









UTP, *Circus*, and Isabelle

Jim Woodcock¹, Ana Cavalcanti¹, Simon Foster¹, Marcel Oliveira³,
Augusto Sampaio², and Frank Zeyda⁴

¹ The University of York, York, UK

{jim.woodcock, ana.cavalcanti, simon.foster}@york.ac.uk

² Universidade Federal de Pernambuco, Recife, Brazil

acas@cin.ufpe.br

³ Universidade Federal do Rio Grande do Norte, Natal, Brazil

marcel@dimap.ufrn.br

⁴ Guadalajara, Mexico

<https://www-users.york.ac.uk/~jw524/>,

<https://www-users.york.ac.uk/~alcc500/>,

<https://www-users.york.ac.uk/~sf786/>, <https://dimap.ufrn.br/~marcel/>,

<https://www.cin.ufpe.br/~acas/>, <https://www.linkedin.com/in/frank-zeyda/>

Abstract. We dedicate this paper with great respect and friendship to He Jifeng on the occasion of his 80th birthday. Our research group owes much to him. The authors have over 150 publications on unifying theories of programming (UTP), a research topic Jifeng created with Tony Hoare. Our objective is to recount the history of *Circus* (a combination of Z, CSP, Dijkstra’s guarded command language, and Morgan’s refinement calculus) and the development of Isabelle/UTP. Our paper is in two parts. (1) We first discuss the activities needed to model systems: we need to formalise data models and their behaviours. We survey our work on these two aspects in the context of *Circus*. (2) Secondly, we describe our practical implementation of UTP in Isabelle/HOL. Mechanising UTP theories is the basis of novel verification tools. We also discuss ongoing and future work related to (1) and (2). Many colleagues have contributed to these works, and we acknowledge their support.

Keywords: *Circus* · CSP · Isabelle/HOL · Isabelle/UTP · refinement calculus · UTP · Unifying theories of programming · He Jifeng · Z

1 Dedication

Jim Woodcock met He Jifeng in Oxford in the early 1980s. Jim was working for GEC Hirst Research Centre and regularly visited Oxford to teach courses for industry. He collaborated with Jifeng in teaching Z, program refinement, and CSP. This collaboration continued after Jifeng moved to the United Nations University in Macau, where Jim became a visiting professor. In 2013, with Zhiming

F. Zeyda—Independent Researcher.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 19–51, 2023.

https://doi.org/10.1007/978-3-031-40436-8_2

Liu and Huibiao Zhu, Jim helped to celebrate Jifeng’s 70th birthday, organising a collection of essays [64], an international training school on UTP [65], and a colloquium on theoretical computer science [63]. Jim’s research groups at the Universities of Oxford, Kent, and York took their intellectual basis from the sound foundations of the Z notation, data refinement, CSP, functional programming, and unifying theories of programming. Jifeng made significant contributions in each of these areas. We were delighted when Jifeng accepted an Honorary Doctorate conferred by the University of York at a ceremony in Beijing on 17 April 2010. All authors are grateful for the inspiration, good taste, and mathematical excellence he provided and continues to provide, which greatly influences our work.

Thank you, Jifeng!

2 Introduction

We describe in this paper our work since 2000 inspired by He Jifeng and Tony Hoare and their unifying theories of programming (UTP). We watched the origins of UTP. Jim recalls enthusiastic but puzzling meetings over lunch in the common room in Oxford with Tony and Jifeng mysteriously discussing *ok'*, *wait'*, and the tradeoff between different fixed points. Tony left Oxford in 1998 and Jifeng shortly afterwards. The UTP book [59] was launched at Tony’s retirement symposium, where we gave a copy to every participant. Tony and Jim gave a short course on UTP at the symposium. This was the origin of Jim’s long-standing course. We describe UTP and its development in [113].

UTP is particularly well suited as a basis for writing and reasoning about heterogeneous models, capturing various aspects of a system: data, reaction, time, architecture, and so on. In UTP, we found the theoretical basis for cyber-physical systems (CPS). In a CPS, computer-based algorithms control and monitor a physical device; potentially, humans interact with networked physical devices. A CPS senses and changes the physical world. Modelling a CPS requires heterogeneous notations: discrete programming models for control; continuous models for physical dynamics, including hydraulics, mechatronics, and others; protocols for human interaction; and continuous and probabilistic models for assumptions about an uncertain environment. The diversity of the heterogeneous semantics required for CPS requires a unifying theory of semantics: UTP.

This paper describes our work, past, present, and future, on *Circus*, a multi-paradigm modelling language. *Circus* is a concrete realisation of UTP.

In Sect. 3, we discuss our research work on *Circus*. It makes the choices for designing a language suitable for UTP semantics with parsers, type-checkers, static and dynamic analysers, model checkers, theorem provers, and code generators. In Sect. 4, we discuss our implementation of UTP and *Circus* in the Isabelle/UTP theorem prover. Additional exciting projects that use *Circus* but did not involve our research group are briefly described in Sect. 5. Section 6 describes ongoing and future directions for research on *Circus*, Isabelle/UTP and their applications, and Sect. 7 summarises our work.

3 *Circus*

Two activities needed to model a system are formalising its data model and behaviour. Model-oriented languages like Z [62, 100, 107, 110] describe state-based aspects, and process algebras like CSP [58] describe behavioural patterns. We add a third dimension for system development: a refinement calculus [75]. Combining these three aspects motivates *Circus*, where a system process groups data and control constructs and the behaviours of all implementations are specified.

Dijkstra, Back, Morris, and Morgan used predicate transformers [31] as the basis of semantic models for imperative refinement calculi [7, 75, 76]. Hoare and Roscoe use different models as the basis of theories of refinement for CSP, the failures-divergences model [58, 90, 91]. Fischer surveyed some of the work that combines the two approaches [38]. Fischer and Smith [39, 99] both provide a failures-divergences model for Object-Z classes to present the semantics for combinations of Object-Z and CSP. Although they consider data refinement for these combinations, they do not give refinement laws.

Woodcock et al. [119] use the failures model to give behavioural semantics to abstract data types. The semantics of *Circus* requires a model combining the notions of refinement for CSP and imperative programs. UTP [59] is a framework that makes this combination possible by unifying the programming discipline across many different computational paradigms.

The semantic setting provided by UTP is the theory of alphabetised relations. Interesting sub-theories are built by defining mappings corresponding to healthiness conditions capturing different aspects of the sub-theory. Hoare and He [59] first create a sub-theory of precondition-postcondition pairs within the relational calculus. This is the theory of designs (see [117] for a tutorial introduction to designs and Harwood et al. [56] for an introduction to Galois connections). Next, they build a theory of reactive processes that is disjoint from the theory of designs. Finally, they use reactive healthiness conditions to embed designs within the theory of reactive processes. The result is the theory of CSP processes (see [24] for a tutorial account of this embedding and connections between it and Roscoe’s semantics based on the failures-divergences model).

In what follows, we survey the main contributions that led to the design, formalisation, extension, and application of *Circus*.

3.1 A Concurrent Language for Refinement [108, 114]

We start by describing the origin of *Circus*. In 2000, Jim Woodcock visited Ana Cavalcanti in Brazil while on sabbatical from Oxford. They formed a reading group with Augusto Sampaio to study Hoare and He’s textbook on UTP [59]. Chapter 8 describes a unifying theory for communication in process algebras. The book considers ACP, CCS, CSP, and the data-flow language SDL.

In the reading group, we were inspired particularly by Theorem 8.2.2 in the book (Closure of CSP Processes). It states two properties: (1) The UTP theory for CSP processes defines a complete lattice that is closed under sequential composition. (2) The lattice also contains $\mathbf{R}(x := e)$, where x is any list of stored program variables, e is a corresponding list of well-defined expressions, and \mathbf{R}

is the healthiness function for reactive processes (see [59, Theorem 8.0.2, p. 208] for the definition of the reactive healthiness conditions). The proof of the first conclusion follows from the following fact:

$$P \text{ is a CSP process iff } P = \mathbf{R}(\neg P[\text{false}, \text{false}/\text{wait}, \text{ok}'] \vdash P[\text{false}, \text{true}/\text{wait}, \text{ok}']) \quad [\dagger]$$

The predicate $P[\text{false}, \text{false}/\text{wait}, \text{ok}']$ describes the divergences of the CSP process P . The process P has been properly started: $\text{wait} = \text{true}$ explicitly and $\text{ok} = \text{true}$ implicitly, since we are in the precondition: before the \vdash . We select divergence: $\text{ok}' = \text{false}$. The complement $\neg P[\text{false}, \text{false}/\text{wait}, \text{ok}']$ describes the situations where P does not diverge. In the postcondition, after \vdash , the predicate $P[\text{false}, \text{true}/\text{wait}, \text{ok}']$ describes the conditions under which P reaches a stable state: $\text{wait} = \text{true}$ and $\text{ok}' = \text{true}$. It describes the stable failures of P .

Two things snagged our attention here. First, the property marked $[\dagger]$ states that every CSP process can be expressed as a reactive design. Every CSP process behaves as described by a reactive assumption-commitment pair. In our subsequent work on *Circus*, we used this property to give uniform, specification-oriented semantics to the operators of *Circus*, establishing a way of specifying *Circus* processes as contracts. Second, the reactive assignment reminded us that the UTP semantics for CSP is *state-rich*. The UTP semantics of CSP describes the representation of states that react to the same input differently depending on the current state value recorded in program variables.

So the reading group asked itself the question:

What if we used the Z notation to specify abstract data types to accompany CSP definitions of processes?

This question is the origin of the *Circus* notation.

We presented *Circus* for the first time at a workshop at Trinity College Dublin [114]. Our formulation of the language gave a calculational approach to writing programs that are similar to *occam* [60] and *Handel-C* [67]. Our paper [114] describes the language, the rationale for its design, and a case study in its use: a reactive ring buffer with a cached head, which became a famous case study for showing off the features of *Circus*. The ring acts as a bounded buffer in the formal sense that it has the following properties:

1. The ring is a FIFO queue.
2. If the ring has spare capacity, it cannot refuse an input.
3. If the ring is not empty, it cannot refuse to output.

Each cell in the buffer is modelled as an active process. To ensure no (perceived) refusal of output (3), we cache the head of the buffer. This avoids the delay required to fetch the head so that it is immediately available.

3.2 The Steam Boiler in *Circus* [115]

Another well-known case study is the steam-boiler problem, which has become a standard benchmark in modelling and verification. It was first proposed by

Bauer [12] and subsequently popularised by Jean-Raymond Abrial as the subject of a Dagstuhl workshop [2]. The workshop proceedings contain the problem description and 22 solutions. Abrial’s solution is published separately [1].

The problem is to program the control system for a steam boiler. The control software exists within a physical environment with the following elements: (1) the steam boiler; (2) a sensor to detect the level of the water in the boiler; (3) a valve to evacuate the boiler; (4) a sensor to measure the quantity of steam being produced; (5) four pumps supplying the boiler with water; (6) four pump controllers; (7) an operator’s desk; and (8) a message transmission system. Our solution to the problem consists of four processes operating in parallel. (1) The *Timer* ensures the cycle begins every five seconds. (2) The *Analyser* inputs messages from the physical units and analyses their content. (3) Once the analysis is complete, it offers an information service to the *Controller*, which decides on the actions to be taken. (4) It generates outputs for the *Reporter*, which offers a reporting service to the *Controller* by gathering its outputs and packaging them for dispatch to the physical units. It then signals the completion of the cycle.

