UNIVERSITY of York

This is a repository copy of *Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/110313/

Version: Accepted Version

Proceedings Paper:
Foster, Simon David orcid.org/0000-0002-9889-9514 and Woodcock, JAMES Charles Paul orcid.org/0000-0001-7955-2702 (2017) Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL. In: Concurrency, Security, and Puzzles:Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, pp. 39-64.

https://doi.org/10.1007/978-3-319-51046-0_3

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL

Simon Foster and Jim Woodcock
{simon.foster,jim.woodcock}@york.ac.uk

University of York, Department of Computer Science, York, YO10 5GH, UK.

Abstract. In this paper, we outline our vision for building verification tools for Cyber-Physical Systems based on Hoare and He's Unifying Theories of Programming (UTP) and interactive proof technology in Isabelle/HOL. We describe our mechanisation and explain some of the design decisions that we have taken to get a convenient and smooth implementation. In particular, we describe our use of lenses to encode state. We illustrate our work with an example UTP theory and describe the implementation of three foundational theories: designs, reactive processes, and the hybrid relational calculus. We conclude by reflecting on how tools are linked by unifying theories.

This paper is dedicated to Bill Roscoe on the occasion of his 60th birthday.

1 Introduction

Cyber-Physical Systems (CPS) are networks of computational devices that interact with the world through an array of sensors and actuators, and combine discrete computation with continuous physical models of their environment. For example, automated, driverless vehicles that are required to sense their environment, construct a real-time model of the present situation, make decisions about developing scenarios, and respond within a sufficiently short amount of time to ensure the safety of its passengers and other road users. Engineering such systems whilst demonstrating their trustworthiness is a major challenge. CPS engineering involves a wide range of modelling and programming paradigms [10], including concurrency, real-time, mobility, continuous variables, differential equations, object orientation, and diagrammatic languages. These aspects are represented by a variety of domain-specific and general-purpose languages, such as Simulink, Modelica, SysML, Java, and C, and thus engineering trustworthy CPS requires that we semantically integrate models in a consistent way, and then form arguments that the system as a whole exhibits certain properties.

Semantic integration has been made possible using the industry-developed standard FMI [5] (Functional Mockup Interface), which describes a CPS using a network of FMUs (Functional Mockup Units) that represent components or constituent systems. An FMU exposes a number of observable continuous variables

that characterise the state of the individual model at a particular instant. Variables can either be of type input, output, or state, depending on whether they are under the control of the FMU or the environment. FMUs can be stepped forward in time, which will cause these variables to evolve. A requested time step may be rejected and require curtailing if an event, such as a zero-crossing, occurs in the meantime, since the other FMUs may need to be notified. A master algorithm manages stepping the individual FMUs forward, and distributing information in between time steps. Aside from this minimal operational interface, each FMU is treated as a black box. An FMU can correspond to an abstract model of behaviour, an implementation of a particular component, or even a physical piece of hardware, which allows for Hardware in the Loop (HiL) simulation and testing. FMI thus allows one to describe heterogeneous multi-models that are described in different notations, and with different underlying semantic models, but are nevertheless integrated through a common operational interface.

Though FMI provides the necessary operational interface between different models and programs, it alone does not provide enough semantic information to verify them. In order to achieve that, we need a way of tackling the inherent semantic heterogeneity of the multi-model, for which we use Hoare and He's *Unifying Theories of Programming* [24,39,8] (UTP), which is a a long-term research agenda to describe different computing paradigms and the formal links between them. It allows us to consider the various semantic aspects of a heterogeneous multi-model as individual theories that characterise a particular abstract programming or modelling paradigm. Hoare & He [24] show how the alphabetised relational calculus can be applied to construct a hierarchy of such theories, including simple imperative programs (relations), designs that correspond to pre- and postcondition specifications, and various theories of concurrent and parallel programs, including the process algebras ACP, CCS, and CSP [23]. Since the advent of UTP, a host of additional UTP theories have been developed that variously tackle paradigms like real-time programming [34], object-oriented programming [32], security and confidentiality [3], mobile processes [33], probabilistic modelling [6], and hybrid systems [15]. Moreover, the FMI API itself has been given a UTP-based semantics [9] that can be used as an interface to the semantic model of individual FMUs, and also allows a network of FMUs to be verified at this level using the FDR3 refinement checker [18]. The UTP approach allows computational theories to be formalised and explored as independent theories, and then later integrated to provide heterogeneous denotational semantic models. This can either be done directly through theory combination, or where theories are not directly compatible, such as in the case of discrete and continuous time, through the use of Galois connections that characterise best approximations.

In order to make UTP theories practically applicable to program verification, tool support is needed, and so we are also developing a theorem prover for UTP based on Isabelle/HOL [28], which we call *Isabelle/UTP* [17,16]. Isabelle is a powerful proof assistant that can be used both for the mechanisation of mathematics, and for the application of such mechanisations to program

verification, which is famously illustrated by the seL4 microkernel verification project [26]. Another excellent example is the use of Kleene algebras to build program verification tools [1], from which Hoare logics, weakest-precondition calculi, rely-guarantee calculi, and separation logics have been created. Specifically of interest for CPS, there has also been a lot of recent work on formalising calculus, analysis, and ordinary differential equations (ODEs) in Isabelle [25], which can then be applied to verification of hybrid systems. Similarly, we are also building a mechanised library of UTP theories[1], including associated laws of programming and verification calculi.

Crucial to all of these developments is the ability to integrate external tools into Isabelle that can provide decision procedures for specific classes of problems. Isabelle is well suited to such integrations due to its architecture that is based on the ML and Scala programming languages, both of which can be used to implement plugins. Isabelle is sometimes referred to as the *Eclipse* of theorem provers [37]. The *sledgehammer* tool [4], for example, integrates a host of first-order automated theorem provers and SMT solvers, which often shoulder the burden of proof effort. *Sledgehammer* is used, for example, by [1], both at the theory engineering level, for constructing an algebraic hierarchy of Kleene algebras, and also at the verification level, where it is used to discharge first-order proof obligations. For verification of hybrid systems, it will also be necessary to integrate Isabelle with Computer Algebra Systems (CAS) like Mathematica, MATLAB, or SageMath, to provide solutions to differential equations, an approach that has been previously well used by the KeYmaera tool [30,31].

Our vision is the use of Isabelle and UTP to provide the basis for CPS verification through formalisation of the fundamental building-block theories of a CPS multi-model, and the integration of tools that implement these theories for coordinated verification. This is, of course, an ambitious task and will require collaboration with a host of domain experts. Nevertheless, the vision of UTP is to provide a domain in which such cooperations can be achieved.

This paper gives an overview of the state of our work towards verification of CPS in UTP. In Section 2, we describe our approach to mechanising UTP in Isabelle/HOL, including its lens-based state model, meta-logical operators, and the alphabetised relational calculus. In Section 3, we show how an example theory can be mechanised and properties proved in *Isabelle/UTP*. In Section 4, we give an overview of the UTP theories of CPS that we have mechanised so far. In Section 5, we conclude.

2 Algebraic foundations of Isabelle/UTP

In this section we summarise the foundations of *Isabelle/UTP*, our semantic embedding of the UTP in Isabelle/HOL, including its lens-based state model, meta-logical functions, and laws. *Isabelle/UTP* includes a model of alphabetised predicates and relations, proof tactics, and a library of proven algebraic laws. Follow-

[1] This library can be viewed at `github.com/isabelle-utp/utp-main`.

ing [11,12], our predicate model is a parametric Isabelle type α upred $\widehat{=} \alpha \Rightarrow bool$ where α is the domain of possible observations, that is, the alphabet.

The predicates-as-sets model is standard for most semantic embeddings of UTP, both deep [29,16,40] and shallow [11,12], and means that the predicate calculus operators can be obtained by simple lifting of the HOL equivalents. This means that we can automatically inherit the fact that, like HOL predicates, UTP predicates also form a complete lattice. Moreover, this facilitates automated proof for UTP predicates, which we make available through the predicate calculus tactic *pred-tac*, which can be used to discharge a large number of conjectures in our UTP theories.

A major difference between *Isabelle/UTP* and the deep embeddings is that we use Isabelle types to model alphabets, rather than representing them as finite sets. Use of types to model alphabets has the advantage that the type checker can be harnessed to ensure that variables mentioned in predicates are indeed present in the alphabet. What the predicate model lacks *a priori* though, is a way of manipulating the variables present in α; for this we use lenses.

