Towards a Unifying Semantics Playground

Andrew Butterfield
Trinity College Dublin, Ireland

ABSTRACT We give a brief presentation of how Unifying Theories of Programming (UTP) can be used to define program languages and their semantics. We describe and show examples of the use of a prototype theorem-prover `reasonEq` that supports equational reasoning for UTP. We discuss how this tool is intended to support work into various formal approaches to concurrency.

KEYWORDS Unifying Theories of Programming, Theory Prototyping, Formal Semantics.

1. Introduction

The Unifying Theories of Programming framework (UTP) (Hoare & He 1998a) uses predicate calculus to define before-after relationships over appropriate collections of free observation variables. The before-variables are undecorated ($e.g.x$), while after-variables have dashes ($e.g.x'$)[1]. A notion of healthiness is used to characterise predicates that correspond to realistic behaviours, e.g. ruling out assertions such as stating that a program has terminated, but has also not started. It models both specifications and code, as well as the refinement relationships that hold between them. It has been applied to a wide variety of specification and programming paradigms (Dunne & Stoddart 2006; Butterfield 2010b; Qin 2010; Wolff et al. 2013; Naumann 2015; Bowen & Zhu 2017; Ribeiro & Sampaio 2019)

We have been developing proof assistance tools to support the development of various UTP theories (Butterfield 2010a, 2012, 2014, 2016). The key issue here has been having tools that help explore theories under development, when most putative "theorems" are actually false. The use of equational reasoning is of key importance in this area. This has culminated in the current tool under development, called `reasonEq`[2].

Here we describe how `reasonEq` can be used to develop and reason about both specifications and programming languages. We will look at a simple imperative language to explain UTP concepts, and illustrate how `reasonEq` is used to model this.

We finish by discussing current work using `reasonEq` to support the developement of Unifying Theories of Concurrent Programming (UTCP) (Butterfield 2017), and to apply UTP to model and verify a technique developed to use Promela models to generate test code for the RTEMS operating system.

JOT reference format:
Andrew Butterfield. *Towards a Unifying Semantics Playground*. Journal of Object Technology. Vol. 25, No. 1, 2026. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0)
http://dx.doi.org/10.5381/jot.2026.25.1.a5

[1] "dash" and "prime" are considered synonyms in UTP
[2] https://github.com/andrewbutterfield/reasonEq

2. Unifying Theories of Programming

2.1. The While Language

We give a very brief introduction to the Unifying Theories of Programming (UTP) framework (Hoare & He 1998b), using a simple While language as an example. We define the following metavariables:

u, v, x, y, z	Program variables
b, c	Condition Predicates (before-variables)
e, f	Expressions (before-variables)
P, Q, R, S, X	General Predicates (before- and after-variables)
σ, σ'	Sets of before- and after-variables

The While language has the expected assignment,sequencing,conditional, and iteration constructs, to which we add skip to denote a computation that makes no state-changes and terminates immediately.

$$
\begin{array}{llll}
P, Q \in \mathit{While} & & & \\
& = & \mathit{II} & \text{skip} \\
& \mid & v := e & \text{assignment} \\
& \mid & P \lhd c \rhd Q & \text{conditional} \\
& \mid & P; Q & \text{sequencing} \\
& \mid & c * P & \text{while loop}
\end{array}
$$

2.2. UTP Basics

A UTP theory defines an *alphabet*, which is a set of variables that capture relevant aspects of observable behaviour. For a language like While we expect to be interested in the values of program variables, both before and after the program has run. So if a program has variable v we could use observation v to denote its before-value, and v' to denote its after-value. This follows the convention used in the Z formal specification language (J. C. P. Woodcock & Davies 1996).

If we had an algorithm in mind that involved three program variables x, y, and z, we would say that our alphabet would contain x, x', y, y', z, z'. We will also find it useful to be able to refer to the set of all available variables, so introduce $\sigma = \{x, y, z\}$ and σ' defined similarly. Also when we refer to an exrpression e, we mean this to be an expression evaluated on before-values, while e' is the same expression over after-values. We could then propose the following semantics for assignment, in which the final value v' is equal to e which is evaluated in the before-state, and all the other variables are unchanged.

$$v := e \quad \widehat{=} \quad v' = e \ \wedge \ \sigma' \setminus v' = \sigma \setminus v$$

The semantics for skip would then be $\mathit{II} \ \widehat{=} \ \sigma' = \sigma$.

2.3. UTP Designs

However, just observing relationships between σ and σ' is not sufficient for a complete semantics. In particular, we need to have some way to observe program *termination*. We do this by introducing two new boolean observation variables, conventionally called ok and ok', that record if the program was started or stopped. At this point we stress that ok refers solely to the point just before the program runs: if **false**, the program has not started, if **true** it has started. Similarly, ok' is **true** if the program has just terminated properly, but **false** if it fails to terminate at all. So, for example, $ok \wedge \neg ok'$ describes a program that enters an infinite loop. This interpretation, which applies to sequential programs, is modified later on (see §2.5.2).

The idea now is to be able to take a precondition P and postcondition Q, both defined over σ but not mentioning ok or ok' and use them in the following construct:

$$ok \wedge P \implies ok' \wedge Q$$

If the program has started ($ok = true$) and P is satisfied, then the program will terminate ($ok' = true$) and Q will be satisfied.

We refer to this construct as a *Design*, and we use the notation $P \vdash Q$ as a shorthand for the above. This gives us a total-correctness semantics for While:

$$
\begin{array}{rcl}
\mathit{II} & \widehat{=} & \mathbf{true} \vdash \sigma' = \sigma \\
v := e & \widehat{=} & \mathbf{true} \vdash v' = e \wedge \sigma' \setminus v' = \sigma \setminus v \\
P \lhd c \rhd Q & \widehat{=} & c \wedge P \vee \neg c \wedge Q \\
P; Q & \widehat{=} & \exists \sigma_m \bullet P[\sigma_m/\sigma'] \wedge Q[\sigma_m/\sigma] \\
c * P & \widehat{=} & \mu X \bullet (P; X) \lhd c \rhd \mathit{II}
\end{array}
$$

The definition of sequential composition $P; Q$ basically selects a fresh set σ_m of midpoint variables, replaces the after variables in P and the before variables in Q by them, and then hides them under existential quantification. Note that the semantic constructs for conditionals, sequencing and iteration preserve "design-hood" — this is covered in UTP under the concept of so-called "healthiness conditions".

In the above semantics we are expressing the semantics of iteration as a least fixed-point in the usual way. The underlying ordering is based on the fact that predicates form a complete lattice. It can be shown that the subset of predicates that comprise designs is also a complete lattice. This is important because UTP isn't just about program languages, but is also about specification languages and refinements between them. In the context of designs, this means that $P \vdash Q$ can be considered a pre/post specification, e.g., pick a number between one and ten: $\mathbf{true} \vdash n' \in \{1, \ldots, 10\}$. A possible implementation of this is $n := 7$. We can now extend our language to be able to state that program P refines specification S ($P \sqsupseteq S$) and it then makes sense to add non-deterministic choice ($\sqcap$) to our extended language:

$$
\begin{array}{rcl}
P \sqsupseteq S & \widehat{=} & [S \implies P] \\
P \sqcap Q & \widehat{=} & P \vee Q
\end{array}
$$

Here the notation $[P]$ denotes the universal closure of P, in which we quantify over all observables: $\forall \alpha, \alpha' \bullet P$, where $\alpha = \sigma \cup \{ok\}$.

Now we state and prove that $n := 7 \sqsupseteq \mathbf{true} \vdash n' \in \{1, \ldots, 10\}$, and can describe a partial refinement: $n := 3 \sqcap n := 7$.

As we have a complete lattice, we have both top and bottom. The bottom of the lattice ($\bot$) is $(\mathbf{false} \vdash \mathbf{true})$, which reduces to $\mathbf{true}$, that denotes complete unpredictability, and usually models unrecoverable error conditions. We typically call it "abort" or "chaos". The top of the lattice ($\top$) is $(\mathbf{true} \vdash \mathbf{false})$ which reduces to $\neg ok$, and denotes the program that cannot be started. It refines everything below it, so if it could be started it would satisfy any specification, which is why it is generally referred to as "miracle". More prosaically, it denotes infeasibility. Many approaches to formal semantics are happy to have an ordering with bottom, but avoid closing over the top, for precisely this reason. In UTP we keep miracle around because it arises when something is infeasible, and can also be useful when specifying certain kinds of behaviours.

