Unifying Model Execution and Deductive Verification with Interaction Trees in Isabelle/HOL

SIMON FOSTER, University of York, UK
CHUNG-KIL HUR, Seoul National University, South Korea
JIM WOODCOCK, Southwest University, China, Aarhus University, Denmark, and University of York, UK

Model execution allows us to prototype and analyse software engineering models by stepping through their possible behaviours, using techniques like animation and simulation. On the other hand, deductive verification allows us to construct formal proofs demonstrating satisfaction of certain critical properties in support of high-assurance software engineering. To ensure coherent results between execution and proof, we need unifying semantics and automation. In this paper, we mechanise Interaction Trees (ITrees) in Isabelle/HOL to produce an execution and verification framework. ITrees are coinductive structures that allow us to encode infinite labelled transition systems, yet they are inherently executable. We use ITrees to create verification tools for stateful imperative programs, concurrent programs with message passing in the form of the CSP and *Circus* languages, and abstract system models in the style of the Z and B methods. We demonstrate how ITrees can account for diverse semantic presentations, such as structural operational semantics, a relational program model, and CSP's failures-divergences trace model. Finally, we demonstrate how ITrees can be executed using the Isabelle code generator to support the animation of models.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Program reasoning**; • **Computing methodologies → Concurrent computing methodologies**; **Simulation tools**; **Modeling methodologies**; **Model verification and validation**; • **Software and its engineering → Formal methods**; **Software verification and validation**; **Software verification**.

Additional Key Words and Phrases: Interaction Trees (ITrees), Isabelle/HOL, Model Execution, Deductive Verification, Coinductive Structures, Formal Methods, CSP (Communicating Sequential Processes), Circus Language, Z and B Methods, Structural Operational Semantics, Theorem Proving, Proof Automation, Software Engineering Models, Model Execution, High-Assurance Software, Automated Program Verification, Code Generation, Animation and Simulation, UTP (Unifying Theories of Programming), Imperative Programming Verification

ACM Reference Format:
Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2024. Unifying Model Execution and Deductive Verification with Interaction Trees in Isabelle/HOL. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (August 2024), 40 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Simon Foster, University of York, York, YO10 5GH, UK, simon.foster@york.ac.uk; Chung-Kil Hur, Seoul National University, Seoul, 08826, South Korea, gil.hur@sf.snu.ac.kr; Jim Woodcock, Southwest University, Chongqing, 400751, China and Aarhus University, Aarhua, 8200 Aarhus N, Denmark and University of York, York, YO10 5GH, UK, jim.woodcock@york.ac.uk.

1049-331X/2024/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Model-based engineering uses models to produce software with a high level of assurance [20, 42, 62]. Typically, engineers create behavioural models, such as state machines and activity diagrams, which abstractly specify a system's behaviour and can be subjected to prototyping using animation, simulation, and testing. These techniques require that models are executable, so we step through their behaviour [12, 17]. When models are accompanied by suitable formal semantics, they can be further subjected to formal verification to ensure they satisfy the requirements in every possible state [48]. The models can be refined further to produce verified code and related artefacts, creating high-integrity software with traceable links to the original requirements.

To ensure that these heterogeneous artefacts and analysis results can be applied coherently, there is a need to tie them together using unifying formal semantics to avoid semantic gaps that can introduce weaknesses [33, 51]. This semantics should allow us to give a formal mathematical meaning to each model used in the development hierarchy and account for the relations between them. It should also support execution to support early-stage prototyping [12]. Moreover, the semantics must have tool support with a high level of automation to minimise the expertise engineers require. Whilst semantic frameworks exist that support such a unification, such as Hoare and He's *Unifying Theories of Programming* [40] (UTP), the models are expressive but not usually executable. For formal methods to be accessible, we, therefore, need to support models that are inherently executable and verifiable.

Theorem proving is a powerful verification technique for analysing software engineering models and code by automating deductive proof steps. Proof assistants like Coq, Isabelle, and Lean provide a flexible foundation for mathematical reasoning. They can be applied to many engineering paradigms, from high-level design models [28, 30], potentially including interactions with the physical environment [31], to low-level code. Moreover, theorem provers can support the verification of systems with a very large, or even infinite, state space through symbolic logic techniques and compositional reasoning. However, unlike simulation and model-checking techniques, proof assistants typically have a high entry bar and require significant investment before meaningful results can be obtained. Consequently, to harness the benefits of theorem proving in software engineering, we need to improve access with early-stage prototyping techniques, such as animation[1] and simulation, and high levels of automation.

The contribution of this article is an Isabelle-based framework to support model-based engineering called Isabelle/ITrees. Our library implements the Interaction Tree (ITree) formalism of Xia et al. [66], which crucially supports formal models that are both directly executable and subject to verification by proof [67]. ITrees provide a natural encoding of operational semantics using coinductive techniques, where we can step through a model's behaviour in terms of its internal steps and external interactions. Though ITrees are intrinsically elementary structures, they have the potential to act as a unifying semantics model for a variety of software engineering artefacts. Our tool supports a tight development cycle where animation and verification activities can be intertwined.

ITrees are coinductive structures, which intuitively correspond to symbolic labelled transition systems. They intrinsically support mutable states and events and can model complex infinite behaviours. Our mechanisation of ITrees generalises the original work [66] by using partial functions to model visible events. This allows us to support both external choice

[1] In this context, animation refers to the interactive probing of a model's behaviour. Simulation is similar but is typically less interactive and, on the whole, more sophisticated.

and deadlock in the style of the CSP process algebra [13, 38, 55], along with the algebraic semantics of these operators, which broadens our implementation's application.

Our general, highly extensible framework can be applied to software engineering artefacts at various abstraction levels. We use this to provide shallow embeddings of imperative programs, communicating processes, and high-level system models in the style of the Z [58] and B [1] specification languages. This is supported through results from Hoare and He's Unifying Theories of Programming [25, 40] (UTP) to unify denotational, operational and axiomatic semantics, and in particular, the UTP semantics for the *Circus* process language [26, 29, 64].

Our tool benefits from Isabelle's powerful proof tools, notably the **sledgehammer** theorem prover integration [9], to automate the discharge of verification conditions and other proof obligations. Moreover, we employ Isabelle's code generator to provide execution of programs and animation of high-level models.

The structure of our paper is as follows. In §3, we show how ITrees are mechanised in Isabelle/HOL, including the core operators. We show how to derive structural operational semantics from ITrees, characterise weak bisimulation, which allows the abstraction of silent events, and provide theorems for reasoning about process iteration using chains. In §4, we show how to model and verify imperative programs using ITrees, demonstrate a link with our previous UTP-based relation semantics, and provide automated program verification using Hoare logic. In §5, we show how deterministic CSP and *Circus* processes can be semantically embedded into ITrees, including operators like external choice and parallel composition. We also link ITrees with the standard failures-divergences semantic model for CSP, which justifies their integration with other CSP-based techniques. In §6, we show how the code generator can be used to generate animations. In §7, we apply our library to develop a simple automated formal method for modelling systems, similar to the B-method [1]. In §8, we consider related work, and in §9, we conclude.

This paper extends our previous CONCUR 2021 paper [27]. We add results in the new section on imperative programming (§4), including total correctness Hoare logic and UTP-style predicative semantics, a new section on modelling with Z-Machines (§7), new theorems on iteration chains (§3), and additional narrative and more minor results throughout. All our results have been mechanised and can be found in the accompanying repository[2], and clickable icon links next to each specific result, with for Isabelle and for Haskell.

Notation. Our presentation uses both textbook-style notation and Isabelle code, though we generally prefer the former. This mixture is unavoidable, as the Isabelle code, though ultimately the single source of truth, is often more pedantic than necessary for a human reader and less accessible. We largely restrict Isabelle's code to modelling and verification examples to support the use of our tools. The reader interested in how the textbook mathematics is mechanised can follow the Isabelle links ( ).

2 BACKGROUND

This section introduces the foundational concepts used in this paper: Isabelle/HOL and the *Circus* language. *Circus* is used to motivate the value of ITrees in providing formal semantics for process algebraic languages. We also use the Z mathematical toolkit in our semantic definitions, which is used in *Circus*.

[2] https://github.com/isabelle-utp/interaction-trees

2.1 Isabelle/HOL

Isabelle/HOL, at its core, is a proof assistant for Higher Order Logic (HOL). It implements a Gentzen-style natural deduction system, which can be used to prove or falsify the validity of arguments formalised using predicate logic. The language of HOL is a strongly typed polymorphic λ-calculus, which can be used to formalise mathematical theories in a functional style. In particular, Isabelle/HOL provides a typed set theory, arithmetic theories (natural numbers, integers, real numbers, etc.), and various data structures, such as lists and records. As in functional programming languages, programs can specified using algebraic data types and recursive functions, with termination checks provided. These features give an expressive and extensible mathematical language in which various programming and modelling notations can be described.

Several facilities complement these modelling features for automating proof. Isabelle provides a simplifier (`simp`), which automates equational rewriting of terms, and a classical reasoner (`blast`), which implements the tableaux method for automating natural deduction. Additionally, there is a resolution prover (`metis`) for first-order predicate calculus and access to several SMT solvers, such as CVC4 and Z3, in the `smt` method. These various proof methods can be coordinated using the **`sledgehammer`** tool, which constructs proofs automatically using external automated proof tools.

The development of theories in Isabelle is centred around *theory documents*, which are used for modelling and proof. A theory document (extension `.thy`) consists of a sequence of commands, manipulating Isabelle's state by defining a function or starting a proof. The document model has two levels of syntax: (1) outer-syntax, which gives the syntax to individual commands, and (2) inner-syntax, which gives syntax to terms of the logic in typed λ-calculus. An example definition command is given below:

```
definition square :: "nat ⇒ nat" where
"square x = x * x"
```

The command begins with a major keyword (**`definition`**), which is highlighted, followed by a type declaration for a new constant, `square`, which is a total function from natural numbers to natural numbers (`nat ⇒ nat`). Following the type declaration, there is a minor keyword (**`where`**) and then the definitional equation for the function. Speech marks delimit this definitional equation since it is inner-syntax formed using the term language. The document model of Isabelle is extensible so that new commands can be implemented for bespoke modelling tasks using the meta-language Isabelle/ML.

The combination of an expressive and rigorous mathematical language and a high degree of automated proof make Isabelle ideally suited to formal verification for various languages. This requires that the semantics of the target language first be formalised as an Isabelle theory package and a suitable proof calculus (such as Hoare logic) be provided to form specifications and verify programs. Isabelle also provides a code generator for mathematical programs, which can be used to automate code production from verified artefacts.

2.2 Circus and Z

Circus is a formal language for modelling imperative and concurrent systems. It combines the communication primitives from the CSP process algebra with imperative programming primitives from Dijkstra's guarded command language (GCL) and rich state modelling facilities as provided by the Z notation [58].

Z is a formal language for specifying software using set theory and relational calculus. CSP is a language for modelling concurrent systems, such as protocols. It provides several modelling primitives, including.

- event prefix ($a \rightarrow P$): perform event a and then enable P;
- guard ($B \mathbin{\&} P$): enable P only when condition B is true;
- external choice ($P \Box Q$): allow the environment to choose P or Q;
- parallel composition ($P \,[\![A]\!]\, Q$): run P and Q in parallel synchronising on events in A;
- hiding ($P \setminus A$): internal events in set A.

Events are typically input events $a?x \rightarrow P(x)$ or output events $b!v \rightarrow Q$, where a and b are channels carrying typed data. The standard semantics for CSP is called the failures-divergences model [55], a denotational semantics based on traces, which we cover in §5.3.

In addition to these CSP operators, *Circus* also contains typical imperative programming operators from GCL like assignment ($x := e$), sequential composition ($P \fatsemi Q$), and iteration ($\mathit{while}\ B\ \mathit{do}\ C\ \mathit{od}$). From Z, it gains a mathematical toolkit, including data structures like sets, partial functions ($A \pfun B$), finite functions ($A \ffun B$), and sequences (lists), and also the ability to form abstract data types using Z schemas.

We also use the Z mathematical toolkit for partial functions in our semantic definitions. We can specify partial functions using $\lambda\, x \in A \bullet f(x)$, which restricts a function f to the domain A[3]. We can calculate the domain of a partial function f with $\mathrm{dom}(f)$. An empty partial function $\{\mapsto\}$ has an empty domain. We also use the domain restriction ($\lhd$) and override operators ($\oplus$) from the Z mathematical toolkit, which have the following definitions:

$$A \lhd f \mathrel{\hat{=}} (\lambda\, x \in A \cap \mathrm{dom}(f) \bullet f(x))$$
$$f \oplus g \mathrel{\hat{=}} (\lambda\, x \in \mathrm{dom}(f) \cup \mathrm{dom}(g) \bullet \mathit{if}\ x \in \mathrm{dom}(g)\ \mathit{then}\ g(x)\ \mathit{else}\ f(x))$$

We have implemented the Z mathematical toolkit in an Isabelle library as a hierarchy of types[4]. With the associated theorems, we can use Isabelle's simplifier to automate the calculation of a partial function's domain and other properties.

As an example *Circus* process, we formalise a simple reactive buffer. We introduce three channels: *Input* to accept a new input, *Output* to offer an output, and *State* to show the current state of the buffer. We consider a buffer containing a sequence of natural numbers $\mathbb{N}$ for simplicity. We also introduce a single variable, *buf*, which stores the values present in the buffer. The reactive behaviour of the buffer is then specified below:

Example 2.1 (Unbounded Buffer in Circus).

$$\begin{aligned} \mathit{buf} := [] \fatsemi\ & \mathit{while}\ \mathit{true}\ \mathit{do} \\ & \quad \mathit{Input}?(i) \rightarrow \mathit{buf} := \mathit{buf} \mathbin{@} [i] \\ & \Box\ (\mathit{length}(\mathit{buf}) > 0) \mathbin{\&} \mathit{Output}!(\mathit{hd}\ \mathit{buf}) \rightarrow \mathit{buf} := \mathit{tl}\ \mathit{buf} \\ & \Box\ \mathit{State}!(\mathit{buf}) \rightarrow \mathit{Skip} \\ & \mathit{od} \end{aligned}$$

The buffer enters a reactive infinite loop after initially setting the buffer to be an empty list ($[]$). The buffer provides three options using the external choice operator ($\Box$). It can accept an input over *Input*, extending the buffer using the list append operator ($xs \mathbin{@} ys$). If the buffer is not empty (its length is non-zero), it can offer the head of the buffer over

[3]This is distinguished from the total function notation $\lambda\, x.\, g(x)$ by a different separator ($\bullet$)
[4]Z-Toolkit library: https://github.com/isabelle-utp/Z_Toolkit.

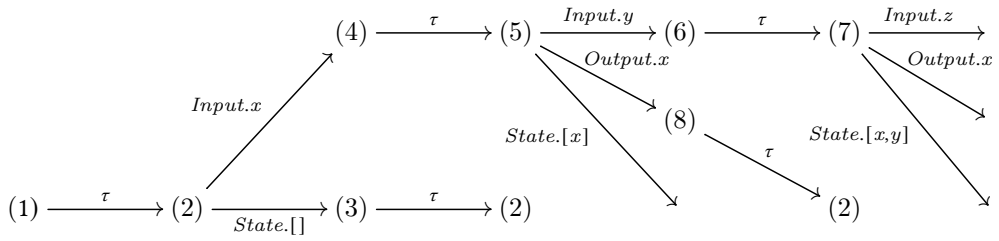

Fig. 1. An ITree fragment for the buffer example (approximate)

Output and contract the buffer by removing the head. Finally, it can display the current state of the buffer over the *State* channel.

3 INTERACTION TREES AND OPERATIONAL SEMANTICS

This section introduces Interaction Trees (ITrees), develops the main theory in Isabelle/HOL, derives operational semantics, and provides several novel results. ITrees were originally mechanised in Coq by Xia et al. [66]. Our mechanisation in Isabelle/HOL brings unique advantages, including a flexible front-end syntax, automated proof tools, and code generation to several languages.

3.1 Interaction Trees as Codatatypes

ITrees are potentially infinite trees whose edges are decorated with events, representing the interactions between a process and its environment. For intuition, an example ITree is shown in Figure 1 for the buffer in Example 2.1. The nodes are labelled with numbers for reference, and the edges with events, including visible events, such as $Input.x$, and invisible events (τ).

From the initial node (1), a single τ event is possible, which corresponds to assigning [] to the *buf* state variable ($buf := []$). Two visible events are presented to the environment, $Input.x$ and $State.[]$. The latter event, $State.[]$ indicates that the buffer is empty, so an *Output* event is unavailable. The former event, $Input.x$, corresponds to an infinite family of events for each possible value the channel can carry, such as $x = 0$, $x = 1$, $x = 3$ and so on. We can describe such infinite families in Isabelle/HOL symbolically as a term containing a free variable (x), such that ITrees can have infinite breadth.

If an input is received, we transition to node (4), from which a single τ event occurs, which corresponds to the assignment appending x to the buffer ($buf := buf \mathbin{@} [x]$). From node (5), three events are possible: $Input.y$, $Output.x$, and $State.[x]$. At this point, the buffer contains a single value x, which we can output or input another value y. Again, x and y are families of possible values carried by channels *Input* and *Output*. If another value y is input, the buffer is updated accordingly, leading to node (7). The tree continues in this manner and thus has an infinite depth. It can alternatively be considered as an unfolding of a labelled transition system.