2.1 Lenses in brief

UTP is based on the paradigm of predicative programming, where programs are predicates [22]. This view results in a great simplification, with much of the machinery of traditional denotational semantics swept away, including the brackets mapping fragments of syntax to their denotation, as well as the environment used to evaluate variables in context. As an example of the latter, $x := 1$ is just another way of writing the predicate $x' = x + 1$. This simplified view of an environment-free semantics is difficult to maintain when thinking about more sophisticated programming techniques, such as aliasing between variables. See, for example a UTP semantics for separation logic [38], where environments are reintroduced to record variables stored on the heap and the stack. This raises the general methodological question of what is the most convenient way of modelling the state space for a UTP theory? The answer to this is especially important for our mechanisation in Isabelle, if we are to provide a generally reusable technique.

Rather than characterising variables as syntactic entities [16], we instead algebraically characterise the behaviour of variables using lenses [17,14]. Lenses allow us to represent variables as abstract projections on a state space with convenient ways to query and update in a uniform, compositional way. Variables are thus represented by regions of the state space that can be variously related, namelessly and spatially; these regions can be nested in arbitrary ways. Lenses are equipped with operators for transforming and decomposing the state space, enabling a purely algebraic account of state manipulations, including consistent viewing, updating, and composition. Importantly, the theory of lenses allows us to formalise meta-logical operations in the predicate calculus, such as freshness of variables and substitution of expressions for variable names.

A lens X from a view type V to a bigger source type S is a function $X : V \Longrightarrow S$ that allows us to focus on V independently of S. The signature of

a lens consists of two functions:

$$\begin{aligned}&\mathrm{get} : S \to V\\ &\mathrm{put} : S \to V \to S\end{aligned}$$

Consider as an example, a record lens. For the record

$$\langle\!|\ \mathit{forename} : \mathit{String}, \mathit{surname} : \mathit{String}, \mathit{age} : \mathit{Int}\ |\!\rangle$$

there are seven lenses (the record has three components, so there are $2^3 - 1$ ways of decomposing it). Other examples include product, function, list, and finite map lenses. A number of algebraic laws might be satisfied by a particular lens:

$$\begin{aligned}\mathrm{get}\ (\mathrm{put}\ s\ v) &= v && \text{(PutGet)}\\ \mathrm{put}\ (\mathrm{put}\ s\ v')\ v &= \mathrm{put}\ s\ v && \text{(PutPut)}\\ \mathrm{put}\ s\ (\mathrm{get}\ s) &= s && \text{(GetPut)}\end{aligned}$$

Lenses that satisfy combinations of these laws are classified in different ways [14,17]:

Well-behaved lenses	PutGet + GetPut
Very well-behaved lenses	addition of PutPut
Mainly well-behaved lenses	PutGet + PutPut

The majority of laws in *Isabelle/UTP* require variables to be modelled as mainly well-behaved lenses of type $\tau \Longrightarrow \alpha$, where τ is the variable type, though some laws depend on them being very well-behaved. From these axiomatic bases we define operations for querying and composing lenses. These include independence ($X \bowtie Y$), sublens ($X \subseteq_\iota Y$), equivalence ($X \approx_\iota Y$), composition ($X ;_\iota Y$), and summation ($X +_\iota Y$). All of these operations can be given denotations in terms of the get and put [17]; here we focus on the intuition and algebraic laws.

Independence, $X \bowtie Y$, describes when lenses $X : V_1 \Longrightarrow S$ and $Y : V_2 \Longrightarrow S$ identify disjoint regions of the common source S. Essentially, this is defined requiring that their put functions commute. In our example, the *forename* and *surname* lenses can be updated independently and thus $\mathit{forename} \bowtie \mathit{surname}$. Lens independence is thus useful to describe when two variables are different. The sublens partial order, $X \subseteq_\iota Y$, conversely, describes the situation when X is spatially within Y, and thus an update to Y must affect X. From this partial order we can also define an equivalence relation on lenses in the usual way:

$$X \approx_\iota Y \ \mathrel{\widehat{=}}\ X \subseteq_\iota Y \wedge Y \subseteq_\iota X$$

Lens composition $X ;_\iota Y : V_1 \Longrightarrow S$, for $X : V_1 \Longrightarrow V_2$ and $Y : V_2 \Longrightarrow S$, allows one to focus on regions within larger regions, and thus allows for state space nesting. For example, if a record has a field that is itself a record, then lens composition allows one to focus on the inner fields by composing the lenses for the outer with those of the inner record. Lens composition is closed under all the algebraic lens classes. We also define the unit lens, $\mathbf{0}_\iota : \mathit{unit} \Longrightarrow S$, which has an empty view, and the identity lens, $\mathbf{1}_\iota : S \Longrightarrow S$, whose view

is the whole source. Both of these lenses are also very well-behaved. Lens sum, $X +_\iota Y : V_1 \times V_2 \Longrightarrow S$, parallel composes two independent lenses $X : V_1 \Longrightarrow S$ and $Y : V_2 \Longrightarrow S$. This combined lens characterises the regions of both X and Y. For example, the lens $\mathit{forename} +_\iota \mathit{age}$ allows us to query and updates both fields simultaneously, whilst leaving *surname* alone. Finally, the associated lenses $\mathrm{fst}_\iota : V_1 \Longrightarrow V_1 \times V_2$ and $\mathrm{snd}_\iota : V_2 \Longrightarrow V_1 \times V_2$ allow us to view the left and right elements of a product source-type.

Our lenses operations satisfy the following algebraic laws, all of which has been mechanised [17], assuming X, Y, and Z are well-behaved lenses:

Theorem 1. *Lens algebraic laws*

$$X ;_\iota (Y ;_\iota Z) = (X ;_\iota Y) ;_\iota Z \tag{L1}$$

$$X ;_\iota \mathbf{1}_\iota = \mathbf{1}_\iota ;_\iota X = X \tag{L2}$$

$$X \bowtie Y \Leftrightarrow Y \bowtie X \tag{L3}$$

$$X +_\iota (Y +_\iota Z) \approx_\iota (X +_\iota Y) +_\iota Z \qquad X \bowtie Y, X \bowtie Z, Y \bowtie Z \tag{L4}$$

$$X +_\iota Y \approx_\iota Y +_\iota X \qquad X \bowtie Y \tag{L5}$$

$$X +_\iota \mathbf{0}_\iota \approx_\iota X \tag{L6}$$

$$X \subseteq_\iota X +_\iota Y \qquad X \bowtie Y \tag{L7}$$

$$\mathrm{fst}_\iota \bowtie \mathrm{snd}_\iota \tag{L8}$$

$$\mathrm{fst}_\iota ;_\iota (X +_\iota Y) = X \qquad X \bowtie Y \tag{L9}$$

$$X \bowtie (Y +_\iota Z) \qquad X \bowtie Y, X \bowtie Z \tag{L10}$$

The majority of these laws are self explanatory, however we comment on a few. Sum laws like L4 use lens equality rather than homogeneous HOL equality since the left- and right-hand sides have different types. Law L9 shows how the fst_ι lens extracts the left-hand side of a product. Interestingly, these laws contain the separation algebra axioms [7], where separateness is characterised by $\bowtie$, and thus shows how our lens approach also generalises memory heap modelling. Thus we have an abstract model of state and an algebraic account of variables.

2.2 Expressions

Expressions have a similar type to predicates: $(\tau, \alpha)\,\mathsf{uexpr} \mathrel{\widehat{=}} \alpha \Rightarrow \tau$, where τ is the return type and α is the alphabet. We thus harness the HOL type system for ensuring well-formedness of expressions. HOL contains a large library of expression operators, such as arithmetic, and we lift these to UTP expressions. We also introduce the following core expressions constructs:

- $e =_u f$: equality of UTP expressions.
- $\&x$: obtains the value of lens $x : \alpha \Longrightarrow \tau$ in the state space.
- $\ll v \gg$: embeds a HOL expression of type τ into a UTP expression.

In general for expressions, we try to follow the standard mathematical syntax from the UTP book [24] and associated tutorials [39,8]. For example, for the

predicate operators we introduce overloaded constants so that the type system must determine whether operators like $\wedge$ and $\neg$ are the HOL or UTP versions. Where this is not possible, for example equality, we add a u subscript.

2.3 Meta-logical functions

Isabelle/UTP is based on a semantic model for alphabetised predicates, rather than syntax. Since we do not formalise a fixed abstract syntax tree for UTP predicates, there are no notions such as free variables or substitution that ordinarily would be recursive functions on the tree. Instead, we introduce weaker semantic notions that are sufficient to characterise the laws of programming:

- *Unrestriction*, $x \sharp P$, for lens x and predicate P, that semantically characterises variables that are fresh.
- *Semantic substitution*, $\sigma \dagger P$, for substitution function σ.
- *Alphabet extrusion*, $P \oplus_p a$, for lens a.
- *Alphabet restriction*, $P \upharpoonright_p a$, for lens a.

