:

  $$\mathcal{M}(P ; Q) \equiv \text{seq}(\mathcal{M}(P), \mathcal{M}(Q))$$

Here $\text{seq}$ is the semantics of sequential composition.

- This ensures that the semantics will scale to large programs.
Algebraic Semantics

- Defined as rules defining equalities between program texts (Often referred to as Laws of Programming)

- E.g.:

  \[ x := k_1; y := k_2 \equiv y := k_2; x := k_1 \]
  \[ x := e_1; x := e_2 \equiv x := e_2 \quad x \notin e_2 \]
  \[ x := e_1; x := e_2 \equiv x := e_2[e_1/x] \]
Advantages

Denotational
Best way to ensure a sound correct theory.
Used to justify Algebraic semantics.

Operational
Easiest to understand — close to real machine level.
Easiest for implementing and verifying compilers.
Easiest for implementing Model Checkers.
The most commonly used form of formal semantics.

Algebraic
Allows equational reasoning to be used to prove program correctness in a compositional way.
Disadvantages

Denotational
Can be very complex, to the extent that it has not been done for some languages.

Operational
Hard to use to prove correctness - not compositional!
Hard to “debug” the rules to avoid inconsistencies.

Algebraic
Hard to know when we have enough laws.
Hard to know we have no internal contradictions.
The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

“. . . today . . . 1,700 special programming languages used to ‘communicate’ in over 700 application areas.”—Computer Software Issues, an American Mathematical Association Prospectus, July 1965.

specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

\[ x(x+a) \quad \text{where} \quad x = b + 2c \]
\[ f(b+2c) \quad \text{where} \quad f(x) = x(x+a) \]

Iswim is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called “A Correspondence between \( x \) and Church’s \( \lambda \)-notation.”
700 Formal Methods?

FM varieties:

- **Nth-Order Logic**, for $N \in \{1, 2, \ldots, \infty\}$
- **Functions defined over tailored partial orders**
  - Monotonicity, continuity, fixpoint theory
- **High-level approaches**: 
  - Operational 
  - Algebraic/Axiomatic 
  - Denotational
- **Certainly scope for many...**
Unifying Theories of Programming (UTP)

- Who: Tony Hoare and He Jifeng
- Idea: build a general semantics framework that can capture all of the above.
- Formalism: Programs are Predicates
- Available online (free): http://unifyingtheories.org/
Predicates

\[ P \in \text{Pred} \quad ::= \quad \text{true} \quad \text{always true} \]
\[ \quad \quad \text{false} \quad \text{always false} \]
\[ \quad \quad A \quad \text{Atomic predicates} \]
\[ \quad \quad P \quad \text{Predicate variables} \]
\[ \quad \quad \neg P \quad \text{Logical negation} \]
\[ \quad \quad P_1 \land P_2 \quad \text{Conjunction (logic-and)} \]
\[ \quad \quad P_1 \lor P_2 \quad \text{Disjunction (logic-or)} \]
\[ \quad \quad P_1 \implies P_2 \quad \text{Implication} \]
\[ \quad \quad P_1 \equiv P_2 \quad \text{Equivalence} \]
\[ \quad \quad \forall x \bullet P \quad \text{Universal quantification (for-all)} \]
\[ \quad \quad \exists x \bullet P \quad \text{Existential quantification (there-exists)} \]
\[ \quad \quad [P] \quad \text{Universal Closure} \]
\[ \quad \quad P[E/x] \quad \text{Free-variable Substitution} \]
Programs as Predicates (syntax)

The **While** language – a simple language to explain ideas:

\[
P, Q \in Pred ::= \ldots
\]

- \( \text{II} \) 
  - \( \text{skip} \)
- \( v := e \) 
  - \( v := e \)
- \( P \triangleleft c \triangleright Q \) 
  - if \( c \) then \( P \) else \( Q \)
- \( P; Q \) 
  - \( P ; Q \)
- \( c \ominus P \) 
  - while \( c \) do \( P \)
Programs as Predicates (semantics)

\[
\begin{align*}
\text{II} & \triangleq S' = S \\
\text{v := e} & \triangleq v' = e \land S' \setminus v' = S \setminus v \\
P \triangleleft c \triangleright Q & \triangleq c \land P \lor \neg c \land Q \\
P ; Q & \triangleq \exists S_m \bullet P[S_m/S'] \land Q[S_m/S] \\
c \otimes P & \triangleq (P ; c \otimes P) \triangleleft c \triangleright \text{II}
\end{align*}
\]

Here \( S \) stands for the program before-state (values of all variables), and \( S' \) stands for the program after-state.