We now describe the type in Isabelle that allows us to denote ITrees formally. ITrees are parametrised over two sorts (types): E of events and R of return values (or states). There are three possible interactions: (1) termination, returning a value in R; (2) an internal event (τ) followed by a successor ITree; or (3) a choice between several visible events. In Isabelle/HOL, we encode ITrees using a codatatype [8, 11]. A codatatype is similar to an algebraic datatype, having several disjoint constructors. However, the crucial difference is that whereas elements of a datatype are finite, elements of a codatatype may be infinite.

Definition 3.1 (Interaction Tree Codatatype).

```
codatatype ('e, 'r) itree =
  Ret 'r | Sil "('e, 'r) itree" |  Vis "'e ↠ ('e, 'r) itree"
```

The **codatatype** command creates a type called **itree**, with two type parameters **'e** and **'r**, and three constructors. Type parameters **'e** and **'r** encode the sorts E and R. Constructor *Ret* represents a return value, and *Sil* is an internal event that evolves to a further ITree. A visible event choice (*Vis*) is represented by a partial function ($A \twoheadrightarrow B$) from events to ITrees, with a potentially infinite domain. For example, in Figure 1 at node (2), a visible event choice is presented whose domain is $\{Input.x \mid x \in \mathbb{N}\} \cup \{State.[]\}$.

The representation of visible events is the main deviation from ITrees in Coq [66], which has visible events composed of output to the environment, followed by the answer. The benefit of using a partial function is to allow a straightforward encoding of deadlock and external choice, where the ITree offers several events to the environment (for a more detailed comparison, see §8). Moreover, a side effect of this design decision is that we only need rank-1 polymorphism for the encoding, which makes the development in Isabelle possible.

We also use the notation $\lambda\, c.x \mid B(x) \bullet P(x)$, which pattern matches on events over channel c, whose parameters x also satisfy predicate B. With this notation, we can describe the main choice block of the buffer example:

Example 3.2 (Buffer: Single Step ITree).

$$BufBody(buf) \triangleq Vis \left(\begin{array}{l} (\lambda\, Input.x \bullet Ret(buf \mathbin{@} [x])) \\ \oplus\, (\lambda\, Output.v \mid \#buf > 0 \wedge v = hd(buf) \bullet Ret(tl(buf))) \\ \oplus\, (\lambda\, State.s \mid s = buf \bullet Ret(buf)) \end{array} \right)$$

This constructs a visible event choice over a partial function composed of three parts using the override operator ($\oplus$). Here, parameter *buf* is a list of natural numbers, which is the current contents of the buffer. The first function accepts a value x over channel *Input* and returns the buffer with x appended. The second function allows us to output a value v over *Output*, but only when the buffer is non-empty. Then, v is the head of the buffer, and the function returns the contracted buffer. The third function allows us to advertise the current values in the buffer but leaves the buffer unchanged. Since the three functions have disjoint domains, they can be commuted over the override operator. Such an example is encoded more naturally using the *Circus* operators, but we defer denoting these to §5.

We sometimes use $\checkmark_v$ to denote *Ret* v, τP to denote *Sil* P, and $[]\ e \in E \rightarrow P(e)$ to denote *Vis* $(\lambda\, e \in E \bullet P(e))$, which are more concise and suggestive of their process algebra equivalents. We write $e_1 \rightarrow P_1\ [] \cdots []\ e_n \rightarrow P_n$ for an enumerated choice with $E = \{e_1, \cdots, e_n\}$. We use $\tau^n P$ for an ITree prefixed by $n \in \mathbb{N}$ internal events. We define $stop \triangleq Vis\,\{\mapsto\}$, a deadlock situation where no event is possible. An example is $a \rightarrow \tau(\checkmark_x)\ []\ b \rightarrow stop$, which can either perform the event a followed by a τ, and then terminate returning x, or perform the event b and then deadlock.

We call an ITree *unstable* if it has the form τP, and *stable* otherwise. The ITree in Figure 1 is stable in nodes (2), (5), and (7) and unstable in all other numbered nodes. An ITree stabilises, written $P \Downarrow$, if it becomes stable after a finite sequence of τ events, that is $\exists\, n\, P' \bullet P = \tau^n P' \wedge stable(P')$. An ITree that does not stabilise is divergent, written $P \Uparrow \triangleq \neg(P \Downarrow)$. We call an ITree *pure* if it has the form of $\tau^n P$, where P has the form of either $\checkmark_x$, *stop*, or *diverge*. The external environment cannot influence a pure ITree, which must either terminate, deadlock (abort) or diverge.

3.2 ITree Combinators

Using the constructors mentioned so far, we can only specify ITrees of finite depth. Infinite ITrees can be specified using primitive corecursion [8], which is the dual of recursion but allows non-terminating productive definitions. We define such an ITree below:

Definition 3.3 (Divergent ITree).

```
primcorec div :: "('e, 's) itree" where "div = τ div"
```

The **primcorec** command creates a typed constant which obeys several corecursive equations (following the **where**). Each definition requires that a constructor guards every corecursive call on the right-hand side of an equation, ensuring it is productive. This means that, though the definition does not terminate, it is always possible to strip off the next constructor.

ITree *div* represents the divergent ITree that does not terminate and only performs internal activity. Though self-referential and non-terminating, its definition is productive since we can always remove the next τ. Since *div* never stabilises, it is divergent, $div\Uparrow$. Moreover, we can show that *div* is the unique fixed-point of τ^{n+1} for any $n \in \mathbb{N}$, $\tau^{n+1} P = P \Leftrightarrow P = div$, and consequently *div* is the only divergent ITree: $P\Uparrow \rightarrow P = div$.

We give another infinite ITree below:

Definition 3.4 (Run ITree).

```
primcorec run :: "'e set ⇒ ('e, 's) itree" where
"run E = Vis (map_pfun (λ x. run E) (pId_on E))"
```

Here, the type `'e set` denotes the set of all subsets of type `'e`. ITree *run* E can repeatedly perform any $e \in E$ without ceasing. It has the equivalent definition of $run\, E \triangleq []\, e \in E \rightarrow run\, E$, an ITree that can repeatedly choose any event in E. It also has the case $run\, \emptyset = stop$. The formulation above uses the function `map_pfun :: ('b ⇒ 'c) ⇒ ('a ↛ 'b) ⇒ ('a ↛ 'c)` which maps a total function over every output of a partial function. The function `pId_on E` is the identity partial function with domain `E`. This formulation is required to satisfy the syntactic guardedness requirements. For the sake of readability, we omit these details in the following definitions.

Corecursive definitions can have several equations ordered by priority, like a recursive function. Using such a set of equations, we specify a monadic bind operator for ITrees [66].

Definition 3.5 (Interaction Tree Bind). We fix $P, P' : (E, R)\, itree$, $K : R \Rightarrow (E, S)\, itree$, $r : R$, and $F : E \nrightarrow (E, S)\, itree$. Then, $P \gg\!= K$ is defined corecursively by the equations

$$\begin{aligned} \checkmark_r \gg\!= K &= K\, r \\ \tau P' \gg\!= K &= \tau(P' \gg\!= K) \\ Vis\, F \gg\!= K &= Vis\, (\lambda\, e \in \mathrm{dom}(F) \bullet F(e) \gg\!= K) \end{aligned}$$

The intuition of $P \gg\!= K$ is to execute P, and whenever it terminates ($\checkmark_r$), pass the given value r on to the continuation K, yielding $K\, r$. If the first ITree can perform a τ event, this is performed first, and the remaining ITree is bound to K. If the first ITree can perform a visible event $e \in \mathrm{dom}(F)$, then we perform e, pass this on to F, and bind the result to K.

We term K a Kleisli tree [66], or KTree since it is a Kleisli lifting of an ITree. KTrees are important for defining processes that depend on a previous state. For this, we define the type synonym $(E, S)\, htree \triangleq (S \Rightarrow (E, S)\, itree)$ for a homogeneous KTree. For example, *BufBody* is homogeneous Kleisli tree of type $int\ list \Rightarrow (E, int\ list)\, itree$. Intuitively, the construction $K(s)$ can be read as "the Klesli tree K started in the initial state s.". We define the Kleisli

composition operator $P \fatsemi Q \triangleq (\lambda\, x.\, P\, x \gg\!= Q)$, symbolised because it is used as a sequential composition. Bind satisfies several algebraic laws:

Theorem 3.6 (Interaction Tree Bind Laws).

$$\begin{aligned} \mathit{Ret}\, x \gg\!= K &= K\, x & \qquad \mathit{Ret} \fatsemi K &= K \\ P \gg\!= \mathit{Ret} &= P & \qquad K \fatsemi \mathit{Ret} &= K \\ P \gg\!= (Q \fatsemi R) &= (P \gg\!= Q) \gg\!= R & \qquad K_1 \fatsemi (K_2 \fatsemi K_3) &= (K_1 \fatsemi K_2) \fatsemi K_3 \\ \mathit{div} \gg\!= K &= \mathit{div} & \qquad \mathit{run}\, E \gg\!= K &= \mathit{run}\, E \end{aligned}$$

Bind satisfies the three monad laws: it has *Ret* as left and right units and is essentially associative. Moreover, both *div* and *run* are left annihilators for bind since they do not terminate. The monad laws show that $(\fatsemi, \mathit{Ret})$ also forms a monoid; $\fatsemi$ is commutative and has *Ret* as its left and right units.

The laws of Theorem 3.6 are proved by coinduction, using the following derivation rule.

Theorem 3.7 (ITree Coinduction). *We fix a relation* $\mathcal{R} : (E, R)\mathit{itree} \leftrightarrow (E, R)\mathit{itree}$. *Then, given* $(P, Q) \in \mathcal{R}$ *we can deduce* $P = Q$ *provided that the following conditions hold:*

(1) $\forall (P', Q') \in \mathcal{R}.\ \mathit{is\_Ret}(P') = \mathit{is\_Ret}(Q') \wedge \mathit{is\_Sil}(P') = \mathit{is\_Sil}(Q') \wedge \mathit{is\_Vis}(P') = \mathit{is\_Vis}(Q')$;
(2) $\forall (x, y).\ (\mathit{Ret}\, x, \mathit{Ret}\, y) \in \mathcal{R} \longrightarrow x = y$;
(3) $\forall (P', Q')\ (\mathit{Sil}\, P', \mathit{Sil}\, Q') \in \mathcal{R} \longrightarrow (P', Q') \in \mathcal{R}$;
(4) $\forall (F, G)\ (\mathit{Vis}\, F, \mathit{Vis}\, G) \in \mathcal{R} \longrightarrow (\mathrm{dom}(F) = \mathrm{dom}(G) \wedge (\forall\, e \in \mathrm{dom}(F) \bullet (F(e), G(e)) \in \mathcal{R}))$.

To show that $P = Q$, we need to construct a (strong) bisimulation relation $\mathcal{R}$, which intuitively relates two ITrees, and show that $(P, Q) \in \mathcal{R}$. There are four provisos to show that $\mathcal{R}$ is a bisimulation. The first requires that only ITrees of the same kind are related; that is, a *Ret* is only related to a *Ret*, a *Sil* with a *Sil*, and a *Vis* with a *Vis*. Here, *is_Ret*, *is_Sil*, and *is_Vis* distinguish the three cases. The second proviso states that if $(\checkmark_x, \checkmark_y) \in \mathcal{R}$ then $x = y$, two related ITrees must return equal values. The third proviso states that internal events must yield bisimilar continuations: $(\tau P, \tau Q) \in \mathcal{R} \longrightarrow (P, Q) \in \mathcal{R}$. The final proviso states that for two visible interactions, the two functions must have the same domain $(\mathrm{dom}(F) = \mathrm{dom}(G))$, and every event $e \in \mathrm{dom}(F)$ must lead to bisimilar continuations. Most of our ITree proofs in Isabelle apply this law and then use a mixture of equational simplification and automated reasoning with *sledgehammer* to generate proofs that discharge the resulting provisos.

Next, we define an operator for iterating ITrees in the style of a while-loop:

Definition 3.8 (Iteration).

```
corec while :: "('s ⇒ bool) ⇒ ('e, 's) htree ⇒ ('e, 's) htree" where
"while b P s = (if (b s) then Sil (P s ≫= while b P) else Ret s)."
```

This is not primitively corecursive since the corecursive call uses $\gg\!=$, and so we define it using the **corec** command [7, 10] instead of **primcorec**. This requires us to show that $\gg\!=$ is a "friendly" corecursive function [7]: it consumes at most one input constructor to produce one output constructor. A while loop iterates whilst the condition b is satisfied by state s. In this case, a τ event is followed by the loop body and the corecursive call. If the condition is false, the current state is returned. We introduce the exceptional cases $\mathit{loop}\, F \triangleq \mathit{while}\, (\lambda\, s.\, \mathit{True})\, F$ and $\mathit{iter}\, P \triangleq \mathit{loop}\, (\lambda\, s.\, P)\, ()$, which represent infinite loops with and without state, respectively. We can show that $\mathit{iter}\, (\checkmark_{()}) = \mathit{div}$ since it never terminates and has no visible behaviour.

With *while*, we can easily complete the definition of the buffer: $Buffer \triangleq while\ BufBody\ []$. This iterates the buffer body in Example 3.2 over and over to provide the complete ITree shown in Figure 1. The initial empty state of the buffer is provided with the parameter [].

3.3 Structural Operational Semantics and Weak Bisimulation

The ITree model allows us to naturally describe structural operational semantics for our abstract language. We give a big-step operational semantics to ITrees using an inductive predicate.

Definition 3.9 (Big-Step Operational Semantics).

$$\frac{-}{P \xrightarrow{[]} P} \qquad \frac{P \xrightarrow{tr} P'}{\tau P \xrightarrow{tr} P'} \qquad \frac{e \in E \quad F(e) \xrightarrow{tr} P'}{([]\ x \in E \bullet F(x)) \xrightarrow{e\#tr} P'}$$

The relation $P \xrightarrow{tr} Q$ means that P can perform the trace of visible events contained in the list $tr : E\ list$ and evolve to the ITree Q. This relation skips over τ events. The first rule states that any ITree may perform an empty trace ([]) and remain in the same state. We sometimes omit the trace and write $P \rightarrow P'$. The second rule states that if P can evolve to P' by performing tr, then so can τP. The final rule states that if e is an enabled visible event, and $P(e)$ can evolve to P' by doing tr, then the event choice can evolve to P' via $e\#tr$, which is tr with e inserted at the head. This inductive predicate is different from the trace predicate (`is_trace_of`) in [66], since $P \xrightarrow{tr} P'$ records both the trace and the continuation ITree. It is, therefore, more general and provides the foundation for characterising structural operational and denotational semantics.

We next prove some important theorems of the transition relation.

Theorem 3.10 (Transition Relation Properties).

$$(P \xrightarrow{tr_1} Q \wedge Q \xrightarrow{tr_2} R) \rightarrow (P \xrightarrow{tr_1 \,@\, tr_2} R) \qquad \text{(sequential transitions)}$$

$$(P \xrightarrow{tr} Vis\, F \wedge P \xrightarrow{tr\,@\,[e]} P') \rightarrow e \in \mathrm{dom}(F) \qquad \text{(events resolve choices)}$$

$$(P \xrightarrow{tr} Ret\, x \wedge P \xrightarrow{tr} Ret\, y) \rightarrow x = y \qquad \text{(termination is deterministic)}$$

$$(P \xrightarrow{tr_1} Q \wedge tr_2 \leq tr_1) \rightarrow (\exists\, R.\, P \xrightarrow{tr_2} R) \qquad \text{(prefix closure)}$$

A pair of sequential transitions can be combined by appending the two traces, tr_1 and tr_2. Whenever an event e follows a visible choice over F, that event must have been enabled by F. If we can reach two return ITrees by the same trace, then the two values returned must be equal – termination is deterministic. Finally, whenever P can reach Q by performing tr_1, every prefix of tr_2 must also have an intermediate successor ITree R.

With these laws, we can prove the usual operational laws for sequential composition as theorems:

Theorem 3.11 (Sequential Operational Semantics).

$$\frac{-}{skip \rightarrow \checkmark_{()}} \qquad \frac{P \xrightarrow{tr} P'}{(P \gg Q) \xrightarrow{tr} (P' \gg Q)} \qquad \frac{P \xrightarrow{tr_1} \checkmark_x \quad Q(x) \xrightarrow{tr_2} Q'}{(P \gg Q) \xrightarrow{tr_1 \,@\, tr_2} Q'}$$

The *skip* process immediately terminates, returning (). If the left-hand side P of $\gg$ can evolve to P' performing the events in tr, the overall bind evolves similarly. If P can terminate after doing tr_1, returning x, and the continuation $Q(x)$ can evolve over tr_2 to Q' then the overall $\gg$ can also evolve over the concatenation of tr_1 and tr_2, $tr_1 \,@\, tr_2$, to Q'.