Intuitively, $x \sharp P$ holds, provided that P's valuation does not depend on x. For example, it follows that $x \sharp \mathit{true}$, $x \sharp \ll v \gg$, and $x \sharp (\exists\, x \bullet x >_u y)$, but not that $x \sharp (x =_u 1 \wedge y =_u 2)$. What differentiates it from syntactic freshness is that $x \sharp (x =_u 0 \vee x \neq_u 0)$, because the semantic valuation of this predicate is always true. Unrestriction can alternatively be characterised as predicates which satisfy the fixed point $P = (\exists\, x \bullet P)$ for very well-behaved lens x. Substitution application $\sigma \dagger P$ applies a substitution σ to P. A substitution function $\sigma : \alpha\,\mathsf{usubst}$ ($\widehat{=}\ \alpha \Rightarrow \alpha$) is a mapping from variables in the predicate's alphabet α to expressions to be inserted. Substitution update $\sigma(x \mapsto_s e)$ assigns the expression e to variable x in σ, and

$$[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] = \mathsf{id}(x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n)$$

creates a substitution for n variables. A substitution $P[\![e_1, \cdots, e_n/x_1, \cdots, x_n]\!]$ of n expressions to corresponding variables is then expressed as

$$[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] \dagger P$$

We now present some of the proven laws of substitutions.

Theorem 2 (Substitution query laws).

$$\langle \sigma(x \mapsto_s e) \rangle_s\, x = e \tag{SQ1}$$

$$\langle \sigma(y \mapsto_s e) \rangle_s\, x = \langle \sigma \rangle_s\, x \qquad \textit{if } x \bowtie y \tag{SQ2}$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f) \qquad \textit{if } x \subseteq_\iota y \tag{SQ3}$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f, x \mapsto_s e) \qquad \textit{if } x \bowtie y \tag{SQ4}$$

SQ1 and SQ2 show how substitution lookup is evaluated. SQ3 shows that an assignment to a larger lens overrides a previous assignment to a small lens and SQ4 shows that independent lens assignments can commute.

Theorem 3 (Substitution application laws).

$$\sigma \dagger \&x = \sigma(x) \tag{SA1}$$
$$\sigma(x \mapsto_s e) \dagger P = \sigma \dagger P \qquad \textit{if } x \sharp P \tag{SA2}$$
$$\sigma \dagger (\neg P) = \neg (\sigma \dagger P) \tag{SA3}$$
$$\sigma \dagger (P \wedge Q) = (\sigma \dagger P) \wedge (\sigma \dagger Q) \tag{SA4}$$
$$(\exists\, y \bullet P)[\![e/x]\!] = (\exists\, y \bullet P[\![e/x]\!]) \qquad \textit{if } x \bowtie y, y \sharp e \tag{SA5}$$

These laws effectively subsume the usual syntactic substitution laws, for an arbitrary number of variables, many of which simply show how substitution distributes through predicate operators.

Alphabet extrusion $P \oplus_p a$, for $P : \alpha\,\mathsf{upred}$, extends the alphabet type using lens $a : \alpha \Longrightarrow \beta$: it projects the predicate's alphabet α to "larger" alphabet type β. Lens a can be seen as a coercion that shows how the original state space α can be embedded into β. Effectively alphabet extrusion retains the predicate's characteristic valuation set over α, whilst filling in the rest of the variables in source alphabet β with arbitrary values.

Alphabet extrusion can be used to map a predicate $\alpha\,\mathsf{upred}$ to a relation $(\alpha \times \alpha)\,\mathsf{upred}$ by application of the lens $\mathrm{fst}_\lrcorner$ or $\mathrm{snd}_\lrcorner$, depending on whether a precondition or postcondition is desired. We give these two lifting operations the syntax $\lceil p \rceil_< \mathrel{\widehat{=}} p \oplus_p \mathrm{fst}_\lrcorner$ and $\lceil p \rceil_> \mathrel{\widehat{=}} p \oplus_p \mathrm{snd}_\lrcorner$, respectively, where p is a predicate in only undashed variables. We similarly create the substitution extension operator $\lceil \sigma \rceil_s$ that maps all variables and expressions to relational equivalents in undashed variables. Alphabet restriction is simply the inverse of extrusion: $P \restriction_p a$, for $P : \beta\,\mathsf{upred}$ and $a : \alpha \Longrightarrow \beta$, yields a predicate of alphabet α. Unlike extrusion this operation can be destructive if the predicate refers to variables in β but not in α. We demonstrate the following laws for extrusion and restriction:

Theorem 4 (Alphabet laws).

$$\mathit{true} \oplus_p a = \mathit{true} \tag{AE1}$$
$$\ll v \gg \oplus_p a = \ll v \gg \tag{AE2}$$
$$(P \wedge Q) \oplus_p a = (P \oplus_p a) \wedge (Q \oplus_p a) \tag{AE3}$$
$$\&x \oplus_p a = \&(x ;_\lrcorner a) \tag{AE4}$$
$$x \bowtie a \Rightarrow x \sharp (P \oplus_p a) \tag{AE5}$$
$$(P \oplus_p a) \restriction_p a = P \tag{AE6}$$

As indicated by laws AE1 and AE2, alphabet extrusion changes only the type of predicates with no variables; the body is left unchanged. Extrusion distributes through all the predicate operators, as expected, as indicated by law AE3. Applied to a variable, extrusion precomposes the variable lens with the given alphabet lens, as law AE4 demonstrates. Law AE5 shows that extrusion yields a predicate unrestricted by any variable x in the state-space extension. Finally, AE6 shows that alphabet restriction inverts alphabet extrusion.

2.4 Relations and Laws of Programming

A relation is a predicate with a product state space: $\alpha\,\mathsf{relation} \mathrel{\widehat{=}} (\alpha \times \alpha)\,\mathsf{upred}$. Variables of α can therefore be lifted to input or output variables by composing the corresponding lens with $\mathrm{fst}_{\scriptscriptstyle L}$ or $\mathrm{snd}_{\scriptscriptstyle L}$ respectively.

Definition 1 (Relational variables).

$$\$x \mathrel{\widehat{=}} x ;_{\scriptscriptstyle L} \mathrm{fst}_{\scriptscriptstyle L} \qquad\qquad \$x' \mathrel{\widehat{=}} x ;_{\scriptscriptstyle L} \mathrm{snd}_{\scriptscriptstyle L}$$

It is important to note that "$\$x$" is distinguished from "$\&x$": the former has a product alphabet whilst the latter has a scalar one. Thus $\&x$ is useful when writing predicates which should not contain dashed variables: $\$x =_u \&y$ will usually result in a type error. Alphabet coercion can be used to convert between relations and predicates, and in particular it follows that $\lceil \&x \rceil_< = \$x$.

We define the relational calculus operators like $P \,;\, Q$[2] and II by lifting of the constructs for HOL relations. Again, this gives us access to various built-in laws for binary relations, and allows us to produce a tactic for relational calculus, *rel-tac*. Conditional (if-then-else) is introduced using predicate operators as $P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg\, b \wedge Q)$. Assignment is defined as a general construct over a state substitution: $\langle \sigma \rangle_a : \alpha\,\mathsf{relation}$ updates the state by applying the substitution $\sigma : \alpha\,\mathsf{usubst}$ to the previous state. The alphabet of the substitution is α rather than $\alpha \times \alpha$ as this ensures that the assigned expressions cannot refer to post variables, as usual. The unary substitution $x := e$ can then be defined as $\langle [x \mapsto_s e] \rangle_a$, and similarly for simultaneous assignment of n variables. This has the advantage that the duality between substitution and assignment is clear in the corresponding algebraic laws. We have proven a large library of laws for relations, a selection of which is shown below, accompanied by the Isabelle names.