Our solution structure was guided by the drive to efficiently use the FDR model checker [53]. We had to overcome two obstacles: the state explosion problem and the use of loose constants. The latter complicates model checking because loose constants must be given specific values that define a concrete finite model. An argument is then required to extrapolate these specific values to arbitrary ones (a small model theorem). The steam boiler depends on several of these constants. Any practical instantiation leads to a massive number of states.

Our solution separates the *Controller* and its finite-state machine from the *Analyser* and the rich state it constructs from input message history. The *Analyser* digests the incoming messages and makes this digest available to the *Controller* as abstract events. This makes the *Controller* amenable to fully automatic model checking using FDR. Significantly, the *Analyser*’s abstract events correspond to concepts in the requirements, so they help validate the *Analyser*’s behaviour. Extrapolation from the abstract behaviour of the *Controller* to the concrete realities of the requirements is provided by the *Analyser*. It is like a retrieve function from the concrete details of the state to an abstract interpretation of those details, in the sense of data refinement [110].

3.3 The Semantics of *Circus* [116]

The semantics of *Circus* provides a model for processes and their components. In [116], we use a Z specification to describe the semantics of *Circus* processes and of *Circus* actions, which have an imperative state, as relations. The process model is a Z specification, and the action model is a Z schema. We used Z as a concrete notation for UTP’s relational calculus because we could parse and type-check it and prove various consistency results.

Circus includes support to define imperative assignments, conditionals, loops, and the reactive behaviour of communication, parallelism, and internal and external choice. All combinations of model-based formalisms and process algebras that had been published before we defined the semantics of *Circus* describe

concurrent programs as communicating abstract datatypes. For example, this is the case with CSP_Z [38] and $CSP \parallel B$ [95]. Communicating abstract datatype is a valuable but limited design pattern. We took a different approach and did not identify events with datatype operations. The result is a programming language suitable for developing concurrent programs in a more general style.

Our goals in designing the semantics of *Circus* were: (1) ease of use for those familiar with Z and CSP; (2) encapsulation of the process model; and (3) the possibility of reusing existing theories, techniques, and tools. We had to decide how best to formulate the semantics. Imperative refinement calculi like those of Back [7], Morgan [75], and Morris [76], are normally given predicate transformer semantics. Theories of refinement for CSP are based on the failures-divergences model [58,90]. A connection between weakest preconditions and CSP exists [74], and a sound and complete refinement theory has been developed based on it [120]. We use a fourth approach: UTP, where both state and communication aspects of concurrent systems are integrated with a state-based failures-divergences model described pointwise. This leads to a simple and elegant definition of refinement and a sound foundation for refinement calculi.

3.4 Refinement in *Circus* [26,27,94]

Having set out the semantics of *Circus*, our next step was to define its refinement relation [94]. Each *Circus* process has a state and accompanying actions that define both internal state transitions and changes in control flow during execution. We explained the meaning of refinement for processes and their actions and proposed a sound data refinement technique. Refinement laws for CSP and Z are directly relevant and applicable to *Circus*, but our focus was on new laws for processes that integrate state and control. We presented new results about the distribution of data refinement through CSP operators adopted in *Circus*.

We illustrated our ideas with the development of a distributed system of cooperating processes. We proposed a refinement approach whose typical starting point is a centralised specification of an application. The development process moves towards a distributed solution. The approach is supported by two families of laws (for algorithmic and data refinement) that allow the incremental splitting of *Circus* processes using parallelism. The overall approach is illustrated by a case study (the reactive buffer again) that, although simple, is interesting enough to demonstrate the proposed strategy in all its relevant details.

A *Circus* system describes a set of processes. Each process encapsulates a local state and has its reactive behaviour defined by actions in that state. In [26], we present refinement laws to support the development of these actions from more abstract descriptions. These laws form the basis of a systematic development strategy for *Circus* based on formal refinement, addressing all the language's constructs. It complements the work in [94] by proposing laws of actions, including the laws of CSP [58,91] and of ZRC [28], a refinement calculus for Z .

In addition, since *Circus* allows us to specify actions using a mixture of Z schemas and CSP constructs, we require new laws. For example, there are novel laws to introduce parallelism and external choice from Z schema expressions.

These laws are added to a comprehensive set of refinement laws of CSP to support program development in *Circus*. The work extends the forward simulation laws proposed for *Circus* [94] to address all the action operators of *Circus*. It illustrates how these laws can be proved from the semantics of *Circus*. Parts of the development of the distributed cached-head ring buffer from its centralised specification are used to illustrate the laws of actions and forward simulation.

In [27], we present a refinement strategy for *Circus*. The strategy unifies the theories of refinement for processes and their constituent actions and provides a coherent technique for stepwise refinement of concurrent and distributed programs involving rich data structures. This kind of development is carried out using *Circus*'s refinement calculus. We describe some of its laws for the simultaneous refinement of state and control behaviour, including splitting a process into parallel components. We illustrate the strategy and the laws using a case study that shows the complete development of a distributed program.

3.5 Predicate Transformers in the Semantics of *Circus* [29]

One of the main objectives of the *Circus* work is the definition of refinement methods for concurrent programs. The original semantic model for *Circus* is defined using UTP, expressed in Z. In [29], we present equivalent semantics based on predicate transformers. With this new model, we provide an adequate basis for formalising refinement and verification-condition generation rules.

This new framework makes it possible to include logical variables and angelic nondeterminism in *Circus*, neither of which are straightforward in the relational setting. The consistency of the relational and predicate transformer models gives us confidence in their accuracy. Only much later did we study angelic nondeterminism in a relational setting [25]. The work to define a UTP theory to study *Circus* processes and angelic nondeterminism was led by Pedro Ribeiro [85–87].

We present in [29] a new predicate transformer: the weakest reactive precondition. It characterises the weakest precondition that guarantees that a given condition holds in all later observable, not necessarily final, states of a reactive program. We define the weakest reactive precondition of a unifying theory relation that defines a reactive system. From this, we calculate the weakest reactive precondition semantics for *Circus*. This new semantic model is a convenient step towards the complete justification of our extension to an existing refinement calculus for Z [28] that includes all *Circus* constructs.

Roscoe and Hoare [92] present laws that completely characterise *occam* and that are cast in terms of the *occam*'s denotational semantics [89], although no proof of equivalence was carried out. The laws presented in that work are equality-based algebraic semantics. Unlike our work, they are not intended to support the development of programs by refinement.

3.6 A *Circus* Semantics for Ravenscar Protected Objects [6]

Burns et al.'s Ravenscar profile [14] is a subset of the Ada 95 tasking model [10]. The Ravenscar profile does not allow Ada's rendezvous construct for task com-

munication. Instead, tasks in Ravenscar communicate through shared variables, usually encapsulated inside protected objects. This makes protected objects fundamental building blocks in Ravenscar programs, providing a safe mechanism for accessing the shared data between various tasks.

The Ravenscar profile is intended to be certifiable and deterministic, to support schedulability analysis, and to meet tight memory constraints and performance requirements. With Atiya and King [6], we give semantics to protected objects using *Circus* and prove several of its essential properties: consistency, determinism, deadlock-freedom, livelock-freedom, totality, and non-stopping behaviour. This was the first time that these properties had been verified. Interestingly, all the proofs are conducted in Z , even those concerning reactive behaviour. A compliance notation for concurrent systems [5] provides a cost-effective technique for verifying Ravenscar programs based on this formal semantics.

Lundqvist et al. [68] provide an alternative formal model of Ravenscar’s protected objects in UPPAAL [13]. Their model deals specifically with the timing of calls to protected objects. Model checking is used to verify the protected object model considering only a few tasks: three. No statement was made about the model’s validity for more tasks. Our proofs are valid for any number of tasks.

3.7 Using *Circus* for Safety-Critical Applications [111]

In [111], we illustrate the use of *Circus* via the example of the steam boiler discussed in Sect. 3.2. We focus on an interesting semantic gap between synchronisation in CSP and, therefore, *Circus*, and in programs: a kind of abstract event. In CSP, an abstraction is sometimes used in which atomic synchronisations can be system-wide, between many processes, rather than being restricted to only two participants. In [111], we deal with a simple instance of this phenomenon of multi-synchronisation, which shows the power of *Circus*’s calculational approach to reasoning about reactive systems via refinement of abstract models.

We base our model of the steam boiler controller here on O’Halloran’s description [78] that expresses its functional requirements as firing rules. These are in the form **if a then b** , where event a enables event b , subject to environmental constraints. The implicit inference engine defined by these firing rules is non-monotonic, as it must forget previously inferred facts as the system evolves. The result is a valuable design pattern for synthesising reactive controllers.

A suitable language to implement this model as a controller for an actual steam boiler is *occam* [60], given its close relationship with CSP. We might likely choose Communicating Sequential Processes for Java (JCSP), a Java class library that implements CSP processes and process combinators [106]. We immediately, however, face the semantic gap mentioned above. CSP allows the synchronisation of events between many processes, but *occam* and JCSP restrict this, for efficiency reasons, to just two participants. In our paper [111], we apply *Circus*’s refinement calculus to bridge this semantic gap.

In this work, we consider a collection of parallel processes indexed over I , each repeatedly executing some individual transaction, represented by the event

$t.i$, with $i \in I$ and synchronising the transactions by alternating them with a globally shared event m . The *Circus* process models this:

$$\left(\parallel_m i : I \bullet (\mu X \bullet m \rightarrow t.i \rightarrow X) \right) \setminus \{m\}$$

Every process participates in the multi-way synchronisation on m , whereas only the i -th process participates in the independent event $t.i$. The event m is hidden from the environment, so crucially for this development, we know the identities of all of m 's participants. If the membership were dynamic, then we would need to develop a protocol to manage its membership.

We use *Circus*'s refinement calculus to derive a protocol equivalent to this system of parallel processes, but where there is no multi-way synchronisation. Our first step is to convert the i -th process into an action system [8]. We then re-introduce parallelism to create a simple protocol that synchronises transactions. It is now at the code level of *occam* or JCSP.

A more interesting problem occurs when the multi-way synchronisation is part of an external choice, and our solution above is not applicable in such a situation. We have calculated efficient two-phase commit protocols to deal with these synchronisation patterns. Although these programs are much more complex than the one calculated in this paper, the same development strategy is used. The abstract program is reduced to a normal form, which contains no multi-way synchronisations since it is sequential. This normal form is then partitioned into new parallel processes that implement a protocol for synchronising individual transactions. This approach is later adopted in [50] in the context of an automated strategy for translation from *Circus* to JCSP.

3.8 Formal Development of Industrial-Scale Systems in *Circus* [81]

In [80], we present the use of the *Circus* refinement strategy to derive a concrete distributed fire-control system from an abstract centralised *Circus* specification. This real-world system is one of the most significant case studies on the *Circus* refinement strategy [27] and translation rules [81].