Strong bisimulation is a useful equivalence, but we often wish to abstract over τs. We, therefore, also introduce weak bisimulation, $P \approx Q$, as a coinductive-inductive predicate. Given a relation $\mathcal{R}$, we define $\approx_{\mathcal{R}}$ inductively:

Definition 3.12 (Weak Bisimulation).

$$\frac{-}{\checkmark_x \approx_{\mathcal{R}} \checkmark_x} \qquad \frac{P \approx_{\mathcal{R}} Q}{\tau P \approx_{\mathcal{R}} Q} \qquad \frac{P \approx_{\mathcal{R}} Q}{P \approx_{\mathcal{R}} \tau Q} \qquad \frac{\forall\, e \in E \bullet \mathcal{R}(F(e), G(e))}{(\Box\, x \in E \bullet F(x)) \approx_{\mathcal{R}} (\Box\, x \in E \bullet G(x))}$$

$$(\approx) \triangleq \bigcup\{\mathcal{R} \mid \mathcal{R} \subseteq \{(\mathit{div}, \mathit{div})\} \cup (\approx_{\mathcal{R}})\}$$

It requires us to construct a relation $\mathcal{R}$ such that whenever (P, Q) in $\mathcal{R}$ both stabilise, all their visible event continuations are also related by $\mathcal{R}$. For example, $\tau^m\, P \approx \tau^n\, Q$ whenever $P \approx Q$. We have proved that $\approx$ is an equivalence relation, and $P \approx \mathit{div} \rightarrow P = \mathit{div}$.

3.4 Iteration Chains

To reason about iteration ($\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}$), as usual, we need to characterise iteration chains. This, for example, is necessary for us to verify the properties of the buffer example. A chain is typically a sequence of states reached during an iteration's successive stages. For ITrees, we also need to consider the events that occur during iteration.

We adopt the notation $s \vdash P \xrightarrow{chn}{}^{*} s'$ to mean that state $s' :: S$ can be reached when the loop body $P :: (E, S)\mathit{htree}$ is started in state $s :: S$, by following the chain chn. Here, $chn :: (E\ \mathit{list} \times S)\ \mathit{list}$ is a list of trace and state pairs, each element of which denotes a single terminating execution of P. The formal definition of an iteration chain is given using the inductive predicate below.

Definition 3.13 (Iteration Chains).

$$\frac{-}{s \vdash P \xrightarrow{[]}{}^{*} s} \qquad \frac{P(s) \xrightarrow{tr} \checkmark_{s_0} \qquad s_0 \vdash P \xrightarrow{chn}{}^{*} s_1}{s \vdash P \xrightarrow{(tr, s_0) \# chn}{}^{*} s_1}$$

This does not yet consider the loop condition, which will be added subsequently. The first rule states that P can complete execution at state s by performing zero iterations starting in s. This occurs when the condition of a loop is false initially. The second rule allows a chain extension by a single execution of P. If P, when started in state s, terminates in the intermediate state s_0 having performed the trace tr, and P can further transition to s_1, when started from s_0 via chain chn, then we can prefix chn with the element (tr, s_0). For example, s_0 may result from the loop body's first iteration with the trace tr, and then chn characterises all subsequent iterations.

Next, we use chains to define a partial iteration $(b, s) \vdash P \xrightarrow{tr}{}^{*}_{\checkmark} s'$, which intuitively means that P is executed several times, starting in s and reaching s', whilst yielding trace tr. Moreover, in each intermediate state, the condition b remains satisfied. We define this operator directly using iteration chains:

Definition 3.14 (Partial Iteration).

$$(b, s) \vdash P \xrightarrow{tr}{}^{*}_{\checkmark} s' \Leftrightarrow \exists (chn, s_0, tr_0) \bullet \left(\begin{array}{l} b(s) \wedge s \vdash P \xrightarrow{chn}{}^{*} s_0 \wedge (\forall\, s \in \mathit{states}(chn) \bullet b(s)) \\ \wedge\ P(s_0) \xrightarrow{tr_0} \checkmark_{s'} \wedge\ tr = \mathit{trace}(chn)\ @\ tr_0 \end{array} \right)$$

Here, the function *states* extracts the set of all states a chain encounters, and *trace* is the concatenated trace described by the whole chain. Using this definition, we can now state the main theorem for reasoning about terminating loops.

Theorem 3.15 (Terminating Loops). 

$$(\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od})(s) \xrightarrow{tr} \checkmark_{s'} \ \Leftrightarrow\ (\neg b(s) \wedge s = s' \wedge tr = []) \vee \left(b(s) \wedge (b, s) \xrightarrow{tr}{}^{*}_{\checkmark}\ s' \wedge \neg b(s')\right)$$

If a loop terminates in state s', when started in initial state s, then there are two possibilities. Firstly, s does not satisfy the condition, so s' is the same as s, and an empty trace is emitted. Secondly, s does satisfy the condition; there is a partial iteration from P to s' emitting tr, and s' does not satisfy the condition. In other words, the loop is executed several times, with each intermediate satisfying b, and ends in a state that exits the loop. A consequence of this theorem is that a chain leads to the existing terminating state whenever a loop terminates. This theorem equips us to reason about the partial and total correctness of programs in §4.

The proof of this theorem is complex and requires induction on the structure of the transition relation in Definition 3.9. Our approach is to show that every transition of an iteration leads to an ITree of the form $Q \gg\!= \mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}$, that is a prefixed iteration, where the prefix Q is a partial execution of the loop body. The interested reader is directed to our proofs in Isabelle/HOL, which total about 300 lines of Isar.

We have now completed the foundational mechanisation of ITrees. In the next section, we will apply our theory to the modelling and verification of imperative programs before further considering reactive and concurrent programs in §5.

4 IMPERATIVE PROGRAMS AND AXIOMATIC SEMANTICS

This section builds on ITrees to develop a theory of Dijkstra-style imperative programs and an associated Hoare logic for partial and total correctness, which can be used to verify programs. The language is implemented as a shallow embedding in Isabelle/ITrees, thus maximising the scope for proof automation. We also develop the weakest precondition calculus and a link with a UTP-style predicative semantics [40], which provides the basis for a refinement calculus.

4.1 Modelling Imperative Programs

Imperative programs can be modelled as homogeneous Kleisli trees, $\mathcal{S} \Rightarrow (E, \mathcal{S})\mathit{itree}$, where $\mathcal{S}$ is the program's store type. Programs are typically pure for every initial state, meaning they depend only on their internal store for computation. An exception is nondeterministic programs, which we model using a special event to resolve any internal choices (see §4.3).

The store of an imperative program consists of a finite set of mutable state variables. In our work [25, 26, 29], each state variable is modelled as a lens [23], $x :: \mathcal{V} \Longrightarrow \mathcal{S}$, where $\mathcal{V}$ is the variable's type, and $\mathcal{S}$ is the store type. A lens is a pair of functions $\mathit{get} :: \mathcal{V} \Rightarrow \mathcal{S}$ and $\mathit{put} :: \mathcal{S} \Rightarrow \mathcal{V} \Rightarrow \mathcal{S}$, which query and update the variables present in state $\mathcal{S}$, and satisfy intuitive algebraic laws [25]. They allow an abstract representation of stores, where no explicit model is required to support the laws of programming [39]. Lenses can be designated as independent, $x \bowtie y$, meaning they refer to different regions of $\mathcal{S}$.

An expression or assertion over the state variables is a function $e :: \mathcal{S} \Rightarrow \mathcal{V}$, where $\mathcal{V}$ is the return type. For example, if x and y are state variables, then the expression $x + y$ is denoted by $\lambda\, s.\, \mathit{get}_x\, s + \mathit{get}_y\, s$. This function retrieves the values of x and y from the state s and adds them together. We can check whether an expression e uses a lens x using unrestriction, written $x \sharp e$. If $x \sharp e$, then e does not use x in its valuation, for example $x \sharp (y + 1)$, when

$x \bowtie y$. Updates to variables can be expressed as a sequence of maplets using the notation $[x_1 \rightsquigarrow e_1, x_2 \rightsquigarrow e_2, \cdots]$, with $x_i :: \mathcal{V}_i \Longrightarrow \mathcal{S}$ and $e_i :: \mathcal{S} \Rightarrow \mathcal{V}_i$, which represents a function $\mathcal{S} \Rightarrow \mathcal{S}$.

Creation of program store types is facilitated by the **zstore** command in Isabelle/HOL:

```
zstore S = x1::T1  x2::T2 ... xn::Tn where "P(x1, x2, ..., xn)"
```

This generates a set of lenses $x_1 \cdots x_n$, which have type $T_i \Longrightarrow S$, for $i \in \{1..n\}$. An independence property is also generated for each pair of lenses: $x_i \bowtie x_j$ where $i \neq j$. Our expression parser automates the lifting of terms containing such lenses so that expressions like $x + y$ are semantically interpreted as $\lambda\, s.\, get_x\, s + get_y\, s$. Options invariant predicates following the **where** clause can also accompany store types. Internally, a store is compiled into a record type S with a collection of lenses and an invariant assertion $S\_inv : S \Rightarrow \mathbb{B}$.

We can now denote the operators of an idealised imperative programming language. Sequential composition is modelled by Kleisli composition ($P \fatsemi Q$). The remaining operators are given below:

Definition 4.1 (Imperative Program Operators).

$$\begin{aligned}
C_1 \lhd P \rhd C_2 &\triangleq (\lambda\, s.\, \mathit{if}\ P(s)\ \mathit{then}\ C_1(s)\ \mathit{else}\ C_2(s)) \\
\langle \sigma \rangle &\triangleq (\lambda\, s.\, \mathit{Ret}(\sigma(s))) \\
x := e &\triangleq \langle [x \rightsquigarrow e] \rangle \\
\mathit{Skip} &\triangleq \langle [\rightsquigarrow] \rangle \\
\mathit{Stop} &\triangleq (\lambda\, s.\, \mathit{stop}) \\
\mathit{Div} &\triangleq (\lambda\, s.\, \mathit{div}) \\
¿P? &\triangleq \mathit{Skip} \lhd P \rhd \mathit{Stop}
\end{aligned}$$

$C_1 \lhd P \rhd C_2$ is our algebraic notation for a conditional statement (if-then-else), where P is the condition. Operator $\langle \sigma \rangle$ lifts a function $\sigma : \mathcal{S} \Rightarrow \mathcal{S}$ to a KTree. It is principally used to represent assignments, which can be constructed using our maplet notation, such that a single assignment $x := e$ is $\langle [x \rightsquigarrow e] \rangle$. Since substitutions can assign multiple variables, they can also represent simultaneous assignment, $(x, y) := (e, f)$. Similarly, the vacuous *Skip* statement is denoted by an empty assignment. *Stop* is simply a Kleisli-lifted version of the ITree *stop*, which deadlocks (or aborts) in any initial state, and *Div* similarly diverges in every initial state. Finally, ¿P? is a test operator, which deadlocks when P is false and otherwise has no effect. These operators satisfy all the usual laws of programming [39], a small selection of which is shown below. These laws give equational algebraic semantics for imperative programs.

Theorem 4.2 (Laws of programming).

$$\begin{aligned}
\mathit{Skip} \fatsemi C = C \fatsemi \mathit{Skip} &= C \\
x := e \fatsemi y := f &= y := f \fatsemi x := e && \mathit{if}\ x \bowtie y, x \sharp f, y \sharp e \\
\langle \sigma \rangle \fatsemi \langle \rho \rangle &= \langle \rho \circ \sigma \rangle \\
x := e \fatsemi (C_1 \lhd P \rhd C_2) &= (x := e \fatsemi C_1) \lhd P[e/x] \rhd (x := e \fatsemi C_2) \\
C_1 \lhd P \rhd (C_2 \lhd P \rhd C_3) &= C_1 \lhd P \rhd C_3
\end{aligned}$$

Skip is the unit of sequential composition. Two variable assignments commute provided their variables are independent ($x \bowtie y$), and their respective expressions do not depend on the adjacent variable. More generally, the sequential composition of two state updates σ and ρ entails their functional composition. Assignment can be pushed into a conditional by first substituting the assignment into the condition P. Finally, an outer conditional masks an inner one, meaning that C_2 is an unreachable branch. Such laws can be used for symbolic execution and optimisation of imperative programs.

4.2 Concrete and Symbolic Execution

A particular benefit of our ITree-based semantics is that imperative programs can be directly executed. A non-divergent and non-aborting pure ITree reduces to the form of $\tau^n\,(\checkmark_{s'})$, for $n \in \mathbb{N}$, where s' is the final state of the program. This is a particular case of a stable ITree. Consequently, an imperative program can be executed by supplying an initial state s and stripping off all the τs (internal steps) until s' is reached. If the program is divergent (i.e., non-terminating), it will never get a final state, so that that execution will hang.

To aid the modelling of programs in our tool, we provide the following command:

Definition 4.3 (Program command).

$$[\![\begin{array}{l} \textbf{program}\ Pr(x_1 :: T_1, \cdots, x_n :: T_n) \\ \quad \textbf{over}\ \mathcal{S} = Body(x_1, \cdots, x_n) \end{array}]\!] = \left(\begin{array}{l} Pr :: T_1 \times \cdots \times T_n \Rightarrow \mathcal{S} \Rightarrow (E, \mathcal{S})\,\mathit{htree} \\ Pr \triangleq (\lambda(x_1, \cdots, x_n).\ Body(x_1, \cdots, x_n)) \end{array} \right)$$

A program takes a tuple of parameters $(x_1, \cdots, x_n)$ and operates over a store $\mathcal{S}$. The program's body is a parametric ITree in $x_1 \cdots x_n$. These parameters are not program variables (lenses) but logical variables. Intuitively, they are constants that cannot be written to.

As an example, below is the definition of a simple imperative program for reversing a list:

Example 4.4 (List Reversal Program).

```
program reverse (xs :: int list) over state =
"ys := []; i := 0;
 while i < length xs
 do
    ys := xs!i # ys;
    i := i + 1
 od"
```

We define the program `reverse`, with input parameter `xs :: int list`. It operates over the store type `state`, containing the variables `i :: nat` and `ys :: int list`. The program iterates through the input `xs`, pushing each element on `ys`, with the result that `xs` is reversed.

We define a command **`execute`** that executes an ITree-based program with given arguments. It depends on the definition of a global constant called $MAX\_SIL\_STEPS :: \mathit{nat}$, an upper bound on the number of τ events that can be skipped over and acts as a timeout for execution. A program is executed using a function $un\_Sils\_n :: \mathit{nat} \Rightarrow (E, \mathcal{S})\,\mathit{itree} \Rightarrow (E, \mathcal{S})\,\mathit{itree}$, which strips a number n of τ events of an ITree, with $n \leq MAX\_SIL\_STEPS$. We then execute a program using Isabelle's evaluation mechanism, as present in the **`value`** command, which evaluates an executable term [2, 36]. The evaluator can perform concrete execution using the SML code generator and symbolic execution using normalisation by evaluation or the simplifier. The former is most efficient, and so is the default behaviour for **`execute`**.

An execution can produce one of four possible results: (1) termination with a final state, (2) an abort, (3) a visible event, and (4) a timeout. A timeout occurs if *MAX_SIL_STEPS* τ events have occurred without producing a return or visible event. A termination results if execution encounters a *Ret* before reaching the maximum number of τs. This being the case, the interface displays the final state of each variable. For example, if we call **execute** `"reverse [1,2,3]"`, the command generates code, executes it, and then reports termination with the final state $[y \rightsquigarrow [3,2,1], i \rightsquigarrow 3]$.

Abortion occurs when an empty visible event is encountered (i.e. *stop*), following a finite number of τ events. Thus, if the execute command encounters a *Vis* constructor, it checks whether the choice function is empty. If it is empty, then the execution has aborted. Otherwise, it indicates that an event choice was encountered and goes no further. For ITrees that use visible events, we typically cannot use such a non-interactive execution. We must instead rely on animation (§6), which allows further user input when a visible event is presented.

4.3 Nondeterminism

The operators given so far allow us to model only deterministic programs, which typically reduce to pure functions on the state. However, nondeterminism is useful both as a specification device and where design choices are deferred. Nondeterministic decisions can be encoded by introducing a special channel *nd*, which the environment can conceptually use to resolve, acting as an oracle. Here, I is an index type, which denotes the maximum cardinality of any choices. Whilst, in theory, I can be any type, we can typically only animate countable choices, and therefore, for now, we set $I \subseteq \mathbb{N}$. We can now use this to define the internal choice operator.

Definition 4.5 (Countable nondeterminism).

Assume a channel *nd* carrying a value of type $\mathbb{N}$ and a set $I \subset \mathbb{N}$ exists. Then, we encode nondeterministic choice as

$$\bigsqcap_{i \in I} \bullet C(i) \triangleq \mathit{Vis}\,(\lambda\, nd.i \mid i \in I \bullet P(i)).$$

This constructs a visible event choice over *nd* events parameterised by the elements of I. The particular index chosen is passed to P as a parameter. We can then define a binary nondeterministic choice as $C_1 \sqcap C_2 \triangleq \left(\bigsqcap_{i \in \{0,1\}} \bullet C_1 \lhd i = 0 \rhd C_2\right)$. Programs containing nondeterministic choices cannot be directly executed using the **execute** command, as the events must be resolved using animation (see §6).

4.4 Predicative Semantics and Refinement

We now focus on a predicative semantic interpretation for ITrees, which allows us to link with the established UTP predicative semantics for imperative programs [15, 40]. This semantics has many uses, but one particular use is to provide a notion of refinement for nondeterministic imperative programs.

UTP uses predicate calculus as a unifying language for programs and specifications. Dijkstra-style programs can be denoted as alphabetised relations, predicates that relate the initial values of variables to their final values. For example, assuming a store with three integer variables x, y, and z, an assignment $x := x + 1$ can be denoted by the predicate $x' = x + 1 \wedge y' = y \wedge z' = z$, where x is the initial value of x and x' is its final value.