Theorem 5. *Relational laws of programming*

$$P \,;\, (Q \,;\, R) = (P \,;\, Q) \,;\, R \qquad \text{(seqr-assoc)}$$

$$\mathit{II} \,;\, P = P \qquad \text{(seqr-left-unit)}$$

$$\mathit{false} \,;\, P = \mathit{false} \qquad \text{(seqr-left-zero)}$$

$$(P \lhd b \rhd (Q \lhd b \rhd R)) = (P \lhd b \rhd R) \qquad \text{(cond-shadow)}$$

$$\lceil p \rceil_< \wedge \mathit{II} = \mathit{II} \wedge \lceil p \rceil_> \qquad \text{(pre-skip-post)}$$

$$(p \,;\, \mathit{true}) = p \Leftrightarrow \mathrm{snd}_{\scriptscriptstyle L} \sharp\, p \qquad \text{(precond-equiv)}$$

$$P \,;\, Q = (\exists\, v \bullet P[\![\langle\!\langle v \rangle\!\rangle / \$x']\!] \,;\, Q[\![\langle\!\langle v \rangle\!\rangle / \$x]\!]) \qquad \text{(seqr-middle)}$$

$$\langle \sigma \rangle_a \,;\, P = \lceil \sigma \rceil_s \dagger P \qquad \text{(assigns-r-comp)}$$

$$\langle \sigma \rangle_a \,;\, \langle \rho \rangle_a = \langle \rho \circ \sigma \rangle_a \qquad \text{(assigns-comp)}$$

We comment on a few of these. Law pre-skip-post shows that a precondition conjoined with relational identity can become a postcondition, since all variables are

[2] This is written as $P \,;;\, Q$ in Isabelle since ; is a delimiter for assumptions

identified. Law seqr-middle allows us to extract the intermediate value of a single variable in a sequential composition. Constant v is not a UTP state variable, but rather a logical HOL variable indicated by use of quoting. Law assigns-r-comp is a generalised version of the law $x := v \ ; \ P = P[v/x]$—it states that an assignment of σ followed by P equates to a substitution on P. We have to extend the alphabet of σ to match the relational alphabet of P using $\lceil \sigma \rceil_s$. Finally, law assigns-comp states that the sequential composition of two assignments corresponds to the functional composition of the two substitutions. From this law we can prove the assignment commutativity law:

Theorem 6. *Assignment commutativity*

$$(x := e \ ; \ y := f) = (y := f \ ; \ x := e) \quad \textit{if } x \bowtie y, x \sharp f, y \sharp e \quad \text{(assign-commute)}$$

Proof. By combination of laws assigns-comp and SQ4. □

Altogether we have proven over 200 hundred laws of predicate and relational calculus, many of which can be imported either from HOL or by Armstrong's algebraic hierarchy [1]. This then gives us the foundation on which to build UTP theories for Cyber-Physical Systems.

3 Example UTP theory

In order to exemplify the use of Isabelle/UTP, we mechanise a simple theory representing Boyle's law. Boyle's law states that, for an ideal gas at fixed temperature, pressure p is inversely proportional to volume V, or more formally that for $k = p \cdot V$ is invariant, for constant k. We here encode this as a simple UTP theory. We first create a record to represent the alphabet of the theory consisting of the three variables k, p and V.

```
record alpha-boyle =
  boyle-k :: real
  boyle-p :: real
  boyle-V :: real
```

For now we have to explicitly cast the fields to lenses using the VAR syntactic transformation function [11] – in the future this will be automated. We also have to add the definitional equations for these variables to the simplification set for predicates to enable automated proof through our tactics.

```
definition k :: real ⟹ alpha-boyle where k = VAR boyle-k
definition p :: real ⟹ alpha-boyle where p = VAR boyle-p
definition V :: real ⟹ alpha-boyle where V = VAR boyle-V

declare k-def [upred-defs] and p-def [upred-defs] and V-def [upred-defs]
```

We also prove that our new lenses are well-behaved and independent of each other. A selection of these properties is shown below.

```
lemma vwb-lens-k [simp]: vwb-lens k by (unfold-locales, simp-all add: k-def)
lemma boyle-indeps [simp]: k ⋈ p  p ⋈ k  k ⋈ V  V ⋈ k  p ⋈ V  V ⋈ p
  by (simp-all add: k-def p-def V-def lens-indep-def)
```

3.1 Static invariant

We first create a simple UTP theory representing Boyle's laws on a single state, as a static invariant healthiness condition. We state Boyle's law using the function B, which recalculates the value of the constant k based on p and V.

definition $B(\varphi) = ((\exists\ k \cdot \varphi) \wedge (\&k =_u \&p{\cdot}\&V))$

We can then prove that B is both idempotent and monotone simply by application of the predicate tactic. Idempotence means that healthy predicates cannot be made more healthy. Together with idempotence, monotonicity ensures that the image of the healthiness function forms a complete lattice, which is useful to allow the representation of recursive constructions with the theory.

lemma *B-idempotent*: $B(B(P)) = B(P)$ **by** *pred-tac*
lemma *B-monotone*: $X \sqsubseteq Y \Longrightarrow B(X) \sqsubseteq B(Y)$ **by** *pred-tac*

We also create some example observations; the first (φ_1) satisfies Boyle's law and the second doesn't (φ_2).

definition $\varphi_1 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 50))$
definition $\varphi_2 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$

We first prove an obvious property: that these two predicates are different observations. We must show that there exists a valuation of one which is not of the other. This is achieved through application of *pred-tac*, followed by *sledgehammer* [4] which yields a *metis* proof.

lemma φ_1-*diff*-φ_2: $\varphi_1 \neq \varphi_2$
  by (*pred-tac*, *metis select-convs num.distinct*(5) *numeral-eq-iff semiring-norm*(87))

We prove that φ_1 satisfies Boyle's law by application of the predicate calculus tactic, *pred-tac*.

lemma *B*-φ_1: φ_1 *is B* **by** (*pred-tac*)

We prove that φ_2 does not satisfy Boyle's law by showing that applying B to it results in φ_1. We prove this using Isabelle's natural proof language, Isar, details of which can be found in the reference manual [36]. The proof below is annotated with comments.

lemma *B*-φ_2: $B(\varphi_2) = \varphi_1$
proof −
  — We first expand out the definition of φ_2
  have $B(\varphi_2) = B(\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 100)$
    by (*simp add*: φ_2-*def*)
  — Then the definition of B
  also have ... $= ((\exists\ k \cdot \&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 100) \wedge \&k =_u \&p{\cdot}\&V)$
    by (*simp add*: *B-def*)
  — The existentially quantifier k can be removed
  also have ... $= (\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u \&p{\cdot}\&V)$
    by *pred-tac*
  — We show that $(10::'a) \cdot (5::'a) = (50::'a)$
  also have ... $= (\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 50)$

by *pred-tac*
— This is then definitionally equal to φ_1
also have ... = φ_1
by (*simp add*: φ_1-*def*)
— Finally we show the overall thesis
finally show *?thesis* **.**
qed

3.2 Dynamic invariants

Next we build a relational theory that allows the pressure and volume to be changed, whilst still respecting Boyle's law. We create two dynamic invariants for this purpose.

definition $D1(P) = (((\$k =_u \$p \cdot \$V \Rightarrow \$k\acute{} =_u \$p\acute{} \cdot \$V\acute{}) \wedge P)$
definition $D2(P) = (\$k\acute{} =_u \$k \wedge P)$

D1 states that if Boyle's law satisfied in the previous state, then it should be satisfied in the next state. We define this by conjunction of the formal specification of this property with the predicate. The annotations $\$p$ and $\$p\acute{}$ refer to relational variables p and p'. *D2* states that the constant k indeed remains constant throughout the evolution of the system, which is also specified as a conjunctive healthiness condition. As before we demonstrate that *D1* and *D2* are both idempotent and monotone.

lemma *D1-idempotent*: $D1(D1(P)) = D1(P)$ **by** *rel-tac*
lemma *D2-idempotent*: $D2(D2(P)) = D2(P)$ **by** *rel-tac*

lemma *D1-monotone*: $X \sqsubseteq Y \Longrightarrow D1(X) \sqsubseteq D1(Y)$ **by** *rel-tac*
lemma *D2-monotone*: $X \sqsubseteq Y \Longrightarrow D2(X) \sqsubseteq D2(Y)$ **by** *rel-tac*

Since these properties are relational, we discharge them using our relational calculus tactic *rel-tac*. Next we specify three operations that make up the signature of the theory.

definition *InitSys ip iV*
$= ((\ll ip \gg >_u 0 \wedge \ll iV \gg >_u 0)^\top \;;;\; p,V,k := \ll ip \gg, \ll iV \gg, (\ll ip \gg \cdot \ll iV \gg))$

definition *ChPres dp*
$= ((\&p + \ll dp \gg >_u 0)^\top \;;;\; p := \&p + \ll dp \gg \;;;\; V := (\&k / \&p))$

definition *ChVol dV*
$= ((\&V + \ll dV \gg >_u 0)^\top \;;;\; V := \&V + \ll dV \gg \;;;\; p := (\&k / \&V))$

InitSys initialises the system with a given initial pressure (ip) and volume (iV). It assumes that both are greater than 0 using the assumption construct $c^\top$ which equates to *II* if c is true and *false* (i.e. errant) otherwise. It then creates a state assignment for p and V, uses the B healthiness condition to make it healthy (by calculating k), and finally turns the predicate into a postcondition using the $\lceil P \rceil_>$ function.