The fire-control system considers two building areas, each divided into two zones. Two extra zones are used for detection only. Fire detection happens in a zone, and a gas discharge may occur in the area that contains that zone. The system includes a display panel with lamps to indicate whether the system is on or off, system faults, whether a fire has been detected, whether the alarm has been silenced, and the need to replace actuators and gas discharges. The system can be in one of three modes: manual, automatic, or disabled.

In manual mode, an alarm sounds when a fire is detected, and the corresponding detection lamp is lit on the display. The alarm can be silenced, and the system returns to normal when the reset button is pressed. In manual mode, a gas discharge needs to be manually initiated. In automatic mode, fire detection is followed by the alarm being sounded; however, if a fire is detected in the second zone of the same area, the second stage alarm is sounded, and a countdown starts. When the countdown finishes, the gas is discharged, and the circuit fault

lamp is illuminated in the display; the system mode is switched to disabled. In disabled mode, the system only indicates the need to replace the actuators, identify relevant faults, and reset. The system returns to its normal mode after the actuators are replaced, and the reset button is pressed.

The motivation for the fire-control system refinement is the distribution of the control for efficiency. In [80], we use the refinement strategy in [23] to develop a concrete distributed system using three refinement iterations: the first one splits the system into an internal controller and a controller for the areas. In the second iteration, the internal controller is subdivided into two further controllers, separating a controller just for the display. Finally, the third iteration splits the controller's areas into individual controllers for each area.

The result of refining a *Circus* specification is a *Circus* program written in a combination of CSP and guarded commands. We, therefore, need a link between *Circus* and a practical programming language to implement this program.

In [81], we present rules to translate *Circus* programs to Java programs that use JCSP (see [106] and the discussion in Sect. 3.7). These rules can be used as a complement to the *Circus* algebraic refinement technique or as a guideline for implementation. They link the results of refinement in the context of *Circus* and a practical programming language in current use. The rules can also be used as the basis for a tool that mechanises translation [11, 50]. In [81], we demonstrate the application of the rules using the industrial fire-control system.

The main objective of that work was to provide a translation strategy for implementing *Circus* programs in a widely used language. Using the JCSP [105, 106] library and a rule-based approach ensures that the obtained programs can be traced back to the *Circus* model. The rules justify and generalise our development of the fire-control system. With this work, we provide empirical evidence of the expressive power of *Circus* and that the refinement strategy in [27] and the translation to Java apply to industrial systems.

3.9 A Denotational Semantics for *Circus* [82, 83]

Although usable for reasoning about *Circus* specifications, the semantics in [116] is not appropriate to prove properties of *Circus* itself. This is because it is a shallow embedding in which *Circus* constructs are defined as a Z specification. Yet another language is used as a metalanguage to define the semantics. The main drawback is that we can not use shallow embedding to prove the laws of *Circus*'s distinguishing development technique. In [82], we present an alternative: a definitive reference for the denotational semantics using UTP.

We redefined the *Circus* semantics. We mechanised it using ProofPower-Z [4], a commercial HOL-based theorem prover for Z. We implemented the UTP theories needed for the semantics of state-rich CSP (relations, designs, reactive processes, and the CSP healthiness conditions) [84]. Our semantics for *Circus* is then given using reactive designs. We proved over 90% of the 146 proposed refinement laws. These proofs range over the structure of the language and include all the data simulation laws. Their proofs can be found in [79].

We used a simple strategy to prove $P = Q$ or $P \sqsubseteq Q$. (1) Flatten P to a single reactive design $\mathbf{R}(pre_P \vdash post_P)$. (2) Flatten Q to a single reactive design $\mathbf{R}(pre_Q \vdash post_Q)$. (3) Use lemmas and theorems from the ProofPower UTP library and predicate calculus to transform the first reactive design into the second one (in case of refinement, an inverse implication is the required result). Flattening the programs involves definitions and theorems that transform program structures into a single reactive design. For instance, if P is the sequence $P_1; P_2$, the following lemma transforms it into a single reactive design.

Lemma 1.

$$\begin{aligned} & \mathbf{R}(P_1 \vdash Q_1); \mathbf{R}(P_2 \vdash Q_2) = \\ & \quad \mathbf{R}(P_1 \wedge \neg((okay' \wedge \neg wait' \wedge Q_1); \neg P_2) \\ & \quad \vdash \\ & \quad \quad ((wait' \wedge Q_1) \vee (okay' \wedge \neg wait' \wedge Q_1); Q_2)) \end{aligned}$$

for P_1 not mentioning dashed variables and P_1, Q_1, P_2 , and Q_2 all **R2**-healthy.

The result of our mechanisation is a definitive reference for the denotational semantics of *Circus* using UTP and reactive designs.

Finally, we note that *Circus* also has an operational semantics [51, 118]. In [51], there are considerations on a formal link to the denotational semantics. Furthermore, as we have already explained, the algebraic laws have been proved from the denotational semantics, establishing the usual links suggested by UTP.

3.10 Time and Synchronicity in *Circus* [16, 97]

*Circus*Time Action (CTA) is a timed version of *Circus*, explored by Sherif and others, including He Jifeng [96, 97]. It introduces discrete-time slots of event sequences. CTA provides a two-tier view of history. The top-level records history as a sequence of time slots. The bottom-level records history as an event sequence within a given slot. This is reminiscent of super-dense time, an important tool for modelling simultaneity in discrete-event simulations. The slots model events separated in time, whilst each slot models simultaneous but ordered events.

We worked with Andrew Butterfield on a synchronous version of *Circus*. Our work in [16] takes inspiration from CTA and is compatible with the general structure of the *Circus* language. We develop a generic framework of UTP theories for describing systems whose behaviour is characterised by regular (top-level) time slots. The slotted-*Circus* framework is parametrised by how event histories are observable within a slot (the bottom level). We instantiate this bottom-level history in a variety of ways: as simple traces or multisets of events or as the more complex micro-slot structures used in our operational semantics for Handel-C (a high-level programming language that targets low-level hardware, most commonly used in the programming of FPGAs) [18].

One of the original motivations behind this work was to re-cast existing semantics for Handel-C into the UTP framework so that *Circus* can be used

as a specification language. Using this time-slot model, the Handel-C denotational [17] and operational semantics are defined. Still, the slot structure has varying complexity, depending on which language constructs we wish to support. The slotted-*Circus* framework is a foundation for formulating the common parts of these models, making it easier to explore the key differences.

3.11 The Miracle of Reactive Programming [112]

UTP uses Tarski’s relational calculus, with theories defined by complete lattices of predicates ordered under refinement. Roscoe’s semantics for CSP uses a complete partial order (CPO) [90]. So UTP offers an exciting addition: the reactive miracle, the top of the lattice. In [112], we present two simple properties of reactive miracles: prefixing a miracle with an event and offering an external choice between a process and a miracle. Both processes have interesting properties: each violates an essential axiom of the standard failures-divergences model for CSP. Of course, that is why the reactive miracle is not in Roscoe’s CPO.

All three UTP theories involved in modelling CSP processes are complete lattices rather than the CPOs of the standard models for CSP. As complete lattices, they each have a top element. The top of the design lattice is the familiar miracle from the refinement calculus: $w : [true, false]$ [75]. This design is always guaranteed to terminate if it is started (precondition *true*), and when it does terminate, it achieves the impossible (it makes *false true*).

Morgan demonstrates a specific application of miracles [73]. He shows that a miracle can enable conditional data refinement even when the condition involves concrete variables. Some reasoning is then needed at the concrete level to eliminate the miracle, which can never be executed. Morgan illustrates another use for miracles: a naked guarded command can be given weakest precondition semantics. For guard G , command *com*, and postcondition α , the weakest precondition for the guarded command $\mathbf{wp}(G \rightarrow com, \alpha)$ is $G \Rightarrow \mathbf{wp}(com, \alpha)$. We note that a guarded command does not satisfy the Law of the Excluded Miracle [31]: $\mathbf{wp}(com, false) = false$; for example, $\mathbf{wp}(G \rightarrow com, false)$ is $G \Rightarrow \mathbf{wp}(com, false)$, which is different from *false*. In [74], Morgan uses this definition to give semantics to an action system [8] (see also [120]).

The tops of the reactive and the CSP lattices in UTP were unexplored when we wrote [112]. The reactive miracle is $\top = \mathbf{R1}(true \vdash wait \wedge \mathbb{II})$. This is reactive-healthy but infeasible (miraculous) if properly started. We proved the following result for an external choice between a prefixed process and a miracle:

$$a \rightarrow Skip \sqcap \top = (true \vdash (\mathbb{II} \triangleleft wait \triangleright \neg wait' \wedge tr' = tr \wedge \langle a \rangle \wedge v' = v))$$

This process terminates immediately, having performed the event a . There is no state in which the process is waiting for the environment to perform a : it happens *instantly*. This makes the event a urgent.

In [112], we explore some applications of miracles. We show how to make two events a and b simultaneous, but ordered: we prune away the state between a and b . Next, we show how to implement deadlines. For example, if b must occur

within 10 time units, we can model this using a new deadline operator: we write b **deadline** 10 $\hat{=} (b \rightarrow \text{Skip}) \triangleright_{10} \top$, where \triangleright_{10} is the timeout operator. In this process, there are no states 10 time units from initiation in which b has not happened. This captures a very strong requirement: there is no alternative to meeting the deadline. Further applications of miracles are explored in [103, 104].

Reactive miracles have proved indispensable to provide a sound semantic basis for real-time extensions of *Circus*. A real-time variant of *Circus* has, for instance, been used to give an architectural infrastructure model [72] of Safety-Critical Java (JRS 302) [66]—a subset of Java tailored for the engineering of safety-critical real-time systems. Nelson already realised that, despite their unimplementability, miracles are useful in refinement-based systems development, much like complex numbers in solving differential equations. Our work rediscovers and reiterates this claim in the context of reactive programming in general, and *Circus* in particular, with UTP giving us the right framework and vocabulary to make this integration as smooth as possible.

4 Isabelle/UTP

We describe Isabelle/UTP¹, our practical implementation of UTP that can be used to mechanise UTP theories and turn them into verification tools. We cover the history of Isabelle/UTP and motivate the design decisions behind its development: in each section, we account for a major step in the Isabelle/UTP design as it evolved. Isabelle/UTP was born out of necessity to support UTP-based software engineering, and this continues to be our motivation to this day.

4.1 Beginnings

Isabelle/UTP [46] is a shallow embedding of the UTP in Isabelle. Its development began in 2012, during the COMPASS project². Nevertheless, Isabelle/UTP is a natural development of previous UTP mechanisations, notably by Marcel Oliveira [84] and Abderrahmane Feliachi [32, 36] (with Burkhart Wolff).

COMPASS created a sophisticated toolset for modelling and verifying “systems of systems”. We developed a modelling language, CML (COMPASS Modelling Language), with formal UTP semantics, a task led by Jim Woodcock. Thus, an applicable verification tool for UTP was needed. Simon Foster’s task was to develop a theorem prover for UTP and CML based on Isabelle/HOL.

As envisioned, this tool needed to combine two important characteristics. On the one hand, it needed to provide the fidelity necessary to express refinement laws, including side conditions, which often imposed syntactic constraints. On the other hand, it needed to be suitable for scalable verification. Whilst these had been separately achieved in the mechanisation of Oliveira and Feliachi, they had not been achieved in either work. Oliveira’s mechanisation [84], as a relatively

¹ Isabelle/UTP Website: <http://isabelle-utp.york.ac.uk>.

