Formally Verified Simulations of State-Rich Processes Using Interaction Trees in Isabelle/HOL

Simon Foster
University of York, UK

Chung-Kil Hur
Seoul National University, South Korea

Jim Woodcock
University of York, UK

Abstract

Simulation and formal verification are important complementary techniques necessary in high assurance model-based systems development. In order to support coherent results, it is necessary to provide unifying semantics and automation for both activities. In this paper we apply Interaction Trees in Isabelle/HOL to produce a verification and simulation framework for state-rich process languages. We develop the core theory and verification techniques for Interaction Trees, use them to give a semantics to the CSP and *Circus* languages, and formally link our new semantics with the failures-divergences semantic model. We also show how the Isabelle code generator can be used to generate verified executable simulations for reactive and concurrent programs.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Coinduction, Process Algebra, Theorem Proving, Simulation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.20

Related Version *Previous Version*: `https://arxiv.org/abs/2105.05133`

Supplementary Material *Software (Source Code)*: `https://github.com/isabelle-utp/interaction-trees`; archived at `swh:1:dir:fe26a447f5611bbba15fbbc47253dba2075e1cf3`

Funding *Simon Foster*: EPSRC EP/S001190/1 (CyPhyAssure).
Jim Woodcock: EPSRC EP/V026801/1 (TAS Verifiability), EP/M025756/1 (RoboCalc).

Acknowledgements We would like to thank the anonymous reviewers of our paper, whose helpful and insightful comments have improved the content and presentation.

1 Introduction

Simulation is an important technique for prototyping system models, which is widely used in several engineering domains, notably robotics and autonomous systems [9]. For such high assurance systems, it is also necessary that controller software be formally verified, to ensure absence of faults. In order for results from simulation and formal verification to be used coherently, it is important that they are tied together using a unifying formal semantics.

Interaction trees (ITrees) have been introduced by Xia et al. [43] as a semantic technique for reactive and concurrent programming, mechanised in the Coq theorem prover. They are coinductive structures, and therefore can model infinite behaviours supported by a variety of proof techniques. Moreover, ITrees are deterministic and executable structures and so they can provide a route to both verified simulators and implementations.

Previously, we have demonstrated an Isabelle-based theory library and verification tool for reactive systems [15, 16]. This supports verification and step-wise development of nondeterministic and infinite state systems, based on the CSP [8, 21] and *Circus* [42] process languages. This includes a specification mechanism, called reactive contracts, and

calculational proof strategy. Extensions of our theory support reasoning about hybrid dynamical systems, which make it ideal for verifying autonomous robots. Recently, the set-based theory of CSP has also been mechanised [39]. However, such reactive specifications, even if deterministic, are not executable and so there is a semantic gap with implementations.

In this paper, we demonstrate how ITrees can be used as a foundation for verification and simulation of state-rich concurrent systems. For this, we present a novel mechanisation of ITrees in Isabelle/HOL, which requires substantial adaptation from the original work. The benefit is access to Isabelle's powerful proof tools, notably the *sledgehammer* automated theorem prover integration [5], but also the variety of other tools we have created in Isabelle/UTP [14], such as Hoare logic and refinement calculus [1, 30]. Isabelle's code generator allows us to automatically produce ITree-based simulations, which allows a tight development loop, where simulation and verification activities are intertwined. All our results have been mechanised, and can be found in the accompanying repository[1], and clickable icon links next to each specific result, with for Isabelle code and for Haskell code.

The structure of our paper is as follows. In §2 we show how ITrees are mechanised in Isabelle/HOL, including the core operators, and strong and weak bisimulation techniques. In §3 we show how deterministic CSP and *Circus* processes can be semantically embedded into ITrees, including operators like external choice and parallel composition. In §4 we link ITrees with the standard failures-divergences semantic model for CSP, which justifies their integration with other CSP-based techniques. In §5 we show how the code generator can be used to generate simulations. In §6 we briefly consider related work, and in §7 we conclude.

2 Interaction Trees in Isabelle/HOL

Here, we introduce Interaction Trees (ITrees) and develop the main theory in Isabelle/HOL, along with several novel results. ITrees were originally mechanised in Coq by Xia et al. [43]. Our mechanisation in Isabelle/HOL brings unique advantages, including a flexible frontend syntax, an array of automated proof tools, and code generation to several languages.

ITrees are potentially infinite trees whose edges are decorated with events, representing the interactions between a process and its environment. They 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); or (3) a choice between several visible events. In Isabelle/HOL, we encode ITrees using a codatatype [4, 7]:

▶ **Definition 1** (Interaction Tree Codatatype). 