ChPres raises or lowers the pressure based on an input *dp*. It assumes that the resulting pressure change would not result in a zero or negative pressure, i.e. $p + dp > 0$. It assigns the updated value to p and recalculates V using the original value of k. *ChVol* is similar but updates the volume.

lemma *D1-InitSystem*: *D1* (*InitSys ip iV*) = *InitSys ip iV* **by** *rel-tac*

InitSys is *D1*, since it establishes the invariant for the system. However, it is not *D2* since it sets the global value of k and thus can change its value. We can however show that both *ChPres* and *ChVol* are healthy relations.

lemma *D1*: *D1* (*ChPres dp*) = *ChPres dp* **and** *D1* (*ChVol dV*) = *ChVol dV*
by (*rel-tac*, *rel-tac*)

lemma *D2*: *D2* (*ChPres dp*) = *ChPres dp* **and** *D2* (*ChVol dV*) = *ChVol dV*
by (*rel-tac*, *rel-tac*)

Finally we show a calculation for a simple animation of Boyle's law, where the initial pressure and volume are set to 10 and 4, respectively, and then the pressure is lowered by 2.

lemma *ChPres-example*:
(*InitSys 10 4* ;; *ChPres* (−*2*)) = *p*,*V*,*k* := *8*,*5*,*40*
proof −
— *InitSys* yields an assignment to the three variables
have *InitSys 10 4* = *p*,*V*,*k* := *10*,*4*,*40*
by (*rel-tac*)
— This assignment becomes a substitution
hence (*InitSys 10 4* ;; *ChPres* (−*2*))
= (*ChPres* (−*2*))⟦*10*,*4*,*40*/$*p*,$*V*,$*k*⟧
by (*simp add*: *assigns-r-comp alpha*)
— Unfold definition of *ChPres*
also have ... = ((&*p* − *2* >$_u$ *0*)$^\top$⟦*10*,*4*,*40*/$*p*,$*V*,$*k*⟧
;; *p* := &*p* − *2* ;; *V* := &*k* / &*p*)
by (*simp add*: *ChPres-def lit-num-simps usubst unrest*)
— Unfold definition of assumption
also have ... = ((*p*,*V*,*k* := *10*,*4*,*40* ◁ (*8* :$_u$ *real*) >$_u$ *0* ▷ *false*)
;; *p* := &*p* − *2* ;; *V* := &*k* / &*p*)
by (*simp add*: *rassume-def usubst alpha unrest*)
— (*0*::′*a*) < (*8*::′*a*) is true; simplify conditional
also have ... = (*p*,*V*,*k* := *10*,*4*,*40* ;; *p* := &*p* − *2* ;; *V* := &*k* / &*p*)
by *rel-tac*
— Application of both assignments
also have ... = *p*,*V*,*k* := *8*,*5*,*40*
by *rel-tac*
finally show *?thesis* **.**
qed

4 Theories of Cyber-Physical Systems

In this section we describe some the key UTP theories we have mechanised which form the basis for our future semantic model of Cyber-Physical Systems.

4.1 Designs

The simplest theory in UTP is that of a nondeterministic imperative programming language expressed in the relational calculus of alphabetised predicates arranged in a complete lattice. The ordering is refinement, which is defined as universal inverse implication: $(P \sqsubseteq Q) = [Q \Rightarrow P]$ (here the brackets are universal closure over the alphabet). The worst program, the bottom of the lattice, is abort, with semantics ***true***; the best program, the top of the lattice, is miracle, with semantics ***false***. This theory of nondeterministic programming is that of partial correctness, with recursion given a strongest fixed-point semantics. The choice of semantics for recursion is a very practical one to make the theory work. If the weakest fixed-point were chosen, then some desirable laws would fail to hold. For example, we'd certainly like the following law to hold: $abort\ ;\ P = abort$. Choosing a weakest fixed-point semantics gives us the equation $(\textbf{\textit{true}}\ ;\ x := 0) = x := 0$, for a state with a single variable x: it is possible to recover from abort (for example, a non-termination recursion) and behave as though it had never happened. On the other hand, the choice of the strongest fixed-point would validate the law, thus: $(\textbf{\textit{false}}\ ;\ x := 0) = \textbf{\textit{false}}$. It turns out that the strongest fixed-point is also easier to reason with. Compare the laws defining the extreme properties of the two operators:

$$(F(P) \sqsubseteq P) \Rightarrow (\mu\, F \sqsubseteq P) \qquad\qquad (S \sqsubseteq F(S)) \Rightarrow (S \sqsubseteq \nu F)$$

The left-hand law states that if P is a pre-fixed-point of F, then it can't be any weaker than the weakest fixed-point. This would be useful in reasoning about a recursive specification $\mu\, F$ of a program P. The right-hand law states that if S is a post-fixed-point of F, then it can't be any stronger than the strongest fixed-point. This would be useful in reasoning about a recursive implementation νF of a specification S. The left-hand law seems more practically useful than the right-hand one. The cost of this practical benefit is an inescapable law: $S \sqsubseteq abort$, for every specification S, since abort, with a strongest fixed-point semantics, is the top of the lattice. So the result is a theory of partial correctness: if we have $S \sqsubseteq P$, and the P terminates (that is, it is not $abort$), then P is correct. For this price, a simple rule is obtained in Hoare logic for reasoning about the (partial) correctness of loops:

$$\frac{\{\ b \wedge c\ \}\ \ P\ \ \{\ c\ \}}{\{\ b \wedge c\ \}\ \ \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P\ \ \{\ \neg\, b \wedge c\ \}}$$

So it was that the proof rules for fixed-points determined the early emphasis of partial correctness in program verification.

UTP's theory of designs extends the treatment of the nondeterministic imperative programming language from partial to total correctness. This is done by restricting attention to a subclass of predicate for which the left and right-zero laws actually hold: (true ; P) = true = (P ; true). These predicates are called designs.

The insight is to capture the theory of assertional reasoning and assumption-commitment pairs as single relations by adding two observations: that a program has started ok and that a program has terminated ok'. A design is then a precondition-postcondition pair

$$(P \vdash Q) \quad \widehat{=} \quad (ok \wedge P \Rightarrow ok' \wedge Q) \quad \text{for } P \text{ and } Q \text{ not containing } ok \text{ or } ok'$$

This is read as "if the program has started (ok) and the precondition P holds, then it must terminate (ok') in a state where the postcondition Q holds." This is clearly a statement of total correctness. Notice that, although the syntax of a design is a pair of alphabetised predicates, its meaning is encoded as a single predicate.

Designs form a complete lattice with $\mathit{false} \vdash Q$ (abort) at the bottom and $\mathit{true} \vdash \mathit{false}$ (miracle) at the top. These two definitions can be simplified as true and $\neg\ ok$, respectively. Thus, abort permits any and every behaviour, whilst a miracle, quite properly, cannot be started, and so has no behaviours at all.

A theory in UTP has three components. The first is the signature; here this is the syntax of the programming language and the syntax of a design pair. The second component is the alphabet; here this is the two boolean observations ok and ok' NS ny program variables. The third component is a set of healthiness conditions characterising membership of the theory. In the case of designs, there are two healthiness conditions, one concerning each observational variable. The first states that no observation may be made of a program before it has started. This is necessary for proper initialisation and to make sequential composition work properly.

$$\mathit{H1}(P) \quad \widehat{=} \quad ok \Rightarrow P$$

The healthiness condition is presented as a monotone idempotent function; its fixed points are its healthy predicates.

The second healthiness condition concerns termination and seeks to eliminate the specification that would require a program not to terminate: $\neg\ ok'$. Refinement allows us to write a correct program that improves on what a specification requires. In our programming methodology, anything is better than nontermination, so you should not be allowed to require nontermination. The following healthiness condition formalises this:

$$\mathit{H2}(P) = P \quad \Leftrightarrow \quad [\ P^f \Rightarrow P^t\ ]$$

where $P^f \ \widehat{=}\ P[\mathbf{false}/ok']$ and $P^t \ \widehat{=}\ P[\mathbf{true}/ok']$. Hoare & He show how to present this condition in terms of the fixed points of the monotone idempotent function $\mathit{H2}$ [H&H]. They also shows how to characterise the space of designs in

three equivalent ways: syntactically, as the fixed points of these two healthiness conditions, and as the solutions of algebraic equations (left unit and left zero). Finally, they prove that the lattice of designs is closed under the nondeterministic programming language's combinators with assignment as the basis.

