


# 1 Formally Verified Simulations of State-Rich 2 Processes using Interaction Trees in Isabelle/HOL

3 Simon Foster ✉ 

4 University of York

5 Chung-Kil Hur ✉

6 Seoul National University

7 Jim Woodcock ✉ 

8 University of York

## 9 — Abstract —

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

18 **2012 ACM Subject Classification** Theory of computation → Concurrency

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

20 **Digital Object Identifier** [10.4230/LIPIcs.CONCUR.2021.17](https://doi.org/10.4230/LIPIcs.CONCUR.2021.17)

21 **Related Version** *Previous Version:* <https://arxiv.org/abs/2105.05133>

22 **Funding** *Simon Foster:* EPSRC EP/S001190/1 (CyPhyAssure)

23 *Jim Woodcock:* EPSRC EP/V026801/1 (TAS Verifiability), EP/M025756/1 (RoboCalc)

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

## 26 **1** Introduction

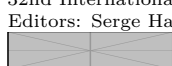
27 Simulation is an important technique for prototyping system models, which is widely used in  
28 several engineering domains, notably robotics and autonomous systems [9]. For such high  
29 assurance systems, it is also necessary that controller software be formally verified, to ensure  
30 absence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

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

37 Previously, we have demonstrated an Isabelle-based theory library and verification  
38 tool for reactive systems [15, 16]. This supports verification and step-wise development  
39 of nondeterministic and infinite state systems, based on the CSP [8, 21] and *Circus* [42]  
40 process languages. This includes a specification mechanism, called reactive contracts, and  
41 calculational proof strategy. Extensions of our theory support reasoning about hybrid  
42 dynamical systems, which make it ideal for verifying autonomous robots. Recently, the





© Simon Foster, Chung-Kil Hur, and Jim Woodcock;  
licensed under Creative Commons License CC-BY 4.0  
32nd International Conference on Concurrency Theory (CONCUR 2021).



Editors: Serge Haddad and Daniele Varacca; Article No. 17; pp. 17:1–17:18  
Leibniz International Proceedings in Informatics  
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

43 set-based theory of CSP has also been mechanised [39]. However, such reactive specifications,  
44 even if deterministic, are not executable and so there is a semantic gap with implementations.


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

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

## 62 2 Interaction Trees in Isabelle/HOL

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

67 ITrees are potentially infinite trees whose edges are decorated with events, representing  
68 the interactions between a process and its environment. They are parametrised over two sorts  
69 (types):  $E$  of events and  $R$  of return values (or states). There are three possible interactions:  
70 (1) termination, returning a value in  $R$ ; (2) an internal event ( $\tau$ ); or (3) a choice between  
71 several visible events. In Isabelle/HOL, we encode ITrees using a codatatype [4, 7]:

72 ► **Definition 1** (Interaction Tree Codatatype). 


```
73 codatatype ('e, 'r) itree =  
74   Ret 'r | Sil "( 'e, 'r) itree" | Vis "'e → ('e, 'r) itree"
```

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

<sup>1</sup> <https://github.com/isabelle-utp/interaction-trees>

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

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


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

```
96 primcorec div :: "('e, 's) itree" where "div =  $\tau$  div"
97 primcorec run :: "'e set  $\Rightarrow$  ('e, 's) itree" where
98   "run E = Vis (map_pfun ( $\lambda$  x. run E) (pId_on E))"
```

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


104 ITree *run E* can repeatedly perform any  $e \in E$  without ceasing. It has the equivalent  
 105 definition of  $run\ E \triangleq \llbracket e \in E \rightarrow run\ E \rrbracket$ , and thus the special case  $run\ \emptyset = stop$ . The formulation  
 106 above uses the function **map\_pfun** ::  $('b \Rightarrow 'c) \Rightarrow ('a \mapsto 'b) \Rightarrow ('a \mapsto 'c)$  which maps a total  
 107 function over every output of a partial function. Function **pId\_on E** is the identity partial  
 108 function with domain **E**. This formulation is required to satisfy the syntactic guardedness  
 109 requirements. For the sake of readability, we elide these details in the definitions that follow.