Note: the above semantics is Denotational.
Programs as Predicates (laws)

\[ II = x := x, \quad x \in A \quad \text{SKIP-ALT} \]
\[ II; P = P \quad \text{;-L-UNIT} \]
\[ P; II = P \quad \text{;-R-UNIT} \]
\[ P; (Q; R) = (P; Q); R \quad \text{;-ASSOC} \]
\[ x := e; x := f = x := f[e/x] \quad \text{==SEQ} \]
\[ x := e; y := f = y := f[e/x]; x := e \quad \text{==SWAP} \]
\[ x := k; P = P[k/x] \quad \text{==INIT} \]
\[ P \triangleright \text{true} \triangleright Q = P \quad \text{ corporation } \text{TRUE} \]
\[ P \triangleright \text{false} \triangleright Q = Q \quad \text{ corporation } \text{FALSE} \]
\[ (P \triangleright c \triangleright Q); R = (P; R) \triangleright c \triangleright (Q; R) \quad \text{ corporation } \text{-;DISTR} \]
\[ c \otimes P = (P; c \otimes P) \triangleright c \triangleright II \quad \text{ corporation } \text{-UNROLL} \]
What’s this got to do with Unification?

• We’ve just got started!
• Missing is stuff like:
  • declaring global and local variables
  • functions and procedures
  • ... but this could be for this language
• Different languages:
  • Add in Object Orientation
  • Functional Languages
  • Logic Programming
  • ...

The really interesting challenge for unification is concurrency
Concurrenty is HARD! Why?

Consider this simple program (\texttt{incx}):

\texttt{incx = temp := x ; temp := temp+1; x := temp}

We expect its behaviour to satisfy the property: \( x' = x+1 \)

Now, let’s add a parallelism construct to our language: \( P \parallel Q \)

What property should \( \texttt{incx} \parallel \texttt{incx} \) satisfy?

\( x' = x+2 \) ?
It all depends on our model of concurrency

One approach is that each instance of \texttt{incx} has its own state (here \texttt{x} and \texttt{temp}).

\[
x_1' = x_1 + 1 \quad x_2' = x_2 + 1
\]

This then requires some way of communicating between parallel “threads” if we want to have interactions.

Another approach: \texttt{x} is global and shared, \texttt{temp} is local to each instance. Now, how do we schedule the running of the instances?

\[
\begin{align*}
\text{incx}_1 &: \quad \texttt{temp}_1 := \texttt{x}; \quad \texttt{temp}_1 = \texttt{temp}_1 + 1; \quad \texttt{x} := \texttt{temp}_1 \\
\text{incx}_2 &: \quad \texttt{temp}_2 := \texttt{x}; \quad \texttt{temp}_2 := \texttt{temp}_2 + 1; \quad \texttt{x} := \texttt{temp}_2
\end{align*}
\]

Here we observe \( x' = x + 1 \)!!

(this is \texttt{pthreads} behaviour!)
Specifying concurrency

Sharing global variables is dangerous – requires careful techniques to get right

Many specification languages keep state local, and communicate via visible events

Many of these use operational semantics

Denotational semantics will typically map such languages to sets of event-sequences
Communicating Sequential Processes (CSP)

Devised by Tony Hoare and colleagues

Idea: processes perform events and synchronise on some:

- \texttt{skip} : Terminate
- \texttt{e} \rightarrow \texttt{P} : Perform event \texttt{e} and behave like \texttt{P}
- \texttt{P;Q} : Run \texttt{P} first, then \texttt{Q}
- \texttt{P|e|Q} : Run in parallel synchronising on event \texttt{e}

What do these look like as predicates?