The theory of designs has been mechanised in *Isabelle/UTP* and we show an excerpt from this theory. We introduce the alphabet by parametric type $'\alpha$ *alphabet-d* [11,12] which extends the alphabet $'\alpha$ with the variable lens *ok*. Moreover, we add the useful type synonym

type-synonym $'\alpha$ *hrelation-d* = ($'\alpha$ *alphabet-d*, $'\alpha$ *alphabet-d*) *relation*

which describes a homogeneous relation with a design alphabet. We then use these to create the signature and healthiness conditions of designs in a similar way to the theory demonstrated in Section 3. Then many standard laws of designs can be proved automatically, as the following demonstrates.

theorem *design-false-pre*: (*false* $\vdash$ *P*) = *true* **by** *rel-tac*

Of course not all properties can be proved this way, and in any case there is great value in presenting the intuition behind a theorem through proof. We demonstrate this firstly that the syntactic form of designs is equivalent to the healthiness conditions.

theorem *H1-H2-eq-design*: *H1* (*H2 P*) = ($\neg$ P^f) $\vdash$ P^t
proof −
  have *H1* (*H2 P*) = (\$*ok* $\Rightarrow$ *H2*(*P*))
    by (*simp add*: *H1-def*)
  also have ... = (\$*ok* $\Rightarrow$ (P^f $\vee$ (P^t $\wedge$ \$*ok´*)))
    by (*metis H2-split*)
  also have ... = (\$*ok* $\wedge$ ($\neg$ P^f) $\Rightarrow$ \$*ok´* $\wedge$ \$*ok* $\wedge$ P^t)
    by *rel-tac*
  also have ... = ($\neg$ P^f) $\vdash$ P^t
    by *rel-tac*
  finally show *?thesis* **.**
qed

This proof makes use of the auxiliary theorem *H2-split* to expand out ***H2*** which states that ***H2***$(P) = P^f \vee (P^t \wedge ok')$. We also show that the design identity II_D is a right unit of any design. We define this element of the signature as follows:

definition *skip-d* :: $'\alpha$ *hrelation-d* (II_D) **where** II_D = (*true* $\vdash_r$ *II*)

The turnstile $P \vdash_r Q$ is a specialisation of $P \vdash Q$ which requires that neither P nor Q have ok, ok' in their alphabets. It use alphabet extrusion and the Isabelle type system to ensure this: $ok \vdash_r P$ entails a type error. Proof of the right unit law requires that we can calculate the sequential composition of two designs, which the following theorem demonstrates.

theorem *rdesign-composition-cond*:
  assumes *out*α $\sharp$ p_1
  shows (($p_1 \vdash_r Q_1$) ;; ($P_2 \vdash_r Q_2$)) = (($p_1 \wedge \neg$ (Q_1 ;; ($\neg$ P_2))) $\vdash_r$ (Q_1 ;; Q_2))
  — proof omitted

This is itself a specialisation of the more complex design composition law [8] which adds the requirement that the assumption of the first design be a condition. Thus the theorem assumes $p1$ does not refer to variables in the output alphabet, $out\alpha$, which is just shorthand for $\mathrm{fst}_{\vartriangleleft}$. The law demonstrates the advantages of the alphabets-as-types approach: we do not require provisos that p_1, Q_1, P_2, and Q_2 do not refer to ok and ok' which greatly simplifies the theorem and its application. We can now prove the unit law, which we do in Isar.

theorem *rdesign-left-unit*:
  fixes P Q :: $'\alpha$ *hrelation-d*
  shows $(II_D \;;;\; P \vdash_r Q) = (P \vdash_r Q)$
proof −
  — We first expand out the definition of the design identity
  have $(II_D \;;;\; P \vdash_r Q) = (true \vdash_r II \;;;\; P \vdash_r Q)$
    by (*simp add*: *skip-d-def*)
  — Next, we apply the design composition law above in a subproof
  also have ... $= (true \wedge \neg\, (II \;;;\; \neg\, P)) \vdash_r (II \;;;\; Q)$
  proof −
    — The assumption of identity is ***true*** so it is easy to discharge the proviso
    have $out\alpha \sharp true$
      by *unrest-tac*
    — From this we can apply the composition law
    thus *?thesis*
      using *rdesign-composition-cond* **by** *blast*
  qed
  — Simplification then allows us to remove extraneous terms
  also have ... $= (\neg\, (\neg\, P)) \vdash_r Q$
    by *simp*
  — Finally, we can show the thesis
  finally show *?thesis* **by** *simp*
qed

4.2 Reactive processes

A more sophisticated UTP theory is that of reactive processes. In the reactive paradigm, a process is a pattern of behaviour expressed in terms of observable events. In general, the behaviour is as follows. The process minds its own business internally until it's ready to interact with its environment; it then pauses and waits until its environment is cooperative, whereupon it reacts and then returns to its own business; this behaviour is repeated. A reactive process characteristically has two sorts or after-states: intermediate states, where the process is waiting for interaction with its environment; and final states, where the process has reached its ultimate computation, completed its behaviour, and terminated.

We investigate this paradigm in terms of its three components as a UTP theory.

First, we consider the signature of the theory. We consider a simple extension of the nondeterministic programming language in the previous section, augmented by an operator that synchronises on an event with the environment.

If P is a reactive process, then $a \to P$ is another process that first engages in the synchronisation of the event a and then behaves like the process P.

Next, we consider the alphabet of observational variables.

We can observe the sequence of events synchronised by an individual reactive process. We call this sequence a trace, and denote its before-value by tr and its intermediate or final value by tr'. It is a sequence over the set of events.

We can also observe whether a reactive process is in one of its waiting states. This is an observation that we denote by the boolean variables $wait$, in the before state, and $wait'$ in the intermediate or final state.

The stability of a reactive process is described in the same way as the termination of a nondeterministic program. That is, ok' describes whether the reactive process has reached a stable state, whether it be intermediate or final. Thus, the combination of ok' and $wait'$ is of interest. If $ok' \land wait'$, then the process has reached a stable intermediate state. If $ok' \land \neg\, wait'$, then the process has reached a stable final state. Regardless of the value of $wait'$, if $\neg\, ok'$, then the process is in a divergent state.

The final observation that may be made of a reactive process concerns its liveness. The process $a \to SKIP$ is waiting to perform the event a and then terminate ($SKIP$). While it is waiting, it cannot refuse to perform a. The observational variable ref' records this fact. We can think of the value of ref' as an experiment offered by the environment: will the process deadlock if we offer these events? Suppose that the universe of events is $\{a, b, c\}$. Our process will deadlock if we offer it the empty experiment $\varnothing$ (all processes have this property). It will also deadlock if we offer it either or both b or c. The maximal refusal is the pair $\{b, c\}$; note that the process will refuse any subset: ref' is downward closed. Now consider the nondeterministic process $a \to SKIP \sqcap b \to SKIP$. The nondeterministic choice can be resolved in two ways: if the first branch is taken, then it may refuse b; if the second branch is taken, then it may refuse a. Note that although ref' is downward closed, there is no maximal refusal set. Recording a refusal set is one way of capturing this kind of nondeterministic choice. Our process is then partially specified by the predicate

$$\begin{array}{l}\textbf{if } wait' \textbf{ then} \\ \quad (tr' = tr) \land (ref' \subseteq \{b, c\} \lor ref' \subseteq \{a, c\}) \land ok' \\ \textbf{else} \\ \quad ((tr' = tr \frown \langle a \rangle) \lor (tr' = tr \frown \langle b \rangle)) \land ok'\end{array}$$

Reactive processes have three healthiness conditions. The first requires that the trace grows monotonically, so that history is never edited.

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

(Here, $\leq$ denotes the sequence prefix relation.)

The second healthiness condition requires that a process P is insensitive to the trace of events that occur before P is initiated:

$$\mathbf{R2}(P) \;\widehat{=}\; P[\langle\rangle, tr' - tr / tr, tr'] \lhd tr \leq tr' \rhd P$$

(Here we use the sequence subtraction operator.)

Finally, sequential composition must be made to work as it does in a programming language, and not merely as relational composition. In the sequence $P\ ;\ Q$, if P is waiting, then Q must not be initiated. Define

$$\mathit{II}_{rea} \ \widehat{=}\ \mathbf{R1} \circ \mathbf{H1}((ok', wait', tr', ref', x') = (ok, wait, tr, ref, x))$$

where x is a list of the process's state variables. Our healthiness condition is

$$\mathbf{R3}(P) \ \widehat{=}\ (\mathit{II}_{rea} \lhd wait \rhd P)$$

For the full semantics, other healthiness conditions are needed, but almost all the process algebraic laws for CSP can be proved correct based on the semantics presented so far, providing we add two more healthiness conditions concerning ok and ok'. Fortunately, we have already presented them: they are **H1** and **H2**, simply adjusted for the larger alphabet of reactive processes.

The CSP processes are the fixed points of the montone idempotent function

$$\mathbf{CSP} \ \widehat{=}\ \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \circ \mathbf{H1} \circ \mathbf{H2}$$

Equivalently, by theorem *H1-H2-eq-design* every **CSP** relation can be stated as a reactive design of the form $\mathbf{R}(P \vdash Q)$, where $\mathbf{R} \mathrel{\widehat{=}} \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$, and P and Q are assumptions and commitments over the trace and program variables. For example, the worst CSP process is $\mathbf{Chaos} \mathrel{\widehat{=}} \mathbf{R}(\mathbf{false} \vdash \mathbf{true})$, which fails to satisfy its assumption and thus establishes nothing other than that the trace must increase monotonically (by **R1**). Every CSP process can be expressed as such a reactive design [8].