Central to UTP is a notion of refinement $P \sqsubseteq Q$ for alphabetised relations P and Q, which means that Q is more deterministic or concrete than Q. For example, we can write the

specification $x' > x$, which means that in the final state, x should be strictly greater than its initial value. Then, using refinement, we can demonstrate that $x' > x \sqsubseteq x := x + 1$, meaning that the program on the right implements the specification on the left. The refinement order induces a complete lattice of alphabetised relations. It gives rise to fixed-point operators μF (least fixed point) and νF (greatest fixed point), which can specify iterative and recursive behaviour. UTP provides the predicative semantics for the *Circus* language [50].

To relate our ITree-based semantics with such predicative semantics, we must distinguish a program's terminating states from divergence. We can reason about termination and divergence with our transition relation, $P \xrightarrow{tr} Q$. Terminating imperative programs are characterised by pure ITrees that eventually reach a *Ret*. We define the set of return values of an ITree using the following function:

Definition 4.6 (Return Values). $\boldsymbol{R}(P) = \{x \mid \exists\, tr.\, P \xrightarrow{tr} \checkmark_x\}$.

$\boldsymbol{R}(P)$ induces the set of possible values a process P may return, whenever P terminates. In other words, $\boldsymbol{R}(P)$ is the set of reachable final states. If $\boldsymbol{R}(P) = \emptyset$, then P can never terminate. $\boldsymbol{R}(P)$ abstracts over all possible traces through existential quantification, and therefore, it does not distinguish return values that arise from different event interactions. All events are, therefore, effectively treated as nondeterminism in this semantic interpretation. Below, we give the valuations of $\boldsymbol{R}(P)$ for the main ITree constructors.

Theorem 4.7 (Return Values for ITree Constructors).

$$
\begin{aligned}
\boldsymbol{R}(\checkmark_x) &= \{x\} \\
\boldsymbol{R}(\tau P) &= \boldsymbol{R}(P) \\
\boldsymbol{R}(\mathit{Vis}(F)) &= \bigcup \{\boldsymbol{R}(P) \mid P \in \mathrm{ran}(F)\} \\
\boldsymbol{R}(P \gg\!= Q) &= \bigcup \{\boldsymbol{R}(Q(x)) \mid x \in \boldsymbol{R}(P)\} \\
\boldsymbol{R}(\mathit{stop}) &= \boldsymbol{R}(\mathit{div}) = \emptyset
\end{aligned}
$$

A *Ret* returns a single value. A *Sil* returns the values following the τ event. A visible event (*Vis*) returns all possible values returned by the continuation ITrees, $P \in \mathrm{ran}(F)$. If we view the ITree as a transition graph, we take the values returned on all paths. A bind $P \gg\!= Q$ first calculates the return values of P, then uses these as the possible inputs for Q, and calculates all the resulting return values. Neither *stop* or *div* have any return values because they do not successfully terminate. We can now use this function to provide a predicative semantic interpretation for ITrees.

Definition 4.8 (Predicative semantics). $[\![P]\!]_p = (\lambda(s, s').\, s' \in \boldsymbol{R}(P(s)))$

The function $[\![P]\!]_p$ induces a predicate of type $\mathcal{S} \times \mathcal{S} \Rightarrow \mathbb{B}$ for the homogeneous ITree P, which corresponds to a binary relation. Thus, $[\![P]\!]_p(s, s')$ holds whenever s' is reachable from the start state s. With this function, we can show that our imperative programs respect a UTP-style predicative semantics [40].

Theorem 4.9 (Predicative semantics of loop-free imperative programs).

$$
\begin{aligned}
[\![\langle\sigma\rangle]\!]_p(s, s') &= (s' = \sigma(s)) \\
[\![P \fatsemi Q]\!]_p(s, s') &= \big(\exists\, s_0 \bullet [\![P]\!]_p(s, s_0) \wedge [\![Q]\!]_p(s_0, s')\big) \\
[\![P \lhd B \rhd Q]\!]_p &= \big(\big(B(s) \wedge [\![P]\!]_p(s, s')\big) \vee \big(\neg B(s) \wedge [\![Q]\!]_p(s, s')\big)\big)
\end{aligned}
$$

$$[\![\mathit{Stop}]\!]_p(s, s') \;=\; [\![\mathit{Div}]\!]_p(s, s') \;=\; \mathit{False}$$
$$[\![P \sqcap Q]\!]_p(s, s') \;=\; \big([\![P]\!]_p(s, s') \vee [\![Q]\!]_p(s, s')\big)$$

A state update applies the update function to the initial state s to obtain the final state s'. The semantics of an assignment $x := e$ is a special case, conceptually $x' = e(s) \wedge y' = y$, for all other variables y in $\mathcal{S}$. The predicative semantics for $P \fatsemi Q$ yields the usual definition of relational composition: an intermediate state s_0, a final state for P and an initial state for Q. Conditional behaves as P when B is true in the initial state and Q otherwise. Both *Stop* and *Div* have the same interpretation *False*, as this semantics cannot distinguish between deadlock and divergence. Finally, a state pair is satisfied by a nondeterministic choice $P \sqcap Q$ if it is satisfied by either P or Q, which is the usual UTP interpretation of nondeterministic choice as disjunction [40].

Next, we consider the predicative semantics of iteration. First of all, we note the following corollary of Theorem 3.15:

Corollary 4.10 (Iteration return values).

$$\boldsymbol{R}((\mathit{while}\, b\, \mathit{do}\, P\, \mathit{od})(s)) = \{s' \mid (\neg P(s) \wedge s = s') \vee (\exists\, tr.\, B(s) \wedge (B, s) \xrightarrow{tr}{}^{*}_{\checkmark} s' \wedge \neg B(s'))\}$$

The return values for a loop started in state s is precisely the set of states for which there is a number of iterations of P yielding some trace tr. Whilst we could now express the predicative semantics in these terms, it is more convenient and concise to do this in terms of the reflexive transitive closure operation R^*. We first reiterate a result of the Isabelle/HOL standard library:

Theorem 4.11 (Reflexive Transitive Closure paths).

$$R^*(s, s') \Leftrightarrow s = s' \vee (\exists\, xs.\, \forall\, i < \mathit{length}(xs).\, R((s\#xs)!i, xs!i) \wedge x' = \mathit{last}(xs))$$

A pair of states (s, s') are related by R^* either when $s = s'$, or there is a path xs leading from s to s' through several iterations of R. Here, the path is a list of states, where each consecutive pair of states, starting from s and ending with s', are related by R. With this result, we can now express the predicative semantics of iteration:

Theorem 4.12 (Predicative semantics of iteration).

$$[\![\mathit{while}\, B\, \mathit{do}\, C\, \mathit{od}]\!]_p \;=\; (¿B?\,;\, [\![C]\!]_p)^*\,;\, ¿\neg B?$$

For conciseness, the predicate semantics for *while* is expressed point-free. The notation $¿P?$ denotes a test, i.e. $\lambda(s, s').\, P(s) \wedge s' = s$, which skips states that satisfy P. The semicolon operator $(P\,;\, Q)$ denotes relational composition, such that $[\![P \fatsemi Q]\!]_p = ([\![P]\!]_p\,;\, [\![Q]\!]_p)$. In this relational context, a *while* loop iterates C when B is true and ends when B is false. This corresponds to the usual Kleene algebra interpretation of iteration [3, 34].

The predicative interpretation in Theorem 4.9 induces a homomorphism between the ITree semantics and the relational semantics for each of the imperative programming operators (:=, $\fatsemi$, $\triangleleft\, b \,\triangleright$, etc.). This homomorphism is not only of theoretical interest but also practical benefit. Using the equations as code equations for the Isabelle/HOL code generator [36] allows us to employ the ITree semantics as a means to generate code for and execute relational imperative programs (see §6). We can also use our predicative semantics to obtain a notion of refinement for ITrees. We first recall the usual definition of refinement for relational programs in UTP:

Definition 4.13 (Predicative refinement). $(P \sqsubseteq Q) \triangleq (\forall (s, s').\, Q(s, s') \rightarrow P(s, s'))$

This is the usual UTP definition of refinement as a universally closed reverse implication. Specifically, P is refined by Q ($P \sqsubseteq Q$) if Q contains no more observable behaviours than P. Since we can interpret an ITree as a predicate, we can define $(P \sqsubseteq_p Q) \triangleq [\![P]\!]_p \sqsubseteq [\![Q]\!]_p$. In particular, we can now use refinement to reduce nondeterminism: $P \sqcap Q \sqsubseteq_p P$. This refinement relation forms a preorder, but it is not antisymmetric. This is because the predicative semantics is too coarse and does not, for example, distinguish *Stop* and *Div*, which are both *False*. For antisymmetry, we would need a finer predicative interpretation, such as the UTP theory of designs [15] or reactive designs [26], but this is out of the scope of this paper.

4.5 Hoare logic and Weakest Preconditions

We can now use our predicative interpretation of ITrees to define a partial correctness Hoare logic.

Definition 4.14 (Partial Correctness Hoare Logic).

$$\{\, P \,\}\, C \,\{\, Q \,\} \triangleq (\forall (s, s', tr) \bullet P(s) \wedge C(s) \xrightarrow{tr} \checkmark_{s'} \rightarrow Q(s'))$$

Whenever P is satisfied by initial state s, and C when started in s terminates in final state s', it follows that Q is satisfied by s'. This is partial correctness because we do not commit if C aborts or does not terminate. We can handle these additional aspects separately through deadlock-freedom and termination checks or by a total correctness Hoare logic. Our definition of the Hoare triple can alternatively be characterised directly using refinement in the UTP style, as the following theorem demonstrates.

Theorem 4.15. $\{\, P \,\}\, C \,\{\, Q \,\}$ *if and only if* $(P^{\bullet} \rightarrow Q^{\bullet}) \sqsubseteq [\![C]\!]_p$

Here, $P^{\bullet}$ and $Q^{\bullet}$ are shorthands for $\lambda(s, s').\, P(s)$ and $\lambda(s, s').\, Q(s')$ lift these predicate expressions to pre- and postconditions. We construct a relational specification for the program and then use it to assert a refinement. This allows us to obtain all the laws of Hoare logic for straight-line programs (cf. [25]), for example:

Theorem 4.16 (Hoare logic laws).

$$\frac{P \rightarrow Q[e/x]}{\{\, P \,\}\, x := e \,\{\, Q \,\}} \qquad \frac{\{\, P \,\}\, C_1 \,\{\, Q \,\} \qquad \{\, Q \,\}\, C_2 \,\{\, R \,\}}{\{\, P \,\}\, C_1 \fatsemi C_2 \,\{\, R \,\}} \qquad \frac{\{\, P \,\}\, C_1 \,\{\, Q \,\} \qquad \{\, P \,\}\, C_2 \,\{\, Q \,\}}{\{\, P \,\}\, C_1 \sqcap C_2 \,\{\, Q \,\}}$$

For while loops, using the construct introduced in Definition 3.8, there is a little more work to be done. Recall the partial correctness law for Hoare logic:

Theorem 4.17 (Partial Correctness While law).

$$\frac{\{\, P \wedge B \,\}\, C \,\{\, P \,\}}{\{\, P \,\}\, \mathit{while}\, B\, \mathit{do}\, C\, \mathit{od} \,\{\, \neg B \wedge P \,\}}$$

Here, P is the loop invariant, which must remain true whenever the body C is executed. We outline the mechanised proof below, which uses Theorem 3.15.

Proof. From Definition 4.14, we need to show that given an initial state s satisfying P, whenever $(\mathit{while}\, B\, \mathit{do}\, S\, \mathit{od})(s) \rightarrow \checkmark_{s'}$, then it follows that s' satisfies $\neg B$ and P (partial correctness). From Theorem 3.15, we know that the loop terminates immediately or executes several times. Suppose it terminates immediately, then clearly $P(s)$ and $\neg B(s)$. Suppose it executes, a chain *chn* leads to s' such that $\neg B(s')$. The premises of the loop invariant law

tell us that for any state s_0, such that $P(s_0)$ and $B(s_0)$, whenever $C(s_0) \rightarrow \checkmark_{s_1}$ then also $P(s_1)$. As a result, we can deduce that any $s_0 \in states(chn)$ and subsequent state s_1 must maintain the invariant. This being the case, it also particularly follows that $P(s')$, since s' is such an s_1 state. This completes the proof. □

In addition to Hoare logic, we can also characterise the weakest (liberal) preconditions:

Definition 4.18 (Weakest Preconditions).

$$wp\ C\ P \triangleq (\lambda\, s.\ \exists\, s'.\ [\![C]\!]_p(s, s') \wedge P(s'))$$
$$wlp\ C\ P \triangleq (\lambda\, s.\ \forall\, s'.\ [\![C]\!]_p(s, s') \rightarrow P(s'))$$

The weakest precondition $wp\ C\ P$ obtains the weakest precondition required for C to reach a state satisfying P. It formally requires that for any initial state s, there is a final state s', such that $P(s')$. In particular, we can use the weakest precondition to calculate the domain or "precondition" of a program: $pre(C) \triangleq wp\ C\ true$. For ITrees, this is the set of initial states that do not lead to deadlock or divergence. For imperative programs specifically, this can be considered the initial states for which the program terminates. The weakest liberal precondition is similar, but for any final state s' of C that $P(s')$ holds, it does not require such an s' exists. Both of these laws satisfy the standard laws [19], which we have previously presented for Isabelle/UTP [25].

As usual, we can use the simplifier to calculate the weakest preconditions for a program in Isabelle/HOL equationally. Moreover, we also prove the following standard theorem linking Hoare logic and *wlp*:

Theorem 4.19. $\{\, P \,\}\ C\ \{\, Q \,\} \Leftrightarrow (P \rightarrow wlp\ C\ Q)$

We can prove a Hoare triple by calculating the *wlp*, and then proving the precondition P satisfies the resulting predicate. Finally, we can use *wp* to define the total correctness Hoare triple:

Definition 4.20 (Total Correctness Hoare Logic).

$$[\, P \,]\ C\ [\, Q \,] \triangleq \big(\{\, P \,\}\ C\ \{\, Q \,\} \wedge (P \rightarrow pre(C)\big)$$

This follows the usual intuition of *total correctness = partial correctness + termination*. Here, $P \rightarrow pre(C)$ means that the precondition is a sufficient condition to ensure that C terminates. With this definition, we can obtain the corresponding laws to those in 4.16, and also the total correctness law for loops, which requires a decreasing variant expression V:

Theorem 4.21 (Total Correctness While law).

$$\frac{[\, P \wedge B \wedge V = z \,]\ C\ [\, P \wedge V < z \,]}{[\, P \,]\ while\ B\ do\ C\ od\ [\, \neg B \wedge P \,]}$$

The proof of this depends on Theorem 3.15.

We will make further use of the weakest preconditions when we develop our Z-Machine tool in Section 7. For now, we are turning our attention to the automation of program verification.

4.6 Verification Condition Generation

Automation of program verification is conducted, as usual, through a verification condition generator (VCG). Our VCG method repeatedly applies Hoare logic laws to obtain a collection of verification condition predicates. These predicates can often be discharged by Isabelle's

automated proof methods, like *blast*, *metis*, and *smt*, with the help of *sledgehammer*. If verification fails, we can find errors using counterexample finders, like *nitpick* and *quickcheck*.

For automated reasoning, we need laws that avoid the introduction of meta-variables, as these can introduce backtracking and hamper automation. For example, the general sequential composition law in Theorem 4.16 and iteration law in Theorem 4.17 introduce variables that only appear in the hypotheses, and a suitable witness must be supplied. Instead, we specialise the Hoare logic theorems to avoid this. In particular, we introduce the following two corollaries for assignment.

Corollary 4.22 (Forward and Backward Assignment Laws).

$$\frac{\{\, x = e[x_0/x] \wedge P[x_0/x] \,\}\ C\ \{\, Q \,\}}{\{\, P \,\}\ x := e \fatsemi C\ \{\, Q \,\}}\ x_0 \notin fv(e, P) \qquad \frac{\{\, P \,\}\ C\ \{\, Q[e/x] \,\}}{\{\, P \,\}\ C \fatsemi x := e\ \{\, Q \,\}}$$

The forward law allows us to push the effect of the assignment into the precondition. We introduce a new fixed logical variable, x_0, which stands for the initial value of x before the assignment occurred. We substitute x for x_0 in the assigned expression e and the precondition P. The backward law similarly applies the assignment to the postcondition.

VCG, as usual, depends on the annotation of loops with invariants. We adopt the approach of Armstrong et al. [3] and introduce the syntax *while* B *invariant* I *do* C *od*, which annotates with the invariant I. This annotation is semantically vacuous and exists only to help proof automation using the following derived law.

Theorem 4.23 (Loop Invariant Annotation).

$$\frac{\{\, I \wedge B \,\}\ C\ \{\, I \,\} \qquad P \to I \qquad \neg B \wedge I \to Q}{\{\, P \,\}\ \textit{while}\ B\ \textit{invariant}\ I\ \textit{do}\ C\ \textit{od}\ \{\, Q \,\}}$$

This uses requires we prove that I is an invariant of the loop body, I weakens precondition P, and I strengthens postcondition Q when the loop condition does not hold. The proof combines the consequence law and the partial correctness law for while loops. Similarly, we derive a corresponding total correctness law, which uses a variant annotation: *while* B *invariant* I *variant* V *do* C *od*, where V is the variant expression.