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

Type parameters `'e` and `'r` encode the sorts E and R. Constructor *Ret* represents a return value, and *Sil* an internal event, which evolves to a further ITree. A visible event choice (*Vis*) is represented by a partial function ($A \nrightarrow B$) from events to ITrees, with a potentially infinite domain. This representation is the main deviation from ITrees in Coq [43] (see §6). Here, $A \nrightarrow B$ is isomorphic to `A ⇒ B option`, where `B option` can take the value `None` or `Some x` for `x::B`. We usually specify partial functions using $\lambda\, x \in A \bullet f(x)$, which restricts a function f to the domain A. We write $\{\mapsto\}$ for an empty function, and adopt several operators from the Z notation [38], such as dom, override ($F \oplus G$), and domain restriction ($A \lhd F$). With the associated theorems, we can use Isabelle's simplifier to equationally calculate the domain and other properties of choice partial functions, which provides a high degree of proof automation.

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

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

We call an ITree *unstable* if it has the form τP, and *stable* otherwise. An ITree stabilises, written $P \Downarrow$, if it becomes stable after a finite sequence of τ events, that is $\exists\, n\, P' \bullet P = \tau^n P' \wedge \mathit{stable}(P')$. An ITree that does not stabilise is divergent, written $P \Uparrow \triangleq \neg(P \Downarrow)$.

Using the operators mentioned so far, we can specify only ITrees of finite depth. Infinite ITrees can be specified using primitive corecursion [4], as exemplified below.

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

The **primcorec** command requires that every corecursive call on the right-hand side of an equation is guarded by a constructor. ITree *div* represents the divergent ITree that does not terminate, and only performs internal activity. It is divergent, $\mathit{div} \Uparrow$, since it never stabilises. Moreover, we can show that *div* is the unique fixed-point of τ^{n+1} for any $n \in \mathbb{N}$, $\tau^{n+1} P = P \Leftrightarrow P = \mathit{div}$, and consequently *div* is the only divergent ITree: $P \Uparrow \Rightarrow P = \mathit{div}$.

ITree $\mathit{run}\, E$ can repeatedly perform any $e \in E$ without ceasing. It has the equivalent definition of $\mathit{run}\, E \triangleq []\, e \in E \to \mathit{run}\, E$, and thus the special case $\mathit{run}\, \emptyset = \mathit{stop}$. 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. 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 elide these details in the definitions that follow.

Corecursive definitions can have several equations ordered by priority, like a recursive function. We specify a monadic bind operator for ITrees [43] using such a set of equations.

▶ **Definition 2** (Interaction Tree Bind). *We fix $P, P' : (E, R)\mathit{itree}$, $K : R \Rightarrow (E, S)\mathit{itree}$, $r : R$, and $F : E \nrightarrow (E, S)\mathit{itree}$. Then, $P \gg\!= K$ is defined corecursively by the equations*

$$\checkmark_r \gg\!= K = K\, r \quad \tau P' \gg\!= K = \tau(P' \gg\!= K) \quad \mathit{Vis}\, F \gg\!= K = \mathit{Vis}\,(\lambda\, e \in \mathrm{dom}(F) \bullet F(x) \gg\!= K)$$

The intuition of $P \gg\!= K$ is to execute P, and whenever it terminates ($\checkmark_x$), pass the given value x on to the continuation K. We term K a Kleisli tree [43], or KTree, since it is a Klesli lifting of an ITree. KTrees are of great importance for defining processes that depend on a previous state. For this, we define the type synonym $(E, S)\mathit{htree} \triangleq (S \Rightarrow (E, S)\mathit{itree})$ for a homogeneous KTree. We define the Kleisli composition operator $P \fatsemi Q \triangleq (\lambda\, x.Px \gg\!= Q)$, so symbolised because it is used as sequential composition. Bind satisfies several algebraic laws:

▶ **Theorem 3** (Interaction Tree Bind Laws).

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

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. From the monad laws, we can show that $(\fatsemi, \mathit{Ret})$ also forms a monoid.

The laws of Theorem 3 are proved by coinduction, using the following derivation rule.

▶ **Theorem 4** (ITree Coinduction). *We fix a relation* $\mathcal{R} : (E, R)\mathit{itree} \leftrightarrow (E, R)\mathit{itree}$ *and then given* $(P, Q) \in \mathcal{R}$ *we can deduce* $P = Q$ *provided that the following conditions of* $\mathcal{R}$ *hold:*

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

To show $P = Q$, we need to construct a (strong) bisimulation $\mathcal{R}$ 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, where *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 $x = y$. The third proviso states that internal events must yield bisimilar continuations: $(\tau P, \tau Q) \in \mathcal{R} \Rightarrow (P, Q) \in \mathcal{R}$. 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. The majority of our ITree proofs in Isabelle apply this law, and then use a mixture of equational simplification and automated reasoning with *sledgehammer* to discharge the resulting provisos.