\begin{align*}
\texttt{skip} &= \texttt{ok} \Rightarrow \texttt{ok}' \land \texttt{tr}' = \texttt{tr} \\
\texttt{e} \rightarrow \texttt{skip} &= \texttt{ok} \Rightarrow \texttt{ok}' \land \texttt{tr}' = \texttt{tr} + [\texttt{e}] \\
\texttt{e} \rightarrow \texttt{P} &= \texttt{e} \rightarrow \texttt{skip}; \texttt{P}
\end{align*}

Here state $S = \{\texttt{ok}, \texttt{tr}\}$ so \texttt{skip} and \texttt{P;Q} are defined as for the sequential code

Started: ok
Finished: ok'
Initial event-seq: tr
Final event-seq: tr'
UTP Alphabets

UTP is based on the idea of alphabets:
– set of observation variables, both before- and after-values.

For the While language the alphabet is all the variables in scope
\{x, temp, x’, temp’, …\}

For CSP the alphabet is \{ok, ok’, tr, tr’, …\}
UTP Healthiness Conditions

The idea behind “healthiness conditions” is to rule out “unhealthy” predicates.

Unhealthy predicates make statements that are either impossible, or undesirable.

Impossible predicate in CSP: \( tr = tr' + [e] \) -- can’t erase history
Relevant healthiness condition: \( H1 : tr \leq tr' \) (prefix-of)

Often we can enforce/specify healthiness using a Predicate Transformer:

\[
H1(P) = P \land tr \leq tr'
\]

(now we are no longer doing 1st-order logic!)
Unification Example: Circus


Circus is a unification of two formal specification languages:
CSP – already discussed
Z – a specification language for While-like programs

The alphabet: \{ok, tr, ref, wait, state, ok', tr', ref', wait', state'\}
where state records variable values.

It fused the observations from the two theories, and identified a common notion of healthiness called a Design:

\[ ok \land P \Rightarrow ok' \land Q \]

If the program starts and P is true initially, then the program terminates as Q is true on termination.
Not the whole unification story

UTP looks at unifying the three main semantic techniques

Algebraic

\[ \begin{align*}
(\exists c \cdot D(c) \land L(c, a)) & \Rightarrow S(a) \quad \text{iff} \quad [D(c) \Rightarrow (\forall a \cdot L(c, a) \Rightarrow S(a))] \\
\end{align*} \]

Refinement!

A Galois Connection!
My contributions to UTP (1) "Slotted-Circus"

A timed variant of Circus was developed with events organised into time-slots.


I developed a version where the time-slot itself had a richer structure, to give a semantics for Handel-C, a C-like language that compiled to FPGA hardware.

My contributions to UTP (2) “Slotted-Circus++”

Adding priority to slotted-Circus


\[
\begin{align*}
L \Box H & \equiv H \Box L \\
H \Box L & \equiv H \land L \land \text{Stop} \lor \text{Choice}(H, L) \lor \text{WeakChoice}(L, H)
\end{align*}
\]

\[
\text{WeakChoice}(L, H) \equiv \text{CSP2}(L \land \left( H \land \text{NOEVTS} \land \text{wait}' \right) ;
\begin{align*}
\text{PRI} \left( \exists E \cdot 
\left( \text{FSTEVTS}(E) \land E \neq \emptyset \land E \subseteq \text{sref(tail(slots))} \right) \lor \text{slots} \cong \text{slots'} \land (\neg \text{wait}' \lor \neg \text{ok}') \right)
\right)
\]


\[
\begin{align*}
\text{abort} & \equiv \text{true} \\
\text{skip} & \equiv \delta' = \delta \\
\overline{x} := e & \equiv \delta' = \delta \{ e/x \} \\
A; B & \equiv \exists \delta_m \cdot A(\delta, \delta_m) \land B(\delta_m, \delta') \\
A \triangleleft c \triangleright B & \equiv \exists \delta_A, \delta_B \cdot A(\delta(c), \delta_A) \land B(\delta(-c), \delta_B) \land \delta' = \delta_A + \delta_B \\
A \triangledown_p B & \equiv \exists \delta_A, \delta_B \cdot A(p \cdot \delta, \delta_A) \land B((1 - p) \cdot \delta, \delta_B) \land \delta' = \delta_A + \delta_B
\end{align*}
\]
My contributions to UTP (3)