Finally, we implement the `vcg` proof method, which implements the following steps:

(1) Atomise assignments and conditionals where possible, using the Theorem 4.2;
(2) Repeatedly apply specialised Hoare logic laws as introduction rules to decompose goal;
(3) Evaluate substitutions in resulting expressions and convert them to HOL proof obligations.

The result is a set of VCs for which discharge can be attempted. We can now annotate our imperative list reversal program from Example 4.4 with an invariant and a variant to allow its verification:

Example 4.24 (Annotated List Reversal Program).

```
program reverse (xs :: int list) over state =
"ys := []; i := 0;
 while i < length xs
 invariant ys = rev (take i xs)
 variant length xs - i
 do
    ys := xs!i # ys;
```

```
    i := i + 1
 od"
```

We supply the invariant `ys = rev(take i xs)`, since `ys` is always the first `i` elements of `xs` in reverse. The functions `take` and `rev` are defined in Isabelle/HOL. The variant `length xs - i` counts down from the length of `xs` to zero.

With this, we want to prove the Hoare triple $[\ True\ ]\ reverse(xs)\ [\ ys = rev(xs)\ ]$: the imperative program satisfies the functional specification provided by the *rev* function. Applying the `vcg` method to this proof goal yields a single verification condition:

`xs ! i # rev (take i xs) = rev (take (i + 1) xs)` for `i < length(xs)`.

This states that taking the first `i + 1` elements of `xs` and then reversing it can be achieved by appending the `i`th element of `xs` at the beginning of the reversed `i` elements. This can be discharged by *sledgehammer* using the built-in laws from Isabelle/HOL. The variant proof is straightforward and discharged simply by the simplifier.

Although this is a trivial example, we have verified more substantial benchmark examples, such as sorting algorithms, with a high level of automation provided by *sledgehammer*.

5 REACTIVE AND CONCURRENT PROGRAMMING

In this section, we move on from imperative programs and give an ITree semantics to deterministic fragments of the CSP [13, 38] and *Circus* [50, 64] process languages. Our deterministic CSP fragment is consistent with the one identified by Roscoe [56, Section 10.5]. The standard CSP denotational semantics is provided by the failures-divergences model [13, 56], and we provide preliminary results on linking to this in §5.3.

5.1 CSP

CSP processes are parametrised by an event alphabet (Σ), which specifies the possible ways a process communicates with its environment. For ITrees, Σ is provided by the type parameter E. Whilst the event sort of an ITree E is typically infinite, in process algebraic languages, like CSP, it is usually expressed in a finite set of channels, which can carry data of various types. Here, we characterise channels abstractly using prisms [52], a concept well known in the functional programming world:

Definition 5.1 (Prisms). A prism is a quadruple $(\mathcal{V}, \Sigma, match, build)$ where $\mathcal{V}$ and Σ are non-empty sets. Functions $match : \Sigma \nrightarrow \mathcal{V}$ and $build : \mathcal{V} \Rightarrow \Sigma$ satisfy the following laws:

$$match(build\, x) = x \qquad y \in \mathrm{dom}(match) \longrightarrow build\,(match\, y) = y$$

We write $X : V \xrightarrow{\Delta} E$ if X is a prism with $\Sigma_X = E$ and $\mathcal{V}_X = V$.

Intuitively, a prism abstractly characterises a datatype constructor, E, taking a value of type $\mathcal{V}$. Then, *build* is the constructor, and *match* is the destructor, which is partial due to the possibility of several disjoint constructors. For CSP, each prism models a channel in E carrying a value of type $\mathcal{V}$. We have created a command **chantype**, which automates the creation of prism-based event alphabets. Technically this is achieved by creation of an algebraic data type, with a constructor for each channel, and corresponding prism for each constructor.

CSP processes typically do not return data, though their components may, and so they are typically denoted as ITrees of type $(E, ())itree$, returning the unit type (). An example is $skip \triangleq Ret\,()$, which is a degenerate form of *Ret*. We now define the basic CSP operators.

Definition 5.2 (Basic CSP Constructs).

$$\mathit{inp} :: (V \xrightarrow{\Delta} E) \Rightarrow V\ \mathit{set} \Rightarrow (E, V)\mathit{itree}$$
$$\mathit{inp}\ c\ A \triangleq \mathit{Vis}\ (\lambda\ e \in \mathrm{dom}(\mathit{match}_c) \cap \mathit{build}_c(\!|\ A\ |\!) \bullet \mathit{Ret}\ (\mathit{match}_c\ e))$$

$$\mathit{outp} :: (V \xrightarrow{\Delta} E) \Rightarrow V \Rightarrow (E, ())\mathit{itree} \qquad \mathit{guard}\ b :: \mathbb{B} \Rightarrow (E, ())\mathit{itree}$$
$$\mathit{outp}\ c\ v \triangleq \mathit{Vis}\ \{\mathit{build}_c\ v \mapsto \mathit{Ret}\ ()\} \qquad \mathit{guard}\ b \triangleq (\mathit{if}\ b\ \mathit{then}\ \mathit{skip}\ \mathit{else}\ \mathit{stop})$$

An input event ($\mathit{inp}\ c\ A$) permits any event over the channel c, that is $e \in \mathrm{dom}(\mathit{match}_c)$, provided that its parameter is in A ($e \in \mathit{build}_c(\!|\ A\ |\!)$). It returns the value received for use by a continuation. It corresponds to the `trigger` construct in [66]. With this and monadic bind, the usual CSP input prefix can be denoted as

$$c?x \rightarrow P(x) \triangleq (\mathit{inp}\ c\ \mathit{UNIV} \gg\!= P)$$

where UNIV is the set of all values of a particular type. The input prefix receives any value over c and then passes it on to P.

An output event ($\mathit{outp}\ c\ v$) permits a single event, v, on channel c and returns a null value of type (). We can then denote the standard CSP output prefix as

$$c!v \rightarrow Q \triangleq (\mathit{outp}\ c\ v \gg\!= (\lambda\ x.\ Q)$$

We also define the special case $\mathit{sync}\ e \triangleq \mathit{outp}\ e\ ()$ for a basic event $e :: () \xrightarrow{\Delta} E$. A $\mathit{guard}\ b$ behaves as skip if $b = \mathit{true}$ and otherwise deadlocks. It corresponds to the guard in CSP, which can be defined as $b\ \&\ P \triangleq (\mathit{guard}\ b \gg\!= (\lambda\ x.\ P))$.

Using the monadic "do" notation, which boils down to applications of $\gg\!=$, we can now write simple reactive programs such as $\mathit{do}\{x \leftarrow \mathit{inp}\ c;\ \mathit{outp}\ d\ (2 \cdot x);\ \mathit{Ret}\ x\}$, which inputs x over channel $c : \mathbb{N} \xrightarrow{\Delta} E$, outputs $2 \cdot x$ over channel d, and finally terminates, returning x.

Next, we define the external choice operator, $P \Box Q$, where the environment resolves the choice with an initial event of P or Q. In CSP, $\Box$ can also introduce nondeterminism; for example, $(a \rightarrow P) \Box (a \rightarrow Q)$ introduces an internal choice since the a event can lead to P or Q, and is equal to $a \rightarrow (P \sqcap Q)$. Since we explicitly wish to avoid introducing such nondeterminism, we make a design choice to exclude this possibility by construction. There are other possibilities for handling nondeterminism in ITrees, which we consider in §9. As for $\gg\!=$, we define external choice corecursively using a set of ordered equations.

Definition 5.3 (External choice). $P \Box Q$, is defined by the following set of equations:

$$(\mathit{Vis}\ F) \Box (\mathit{Vis}\ G) = \mathit{Vis}\ (F \odot G) \qquad (\mathit{Ret}\ x) \Box (\mathit{Vis}\ G) = \mathit{Ret}\ x$$
$$(\mathit{Sil}\ P') \Box Q = \mathit{Sil}\ (P' \Box Q) \qquad (\mathit{Vis}\ F) \Box (\mathit{Ret}\ y) = \mathit{Ret}\ y$$
$$P \Box (\mathit{Sil}\ Q') = \mathit{Sil}\ (P \Box Q') \qquad (\mathit{Ret}\ x) \Box (\mathit{Ret}\ y) = (\mathit{if}\ x = y\ \mathit{then}\ (\mathit{Ret}\ x)\ \mathit{else}\ \mathit{stop})$$

where $F \odot G \triangleq (\mathrm{dom}(G) \lhd F) \oplus (\mathrm{dom}(F) \lhd G)$

An external choice between two functions, F and G, essentially combines all the choices presented using $F \odot G$. The caveat is that if the domains of F and G overlap, then any events in common are excluded. Thus, $\odot$ restricts the domain of F to maplets $e \mapsto P$ where $e \notin \mathrm{dom}(G)$, and vice-versa. This has the effect that $(a \rightarrow P) \Box (a \rightarrow Q) = \mathit{stop}$, for example. In the special case that $\mathrm{dom}(F) \cap \mathrm{dom}(G) = \emptyset$, $P \odot Q = P \oplus Q$. We chose this behaviour to ensure that $\Box$ is commutative, though we could alternatively bias one side.

Internal steps on either side of $\Box$ are greedily consumed. Due to the equation order, τ events have the highest priority, following a maximal progress assumption [37]. Return events

also have priority over visible events. If two returns are present, then they must agree on the value. Otherwise, they deadlock. External choice satisfies several essential properties:

Theorem 5.4 (External Choice Properties).

$$P \,\Box\, Q = Q \,\Box\, P \quad \mathit{stop} \,\Box\, P = P \quad \mathit{div} \,\Box\, P = \mathit{div} \quad P \,\Box\, (\tau^n\, Q) = (\tau^n\, P) \,\Box\, Q = \tau^n (P \,\Box\, Q)$$

$$(\mathit{Vis}\, F \,\Box\, \mathit{Vis}\, G) \gg\!= H = (\mathit{Vis}\, F \gg\!= H) \,\Box\, (\mathit{Vis}\, G \gg\!= H)$$

The external choice is commutative and has *stop* as a unit. It has *div* as an annihilator because the τ events mean no other activity is chosen. A finite number of τ events on the left or right can be extracted to the front. Finally, bind distributes from the left across a visible event choice. We prove these properties using coinduction (Theorem 3.7), case analysis on stability of constituent processes, followed by several invocations of *sledgehammer* to discharge the resulting provisos.

Using the operators defined so far, we can implement the buffer from Examples 2.1 using a monadic syntax:

```
chantype Chan = Input::int  Output::int  State::"int list"

definition buffer :: "int list ⇒ (Chan, int list) itree" where
"buffer = loop (λ s.
                  do { i ← inp Input {0..}; Ret (s @ [i]) }
                □ do { guard(length s > 0); outp Output (hd s); Ret (tl s) }
                □ do { outp State s; Ret s })"
```

We first create a channel type `Chan`, which has channels (prisms) for inputs and outputs and to view the current buffer state. We define the buffer process as a simple loop with a choice of three branches inside. The variable `s::int list` denotes the state. The first branch allows a value to be received over `Input`, and then returns `s` with the new `i` value appended, and then iterates. The second branch is only active when the buffer is not empty. It outputs the head on `Output` and returns the tail. The final branch outputs the current state. In §6, we will see how such an example can be animated.

Next, we tackle parallel composition. The objective is to define the usual CSP operator $P \,[\![\, E \,]\!]\, Q$, which requires that P and Q synchronise on the events in E and can otherwise evolve independently. We first define an auxiliary operator for merging choice functions.

$$\begin{aligned} \mathit{merge}_E(F, G) = {} & (\lambda\, e \in \mathrm{dom}(F) \setminus (\mathrm{dom}(G) \cup E) \bullet \mathit{Left}(F(e))) \\ & \oplus (\lambda\, e \in \mathrm{dom}(G) \setminus (\mathrm{dom}(F) \cup E) \bullet \mathit{Right}(G(e))) \\ & \oplus (\lambda\, e \in \mathrm{dom}(F) \cap \mathrm{dom}(G) \cap E \bullet \mathit{Both}(F(e), G(e)) \end{aligned}$$

Operator $\mathit{merge}_E(F, G)$ merges two event functions, which are being offered by two parallel composed ITrees. Each event is tagged depending on whether it occurs on the *Left*, *Right*, or *Both* sides of a parallel composition. An event in $\mathrm{dom}(F)$ can occur independently when not in E or $\mathrm{dom}(G)$. The latter proviso is required, like for $\Box$, to prevent nondeterminism by disallowing the same event from occurring independently on both sides. An event in $\mathrm{dom}(G)$ can occur independently through the symmetric case for $\mathrm{dom}(F)$. An event can synchronise provided it is in the domain of choice functions and the set E. We use this operator to define the generalised parallel composition. For the sake of presentation, we present partial functions as sets.

Definition 5.5. $P \parallel_E Q$ is defined corecursively by the following equations:

$$(\mathit{Vis}\ F) \parallel_E (\mathit{Vis}\ G) = \mathit{Vis}\left(\begin{array}{l}\{e \mapsto (P' \parallel_E (\mathit{Vis}\ G)) \mid (e \mapsto \mathit{Left}(P')) \in \mathit{merge}_A(F, G)\} \\ \oplus \{e \mapsto ((\mathit{Vis}\ F) \parallel_E Q') \mid (e \mapsto \mathit{Right}(Q')) \in \mathit{merge}_E(F, G)\} \\ \oplus \{e \mapsto (P' \parallel_E Q') \mid (e \mapsto \mathit{Both}(P', Q')) \in \mathit{merge}_E(F, G)\}\end{array}\right)$$

$$(\mathit{Sil}\ P') \parallel_E Q = \mathit{Sil}\ (P' \parallel_E Q) \qquad P \parallel_E (\mathit{Sil}\ Q') = \mathit{Sil}\ (P \parallel_E Q')$$

$$(\mathit{Ret}\ x) \parallel_E (\mathit{Ret}\ y) = \mathit{Ret}\ (x, y)$$

$$(\mathit{Ret}\ x) \parallel_E (\mathit{Vis}\ G) = \mathit{Vis}\ \{e \mapsto \mathit{Ret}\ x \parallel_E Q' \mid (e \mapsto Q') \in G\}$$

$$(\mathit{Vis}\ F) \parallel_E (\mathit{Ret}\ y) = \mathit{Vis}\ \{e \mapsto P' \parallel_E \mathit{Ret}\ y \mid (e \mapsto P') \in F\}$$

The most complex case is for *Vis*, which constructs a new choice function by merging F and G. Three partial functions again represent the three cases. The first two allow the left and right to evolve independently to P' and Q', respectively, using one of their enabled events, leaving their opposing side, *Vis* G and *Vis* F, respectively, unchanged. The third case allows them both to evolve simultaneously on a synchronised event.

The *Sil* cases allow τ events to happen independently and with priority. If both sides can return a value, x and y, respectively, then the parallel composition returns a pair, which can later be merged if desired. The final two cases show what happens when only one side has a return value, and the other has visible events. In this case, the *Ret* value is retained and pushed through the parallel composition until the other side also terminates.

We use $\parallel_E$ to define two special cases for CSP: $P \,[\![\, E \,]\!]\, Q \triangleq (P \parallel_E Q) \gg\!= (\lambda(x, y).\ \mathit{Ret}\ ())$ and $P \mid\mid\mid Q \triangleq P \,[\![\, \emptyset \,]\!]\, Q$. As usual in CSP, these operators do not return any values, and so $P, Q :: (E, ())\,\mathit{itree}$. The $P \,[\![\, E \,]\!]\, Q$ operator is similar to $\parallel_E$, except if both sides terminate, any resultant values are discarded, and a null value is returned. This is achieved by binding to a simple merge function. P and Q do not return values, so this does not affect the behaviour, just the typing. The interleaving operator $P \mid\mid\mid Q$, where there is no synchronisation, is defined as $P \,[\![\, \emptyset \,]\!]\, Q$. We prove several algebraic laws:

$$(P \parallel_E Q) = (Q \parallel_E P) \gg\!= (\lambda(x, y).\ \mathit{Ret}\ (y, x)) \quad \mathit{div} \parallel_E P = \mathit{div}$$

$$P \,[\![\, E \,]\!]\, Q = Q \,[\![\, E \,]\!]\, P \quad P \mid\mid\mid Q = Q \mid\mid\mid P \quad \mathit{skip} \mid\mid\mid P = P$$

Parallel composition is commutative, except that we must swap the outputs, and so $[\![E]\!]$ and $\mid\mid\mid$ are commutative as well. Parallel has *div* as an annihilator for similar reasons to $\Box$. For $\mid\mid\mid$, *skip* is a unit since there is no possibility of communication and no values are returned.