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

► **Definition 2** (Interaction Tree Bind). We fix  $P, P' : (E, R)itree$ ,  $K : R \Rightarrow (E, S)itree$ ,  $r : R$ ,  
 and  $F : E \mapsto (E, S)itree$ . Then,  $P \gg K$  is defined corecursively by the equations 

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

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

118 ► **Theorem 3** (Interaction Tree Bind Laws). 

$$\begin{array}{ll} Ret\ x \gg K = K\ x & Ret \mathbin{;} K = K \\ P \gg Ret = P & K \mathbin{;} Ret = K \\ P \gg (\lambda x. (Q\ x \gg R)) = (P \gg Q) \gg R & K_1 \mathbin{;} (K_2 \mathbin{;} K_3) = (K_1 \mathbin{;} K_2) \mathbin{;} K_3 \\ div \gg K = div & run\ E \gg K = run\ E \end{array}$$

120 Bind satisfies the three monad laws: it has *Ret* as left and right units, and is essentially  
 121 associative. Moreover, both *div* and *run* are left annihilators for bind, since they do not  
 122 terminate. From the monad laws, we can show that  $(\mathbin{;}, Ret)$  also forms a monoid.

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

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

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

124 To show  $P = Q$ , we need to construct a (strong) bisimulation  $\mathcal{R}$  and show that  $(P, Q) \in \mathcal{R}$ .  
 125 There are four provisos to show that  $\mathcal{R}$  is a bisimulation. The first requires that only ITrees  
 126 of the same kind are related, where *is\_Ret*, *is\_Sil*, and *is\_Vis* distinguish the three cases.  
 127 The second proviso states that if  $(\checkmark_x, \checkmark_y) \in \mathcal{R}$  then  $x = y$ . The third proviso states that  
 128 internal events must yield bisimilar continuations:  $(\tau P, \tau Q) \in \mathcal{R} \Rightarrow (P, Q) \in \mathcal{R}$ . The final  
 129 proviso states that for two visible interactions the two functions must have the same domain  
 130 ( $\text{dom}(F) = \text{dom}(G)$ ) and every event  $e \in \text{dom}(F)$  must lead to bisimilar continuations. The  
 131 majority of our ITree proofs in Isabelle apply this law, and then use a mixture of equational  
 132 simplification and automated reasoning with *sledgehammer* to discharge the resulting provisos.

133 Next, we define an operator for iterating ITrees: 🍷

134 **corec** while :: "(*s*  $\Rightarrow$  bool)  $\Rightarrow$  (*e*, *s*) htree  $\Rightarrow$  (*e*, *s*) htree" **where**  
 135 "while b P s = (if (b s) then Sil (P s  $\gg$  while b P) else Ret s)"

136 This is not primitively corecursive, since the corecursive call uses  $\gg$ , and so we define it  
 137 using the **corec** command [6, 3] instead of **primcorec**. This requires us to show that  $\gg$  is a  
 138 “friendly” corecursive function [3]: it consumes at most one input constructor to produce one  
 139 output constructor. A while loop iterates whilst the condition *b* is satisfied by state *s*. In this  
 140 case, a  $\tau$  event is followed by the loop body and the corecursive call. If the condition is false,  
 141 the current state is returned. We introduce the special cases *loop F*  $\triangleq$  *while* ( $\lambda s \bullet \text{True}$ ) *F* and  
 142 *iter P*  $\triangleq$  *loop* ( $\lambda s \bullet P$ ) (), which represent infinite loops with and without state, respectively.  
 143 We can show that *iter* ( $\checkmark_{\perp}$ ) = *div*, since it never terminates and has no visible behaviour.

144 Though strong bisimulation is a useful equivalence, we often wish to abstract over  $\tau$ s.  
 145 We therefore also introduce weak bisimulation,  $P \approx Q$ , as a coinductive-inductive predicate.  
 146 It requires us to construct a relation  $\mathcal{R}$  such that whenever  $(P, Q) \in \mathcal{R}$  both stabilise, all  
 147 their visible event continuations are also related by  $\mathcal{R}$ . For example,  $\tau^m P \approx \tau^n Q$  whenever  
 148  $P \approx Q$ . We have proved that  $\approx$  is an equivalence relation, and  $P \approx \text{div} \Rightarrow P = \text{div}$ . 🍷

### 149 3 CSP and Circus

150 Here, we give an ITree semantics to deterministic fragments of the CSP [8, 21] and *Circus* [42,  
 151 32] languages. Our deterministic CSP fragment is consistent with the one identified by  
 152 Roscoe [36, Section 10.5]. The standard CSP denotational semantics is provided by the  
 153 failures-divergences model [8, 36], and we provide preliminary results on linking to this in §4.