Unifying Theory of Concurrent Programs (UTCP)

\[ P ;; Q \equiv W(P[g_1, \ell_g/g, out] \lor Q[g_2, \ell_g/g, in]) \]
\[ P \parallel Q \equiv W( A(in \mid ii \mid \ell_{g_1}, \ell_{g_2}) \lor P[g_1, \ell_{g_1}, \ell_{g_1}/g, in, out] \lor Q[g_2, \ell_{g_2}, \ell_{g_2}/g, in, out] \lor A(\ell_{g_1}, \ell_{g_2} : | ii | out) ) \]
\[ P + Q \equiv W( P[g_1/g] \lor Q[g_2/g] ) \]
\[ P^* \equiv W( A(in \mid ii \mid \ell_g) \lor A(\ell_g \mid ii \mid \ell_g) \lor A(\ell_g \mid ii \mid out) \lor P[g::, \ell::, \ell_g/g, in, out] ) \]

Why is Logic so hard?
Inference Rules

- Inference rules take zero or more assumptions, and describe how they support a conclusion.
  (i.e. from assumptions $A_1, A_2, \ldots, A_n$ we can conclude $C$)

- The standard presentation of these rules places assumptions above a horizontal line and the conclusion below:

  $$\text{inference-name: } \frac{A_1 \quad A_2 \quad \ldots \quad A_n}{C} \quad [\text{side conditions}]$$

  Side conditions are properties of predicates

- A rule with no assumptions is an axiom—something taken as (self evidently) true.

  $$\text{reflexivity-of-equals: } \frac{}{x = x}$$
Axioms for Propositional Logic

\(\neg\text{-I: } P \implies Q \quad P \implies \neg Q\)

\(\neg\text{-E: } \neg P \quad P \implies R\)

\(\neg\neg\text{-E: } \neg \neg P \quad P\)

\(\land\text{-I: } P \quad Q \quad P \land Q\)

\(\land\text{-E: } P \quad Q \quad P \land Q\)

\(\land_2\text{-E: } P \land Q \quad Q\)

\(\lor_1\text{-I: } P \quad P \lor Q\)

\(\lor_2\text{-I: } Q \quad P \lor Q\)

\(\lor\text{-E: } P \lor Q \quad P \implies R \quad Q \implies R \quad R\)

\(\implies \text{-E: } P \quad P \implies Q\)

\(\implies \text{-I: } (P \vdash Q) \quad P \implies Q\)
Proof Towers/Trees for Propositions

\[ \begin{array}{c}
A \vdash A \\
\\vdash \neg A, A \\
A \lor \neg A, A \\
\\vdash A \lor \neg A, A \lor \neg A \\
\\vdash A \lor \neg A
\end{array} \]

\[ \begin{array}{c}
B \vdash B \\
C \vdash C \\
B \lor C \vdash B, C \\
\quad \vdash B \lor C, B \\
\quad \vdash B \lor C, \neg C \vdash B \\
\quad \vdash \neg A \vdash \neg A \\
\quad \vdash (B \lor C), \neg C, (B \rightarrow \neg A) \vdash \neg A \\
\quad \vdash (B \lor C), \neg C, ((B \rightarrow \neg A) \land \neg C) \vdash \neg A \\
\quad \vdash (B \lor C), ((B \rightarrow \neg A) \land \neg C), \neg C \vdash \neg A \\
\quad \vdash (B \lor C), ((B \rightarrow \neg A) \land \neg C), ((B \rightarrow \neg A) \land \neg C), \neg C \vdash \neg A \\
\quad \vdash (B \lor C), ((B \rightarrow \neg A) \land \neg C), ((B \rightarrow \neg A) \land \neg C), (B \lor C) \vdash \neg A \\
\quad \vdash (B \lor C), ((B \rightarrow \neg A) \land \neg C), (B \lor C) \vdash \neg A \\
\quad \vdash (B \rightarrow \neg A) \land \neg C, (A \rightarrow (B \lor C)) \vdash \neg A, \neg A \\
\quad \vdash (B \rightarrow \neg A) \land \neg C, (A \rightarrow (B \lor C)) \vdash \neg A \\
\quad \vdash (A \rightarrow (B \lor C)), ((B \rightarrow \neg A) \land \neg C) \vdash \neg A \\
\quad \vdash (A \rightarrow (B \lor C)) \vdash (((B \rightarrow \neg A) \land \neg C) \rightarrow \neg A) \\
\quad \vdash ((A \rightarrow (B \lor C)) \rightarrow (((B \rightarrow \neg A) \land \neg C) \rightarrow \neg A))
\end{array} \]
### A more linear approach - “Natural” Deduction