Next, we define an operator for iterating ITrees:

```
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 [6, 3] instead of **primcorec**. This requires us to show that $\gg=$ is a "friendly" corecursive function [3]: 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 special cases $\mathit{loop}\, F \triangleq \mathit{while}\, (\lambda\, s \bullet \mathit{True})\, F$ and $\mathit{iter}\, P \triangleq \mathit{loop}\, (\lambda\, s \bullet P)\, ()$, 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.

Though strong bisimulation is a useful equivalence, we often wish to abstract over τs. We therefore also introduce weak bisimulation, $P \approx Q$, as a coinductive-inductive predicate. 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 CSP and Circus

Here, we give an ITree semantics to deterministic fragments of the CSP [8, 21] and *Circus* [42, 32] languages. Our deterministic CSP fragment is consistent with the one identified by Roscoe [36, Section 10.5]. The standard CSP denotational semantics is provided by the failures-divergences model [8, 36], and we provide preliminary results on linking to this in §4.

3.1 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 E is typically infinite, it is usually expressed in terms of a finite set of channels, which can carry data of various types. Here, we characterise channels abstractly using prisms [33], a concept well known in the functional programming world:

▶ **Definition 5** (Prisms). *A prism is a quadruple* $(\mathcal{V}, \Sigma, \mathit{match}, \mathit{build})$ *where* $\mathcal{V}$ *and* Σ *are non-empty sets. Functions* $\mathit{match} : \Sigma \nrightarrow \mathcal{V}$ *and* $\mathit{build} : \mathcal{V} \Rightarrow \Sigma$ *satisfy the following laws:*

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

We write $X : V \xrightarrow{\triangle} E$ *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.

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

▶ **Definition 6** (Basic CSP Constructs).

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

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

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 ($e \in \mathit{build}_c(\!|\ A\ |\!)$), and it returns the value received for use by a continuation. It corresponds to the `trigger` construct in [43]. An output event $(\mathit{outp}\, c\, v)$ permits a single event, v on channel c, and returns a null value of type $()$. We also define the special case $\mathit{sync}\, e \triangleq \mathit{outp}\, e\, ()$ for a basic event $e :: () \xrightarrow{\triangle} E$. A *guard* b behaves as *skip* if $b = \mathit{true}$ and otherwise deadlocks. It corresponds to the guard in CSP, which can be defined as $b \mathbin{\&} P \triangleq (\mathit{guard}\, b \gg\!= (\lambda\, x \bullet P))$.

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 $c : \mathbb{N} \xrightarrow{\triangle} E$, 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 $(a \to P) \,\Box\, (a \to Q)$ introduces an internal choice, since the a event can lead to P or Q, and is equal to $a \to (P \sqcap Q)$. 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 §7. As for $\gg\!=$, we define external choice corecursively using a set of ordered equations.

▶ **Definition 7** (External choice). $P \,\Box\, Q$, *is defined by the following set of equations:*

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

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 $(a \to P) \,\Box\, (a \to Q) = \mathit{stop}$, 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 [20]. 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 important properties:

▶ **Theorem 8** (External Choice Properties).

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

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

External choice is commutative and has *stop* as a unit. It has *div* as an annihilator, because the τ events means that no other activity is chosen. A finite number of τ events on either 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 4), followed by several invocations of *sledgehammer* to discharge the resulting provisos.

Using the operators defined so far, we can implement a simple buffer process:

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

definition buffer :: "integer list ⇒ (Chan, integer 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 with three branches inside. The variable `s::integer list` denotes the state. The first branch allows a value to be received over `Input`, and then returns `s` with the new value added, and then iterates. The second branch is only active when the buffer is not empty. It outputs the head on `Output`, and then returns the tail. The final branch simply outputs the current state. In §5 we will see how such an example can be simulated.

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. 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 it is not in E, and also not in $\mathrm{dom}(G)$. 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 both choice functions and the set E. We use this operator to define generalised parallel composition. For the sake of presentation, we present partial functions as sets.

▶ **Definition 9.** *$P \parallel_E Q$ is defined corecursively by the following equations:*

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

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

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

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

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

The most complex case is for *Vis*, which constructs a new choice function by merging F and G. The three cases are again represented by three partial functions. 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 side 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: $P \,[\![E]\!]\, Q \triangleq (P \parallel_E Q) \gg\!= (\lambda(x, y) \bullet \mathit{Ret}\,())$ and $P \,|||\, Q \triangleq P \,[\![\emptyset]\!]\, Q$. 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 that 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, and so this has no effect on the behaviour, just the typing. The interleaving operator $P \,|||\, Q$, where there is no synchronisation, is simply defined as $P \,[\![\emptyset]\!]\, Q$. We prove several algebraic laws:

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

$$P \,[\![E]\!]\, Q = Q \,[\![E]\!]\, P \qquad P \,|||\, Q = Q \,|||\, P \qquad \mathit{skip} \,|||\, P = P$$

Parallel composition is commutative, except that we must swap the outputs, and so $[\![E]\!]$ and $|||$ are commutative as well. Parallel has *div* as an annihilator for similar reasons to $\Box$. For $|||$, *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 10** (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. In spite of the significant restrictions on hiding, it supports the common 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: $(\mathit{iter}\,(\mathit{sync}\,e)) \setminus e = \mathit{div}$.

3.2 Circus

Whilst CSP processes can be parametrised to allow modelling state, 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* [42, 32] is an extension of CSP that allows state variables. Given a state variable `buf::integer list`, the buffer example can be expressed in *Circus* as follows:

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

We update the state with assignments, which are threaded through sequential composition.

In our work [15, 14, 16], each state variable is modelled as a lens [12], $x :: \mathcal{V} \Longrightarrow \mathcal{S}$. This 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 [14]. They allow an abstract representation of state spaces, where no explicit model is required to support the laws of programming [22]. Lenses can be designated as independent, $x \bowtie y$, meaning they refer to different regions of $\mathcal{S}$. An expression on state variables is simply a function $e :: \mathcal{S} \Rightarrow \mathcal{V}$, where $\mathcal{V}$ is the return type. 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 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 $\mathcal{S} \Rightarrow \mathcal{S}$.

We can characterise *Circus* through a Kleisli lifting of CSP processes that return values, so that *Circus* actions are simply homogeneous KTrees. We define the core operators below:

▶ **Definition 11** (Circus Operators).

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