² Comprehensive Modelling for Advanced Systems of Systems, EU FP7 Project 287829.

deep embedding, had fidelity but lacked the automation necessary to make it scalable. Feliachi’s mechanisation [32,36] had automation and scalability as a shallow embedding but could not express syntactic side conditions.

4.2 Laws and Side Conditions, and the Deep Model

We consider the well-known assignment commutativity law:

$$(x := e; y := f) = (y := f; x := e) \text{ provided } x \neq y, x \notin fv(f), y \notin fv(e)$$

To express this law, as written, we need to (1) compare different program variables and (2) check the variables mentioned in an expression. However, a function like fv , which determines the free variables of an expression, is *meta-logical* since it allows us to make arguments based on the syntactic structure of a term. It exists in Isabelle and most other provers but as a function in Isabelle/ML inaccessible from HOL. This is important because if fv were an Isabelle function, then equality would cease to be useful, as we could not, for instance, prove that $x \cdot 0 = 0$ (for $x \in \mathbb{R}$), since $fv(x \cdot 0) = \{x\} \neq \{\} = fv(0)$.

At the same time, formal methods are awash with laws that use such side conditions. Another example is the frame rule from separation logic:

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \text{ mod}(C) \cap fv(R)$$

This likewise requires that we calculate the free variables in R and the set of variables that command C modifies. We seem to have hit a roadblock—we cannot have fv and similar syntax functions without breaking our logic.

However, Oliveira [84] discovered a neat solution. He created a function $UnrestVar : REL_PRED \rightarrow \mathbb{P} NAME$, which calculates the set of names (i.e. variables) that a predicate does not depend on, i.e. those that are “unrestricted”. Unlike fv , $UnrestVar$ is a *semantic* rather than a syntactic function. It does not compute the syntactically present names but those that have some bearing on the predicate’s meaning. For example, $x \cdot 0$ does not depend on x since it always evaluates to 0. Thus, $UnrestVar(x \cdot 0) = NAME$, since this expression does not depend on any variable: it is semantically equivalent to 0.

It turns out that $UnrestVar$ is sufficient to express the side conditions of our assignment law and similar laws. This function has a much older pedigree: an analogue is found in Tarski’s famous Cylindric Algebra [57], an algebraic basis for first-order logic with equality. In this setting, we can express $UnrestVar$ as the greatest set A of names such that $(\exists A. P) = P$. Quantifying the names in A does not change P because P does not depend on them.

Oliveira’s solution avoids the need for fv . However, there is still a problem because $UnrestVar$ requires that we formalise names and, as later realised in the work of Zeyda [123], types. The problem is that names and types are also meta-logical. If we formalise them, we cut ourselves off from the proof assistant’s

representation of names and types, with a resulting loss of algorithms like α -renaming and type checking. We have to implement these ourselves.

So, when we first developed Isabelle/UTP, we followed Oliveira [84] and Zeyda [123] in building our representation of names, types, and a value universe [45]. We call this a “deep model”, rather than a “deep embedding” because we do not formalise a syntax tree for predicates, just for the underlying value universe. In this approach, a predicate is denoted as a set of functions recording the possible values of the correct type that a variable can take. We must, therefore, formalise names, types, values, and the typing relation.

In HOL, types are bounded by a given cardinal, and so the value universe is technically limited to a strict subset of the possible types constructible in HOL. Thus, we needed to exhibit an explicit injection into our universe whenever we wanted to use a type in a UTP predicate or program. We did find a way of automating it somewhat, but this did not work well. Though we retained the fidelity of Oliveira’s model, we could not match the automation of Feliachi. This became obvious even for small examples in CML. Our technique did not scale to allow model verification. This was all too painfully pointed out in Wolff’s gracious and factual review of our UTP 2014 paper [45].

Nevertheless, whilst we could not support verification, our techniques substantially benefited from Isabelle’s proof automation. Using automated theorem provers, through the *sledgehammer* interface, we proved many more theorems with much less effort than Oliveira with ProofPower-Z. However, we must credit Oliveira’s achievement, for he went remarkably far in mechanising UTP, with some proof scripts running hundreds of lines. We learned valuable lessons, but a new foundation for Isabelle/UTP was needed.

4.3 Lenses

Burkhart pointed out that our value universe injections could be expressed more generically using lenses [40]. This was vital for the next version [49]. Lenses are simple algebraic structures: for a set S of states and V of values, a lens $x : V \Rightarrow S$ is a pair of functions $get : S \rightarrow V$ and $put : V \rightarrow S \rightarrow S$, which obey three intuitive algebraic laws, such as $get(put\ x\ s) = x$. Lenses are ubiquitous in the foundations of computer science; for example, Back and von Wright [7] use a similar algebraic structure to characterise variables.

We have described lenses in Isabelle/UTP at length [46]. We use them to model program variables and their mutations for a given state. Every variable x of type A in a given state space S is allocated a lens $x : V \Rightarrow S$. Lenses can also be used to characterise sets of variables using a combinator $x \oplus y$ that produces a lens of type $A \times B \Rightarrow S$ with a product view. Thus, we can also model a program’s frame (or “footprint”), part of the state space that a program can modify. Crucially, lenses allow us to escape the need to formalise names and types. Each variable name is in the host logic, and its type is given by its view.

Lenses can be semantically compared in various ways, providing a means to express side conditions. We have the independence relation $x \bowtie y$, which means

that x and y refer to different parts of the state, and it is algebraically characterised as the commutativity of the *put* functions. A preorder $a \preceq b$ states that a characterises a smaller region than b ; for example, $x \preceq x \oplus y$. If we consider lenses a and b as “sets”, then \preceq is the subset relation, and \oplus is the set union operator. Finally, we have an equivalence formed by the cycle of \preceq , that is $x \approx y = (x \preceq y \wedge y \preceq x)$, which allows us to characterise laws like $x \oplus y \approx y \oplus x$.

These relations do not compare lenses based on their (meta-logical) names but their semantics. The use of lenses, therefore, allows us to reuse all the host logic facilities for manipulating names. Moreover, this approach avoids the aliasing problem. Even if we have two variables with different names, they will not be independent if they point to the same store region.

UnrestVar also finds an elegant characterisation with lenses. First, we note that program expressions and assertions are modelled as functions as usual in a shallow embedding. An expression operating over store S with type V is a total function $S \rightarrow V$. An expression like $x + y > 5$ can be modelled, using lenses, as $\lambda s. get_x(s) + get_y(s) > 5$, though this translation was facilitated through a deep expression syntax. We can then ask whether such an expression semantically depends on a particular lens. If an expression’s output value does not change when we change a variable, then clearly, there is no dependence on it. We therefore define $x \# e \Leftrightarrow (\forall v s. e(put_x v s) = e(s))$, our version of *UnrestVar*, which tells us that x is unrestricted in e . As shown below, this relation is precisely what we need to characterise the assignment commutativity law and other related laws:

$$(x := e; y := f) = (y := f; x := e) \text{ provided } x \bowtie y, x \# f, y \# e$$

In words, the assignments commute provided that (1) the variables are independent; (2) f does not depend on x ; and (3) e does not depend on y . This law and other “laws of programming” are theorems of our definitions [46]. As we see later, we can also express a variant of the frame rule.

4.4 UTP and Designs

With a scalable foundation for Isabelle/UTP, we could tackle a significant challenge: mechanisation of the reactive-design hierarchy and the *Circus* language with its UTP semantics. During the development of our various versions of Isabelle/UTP, *Circus* served as a baseline, and we had several iterations of the mechanisation of the operators and healthiness conditions.

Upon our lens-based expression model, we developed UTP’s relational calculus. A (potentially heterogeneous) relation in Isabelle/UTP is an expression $S_1 \times S_2 \rightarrow \mathbb{B}$. As in the Z notation, relations typically range over unprimed (x) and primed variables (x'). In Isabelle/UTP, this is achieved using lenses $fst : S_1 \Rightarrow S_1 \times S_2$ and $snd : S_2 \Rightarrow S_1 \times S_2$ that project the pre- and post-states. We distinguish lenses on the “flat state” (S) from those in the relational state ($S \times S$), a distinction implicit in languages like Z. We then proved the relational calculus laws found in the UTP book and related publications. This includes

a detailed account of UTP theories, defined by a set of idempotent healthiness functions, and accompanying theorems, including Knaster-Tarski.

We tackled the design theory from here, allowing us to model relational programs that may exhibit divergence. Our mechanisation raised many questions about the encoding of the design turnstile notation ($P \vdash Q$). Should P and Q be permitted to refer to the *ok* variable? Should P be allowed to refer to only the pre-state or also the post-state? In Isabelle, we can use the type system to impose restrictions like this by construction. The answer to the first question became clear: *ok* is semantic machinery used only by \vdash , so a UTP theory should not touch it when defining concrete specifications.

The answer to the second question is a little less clear since, in the UTP theory of reactive designs, the precondition can refer to the post-state value of the trace tr to express constraints on permitted communications. As a result, Isabelle/UTP has two turnstile operators: $P \vdash_r Q$, with $P : S_1 \times S_2 \rightarrow \mathbb{B}$ and $p \vdash_n Q$, with $p : S_1 \rightarrow \mathbb{B}$. The latter is sometimes called a “normal design” [55], hence the n subscript. The benefit of using this second turnstile operator is that side conditions in many theorems can be avoided, thanks to the type system. This improves the efficiency of verification for normal designs.

4.5 UTP Theories

In more detail, a UTP theory consists of (1) a set of observational variables; (2) a set of healthiness conditions; and (3) a signature for constructing elements of the theory that satisfy the healthiness conditions. Like classes in object-oriented programming, UTP theories are extensible by adding more observational variables and healthiness conditions. These conditions can be seen as invariants.

Feliachi’s encoding of UTP included an elegant approach to encoding alphabets using extensible records. Each UTP theory is allocated its record type, which gives the observational variables as fields. Since alphabets are types, using variables outside the alphabet equates to a type error. Successive extensions of the UTP theory add fields to the alphabet types. Thanks to Isabelle’s polymorphism, functions defined and theorems proved in super-theories are then applicable in sub-theories. For example, a theorem proved in designs is applicable in reactive designs. Moreover, type inference can determine the hierarchy’s most general alphabet of a relation. A downside is that multiple inheritance is unsupported because extensible records are implemented using type variables. Nevertheless, this limitation can be mitigated if the hierarchy is carefully constructed.

A side effect of lenses is that we could easily adapt and expand on this approach. We can express constraints on how relations use observational variables with lenses. For example, the alphabet of a design consists of *ok*, *ok'*, and the program variables and their dashed counterparts. We model this with a parametric alphabet type: a record type $\alpha \text{ des}$ enriched with lenses, where the type parameter α can extend the alphabet with the program variables. The healthiness functions then have types like $(\alpha \text{ des}) \text{ hrel} \rightarrow (\alpha \text{ des}) \text{ hrel}$ of functions over a homogeneous relation whose alphabet contains *ok*. (The actual type is more general because we also support heterogeneous relations.)