2.4. UTP Algebra and more...

The semantic framework described so far is basically a denotational one, but UTP is also about algebraic and operational semantics. In particular, we can also define a algebraic semantics for While, with some laws illustrated below:

$$\begin{aligned}
I\!I;P &= P\\
\mathbf{true};P &= \mathbf{true}\\
P \sqcap P &= P\\
P \sqcap Q &= Q \sqcap P\\
P;(Q;R) &= (P;Q);R\\
P \lhd b \rhd P &\equiv P\\
(P \lhd b \rhd Q) \lhd c \rhd R &\equiv P \lhd b \land c \rhd (Q \lhd c \rhd R)\\
(P \sqcap Q);R &= (P;R) \sqcap (Q;R)\\
c * P &= (P;c * P) \lhd c \rhd I\!I
\end{aligned}$$

All of these can be proven from the denotational semantics. It is also possible to develop an operational semantics in UTP, and be able to connect it to the denotational and algebraic forms. Most of the original book focusses on starting with the denotational semantics, and then using that to prove the correctness of the albebraic laws. It briefly discusses operational semantics at the end, including bisimulation, and then uses it to validate algebraic laws.

A lot of recent work is looking at starting with one of the above approaches to define a semantics, and then using UTP principles to systematically derive the other two.

An alphabet is called *homogeneous* if every before-observation (x) is accompanied by the corresponding after-observation (x'). Many UTP theories have homogeneous alphabets, but not all. Exceptions include concurrent logic programming (Hoare & He 1998b, §7.6), and much work on probabilistic languages (Zhu et al. 2012). Almost all UTP theories include nondeterminism, refinement, abort and miracle. The definitions of nondeterminism and refinement rarely change, that of sequential compositions needs to be different if the alphabet is non-homogeneous, and the definitions, *and interpretations*, of abort and miracle depend on the relevant healthiness conditions.

2.5. CSP in UTP

To illustrate this, let us now take a quick look at some of the UTP semantics for something quite different: the process algebra for Communicating Sequential Processes (CSP) (Hoare 1985). This is a notation for describing mutiple concurrent processes that participate in *Events* (atomic observable happenings) with constraints regarding which events require which processes to synchronise regarding their occurrence.

2.5.1. CSP Description We assume a set $Events$ of all possible events, ands let $e, \ldots$ range over such events, and $A, \ldots$ range over sets of events ($A \subseteq Events$).

$$\begin{array}{lll}
P, Q \in CSP & & \\
\quad = & Stop & \text{deadlock}\\
\quad \mid & Skip & \text{terminate now}\\
\quad \mid & e \rightarrow P & \text{prefix}\\
\quad \mid & P;Q & \text{sequencing}\\
\quad \mid & P \Box Q & \text{external choice}\\
\quad \mid & P \sqcap Q & \text{non-deterministic choice}\\
\quad \mid & P \parallel_A Q & \text{parallel composition}\\
\quad \mid & P \setminus A & \text{hiding}
\end{array}$$

Here $e \rightarrow P$ describes a process willing to participate in event e, after which it will behave as P. We say that it "accepts" e. The semantics of CSP is based on the notion of defining the behaviour of a given process P with respect to an external *environment*, which is typically other CSP processes. External choice $P \Box Q$ allows other processes to "offer" events, and which of P or Q runs depends on if they accept the offered events. The way in which process environments are built up is via the parallel construct. Process $P \parallel_A Q$ runs P and Q together, so Q is part of the environment of P, and vice-versa. It also requires that P and Q synchronise on events in A. What this means precisely is that if P and Q accept event $a \in A$, then they both participate in a single instance of that event. Events not in A are not constrained, and if common to both P and Q, are performed as independent instances. Hiding ($P \setminus A$) means that events in A are performed by P, but are hidden to are visible to the environment.

2.5.2. CSP Alphabet The alphabet for this theory is quite different, with the most prominent being the lack of variables with side-effects. Instead the semantics of CSP is based on sets of sequences of events, associated with an indication of which events a process is willing to participate in at any given point in its execution. Another key difference is that the semantics now need to be able to talk about running processes that have paused waiting for some other process to do something. Finally, unlike in the While language, non-termination is not a failure mode, but is often a required feature (e.g. most server applications).

The UTP alphabet for a CSP process defines the set of possible events $Event$, and uses the following CSP-specific observations:

- $tr : Event^*$ is the sequence of events observed so far
- $wait : \mathbb{B}$ is true if the process is paused waiting for events
- $ref : \mathcal{P}Event$ is the set of events currently being refused.

It might come as a surprise that the CSP alphabet also includes ok and ok', and that CSP process are also designs. This is because, in UTP, ok actually denotes *stability*, which is the absence of serious errors or failures. For the While language, stability means being started properly, and eventually terminating. In CSP, stability is being started in a non-divergent state, and remaining so when ever observed. In CSP divergence is characterised by a process that keeps running without performing any events, which is basically livelock.

– $ok : \mathbb{B}$ is true if the process is not divergent

We also have final observations ok', tr', $wait'$, ref' to define a homogeneous relation.

For sequential programs, ok and ok' talk about pre- and post-conditions, and so are anchored to the beginning and end points of the program run. With process algebras like CSP, ok, and $wait$ are still anchored to the start, but ok' and $wait'$ can refer to points during the evolution of the process. This asymmetrical view works fine for process algebras whose semantics is based on sets of traces, and which does not support shared mutable state. Shared mutable state changes things further (see end of §2.6.3).

2.5.3. Reactive Systems In UTP, the alphabet above, coupled with three healthiness conditions, characterises what are termed *Reactive* systems, that capture a wide range of event-based concurrency models, including ACP, CCS, Data Flow and CSP. Healthiness conditions are described by laws that healthy predicates are required to obey. Often these can be defined by a monotonic, idempotent predicate transformer, which makes the theory easy to reason about. Here we use these to describe healthy *reactive processes*, which satisfy $P = \mathbf{R}(P)$, where $\mathbf{R}{=}\mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}$, defined below:

– The after-trace can only extend the before-trace:

$$\mathbf{R1}(P) \mathrel{\widehat=} P \land tr \leq tr'$$

In other words we cannot go back in time.

– We only define how the trace is extended:

$$\mathbf{R2}(P) \mathrel{\widehat=} \exists s \bullet P[s, s \frown (tr' - tr)]$$

In other words we cannot observe any past history.

– While waiting at the start there is no state change

$$\mathbf{R3}(P) \mathrel{\widehat=} II \lhd wait \rhd P$$

Here II is so-called reactive-skip.

Reactive-skip II, basically states that if we are divergent at the start ($\neg ok$), then the only thing we can rely on is that any observed final trace will be an extension of the starting trace. If not divergent, we behave like design II:

$$\begin{aligned} II \quad &\mathrel{\widehat=} \quad \neg ok \land tr \leq tr' \\ & \qquad \lor ok' \land tr' = tr \land wait' = wait \land ref' = ref \end{aligned}$$

2.5.4. CSP Semantics CSP is a *Reactive Design*, in that it satisfies the healthiness conditions of both designs and reactive processes. It also has its own healthiness conditions that distinguish it from other reactive theories (e.g., ACP, CCS, etc.):

$$\begin{aligned} \mathbf{CSP1}(P) &\mathrel{\widehat=} P \lor \neg ok \land tr \leq tr' \\ \mathbf{CSP2}(P) &\mathrel{\widehat=} P ; J \\ J &\mathrel{\widehat=} (ok \implies ok') \land \\ & \qquad tr' = tr \land wait' = wait \land ref' = ref \end{aligned}$$

CSP processes form a complete lattice, and any CSP process satisfies:

$$P = \mathbf{R}(\neg P[false, false/wait, ok'] \vdash P[false, true/wait, ok'])$$

Literally a reactive design!