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 $\langle [x \rightsquigarrow e] \rangle$. Most of the remaining operators are defined by 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 $\|_E$, and composes this with a merge function. The merge function constructs a state that is composed of the ns_1 region from the final state of P, the ns_2 region from Q, and the remainder coming 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.

Our *Circus* operators satisfy many standard laws [32, 16], beyond the CSP laws:

$$\begin{aligned}
\langle\sigma\rangle \fatsemi \langle\rho\rangle &= \langle\rho \circ \sigma\rangle \\
\langle\sigma\rangle \fatsemi (P \Box Q) &= (\langle\sigma\rangle \fatsemi P) \Box (\langle\sigma\rangle \fatsemi Q) \\
x := e \fatsemi y := f &= y := f \fatsemi x := e && \text{if } x \bowtie y, x \sharp f, y \sharp e \\
P \,[\![ns_1 | E | ns_2]\!]\, Q &= Q \,[\![ns_2 | E | ns_1]\!]\, P && \text{if } ns_1 \bowtie ns_2
\end{aligned}$$

Sequential composition of two state updates σ and ρ entails their functional composition. State updates distribute through external choice from the left. Two variable assignments commute provided their variables are independent ($x \bowtie y$) and their respective expressions do not depend on the adjacent variable. *Circus* parallel composition is commutative, provided that we also switch the name sets.

4 Linking to Failures-Divergences Semantics

Next, we show how ITrees are related to the standard failures-divergences semantics of CSP [8]. 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 existing the mechanisations of the CSP set-based and relational semantics [39, 16] 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 after performing event b, the process diverges. 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 §3.

We begin by giving a big-step operational semantics to ITrees, using an inductive predicate.

▶ **Definition 12** (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 $tr : E\ \mathit{list}$ 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 at the same state. 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 [43], since $P \xrightarrow{tr} P'$ records both the trace and the continuation ITree. It is therefore more general, and provides the foundation for characterising both structural operational and denotational semantics. With these laws, we can prove the usual operational laws for sequential composition as theorems:

▶ **Theorem 13** (Sequential Operational Semantics).

$$\frac{-}{\mathit{skip} \to \checkmark_{()}} \qquad \frac{P \xrightarrow{tr} P'}{(P \gg\!= Q) \xrightarrow{tr} (P' \gg\!= Q)} \qquad \frac{P \xrightarrow{tr_1} \checkmark_x \quad Q(x) \xrightarrow{tr_2} Q'}{(P \gg\!= Q) \xrightarrow{tr_1 \mathbin{@} tr_2} Q'}$$

The *skip* process immediately terminates, returning (). If the left-hand side P of $\gg\!=$ can evolve to P' performing the events in tr, then 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, $tr_1 \mathbin{@} tr_2$, to Q'.

Often in CSP, one likes to show that there are no divergent states, a property called divergence freedom. It is captured by the following inductive-coinductive definition:

▶ **Definition 14** (Divergence Freedom).

$$\frac{-}{\checkmark_x \not\searrow \mathcal{R}} \qquad \frac{P \not\searrow \mathcal{R}}{\tau P \not\searrow \mathcal{R}} \qquad \frac{\mathrm{ran}(F) \subseteq \mathcal{R}}{\mathit{Vis}\, F \not\searrow \mathcal{R}} \qquad \mathit{div\text{-}free} \triangleq \bigcup \{\mathcal{R} \mid \mathcal{R} \subseteq \{P \mid P \not\searrow \mathcal{R}\}\}$$

Predicate $P \not\searrow \mathcal{R}$ is defined inductively. It requires that P stabilises to a *Ret*, or to a *Vis* whose coninuations are all contained in $\mathcal{R}$. Then, *div-free* is the largest set consisting of all sets $\mathcal{R} = \{P \mid P \not\searrow \mathcal{R}\}$, and is coinductively defined. If we can find an $\mathcal{R}$ such that for every $P \in \mathcal{R}$, it follows that $P \not\searrow \mathcal{R}$, 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 is used to link the operational and denotational semantics of CSP [36, Section 9.5]. The utility of this definition, and the theorems that follow, is to permit symbolic verification of CSP processes by calculating their set-based characterisation.

▶ **Definition 15** (Roscoe's Step Relation).

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

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 [34], which derives from CSP. A termination is signalled when the transition relation reaches a $\mathit{Ret}\, x$ in the ITree, in which case 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 [36]:

▶ **Definition 16** (Traces, Failures, and Divergences). 

$$\begin{aligned}
\mathit{traces}(P) &\triangleq \{s \mid \mathit{set}(s) \subseteq \Sigma^{\checkmark} \land (\exists\, P' \bullet P \overset{s}{\Rightarrow} P')\} \\
P\ \mathit{ref}\ E &\triangleq ((\exists\, F \bullet P = \mathit{Vis}\, F \land E \cap \mathrm{dom}(F) = \emptyset) \lor (\exists\, x \bullet P = \mathit{Ret}\, x \land \checkmark_x \notin E)) \\
\mathit{failures}(P) &\triangleq \left\{(s, X) \mid \mathit{set}(s) \subseteq \Sigma^{\checkmark} \land (\exists\, Q \bullet P \overset{s}{\Rightarrow} Q \land Q\ \mathit{ref}\ X)\right\} \\
\mathit{divergences}(P) &\triangleq \{s \mathbin{@} t \mid \mathit{set}(s) \subseteq \Sigma \land \mathit{set}(t) \subseteq \Sigma \land (\exists\, Q \bullet P \overset{s}{\Rightarrow} Q \land Q{\Uparrow})\}
\end{aligned}$$

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:

$$\begin{aligned}
\mathit{failures}(\mathit{inp}\, c\, A) = {} & \{([], E) \mid \forall\, x \in A \bullet c.x \notin E\} \cup \{([c.x], E) \mid x \in A \land \checkmark \notin E\} \\
& \cup \{([c.x, \checkmark_{()}], E) \mid x \in A\} \\
\mathit{failures}(P \gg\!= Q) = {} & \{(s, X) \mid \mathit{set}(s) \subseteq \Sigma \land (s, X \cup \{\checkmark_x \mid x \in \mathcal{S}\}) \in \mathit{failures}(P)\} \\
& \cup \{(s \mathbin{@} t, X) \mid \exists\, v \bullet s \mathbin{@} [\checkmark_v] \in \mathit{traces}(P) \land (t, X) \in \mathit{failures}(Q(v))\}
\end{aligned}$$

The failures of $\mathit{inp}\, c\, A$ consists 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 automatically calculate the failures and divergences, which can be easier to reason with than directly applying coinduction.

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

▶ **Theorem 17** (Semantic Model Properties). 

$$\begin{aligned}
&(s, X) \in \mathit{failures}(P) \land (Y \cap \{x \mid s \mathbin{@} [x] \in \mathit{traces}(P)\} = \emptyset) \Rightarrow (s, X \cup Y) \in \mathit{failures}(P) \\
&s \in \mathit{divergences}(P) \land \mathit{set}(t) \subseteq \Sigma \Rightarrow s \mathbin{@} t \in \mathit{divergences}(P) \\
&P \approx Q \Rightarrow (\mathit{failures}(P) = \mathit{failures}(Q) \land \mathit{divergences}(P) = \mathit{divergences}(Q)) \\
&P \in \mathit{div\text{-}free} \Leftrightarrow \mathit{divergences}(P) = \emptyset \\
&P \in \mathit{div\text{-}free} \Rightarrow (\forall\, s\, a \bullet s \mathbin{@} [a] \in \mathit{traces}(P) \Rightarrow (s, \{a\}) \notin \mathit{failures}(P))
\end{aligned}$$

The first two are standard healthiness conditions of the failures-divergences model [36], 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 set of divergences and failures. The next 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 [36]: if an ITree P is divergence free then there is no trace after which an event can be both accepted and also refused.

5 Simulation by Code Generation

The Isabelle code generator [19, 18] 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, and so we can step through the behaviour of an ITree. Code generation then allows us to support generation of verified simulators, 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, all that we can do is apply it to a value and see whether it yields an output or not. For simulations and implementations, however, we typically want to determine a menu of enabled events for the user to select from. Moreover, calculation of a 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 not possible for arbitrary partial functions. Instead, we need a concrete implementation and a data refinement [18].

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 pretty printed and so makes the simulator easier to implement. More sophisticated implementations are possible, as the core theory of ITrees is separated from the code generation setup. To allow us to represent partial functions by associative lists, we need to define a mapping function:

```
fun pfun_alist :: "('a × 'b) list ⇒ ('a ↛ 'b)" where
"pfun_alist [] = {↦}" | "pfun_alist ((k,v) # f) = pfun_alist f ⊕ {k ↦ v}"
```

This recursive function converts an associative list to a partial function, by adding each pair in the list as a maplet. We generally assume that associative lists preserve distinctness of keys. However, for this function, keys which occur earlier take priority. With this function we can then demonstrate how the different partial function operators can be computed. We prove the following congruence equations as theorems in Isabelle/HOL.

$$
\begin{aligned}
(\mathit{pfun\_alist}\, f) \oplus (\mathit{pfun\_alist}\, g) &= \mathit{pfun\_alist}\,(g \mathbin{@} f) \\
A \lhd (\mathit{pfun\_alist}\, f) &= \mathit{pfun\_alist}\,(\mathit{AList.restrict}\, A\ m) \\
(\lambda\, x \in (\mathit{set}\, xs) \bullet f(x)) &= \mathit{pfun\_alist}\,(\mathit{map}\,(\lambda\, k \bullet (k, f\ k))\ xs)
\end{aligned}
$$

Override ($\oplus$) is expressed by concatenating the associative lists in reverse order. Domain restriction ($\lhd$) has an efficient implementation in Isabelle, *AList.restrict*, which we use. For a partial λ-abstraction, we assume that the domain set is characterised by a list $(\mathit{set}\, xs)$. Then, a λ term can be computed by mapping the body function f over xs.

With these equations, we can set up the code generator. The idea is to designate certain representations of abstract types as code datatypes in the target language, of which each mapping function is a constructor. For sets, the following Haskell code datatype is produced:

```
data Set a = Set [a] | Coset [a] deriving (Read, Show);
```

A set is represented as a list of values using the constructor `Set`, which corresponds to the function *set*. It is often the case that we wish to capture a complement of another set, and so there is also the constructor `Coset` for a set whose elements are all those not in the given list. Functions on sets are then computed by code equations, which provide the implementation for each concrete representation. The membership function *member* is implemented like this:

```
member :: forall a. (Eq a) => a -> Set a -> Bool;
member x (Coset xs) = not (x `elem` xs); member x (Set xs) = xs `elem` x;
```

```
*CSP_Examples> simulate (buffer [])
Internal Activity...
Events: [State_C [],Input_C 0,Input_C 1,Input_C 2,Input_C 3]
Input_C 3
Internal Activity...
Events: [State_C [3],Output_C 3,Input_C 0,Input_C 1,Input_C 2,Input_C 3]
Input_C 1
Internal Activity...
Events: [State_C [3,1],Output_C 3,Input_C 0,Input_C 1,Input_C 2,Input_C 3]
Input_C 4
Rejected
Events: [State_C [3,1],Output_C 3,Input_C 0,Input_C 1,Input_C 2,Input_C 3]
Output_C 3
Internal Activity...
Events: [State_C [1],Output_C 1,Input_C 0,Input_C 1,Input_C 2,Input_C 3]
Output_C 1
Internal Activity...
Events: [State_C [],Input_C 0,Input_C 1,Input_C 2,Input_C 3]
```

Figure 1 Simulating the CSP buffer in the Glasgow Haskell Interpreter.

Each case for the function corresponds to a code equation. The function **elem** is the Haskell prelude function that checks whether a value is in a list. This kind of representation ensures correctness of the generated code with respect to the Isabelle specifications. Similarly to sets, we can code generate the following representation for partial functions:

```
data Pfun a b = Pfun_alist [(a, b)];

dom :: forall a b. Pfun a b -> Set a;
dom (Pfun_alist xs) = Set (map fst xs);
```

A partial function has a single constructor, although it is possible to augment this with additional representations. Each code equation likewise becomes a case for the corresponding recursive function, as illustrated by the domain function. Finally, we can code generate interaction trees, which are represented by a very compact datatype:

```
data Itree a b = Ret b | Sil (Itree a b) | Vis (Pfun a (Itree a b));
```

Each semantic definition, including corecursive functions, are also automatically mapped to Haskell functions. We illustrate the code generated for external choice below:

```
extchoice :: (Eq a, Eq b) => Itree a b -> Itree a b -> Itree a b;
extchoice p q = (case (p, q) of {
    (Ret r, Ret y) -> (if r == y then Ret r else Vis zero_pfun);
    (Ret _, Sil qa) -> Sil (extchoice p qa); (Ret r, Vis _) -> Ret r;
    (Sil pa, _) -> Sil (extchoice pa q); (Vis _, Ret a) -> Ret a;
    (Vis _, Sil qa) -> Sil (extchoice p qa);
    (Vis f, Vis g) -> Vis (map_prod f g); });
```

The `map_prod` function corresponds to $\odot$, and is defined in terms of the corresponding code generated functions for partial functions. The external choice operator ($\Box$) is simply defined as an infinitely recursive function with each of the corresponding cases in Definition 7.

For constructs like *inp* (Definition 6), there is more work to support code generation, since these can potentially produce an infinite number of events which cannot be captured by an associative list. Consider, for example, $\mathit{inp}\, c\, \{0..\}$, for $c : \mathbb{N} \stackrel{\Delta}{\rightarrow} E$, 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. Thus, we can finally export code for concrete examples using the operator implementations.

We can now implement a simple simulator, the code for which is shown below:

```
sim_cnt :: (Eq e, Show e, Read e, Show s) => Int -> Itree e s -> IO ();
sim_cnt n (Ret x) = putStrLn ("Terminated:␣" ++ show x);
sim_cnt n (Sil p) =
  do { if (n == 0) then putStrLn "Internal␣Activity..." else return ();
       if (n >= 20)
       then do { putStr "Many␣steps␣(>␣20);␣Continue?"; q <- getLine;
                 if (q=="Y") then sim_cnt 0 p else putStrLn "Ended."; }
       else sim_cnt (n + 1) p };
sim_cnt n (Vis (Pfun_alist [])) = putStrLn "Deadlocked.";
sim_cnt n t@(Vis (Pfun_alist m)) =
  do { putStrLn ("Events:␣" ++ show (map fst m)); e <- getLine;
       case (reads e) of
         []       -> do { putStrLn "No␣parse"; sim_cnt n t }
         [(v, _)] -> case (lookup v m) of
                       Nothing -> do { putStrLn "Rejected"; sim_cnt n t }
                       Just k -> sim_cnt 0 k };
simulate = sim_cnt 0;
```

The idea is to step through τs until we reach either a $\checkmark_x$, in which case we terminate, or a *Vis*, in which we case the user can choose an option. Since divergence is a possibility, we limit the number of τs that the will be skipped. After 20 τ steps, the user can choose to continue or abort the simulation. If an empty event choice is encountered, then the simulation terminates due to deadlock. Otherwise, it displays a menu of events, allows the user to choose one, and then recurses following the given continuation. The simulator currently depends on associative lists to represent choices, but other implementations are possible.

In order to apply the simulator, we need only augment the generated code for a particular ITree with the simulator code. Figure 1 shows a simulation of the CSP buffer in §3, with the possible inputs limited to $\{0..3\}$. We provide an empty list as a parameter for the initial state. The simulator tells us the events enabled, and allows us to pick one. If we try and pick a value not enabled, the simulator rejects this. Since lenses and expressions can also be code generated, we can also simulate the *Circus* version of the buffer, with the same output.

As a more sophisticated example, we have implemented a distributed ring buffer, which is adopted from the original *Circus* paper [42]. 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:

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

where $rd.c$ and $wrt.c$ are internal channels for the controller to communicate with the ring. Each cell is a single place buffer with a state variable val, and has the form

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

The cells are arranged through indexed interleaving, and *maxbuff* is the buffer size. The channels *Input* and *Output* are used for communicating with the overall buffer. Space will not permit further details. The simulator can efficiently simulate this example, for a small ring with 5 cells, with a similar output to Figure 1, which is a satisfying result.

We were also able to simulate the ring buffer with 100 cells, which requires about 3 seconds to compute the next step. With 1000 cells, the simulator takes more than a minute to calculate the next transition. The highest number of cells we could reasonably simulate is around 250. However, we have made no attempt to optimise the code, and several data types could be replaced with efficient implementations to improve scalability. Thus, as an approach to simulation and potentially implementation, this is very promising.

6 Related Work

Infinite trees are a ubiquitous model for concurrency [40]. In particular, ITrees can be seen as a restricted encoding of Milner's synchronisation trees [27, 41, 28]. In contrast to ITrees, synchronisation trees allow multiple events from each node, including both visible and τ events. They have seen several generalisations, most recently by Ferlez et al. [10], who formalise Generalized Synchronisation Trees based on partial orders, define bisimulation relations [11], and apply them to hybrid systems. Our work is different, because ITrees use explicit coinduction and corecursion, but there are likely mutual insights to be gained.

ITrees naturally support deterministic interactions, which makes them ideal for implementations. Milner extensively discusses determinism in [28, chapter 11], a property which is imposed by construction in our operators. Similarly, Hoare defines a deterministic choice operator $a \to P \mid b \to Q$ in [21, page 29], which is similar to ours except that Hoare's operator imposes determinism syntactically, where we introduce deadlock.

ITrees [43], and their mechanisation in Coq, have been applied in various projects as a way of defining abstract yet executable semantics [23, 44, 26, 45, 46, 25, 37]. They have been used to verify C programs [23] and a HTTP key-value server [25]. The Coq mechanisation uses features not available in Isabelle, notably type constructor variables. Specifically, in [43] the *Vis* constructor has two parameters, rather than one, for the enabled events (i.e. channels) $e : \mathcal{E}\ \mathcal{A}$ and $k : \mathcal{A} \to \mathit{itree}\ \mathcal{E}\ \mathcal{R}$, a total function, for the continuation. There, $\mathcal{E}$ is a type constructor over $\mathcal{A}$, the type of data. Our work avoids this, with no apparent loss of generality, by fixing an event universe, E; using partial functions to represent visible event choices; and using prisms [33] to characterise channels. We can encode the two parameter $\mathit{Vis}\ e\ k$ as $[\!]\ x \in \mathrm{dom}(\mathit{match}_e) \to k(\mathit{match}_e(x))$ with $e : A \xrightarrow{\triangle} E$. The benefit of having a fixed E is that ITrees become much simpler semantic objects. Traces can be represented as lists, rather than the bespoke type used in [43]. These are amenable to first-order automated proof [5], which has allowed us to develop our library quickly and with minimal effort.

7 Conclusions

In this paper we showed how Interaction Trees [43] can be used to develop verified simulations for state-rich process languages with the help of Isabelle codatatypes [4] and the code generator [19, 18]. Our early results indicate that the technique provides both tractable verification, with the help of Isabelle's proof automation [5] and efficient simulation. We applied our technique to the CSP and *Circus* process languages, though it is applicable to a variety of other process algebraic languages.

So far, we have focused primarily on deterministic processes, since these are easier to implement. This is not, however, a limitation of the approach. There are at least three approaches that we will investigate to handling nondeterminism in the future: (1) use of a dedicated indexed nondeterminism event; (2) extension of ITrees to permit a computable set of events following a τ; (3) a further Kleisli lifting of ITrees into sets. Moreover, we will formally link ITrees to our formalisation of reactive contracts [15, 16], which provides both a denotational semantics for *Circus* and a refinement calculus for reactive systems, building on our link with failures-divergences. We will implement the remaining CSP operators, such as renaming and interruption. We will also further investigate the failures-divergence semantics of our ITree process operators, and determine whether failures-divergences equivalence entails weak bisimulation. Finally we will provide a more user friendly interface for our simulator as found in animators like FDR4's probe tool [17] and ProB [24] for Event-B.

Our work has many practical applications in production of verified simulations. We intend to use it to mechanise a semantics for the RoboChart [29] and RoboSim [9] languages, which are formal UML-like languages for modelling robots with denotational semantics based in CSP. This will require us to consider discrete time, which we believe can be supported using a dedicated time event in ITrees, similar to tock-CSP [35]. This will build on our colleagues' work with ✓-*tock* [2], a new semantics for tock-CSP. This will open up a pathway from graphical models to verified implementations of autonomous robotic controllers. In concert with this, we will also explore links to our other theories for hybrid systems [31, 13], to allow verification of controllers in the presence of a continuously evolving environment.

References

1 R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

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

3 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)*, 2017.

4 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.

5 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.

6 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.

7 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.

8 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.

9 A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. Filho, A. Didier, W. Li, and J. Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019. doi:10.1016/j.scico.2019.01.004.

10 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.

11 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.

12 J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

13 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.

14 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.

15 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.

16 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.

17 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.

18 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.

19 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.

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

21 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

22 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.

23 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.

24 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.

25 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.

26 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.

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

28 Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

29 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.

30 C. Morgan. *Programming from Specifications*. Prentice-Hall, January 1996.

31 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.

32 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.

33 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.

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

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

36 A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

37 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.

38 M. Spivey. *The Z-Notation - A Reference Manual*. Prentice Hall, Englewood Cliffs, N. J., 1989.

39 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.

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

41 G. Winskel. Synchronisation trees. *Theoretical Computer Science*, 34(1-2):33–82, 1984.

42 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.

43 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.

44 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.

45 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.

46 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.