With this setup, we can mechanise one of the most complex UTP operators—alphabet extension, which allows us to add (and remove) variables from a relation’s alphabet. Our encoding gives us a special lens: $more_L : \alpha \Rightarrow \alpha des$. It views the part of the alphabet that does not contain ok , which is the program-variable space or any extension of designs. With this lens, alphabet extension becomes a kind of type coercion, such as $\alpha hrel \rightarrow (\alpha des) hrel$, which lifts a relation into the theory of designs. This is how we implement the design turnstile variants \vdash_r and \vdash_n . Alphabet coercions can become complex. Nevertheless, these coercions are invisible in resulting verification tools and improve user experience by making UTP-based programs and models correct by construction.

4.6 Reactive-Design Hierarchy

With a solid theory of designs in place, we proceeded to mechanise reactive designs. This was a significant task, and the reactive-design hierarchy is the most extensive library in Isabelle/UTP, running to about 14,000 lines of Isabelle code. We now give a summary of the main developments.

We mechanised the theory of reactive processes and several variants [42] motivated by the mechanisation and Andrew Butterfield’s **R3h** [15]. As required by the UTP framework, we proved that the healthiness functions are idempotent, monotonic, continuous, and critical closure results. One crucial design decision was to collect the program variables in a single alphabet variable st , which made separating the program space from semantic machinery (encoded via other alphabet variables such as ok , tr , ref , and so on) more accessible.

We also identified two useful subtheories. Reactive relations express possible behaviours using the alphabet variables tr and tr' , recording observed traces, and st and st' . Reactive relations are typically used in postconditions. Reactive conditions have the additional restriction of not referring to st' and having the trace of events $tr' - tr$ prefix closed. Reactive conditions are used in preconditions.

We generalised reactive processes so that tr is drawn from a “trace algebra” [43, 88], a form of a cancellative monoid. The original account has tr as a sequence of events, but sometimes other trace models are desirable, such as piecewise-continuous functions for hybrid systems. It turns out that none of the libraries of laws in [24, 59] depend on tr being a sequence, and trace algebra is a sufficient basis. Having performed the generalisation, Isabelle/UTP reproved all the laws automatically, illustrating the practical benefits of proof automation. If we had done this on paper, it would have taken weeks instead of minutes.

We then created reactive designs by combining designs and reactive processes. After that, the subsequent significant development, from a verification standpoint, was the introduction of the reactive-contract notation $[P \vdash Q \mid R]$, which is a core constructor of the reactive-design theory [42]. It consists of a precondition P , a postcondition R , and a “pericondition” Q , a new concept suggested by Canham [19]. The precondition is a reactive condition that describes initial states and communicating behaviour that the contract is willing to accept. Violation of the precondition leads to divergence. The pericondition Q and postcondition R describe quiescent (or “intermediate”) and terminating behaviours.

In the context of *Circus* and CSP, P corresponds to the complement of the divergences, Q to the set of failure traces, and R to the set of terminating traces. A significant result is that any reactive design can be expressed as a reactive contract.

There are at least two benefits to the use of reactive contracts. Firstly, it allows us to give uniform denotational semantics to all *Circus* operators. Secondly, it will enable us to automate refinement proofs about *Circus* models [44]. We have a refinement law that weakens the precondition and strengthens the peri- and postconditions. This is combined with a calculational proof strategy that allows us to compile any combination of reactive contracts using *Circus* operators into a single reactive contract, which can then be subjected to proof.

4.7 Optimisation and Modularisation

The development of the reactive-designs hierarchy and a *Circus* verification tool served to justify the overall design decisions of Isabelle/UTP. However, several components, notably the expression model, were suboptimal and hampered automation and usability. As we developed Isabelle/UTP, our knowledge of Isabelle/HOL grew, and we improved the design decisions.

Moreover, there was the question of how researchers outside of York could adopt Isabelle/UTP. The original development model was monolithic, with an ever-growing collection of Isabelle theories with many cross-dependencies. There could be little reuse of the components. Isabelle/UTP was a combination of design decisions you either accepted in full or did not.

For example, the library imposes a relational program model ($\mathbb{P}(S_1 \times S_2)$), although this is not universally popular. An alternative is a state transformer model, $S_1 \rightarrow \mathbb{P}(S_2)$, which though mathematically equivalent, has the advantage of forming a monad. In truth, several parts of Isabelle/UTP do not need to be wedded to this program model, notably the lens and expression library.

As a result, we set out on a campaign of optimisation and modularisation. The resulting components, defined as Isabelle libraries, are as follows.

Optics. This is where the theory of lenses is defined and contains several related algebraic structures, notably symmetric lenses and prisms. These give an abstract characterisation to channels analogously to lenses. The **Optics** library also contains user commands, such as **alphabet** and **chantype** to create alphabet and channel types. This library continues to be under active development.

Shallow Expressions. As explained in Sect. 4.3, the original monolithic theory contained an expression model that mimics a deep embedding by introducing constructors for expressions. The motivation was to allow reasoning with the same granularity as a deep embedding. For example, we could encode laws like $(P \wedge Q)[e/x] = (P[e/x] \wedge Q[e/x])$ and $(\exists x. P)[e/y] = (\exists x. P[e/y])$ if $x \bowtie y$. However, this was a substantial overhead since we had to use the simplifier to execute substitutions. It also turned out to be unnecessary since we can directly harness Isabelle’s internal λ -calculus-based substitution mechanisms.

Thus, the [Shallow-Expressions](#) library, instead of having deep abstract syntax, lifts expressions containing lenses (for example, assertions) to pure HOL expressions; for instance, $x + y$ becomes $\lambda s. \text{get}_x(s) + \text{get}_y(s)$ using Isabelle’s syntax translation mechanism to perform the conversion. Nevertheless, as explained in Sect. 4.3, we can still execute substitutions and evaluate unrestriction conditions, so we retain the benefits of Oliveira’s deep model. We also get a natural representation of ghost variables: they are simply the logical variables provided by HOL, as distinguished from program variables. Finally, with the shallow expressions, Isabelle/HOL also gives us direct access to *sledgehammer* and other proof facilities for reasoning about expressions. This gives us the proof scalability we need and brings us on par with different shallow embeddings.

Z Toolkit. To support *Circus* and related languages, we need the types, operators, and laws of Z [100]. This includes types like partial functions, finite functions, and partial surjections. Whilst the Isabelle/HOL standard library contains some of these, we preferred to develop our own to have greater control over the design decisions. Our [Z_Toolkit](#) library also includes support for code generation so that we can make some specifications executable. Moreover, we have recently worked with Makarius Wenzel (Isabelle’s primary developer) to add the complete Z symbols into the Isabelle Unicode font and symbol library.

UTP. The modularisation leaves the main UTP library as a modest development, formalising predicates, relations, theories, and associated laws. This development continues, and we plan to have each UTP theory in a separate library.

A result of the modularisation is that we have been able to integrate our technology into collaborations that do not use UTP (at least knowingly). A recent development is an Isabelle-based verification tool for hybrid systems [48], which implements an extended version of Platzer’s differential dynamic logic. This tool extensively uses the [Shallow-Expressions](#) library to support techniques like differential induction and differential ghosts. A result that we are pleased with is the inclusion of a separation-logic-style frame rule:

$$\frac{\{P\} C \{Q\} \quad C \text{ nmods } A \quad -A \# R}{\{P \wedge R\} C \{Q \wedge R\}}$$

Here, $C \text{ nmods } A$ is a semantic operator, like unrestriction, requiring that C does not modify any variables in A . We also need the frame invariant R to use no variables inside A . This requires constructing a lens’s complement using an algebraic structure called a “scene”, which is ongoing work (see Sect. 6). This being the case, we can add R as an invariant for a command C . This shows one of the real benefits of the UTP: to link concepts (separation logic and hybrid systems) from apparently very different areas of computer science.

4.8 Interaction Trees

Recently, we have mechanised Interaction Trees (ITrees) in Isabelle/UTP [47, 122]. These are coinductive structures that allow symbolic encoding of deterministic labelled transition systems. They can therefore support encoding and

reasoning about operational semantics using coinductive techniques. Crucially, ITrees are executable, which allows us to take abstract models and programs, generate code for them, and finally animate them. Though ITrees can be infinite, languages like Haskell, which supports lazy evaluation, can evaluate them. Thus, we can use ITrees to animate deterministic *Circus* processes, for example. This is very valuable in software development since engineers can obtain prototypes.

Our ITrees library is built on the *Shallow-Expressions* and *Z_Toolkit* libraries. Integration with the rest of UTP is underway, allowing us to translate relational specifications into executable programs. Though ITrees are intrinsically deterministic, we can model nondeterminism with special events, enabling various strategies for resolving nondeterminism. We have applied this library in the development of a tool called *Z_Machines*, which supports system modelling in the style of Z and B, with both animation and verification support [121].

5 Other Contributions

We now consider two projects using *Circus* that did not involve our research group: the Xenon project and another theorem prover for *Circus*.

Freitas and McDermott used *Circus* in the Xenon project at the Naval Research Laboratory in Washington DC, USA. Xenon is a higher-assurance secure-separation hypervisor that allows a host computer to support multiple separated virtual machines that share memory and processing resources. Xenon is based on re-engineering the well-known Xen open-source hypervisor [70]. Xenon used formal specifications written in Z, CSP, and *Circus* [52,69] in security assurance. Freitas and McDermott modelled the fundamental definition of security, the hypercall interface behaviour, and the internal modular design. Security is based on noninterference expressed as a determinism property [70,93].

The Xenon Project is an industrial-scale application of *Circus*. The specification is 4,500 lines long: a substantial piece of mathematics. Some attractive technical advantages in modelling security properties in *Circus* arise from the combination of state and traces. Usually, proofs of noninterference require an unwinding theorem relating traces and states (see Goguen and Meseguer [54]). This is addressed in the definition of the *Circus* language. Xenon shows how *Circus* provides a powerful and natural way to describe state-rich and trace-rich concurrent behaviour in a single model amenable to refinement calculation.

Felliachi and colleagues developed machine-checked, formal semantics based on a shallow embedding of *Circus* in the Isabelle theorem prover [36]. They derive proof rules from the semantics and implement tactics for refinement of *Circus* processes involving data and behavioural aspects. Their proof environment supports syntax and semantics very close to our presentation of *Circus* in [82,83]. The theories are available in Isabelle's Archive of Formal Proofs [37].

Felliachi et al. used their mechanisation of *Circus* to provide a principled testing environment for concurrent systems [35]. They describe integrating formal testing in a proof environment as *theorem-prover based testing*, which takes advantage of the precise semantics of a specific specification language implemented in the theorem prover. They present a machine-checked formalisation of

a testing theory. They experiment with this theory by testing an industrial case study: a message monitoring module. The component under test is embedded in 5k lines of Java code. It binds together various devices, including pacemaker controllers, using sophisticated data structures and operations, providing the primary source of complexity when testing. More details about this case study can be found in Feliachi’s thesis [34] and in a technical report [33].

6 Quo Vadis *Circus*?

Work on *Circus* and Isabelle/UTP is ongoing and highly active. This section discusses current research and applications, and future directions (Sect. 6.1). We also include a brief industrial roadmap (Sect. 6.2) of outstanding work for transitioning *Circus* to a practical systems engineering and development setting.

