Trinity College Dublin

The University of Dublin

Unifying Theories of Programming

Andrew Butterfield

Trinity College Dublin

Hamilton Institute, Maynooth, 2^{0th} 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

Why doesn't everyone use them?

Too hard!

Unrealistic!

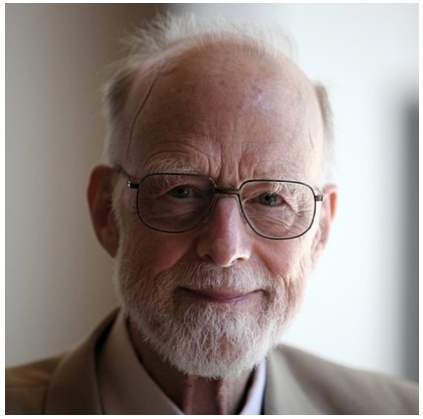
Proof reviewing way harder than code review!

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/

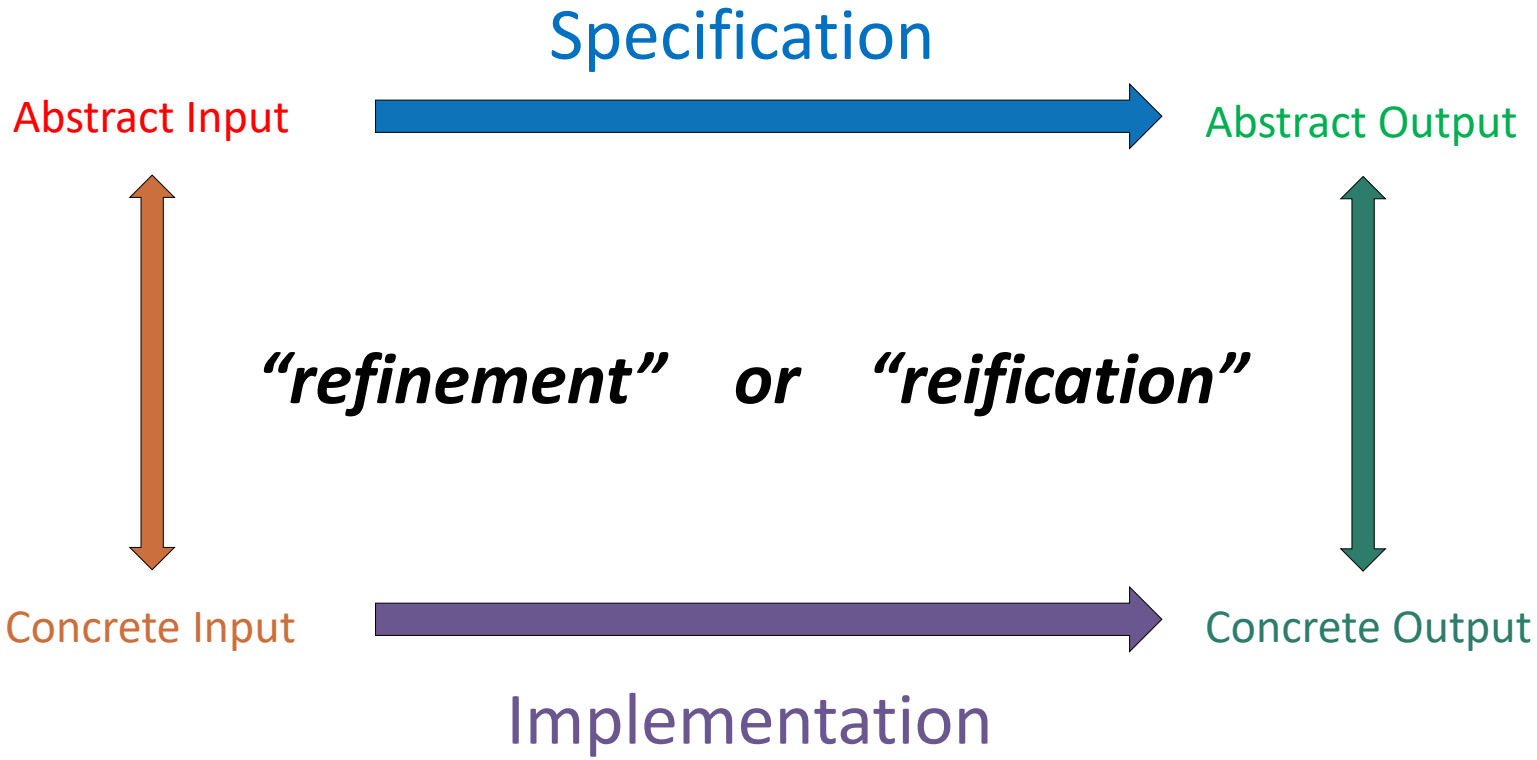
https://www.adacore.com/

https://www.microsoft.com/

https://www.fmeurope.org/

https://fme-industry.github.io/

What to we do with formal methods?