#### 154 3.1 CSP

155 CSP processes are parametrised by an event alphabet ( $\Sigma$ ), which specifies the possible ways a  
 156 process communicates with its environment. For ITrees,  $\Sigma$  is provided by the type parameter  
 157 *E*. Whilst *E* is typically infinite, it is usually expressed in terms of a finite set of channels,  
 158 which can carry data of various types. Here, we characterise channels abstractly using  
 159 prisms [33], a concept well known in the functional programming world:


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

$$\text{match}(\text{build } x) = x \quad y \in \text{dom}(\text{match}) \Rightarrow \text{build}(\text{match } y) = y$$

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

161 Intuitively, a prism abstractly characterises a datatype constructor,  $E$ , taking a value of  
162 type  $\mathcal{V}$ . Then,  $\text{build}$  is the constructor, and  $\text{match}$  is the destructor, which is partial due to  
163 the possibility of several disjoint constructors. For CSP, each prism models a channel in  $E$   
164 carrying a value of type  $\mathcal{V}$ . We have created a command **chantype**, which automates the  
165 creation of prism-based event alphabets.

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

169 ► **Definition 6** (Basic CSP Constructs). 

$$\begin{aligned} 170 \quad \text{inp} &:: (V \xrightarrow{\Delta} E) \Rightarrow V \text{ set} \Rightarrow (E, V)\text{itree} \\ 171 \quad \text{inp } c \ A &\triangleq \text{Vis}(\lambda e \in \text{dom}(\text{match}_c) \cap \text{build}_c(A)) \bullet \text{Ret}(\text{match}_c \ e) \\ 172 \\ 173 \quad \text{outp} &:: (V \xrightarrow{\Delta} E) \Rightarrow V \Rightarrow (E, ())\text{itree} \quad \text{guard } b :: \mathbb{B} \Rightarrow (E, ())\text{itree} \\ &\text{outp } c \ v \triangleq \text{Vis}\{\text{build}_c \ v \mapsto \text{Ret}()\} \quad \text{guard } b \triangleq (\text{if } b \text{ then skip else stop}) \end{aligned}$$

174 An input event  $(\text{inp } c \ A)$  permits any event over the channel  $c$ , that is  $e \in \text{dom}(\text{match}_c)$ ,  
175 provided that its parameter is in  $A$  ( $e \in \text{build}_c(A)$ ), and it returns the value received for  
176 use by a continuation. It corresponds to the **trigger** construct in [43]. An output event  
177  $(\text{outp } c \ v)$  permits a single event,  $v$  on channel  $c$ , and returns a null value of type  $()$ . We also  
178 define the special case  $\text{sync } e \triangleq \text{outp } e \ ()$  for a basic event  $e :: () \xrightarrow{\Delta} E$ . A  $\text{guard } b$  behaves as  
179  $\text{skip}$  if  $b = \text{true}$  and otherwise deadlocks. It corresponds to the guard in CSP, which can be  
180 defined as  $b \ \& \ P \triangleq (\text{guard } b \ \gg (\lambda x \bullet P))$ .

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

184 Next, we define the external choice operator,  $P \square Q$ , where the environment resolves the  
185 choice with an initial event of  $P$  or  $Q$ . In CSP,  $\square$  can also introduce nondeterminism, for  
186 example  $(a \rightarrow P) \square (a \rightarrow Q)$  introduces an internal choice, since the  $a$  event can lead to  
187  $P$  or  $Q$ , and is equal to  $a \rightarrow (P \square Q)$ . Since we explicitly wish to avoid introducing such  
188 nondeterminism, we make a design choice to exclude this possibility by construction. There  
189 are other possibilities for handling nondeterminism in ITrees, which we consider in §7. As  
190 for  $\gg$ , we define external choice corecursively using a set of ordered equations.