6.1 Research Directions

Concerning extensions of *Circus*, we single out the hybrid state-rich process algebra called *CyPhyCircus* [41, 48, 77]. In addition to processes with states (like in *Circus*), a *CyPhyCircus* process can include continuous visible state components. As expected, its foundation is UTP. It is used in the RoboStar framework [20], which provides domain-specific notations for modelling robotics control-software design and simulations, physical platforms, and scenarios. A distinctive feature of RoboStar is that all these notations have formal semantics that is automatically generated and integrated via their common UTP foundations.

CyPhyCircus has been used as a formal framework to give the semantics of RoboSim [20], capturing diagrammatic behavioural models for the platform and scenarios, and RoboWorld [21], a controlled natural language (CNL) used to record assumptions about the environment. The semantics of RoboSim diagrams and RoboWorld documents is a hybrid model due to the platform and environment’s continuous nature, including quantities of interest such as velocity and temperature. From the semantics, it is possible, for instance, to generate tests or check whether the environment assumptions are satisfied by a simulation.

As future work, the main challenges for *CyPhyCircus* as a hybrid process algebra concern automated reasoning. Notably, for the mechanised reasoning to scale, we need theorem-proving facilities. In this respect, we can benefit from the UTP theories and all the encoding already developed in Isabelle/UTP. We are currently developing bespoke automated proof methods to support verification of RoboSim models based on our hybrid verification tool [48]. To further improve automation, the plan is also to support model checking via translating *CyPhyCircus* models to hybrid automata accepted by model checkers [3].

Another exciting research direction is our work on probability. One of its applications is also in the RoboStar context. More specifically, a probabilistic denotational semantics is defined in [109] for the RoboStar design notation, called RoboChart [71]. We base our work on the weakest completion semantics, which is, once more, based on UTP. The work relates standard semantics for a

nondeterministic language with a probabilistic semantic domain via a forgetful function (from the latter to the former) and its converse for the other way around. The embedding using the converse of the forgetful function is proved to preserve the program structure. Finally, the probabilistic choice operator is defined.

In future work, we need to develop techniques for managing uncertainty. Several promising directions include partially observable Markov decision processes [61], dynamic epistemic logic [9], and the epistemic mu-calculus [98]. We will pursue a unifying theory that includes these and other approaches.

Many machine learning methods approximate a function between inputs and outputs. Reasoning about these approximate functions requires probabilistic techniques and presents many challenges. An outline of a probabilistic domain theory for robotics that includes learning components has been proposed by Thrun et al. [102]. We propose to formalise this theory.

A mechanised theory of quantum programming will provide a common framework for classical and quantum specifications, quantum program development, and analysis of program time and space complexity. Applications include quantum cryptographic protocols, where we must use distributed quantum programming with quantum channels. Hehner has established an initial basis for quantum programming in the UTP style [101]. We propose to continue this work.

Regarding work on Isabelle/UTP, current efforts focus on optimisation and modularisation (Sect. 4.7). More specifically, the **Optics** library defining the lenses (Sect. 4.3) contains several related algebraic structures (that is, symmetric lenses and prisms) and provides commands such as **alphabet** to create alphabet types and **chantype** to create channel types. In future, we will create additional commands to ease the creation of formal artifacts to support software engineering, in particular constructs from RoboChart and RoboSim.

We will also enrich this library with an axiomatic value model [124] that provides a convenient way to directly inject HOL types into a single given universe type to model state spaces without the need to instantiate them. We are considering a sound axiomatisation of higher-order UTP ([59, Chap. 9]) as well.

Our work with Interaction Trees has complemented the UTP relational hierarchy with operational semantic models that can be directly verified and executed. We are exploring using the Isabelle code generator to provide verified simulations and controller implementations in Haskell. Our Z-Machines tool [121] is under active development as a usable method for creating and verifying formal models, and we have a growing library of accompanying examples from [110].

Finally, our work on the Isabelle-based verification tool for hybrid systems discussed in Sect. 4.7 is a neat example of using UTP to link concepts from different computer science areas, separation logic and hybrid systems. Our main activity here is in development of case studies to validate the tool, and improve proof automation and scalability. In future, we will extend it to include concurrency primitives to support verification of multi-robotic systems such as swarms.

6.2 Industrial Roadmap

This section describes a roadmap to scale *Circus* adoption in industry. This is, in particular, finding ways to integrate *Circus* into modern workflows for model-driven development and model-based software engineering, including the underlying continuous integration, development, and verification pipelines. Our overarching aim is to make it easier for tool developers to harness the power of *Circus* and Isabelle/UTP. Future efforts may include the definition of a meta-model that can be integrated into common IDEs, such as the Eclipse framework, and plug-ins that encapsulate various checking and verification tasks on *Circus* models by outsourcing them to Isabelle/UTP.

A challenge we will have to face is to ease the learning curve for software engineers to understand, modify, write, and maintain *Circus* models as part of a model-based engineering workflow. AI-powered solutions such as CoPilot [30] are becoming more prevalent in supporting developers in producing models and code, from identifying issues to suggesting solutions based on natural-language queries and requirements. At the same time, projectional editors and low-code techniques may enable developers to produce design models before attaining deep and expert knowledge of the low-level modelling notation per se.

Moreover, many tools and IDEs for formal development and verification are now equipped with mechanisms for giving continuous feedback to the user to flag possible issues in models and code as soon as changes are made, automatically keeping verification conditions and proofs in sync with their models. Similar technology can be developed for *Circus* to facilitate system-level architectural engineering and code verification via a contract language that ties in nicely with commonly used platforms and implementation languages and technology.

We thus envisage an ecosystem of *Circus* tools that allow us to:

- (1) **instantiate** *Circus* models based on common modelling patterns that are geared to particular application domains;
- (2) seamlessly **interface** from IDEs such as Eclipse or Visual Studio Code with Isabelle/UTP to engineer, validate and refactor *Circus* models;
- (3) support manual, semi-automatic, and automatic refinement through a bespoke **refinement editor** that makes system engineering via *Circus* amenable to software architects and industrial software developers;
- (4) **trace** *Circus* models and their artefacts *up the refinement chain*: to informal or semiformal specifications, domain engineering, and product-line engineering models; and *down the refinement chain* to architectures written in UML/SysML or AADL, for instance, code-level contracts, and test cases;
- (5) use a repository of verified **refinement patterns** that can be easily instantiated for particular modelling patterns and used to create a skeleton for implementation activities, including associated code-level contracts;
- (6) integrated *Circus* models into static and run-time **testing** and **verification** activities and popular testing frameworks.

Regarding (1), we have already elicited many such modelling patterns as part of research targetting the application of *Circus* to several complementary applica-

tion domains, including hybrid and control systems, robotics, and safety-critical concurrent and real-time implementations in Java and Ada.

Concerning (2), provers such as Isabelle already provide an API and protocols to communicate with external tools asynchronously. Still, high-level interfaces must be created on top of those low-level protocols to efficiently deal with changes to *Circus* models, and analyse their impact on proofs.

The aim of (3) is to disentangle the application of *Circus* refinement laws from a heavyweight proof framework. Once *Circus* refinement laws are proved in Isabelle/UTP, we may use a more bespoke and efficient tool to apply them and carry out large-scale refinements that may take advantage of a versatile tactic language and user-friendly GUI. Code generation in Isabelle enables us to potentially derive such a (critical) tool rigorously from proven laws.

Traceability (4) is essential when using model and proof artefacts of a *Circus*-based development as certification evidence in assurance cases. We hence require means to place *Circus* into the context of large-scale developments that often use a variety of complementary notations for requirements, architecture, design and HW/SW implementations, with clear traceability links to *Circus* models.

For (5), every modelling pattern should provide at least one refinement pattern and a collection of proved laws. Lastly, for (6), tying in with our work with Gaudel on a testing theory for *Circus* [22], we can leverage *Circus* to automate test-case generation and other testing activities.

The richness of the *Circus* language, and its UTP foundations, inherently opens several opportunities for combined verification solutions.

7 Conclusions

This paper reviewed two decades of our research on the stateful process algebra *Circus*, its UTP foundations, and the Isabelle/UTP theorem prover. Many colleagues and students have helped us to contribute to this agenda. We have published over 150 papers on UTP. This paper reviews only a fraction, and we will take future opportunities to complete the review of all our work.

One point to reflect on is why we have chosen Isabelle to mechanise UTP. The answer is mainly pragmatic. We want to be able to support scalable verification, and that means we want the best possible automation we can. This should not be at the expense of guaranteed soundness or fidelity, which is why we chose a foundational prover with strong support for automation.

Overall, an extensive body of research has already been carried out to (a) provide a firm semantic foundation for the *Circus* family of languages, (b) mechanise it in theorem provers, and (c) show, by way of examples and case studies drawn from both academic literature and the industrial realm, how *Circus* can be used to tackle the refinement-based development of safety-critical systems. Some current and future research directions have been discussed in the previous section, as well as an industrial roadmap to embody the techniques and tools we have developed for *Circus* into practical development environments.

Circus continues to attract interest from academia and industry. Its design is centred on the UTP principles. Jifeng’s joint work with Tony has been the seed and the beautiful semantic infrastructure of our long-term research on *Circus*. We are confident that we will have much more to report in years to come.

Acknowledgements. We gratefully acknowledge all our UTP-based research collaborators, co-authors, and students. Thanks to all of you. This work has recently been funded by the UK EPSRC Grants EP/M025756/1, EP/R025479/1, EP/V026801/2, EP/S001190/1, and by the Royal Academy of Engineering Grant No CiET1718/45. Over the years, many other funding sources have been available to us, as detailed in the cited papers. Thank you.

References

1. Abrial, J.-R.: Steam-boiler control specification problem. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 500–509. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0027252>
2. Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. LNCS, vol. 1165. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0027227>
3. Althoff, M.: An introduction to CORA 2015. In: Frehse, G., Althoff, M. (eds.) 1st and 2nd International Workshop on Applied Verification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 34, pp. 120–151. EasyChair (2015)
4. Arthan, R.: ProofPower. Lemma 1 Ltd. (2017). <https://www.lemma-one.com/ProofPower/index/>
5. Atiya, D.M., King, S.: A compliance notation for verifying concurrent systems. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, pp. 731–732. Association for Computing Machinery (2002). <https://doi.org/10.1145/581339.581475>
6. Atiya, D.-A., King, S., Woodcock, J.C.P.: A *Circus* semantics for Ravenscar protected objects. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 617–635. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_34
7. Back, R.J.R., Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science, Springer, New York (1998). <https://doi.org/10.1007/978-1-4612-1674-2>
8. Back, R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. *Distrib. Comput.* **3**(2), 73–87 (1989). <https://doi.org/10.1007/BF01558665>
9. Baltag, A., Moss, L.S., Solecki, S.: The logic of public announcements and common knowledge and private suspicions. In: Gilboa, I. (ed.) Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-1998), Evanston, IL, USA, 22–24 July 1998, pp. 43–56. Morgan Kaufmann (1998)
10. Barnes, J.: Programming in ADA 95, 2nd edn. Addison-Wesley (1998)
11. Barrocas, S.L.M., Oliveira, M.V.M.: JCircus 2.0: an extension of an automatic translator from Circus to Java. In: Welch, P.H., Barnes, F.R.M., Chalmers, K., Pedersen, J.B., Sampson, A.T. (eds.) 34th Communicating Process Architectures, CPA 2012, Organised Under the Auspices of WoTUG, Dundee, Scotland, UK, 26 August 2012, pp. 15–36. Open Channel Publishing Ltd. (2012)