The final operator we consider is hiding, $P \setminus A$, which turns the events in A into τs:

Definition 5.6 (Hiding). $P \setminus A$ is defined corecursively by the following equations:

$$\mathit{Vis}(F) \setminus A = \begin{cases} \mathit{Sil}\ (F(e) \setminus A) & \text{if } A \cap \mathrm{dom}(F) = \{e\} \\ \mathit{Vis}\ \{(e, P \setminus A) \mid (e, P) \in F\} & \text{if } A \cap \mathrm{dom}(F) = \emptyset \\ \mathit{stop} & \text{otherwise} \end{cases}$$

$$\mathit{Sil}(P) \setminus A = \mathit{Sil}(P \setminus A) \qquad \mathit{Ret}\ x \setminus A = \mathit{Ret}\ x$$

We consider a restricted version of hiding where only one event can be hidden at a time to avoid nondeterminism. When hiding the events of A in the choice function F, there are three cases: (1) there is precisely one event $e \in A$ enabled, in which case it is hidden; (2) no enabled event is in A, in which case the event remains visible; (3) more than one $e \in A$ is enabled, and so we deadlock. We again impose maximal progress here so that an enabled event to be hidden is prioritised over other visible events: $(a \rightarrow P \,[]\, b \rightarrow Q) \setminus \{a\} = \tau P$, for example. Despite the significant restrictions on hiding, it supports the typical pattern where

one output event is matched with an input event. Moreover, a priority can be placed on the order in which events are hidden, rather than deadlocking, by sequentially hiding events. Hiding can introduce divergence, as the following theorem shows: $(\mathit{iter}\ (\mathit{sync}\ e)) \setminus e = \mathit{div}$.

5.2 Circus

While CSP processes can be parametrised to allow modelling states, there is no support for explicit state operators like assignment. The *do* notation somewhat allows variables, but these are immutable and are not preserved across iterations. *Circus* [50, 64] is a CSP extension allowing state variables.

We can characterise *Circus* through a Kleisli lifting of CSP processes that return values so that *Circus* actions are homogeneous KTrees. Then, thanks to the compositionality of our ITree-based semantics, we can use the operators defined in §4, such as assignment $x := e$, to allow manipulation of the state. Then, we define the core operators for concurrency:

Definition 5.7 (Circus Operators).

$$\begin{aligned} c?x{:}A \rightarrow F(x) &\triangleq (\lambda\, s.\ \mathit{inp}\ c\ A \gg\!= (\lambda\, x.\ F(x)\ s)) \\ c!e \rightarrow P &\triangleq (\lambda\, s.\ \mathit{outp}\ c\ (e\ s) \gg\!= (\lambda\, x.\ P\ s)) \\ P \,\Box\, Q &\triangleq (\lambda\, s.\ P(s) \,\Box\, Q(s)) \\ P \,[\![ns_1|E|ns_2]\!]\, Q &\triangleq \big(\lambda\, s.\ (P(s) \parallel_E Q(s)) \gg\!= (\lambda(s_1, s_2).\ s \lhd_{ns_1} s_1 \lhd_{ns_2} s_2)\big) \end{aligned}$$

The operators are defined by the lifting of their CSP equivalents. An output $c!e \rightarrow P$ carries an expression e rather than a value, which can depend on the state variables. The main complexity is the *Circus* parallel operator, $P \,[\![ns_1|E|ns_2]\!]\, Q$, which allows P and Q to act on disjoint portions of the state, characterised by the name sets ns_1 and ns_2. We represent ns_1 and ns_2 as independent lenses, $ns_1 \bowtie ns_2$, though they can be thought of as sets of variables with $ns_1 \cap ns_2 = \emptyset$. The definition of the operator first lifts $\parallel_E$ and composes this with a merge function. The merge function constructs a state consisting of the ns_1 region from the final state of P, the ns_2 region from Q, and the remainder from the initial state s. This is achieved using the lens override operator $s_1 \lhd_X s_2$, which extracts the region described by X from s_2 and overwrites the corresponding region in s_1, leaving the complement unchanged.

We can now model the buffer from Example 2.1 with these operator definitions. Given a state variable `buf::int list`, the buffer can be expressed in Isabelle/HOL as follows:

Example 5.8. Buffer in ITree-based *Circus*

```
buf := [];
loop ((Input?(i) → buf := buf @ [i])
      □ (length(buf) > 0 & Output!(hd buf) → buf := tl buf)
      □ State!(buf) → Skip)
```

We update the state with assignments threaded through sequential composition.

Our *Circus* operators satisfy several standard laws [29, 50], beyond the CSP laws, for example:

$$\begin{aligned} \langle\sigma\rangle \fatsemi (P \,\Box\, Q) &= (\langle\sigma\rangle \fatsemi P) \,\Box\, (\langle\sigma\rangle \fatsemi Q) \\ P \,[\![ns_1|E|ns_2]\!]\, Q &= Q \,[\![ns_2|E|ns_1]\!]\, P \qquad \text{if } ns_1 \bowtie ns_2 \end{aligned}$$

State updates are distributed through external choice from the left. *Circus* parallel composition is commutative, provided that we also switch the name sets.

5.3 Denotational Semantics

Next, we show how ITrees are related to the standard failures-divergences semantics of CSP [13]. The utility of this link is to both allow symbolic verification of ITrees and allow them to act as a target of step-wise refinement. In this way, we can use the existing mechanisation of the CSP set-based and relational semantics [29, 60] to capture and reason about nondeterministic specifications and use ITrees to provide executable implementations.

In the failures-divergences model, a process is characterised by two sets: $F :: (E^{\checkmark}\ \mathit{list} \times E\ \mathit{set})\ \mathit{set}$ and $D :: \mathbb{P}(E\ \mathit{list})$, which are, respectively, the set of failures and divergences. A failure is a trace of events plus a set of events that can be refused at the end of the interaction. A divergence is a trace of events that leads to divergent behaviour. A distinguished event $\checkmark \in E$ is used as the final element of a trace to indicate that this is a terminating observation.

For example, consider the process $a \rightarrow c \rightarrow \mathit{skip} \,\Box\, b \rightarrow \mathit{div}$, which initially permits an a or b event, and following a permits a c event. It exhibits the failure $([], \{c\})$ since before any events are performed, the event c is being refused. A second failure is $([a], \{a, b\})$, since after performing an a, only c is enabled, and the other events are refused. A third failure is $([a, c, \checkmark], \{a, b, c\})$, which represents successful termination, after which all events are refused. This process also has a divergence trace $[b]$ since the process diverges after performing event b. If a divergent state is unreachable, then D is empty. Here, we show how to extract F and D from any ITree, and thus processes constructed from the operators of §5.

In CSP, one likes to show that there are no divergent states, a property called divergence freedom. The following inductive-coinductive definition captures it:

Definition 5.9 (Divergence Freedom).

$$\frac{-}{\checkmark_x \searrow \mathcal{R}} \qquad \frac{P \searrow \mathcal{R}}{\tau P \searrow \mathcal{R}} \qquad \frac{\mathrm{ran}(F) \subseteq \mathcal{R}}{\mathit{Vis}\ F \searrow \mathcal{R}} \qquad \mathit{div\text{-}free} \triangleq \bigcup \{\mathcal{R} \mid \mathcal{R} \subseteq \{P \mid P \searrow \mathcal{R}\}\}$$

Predicate $P \searrow \mathcal{R}$ is defined inductively. It requires that P stabilises to a *Ret* or a *Vis* whose continuations are all contained in $\mathcal{R}$. Then, *div-free* is the largest set consisting of all sets $\mathcal{R} = \{P \mid P \searrow \mathcal{R}\}$ and is coinductively defined. If we can find an $\mathcal{R}$ such that for every $P \in \mathcal{R}$, it follows that $P \searrow \mathcal{R}$, that is $\mathcal{R}$ is closed under stabilisation, then any $P \in \mathcal{R}$ is divergence-free. Essentially, $\mathcal{R}$ needs to enumerate the symbolic post-stable states of an ITree; for example, $\mathcal{R} = \{\mathit{run}\ E\}$ satisfies the provisos and so $\mathit{run}\ E$ is divergence-free. We have proved that $P \in \mathit{div\text{-}free} \Leftrightarrow (\nexists s \bullet P \xrightarrow{s} \mathit{div})$, which gives the operational meaning.

With our transition relation, we can define Roscoe's step relation, which links the operational and denotational semantics of CSP [56, Section 9.5]. The utility of this definition and the following theorems is to permit symbolic verification of CSP processes by calculating their set-based characterisation.

Definition 5.10 (Roscoe's Step Relation).

$$(P \overset{s}{\Longrightarrow} P') \triangleq ((\exists\, t \in \Sigma\ \mathit{list} \bullet s = t \mathbin{@} [\checkmark_x] \wedge P \xrightarrow{t} \checkmark_x \wedge P' = \mathit{stop}) \vee (\mathit{set}(s) \subseteq \Sigma \wedge P \xrightarrow{s} P'))$$

Here, $\mathit{set}(s)$ extracts the set of elements from a list. The step relation is similar to $\xrightarrow{s}$, except that the event type is adjoined with a special termination event $\checkmark$. We define the enlarged set $\Sigma^{\checkmark} \triangleq \Sigma \cup \{\checkmark_x \mid x \in \mathcal{S}\}$, which adds a family of events parametrised by return values, as in the semantics of Occam [54], which derives from CSP. A termination is signalled when the transition relation reaches a $\mathit{Ret}\ x$ in the ITree, where the trace is augmented with $\checkmark_x$ and the successor state is set to *stop*. We often use a condition of the form $\mathit{set}(s) \subseteq \Sigma$ to mean that no $\checkmark_x$ event is in s. We can now define the sets of traces, failures, and divergences [56]:

Definition 5.11 (Traces, Failures, and Divergences).

$$\mathit{traces}(P) \triangleq \{s \mid \mathit{set}(s) \subseteq \Sigma^{\checkmark} \wedge (\exists\, P' \bullet P \overset{s}{\Longrightarrow} P')\}$$

$$P \mathit{\ ref\ } E \triangleq ((\exists\, F \bullet P = \mathit{Vis}\ F \wedge E \cap \mathrm{dom}(F) = \emptyset) \vee (\exists\, x \bullet P = \mathit{Ret}\ x \wedge \checkmark_x \notin E))$$

$$\mathit{failures}(P) \triangleq \left\{(s, X) \mid \mathit{set}(s) \subseteq \Sigma^{\checkmark} \wedge (\exists\, Q \bullet P \overset{s}{\Longrightarrow} Q \wedge Q \mathit{\ ref\ } X)\right\}$$

$$\mathit{divergences}(P) \triangleq \{s \mathbin{@} t \mid \mathit{set}(s) \subseteq \Sigma \wedge \mathit{set}(t) \subseteq \Sigma \wedge (\exists\, Q \bullet P \overset{s}{\Longrightarrow} Q \wedge Q{\Uparrow})\}$$

The set $\mathit{traces}(P)$ is the set of all possible event sequences that P can perform. For $\mathit{failures}(P)$, we need to determine the set of events that an ITree is refusing, $P \mathit{\ ref\ } E$. If P is a visible event, $\mathit{Vis}\ F$, then any set of events E outside of $\mathrm{dom}(F)$ is refused. If P is a return event, $\mathit{Ret}\ x$, then every event other than $\checkmark_x$ is refused. With this, we can implement Roscoe's form for the failures. Finally, the divergences is simply a trace s leading to a divergent state $Q{\Uparrow}$, followed by any trace t. We exemplify these definitions with two calculations of failures:

$$\mathit{failures}(\mathit{inp}\ c\ A) = \begin{array}{l}\{([], E) \mid \forall\, x \in A \bullet c.x \notin E\} \cup \{([c.x], E) \mid x \in A \wedge \checkmark \notin E\} \\ \cup \{([c.x, \checkmark_{()}], E) \mid x \in A\}\end{array}$$

$$\mathit{failures}(P \gg\!= Q) = \begin{array}{l}\{(s, X) \mid \mathit{set}(s) \subseteq \Sigma \wedge (s, X \cup \{\checkmark_x \mid x \in \mathcal{S}\}) \in \mathit{failures}(P)\} \\ \cup \{(s \mathbin{@} t, X) \mid \exists\, v \bullet s \mathbin{@} [\checkmark_v] \in \mathit{traces}(P) \wedge (t, X) \in \mathit{failures}(Q(v))\}\end{array}$$

The failures of $\mathit{inp}\ c\ A$ consist of (1) the empty trace, where no valid input on c is refused; (2) the trace where an input event $c.x$ occurred, and $\checkmark_{()}$ is not being refused; and (3) the trace where both $c.x$ and $\checkmark_{()}$ occurred, and every event is refused. The failures of $P \gg\!= Q$ consist of (1) the failures of P that do not reach a return, and (2) the terminating traces of P, ending in $\checkmark_v$ appended with a failure of $Q(v)$, the continuation. With the help of Isabelle's simplifier, these equations can be used to calculate the failures and divergences automatically, which can be easier to reason with than directly applying coinduction.

We conclude this section with some important properties of our semantic model:

Theorem 5.12 (Semantic Model Properties).

$(s, X) \in \mathit{failures}(P) \wedge (Y \cap \{x \mid s \mathbin{@} [x] \in \mathit{traces}(P)\} = \emptyset) \longrightarrow (s, X \cup Y) \in \mathit{failures}(P)$

$s \in \mathit{divergences}(P) \wedge \mathit{set}(t) \subseteq \Sigma \longrightarrow s \mathbin{@} t \in \mathit{divergences}(P)$

$P \approx Q \longrightarrow (\mathit{failures}(P) = \mathit{failures}(Q) \wedge \mathit{divergences}(P) = \mathit{divergences}(Q))$

$P \in \mathit{div\text{-}free} \Leftrightarrow \mathit{divergences}(P) = \emptyset$

$P \in \mathit{div\text{-}free} \longrightarrow (\forall\, s\ a \bullet s \mathbin{@} [a] \in \mathit{traces}(P) \longrightarrow (s, \{a\}) \notin \mathit{failures}(P))$

The first two are standard healthiness conditions of the failures-divergences model [56], called ***F3*** and ***D1***, respectively. ***F3*** states that if (s, X) is a failure of P then any event that cannot subsequently occur after s, according to the *traces*, must also be refused. ***D1*** states that the set of divergences is extension closed. We have also proved that two weakly bisimilar processes have the same divergences and failures. The following result links the coinductive definition of divergence freedom and the set of divergences. The final result demonstrates that ITrees satisfy Roscoe's definition of determinism for CSP [56]. If an ITree P is divergence-free, there is no trace after which an event can be accepted and refused.

Finally, we have stronger results relating weak bisimulation with the trace and divergence semantics.

Theorem 5.13. $P \approx Q \Leftrightarrow (\mathit{traces}(P) = \mathit{traces}(Q) \wedge \mathit{divergences}(P) = \mathit{divergences}(Q))$

```
animate buffer
```

```
System
Starting ITree Animation...
Events: (1) State []; (2) Input 0; (3) Input 1; (4) Input 2; (5) Input 3;
5
Internal Activity...
Events: (1) State [3]; (2) Output 3; (3) Input 0; (4) Input 1; (5) Input 2; (6) Input 3;
4
Internal Activity...
Events: (1) State [3,1]; (2) Output 3; (3) Input 0; (4) Input 1; (5) Input 2; (6) Input 3;
7
Rejected
Events: (1) State [3,1]; (2) Output 3; (3) Input 0; (4) Input 1; (5) Input 2; (6) Input 3;
2
Internal Activity...
Events: (1) State [1]; (2) Output 1; (3) Input 0; (4) Input 1; (5) Input 2; (6) Input 3;
2
Internal Activity...
Events: (1) State []; (2) Input 0; (3) Input 1; (4) Input 2; (5) Input 3;
```

Fig. 2. Animating the CSP buffer

We can prove a weak bisimulation between P and Q by showing that these processes have the same traces and divergences. We do not need to consider the refusals because this level has no nondeterminism. Alternatively, we could consider nondeterminism similarly to that shown in §4.3 by introducing a distinguished event that the semantic model abstracts. In this case, the refusal information is vital, and this particular result would no longer hold.

6 ANIMATION BY CODE GENERATION

This section shows how ITrees can be animated by code generation and develops a command called **animate**. This command can be used to execute and probe the behaviour of an ITree-based model. In contrast to the **execute** command of §4, it is interactive and requires that the user selects a visible event in order to proceed.

The Isabelle code generator [35, 36] can be used to extract code from (co)datatypes, functions, and other constructs to functional languages like SML, Haskell, and Scala. Although ITrees can be infinite, this is not a problem for languages with lazy evaluation so that we can step through the behaviour of an ITree. Code generation then allows us to support the generation of verified animators and provides a potential route to correct implementations.

The main complexity is a computable representation of partial functions. Whilst $A \nrightarrow B$ is partly computable, we can only apply it to a value and see whether it yields an output. For animations and implementations, however, we typically want to determine a menu of enabled events for the user to select. Moreover, calculating semantics for CSP operators like $\Box$ and $\|$ requires us to compute with partial functions. For this, we need a way of calculating values for functions dom, $\lhd$, and $\oplus$, which is impossible for arbitrary partial functions. Instead, we need a concrete implementation and a data refinement [35]. We choose associative lists as an implementation, $A \nrightarrow B \approx (A \times B)\ \mathit{list}$, which limits us to finite constructions. However, it has the benefit of being easily printed, making the animator easier to implement. ITrees then have the following representation in Haskell:

```
data Pfun a b = Pfun_alist [(a, b)];
data Itree a b = Ret b | Sil(Itree a b) | Vis(Pfun a (Itree a b))
```

Each of the semantic definitions detailed in sections 4 and 5, including corecursive functions, automatically map to Haskell functions operating over this structure. For constructs like *inp* (Definition 5.2), there is more work to support code generation since these can potentially produce an infinite number of events which an associative list cannot capture. Consider, for example, $inp\, c\, \{0..\}$, for $c : \mathbb{N} \xrightarrow{\Delta} E$, which can produce any event $c.i$ for $i \geq 0$. We can code generate this by limiting the value set to be finite, for example, $\{0..3\}$. Then, the code generator maps this to a list $[0, 1, 2, 3]$, which is computable.