Formal Pieces

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

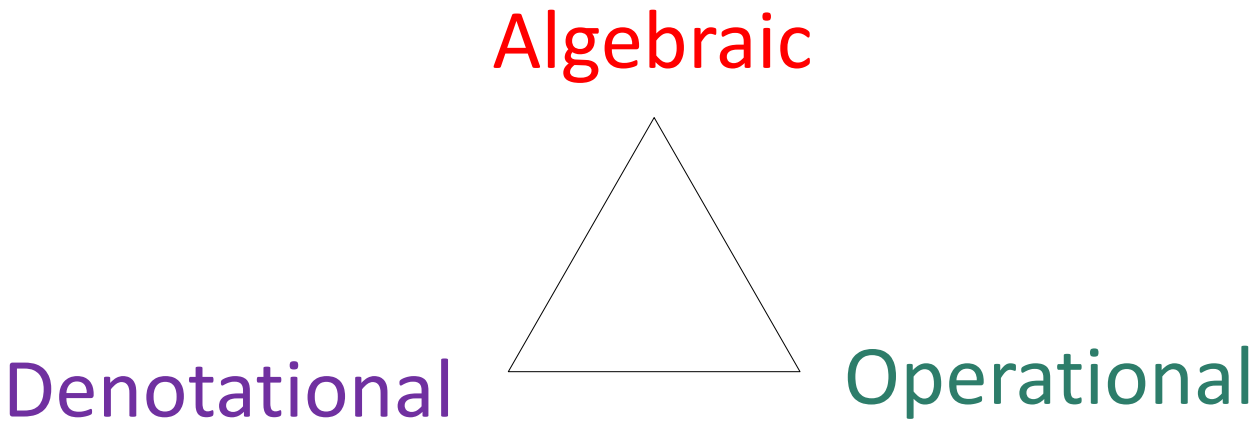
We need Formal Notations for:

- 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:



Operational Semantics

- Defined as some kind of State Machine ($\mathcal{S}$)
  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 $s_1, s_2, s_3 : \mathcal{S}$ are states, then the semantics for `P;Q` might be written:

$$\frac{(s_1, \texttt{P}) \mapsto s_2 \qquad (s_2, \texttt{Q}) \mapsto s_3}{(s_1, \texttt{P;Q}) \mapsto s_3}$$

  "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*:

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

- 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:

$$\mathcal{M}(\texttt{P;Q}) \quad \widehat{=} \quad seq(\mathcal{M}(\texttt{P}), \mathcal{M}(\texttt{Q}))$$

  Here *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.:

$$
\begin{array}{lcll}
x := k_1; y := k_2 & = & y := k_2; x := k_1 & \\
x := e_1; x := e_2 & = & x := e_2 & x \notin e_2 \\
x := e_1; x := e_2 & = & x := e_2[e_1/x] &
\end{array}
$$

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.

P. Landin, 1966

Communications of the ACM, Volume 9, Issue 3, pp 157–166, 1966

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

$x(x+a)$	$f(b+2c)$
where $x = b + 2c$	**where** $f(x) = x(x+a)$

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 λ-notation."[1]

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
- The Book:
  "Unifying Theories of Programming",
  C.A.R. Hoare, He Jifeng,
  Prentice Hall series in Computer Science, 1998.
- Available online (free):
  `http://unifyingtheories.org/`

Predicates

$P \in Pred$	$::=$	true	always true
	$\mid$	false	always false
	$\mid$	A	Atomic predicates
	$\mid$	P	Predicate variables
	$\mid$	$\neg P$	Logical negation
	$\mid$	$P_1 \wedge P_2$	Conjunction (logic-and)
	$\mid$	$P_1 \vee P_2$	Disjunction (logic-or)
	$\mid$	$P_1 \implies P_2$	Implication
	$\mid$	$P_1 \equiv P_2$	Equivalence
	$\mid$	$\forall x \bullet P$	Universal quantification (for-all)
	$\mid$	$\exists x \bullet P$	Existential quantification (there-exists)
	$\mid$	$[P]$	Universal Closure
	$\mid$	$P[E/x]$	Free-variable Substitution

Programs as Predicates (syntax)