12. Bauer, J.C.: Specification for a software program for a boiler water content monitor and control system. Technical report, Institute of Risk Research, University of Waterloo (1993)
13. Behrmann, G., et al.: UPPAAL 4.0. In: 3rd International Conference on the Quantitative Evaluation of Systems, pp. 125–126. IEEE Computer Society (2006)
14. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In: Asplund, L. (ed.) Ada-Europe 1998. LNCS, vol. 1411, pp. 263–275. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055011>
15. Butterfield, A., Gancarski, P., Woodcock, J.C.P.: State visibility and communication in unifying theories of programming. In: Chin, W.N., Qin, S. (eds.) 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 47–54. IEEE Computer Society (2009)
16. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-Circus. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 75–97. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_5
17. Butterfield, A., Woodcock, J.: Semantic domains for Handel-C. In: Flynn, S., et al. (eds.) Second Irish Conference on the Mathematical Foundations of Computer Science and Information Technology, MFCSIT 2002. Electronic Notes in Theoretical Computer Science, Galway, Ireland, 18–19 July 2002, vol. 74, pp. 1–20. Elsevier (2002). [https://doi.org/10.1016/S1571-0661\(04\)80762-X](https://doi.org/10.1016/S1571-0661(04)80762-X)
18. Butterfield, A., Woodcock, J.: prialt in Handel-C: an operational semantics. Int. J. Softw. Tools Technol. Transf. **7**(3), 248–267 (2005). <https://doi.org/10.1007/s10009-004-0181-6>
19. Canham, S., Woodcock, J.: Three approaches to timed external choice in UTP. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 1–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_1
20. Cavalcanti, A., et al.: RoboStar technology: a roboticist’s toolbox for combined proof, simulation, and testing. In: Cavalcanti, A., Dongol, B., Hierons, R., Timmis, J., Woodcock, J. (eds.) Software Engineering for Robotics, pp. 249–293. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-66494-7_9
21. Cavalcanti, A., Baxter, J., Carvalho, G.: RoboWorld: where can my robot work? In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 3–22. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_1
22. Cavalcanti, A.L.C., Gaudel, M.C.: Testing for refinement in *Circus*. Acta Informatica **48**(2), 97–147 (2011). <https://doi.org/10.1007/s00236-011-0133-z>
23. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A refinement strategy for *Circus*. Formal Aspects Comput. **15**(2–3), 146–181 (2003). <https://doi.org/10.1007/s00165-003-0006-5>
24. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in *Unifying Theories of Programming*. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006). https://doi.org/10.1007/11889229_6
25. Cavalcanti, A.L.C., Woodcock, J.C.P., Dunne, S.: Angelic nondeterminism in the unifying theories of programming. Formal Aspects Comput. **18**(3), 288–307 (2006). <https://doi.org/10.1007/s00165-006-0001-8>
26. Cavalcanti, A., Sampaio, A., Woodcock, J.: Refinement of actions in Circus. In: Derrick, J., Boiten, E.A., Woodcock, J., von Wright, J. (eds.) BCS FACS Refinement Workshop 2002, Refine 2002, Satellite Event of FLoC 2002. Electronic Notes in Theoretical Computer Science, Copenhagen, Denmark, 20–21 July 2002, vol.

- 70, pp. 132–162. Elsevier (2002). [https://doi.org/10.1016/S1571-0661\(05\)80489-X](https://doi.org/10.1016/S1571-0661(05)80489-X)
27. Cavalcanti, A., Sampaio, A., Woodcock, J.: A refinement strategy for Circus. *Formal Aspects Comput.* **15**(2–3), 146–181 (2003). <https://doi.org/10.1007/s00165-003-0006-5>
 28. Cavalcanti, A., Woodcock, J.: ZRC – a refinement calculus for Z. *Formal Aspects Comput.* **10**(3), 267–289 (1998). <https://doi.org/10.1007/s001650050016>
 29. Cavalcanti, A., Woodcock, J.: Predicate transformers in the semantics of Circus. *IEE Proc. Softw.* **150**(2), 85–94 (2003). <https://doi.org/10.1049/ip-sen:20030131>
 30. Copilot: Your AI pair programmer. GitHub. <https://copilot.github.com>. Accessed 18 June 2023
 31. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall (1976). <https://www.worldcat.org/oclc/019584451>
 32. Feliachi, A., Gaudel, M.-C., Wolff, B.: Unifying theories in Isabelle/HOL. In: Qin, S. (ed.) *UTP 2010*. LNCS, vol. 6445, pp. 188–206. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16690-7_9
 33. Feliachi, A., Gaudel, M.C., Wolff, B.: Exhaustive testing in HOL-Testgen/CirTa – a case study. Technical report 1562, LRI, July 2013
 34. Feliachi, A.: *Semantics-based testing for Circus*. (Test basé sur la sémantique pour Circus). Ph.D. thesis, University of Paris-Sud, Orsay, France (2012). <https://theses.hal.science/tel-00821836>
 35. Feliachi, A., Gaudel, M.-C., Wenzel, M., Wolff, B.: The *Circus* testing theory revisited in Isabelle/HOL. In: Groves, L., Sun, J. (eds.) *ICFEM 2013*. LNCS, vol. 8144, pp. 131–147. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41202-8_10
 36. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/*Circus*: a process specification and verification environment. In: Joshi, R., Müller, P., Podelski, A. (eds.) *VSTTE 2012*. LNCS, vol. 7152, pp. 243–260. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_20
 37. Feliachi, A., Wolff, B., Gaudel, M.: Isabelle/Circus. *Arch. Formal Proofs* 2012 (2012). <https://www.isa-afp.org/entries/Circus.shtml>
 38. Fischer, C.: How to combine Z with a process algebra. In: Bowen, J.P., Fett, A., Hinchey, M.G. (eds.) *ZUM 1998*. LNCS, vol. 1493, pp. 5–23. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-540-49676-2_2
 39. Fischer, C., Wehrheim, H.: Failure-divergence semantics as a formal basis for an object-oriented integrated formal method. *Bull. EATCS* **71**, 92–101 (2000)
 40. Foster, J.: *Bidirectional programming languages*. Ph.D. thesis, University of Pennsylvania (2009)
 41. Foster, S.: Hybrid relations in Isabelle/UTP. In: Ribeiro, P., Sampaio, A. (eds.) *UTP 2019*. LNCS, vol. 11885, pp. 130–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_7
 42. Foster, S., Cavalcanti, A.L.C., Canham, S., Woodcock, J.C.P., Zeyda, F.: Unifying theories of reactive design contracts. *Theor. Comput. Sci.* **802**, 105–140 (2020). <https://doi.org/10.1016/j.tcs.2019.09.017>
 43. Foster, S., Cavalcanti, A.L.C., Woodcock, J.C.P., Zeyda, F.: Unifying theories of time with generalised reactive processes. *Inf. Process. Lett.* **135**, 47–52 (2018). <https://doi.org/10.1016/j.ipl.2018.02.017>
 44. Foster, S., Ye, K., Cavalcanti, A.L.C., Woodcock, J.C.P.: Automated verification of reactive and concurrent programs by calculation. *J. Log. Algebraic Methods Program.* **121**, 100681 (2021). <https://doi.org/10.1016/j.jlamp.2021.100681>

45. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: a mechanised theory engineering framework. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 21–41. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_2
46. Foster, S., Baxter, J., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Sci. Comput. Program.* **197**, 102510 (2020). <https://doi.org/10.1016/j.scico.2020.102510>
47. Foster, S., Hur, C., Woodcock, J.: Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021. LIPIcs, 24–27 August 2021, Virtual Conference, vol. 203, pp. 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.CONCUR.2021.20>
48. Foster, S., Huerta y Munive, J.J., Gleirscher, M., Struth, G.: Hybrid systems verification with Isabelle/HOL: simpler syntax, better models, faster proofs. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 367–386. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_20
49. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 295–314. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_17
50. Freitas, A., Cavalcanti, A.: Automatic translation from *Circus* to Java. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 115–130. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_9
51. Freitas, L.J.S.: Model checking *Circus*. Ph.D. thesis, University of York, Department of Computer Science (2006)
52. Freitas, L., McDermott, J.P.: Formal methods for security in the Xenon hypervisor. *Int. J. Softw. Tools Technol. Transf.* **13**(5), 463–489 (2011). <https://doi.org/10.1007/s10009-011-0195-9>
53. Gibson-Robinson, T., Armstrong, P.J., Boulgakov, A., Roscoe, A.W.: FDR3: a parallel refinement checker for CSP. *Int. J. Softw. Tools Technol. Transf.* **18**(2), 149–167 (2016). <https://doi.org/10.1007/s10009-015-0377-y>
54. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, 29 April–2 May 1984, pp. 75–87. IEEE Computer Society (1984). <https://doi.org/10.1109/SP.1984.10019>
55. Guttman, W., Möller, B.: Normal design algebra. *J. Log. Algebraic Program.* **79**(2), 144–173 (2010)
56. Harwood, W., Cavalcanti, A., Woodcock, J.: A theory of pointers for the UTP. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 141–155. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85762-4_10
57. Henkin, L., Monk, J., Tarski, A.: *Cylindric Algebras, Part I*. North-Holland (1971)
58. Hoare, C.A.R.: *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall (1985)
59. Hoare, C.A.R., He, J.: *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall (1998)
60. Jones, G., Goldsmith, M.: *Programming in OCCAM 2*. International Series in Computer Science. Prentice Hall (1985)
61. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. *Artif. Intell.* **101**(1–2), 99–134 (1998). [https://doi.org/10.1016/S0004-3702\(98\)00023-X](https://doi.org/10.1016/S0004-3702(98)00023-X)