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

$$\begin{aligned} 192 \quad (\text{Vis } F) \square (\text{Vis } G) &= \text{Vis}(F \odot G) \quad (\text{Ret } x) \square (\text{Vis } G) = \text{Ret } x \\ (\text{Sil } P') \square Q &= \text{Sil}(P' \square Q) \quad (\text{Vis } F) \square (\text{Ret } y) = \text{Ret } y \\ P \square (\text{Sil } Q') &= \text{Sil}(P \square Q') \quad (\text{Ret } x) \square (\text{Ret } y) = (\text{if } x = y \text{ then } (\text{Ret } x) \text{ else stop}) \end{aligned}$$


193 where  $F \odot G \triangleq (\text{dom}(G) \triangleleft F) \oplus (\text{dom}(F) \triangleleft G)$

194 An external choice between two functions  $F$  and  $G$  essentially combines all the choices  
195 presented using  $F \odot G$ . The caveat is that if the domains of  $F$  and  $G$  overlap, then any  
196 events in common are excluded. Thus,  $\odot$  restricts the domain of  $F$  to maplets  $e \mapsto P$

## 17:6 Formally Verified Simulations of State-Rich Processes using Interaction Trees


197 where  $e \notin \text{dom}(G)$ , and vice-versa. This has the effect that  $(a \rightarrow P) \square (a \rightarrow Q) = \text{stop}$ , for  
 198 example. In the special case that  $\text{dom}(F) \cap \text{dom}(G) = \emptyset$ ,  $P \odot Q = P \oplus Q$ . We chose this  
 199 behaviour to ensure that  $\square$  is commutative, though we could alternatively bias one side.

200 Internal steps on either side of  $\square$  are greedily consumed. Due to the equation order,  $\tau$   
 201 events have the highest priority, following a maximal progress assumption [20]. Return events  
 202 also have priority over visible events. If two returns are present then they must agree on the  
 203 value, otherwise they deadlock. External choice satisfies several important properties:

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

$$P \square Q = Q \square P \quad \text{stop} \square P = P \quad \text{div} \square P = \text{div} \quad P \square (\tau^n Q) = (\tau^n P) \square Q = \tau^n(P \square Q) \\ (\text{Vis } F \square \text{Vis } G) \gg H = (\text{Vis } F \gg H) \square (\text{Vis } G \gg H)$$

204 External choice is commutative and has *stop* as a unit. It has *div* as an annihilator, because  
 205 the  $\tau$  events means that no other activity is chosen. A finite number of  $\tau$  events on either  
 206 the left or right can be extracted to the front. Finally, bind distributes from the left across a  
 207 visible event choice. We prove these properties using coinduction (Theorem 4), followed by  
 208 several invocations of *sledgehammer* to discharge the resulting provisos.

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

210 **chantype** Chan = Input::integer Output::integer State::"integer list"

211

212 **definition** buffer :: "integer list  $\Rightarrow$  (Chan, integer list) itree" **where**

213 "buffer = loop ( $\lambda$  s.

214     do { i  $\leftarrow$  inp Input {0..}; Ret (s @ [i]) } }

215     □ do { guard(length s > 0); outp Output (hd s); Ret (tl s) } }

216     □ do { outp State s; Ret s } }"


217 We first create a channel type **Chan**, which has channels (prisms) for inputs and outputs,  
 218 and to view the current buffer state. We define the buffer process as a simple loop with a  
 219 choice with three branches inside. The variable **s::integer list** denotes the state. The  
 220 first branch allows a value to be received over **Input**, and then returns **s** with the new value  
 221 added, and then iterates. The second branch is only active when the buffer is not empty. It  
 222 outputs the head on **Output**, and then returns the tail. The final branch simply outputs the  
 223 current state. In §5 we will see how such an example can be simulated.

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

$$227 \quad \text{merge}_E(F, G) = (\lambda e \in \text{dom}(F) \setminus (\text{dom}(G) \cup E) \bullet \text{Left}(F(e))) \\ 228 \quad \quad \oplus (\lambda e \in \text{dom}(G) \setminus (\text{dom}(F) \cup E) \bullet \text{Right}(G(e))) \\ 229 \quad \quad \oplus (\lambda e \in \text{dom}(F) \cap \text{dom}(G) \cap E \bullet \text{Both}(F(e), G(e)))$$