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

$$
\begin{array}{lcll}
P, Q \in \mathit{Pred} & ::= & \ldots & \\
& \mid & \Pi & \texttt{skip} \\
& \mid & v := e & \texttt{v := e} \\
& \mid & P \lhd c \rhd Q & \texttt{if c then P else Q} \\
& \mid & P ; Q & \texttt{P ; Q} \\
& \mid & c \circledast P & \texttt{while c do P}
\end{array}
$$

Programs as Predicates (semantics)

$$
\begin{array}{rcll}
\mathrm{II} & \widehat{=} & S' = S & \text{SKIP-DEF} \\
v := e & \widehat{=} & v' = e \wedge S' \backslash v' = S \backslash v & :=\text{-DEF} \\
P \lhd c \rhd Q & \widehat{=} & c \wedge P \vee \neg c \wedge Q & \lhd\ \_\ \rhd\ \text{-DEF} \\
P ; Q & \widehat{=} & \exists S_m \bullet P[S_m/S'] \wedge Q[S_m/S] & ;\text{-DEF} \\
c \circledast P & \widehat{=} & (P ; c \circledast P) \lhd c \rhd \mathrm{II} & \circledast\ \text{-DEF}
\end{array}
$$

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)

$$
\begin{array}{rcll}
II & = & x := x, \quad x \in A & \text{SKIP-ALT} \\
II; P & = & P & \text{;-L-UNIT} \\
P; II & = & P & \text{;-R-UNIT} \\
P; (Q; R) & = & (P; Q); R & \text{;-ASSOC} \\
x := e; x := f & = & x := f[e/x] & :=\text{-SEQ} \\
x := e; y := f & = & y := f[e/x]; x := e & :=\text{-SWAP} \\
x := k; P & = & P[k/x] & :=\text{-INIT} \\
P \lhd \mathsf{true} \rhd Q & = & P & \lhd\rhd\text{-TRUE} \\
P \lhd \mathsf{false} \rhd Q & = & Q & \lhd\rhd\text{-FALSE} \\
(P \lhd c \rhd Q); R & = & (P; R) \lhd c \rhd (Q; R) & \lhd\rhd\text{-;-DISTR} \\
c \circledast P & = & (P; c \circledast P) \lhd c \rhd II & \circledast\text{-UNROLL}
\end{array}
$$

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

Concurrency is HARD! Why?