We can show an extract of some of the CSP semantics:

$$\begin{aligned} Stop &\mathrel{\widehat=} \mathbf{R}(wait := true) \\ Skip &\mathrel{\widehat=} \mathbf{R}(\exists ref \bullet II) \\ e \to p &\mathrel{\widehat=} (e \to Skip) ; P \\ e \to Skip &\mathrel{\widehat=} \mathbf{CSP1}(ok' \land do(e)) \\ do(e) &\mathrel{\widehat=} (e \notin ref' \lhd wait' \rhd tr' = tr \frown \langle e \rangle) \\ & \qquad \land (tr' = tr \land wait' \lor tr < tr') \end{aligned}$$

We are using assignment above as an obvious shorthand for stating only $wait$ is modified. Essentially $do(e)$ states that if waiting ($wait'$), the trace has not been extended and the process is not refusing e. If not waiting, then the trace has been extended with event e.

For those wanting more details, the tutorials by Ana Cavalcanti and Jim Woodcock are highly recommended (Cavalcanti & Woodcock 2004).

2.6. Linking Theories

Another key concept in UTP is the ability to link theories with different alphabets, typically when one can be considered a refinement of the other. Basically a Galois connection can be established between the two, using a linking predicate that relates abstract and concrete observations (Hoare & He 1998b, §1.6, pp40-41). Consider an theory of an abstract notation that has alphabet a, and a theory of a concrete notation with alphabet c. We have a specification $S(a)$ and a corresponding design $D(c)$. How do we proceed to show the latter implements the former? We need to have a linking predicate $L(c, a)$ that relates observables in the two theories. Given such a link, a correctness proof can be done by either:

– computing the strongest specification satisfying the the design and showing it implies the specification:

$$(\exists c \bullet D(c) \land L(c, a)) \implies S(a)$$

– computing the weakest design satisfying the specification and showing it is implied by the design:

$$D(c) \implies (\forall a \bullet L(c, a) \implies S)$$

Both are equivalent and form a Galois connection, but one may be easier than the other depending on circumstances.

In (Hoare & He 1998b, Chapter 4) there is a comprehensive treatment of these Galois connections, leading to topics such as prespecification and postspecification (Section 4.3), and the use of simulations to deal with datatype abstration (Section 4.4). In Chapter 6, the issue of connecting abstract theories to implementations are discussed. This basically introduces a notion of a program step, focussing on assembly-level programs.

2.6.1. *Beyond "The Book"*

The publication of Unifying Theories of Programming(Hoare & He 1998b) triggered a lot of research activity and the organisation of a (mostly) biennial series of symposia: (Dunne & Stoddart 2006; Butterfield 2010b; Qin 2010; Wolff et al. 2013; Naumann 2015; Bowen & Zhu 2017; Ribeiro & Sampaio 2019).

2.6.2. *Circus*

One key development for our work was the development in UTP of a unification of Z with CSP called *Circus* (J. Woodcock & Cavalcanti 2001; M. V. M. Oliveira 2005; M. Oliveira et al. 2009). This basically added in program variables to the observables, typically using observations $state$ and $state'$ where both are a finite mapping from variable names to values:

$$\begin{array}{rcl} ok, ok' & : & \mathbb{B} \\ tr, tr' & : & Event^* \\ wait, wait' & : & \mathbb{B} \\ ref, ref' & : & \mathcal{P} Event \\ state, state' & : & Var \nrightarrow Value \end{array}$$

A key challenge was managing $state$ shared between parallel processes.

The development of *Circus* triggered a whole sequence of follow-on developments. Adnan Sherif and Jifeng He developed a timed version called CT^* (Sherif & He 2002). This was a discrete time model in which there was a sequence of time-slots, within which a finite sequence of events and a refusal could take place: $tr, tr' : (Event^*, \mathcal{P}Event)^*$. We had been collaborating with Jim Woodcock on a formal semantics for the Handel-C language that compiled C code with added parallel constructs into synchrous hardware descriptions (Butterfield & Woodcock 2002, 2003, 2005, 2004; Butterfield 2007). It became very clear that the semantic domains for Handel-C were a good fit for the CT^* domain, so we started to explore using UTP for Handel-C. The key idea was to abstract over the structure of a time-slot, and this led to a theory called Slotted-Circus (Butterfield et al. 2007). Work by two Ph.D students added notions of priority (Gancarski (Butterfield et al. 2009; Gancarski & Butterfield 2009, 2010)) and probability (Bresciani (Bresciani & Butterfield 2012b,a, 2014)). These theories were being developed by hand, and it was becoming increasingly clear that tool support was required. The next section will describe the steps taken to provide that support.

In parallel with those steps, UTP theory development continued. In particular, my attention was drawn to a paper introducing a concept called "Views" (Dinsdale-Young et al. 2013). This defines an abstraction that can capture a wide range of techninques for modelling and reasoning about concurrency: Owicki-Gries (Owicki & Gries 1976), Rely-Guarantee (Jones 1983), Separation Logic (Calcagno et al. 2007), among others. It introduced a base-line language of *Commands*, built over a notion of atomic actions $a \in Atom$:

$$\begin{array}{rlll} C \in Commands & & & \\ & = & \langle a \rangle & \text{Atomic Command} \\ & \mid & C \mathbin{;\!;} C & \text{Sequential Composition} \\ & \mid & C + C & \text{Non-deterministic Choice} \\ & \mid & C \parallel C & \text{Parallel Composition} \\ & \mid & C^* & \text{Non-deterministic Iteration} \end{array}$$

This is the weakest View, defining an algebra with maximal nondeterminism. Basically the paper describes how all the various concurrency frameworks could be built over this baseline. The language is given an operational semantics, and is shown to obey a set of algebraic laws. If the concepts of abort and miracle are added, what results is Concurrent Kleene Algebra (CKA). The concept of a CKA has emerged in recent years as a result of work by Tony Hoare and others (Hoare et al. 2009), as being the definitive algrebraic model/semantics of shared-state concurrency.

The Views paper is really about classification: it shows how apparently diverse reasoning techniques have a common underlying structure. However, it doesn't describe how we might unify these in some way. For example, a well-designed concurrent algorithm might have a part whose correctness can be done using Owicki-Gries, while another part relies on separation logic. If would be nice if we could explain how to combine results from those two techniques to verify the algorithm as a whole. This struck us as something that UTP ought to be able to address.

2.6.3. *UTCP*

To this end, we developed Unifying Theories of Concurrent Programming (UTCP) to give a semantics to the Command language. We took work on Unifying Theories of Parallel Programming (UTPP) (J. Woodcock & Hughes 2002) as a baseline and work started to develop the theory (Butterfield et al. 2016; Butterfield 2017, 2024).

Our UTCP language is basically a UTP theory that defines a denotational semantics for the Views Command language with the addition of miracle (Butterfield 2024). As done in the Views paper, we will be able to reason about concurrency by tailoring atomic actions and corresponding axioms to capture any given paradigm for reasoning about concurrency. We hope to explore how we can then combine different approaches.

Initial work on it in `reasonEq` is focussing on defining the concept of atomic actions called A and X (eXtended A). We currently have two definitions (axioms) and two desirable properties (conjectures) as shown in Fig. 1 We won't explain this, as this is to illustrate what we are aiming to handle (Hint: ls is a control variable that contains labels of atomic statements that are currently enabled).

Developing the UTP semantics for Commands involved a lot of trial-and-error, and endless complex test calculations — yes, as a result of the combinatorial explosion that results from shared-variable concurrency! It drove the development of tool support for those calculations as well.

$$
\begin{array}{rcl}
X(E,a,R,N) & \widehat{=} & E \subseteq ls \vdash a \land ls' = (ls \setminus R) \cup N \\
A(E,a,N) & \widehat{=} & X(E,a,E,N) \\
A(E,a,N) & \equiv & E \subseteq ls \vdash a \land ls' = (ls \setminus E) \cup N \\
X(E_1,a,R_1,N_1); X(E_2,b,R_2,N_1) & \equiv & E_2 \cap (R_1 \setminus N_1) = \emptyset \land X(E_1 \cup (E_2 \setminus N_1), (a;b), R_1 \cup R_2, (N_1 \setminus R_2) \cup N_2)
\end{array}
$$

Figure 1 Defining atomic actions in UTCP

Observe also that here we are using Designs to define these atomic actions, so *ok* and *ok'* are involved. A result of all those complex calculations was the realisation that the interpretation of *ok* had to change. Instead of being limited to the start, it can now refer to any intermediate point in the running of the program. The only constraint is that *ok'* refers to an observation point at or after the point to which *ok* refers. This turns out to be crucial to observing state interference.