#### Example of a proof

<table>
<thead>
<tr>
<th>Number</th>
<th>Formula</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
<td>premise</td>
</tr>
<tr>
<td>2</td>
<td>$A \lor A$</td>
<td>From (1) by disjunction introduction</td>
</tr>
<tr>
<td>3</td>
<td>$(A \lor A) \land A$</td>
<td>From (1) and (2) by conjunction introduction</td>
</tr>
<tr>
<td>4</td>
<td>$A$</td>
<td>From (3) by conjunction elimination</td>
</tr>
<tr>
<td>5</td>
<td>$A \vdash A$</td>
<td>Summary of (1) through (4)</td>
</tr>
<tr>
<td>6</td>
<td>$\vdash A \rightarrow A$</td>
<td>From (5) by conditional proof</td>
</tr>
</tbody>
</table>
What is the issue?

Difference between logic as:
1. An object of study in its own right
2. Something for which tool-support is required
3. A tool to use to reason about stuff

The approach on previous slides is suitable for 1 & 2 above

It’s not great for 3.
Equational Inference Rules

Equanimity: \[
\frac{P}{Q} 
\]

Transitivity: \[
\frac{P = Q}{Q = R} \quad \frac{Q = R}{P = R}
\]

Substitution: \[
\frac{P}{P[b \leftrightarrow Q]} \quad \text{[no capture]}
\]

Liebniz: \[
\frac{F = G}{P[v \leftarrow F] = P[v \leftarrow G]} \quad \text{[v a variable]}
\]
Linearising proof trees

\[ A \]
\[ = \quad \text{“why } A = B \text{”} \]
\[ B \]
\[ = \quad \text{“why } B = C \text{”} \]
\[ C \]
\[ = \quad \text{“why } C = D \text{”} \]
\[ D \]
\[ = \quad \text{“why } D = E \text{”} \]
\[ E \]

The proof that \( A = B \) can also be linearised, and can simply be inserted into the proof above where \( \text{“why } A = B \text{”} \) appears.
true

\[ P \equiv P \quad \equiv \text{-REFL} \]

\[ ((P \equiv Q) \equiv R) \equiv (P \equiv (Q \equiv R)) \quad \equiv \text{-ASSOC} \]

\[ P \equiv Q \equiv Q \equiv P \quad \equiv \text{-SYMM} \]

\[
\frac{(\text{true} \equiv P) \equiv P}{= \quad \text{" \equiv\text{-ASSOC (L2R) "}}}
\]

\[ \text{true} \equiv (P \equiv P) \quad \equiv \text{-REFL (all)} \]

\[ \frac{\text{true} \equiv \text{true}}{= \quad \text{" \equiv\text{-REFL (all) "}}}
\]

\[ \text{true} \]
\( \neg(P \equiv Q) \equiv \neg P \equiv Q \quad \neg-\equiv\text{-DISTR} \)

false \equiv \neg \text{true} \quad \text{FALSE-DEF}

\[
\begin{align*}
(\neg P \equiv Q) & \equiv (P \equiv \neg Q) \\
= & \quad " \quad \neg-\equiv\text{-DISTR (R2L)} \quad " \\
(\neg P \equiv Q) & \equiv (P \equiv \neg Q) \\
= & \quad " \quad \equiv\text{-SYMM (L2R)} \quad " \\
(\neg P \equiv Q) & \equiv (\neg Q \equiv P) \\
= & \quad " \quad \neg-\equiv\text{-DISTR (R2L)} \quad " \\
(\neg P \equiv Q) & \equiv (Q \equiv P) \\
= & \quad " \quad \equiv\text{-SYMM (L2R)} \quad " \\
(\neg P \equiv Q) & \equiv (P \equiv Q) \\
= & \quad " \quad \equiv\text{-REFL (all)} \quad "
\end{align*}
\]

true
Why Equational Reasoning is so good

It’s very similar in style to so-called regular mathematics proofs.