Consider this simple program (`incx`):

```
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 || Q`

What property should `incx || incx` satisfy?

$x' = x+2$?

It all depends on our model of concurrency

One approach is that each instance of `incx` has its own state (here `x` and `temp`).

$x_1' = x_1+1$ $x_2' = x_2+1$

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

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

$incx_1$: $temp_1 := x;$ $temp_1=temp_1+1;$ $x := temp_1$

$incx_2$: $temp_2 := x;$ $temp_2 := temp_2+1;$ $x := temp_2$

Here we observe $x' = x + 1$!!
(this is 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:

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

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

What do these look like as predicates?

$$skip = ok \Rightarrow ok' \wedge tr' = tr$$
$$e \rightarrow skip = ok \Rightarrow ok' \wedge tr' = tr + [e]$$
$$e \rightarrow P = e \rightarrow skip; P$$

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

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:

$$\boldsymbol{H1}(P) = P \ \wedge tr \leq tr'$$

(now we are no longer doing 1st-order logic!)

Unification Example: Circus

Oliveira, M., Cavalcanti, A. & Woodcock, J. A UTP semantics for *Circus* . *Form Asp Comp* **21**, 3–32 (2009).

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.

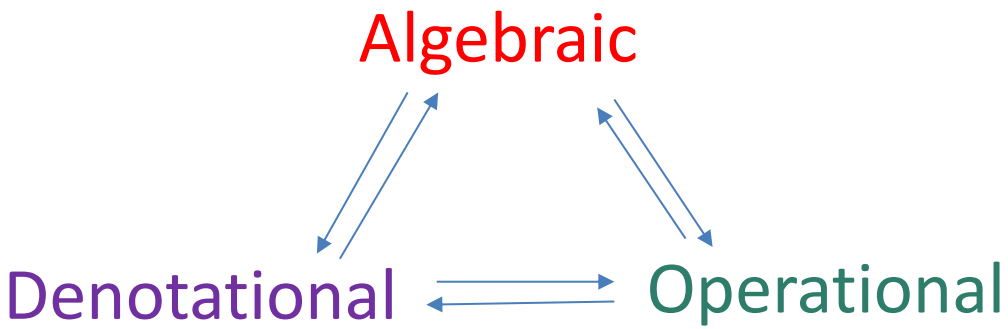
It fused the observations from the two theories,
and identified a common notion of healthiness
called a Design:

$$ok \wedge P \Rightarrow ok' \wedge Q$$

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



UTP also links theories with different alphabets

$$[(\exists c \bullet D(c) \wedge L(c,a)) \Rightarrow S(a)] \quad \textbf{iff} \quad [D(c) \Rightarrow (\forall a \bullet L(c,a) \Rightarrow S(a))]$$

Refinement!

A *Galois Connection* !

My contributions to UTP (1) "Slotted-Circus"

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

Adnan Sherif and Jifeng He. Towards a time model for circus. ICFEM 2002, LNCS 2495,pp 613–624, 2002.

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.

Butterfield, A., Sherif, A., Woodcock, J. (2007). Slotted-Circus. IFM 2007. LNCS 4591.

$$
\begin{aligned}
& A \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, B \\
& \widehat{=}\ \exists\, obs_A, obs_B \bullet \\
& \quad A[obs_A/obs'] \wedge B[obs_B/obs'] \wedge \\
& \quad \left(\begin{array}{l}
\textbf{if} \left(\begin{array}{l} s_A \lhd state_A \neq s_A \lhd state \vee \\ s_B \lhd state_B \neq s_B \lhd state \vee \\ s_A \cap s_B \neq \emptyset \end{array} \right) \\
\textbf{then}\ \ \neg\, ok' \wedge slots \preccurlyeq slots' \\
\textbf{else} \left(\begin{array}{l} ok' = ok_A \wedge ok_B \wedge \\ wait' = (wait_A \vee 1.wait_B) \wedge \\ state' = (s_B \lhd state_A) \oplus (s_A \lhd state_B) \wedge \\ ValidMerge(cs)(slots, slots', slots_A, slots_B) \end{array} \right)
\end{array} \right)
\end{aligned}
$$

My contributions to UTP (2) "Slotted-Circus++"

Adding priority to slotted-Circus

Gancarski, P., Butterfield, A. (2010). Prioritized slotted-Circus . ICTAC 2010. ICTAC 2010. LNCS 6255.

$$
\begin{aligned}
L \vec{\Box} H &\mathrel{\widehat{=}} H \overleftarrow{\Box} L \\
H \overleftarrow{\Box} L &\mathrel{\widehat{=}} H \wedge L \wedge Stop \vee Choice(H, L) \vee WeakChoice(L, H) \\
WeakChoice(L, H) &\mathrel{\widehat{=}} \mathbf{CSP2}\Big(L \wedge \Big(\begin{pmatrix} H \wedge NOEVTS \wedge wait' \end{pmatrix}; \\
&\qquad \mathbf{PRI}\left(\begin{pmatrix} \exists E \bullet \begin{pmatrix} FSTEVTS(E) \\ \wedge\ E \neq \emptyset \wedge E \subseteq sref(tail(slots)) \end{pmatrix} \\ \vee\ slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{pmatrix} \right) \Big)
\end{aligned}
$$

Bresciani, R., Butterfield, A. (2012). A UTP Semantics of pGCL as a Homogeneous Relation. IFM 2012. LNCS 7321.

$$
\begin{aligned}
\texttt{abort} &\mathrel{\widehat{=}} \texttt{true} \\
\texttt{skip} &\mathrel{\widehat{=}} \delta' = \delta \\
\underline{x} := \underline{e} &\mathrel{\widehat{=}} \delta' = \delta\{\!|\underline{e}/\underline{x}|\!\} \\
A ; B &\mathrel{\widehat{=}} \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta') \\
A \lhd c \rhd B &\mathrel{\widehat{=}} \exists \delta_A, \delta_B \bullet A(\delta(\!|c|\!), \delta_A) \wedge B(\delta(\!|\neg c|\!), \delta_B) \wedge \delta' = \delta_A + \delta_B \\
A \ {}_p\!\oplus B &\mathrel{\widehat{=}} \exists \delta_A, \delta_B \bullet A\big(p \cdot \delta, \delta_A\big) \wedge B\big((1-p) \cdot \delta, \delta_B\big) \wedge \delta' = \delta_A + \delta_B
\end{aligned}
$$

My contributions to UTP (3)

Unifying Theory of Concurrent Programs (UTCP)

$$
\begin{array}{rcll}
P \;;;\; Q & \widehat{=} & \mathbf{W}(P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in]) & \langle\!\langle \cdot \mathsf{sem{:}seq} \cdot \rangle\!\rangle \\
P \parallel Q & \widehat{=} & \mathbf{W}(\; A(in \mid ii \mid \ell_{g1}, \ell_{g2}) \vee & \langle\!\langle \cdot \mathsf{sem{:}par} \cdot \rangle\!\rangle \\
 & & \quad P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, in, out] \vee & \\
 & & \quad Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, in, out] \vee & \\
 & & \quad A(\ell_{g1:}, \ell_{g2:} \mid ii \mid out)\;) & \\
P + Q & \widehat{=} & \mathbf{W}(\; P[g_1/g] \vee Q[g_2/g]\;) & \langle\!\langle \cdot \mathsf{sem{:}NDC} \cdot \rangle\!\rangle \\