3. Using `reasonEq`

3.1. Motivation

As the work was progressing with the priority and probability variants of slotted-Circus, it was becoming increasingly clear that doing proofs by hand wasn't scaling well. So we started to explore existing tool support, with emphasis on PVS (Owre et al. 1992), CoQ (Bertot & Castéran 2004), and Isabelle/HOL (Nipkow et al. 2002), as these are well-established, well maintained theorem provers. However none proved to be suitable, and we (reluctantly) decided to build our own (Butterfield 2010a, 2012, 2014, 2016). As we explained in (Butterfield 2010a):

Foundational work in UTP reasons about: predicate transformers, predicates about predicates, User defined languages, meta-properties of language constructs, recursive predicates, undefinedness, and defining new language constructs by treating programs as predicates (Hehner 1984). These require using 2nd-order or higher semi-classical logic. We want tool support that provides all of the above, and more, in as transparent a manner as possible. And finally, we *really* like equational reasoning.

PVS had no way to add syntax, CoQ is very hard to use in practice, and it and Isabelle/HOl had a tendency to do proof steps that resulted in a complex expressions that were hard to read or understand.

Equational reasoning was developed by formal methods researchers to simplify the proof process, pioneered by Dijkstra and colleagues. A good introduction by Gries and Schneider ((Gries & Schneider 1994)) forms the basis and inspiration for our tool development. The key thing about equational reasoning is that each step involves replacing a part of the current goal predicate, by a predicate deemed equivalent by some law in scope:

$$
\begin{array}{ll}
 & predicateWithSubPart(part) \\
= & \text{“using this law: } part = newp\text{”} \\
 & predicateWithSubPart(newp)
\end{array}
$$

The point is that this step depends *only* on the two predicate lines above, and the cited law. The problem with proofs in natural deduction or sequent style is that a given step can depend on many other proof lines, an *arbitrary distance away*. Our choice of prover technology is not based on some deep fundamental lack of power or expressivness, but on the useability of the reasoning framework. We need *adequacy*, not *completness*. Ultimately we were inspired by seeing a demo of the prover in the Raise Development method (George et al. 1995) which used such equational reasoning.

All the tooling we have done has been developed from scratch, written in Haskell. The initial version (Butterfield 2010a) was developed with a GUI interface, using wxHaskell which provided a wrapper around the wxWidget C++ GUI library. This initially restricted its use to machines running Windows, but over time it was possible to run it on Linux. The logic we implemented (Butterfield 2012) is an adaptation of the first-order equational logic described by Tourlakis (Tourlakis 2001), that fully formalises the logic of Dijkstra, Gries and Schneider (Gries & Schneider 1994). We also presented a paper at the UITP 2014 workshop were we talked about “proof by pointing”, (Butterfield 2014). There we reported on student feedback that basically indicated that the learning curve was easy (because experimentation was easy), and most issues had to do with user interface design. We also solicited advice regarding soundness, and the idea that we might pass our proofs into Isabelle/HOL for checking was discussed. Interestingly, Jim Woodcock, Simon Foster and Frank Zeyda had started work using Isabelle to procude Isabelle/UTP (Foster et al. 2014).

Around this time, two key things happened:

1. The theorem prover with its wxHaskell GUI interface was becoming unmaintainable, and it became clear that it needed a complete rewrite (the GUI wasn't the issue, but it was a distraction).

2. Work being done (manually) on the semantics of UTCP was getting bogged down with test calculcations, and the theorem prover was no help with these. What was interesting was that the test calculations followed a fairly regular script: unfold definitions until all done, keep simplifying until done, then repeat this until no further changes occur. Then do something similar with fold instead of unfold until stuck. The outcome was that either the hoped for result emerged, or the calculation would be blocked, and it was time to put the thinking cap back on. The issue was the combinatorial explosion during the unfold phase.

The solution for the UTCP semantics was to develop a "UTP-calculator" (Butterfield 2016) which made running through the calculations very easy, and which ultimately revealed a way in which they could be automated.

We decided to redesgn the theorem prover from scratch, and so `reasonEq` was born. The initial plan was to stick with a terminal user interface (TUI), so we could focus on building the logic, with a plan to a GUI later on. To this end we provided an abstract user interface that sits between the TUI and GUI on the on side, and the underlying theorem prover machinery on the other side. What is interesting is that the TUI works very well, and, apart from some student projects exploring the idea, we have not put any effort into developing a GUI.

3.2. `reasonEq` Overview

The `reasonEq` program implements equational reasoning, with support for UTP theories built-in. From our perspective, a UTP theory is built up from the following components:

Alphabet is a list of the observation variables.

Signature is a list of the language forms available

Axioms are predicates that define the semantics of the signature.

Laws are predicates that express further properties of the language.

The laws should be provable from the given axioms, and in our context are treated as *conjectures* to be proven. The `reasonEq` theorem prover is built around the notion of `reasonEq` *theories*, as described above. In `reasonEq` the alphabet and signatures are combined into a table of *known variables*. We have a general notion of *terms* built up using constants, variables, constructors, quantifiers, which can represent both *predicates* and *expressions*.

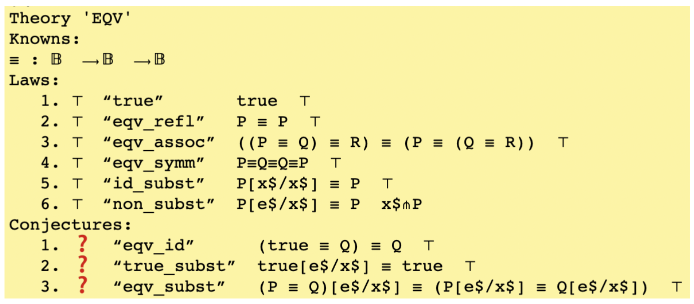
Importantly, we use a generalised form of quantification, where we can have a list of bound variables, and where some of those themselves can actually denote lists or sets of variables. Assume we have the alphabet x, x', y, y'. We can define sequential composition over that alphabet $(P; Q)$ in the following three ways:

$$\begin{array}{rcl} \exists x_m \bullet \exists y_m & \bullet & P[x_m, y_m / x', y,] \land Q[x_m, y_m / x, y] \\ \exists x_m, y_m & \bullet & P[x_m, y_m / x', y,] \land Q[x_m, y_m / x, y] \\ \exists \sigma_m & \bullet & P[\sigma_m / \sigma'] \land Q[\sigma_m / \sigma] \end{array}$$

The last form has the advantage that we can work with it abstractly to prove general laws about sequential composition, but also use it in a context where σ gets instantiated (to x, y say).

We also have a notion of *polymorphic types*, with type-inferencing and checking. All predicates have type boolean. Finally, we have a side-condition language, which is basically a conjuction of relations between designated variables and sets/lists of variables.