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




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

$$\begin{aligned}
240 \quad (Vis F) \parallel_E (Vis G) &= Vis \left( \begin{array}{l} \{e \mapsto (P' \parallel_E (Vis G)) \mid (e \mapsto \mathit{Left}(P')) \in \mathit{merge}_A(F, G)\} \\ \oplus \{e \mapsto ((Vis F) \parallel_E Q') \mid (e \mapsto \mathit{Right}(Q')) \in \mathit{merge}_E(F, G)\} \\ \oplus \{e \mapsto (P' \parallel_E Q') \mid (e \mapsto \mathit{Both}(P', Q')) \in \mathit{merge}_E(F, G)\} \end{array} \right) \\
241 \quad (\mathit{Sil} P') \parallel_E Q &= \mathit{Sil}(P' \parallel_E Q) \quad P \parallel_E (\mathit{Sil} Q') = \mathit{Sil}(P \parallel_E Q') \\
242 \quad (\mathit{Ret} x) \parallel_E (\mathit{Ret} y) &= \mathit{Ret}(x, y) \\
243 \quad (\mathit{Ret} x) \parallel_E (Vis G) &= Vis \{e \mapsto \mathit{Ret} x \parallel_E Q' \mid (e \mapsto Q') \in G\} \\
244 \quad (Vis F) \parallel_E (\mathit{Ret} y) &= Vis \{e \mapsto P' \parallel_E \mathit{Ret} y \mid (e \mapsto P') \in F\}
\end{aligned}$$

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

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

We use  $\parallel_E$  to define two special cases for CSP:  $P \llbracket E \rrbracket Q \triangleq (P \parallel_E Q) \ggg (\lambda(x, y) \bullet \mathit{Ret}())$   
and  $P \parallel Q \triangleq P \llbracket \emptyset \rrbracket Q$ . As usual in CSP, these operators do not return any values and  
so  $P, Q :: (E, ())\mathit{itree}$ . The  $P \llbracket E \rrbracket Q$  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  $P \parallel Q$ , where there is no  
synchronisation, is simply defined as  $P \llbracket \emptyset \rrbracket Q$ . We prove several algebraic laws: 

$$(P \parallel_E Q) = (Q \parallel_E P) \ggg (\lambda(x, y) \bullet \mathit{Ret}(y, x)) \quad \mathit{div} \parallel_E P = \mathit{div}$$

$$P \llbracket E \rrbracket Q = Q \llbracket E \rrbracket P \quad P \parallel Q = Q \parallel P \quad \mathit{skip} \parallel P = P$$

256 Parallel composition is commutative, except that we must swap the outputs, and so  $\llbracket E \rrbracket$  and  
257  $\parallel$  are commutative as well. Parallel has  $\mathit{div}$  as an annihilator for similar reasons to  $\square$ . For  $\parallel$ ,  
258  $\mathit{skip}$  is a unit since there is no possibility of communication and no values are returned.

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


260 ► **Definition 10 (Hiding).**  $P \setminus A$  is defined corecursively by the following equations: 

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

264 We consider a restricted version of hiding where only one event can be hidden at a time, to  
265 avoid nondeterminism. When hiding the events of  $A$  in the choice function  $F$  there are three  
266 cases: (1) there is precisely one event  $e \in A$  enabled, in which case it is hidden; (2) no enabled  
267 event is in  $A$ , in which case the event remains visible; (3) more than one  $e \in A$  is enabled,  
268 and so we deadlock. We again impose maximal progress here, so that an enabled event to be

269 hidden is prioritised over other visible events:  $(a \rightarrow P \parallel b \rightarrow Q) \setminus \{a\} = \tau P$ , for example.  
 270 In spite of the significant restrictions on hiding, it supports the common pattern where one  
 271 output event is matched with an input event. Moreover, a priority can be placed on the  
 272 order in which events are hidden, rather than deadlocking, by sequentially hiding events.  
 273 Hiding can introduce divergence, as the following theorem shows:  $(\text{iter}(\text{sync } e)) \setminus e = \text{div}$ .