We have likewise mechanised the theory of reactive designs, and here show a few of the properties proved, though without proofs for reasons of space. The first property shows that **Chaos** is indeed the bottom of the lattice – every CSP process refines it. The second shows that **Chaos** is a left zero for sequential composition: since $wait'$ is always **false** the second process can never be executed.

theorem *Chaos-least*: **assumes** *P is CSP* **shows** $\mathit{Chaos} \sqsubseteq P$
— proof omitted

theorem *Chaos-left-zero*: **assumes** *P is CSP* **shows** $(\mathit{Chaos}\ ;;\ P) = \mathit{Chaos}$
— proof omitted

More laws we have proved can be found in our online UTP repository[3].

4.3 Hybrid Relational Calculus

Differential Algebraic Equations (DAEs) are often used to model the continuously evolving dynamic behaviour of a system. The theory of hybrid relations in UTP unifies discrete and continuous variables used in such models. We introduce

[3] github.com/isabelle-utp/utp-main/blob/master/utp/utp_reactive.thy

a theory of continuous-time processes that embeds in the theory of alphabetised predicates trajectories of states evolving over time intervals representing piecewise continuous behaviour.

We start with the UTP theory of alphabetised relations, which therefore will not capture continuous process termination or stability. This allows us to treat the behaviour of hybrid processes as an individual phenomenon, before augmenting the theory with additional structure to capture such properties by embedding it in the theory of timed reactive designs [19,35].

Alphabet. Our theory has two variables $ti, ti' : \mathbb{R}_{\geq 0}$ that observe the start and end time of the current computation interval and its duration $\ell = ti' - ti$, as in the Duration Calculus [41]. Following [20], we classify the alphabet of a hybrid relation in three disjoint parts: input variables, $\mathrm{in}\alpha(P)$; output variables, $\mathrm{out}\alpha(P)$; and continuous variables, $\mathrm{con}\alpha(P)$ (such as $\underline{x}, \underline{y}, \underline{z}$). Continuous variables of type $\mathbb{R}$ describe a value at a particular instant of time; trajectory variables of type $\mathbb{R}_{\geq 0} \to \mathbb{R}$ describe the evolution of a value over all time (values outside $[ti, ti')$ are irrelevant).

A junction between the discrete and continuous world is established by making a discrete copies $x, x' : \mathbb{R}$ of the values of each continuous variable $\underline{x} : \mathbb{R}_{\geq 0} \to \mathbb{R}$ at the beginning and end of the interval under observation. Discrete variables that are not surrogates for continuous variables are in the sub-alphabet

$$\mathrm{dis}\alpha(P) \mathrel{\widehat{=}} \{\ x \in \mathrm{in}\alpha(P) \mid \underline{x} \notin \mathrm{con}\alpha(P)\ \} \cup \{\ x' \in \mathrm{out}\alpha(P)\ \mid\ \underline{x} \notin \mathrm{con}\alpha(P)\ \}$$

Following [13], we define a continuous variable lifting operator from a predicate in instant variables to one in trajectory variables:

$$P\,@\,\tau \quad \widehat{=} \quad \{\ \underline{x} \mapsto \underline{x}(\tau) \mid \underline{x} \in \mathrm{con}\alpha(P) \setminus \{\underline{t}\}\ \} \dagger P$$

In $P\,@\,\tau$, we map every flat continuous variable (other than the distinguished time variable $\underline{t} \in [ti..ti')$) to a corresponding variable lifted over the time domain. So the new predicate holds for values of continuous variables at the instant τ, a variable that is free in P. So each flat continuous variable $\underline{x} : T$ is transformed to a time-dependent function $\underline{x} : \mathbb{R} \to T$ type. In this way, we lift time predicates to intervals.

Our hybrid theory has two healthiness conditions:

$$\mathbf{HCT1}(P) \quad \widehat{=} \quad P \wedge ti \leq ti'$$

$$\mathbf{HCT2}(P) \quad \widehat{=} \quad P \wedge \left(ti < ti' \Rightarrow \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} \left(\begin{array}{l} \exists\, I : \mathbb{R}_{\mathsf{oseq}} \bullet \\ \quad \mathrm{ran}(I) \subseteq \{ti \ldots ti'\} \\ \quad \wedge\ \{ti, ti'\} \subseteq \mathrm{ran}(I) \\ \quad \wedge\ (\forall\, n < \#I - 1 \bullet \underline{v}\ \mathsf{cont\text{-}on}\ [I_n, I_{n+1})) \end{array} \right) \right)$$

where $\mathbb{R}_{\mathsf{oseq}} \quad \widehat{=} \quad \{\ x : \mathrm{seq}\,\mathbb{R} \mid \forall\, n < \#x - 1 \bullet x_n < x_{n+1}\ \}$

$$f\ \mathsf{cont\text{-}on}\ [m, n) \quad \widehat{=} \quad \forall\, t \in [m, n) \bullet \lim_{x \to t} f(x) = f(t)$$

HCT1 requires that time advances monotonically. ***HCT2*** requires that every continuous variable $\underline{v}$ is piecewise continuous: for non-empty intervals there is a finite number of discontinuous points (the range of I) between ti and ti'. The set of totally ordered sequences $\mathbb{R}_{\mathsf{oseq}}$ captures the set of discontinuities; the continuity of f is defined in the usual way by requiring that at each point in $[ti, ti')$, the limit correctly predicts the function.

Both healthiness conditions are idempotent, monotone, and commutative, as is their composition $\mathbf{HCT} = \mathbf{HCT2} \circ \mathbf{HCT1}$. The image of $\mathbf{HCT}$ a complete lattice.

The signature of our theory is as follows:

$$P, Q ::= \mathit{II} \mid P \,;\, Q \mid P \lhd b \rhd Q \mid x := e \mid P^* \mid P^\omega \mid \\ [\![P]\!] \mid \langle\, F_n \mid b \,\rangle \mid P \,[\, b \,]\, Q$$

This syntax extends the signature of the alphabetised relational calculus with operators to specify intervals $[\![P]\!]$, differential algebraic equations $\langle\, F_n \mid b \,\rangle$, and behavioural preemption $P \,[\, b \,]\, Q$. P^* and P^ω describe finite and infinite iteration, respectively. The following operators of relational calculus $P \,;\, Q$, $P \lhd b \rhd Q$, P^*, II, $x := v$, ***true***, and ***false*** are $\mathbf{HCT}$ closed.

Finally, we define the interval operator from the Duration Calculus [41] and our own variant.

$$\lceil P \rceil \;\widehat{=}\; \mathbf{HCT2}(\ell > 0 \wedge (\forall\, \underline{t} \in [ti, ti') \bullet P \,@\, \underline{t}))$$

$$[\![P]\!] \;\widehat{=}\; \lceil P \rceil \wedge \textstyle\bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} (v = \underline{v}(ti) \wedge v' = \lim_{t \to ti'} (\underline{v}(t))) \wedge \mathit{II}_{\mathrm{dis}\alpha(P)}$$

$\lceil P \rceil$ is taken from the Duration Calculus: it is a continuous specification statement that P holds at every instant over all non-empty right-open intervals from ti to ti'; we make it healthy with ***HCT2*** for piecewise continuity. $[\![P]\!]$ links discrete and continuous variables with the given property.