62. King, S., Sørensen, I.H., Woodcock, J.: *Z, Grammar and Concrete and Abstract Syntaxes*. Technical Monograph PRG-68. Oxford University Computing Laboratory, Programming Research Group (1988)
63. Liu, Z., Woodcock, J., Zhu, H. (eds.): *ICTAC 2013*. LNCS, vol. 8049. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-39718-9>
64. Liu, Z., Woodcock, J., Zhu, H. (eds.): *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*. LNCS, vol. 8051. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-39698-4>
65. Liu, Z., Woodcock, J., Zhu, H. (eds.): *Unifying Theories of Programming and Formal Engineering Methods*. LNCS, vol. 8050. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-39721-9>
66. Locke, D., et al.: *Safety-Critical Java Technology Specification*, Public Draft. Java Community Process (2011)
67. Celoxica Ltd.: *DK3: Handel-C Language Reference Manual* (2002)
68. Lundqvist, K., Asplund, L., Michell, S.: A formal model of the Ada Ravenscar tasking profile; protected objects. In: González Harbour, M., de la Puente, J.A. (eds.) *Ada-Europe 1999*. LNCS, vol. 1622, pp. 12–25. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48753-0_2
69. McDermott, J.P., Freitas, L.: Using formal methods for security in the Xenon project. In: Sheldon, F.T., Prowell, S.J., Abercrombie, R.K., Krings, A.W. (eds.) *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop, CSIIRW 2010, Oak Ridge, TN, USA, 21–23 April 2010*, p. 67. ACM (2010). <https://doi.org/10.1145/1852666.1852742>
70. McDermott, J.P., Kirby, J., Montrose, B.E., Johnson, T., Kang, M.H.: Re-engineering Xen internals for higher-assurance security. *Inf. Secur. Tech. Rep.* **13**(1), 17–24 (2008). <https://doi.org/10.1016/j.istr.2008.01.001>
71. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J., Woodcock, J.C.P.: RoboChart: modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.* **18**(5), 3097–3149 (2019). <https://doi.org/10.1007/s10270-018-00710-z>
72. Miyazawa, A., Cavalcanti, A., Wellings, A.J.: SCJ-Circus: specification and refinement of safety-critical Java programs. *Sci. Comput. Program.* **181**, 140–176 (2019). <https://doi.org/10.1016/j.scico.2019.01.002>
73. Morgan, C.: Data refinement by miracles. *Inf. Process. Lett.* **26**(5), 243–246 (1988). [https://doi.org/10.1016/0020-0190\(88\)90147-0](https://doi.org/10.1016/0020-0190(88)90147-0)
74. Morgan, C.: Of wp and CSP. In: Feijen, W.H.J., van Gasteren, A.J.M., Gries, D., Misra, J. (eds.) *Beauty Is Our Business*. MCS, pp. 319–326. Springer, New York (1990). https://doi.org/10.1007/978-1-4612-4476-9_37
75. Morgan, C.: *Programming from Specifications*. International Series in Computer Science, 2nd edn. Prentice Hall (1994)
76. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.* **9**(3), 287–306 (1987). [https://doi.org/10.1016/0167-6423\(87\)90011-6](https://doi.org/10.1016/0167-6423(87)90011-6)
77. Foster, S., Huerta y Munive, J.J., Struth, G.: Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: Fahrenberg, U., Jipsen, P., Winter, M. (eds.) *RAMiCS 2020*. LNCS, vol. 12062, pp. 169–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43520-2_11
78. O’Halloran, C.: *Identifying critical requirements*. Technical report, Systems Assurance Group, QinetiQ Malvern (2002)

79. Oliveira, M.V.M.: Formal derivation of state-rich reactive programs using Circus. Ph.D. thesis, University of York, UK (2005). <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.428459>
80. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Refining industrial scale systems in *Circus*. In: East, I., Martin, J., Welch, P., Duce, D., Green, M. (eds.) *Communicating Process Architectures*. Concurrent Systems Engineering Series, vol. 62, pp. 281–309. IOS Press (2004)
81. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Formal development of industrial-scale systems in Circus. *Innov. Syst. Softw. Eng.* **1**(2), 125–146 (2005). <https://doi.org/10.1007/s11334-005-0014-0>
82. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A denotational semantics for Circus. In: Aichernig, B.K., Boiten, E.A., Derrick, J., Groves, L. (eds.) *Proceedings of the 11th Refinement Workshop, Refine@ICFEM 2006*. Electronic Notes in Theoretical Computer Science, Macao, 31 October 2006, vol. 187, pp. 107–123. Elsevier (2006). <https://doi.org/10.1016/j.entcs.2006.08.047>
83. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP semantics for Circus. *Formal Aspects Comput.* **21**(1–2), 3–32 (2009). <https://doi.org/10.1007/s00165-007-0052-5>
84. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Unifying theories in ProofPower-Z. *Formal Aspects Comput.* **25**(1), 133–158 (2013). <https://doi.org/10.1007/s00165-007-0044-5>
85. Ribeiro, P., Cavalcanti, A.L.C.: Designs with angelic nondeterminism. In: 7th International Symposium on Theoretical Aspects of Software Engineering, pp. 71–78. IEEE (2013). <https://doi.org/10.1109/TASE.2013.18>
86. Ribeiro, P., Cavalcanti, A.: Angelicism in the theory of reactive processes. In: Naumann, D. (ed.) *UTP 2014*. LNCS, vol. 8963, pp. 42–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_3
87. Ribeiro, P., Cavalcanti, A.L.C.: Angelic processes for CSP via the UTP. *Theor. Comput. Sci.* **756**, 19–63 (2019). <https://doi.org/10.1016/j.tcs.2018.10.008>
88. Ribeiro, P.: A unary semigroup trace algebra. In: Fahrenberg, U., Jipsen, P., Winter, M. (eds.) *RAMiCS 2020*. LNCS, vol. 12062, pp. 270–285. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43520-2_17
89. Roscoe, A.W.: Denotational semantics for occam. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) *CONCURRENCY 1984*. LNCS, vol. 197, pp. 306–329. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15670-4_15
90. Roscoe, A.W.: *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall (1997)
91. Roscoe, A.W.: *Understanding Concurrent Systems*. Texts in Computer Science, Springer, London (2010). <https://doi.org/10.1007/978-1-84882-258-0>
92. Roscoe, A.W., Hoare, C.A.R.: The laws of OCCAM programming. *Theor. Comput. Sci.* **60**, 177–229 (1988). [https://doi.org/10.1016/0304-3975\(88\)90049-7](https://doi.org/10.1016/0304-3975(88)90049-7)
93. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through determinism. In: Gollmann, D. (ed.) *ESORICS 1994*. LNCS, vol. 875, pp. 31–53. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58618-0_55
94. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in *Circus*. In: Eriksson, L.-H., Lindsay, P.A. (eds.) *FME 2002*. LNCS, vol. 2391, pp. 451–470. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45614-7_26
95. Schneider, S.A., Treharne, H.: CSP theorems for communicating B machines. *Formal Aspects Comput.* **17**(4), 390–422 (2005). <https://doi.org/10.1007/s00165-005-0076-7>

96. Sherif, A., Jifeng, H.: Towards a time model for *Circus*. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_62
97. Sherif, A., Jifeng, H., Cavalcanti, A., Sampaio, A.: A framework for specification and validation of real-time systems using *Circus* actions. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 478–493. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_34
98. Shilov, N.V., Garanina, N.O.: Combining knowledge and fixpoints. Technical report preprint 98, A.P. Ershov Institute of Informatics Systems, Novosibirsk (2002). <https://www.iis.nsk.su/files/preprints/098.pdf>
99. Smith, G.: A semantic integration of object-Z and CSP for the specification of concurrent systems. In: Fitzgerald, J., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63533-5_4
100. Spivey, J.M.: Z Notation – A Reference Manual. International Series in Computer Science, 2nd edn. Prentice Hall (1992)
101. Taffiovič, A., Hehner, E.C.R.: Quantum predicative programming. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 433–454. Springer, Heidelberg (2006). https://doi.org/10.1007/11783596_25
102. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. Intelligent Robotics and Autonomous Agents. MIT Press, Cambridge (2005)
103. Wei, K., Woodcock, J., Burns, A.: A timed model of Circus with the reactive design miracle. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13–18 September 2010, pp. 315–319. IEEE Computer Society (2010). <https://doi.org/10.1109/SEFM.2010.40>
104. Wei, K., Woodcock, J., Burns, A.: Timed Circus: timed CSP with the miracle. In: Perseil, I., Breitman, K.K., Sterritt, R. (eds.) 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27–29 April 2011, pp. 55–64. IEEE Computer Society (2011). <https://doi.org/10.1109/ICECCS.2011.13>
105. Welch, P.: Process oriented design for Java: concurrency for all. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2330, pp. 687–687. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46080-2_72
106. Welch, P.H., Aldous, J.R., Foster, J.: CSP networking for Java (*JCSP.net*). In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2330, pp. 695–708. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46080-2_74
107. Woodcock, J.C.P.: Properties of Z specifications. ACM SIGSOFT Softw. Eng. Notes 14(5), 43–54 (1989). <https://doi.org/10.1145/71633.71634>
108. Woodcock, J.C.P., Cavalcanti, A.L.C.: Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory (2001)
109. Woodcock, J., Cavalcanti, A., Foster, S., Mota, A., Ye, K.: Probabilistic semantics for RoboChart. In: Ribeiro, P., Sampaio, A. (eds.) UTP 2019. LNCS, vol. 11885, pp. 80–105. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_5
110. Woodcock, J.C.P., Davies, J.: Using Z - Specification, Refinement, and Proof. International Series in Computer Science. Prentice Hall (1996)

111. Woodcock, J.: Using Circus for safety-critical applications. In: Cavalcanti, A., Machado, P.D.L. (eds.) Proceedings of the 6th Brazilian Workshop on Formal Methods, WMF 2003. Electronic Notes in Theoretical Computer Science, Campinas Grande, Brazil, 12–14 October 2003, vol. 95, pp. 3–22. Elsevier (2003). <https://doi.org/10.1016/j.entcs.2004.04.003>
112. Woodcock, J.: The miracle of reactive programming. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 202–217. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_12
113. Woodcock, J.: Hoare and He’s unifying theories of programming. In: Jones, C.B., Misra, J. (eds.) Theories of Programming: The Life and Works of Tony Hoare, pp. 285–316. ACM/Morgan & Claypool (2021). <https://doi.org/10.1145/3477355.3477369>
114. Woodcock, J., Cavalcanti, A.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16–17 July 2001. Workshops in Computing, BCS (2001). <https://doi.org/10.14236/ewic/IWFM2001.7>
115. Woodcock, J., Cavalcanti, A.: The steam boiler in a unified theory of Z and CSP. In: 8th Asia-Pacific Software Engineering Conference (APSEC 2001), Macau, China, 4–7 December 2001, pp. 291–298. IEEE Computer Society (2001). <https://doi.org/10.1109/APSEC.2001.991490>
116. Woodcock, J., Cavalcanti, A.: The semantics of *Circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45648-1_10
117. Woodcock, J., Cavalcanti, A.: A tutorial introduction to designs in unifying theories of programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24756-2_4
118. Woodcock, J., Cavalcanti, A., Freitas, L.: Operational semantics for model checking Circus. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 237–252. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_17
119. Woodcock, J., Davies, J., Bolton, C.: Abstract data types and processes. In: Roscoe, A.W., Davies, J., Woodcock, J. (eds.) Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare. Millennial Perspectives in Computer Science, pp. 391–405. Palgrave (2000)
120. Woodcock, J.C.P., Morgan, C.: Refinement of state-based concurrent systems. In: Bjørner, D., Hoare, C.A.R., Langmaack, H. (eds.) VDM 1990. LNCS, vol. 428, pp. 340–351. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52513-0_18
121. Yan, F., Foster, S., Habli, I.: Automated compositional verification for robotic state machines using Isabelle/HOL. In: 27th International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE (2023)
122. Ye, K., Foster, S., Woodcock, J.: Formally verified animation for RoboChart using interaction trees. In: Riesco, A., Zhang, M. (eds.) ICFEM 2022. LNCS, vol. 13478, pp. 404–420. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17244-1_24
123. Zeyda, F., Cavalcanti, A.L.C.: *Circus* model for the SCJ framework. Technical report, University of York, Department of Computer Science, York, UK (2012)
124. Zeyda, F., Foster, S., Freitas, L.: An axiomatic value model for Isabelle/UTP. In: Bowen, J.P., Zhu, H. (eds.) UTP 2016. LNCS, vol. 10134, pp. 155–175. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52228-9_8