### 274 3.2 Circus


275 Whilst CSP processes can be parametrised to allow modelling state, there is no support for  
 276 explicit state operators like assignment. The *do* notation somewhat allows variables, but  
 277 these are immutable and are not preserved across iterations. *Circus* [42, 32] is an extension  
 278 of CSP that allows state variables. Given a state variable `buf::integer list`, the buffer  
 279 example can be expressed in *Circus* as follows: 

```
280   buf := [] ; loop((Input?(i) → buf := buf @ [i])
281                   □ ((length(buf) > 0) & Output!(hd buf) → buf := tl buf)
282                   □ State!(buf) → Skip)
283
```

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

285 In our work [15, 14, 16], each state variable is modelled as a lens [12],  $x :: \mathcal{V} \Longrightarrow \mathcal{S}$ . This  
 286 is a pair of functions  $\text{get} :: \mathcal{V} \Rightarrow \mathcal{S}$  and  $\text{put} :: \mathcal{S} \Rightarrow \mathcal{V} \Rightarrow \mathcal{S}$ , which query and update the  
 287 variables present in state  $\mathcal{S}$ , and satisfy intuitive algebraic laws [14]. They allow an abstract  
 288 representation of state spaces, where no explicit model is required to support the laws of  
 289 programming [22]. Lenses can be designated as independent,  $x \bowtie y$ , meaning they refer to  
 290 different regions of  $\mathcal{S}$ . An expression on state variables is simply a function  $e :: \mathcal{S} \Rightarrow \mathcal{V}$ , where  
 291  $\mathcal{V}$  is the return type. We can check whether an expression  $e$  uses a lens  $x$  using unrestriction,  
 292 written  $x \# e$ . If  $x \# e$ , then  $e$  does not use  $x$  in its valuation, for example  $x \# (y + 1)$ , when  
 293  $x \bowtie y$ . Updates to variables can be expressed using the notation  $[x_1 \rightsquigarrow e_1, x_2 \rightsquigarrow e_2, \dots]$ , with  
 294  $x_i :: \mathcal{V}_i \Longrightarrow \mathcal{S}$  and  $e_i :: \mathcal{S} \Rightarrow \mathcal{V}_i$ , which represents a function  $\mathcal{S} \Rightarrow \mathcal{S}$ .

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


297 ► **Definition 11** (Circus Operators). 

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

305 Operator  $\langle \sigma \rangle$  lifts a function  $\sigma : \mathcal{S} \Rightarrow \mathcal{S}$  to a KTree. It is principally used to represent  
 306 assignments, which can be constructed using our maplet notation, such that a single assign-  
 307 ment  $x := e$  is  $\langle [x \rightsquigarrow e] \rangle$ . Most of the remaining operators are defined by lifting of their  
 308 CSP equivalents. An output  $c!e \rightarrow P$  carries an expression  $e$ , rather than a value, which  
 309 can depend on the state variables. The main complexity is the *Circus* parallel operator,  
 310  $P \parallel_{ns_1 | E | ns_2} Q$ , which allows  $P$  and  $Q$  to act on disjoint portions of the state, characterised  
 311 by the name sets  $ns_1$  and  $ns_2$ . We represent  $ns_1$  and  $ns_2$  as independent lenses,  $ns_1 \bowtie ns_2$ ,  
 312 though they can be thought of as sets of variables with  $ns_1 \cap ns_2 = \emptyset$ . The definition of



313 the operator first lifts  $\parallel_E$ , and composes this with a merge function. The merge function  
 314 constructs a state that is composed of the  $ns_1$  region from the final state of  $P$ , the  $ns_2$  region  
 315 from  $Q$ , and the remainder coming from the initial state  $s$ . This is achieved using the lens  
 316 override operator  $s_1 \triangleleft_X s_2$ , which extracts the region described by  $X$  from  $s_2$  and overwrites  
 317 the corresponding region in  $s_1$ , leaving the complement unchanged.

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

$$\begin{aligned}
 319 \quad & \langle \sigma \rangle \circ \langle \rho \rangle = \langle \rho \circ \sigma \rangle \\
 320 \quad & \langle \sigma \rangle \circ (P \square Q) = (\langle \sigma \rangle \circ P) \square (\langle \sigma \rangle \circ Q) \\
 321 \quad & x := e \circ y := f = y := f \circ x := e \quad \text{if } x \bowtie y, x \# f, y \# e \\
 322 \quad & P \llbracket ns_1 | E | ns_2 \rrbracket Q = Q \llbracket ns_2 | E | ns_1 \rrbracket P \quad \text{if } ns_1 \bowtie ns_2
 \end{aligned}$$

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


## 329 4 Linking to Failures-Divergences Semantics

330 Next, we show how ITrees are related to the standard failures-divergences semantics of CSP [8].  
 331 The utility of this link is to both allow symbolic verification of ITrees and allow them to act  
 332 as a target of step-wise refinement. In this way, we can use existing the mechanisations of the  
 333 CSP set-based and relational semantics [39, 16] to capture and reason about nondeterministic  
 334 specifications, and use ITrees to provide executable implementations.