The code for the animator steps through τs until it reaches either a $\checkmark_x$, in which case we terminate, or a *Vis*, in which case the user can choose an option. Since divergence is a possibility, we limit the number of τs that will be skipped. The user can continue or abort the animation after $n = 20$ τ steps. If an empty event choice is encountered, the animation terminates due to deadlock. Otherwise, it displays a menu of events, allows the user to choose one, and recurses following the given continuation.

We only need to augment the generated code for a particular ITree with the animator code to generate an animator. We develop a command **animate**, which inputs a defined ITree and performs an animation. The command (1) runs the code generator, (2) adds the animator code, (3) compiles the code using the Glasgow Haskell Compiler (GHC), and (4) finally runs the binary on a console. This required us to modify Isabelle to add functionality in the PIDE editor interface to start the animation. Technically, this is provided by a new "active area"[5], which is a clickable part of the Output tab in the interface. When the user places their cursor over the **animate** command in the editor, a "Start animation" link is shown, which the user can click to start the animation using the jEdit command-line console.

Fig. 2 shows an animation of the CSP buffer in §5, with the possible inputs limited to $\{0..3\}$. We provide an empty list as a parameter for the initial state. The animator tells us the events enabled and allows us to pick one. If we try to pick a value that is not enabled, the animator rejects this. Since lenses and expressions can also be code generated, we can also animate the *Circus* version of the buffer with the same output.

As a more sophisticated example, we have implemented a distributed ring buffer adapted from the original *Circus* paper [64]. The idea is to represent a buffer as a ring of one-place cells and a controller that manages the ring.

It has the following form:

$$(Controller \;[\![\; \{rd.c, wrt.c \mid c \in \mathbb{N}\} \;]\!]\; (|||\, i \in \{0..maxbuf\} \bullet Cell(i))) \setminus \{rd.c, wrt.c \mid c \in \mathbb{N}\}$$

where $rd.c$ and $wrt.c$ are internal channels for the controller to communicate with the ring, hidden in the overall process network. The individual cells do not communicate with each other, hence the use of interleaving $|||$, but the controller communicates with all cells. Channel $rd.c$ is used to read the current value of cell c, and $wrt.c$ is used to write a value. Each cell is a single place buffer with a state variable val and has the following form:

Definition 6.1 (Ring Buffer Cell).

$$Cell(i) \triangleq wrt?v \rightarrow val := v \fatsemi loop\,(wrt?v \rightarrow val := v \;\Box\; rd!val \rightarrow Skip)$$

Initially, the cell is empty, awaiting a write command over channel wrt. Following this, the cell can either overwrite its current value or advertise its current value over channel rd. The cells are arranged through indexed interleaving, and the buffer size is $maxbuf + 1$. The channels *input* and *output* communicate with the overall buffer.

[5]Please see `src/Pure/PIDE/active.ML` in the Isabelle source code for more information.

The controller has four state variables: (1) $sz :: \mathbb{N}$, the current buffer size; (2) $rtop :: \mathbb{N}$ a pointer to the next available cell; (3) $rbot :: \mathbb{N}$ the index of the first value stored; (4) $cache :: \int$ the cached first element of the buffer. The controller is described using the following actions:

Definition 6.2 (Ring Buffer Controller).

$$InputCtrl \triangleq \begin{array}{l} sz < maxbuf \;\&\; input?x \rightarrow \\ \left(\begin{array}{l} sz = 0 \;\&\; sz := 1 \fatsemi cache := x \\ \Box\; sz > 0 \;\&\; wrt.rtop!x \rightarrow sz := sz + 1 \fatsemi rtop := (rtop + 1) \bmod maxring \end{array} \right) \end{array}$$

$$OutputCtrl \triangleq \begin{array}{l} sz > 0 \;\&\; output!cache \rightarrow \\ \left(\begin{array}{ll} sz > 1 \;\&\; rd.rbot?x \rightarrow & sz := sz - 1 \fatsemi cache := x \fatsemi \\ & rbot := (rbot + 1) \bmod maxring \\ \Box\; sz = 1 \;\&\; sz := 0 & \end{array} \right) \end{array}$$

$$Controller \triangleq sz := 0 \fatsemi rtop := 0 \fatsemi rbot := 0 \fatsemi loop\,(InputCtrl \;\Box\; OutputCtrl)$$

InputCtrl represents a controller input. An input can be accepted if the size is less than *maxbuf*. If the buffer is empty ($sz = 0$), the element is placed in the *cache*. Otherwise, it is sent to the next available cell at *rtop*. The index *rtop* is updated using modulo arithmetic to characterise the circular nature of the buffer. *OutputCtrl* represents a controller output. If the buffer is non-empty, then the buffer can output the cached head. Following this, if there is more than one element, the controller retrieves the element at *rbot*, decreases the buffer size, updates that cache, and finally updates the *rbot* index. If the buffer only had one element, there is no buffer head to cache. The overall behaviour of the controller is to start empty, with both *rtop* and *rbot* pointing to index 0, and then to iterate a choice between *InputCtrl* and *OutputCtrl*.

We tested the animator on the ring buffer, using an Apple M3 Pro with 18GB of memory as the platform. We set up the example so that we can vary *maxbuf* to observe the scalability of the animator. On this platform, we can efficiently animate this example for a relatively small ring of 100 cells, with a similar output to Figure 2, which is a very satisfying result.

We were also able to animate with a much larger ring with 850 cells, which requires about 3 seconds to compute the next step. With 5000 cells, the animator takes around 50 seconds to calculate the next transition. The highest number of cells we could reasonably animate is around 2000. However, we have not attempted to optimise the code, and several data types could be replaced with efficient implementations to improve scalability. Thus, this approach to animation and potential implementation is very promising.

7 SYSTEM MODELLING WITH Z-MACHINES

In this section, we apply our ITree and *Circus* library to create a formal modelling language and tool called "Z-Machines", which is in the style of the Z specification language [58, 65], and B method [1]. Z-Machines are a form of abstract machine, similar to B machines [1], that use our Z toolkit as the underlying expression language. Z-Machines act as a case study for our ITrees library by demonstrating its applicability in creating accessible verification tools. We implement several Isabelle commands for creating Z-Machine artefacts and a technique for verifying invariants. We illustrate these commands using a variant of the buffer example (Example 2.1), which is bounded to a specific size and can be considered as a specification for the ring buffer in §6.

A Z-Machines consists of a set of operations that act over a state, formally:

Definition 7.1 (Z-Machine). A Z-Machine consists of (1) a state space type S; (2) a set of state invariants $P_i : S \Rightarrow \mathbb{B}$; (3) a set of operations $Op_j : T_j \Rightarrow ((), S)\mathit{htree}$, each parametrised by T_j; and (4) an initialisation $I : S$.

Operations are used to update the value of state variables deterministically and can optionally take inputs and produce outputs. Each operation in a Z-Machine is given a semantics as a parametric homogeneous Kleisli tree. The parameters are used to encode inputs and outputs for the operation. The operations are composed in an action system [5] to produce the overall Z-Machine ITree, which can be verified and animated.

We consider each command to create the Z-Machine components and their formal semantics. Each command is interpreted as a set of updates on the Isabelle document model, which creates definitions and other formal artefacts.

We use the **zstore** command (§4.1), to create the bounded buffer state space:

Example 7.2 (Bounded Buffer Store).

```
consts MAX_SIZE :: "nat" and VAL :: "int set"

zstore Buffer =
  sz  :: "nat"
  buf :: "int list"
where
  "sz = length buf"
  "sz ≤ MAX_SIZE"
```

This creates two variables: *sz* and *buf*, and links them using an invariant. Variable *sz* represents the size of the buffer, and *buf* is its contents. Using the **consts** command, we also declare two abstract constants. *MAX_SIZE* characterises the maximum size of the buffer. *VAL* is a finite set of integers representing the possible values we can insert into the buffer. These abstract constants can be assigned concrete definitions later for animation and verification. The store also has two invariants, requiring (1) that the buffer size is the same as the length and (2) that the size is no greater than the maximum size.

Operations are defined using the **zoperation** command, which has the following syntax:

Definition 7.3 (Operation Syntax).

$$\mathit{param} ::= \mathit{name} \in \mathit{term}$$

$$\mathit{operation} ::= \textbf{zoperation}\ \mathit{name}\ \textbf{params}\ \mathit{param}^*\ \textbf{pre}\ \mathit{term}\ \textbf{update}\ \mathit{assignment}$$

An operation consists of a name, a set of parameters, a precondition term, and an update. A parameter consists of a name and a term, characterising the set of values from which the parameter is drawn. We use this set to bound the possible inputs to an operation to allow animation, similar to §6. The precondition acts as a guard for the operation, which must be satisfied for the operation to be executed. The update is a sequence of simultaneous assignments to variables in state space.

As part of generating the Z-Machine semantics, each operation is assigned a unique event channel, $Op_j^c : T_j \xrightarrow{\Delta} E$, sharing the same name and as part of a generated channel type E. The semantics of an operation is shown below:

Definition 7.4 (Operation Semantics).

$$\left[\!\!\left[\begin{array}{l} \textbf{zoperation}\ Op_j \\ \quad \textbf{params}\ x_1 \in A_1\ \cdots\ x_n \in A_n \\ \quad \textbf{pre}\ P \\ \quad \textbf{update}\ \sigma \end{array} \right]\!\!\right] = \left(Op_j \triangleq Op_j^c?\vec{x} \in \vec{A} \mid P(\vec{x}) \rightarrow \sigma(\vec{x}) \right)$$

where $\vec{x} = (x_1, \cdots, x_n)$ and $\vec{A} = A_1 \times \cdots \times A_n$

Here, $A \times B$ denotes the Cartesian product of the two sets A and B. An operation accepts parameters that inhabit the corresponding parameter sets ($\vec{A}$) and satisfies the precondition P in the context of the current state. The operation update is executed when such parameters are provided, with the parameters as inputs ($\vec{x}$). Parameter sets are specified as expressions, meaning the acceptable parameters can vary from state to state. When the Z-Machine is in a state that satisfies the precondition P of an operation for a particular valuation of parameters ($\vec{x}$), the event $Op_j^c.\vec{x}$ is enabled. If the current state cannot satisfy the precondition, an operation's behaviour is *Stop*.

Below, we define three operations for the bounded buffer:

Example 7.5 (Buffer Operations).

```
zoperation Input =
  params v∈VAL
  pre "sz < MAX_SIZE"
  update "[ sz' = sz + 1, buf' = buf @ [v] ]"

zoperation Output =
  params v∈VAL
  pre "sz > 0" "v = hd buf"
  update "[ sz' = sz - 1, buf' = tl buf ]"

zoperation Size =
  emit sz
```

The first operation, *Input*, adds a value to the buffer. It has one parameter, v, drawn from *VAL*, to enumerate the possible events. The precondition requires that the size is strictly less than the maximum size. The update increases the size and adds the new value to the buffer. The second operation, *Output*, likewise has a single parameter, but this time represents an output. The precondition requires that the buffer is non-empty and that the output value is at the buffer's head. The update decreases the size and removes the head from the buffer. The third operation, *Size*, allows us to view the current size of the buffer. It uses the keyword **emit**, which is shorthand for an operation of a single output parameter equal to the given expression, in this case *sz*.

Once semantics have been assigned for each of the operations, we can give the overall semantics for the Z-Machine itself:

Definition 7.6 (Z-Machine Syntax and Semantics).

$$zmachine ::= \textbf{zmachine}\ name\ \textbf{init}\ term\ \textbf{operations}\ name^*$$

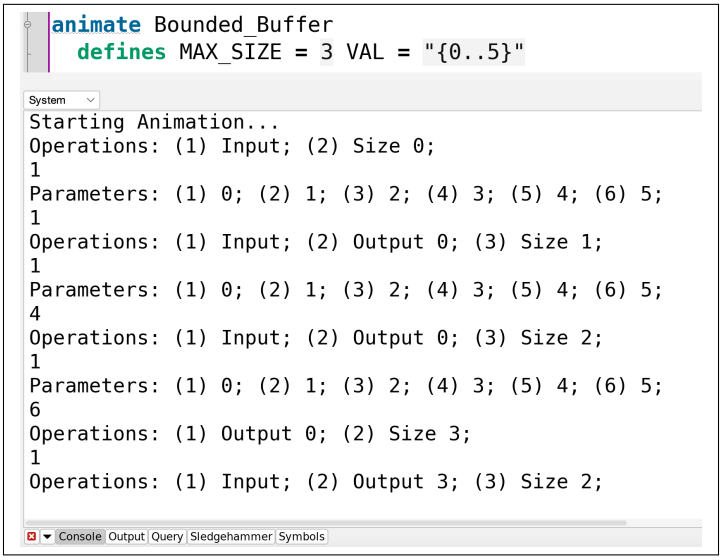

Fig. 3. Animating the Bounded Buffer Z-Machine

$$[\![\textbf{zmachine}\ M \;\; \textbf{init}\ \sigma \;\; \textbf{operations}\ Op_1 \cdots Op_n]\!] = (M \triangleq \langle \sigma \rangle \fatsemi loop(Op_1 \Box Op_2 \Box \cdots \Box Op_n))$$

A Z-Machine consists of an initialisation assignment (σ) and a set of operations. The Z-Machine initialises the state using σ and then enters a loop where the user can choose an enabled operation for execution. Below is the Z-Machine for the bounded buffer:

Example 7.7 (Bounded Buffer Z-Machine).

```
zmachine Bounded_Buffer =
  init "[sz' = 0, buf' = []]"
  operations Input Output Size
```

This Z-Machine initialises *sz* and *buf*, and collects together the operations.

Z-Machines can be animated using the **animate** command developed in §6, provided each operation draws parameters from finite sets or enumerable types. If this is not the case, the code generator will give an error message, and similarly, if any operation uses an undefined abstract constant. An example animation is shown in Figure 3. The animator displays the enabled operations and parameter combinations at each point, and the user can select one. Since the animator needs to enumerate all possibilities, we supply a finite set *VAL* and limit the buffer to size 3.

Verification of Z-Machines involves identifying invariants that characterise specific critical properties. We then need to show that the initialisation establishes the invariants and that each operation preserves them, meaning that the machine satisfies them in any reachable state. Using the weakest preconditions, we can calculate the necessary conditions for an operation to maintain invariants. Specifically, we need to show that $P \rightarrow \mathit{wp}\ Op_j\ P$ for each operation. The weakest precondition calculation results in proof obligations (POs),

which must be discharged to complete the verification. We provide a proof method called `zpog_full` (Z proof obligation generator), which generates the set of POs required for an operation to satisfy the specification. This method implemented by application of the weakest precondition laws using the simplifier (see §4.5), and by Isabelle's automated deduction methods. The POs can typically be discharged with the help of the **sledgehammer** tool. Below, we verify that the bounded buffer Z-Machine satisfies the invariants.

Example 7.8 (Bounded Buffer Invariant Verification).

```
lemma Init_correct: "Init establishes Buffer_inv"
  by zpog_full

lemma Size_correct: "Size(n) preserves Buffer_inv"
  by zpog_full

lemma Input_correct: "Input(v) preserves Buffer_inv"
  by zpog_full

lemma Output_correct: "Output(v) preserves Buffer_inv"
  apply zpog_full
  apply (metis diff_le_self dual_order.trans)
  done
```

Here, C *establishes* P is short-hand for the Hoare triple $\{ \mathit{True} \}\ C\ \{ P \}$, and C *preserves* P is short-hand for $\{ P \}\ C\ \{ P \}$. The proofs proceed by application of the weakest precondition laws by `zpog_full`. For the first three proofs, the simplifier can solve the POs automatically. For the fourth proof obligation, some arithmetic reasoning is required, which we automate with a call to **sledgehammer**. This produces a proof using the resolution prover `metis` and several laws relating to the order on natural numbers.

8 RELATED WORK

Infinite trees are a ubiquitous model for concurrency [61]. In particular, ITrees can be seen as a restricted encoding of Milner's synchronisation trees [46, 47, 63]. In contrast to ITrees, synchronisation trees allow multiple events from each node, including visible and τ events. They have seen several generalisations, most recently by Ferlez et al. [21], who formalise Generalized Synchronisation Trees based on partial orders, define bisimulation relations [22], and apply them to hybrid systems. Our work differs because ITrees use explicit computation and corecursion, but mutual insights will likely be gained.

ITrees [66], and their mechanisation in Coq, have been applied in various projects as a way of defining abstract yet executable semantics [41, 44, 45, 57, 70–72]. They have been used to verify C programs [41] and an HTTP key-value server [44]. Chappe et al. [16] introduce Choice Trees as a conservative extension of ITrees. The main innovation is to add nondeterminism support through constructors *brS* and *brD*, which replace the *Sil* constructor. Whereas *Sil* is deterministic, these two constructors allow a finite number of internal choices. Constructor *brS* represents "stepped" branching, where a τ transition accompanies the resolution of an internal choice. In contrast, *brD* is "delayed" branching, where deadlocked ("stuck" in [16]) branches are eliminated from the choice. The latter choice is similar to the

external choice since deadlocked branches are likewise pruned but more closely resembles angelic nondeterminism [53].