In general, each theory will depend on other theories, those dependencies forming a directed acyclic graph. Basically we start with theories that capture basic propositional calculus, then add quantifiers, and start to layer programming concepts on top of those. The theories that define the basic logic are introduced one logical construct at a time, which each one depending on those before it ($\equiv, \neg, \lor, \land, \implies, \forall, \exists$) as detailed in (Gries & Schneider 1994). We also have a theory about closures (universal and existential). We need to support various types of expressions, so rithmetic, sets and sequences each get their own theory. UTP theories are layered up starting with a base theory, and currently we have theories under construction regarding designs, the While language, and UTCP. In a recent development, a comprehensive deduction system for Linear Temporal Logic (LTL) has been developed (Warford et al. 2021), and we have set up a `reasonEq` theory that captures this.



```
Theory 'EQV'
Knowns:
≡ : 𝔹  ⟶𝔹  ⟶𝔹
Laws:
   1. ⊤  "true"        true  ⊤
   2. ⊤  "eqv_refl"    P ≡ P  ⊤
   3. ⊤  "eqv_assoc"   ((P ≡ Q) ≡ R) ≡ (P ≡ (Q ≡ R))  ⊤
   4. ⊤  "eqv_symm"    P≡Q≡Q≡P  ⊤
   5. ⊤  "id_subst"    P[x$/x$] ≡ P  ⊤
   6. ⊤  "non_subst"   P[e$/x$] ≡ P  x$⋔P
Conjectures:
   1. ?  "eqv_id"       (true ≡ Q) ≡ Q  ⊤
   2. ?  "true_subst"   true[e$/x$] ≡ true  ⊤
   3. ?  "eqv_subst"    (P ≡ Q)[e$/x$] ≡ (P[e$/x$] ≡ Q[e$/x$])  ⊤
```

Figure 2 Screenshot: `EQV` theory as displayed in the TUI

3.3. Proof in `reasonEq`

We shall illustrate how proof works with a simple example from the logical equivalence theory, called `EQV`. Within `reasonEq`, this theory is displayed as shown in Fig. 2. We see that the theory defines one known name "≡", whose type ($\mathbb{B} \to \mathbb{B} \to \mathbb{B}$) says it is a binary operator that combines two predicates to produce a predicate. It then lists six laws, and three conjectures, all numbered. Each lists four components:

Provenance ⊤ denotes an axiom, **?** denotes a conjecture.

Name The underlying name, limited to alphanumeric characters and dash, dot and underscore (used to generate filenames).

Predicate Logical predicate rendered using UTF8 Unicode.

Side-Condition ⊤ denotes true/none, $x\$ \pitchfork P$ is short for $(x\$ \cap \mathsf{fv}.(P) = \emptyset)$.

In (Gries & Schneider 1994), Law 1 is a conjecture, while Conjecture 1 is an axiom. We swap these around to simplify checking for proof completeness. Laws 5 and 6 and Conjectures 2 and 3 are not found in the book, but are needed in `reasonEq`.

We shall now walk through doing a simple proof in `reasonEq`, for Conjecture 1 that says **true** is an identity for ≡. We invoke the prover, and it displays our initial goal with the predicate on the first line, and the (trivial) side-condition on the second:

```
(true ≡ Q) ≡ Q
 ⊤
```

Now we match our current goal ((**true** $\equiv Q) \equiv Q$) to get three successful ranked matches, each identifying which law matched and which "side" of the law was matched. The term shown in blue is the result should that law be applied. The $\top \implies \top$ simply indicates that law/goal side-condition constraints are fine (here both are trivial).

```
3 : "eqv_symm" [≡[3,4]]
    (true ≡ Q) ≡ Q
    ⊤ ⟹ ⊤
2 : "eqv_symm" [≡[1,2]]
    Q ≡ (true ≡ Q)
    ⊤ ⟹ ⊤
1 : "eqv_assoc" [≡lhs]
    true ≡ (Q ≡ Q)
    ⊤ ⟹ ⊤
```

Applying match 1 results in the following change:

```
true ≡ (Q ≡ Q)
 ⊤
Focus = [] :: 𝔹
```

Note that there is a transcript of the proof being collected but we omit that here for clarity. The third line is now being shown, because it says that the prover "focus" is an empty list, and this focus has type $\mathbb{B}$ meaning it is a predicate. Because equational reasoning relies on Liebniz's Law on substitution of equals for equals, we can now focus in on the $(Q \equiv Q)$ part. Notice that the focus sequence now [2] which means we are visiting the 2nd component, viewed left-to-right, and that the colouring denotes the scope of the current focus:

```
true ≡ (Q ≡ Q)
 ⊤
Focus = [2] :: 𝔹
```

We now match again:

```
2 : "eqv_symm" [≡[3,4]]
    Q ≡ Q
    ⊤ ⟹ ⊤
1 : "eqv_refl" [*]
    true
    ⊤ ⟹ ⊤
```

If we apply match 1 here and continue, we eventually complete the proof. This then gives us a proof transcript:

```
Proof for eqv_id
        (true ≡ Q) ≡ Q
        ⊤
by red-All
(true ≡ Q) ≡ Q
   = 'match-lhs ≡_assoc@[]'
true ≡ (Q ≡ Q)
   = 'match-all ≡_refl@[2]'
true ≡ true
   = 'match-all ≡_refl@[]'

Proof Complete
```

Here we can see that the first step matched the lefthand side of the associativity law, while the remaining two steps matches the reflexivity laws in their entirety. Just as is done in (Gries & Schneider 1994), we can exploit a wide range of ways of matching against equivalences. What is shown here is a simple proof, but this approach scales well over the more complicated theories.

One last useful feature in `reasonEq` worth mentioning, is the ability to "assume" conjectures are true, either individually or at theory level. The resulting laws can be used in proofs but their provenance is noted as being assumed. This is very useful when working on a theory that depends on many others.

3.4. Current Status

All the current conjectures in theories from equivalence upto closure have been proven and so can now be safely assumed. The recent addition of the Lists theory has brought up the need for inductions. This was supported in the previous theorem prover tool, and so that just needs to be carried over.

We also intend to add in the Reactive and CSP theories.

3.4.1. Current Focus The logic theories (equivalence ...closure) of `reasonEq` are mature and working fine. The move to working with real UTP theories such as designs and UTCP, and the LTL theory, is smoking out bugs in the prover as new bits are getting exercised.

3.4.2. Current Limitations At the time of writing all theories are hardwired as Haskell code and the only way to add a new theory is to edit `reasonEq` sources to add a new Haskell module that implements that theory. From within a theory, it is possible to add new conjectures that can be saved, but the support for managing those is currently poor.

4. Research Drivers

There are two key drivers of our current research:

- The Views paper
- Using Promela/SPIN to do Test Generation for RTEMS.

4.1. Views

The Views paper (Dinsdale-Young et al. 2013) described an algebraic approach to handling a wide variety of concurrency theories, from Owicki-Gries, to Rely-Guarantee, Concurrent Separation Logic, and others. It started with a simple algebra that is essentially CKA, and then showed how all these diverse approaches could be instantiated on that algebra. Our research agenda is to implement Views in UTP so that we might be able to reason about concurrent programs in a compositional way, by mixing and matching different techniques. This would exploit suitable linking predicates and Galois theory connections.

The View Framework is designed to capture the essence of sound compostional reasoning, with a generic program logic, using the the concept of views with a composition operator $(*)$. As per UTCP, they have atomic commands a and the following generic Command language:

$$C ::= a \mid skip \mid C;C \mid C+C \mid C \| C \mid C^*$$

Given machine states $s : \mathcal{S}$, they define an interpetation so that $[\![a]\!](s)$ is the set of states possible after running a in state s. An identity action id is provided such that $[\![\mathsf{id}]\!](s) = s$. They define an operational semantics where atomic actions transition between commands:

$$\frac{}{a \xrightarrow{a} skip} \qquad \frac{}{skip; C \xrightarrow{\mathsf{id}} C} \qquad \frac{C_1 \xrightarrow{a} C_1'}{C_1; C_2 \xrightarrow{a} C_1'; C_2}$$

$$\frac{}{C^* \xrightarrow{\mathsf{id}} skip} \qquad \frac{}{C^* \xrightarrow{\mathsf{id}} C; C^*} \qquad \frac{i \in \{1,2\}}{C_1 + C_2 \xrightarrow{\mathsf{id}} C_i}$$

$$\frac{}{skip \parallel C \xrightarrow{\mathsf{id}} C} \qquad \frac{C_1 \xrightarrow{a} C_1'}{C_1 \parallel C_2 \xrightarrow{a} C_1' \parallel C_2} \qquad \frac{C_2 \xrightarrow{a} C_2'}{C_1 \parallel C_2 \xrightarrow{a} C_1 \parallel C_2'}$$

A View is then defined as a commutative semigroup $(\mathsf{View}, *)$, which in some cases can be extended to a view monoid $(\mathsf{View}, *, u)$. In their own words:

> "Intuitively, views are resources that embody knowledge about the state and the protocol threads must obey, and rights to modify the state in accordance with the implied protocol. The semigroup operation $*$ (view composition) combines the knowledge and rights of two views."