P^* & \widehat{=} & \mathbf{W}(\; A(in \mid ii \mid \ell_g) \vee & \langle\!\langle \cdot \mathsf{sem{:}star} \cdot \rangle\!\rangle \\
 & & \quad A(\ell_g \mid ii \mid \ell_{g:}) \vee & \\
 & & \quad A(\ell_g \mid ii \mid out) \vee & \\
 & & \quad P[g_{::}, \ell_{g:}, \ell_g/g, in, out]\;) &
\end{array}
$$

Butterfield, A. (2017). UTCP: Compositional Semantics for Shared-Variable Concurrency. SBMF 2017. LNCS 10623.

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:

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

  Side conditions are properties of predicates

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

$$\text{reflexivity-of-equals:}\quad \frac{}{x = x}$$

Axioms for Propositional Logic

$$\neg\text{-I:}\ \frac{P \implies Q \qquad P \implies \neg Q}{\neg P}$$

$$\neg\text{-E:}\ \frac{\neg P}{P \implies R}$$

$$\neg\neg\text{-E:}\ \frac{\neg\neg P}{P}$$

$$\wedge\text{-I:}\ \frac{P \qquad Q}{P \wedge Q}$$

$$\wedge_1\text{-E:}\ \frac{P \wedge Q}{P}$$

$$\wedge_2\text{-E:}\ \frac{P \wedge Q}{Q}$$

$$\vee_1\text{-I:}\ \frac{P}{P \vee Q}$$

$$\vee_2\text{-I:}\ \frac{Q}{P \vee Q}$$

$$\vee\text{-E:}\ \frac{P \vee Q \qquad P \implies R \qquad Q \implies R}{R}$$

$$\implies\text{-E:}\ \frac{P \qquad P \implies Q}{Q}$$

$$\implies\text{-I:}\ \frac{(P \vdash Q)}{P \implies Q}$$

Proof Towers/Trees for Propositions

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}(I)}{\vdash \neg A, A}(\neg R)}{\vdash A \vee \neg A, A}(\vee R_2)}{\vdash A, A \vee \neg A}(PR)}{\vdash A \vee \neg A, A \vee \neg A}(\vee R_1)}{\vdash A \vee \neg A}(CR)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}(I) \quad \dfrac{}{C \vdash C}(I)}{B \vee C \vdash B, C}(\vee L)}{B \vee C \vdash C, B}(PR)}{B \vee C, \neg C \vdash B}(\neg L) \quad \dfrac{}{\neg A \vdash \neg A}(I)}{(B \vee C), \neg C, (B \to \neg A) \vdash \neg A}(\to L)}{(B \vee C), \neg C, ((B \to \neg A) \wedge \neg C) \vdash \neg A}(\wedge L_1)}{(B \vee C), ((B \to \neg A) \wedge \neg C), \neg C \vdash \neg A}(PL)}{(B \vee C), ((B \to \neg A) \wedge \neg C), ((B \to \neg A) \wedge \neg C) \vdash \neg A}(\wedge L_2)}{(B \vee C), ((B \to \neg A) \wedge \neg C) \vdash \neg A}(CL)}{((B \to \neg A) \wedge \neg C), (B \vee C) \vdash \neg A}(PL)}{((B \to \neg A) \wedge \neg C), (A \to (B \vee C)) \vdash \neg A, \neg A}(\to L)}{((B \to \neg A) \wedge \neg C), (A \to (B \vee C)) \vdash \neg A}(CR)}{(A \to (B \vee C)), ((B \to \neg A) \wedge \neg C) \vdash \neg A}(PL)}{(A \to (B \vee C)) \vdash (((B \to \neg A) \wedge \neg C) \to \neg A)}(\to R)}{\vdash ((A \to (B \vee C)) \to (((B \to \neg A) \wedge \neg C) \to \neg A))}(\to R)$$

A more linear approach - “Natural” Deduction

Example of a proof

Number	Formula	Reason
1	A	premise
2	$A \vee A$	From (1) by disjunction introduction
3	$(A \vee A) \wedge A$	From (1) and (2) by conjunction introduction
4	A	From (3) by conjunction elimination
5	$A \vdash A$	Summary of (1) through (4)
6	$\vdash A \to A$	From (5) by conditional proof

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

$$\text{Equanimity:} \quad \frac{P \qquad P = Q}{Q}$$

$$\text{Transitivity:} \quad \frac{P = Q \qquad Q = R}{P = R}$$

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

$$\text{Liebniz:} \quad \frac{F = G}{P[v \leftarrowtail F] = P[v \leftarrowtail G]} \quad [v \text{ a variable}]$$

David Gries, Fred B. Schneider:
A Logical Approach to Discrete Math. Texts and Monographs in Computer Science, Springer 1993, ISBN 0-387-94115-0

Linearising proof trees

$$
\begin{aligned}
& A \\
= \quad & \text{“ why } A = B \text{ ”} \\
& B \\
= \quad & \text{“ why } B = C \text{ ”} \\
& C \\
= \quad & \text{“ why } C = D \text{ ”} \\
& D \\
= \quad & \text{“ why } D = E \text{ ”} \\
& E
\end{aligned}
$$

The proof that $A = B$ can also be linearised, and can simply be inserted into the proof above where “ why $A = B$ ” appears.

true	$\textsc{true}$
$P \equiv P$	$\equiv$-$\textsc{refl}$
$((P \equiv Q) \equiv R) \equiv (P \equiv (Q \equiv R))$	$\equiv$-$\textsc{assoc}$
$P \equiv Q \equiv Q \equiv P$	$\equiv$-$\textsc{symm}$

$$
\begin{array}{ll}
\underline{(\mathsf{true} \equiv P) \equiv P} & \\
= & \text{“ } \equiv\text{-}\textsc{assoc}\ (\mathrm{L2R}) \text{ ”} \\
\quad \mathsf{true} \equiv \underline{(P \equiv P)} & \\
= & \text{“ } \equiv\text{-}\textsc{refl}\ (\mathrm{all}) \text{ ”} \\
\quad \underline{\mathsf{true} \equiv \mathsf{true}} & \\
= & \text{“ } \equiv\text{-}\textsc{refl}\ (\mathrm{all}) \text{ ”} \\
\quad \mathsf{true} &
\end{array}
$$

$\neg(P \equiv Q) \equiv \neg P \equiv Q$ $\quad \neg\text{-}\equiv\text{-DISTR}$

$\mathsf{false} \equiv \neg\mathsf{true}$ $\quad$ FALSE-DEF

$$
\begin{array}{ll}
\underline{(\neg P \equiv Q)} \equiv (P \equiv \neg Q) & \\
= & \text{“ } \neg\text{-}\equiv\text{-DISTR (R2L) ”} \\
\quad \neg(P \equiv Q) \equiv (\underline{P \equiv \neg Q}) & \\
= & \text{“ } \equiv\text{-SYMM (L2R) ”} \\
\quad \neg(P \equiv Q) \equiv (\underline{\neg Q \equiv P}) & \\
= & \text{“ } \neg\text{-}\equiv\text{-DISTR (R2L) ”} \\
\quad \neg(P \equiv Q) \equiv \neg(\underline{Q \equiv P}) & \\
= & \text{“ } \equiv\text{-SYMM (L2R) ”} \\
\quad \underline{\neg(P \equiv Q) \equiv \neg(P \equiv Q)} & \\
= & \text{“ } \equiv\text{-REFL (all) ”} \\
\quad \mathsf{true} &
\end{array}
$$

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!