The Coq mechanisation of ITrees uses features unavailable in Isabelle, notably type constructor variables (rank-n polymorphism). Though this is an apparent weakness, the quest to implement ITrees in the more restrictive type system of Isabelle/HOL has entailed several unique advantages. In [66], the *Vis* constructor has two parameters, rather than one, for the output event $e : \mathcal{E}\,\mathcal{A}$ and $k : \mathcal{A} \to \mathit{itree}\ \mathcal{E}\ \mathcal{R}$, a total function, for the continuation. There, $\mathcal{E}$ is a type constructor representing the output sent to the environment, which is parametric over $\mathcal{A}$, the type of answers received back from the environment. In contrast, our work instead (1) fixes a non-parametric event universe E; (2) uses a partial function of type $E \nrightarrow (E, R)$ *itree*; and (3) uses prisms [52] to characterise channels.

Our one-parameter version of *Vis* is, in some respects, more general than the two-parameter one since it allows a variety of communication paradigms and a natural encoding of external choice [13]. In [66], the focus is on a communication scheme where (1) the process sends an output to the environment in $\mathcal{E}\,\mathcal{A}$, and (2) the environment then answers back with a value in $\mathcal{A}$. In CSP, such a scheme can be encoded either with a single event (e.g. $c!y?x \to P(x)$), where the outputs and inputs are present as parameters, or via two separate events for the input and output communication (e.g. $c!y \to d?x \to P(x)$).

Additionally, Coq ITrees [66] only permit choices to be made by the environment when the answer is returned, not in the output event itself. CSP's external choice operator has yet to be encoded in Coq ITrees, and [16] has only nondeterministic (i.e. internal) choice. Such an encoding could be achieved via a special event in $\mathit{choose} : \mathcal{E}\,\mathcal{I}$ to represent that the ITree is asking the environment to resolve a choice, with $\mathcal{I}$ being the possible inputs. Similarly, modelling deadlock could use a special event $\mathit{deadlock} : \mathcal{E}\,\emptyset$ with an empty return type. This disadvantage of such an encoding is that a custom notion of equality is required to reproduce algebraic laws such as $P \,\Box\, \mathit{stop} = P$. We see our natural encoding of external choice as the central contribution of our work. At the same time, this additional generality comes at a cost since the interpretation combinator of the original works [66], which harnesses the output-input pattern to interpret events as monadic actions, requires more effort to encode.

The use of partial functions in our work means that external choice operators can be straightforwardly implemented by the composition of the underlying choice functions (e.g. using $P \odot Q$). This, in turn, means that the algebraic properties of the choice combinator lift to ITrees directly and allow flexible algebraic semantics. One technical exception to the generality of our work is the situation where an empty answer is used ($\mathcal{A} = \emptyset$), which in the Coq work allows a *Vis* that performs an output but has no continuations. Isabelle/HOL has no empty type since all types must exhibit one element and cannot support empty answers. This behaviour requires a slightly different encoding where the output is sent, and the process immediately deadlocks, though this requires two *Vis* operators rather than one. Aside from this situation, we can usually encode the two-parameter version *Vis* $e\ k$ as $[]\ x \in \mathrm{dom}(\mathit{match}_e) \to k(\mathit{match}_e(x))$ with $e : A \xrightarrow{\Delta} E$.

A further benefit of having a fixed E is that ITrees become simpler semantic objects. For example, traces can be represented simply as lists of events rather than the bespoke type used in [66]. These are amenable to first-order automated proof [9], which has allowed us to develop our library quickly and with minimal effort.

Previously, we have demonstrated an Isabelle-based theory library and verification tool for reactive systems [26, 29]. This supports verification and step-wise development of nondeterministic and infinite-state systems based on the CSP [13, 38] and *Circus* [64]

process languages. This includes a specification mechanism called reactive contracts and a calculational proof strategy. Extensions of our theory support reasoning about hybrid dynamical systems, which makes it ideal for verifying autonomous robots.

The Z notation has been implemented in a HOL-based theorem prover several times, notably in ProofPower-Z [4] and HOL-Z [14]. HOL-Z is also implemented in Isabelle/HOL and includes a parser for Z schemas, formal semantics, and proof support. Our implementation is less advanced, does not have the Z schema calculus, and uses types rather than sets to characterise the hierarchy of data structures in Z. This aids proof automation through the type system but at the expense of fidelity to the Z standard. Nevertheless, our implementation of Z-Machines provides proof and animation support, provided that a Z model can be encoded in our restricted subset.

Several tools are available to analyse CSP processes, including the FDR refinement checker [32], and the PAT model checker [59]. These tools offer a more automated approach to analysis than formal proof, though they require the generation of explicitly labelled transition systems, which is hampered by the state explosion problem. FDR has a simple tool, ProBE, for exploring process behaviour by stepping through their definitions. PAT has a more feature-rich CSP simulator that allows users to perform various simulation tasks: (1) Complete finite-state generation based on the execution graph. (2) Automatic random simulation. (3) User interactive simulation with step-by-step execution and trace display and replay. These tools are valuable complements that co-exist in the verification ecosystem. Our focus is on symbolic analysis of processes using proof, though our ITree animator can be applied to search-based analysis, similar to model checking. Moreover, tools like **sledgehammer** greatly increase automation, making verification via proof a realistic possibility.

Recently, the set-based theory of CSP has also been mechanised in the Isabelle-based HOL-CSP tool [18, 60], which is also based on the failures-divergences model. They have used their library to verify the deadlock-freedom of the famous "dining philosophers" example for an arbitrary number of philosophers $N \geq 2$. A further application is modelling and verifying autonomous vehicles, including the continuous dynamics [18]. This work is complementary to our library since HOL-CSP is not limited to deterministic constructions, but on the other hand, HOL-CSP process specifications are not executable. We hope to combine these libraries to realise these mutual benefits.

9 CONCLUSIONS

In this paper, we have demonstrated how Interaction Trees [66] can be used to unify animation and deductive verification of software models, from high-level system models to lower-level program models, in Isabelle/HOL. Our approach harnesses the codatatype package [8] to encode infinite transition systems, and the code generator [35, 36] to provide animation and execution. Our results indicate that the technique provides both tractable verification, with the help of Isabelle's proof automation [9], and efficient execution. Though ITrees are intrinsically deterministic, we have shown how to model nondeterministic behaviour using special events. We applied our technique to simple imperative programs, the CSP and *Circus* process languages [50], and to an abstract machine notation based on Z [58]. We note, however, that it applies to various other process algebraic and modelling languages.

Our work has many practical applications in producing verified simulations, and we have several associated lines of ongoing work. In a parallel paper [69], we have used our ITree library to mechanise semantics for the RoboChart language [48], a formal UML-like language for modelling robots with denotational semantics based on CSP. However, this semantics

does not yet consider the real-time operators, which will require us to consider discrete time, which we believe can be supported using a dedicated time event in ITrees, similar to tock-CSP [55]. This will build on our colleagues.' work with ✓-*tock* [6], a new semantics for tock-CSP.

Separately, we have also used our Z-Machine formalism to give a simplified semantics to RoboChart state machines [68], for the purpose of compositional invariant-based verification, including deadlock checking. We plan to link Isabelle/HOL with the Eclipse-based RoboTool modelling environment to allow seamless verification and feedback for software engineers. This link will open up a pathway from graphical models to verify implementations of autonomous robotic controllers. In concert with this, we will also explore links to our other theories for hybrid systems [24, 49], to allow verification of controllers in the presence of a continuously evolving environment.

The work described in this paper has also been used pedagogically to support two courses on assured software engineering, one for third-year undergraduates and one for external industrial participants. Our courses use our implementation of imperative programs and Hoare logic to teach program verification and Z-Machines to teach Z-based formal specification. The benefit of this approach is that students need only learn a single tool (Isabelle) to support the different pedagogical goals. Moreover, Isabelle's document model has allowed us to create DSLs that support appropriate abstraction levels to minimise the technical detail we expose to students. Our students' feedback has been universally positive, and we plan to report further on this when we have more data.

In the future, we plan to link ITrees to our formalisation of formally reactive contracts [26, 29], which provides both denotational semantics for *Circus* and a refinement calculus for reactive systems, building on our link with failures-divergences. We will also further investigate the failures-divergence semantics of our ITree process operators and determine whether failures-divergences equivalence entails weak bisimulation. We will also continue to expand our imperative program verification tool by considering more advanced concepts, like memory management and associated separation logic. We could also consider the generation of imperative code, using a suitable mechanised semantics for a language target. Finally, we will provide a more user-friendly interface for our simulator, as found in animators like FDR4's probe tool [32] and ProB [43] for Event-B.

REFERENCES

[1] Jean-Raymond Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[2] K. Aehlig, F. Haftmann, and T. Nipkow. A compiled implementation of normalisation by evaluation. *Journal of Functional Programming*, 22(1):9–30, 2012.

[3] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2), 2015.

[4] R. Arthan. On formal specification of a proof tool. In *Formal Software Development Methods*, volume 551 of *LNCS*. Springer, 1991.

[5] R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, June 1989.

[6] J. Baxter, P. Ribeiro, and A. Cavalcanti. Sound reasoning in tock-CSP. *Acta Informatica*, April 2021. doi:10.1007/s00236-020-00394-3.

[7] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with Benefits: Implementing Corecursion in Foundational Proof Assistants. In *Programming Languages and Systems, 26th European Symposium on Programming (ESOP)*, April 2017.

[8] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *5th Intl. Conf. on Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[9] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1), 2016. doi:10.6092/issn.1972-5787/4593.

[10] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof assistant perspective. In *20th Intl. Conf. on Functional Programming (ICFP)*, pages 192–204. ACM, August 2015. doi:10.1145/2858949.2784732.

[11] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58:149–179, 2017. doi:10.1007/s10817-016-9391-3.

[12] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 84–89. Association for Computing Machinery, October 2016. doi:10.1145/2997364.2997384.

[13] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984. doi:10.1145/828.833.

[14] A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2), February 2003.

[15] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.

[16] N. Chappe, P. He, L. Henrio, Y. Zakowski, and S. Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in Coq. In *Proc. ACM Programming Lang. (POPL)*, volume 61. ACM, January 2023.

[17] F. Ciccozzi, I. Malavolta, and B. Selic. Execution of UML models: A systematic review of research and practice. *Software & Systems Modeling*, 18(3):2313–2360, June 2019. doi:10.1007/s10270-018-0675-4.

[18] P. Crisafulli, S. Taha, and B. Wolff. Modeling and analysing cyber-physical systems in HOL-CSP. *Robotics and Autonomous Systems*, 170, 2023.

[19] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[20] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. SEI Series in Software Engineering. Addison-Wesley Professional, 2012.

[21] J. Ferlez, R. Cleaveland, and S. Marcus. Generalized synchronization trees. In *Proc. 17th Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 8412 of *LNCS*, pages 304–319. Springer, 2014. doi:10.1007/978-3-642-54830-7_20.

[22] J. Ferlez, R. Cleaveland, and S. I. Marcus. Bisimulation in behavioral dynamical systems and generalized synchronization trees. In *Proc. 2018 IEEE Conf. on Decision and Control (CDC)*, pages 751–758. IEEE, 2018. doi:10.1109/CDC.2018.8619607.

[23] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

[24] S. Foster. Hybrid relations in Isabelle/UTP. In *7th Intl. Symp. on Unifying Theories of Programming (UTP)*, volume 11885 of *LNCS*, pages 130–153. Springer, 2019.

[25] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, October 2020. doi:10.1016/j.scico.2020.102510.

[26] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Theoretical Computer Science*, 802:105–140, January 2020. doi:10.1016/j.tcs.2019.09.017.

[27] S. Foster, C.-K. Hur, and J. Woodcock. Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL. In *32nd Intl. Conf. on Concurrency Theory (CONCUR)*, volume 203 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[28] S. Foster, Y. Nemouchi, M. Gleirscher, R. Wei, and T. Kelly. Integration of formal proof into unified assurance cases with Isabelle/SACM. *Formal Aspects of Computing*, 2021.

[29] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Automated verification of reactive and concurrent programs by calculation. *Journal of Logical and Algebraic Methods in Programming*, 121, June 2021. doi:10.1016/j.jlamp.2021.100681.

[30] Simon Foster, Yakoub Nemouchi, Colin O'Halloran, Karen Stephenson, and Nick Tudor. Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 11–21, 2020.

[31] Simon Foster, Jonathan Julián Huerta y Munive, Mario Gleirscher, and Georg Struth. Hybrid systems verification with Isabelle/HOL: Simpler syntax, better models, faster proofs. In *FM 2021*, volume 13047 of *LNCS*, pages 367–386, Heidelberg, 2021. Springer. doi:https://doi.org/10.1007/978-3-030-90870-6_20.

[32] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 187–201, 2014.

[33] M. Gleirscher, S. Foster, and J. Woodcock. New opportunities for integrated formal methods. *ACM Comput. Surv.*, 52(6), 2019.

[34] V. B. F Gomes and G. Struth. Modal Kleene algebra applied to program correctness. In *Formal Methods*, volume 9995 of *LNCS*, pages 310–325. Springer, 2016.

[35] F. Haftmann, A. Krauss, O. Kuncar, and T. Nipkow. Data refinement in Isabelle/HOL. In *Proc. 4th Intl. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 100–115. Springer, 2013.

[36] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *10th Intl. Symp. on Functional and Logic Programming (FLOPS)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[37] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

[38] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[39] C. A. R. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.

[40] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[41] Nicolas Koh, Yao Li, Yishuai Li, Li yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proc. 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2019. doi:10.1145/3293880.3294106.

[42] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-69927-9_4.

[43] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Int J Softw Tools Technol Transf*, 10:185–203, 2008. doi:10.1007/s10009-007-0063-9.

[44] Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked applications. In *Proc. 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[45] William Mansky, Wolf Honoré, and Andrew W. Appel. Connecting higher-order separation logic to a first-order outside world. In *Proc. 29th European Symposium on Programming (ESOP)*, 2020.

[46] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[47] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[48] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Software and Systems Modelling*, January 2019. doi:10.1007/s10270-018-00710-z.

[49] J. H. Y. Munive, G. Struth, and S. Foster. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *RAMiCS*, volume 12062 of *LNCS*. Springer, April 2020. doi:10.1007/978-3-030-43520-2_11.

[50] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, 2009. doi:10.1007/s00165-007-0052-5.

[51] R. F. Paige. A meta-method for formal method integration. In *Proc. 4th. Intl. Symp. on Formal Methods Europe (FME)*, volume 1313 of *LNCS*, pages 473–494. Springer, 1997.

[52] M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *The Art, Science, and Engineering of Programming*, 1(2), 2017. doi:10.22152/programming-journal.org/2017/1/7.

[53] P. Ribeiro and A. Cavalcanti. Angelic processes for CSP via the UTP. *Theoretical Computer Science*, 2019.

[54] A. W. Roscoe. Denotational semantics for textsfoccam. In *Intl. Seminar on Concurrency*, volume 197 of *LNCS*, pages 306–329. Springer, 1984.

[55] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 2005.

[56] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[57] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: Termination-sensitive specifications for Interaction Trees. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. doi:10.1145/3434307.

[58] M. Spivey. *The Z Notation — A Reference Manual*. Prentice Hall, Englewood Cliffs, N. J., 1989.

[59] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, 5643:709–714, 2009.

[60] S. Taha, B. Wolff, and L. Ye. Philosophers may dine — definitively! In *Proc. 16th Intl. Conf. on Integrated Formal Methods*, LNCS. Springer, 2020. doi:10.1007/978-3-030-63461-2_23.

[61] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 1997.

[62] Ran Wei, Simon Foster, Haitao Mei, Fang Yan, Ruizhe Yang, Ibrahim Habli, Colin O'Halloran, Nick Tudor, Tim Kelly, and Yakoub Nemouchi. Access: Assurance case centric engineering of safety–critical systems. *Journal of Systems and Software*, 213, 2024. URL: https://www.sciencedirect.com/science/article/pii/S0164121224000773, doi:https://doi.org/10.1016/j.jss.2024.112034.

[63] G. Winsel. Synchronisation trees. *Theoretical Computer Science*, 34(1-2):33–82, 1984.

[64] J. Woodcock and A. Cavalcanti. A concurrent language for refinement. In A. Butterfield, G. Strong, and C. Pahl, editors, *Proc. 5th Irish Workshop on Formal Methods (IWFM)*, Workshops in Computing. BCS, July 2001.

[65] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[66] L.-Y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: Representing recursive and impure programs in Coq. In *Proc. 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2020. doi:10.1145/3371119.

[67] Li-yao Xia. *Executable Denotational Semantics with Interaction Trees*. Ph.D., University of Pennsylvania, United States – Pennsylvania, 2022. ISBN: 9798351434490.

[68] F. Yan, S. Foster, and I. Habli. Automated compositional verification for robotic state machines using Isabelle/HOL. In *Proc. 27th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS)*. IEEE, June 2023. doi:10.1109/ICECCS59891.2023.00029.

[69] K. Ye, S. Foster, and J. Woodcock. Formally verified animation for RoboChart using interaction trees. *Journal of Logical and Algebraic Methods in Programming*, 137, February 2024.

[70] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proc. 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2020. doi:10.1145/3372885.3373813.

[71] Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. Verified translation between purely functional and imperative domain specific languages in HELIX. In *Proc. 12th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, 2020.

[72] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with Interaction Trees and VST. In *Proc. 12th International Conference on Interactive Theorem Proving (ITP)*, 2021. doi:10.4230/LIPIcs.ITP.2021.32.