This is then used to define a program logic based on axioms of the form (p, a, q) where p and q are views, and a logic judgement of the form $\vdash \{p\}C\{q\}$, where p and q are views that define the precondition and postcondition respectively. Proof rules are then provided, which are the standard rules of disjoint concurrent separation logic, and have a partial correctness interpretation. The paper goes on to show how to link the program logic to the operational semantics, and is interspersed with many examples from many concurrency theories.

4.2. RTEMS Test Generation

4.2.1. Background

We were involved in a project funded by the European Space Agency that used Promela/Spin to model various parts of the RTEMS operating system, and then automatically derived C test code. This is still ongoing work. We constructed a way to get useful information out of Spin in a tailored observation language, using YAML dictionaries to translate this to C code. Our research agenda is to use this approach as a real-world example to drive UTP development (Butterfield & Tuong 2023a). The plan is to use UTP to give semantics to Promela, the observations, and the C subset used for RTEMS testing, and to derive a refinement linking predicate from the YAML file. This would be used to prove that the tests do faithfully represent the Promela model.

RTEMS code verification is mainly based on testsuites that target the APIs exposed to developers using it for their real-time projects, as documented for example, in the Classic API Guide (https://docs.rtems.org/docs/main/c-user/).

Versions of RTEMS have been qualified for use in ESA space missions to criticality level B. The first was done for single-core processors, while a more recent activity looked at multicore, and has produced qualification artifacts and tools that are publicly available (https://rtems-qual.io.esa.int/).

We were involved in applying formal methods techniques to contribute to the verification effort. The code we delivered can be found at https://gitlab.rtems.org/rtems/prequal/rtems-central in the `formal` subdirectory. Documentation for this can be found in the form of a new Formal Verification chapter in the RTEMS Software Engineering manual (https://docs.rtems.org/docs/main/eng/, Chapter 9). Ongoing work in this space is continuing, and it available from https://github.com/andrewbutterfield/RTEMS-SMP-Formal.

4.2.2. What we did

Most of the RTEMS API calls are grouped in what are termed *Managers*, each of which provides a well define OS service. We focussed on Managers that involved concurrency, such as those handling Semaphores, Events, Messaging, Barriers, and Tasks. Our initial work started with the Events Manager because it was relatively simple (only two API calls: send and receive), while still exposing complex aspects, such as timeouts and interactions between blocked tasks and priorities. So for example, the Event Send API call has the following usage:

```
rc = rtems_event_send(id,events)
```

What we want to do is cover all the relevant combinations of `id` and `events` to cover all error cases and expected behaviours. For each case this means producing the relevant call, checking the expected return code (`rc`), and any other relevant changes.

We built a Promela model of the send and receive API calls, along with models of the relevant parts of the operating system, most notably the scheduler, and timers. For Event Send, we have the following model:

```
inline event_send(self,tid,evts,rc) {
atomic{
    if
    ::  tid >= BAD_ID -> rc = RC_InvId
    ::  tid < BAD_ID ->
        tasks[tid].pending = tasks[tid].pending | evts
        unsigned got : NO_OF_EVENTS;
        bool sat;
        satisfied(tasks[tid],got,sat);
        if
        ::  sat ->
            tasks[tid].state = Ready;
            preemptIfRequired(self,tid) ;
            waitUntilReady(self);
        ::  else -> skip
        fi
        rc = RC_OK;
    fi }}
```

We used a standard technique to get the Spin model checker to generate all possible correct behaviours, by negating the properties already shown as true on the model and asking Spin to find all counterexamples.

The Spin output for these counterexamples is very hard to parse and process — it shows executed model line-numbers and variable values, and is intended to be read and iterpreted by the model author. To allow automatic processing, we developed a simple Observation language that we could use to log significant events. Promela has a `printf` statement we could use for this. For example, `sendrc = event_send(proc2,10)` would result in the following Observble output:

```
@@@ 3 CALL event_send 1 2 10 sendrc
@@@ 3 SCALAR sendrc 0
```

The next step was to define a refinement from the Promela model (via its manfestation as Observations) to the corresponding RTEMS C test code. The RTEMS build system is written in Python, and it make sense for us to use that, so we chose to use Python dictionaries (YAML files) to capture this. The Observables example above would require the following dictionary entries:

```
event_send : |
{3} = rtems_event_send ( {1}, {2} );
sendrc :
T_rsc ( sendrc , {0} );
```

which woud result in the following snippet of RTEMS test code:

```
sendrc = rtems_event_send ( 2, 10 );
T_rsc ( sendrc , 0 );
```

The resulting tests were deployed in the same manner as other RTEMS tests are. And indeed, we were able to get tests running successfully for all of the Managers we modelled.

However, there is one potential issue. How do we know that what was tested on RTEMS, is actually what was modelled in Promela? This suggests an interest case study for UTP: can we provide UPT semantics to Promela, the Observation language, the refinement written in YAML, and the C test code, and link them together to show it is all saying the same thing?

4.3. Connecting Views and RTEMS

While the snippets above show sequential code, it all lives in a scenario which has several RTEMS Tasks running concurrently. These tests need to be deterministic (repeatable), so the apparent concurrency is eliminated by using special simple binary semaphores to effectively "schedule" the thread. These can be modelled in explicitly in Promela, but their use can also be inferred in the way he Observation launguage is translated using the dictionary. So correctness here is very much a concurrency issue, and we see UTCP (inspired by Views) as being a baseline on top of which we can build the various semantics, define the refinement, and reason about their correctness. Recent material that goes into more detail can be found in (Butterfield & Tuong 2023b)

5. Conclusions

5.1. Summary

We started by introducing UTP and giving a general description of how it works. We did this by showing both a simple imperative While language and the process algebra CSP can be modelled in UTP, and talking about how those can then be blended as done for *Circus*. We noted how distinct (not blended) theories can be formally linked using Galois connections.

We then presented the `reasonEq` theorem prover we are implementing> We explained our motivation for yet another theorem prover along with some history of its development. We then explained the concept of a "theory" in `reasonEq`, and how a proof works from the users perspective.

We then looked a key drivers of our research: these being the Views framework, and the system we developed for using formal Promela models to generate RTEMS C test code. The Views Framework shows that a lot of concurrent verification techniques have a common pattern, whihc can associated with an operational semantics, and a program logic. Our work on UTCP is inspired by this, adm we see it as the corresponding "glue" we can use to unifying the semantics of Promela, Observables and (some) C.

5.2. Future Work

The plan is to seek to give a complete formal semantics for the formal RTEMS test generation process, using it as a driver to develop and improve the `reasonEq` theorem prover. Exploring how we can exploit Isabelle/UTP to check proofs is a key part of this.

Acknowledgments

We would like to thank the reviewers for their helpful comments and suggestions.

References

Bertot, Y., & Castéran, P. P. (2004). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Verlag.

Bowen, J. P., & Zhu, H. (Eds.). (2017). *Unifying theories of programming - 6th international symposium, UTP 2016, reykjavik, iceland, june 4-5, 2016, revised selected papers* (Vol. 10134). Springer. Retrieved from https://doi.org/10.1007/978-3-319-52228-9 doi: 10.1007/978-3-319-52228-9

Bresciani, R., & Butterfield, A. (2012a). A probabilistic theory of designs based on distributions. In B. Wolff, M. Gaudel, & A. Feliachi (Eds.), *Unifying theories of programming, 4th international symposium, UTP 2012, paris, france, august 27-28, 2012, revised selected papers* (Vol. 7681, pp. 105–123). Springer. Retrieved from https://doi.org/10.1007/978-3-642-35705-3_5 doi: 10.1007/978-3-642-35705-3_5

Bresciani, R., & Butterfield, A. (2012b). A UTP semantics of pgcl as a homogeneous relation. In J. Derrick, S. Gnesi, D. Latella, & H. Treharne (Eds.), *Integrated formal methods - 9th international conference, IFM 2012, pisa, italy, june 18-21, 2012. proceedings* (Vol. 7321, pp. 191–205). Springer. Retrieved from https://doi.org/10.1007/978-3-642-30729-4_14 doi: 10.1007/978-3-642-30729-4_14

Bresciani, R., & Butterfield, A. (2014). A UTP approach towards probabilistic protocol verification. *Secur. Commun. Networks*, *7*(1), 99–107. Retrieved from https://doi.org/10.1002/sec.546 doi: 10.1002/SEC.546