```
cond_symm : (P◁b▷Q)≡(Q◁¬ b▷P)
by 'red-R2L'
---
(Q◁¬ b▷P)
 = 'match-lhs cond_def @[]'
    { P ↦Q, Q ↦P, b ↦¬ b }
¬ b∧Q∨¬(¬ b)∧P
 = 'match-lhs lnot_invol @[2,1]'
    { P ↦b }
¬ b∧Q∨b∧P
 = 'match-lhs lor_symm @[]'
    { P ↦¬ b∧Q, Q ↦b∧P }
b∧P∨¬ b∧Q
 = 'match-rhs cond_def @[]'
    { P ↦P, Q ↦Q, b ↦«BI (Id "b" 0)» }
(P◁b▷Q)
```

```
sqsupseteq_lor_distr : (P∨Q⊒R)≡(P⊒R)∧(Q⊒R)
by 'red-L2R'
---
P∨Q⊒R
 = 'match-lhs sqsupseteq_def @[]'
    { P ↦P∨Q, Q ↦R }
[P∨Q⟹ R]
 = 'match-lhs ante_lor_distr @[1]'
    { P ↦P, Q ↦Q, R ↦R }
[(P⟹ R)∧(Q⟹ R)]
 = 'match-rhs land_[]_distr @[]'
    { P ↦P⟹ R, Q ↦Q⟹ R }
[P⟹ R]∧[Q⟹ R]
 = 'match-rhs sqsupseteq_def @[1]'
    { P ↦P, Q ↦R }
(P⊒R)∧[Q⟹ R]
 = 'match-rhs sqsupseteq_def @[2]'
    { P ↦Q, Q ↦R }
(P⊒R)  ∧  (Q⊒R)
```