Each step in the proof only depends on the law being applied, and the two expressions involved in that step – simple, and local.

Developed in 70s and 80s by Formal Method researchers.
So good I wrote a theorem-prover!
ReasonEq is a theorem prover designed to support equational reasoning, from the ground up.

- Text-based user interface (GUIs are possible)
- Written using Haskell.
  Can run on Unix, macOS, and Windows (WSL2 is best)
- Open-source
  (https://github.com/andrewbutterfield/reasonEq).
- Under development, but already useable.
History

Saoithín

Early development version, with wxWindows GUI, ran on Windows only.


$U \cdot (TP)^2$

Improved (renamed) version, with GUI, ran on Windows, Ubuntu.

“The Logic of $U \cdot (TP)^2$”, UTP 2012.

$U \cdot (TP)^2$: Higher-Order Equational Reasoning by Pointing”, UTP 2014.

ReasonEq

Completely re-engineered version, abstract UI motivated by ongoing difficulties with wxHaskell.

UTPCalc

Predicate “calculator” for rapid prototyping of UTP theories. Requires knowledge of Haskell.


Future?

ReasonEq and UTPCalc will merge
Why Formal Methods is making a comeback (1)

Tool Support – “Industrial-Strength”

https://isabelle.in.tum.de/

https://coq.inria.fr/

https://www.adacore.com/

https://pvs.csl.sri.com/
Why Formal Methods is making a comeback (2)

Multi-core!
A change in direction: ESA

2010: Lero enters talks with European Space Agency on research collaboration

2011: we start a project looking at
• Autonomous systems (Mike Hinchey, Emil Vassev)
• Software product lines (Goetz Botterveck)
• Formal modelling and verification of hypervisors (myself, David Sanan)

I do another project as a consultant
– involving the big players: Thales, Airbus and CNES
RTEMS pre-Qualification

- Open-source, freely available, BSD-2 license

RTEMS was qualified for spaceflight in 2006-2008, for a single-core processor.
ESA clients want to use multi-core

A recent ESA-sponsored activity (2018-2021):
• Qualified RTEMS on multicore for spaceflight
• All resources produced fed back to RTEMS open-source
• (https://git.rtems.org/rtems-central/.)
• All such resources fit with RTEMS community guidelines

My role (with Frédéric Tuong) was to investigate the role Formal Methods could play in spaceflight software qualification
Chosen Formal Verification approach: Promela/SPIN

Programmatic modelling language

Builds a non-deterministic finite-state machine (a.k.a. model)

Properties are checked by exhaustive search

A failed check produces the path through the model that leads to failure:
A “counter-example”
Test Generation ???

Lie to the model-checker !
Modelling an API Call

```c
inline barrier_release(self, bid, nreleased, rc) {
    atomic{
        if
        :: nreleased == 0 -> rc = RC_InvAddr;
        :: bid >= MAX_BARRIERS || bid <= 0 || !barriers[bid].isInitialised ->
            rc = RC_InvId
        :: else ->
            nreleased = barriers[bid].waiterCount;
            barrierRelease(self, bid);
            printf("@@@ %d LOG Barrier %d manually released\n", _pid, bid)
            rc = RC_OK
        fi
    }
}
```

What the @@? (later!)

Model bad calls!

Model correct behaviour too!
Counter-Examples

spin: barrier-mgr-model.pml:1154, Error: assertion violated
spin: text of failed assertion: assert(0)
Barrier Manager Model finished!
spin: trail ends after 354 steps
#processes: 1
  tasks[1].nodeid = 1
  tasks[1].state = Zombie
  ...
  barriers[1].isAutomatic = 0
  ...
  semaphore[1] = 0
  ...
  task_in[1].doCreate = 1
  ...
  task_in[2].doAcquire = 1
  ...
  task_in[3].doAcquire = 1
  ...
  task2Sema = 1
  task1Core = 1
  task2Core = 0
  ...
  scenario = ManAcqRel
  stopclock = 1
354:   proc  0 (:init::1) barrier-mgr-model.pml:1158 (state 142) <valid end state>
  6 processes created

SPIN counterexamples look like this!