Butterfield, A. (2007). A denotational semantics for handel-c. In C. B. Jones, Z. Liu, & J. Woodcock (Eds.), *Formal methods and hybrid real-time systems, essays in honor of dines bjørner and chaochen zhou on the occasion of their 70th birthdays, papers presented at a symposium held in macao, china, september 24-25, 2007* (Vol. 4700, pp. 45–66). Springer. Retrieved from https://doi.org/10.1007/978-3-540-75221-9_3 doi: 10.1007/978-3-540-75221-9_3

Butterfield, A. (2010a). Saoithín: A theorem prover for UTP. In *Unifying theories of programming - third international symposium, UTP 2010, shanghai, china, november 15-16,* *2010. proceedings* (pp. 137–156). Retrieved from http://dx.doi.org/10.1007/978-3-642-16690-7_6 doi: 10.1007/978-3-642-16690-7_6

Butterfield, A. (Ed.). (2010b). *Unifying theories of programming, second international symposium, UTP 2008, dublin, ireland, september 8-10, 2008, revised selected papers* (Vol. 5713). Springer. Retrieved from https://doi.org/10.1007/978-3-642-14521-6 doi: 10.1007/978-3-642-14521-6

Butterfield, A. (2012). The logic of $U \cdot (TP)^2$. In *Unifying theories of programming, 4th international symposium, UTP 2012, paris, france, august 27-28, 2012, revised selected papers* (pp. 124–143). Retrieved from http://dx.doi.org/10.1007/978-3-642-35705-3_6 doi: 10.1007/978-3-642-35705-3_6

Butterfield, A. (2014). UTP2: higher-order equational reasoning by pointing. In C. Benzmüller & B. W. Paleo (Eds.), *Proceedings eleventh workshop on user interfaces for theorem provers, UITP 2014, vienna, austria, 17th july 2014* (Vol. 167, pp. 14–22). Retrieved from https://doi.org/10.4204/EPTCS.167.4 doi: 10.4204/EPTCS.167.4

Butterfield, A. (2016). UTPCalc - A Calculator for UTP Predicates. In J. P. Bowen & H. Zhu (Eds.), *Unifying theories of programming - 6th international symposium, UTP 2016, reykjavik, iceland, june 4-5, 2016, revised selected papers* (Vol. 10134, pp. 197–216). Springer. Retrieved from https://doi.org/10.1007/978-3-319-52228-9_10 doi: 10.1007/978-3-319-52228-9_10

Butterfield, A. (2017). UTCP: compositional semantics for shared-variable concurrency. In S. A. da Costa Cavalheiro & J. L. Fiadeiro (Eds.), *Formal methods: Foundations and applications - 20th brazilian symposium, SBMF 2017, recife, brazil, november 29 - december 1, 2017, proceedings* (Vol. 10623, pp. 253–270). Springer. Retrieved from https://doi.org/10.1007/978-3-319-70848-5_16 doi: 10.1007/978-3-319-70848-5_16

Butterfield, A. (2024). Towards an algebra for unifying theories of concurrent programming (utcp). In S. Foster & A. Sampaio (Eds.), *The application of formal methods: Essays dedicated to jim woodcock on the occasion of his retirement* (pp. 203–232). Cham: Springer Nature Switzerland. Retrieved from https://doi.org/10.1007/978-3-031-67114-2_9 doi: 10.1007/978-3-031-67114-2_9

Butterfield, A., Gancarski, P., & Woodcock, J. (2009). State visibility and communication in unifying theories of programming. In W. Chin & S. Qin (Eds.), *TASE 2009, third IEEE international symposium on theoretical aspects of software engineering, 29-31 july 2009, tianjin, china* (pp. 47–54). IEEE Computer Society. Retrieved from https://doi.org/10.1109/TASE.2009.57 doi: 10.1109/TASE.2009.57

Butterfield, A., Mjeda, A., & Noll, J. (2016). UTP semantics for shared-state, concurrent, context-sensitive process models. In *10th international symposium on theoretical aspects of software engineering, TASE 2016, shanghai, china, july 17-19, 2016* (pp. 93–100). IEEE Computer Society. Retrieved from https://doi.org/10.1109/TASE.2016.22 doi: 10.1109/TASE.2016.22

Butterfield, A., Sherif, A., & Woodcock, J. (2007). Slotted-circus. In J. Davies & J. Gibbons (Eds.), *Integrated formal methods, 6th international conference, IFM 2007, oxford, uk, july 2-5, 2007, proceedings* (Vol. 4591, pp. 75–97). Springer. Retrieved from https://doi.org/10.1007/978-3-540-73210-5_5 doi: 10.1007/978-3-540-73210-5_5

Butterfield, A., & Tuong, F. (2023a). Applying formal verification to an open-source real-time operating system. In J. P. Bowen, Q. Li, & Q. Xu (Eds.), *Theories of programming and formal methods: Essays dedicated to jifeng he on the occasion of his 80th birthday* (pp. 348–366). Cham: Springer Nature Switzerland. Retrieved from https://doi.org/10.1007/978-3-031-40436-8_13 doi: 10.1007/978-3-031-40436-8_13

Butterfield, A., & Tuong, F. (2023b). Applying formal verification to an open-source real-time operating system. In J. P. Bowen, Q. Li, & Q. Xu (Eds.), *Theories of programming and formal methods - essays dedicated to jifeng he on the occasion of his 80th birthday* (Vol. 14080, pp. 348–366). Springer. Retrieved from https://doi.org/10.1007/978-3-031-40436-8_13 doi: 10.1007/978-3-031-40436-8_13

Butterfield, A., & Woodcock, J. (2002). Semantic domains for handel-c. In S. Flynn et al. (Eds.), *Second irish conference on the mathematical foundations of computer science and information technology, MFCSIT 2002, galway, ireland, july 18-19, 2002* (Vol. 74, pp. 1–20). Elsevier. Retrieved from https://doi.org/10.1016/S1571-0661(04)80762-X doi: 10.1016/S1571-0661(04)80762-X

Butterfield, A., & Woodcock, J. (2003). An operational semantics for handel-c. In T. Arts & W. J. Fokkink (Eds.), *Eighth international workshop on formal methods for industrial critical systems, FMICS 2003, roros, norway, june 5-7, 2003* (Vol. 80, pp. 235–250). Elsevier. Retrieved from https://doi.org/10.1016/S1571-0661(04)80821-1 doi: 10.1016/S1571-0661(04)80821-1

Butterfield, A., & Woodcock, J. (2004). A "hardware compiler" semantics for handel-c. In A. K. Seda, T. Hurley, M. P. Schellekens, M. M. an Airchinnigh, & G. Strong (Eds.), *Proceedings of the third irish conference on the mathematical foundations of computer science and information technology, MFCSIT 2004, dublin, ireland, july 22-23, 2004* (Vol. 161, pp. 73–90). Elsevier. Retrieved from https://doi.org/10.1016/j.entcs.2006.04.026 doi: 10.1016/J.ENTCS.2006.04.026

Butterfield, A., & Woodcock, J. (2005). prialt in handel-c: an operational semantics. *Int. J. Softw. Tools Technol. Transf.*, *7*(3), 248–267. Retrieved from https://doi.org/10.1007/s10009-004-0181-6 doi: 10.1007/S10009-004-0181-6

Calcagno, C., O'Hearn, P. W., & Yang, H. (2007). Local action and abstract separation logic. In (pp. 366–378). IEEE Computer Society. Retrieved from http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4276538

Cavalcanti, A., & Woodcock, J. (2004). A tutorial introduction to CSP in *Unifying Theories of Programming*. In A. Cavalcanti, A. Sampaio, & J. Woodcock (Eds.), *Refinement techniques in software engineering, first pernambuco summer school on software engineering, PSSE 2004, recife, brazil, november 23-december 5, 2004, revised lectures* (Vol. 3167, pp. 220–268). Springer. Retrieved from https://doi.org/

10.1007/11889229_6 doi: 10.1007/11889229_6

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., & Yang, H. (2013). Views: compositional reasoning for concurrent programs. In R. Giacobazzi & R. Cousot (Eds.), *The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, rome, italy - january 23 - 25, 2013* (pp. 287–300). ACM.