```
sqsupseteq_trans : (P⊒Q)∧(Q⊒R)⟹ (P⊒R)
by 'red-All'
---
(P⊒Q)∧(Q⊒R)⟹ (P⊒R)
 = 'match-lhs sqsupseteq_def @[2]'
    { P ↦P, Q ↦R }
(P⊒Q)∧(Q⊒R)⟹ [P⟹ R]
 = 'match-lhs sqsupseteq_def @[1,1]'
    { P ↦P, Q ↦Q }
[P⟹ Q]∧(Q⊒R)⟹ [P⟹ R]
 = 'match-lhs sqsupseteq_def @[1,2]'
    { P ↦Q, Q ↦R }
[P⟹ Q]∧[Q⟹ R]⟹ [P⟹ R]
 = 'match-lhs land_[]_distr @[1]'
    { P ↦P⟹ Q, Q ↦Q⟹ R }
[(P⟹ Q)∧(Q⟹ R)]⟹ [P⟹ R]
 = 'match-ante implies_trans @[1,1]'
    { P ↦P, Q ↦Q, R ↦R }
[((P⟹ Q)∧(Q⟹ R))∧(P⟹ R)]⟹ [P⟹ R]
 = 'match-rhs land_[]_distr @[1]'
    { P ↦(P⟹ Q)∧(Q⟹ R), Q ↦P⟹ R }
[(P⟹ Q)∧(Q⟹ R)]∧[P⟹ R]⟹ [P⟹ R]
 = 'match-lhs implies_def @[]'
    { P ↦[(P⟹ Q)∧(Q⟹ R)]∧[P⟹ R], Q ↦[P⟹ R] }
[(P⟹ Q)∧(Q⟹ R)]∧[P⟹ R]∨[P⟹ R]≡[P⟹ R]
 = 'match-rhs lor_symm @[1]'
    { P ↦[P⟹ R], Q ↦[(P⟹ Q)∧(Q⟹ R)]∧[P⟹ R] }
[P⟹ R]∨[(P⟹ Q)∧(Q⟹ R)]∧[P⟹ R]≡[P⟹ R]
 = 'match-lhs land_symm @[1,2]'
    { P ↦[(P⟹ Q)∧(Q⟹ R)], Q ↦[P⟹ R] }
[P⟹ R]∨[P⟹ R]∧[(P⟹ Q)∧(Q⟹ R)]≡[P⟹ R]
 = 'match-lhs lor_land_absorb @[1]'
    { P ↦[P⟹ R], Q ↦[(P⟹ Q)∧(Q⟹ R)] }
[P⟹ R]≡[P⟹ R]
 = 'match-all equiv_refl @[]'
    { P ↦[P⟹ R] }
true
```

reasonEq

- 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.

"Saoithín: A Theorem Prover for UTP", UTP 2010.

$U \cdot (TP)^2$ 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", UITP 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.

"UTPCalc —– A Calculator for UTP Predicates", UTP2016.

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/

AdaCore
The GNAT Pro Company

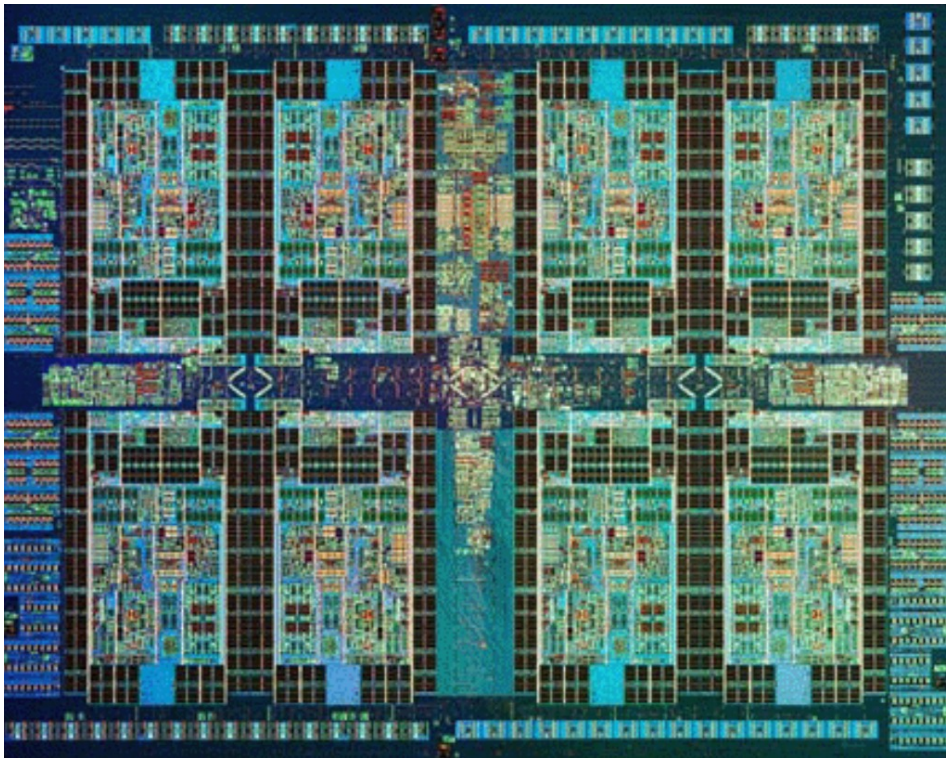
https://www.adacore.com/

PVS

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

- RTEMS = Real-Time Executive for Multiprocessor Systems (www.rtems.org).
- 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

Model bad calls !

Model correct behaviour too!