Dump of final variable values,
And references to code line-numbers

Hard to parse!

Even harder to process!!
Observations

Promela has a `printf()` function. It is ignored for verification, but not for counter-example display.

We have defined an “observation language” using it. This abstracts the key features we want to observe.
Refinement

A YAML file defines a mapping from the observation language to C code in which arguments get substituted.

Simple binary semaphores

```c
SIGNAL: |
    ReleaseSema( semaphore[{}] );

WAIT: |
    ObtainSema( semaphore[{}] );
```

C test code for a call to `rtems_barrier_create`:

```c
barrier_create: |
    T_log(T_NORMAL, "Calling BarrierCreate(%d,%d,%d,%d)", {0}, {1}, {2}, {3} );
    rtems_id bid;
    initialise_id(&bid);
    rtems_status_code rc;
    rtems_attribute attribs;
    attribs = mergeattributes({1});
    rc = rtems_barrier_create({0}, attribs, {2}, {3} ? &bid : NULL);
    T_log(T_NORMAL, "Returned 0x%lx from Create", rc );
```
9.1. Formal Verification Overview

Formal Verification is a technique based on writing key design artifacts using notations that have a well-defined mathematical semantics. This means that these descriptions can be rigorously analyzed using logic and other mathematical tools. The term formal model is used to refer to any such description.

Having a formal model of a software engineering artifact (requirements, specification, code) allows it to be analyzed to assess the behavior it describes. This means checks can be done that the model has desired properties, and that it lacks undesired ones. A key feature of having a formal description is that tools can be developed that parse the notation and perform much, if not most, of the analysis. An industrial-strength formalism is one that has very good tool support.

Having two formal models of the same software object at different levels of abstraction (specification and code, say) allows their comparison. In particular, a formal analysis can establish if a lower level artifact like code satisfies the properties described by a higher level, such as a specification. This relationship is commonly referred to as a refinement.

Often it is quite difficult to get a useful formal model of real code. Some formal modelling approaches are capable of generating machine-readable scenarios that describe possible correct behaviors of the system at the relevant level of abstraction. A refinement for these can be defined by using them to generate test code. This is the technique that is used in Test Generation Methodology to verify parts of RTEMS. Formal models are constructed based on requirements documentation, and are used as a basis for test generation.
Model-based Test Generation is a big hit!

Developers don’t need to understand the model

They really understand the concrete test code.
Future Plans

• More Semaphore Manager
  • Gateway to scheduler thread queues
• Refactoring
• Beyond Synchronisation
  • E.g., Tasks? Objects?
• Beyond Classic API
  • POSIX?
• Beyond RTEMS?
  • This technology is not tied to RTEMS, or C!

```python
def refineSPINLine(self, spin_line):
    match spin_line:
        # NAME <name>
        # Keyword Refinement (NAME)
        case [pid, 'LOG', *rest]:
            if self.outputLog:
                ln = ' '.join(rest)
                self.addCode(pid, ['T_log(T_NORMAL,{});".format(ln)'])
            else:
                pass
        case [pid, 'NAME', name]:
            if 'NAME' not in self.ref_dict_keys:
                self.addCode(pid, [self.EOLC+" CANNOT REFINE 'NAME'"])
            else:
                self.addCode(pid, self.ref_dict['NAME'].rsplit(' \n'))
        # INIT
        # Keyword Refinement (INIT)
        case [pid, 'INIT']:
            if 'INIT' not in self.ref_dict_keys:
                self.addCode(pid, [self.EOLC+" CANNOT REFINE 'INIT'"])
            else:
                self.addCode(pid, self.ref_dict['INIT'].rsplit(' \n'))
        # TASK <name>
        # Name Refinement (<name>)
        case [pid, 'TASK', task_name]:
```
Final Thoughts

- We have successfully used Promela/SPIN to do RTEMS Test Generation
  - Some is already part of RTEMS
  - We have more ready add at the moment (it’s a process)
  - We expect to keep going
- We have three languages involved
  - Promela
  - our Observation language
  - RTEMS test framework (C + libraries)
A case study for UTP unification
Thank you! Any (further) Qs?