Dunne, S., & Stoddart, B. (Eds.). (2006). *Unifying theories of programming, first international symposium, UTP 2006, walworth castle, county durham, uk, february 5-7, 2006, revised selected papers* (Vol. 4010). Springer. Retrieved from https://doi.org/10.1007/11768173 doi: 10.1007/11768173

Foster, S., Zeyda, F., & Woodcock, J. (2014). Isabelle/UTP: A mechanised theory engineering framework. In D. Naumann (Ed.), *Unifying theories of programming - 5th international symposium, UTP 2014, singapore, may 13, 2014, revised selected papers* (Vol. 8963, pp. 21–41). Springer. Retrieved from http://dx.doi.org/10.1007/978-3-319-14806-9_2 doi: 10.1007/978-3-319-14806-9_2

Gancarski, P., & Butterfield, A. (2009). The denotational semantics of slotted-circus. In A. Cavalcanti & D. Dams (Eds.), *FM 2009: Formal methods, second world congress, eindhoven, the netherlands, november 2-6, 2009. proceedings* (Vol. 5850, pp. 451–466). Springer. Retrieved from https://doi.org/10.1007/978-3-642-05089-3_29 doi: 10.1007/978-3-642-05089-3_29

Gancarski, P., & Butterfield, A. (2010). Prioritized slotted-circus. In A. Cavalcanti, D. Déharbe, M. Gaudel, & J. Woodcock (Eds.), *Theoretical aspects of computing - ICTAC 2010, 7th international colloquium, natal, rio grande do norte, brazil, september 1-3, 2010. proceedings* (Vol. 6255, pp. 91–105). Springer. Retrieved from https://doi.org/10.1007/978-3-642-14808-8_7 doi: 10.1007/978-3-642-14808-8_7

George, C., Haxthausen, A. E., Hughes, S., Milne, R., Prehn, S., & Pedersen, J. S. (1995). *The RAISE development method.* Prentice Hall Int.

Gries, D., & Schneider, F. B. (1994). *A logical approach to discrete math.* New York, NY: Springer-Verlag.

Hehner, E. C. R. (1984, February). Predicative programming part i & ii. *Commun. ACM*, *27*(2), 134–151.

Hoare, C. A. R. (1985). *Communicating sequential processes.* Prentice-Hall.

Hoare, C. A. R., & He, J. (1998a). *Unifying theories of programming.* Englewood Cliffs, NJ: Prentice-Hall International.

Hoare, C. A. R., & He, J. (1998b). *Unifying theories of programming.* Prentice-Hall.

Hoare, C. A. R., Möller, B., Struth, G., & Wehrman, I. (2009). Concurrent kleene algebra. In M. Bravetti & G. Zavattaro (Eds.), *CONCUR 2009 - concurrency theory, 20th international conference, CONCUR 2009, bologna, italy, september 1-4, 2009. proceedings* (Vol. 5710, pp. 399–414). Springer. Retrieved from https://doi.org/10.1007/978-3-642-04081-8_27 doi: 10.1007/978-3-642-04081-8_27

Jones, C. B. (1983). Tentative steps toward a development method for interfering programs. *TOPLAS*, *5*(4), 596–619.

Naumann, D. (Ed.). (2015). *Unifying theories of programming - 5th international symposium, UTP 2014, singapore, may 13, 2014, revised selected papers* (Vol. 8963). Springer. Retrieved from http://dx.doi.org/10.1007/978-3-319-14806-9 doi: 10.1007/978-3-319-14806-9

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL - A proof assistant for higher-order logic* (Vol. 2283). Springer. Retrieved from http://link.springer.de/link/service/series/0558/tocs/t2283.htm

Oliveira, M., Cavalcanti, A., & Woodcock, J. (2009). A UTP semantics for circus. *Formal Asp. Comput*, *21*(1-2), 3–32. Retrieved from http://dx.doi.org/10.1007/s00165-007-0052-5

Oliveira, M. V. M. (2005). *Formal derivation of state-rich reactive programs using Circus* (Unpublished doctoral dissertation). University of York.

Owicki, S., & Gries, D. (1976). An axiomatic proof technique for parallel programs. *Acta Informatica*, *6*(4), 319–340.

Owre, S., Rushby, J. M., & Shankar, N. (1992). PVS: A prototype verification system. In D. Kapur (Ed.), *Automated deduction - cade-11, 11th international conference on automated deduction, saratoga springs, ny, usa, june 15-18, 1992, proceedings* (Vol. 607, pp. 748–752). Springer. Retrieved from https://doi.org/10.1007/3-540-55602-8_217 doi: 10.1007/3-540-55602-8_217

Qin, S. (Ed.). (2010). *Unifying theories of programming - third international symposium, UTP 2010, shanghai, china, november 15-16, 2010. proceedings* (Vol. 6445). Springer. Retrieved from http://dx.doi.org/10.1007/978-3-642-16690-7 doi: 10.1007/978-3-642-16690-7

Ribeiro, P., & Sampaio, A. (Eds.). (2019). *Unifying theories of programming - 7th international symposium, UTP 2019, dedicated to tony hoare on the occasion of his 85th birthday, porto, portugal, october 8, 2019, proceedings* (Vol. 11885). Springer. Retrieved from https://doi.org/10.1007/978-3-030-31038-7 doi: 10.1007/978-3-030-31038-7

Sherif, A., & He, J. (2002). Towards a time model for circus. In C. George & H. Miao (Eds.), *Formal methods and software engineering, 4th international conference on formal engineering methods, ICFEM 2002 shanghai, china, october 21-25, 2002, proceedings* (Vol. 2495, pp. 613–624). Springer. Retrieved from https://doi.org/10.1007/3-540-36103-0_62 doi: 10.1007/3-540-36103-0_62

Tourlakis, G. (2001). On the soundness and completeness of equational predicate logics. *J. Log. Comput.*, *11*(4), 623-653.

Warford, J. S., Vega, D., & Staley, S. M. (2021). A calculational deductive system for linear temporal logic. *ACM Comput. Surv.*, *53*(3), 53:1–53:38. Retrieved from https://doi.org/10.1145/3387109 doi: 10.1145/3387109

Wolff, B., Gaudel, M., & Feliachi, A. (Eds.). (2013). *Unifying theories of programming, 4th international symposium, UTP 2012, paris, france, august 27-28, 2012, revised selected papers* (Vol. 7681). Springer. Retrieved from http://dx.doi.org/10.1007/978-3-642-35705-3 doi: 10.1007/978-3-642-35705-3

Woodcock, J., & Cavalcanti, A. (2001). A concurrent language for refinement. In A. Butterfield, G. Strong, & C. Pahl (Eds.), *5th irish workshop on formal methods, IWFM 2001, dublin, ireland, 16-17 july 2001.* BCS. Retrieved from http://ewic

.bcs.org/content/ConWebDoc/4146

Woodcock, J., & Hughes, A. P. (2002). Unifying theories of parallel programming. In C. George & H. Miao (Eds.), *Formal methods and software engineering, 4th international conference on formal engineering methods, ICFEM 2002 shanghai, china, october 21-25, 2002, proceedings* (Vol. 2495, pp. 24–37). Springer. Retrieved from http://dx.doi.org/10.1007/3-540-36103-0_5 doi: 10.1007/3-540-36103-0_5

Woodcock, J. C. P., & Davies, J. (1996). *Using Z - specification, refinement, and proof.* Prentice Hall.

Zhu, H., Sanders, J. W., He, J., & Qin, S. (2012). Denotational semantics for a probabilistic timed shared-variable language. In B. Wolff, M. Gaudel, & A. Feliachi (Eds.), *Unifying theories of programming, 4th international symposium, UTP 2012, paris, france, august 27-28, 2012, revised selected papers* (Vol. 7681, pp. 224–247). Springer. Retrieved from http://dx.doi.org/10.1007/978-3-642-35705-3_11 doi: 10.1007/978-3-642-35705-3_11

About the author

Andrew Butterfield is a professor at the School of Computer Science and Statistics in Trinity College Dublin. He is head of the Software Foundations & Verification Group and is interested in using Formal Methods for the verification of critical systems. You can contact the author at butrfeld@tcd.ie.