```
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!)

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.

```
@@@ 1 LOG System running...
@@@ 1 LOG Loop through tasks...
@@@ 2 LOG Clock Started
@@@ 5 TASK Worker1
@@@ 5 CALL NormalPriority
@@@ 5 CALL SetProcessor 0
@@@ 5 WAIT 2
@@@ 4 TASK Worker0
@@@ 4 CALL NormalPriority
@@@ 4 CALL SetProcessor 0
@@@ 4 WAIT 1
@@@ 3 TASK Runner
@@@ 3 CALL NormalPriority
@@@ 3 CALL SetProcessor 1
@@@ 3 CALL barrier_create 0 0 0 1
@@@ 3 SCALAR rc 3
@@@ 3 SIGNAL 1
@@@ 4 LOG WAIT 1 Over
```

```
Barrier Manager Model running.
Setup...
@@@ 0 LOG TestName: Barrier_Manager_TestGen
@@@ 0 DEF MAX_BARRIERS 2
@@@ 0 DEF BARRIER_MAN 0
@@@ 0 DEF BARRIER_AUTO 1
@@@ 0 DEF MAX_WAITERS 3
@@@ 0 DEF TASK_MAX 4
@@@ 0 DEF SEMA_MAX 3
@@@ 0 DCLARRAY Semaphore semaphore SEMA_MAX
@@@ 0 INIT
@@@ 0 LOG scenario ManAcqRel
@@@ 0 LOG sub-senario bad create, invalid name
@@@ 0 LOG sub-senario multicore enabled, cores:(1,0,0)
```

Refinement

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

Simple binary semaphores

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

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

C test code for a call to rtems_barrier_create:

```
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 = mergeattribs({1});
  rc = rtems_barrier_create({0}, attribs, {2}, {3} ? &bid : NULL);
  T_log(T_NORMAL, "Returned 0x%x 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.

Previous Next

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 !


```
# Refinement Mechanisms
#   Direct Output - no lookup
#   Keyword Refinement - lookup (the
#   Name Refinement - lookup (<s)
#    The same name may appear in dif
#    We add '_XXX' suffixes to looku
#     _DCL - A variable declaration
#     _PTR - The pointer value itself
#     _FSCALAR - A scalar that is a struct field
#     _FPTR - A pointer that is a struct field
#   Type Refinement - lookup (first) type
#   Typed-Name Refinement - lookup type-name

# LOG <word1> ... <wordN>
#  Direct Output


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

Trinity College Dublin

The University of Dublin

Thank you! Any (further) Qs?