By making x' the limit of $\underline{x}$, rather than its value at the end of the interval, we do not constrain the trajectory valuation at ti'; so it can be defined by a suitable discontinuous discrete assignment at this final instant. Following [21], we use the interval operator to give the basis of systems of differential equations. As a result, we can refine a DAE, under given initial conditions, to a suitable solution expressed using the interval operator. Intervals satisfy a number of standard laws.

$$\lceil \mathit{true} \rceil = \ell > 0 \qquad \lceil \mathit{false} \rceil = \mathbf{false} \qquad \lceil P \wedge Q \rceil = \lceil P \rceil \wedge \lceil Q \rceil$$

$$\lceil P \vee Q \rceil \sqsubseteq \lceil P \rceil \vee \lceil Q \rceil \qquad [\![P]\!] \sqsubseteq [\![P]\!] \,;\, [\![P]\!]$$

The evolution of a DAE system in semi-explicit form is modelled by an operator, adapted from $\mathcal{HCSP}$ [41,27].

$$\langle\, \dot{\underline{v}}_1 = f_1; \cdots; \dot{\underline{v}}_n = f_n \mid 0 = b_1; \cdots; 0 = b_m \,\rangle \\ \widehat{=}\; [\![\; (\forall\, i \in 1..n, \forall\, j \in 1..m \bullet \\ \dot{\underline{v}}_i(\underline{t}) = f_i(\underline{t}, \underline{v}_1(\underline{t}), \cdots, \underline{v}_n(\underline{t}), \underline{w}_1(\underline{t}), \cdots, \underline{w}_m(\underline{t}))) \\ \wedge\, 0 = b_j(\underline{t}, \underline{v}_1(\underline{t}), \cdots, \underline{v}_n(\underline{t}), \underline{w}_1(\underline{t}), \cdots, \underline{w}_m(\underline{t})) \;]\!]$$

A DAE $\langle\, F_n \mid B_m \,\rangle$ consists of a set of n functions $f_i : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, which define the derivative of variable $\underline{v}_i$ in terms of the independent time variable $\underline{t}$ and $n + m$ dependent variables. It also contains algebraic constraints $b_j : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ that must be invariant for any solution and do not refer to derivatives. For $m = 0$ the DAE corresponds to an ODE, which we write as $\langle\, F_n \,\rangle$. The DAE operator is defined using the interval operator to be all non-empty intervals over which a solution satisfying both the ODEs and algebraic constraint exists. Non-emptiness is important as it means that a DAE must make progress: it cannot simply take zero time since $\ell > 0$, and so a DAE cannot directly cause "chattering Zeno" effects when placed in the context of a loop, though normal Zeno effects remain a possibility.

To obtain a well defined problem description, we require the following conditions to hold [2]: (i) The system of equations is consistent and neither underdetermined nor overdetermined. (ii) the discrete input variables v_i provide consistent initial conditions. (iii) the equations are specific enough to define a unique solution during the interval ℓ. The system is then allowed to evolve from this point in the interval between ti and ti' according to the DAEs. At the end of the interval, the corresponding output discrete variables are assigned. During the evolution all discrete variables and unconstrained continuous variables are held constant.

Finally, we define the preemption operator, adapted from $\mathcal{HCSP}$.

$$P\,[\,b\,]\,Q \quad \widehat{=} \quad (Q \lhd b \,@\, ti \rhd (P \wedge \lceil \neg b \rceil)) \vee ((\lceil \neg b \rceil \wedge b \,@\, ti' \wedge P)\,;\,Q)$$

P is a continuous process that evolves until the predicate B is satisfied, at which point Q is activated. The semantics is defined as a disjunction of two predicates. The first predicate states that, if B holds in the initial state of ti, then Q is activated immediately. Otherwise, P is activated and can evolve while B remains false (potentially indefinitely). The second predicate states that $\neg B$ holds on the interval $[ti, ti')$ until instant ti', when B switches to a true valuation; during that interval P is executing. Following this, P is terminated and Q is activated.

Although space does not permit us to go into details, we have mechanised this theory in Isabelle/UTP[4].

5 Conclusions

In this paper we describe our work towards building a mechanised library of computational theories in the context of the UTP, including those for concurrent and hybrid systems. Our aim in the future is to use these theories to enable integration of heterogeneous multi-model semantics, as described by FMI, for the purpose of multi-pronged verification. We are currently working on integrating hybrid relations and reactive in order to mechanise *hybrid reactive designs*. A hybrid reactive design has the form $\boldsymbol{R}(P \wedge [\![R]\!] \vdash Q \wedge [\![G]\!])$, where P and Q are the precondition and postcondition on the discrete state, and R and G are

[4] See github.com/isabelle-utp/utp-main/blob/master/utp/utp_hybrid.thy.

assumptions and commitments on the continuous variables. Such a construction will enable us to apply contractual-style program construction and reasoning to concurrent Cyber-Physical Systems. Moreover work is underway to explore other theories relevant for CPS, in particular real-time modelling and probability. Once these theories are mechanised we will also explore the links between them, in particular useful Galois connections between discrete and continuous time domains, which are practically applicable for verification.

Though our *Isabelle/UTP* theory library is a step forward, further work in needed particularly in the direction of tool integration. As Hoare and He pointed out in Chapter 0 of the UTP book [24]:

> *At present, the main available mechanised mathematical tools are programmed for use in isolation [...] it will be necessary to build within each tool a structured library of programming design aids which take the advantage of the particular strengths of that tool. To ensure the tools may safely be used in combination, it is essential that these theories be unified.*

We believe that the Isabelle framework is a significant step towards acquisition of this goal. Nevertheless, there is certainly more to be done, particularly in the area of mechanisation of continuous mathematics and application of associated computational algebra tools.

Acknowledgements. This work is partly by EU H2020 project *INTO-CPS*, grant agreement 644047, http://into-cps.au.dk/. We would like to thank our colleagues Ana Cavalcanti, Bernhard Thiele, and Burkhart Wolff for their collaboration on this work.

References

1. A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.
2. Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. Robust initialization of differential algebraic equations. In Peter Fritzson, François E. Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, EOOLT 2007, Berlin, Germany, July 30, 2007*, volume 24 of *Linköping Electronic Conference Proceedings*, pages 151–163. Linköping University Electronic Press, 2007.
3. M. Banks and J. Jacobs. Unifying theories of confidentiality. In *UTP*, volume 6445 of *LNCS*, pages 120–136. Springer, 2010.
4. J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
5. T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. V. Peetz, S. Wolf, and C. Clauß. The Functional Mockup Interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114, 2011.

6. R. Bresciani and A. Butterfield. A UTP semantics of pGCL as a homogeneous relation. In *Proc. 9th Intl. Conf. on Integrated Formal Methods (iFM)*, volume 7321 of *LNCS*, pages 191–205. Springer, 2012.
7. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE, July 2007.
8. Ana Cavalcanti and Jim Woodcock. A tutorial introduction to CSP in *Unifying Theories of Programming*. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23–December 5, 2004, Revised Lectures*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, 2006.
9. Ana Cavalcanti, Jim Woodcock, and Nuno Amálio. Behavioural models for FMI co-simulations. In *ICTAC*, 2016. In press.
10. Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):13–28, January 2012.
11. A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.
12. A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
13. Colin J. Fidge. Modelling discrete behaviour in a continuous-time formalism. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Integrated Formal Methods, Proceedings of the 1st International Conference on Integrated Formal Methods, IFM 99, York, UK, 28–29 June 1999*, pages 170–188. Springer, 1999.
14. J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
15. S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP semantics for Modelica. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, June 2016. To appear.
16. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
17. S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.
18. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
19. Ian J. Hayes, Steve Dunne, and Larissa Meinicke. Linking unifying theories of program refinement. *Sci. Comput. Program.*, 78(11):2086–2107, 2013.
20. J. He. HRML: a hybrid relational modelling language. In *IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)*, 2015.
21. Jifeng He. From CSP to hybrid systems. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, pages 171—189. Prentice Hall, 1994.
22. E. C. R. Hehner. Predicative programming, parts 1 & 2. *Communications of the ACM*, 59:134–151, 1984.
23. T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
24. T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

25. F. Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *Proc. 6th NASA Formal Methods Symposium (NFM)*, volume 8430 of *LNCS*. Springer, 2014.
26. G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. 22nd Symp. on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.
27. Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for Hybrid CSP. In Kazunori Ueda, editor, *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 – December 1, 2010. Proceedings*, pages 1–15. Springer Berlin Heidelberg, 2010.
28. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
29. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. *Formal Aspects of Computing*, 25(1):133–158, January 2013.
30. André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *Journal of Logic and Computation*, 20(1):309–352, 2010.
31. André Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.
32. Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 20–38. Springer-Verlag, 2006.
33. Xinbei Tang and Jim Woodcock. Travelling processes. In Dexter Kozen, editor, *7th Intl. Conf. on Mathematics of Program Construction (MPC)*, volume 3125 of *Springer*, pages 381–399. Springer, 2004.
34. Kun Wei. Reactive designs of interrupts in Circus Time. In *10th Intl. Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 8049 of *LNCS*, pages 373–390. Springer, 2013.
35. Kun Wei, Jim Woodcock, and Ana Cavalcanti. *Circus Time* with reactive designs. In *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27–28, 2012, Revised Selected Papers*, pages 68–87, 2012.
36. M. Wenzel et al. *The Isabelle/Isar Reference Manual*, February 2016. http://isabelle.in.tum.de/doc/isar-ref.pdf.
37. M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In *20th Intl. Conf. on Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 352–367. Springer, 2007.
38. Jim Woodcock. Engineering UToPiA—formal semantics for CML. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods — 19th International Symposium, Singapore, May 12–16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2014.
39. Jim Woodcock and Ana Cavalcanti. A tutorial introduction to designs in Unifying Theories of Programming. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4–7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer, 2004.
40. F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, 2016. To appear.
41. Chaochen Zhou and Michael R. Hansen. *Duration Calculus—A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004.