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

340 For example, consider the process  $a \rightarrow c \rightarrow \text{skip} \square b \rightarrow \text{div}$ , which initially permits an  $a$   
 341 or  $b$  event, and following  $a$  permits a  $c$  event. It exhibits the failure  $([], \{c\})$ , since before  
 342 any events are performed, the event  $c$  is being refused. A second failure is  $([a], \{a, b\})$ , since  
 343 after performing an  $a$ , only  $c$  is enabled and the other events are refused. A third failure  
 344 is  $([a, c, \checkmark], \{a, b, c\})$ , which represents successful termination, after which all events are  
 345 refused. This process also has a divergence trace  $[b]$ , since after performing event  $b$ , the  
 346 process diverges. If a divergent state is unreachable then  $D$  is empty. Here, we show how to  
 347 extract  $F$  and  $D$  from any ITree, and thus processes constructed from the operators of §3.

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


349 ► **Definition 12** (Big-Step Operational Semantics). 

$$\frac{}{P \Downarrow P} \quad \frac{P \xrightarrow{tr} P'}{\tau P \xrightarrow{tr} P'} \quad \frac{e \in E \quad F(e) \xrightarrow{tr} P'}{(\llbracket x \in E \bullet F(x) \rrbracket) \xrightarrow{e \# tr} P'}$$

350 The relation  $P \xrightarrow{tr} Q$  means that  $P$  can perform the trace of visible events contained in the  
 351 list  $tr : E \text{ list}$  and evolve to the ITree  $Q$ . This relation skips over  $\tau$  events. The first rule  
 352 states that any ITree may perform an empty trace  $([])$  and remain at the same state. The  
 353 second rule states that if  $P$  can evolve to  $P'$  by performing  $tr$ , then so can  $\tau P$ . The final rule  
 354 states that if  $e$  is an enabled visible event, and  $P(e)$  can evolve to  $P'$  by doing  $tr$ , then the

## 17:10 Formally Verified Simulations of State-Rich Processes using Interaction Trees


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). 

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

The `skip` process immediately terminates, returning  $()$ . If the left-hand side  $P$  of  $\gg$  can evolve to  $P'$  performing the events in  $tr$ , 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  $tr_2$ ,  $tr_1 @ tr_2$ , 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). 

$$\frac{-}{\check{x} \Downarrow \mathcal{R}} \quad \frac{P \Downarrow \mathcal{R}}{\tau P \Downarrow \mathcal{R}} \quad \frac{\text{ran}(F) \subseteq \mathcal{R}}{\text{Vis } F \Downarrow \mathcal{R}} \quad \text{div-free} \triangleq \bigcup \{ \mathcal{R} \mid \mathcal{R} \subseteq \{ P \mid P \Downarrow \mathcal{R} \} \}$$


Predicate  $P \Downarrow \mathcal{R}$  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  $\mathcal{R} = \{ P \mid P \Downarrow \mathcal{R} \}$ , and is coinductively defined. If we can find an  $\mathcal{R}$  such that for every  $P \in \mathcal{R}$ , it follows that  $P \Downarrow \mathcal{R}$ , 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  $\mathcal{R} = \{ \text{run } E \}$  satisfies the provisos and so `run E` is divergence free. We have proved that  $P \in \text{div-free} \Leftrightarrow (\nexists s \bullet P \xrightarrow{s} \text{div})$ , 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). 

$$(P \xRightarrow{s} P') \triangleq ((\exists t \in \Sigma \text{ list} \bullet s = t @ [\check{x}] \wedge P \xrightarrow{t} \check{x} \wedge P' = \text{stop}) \vee (\text{set}(s) \subseteq \Sigma \wedge P \xrightarrow{s} P'))$$

Here, `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  $\check{}$ . We define the enlarged set  $\Sigma^\check{} \triangleq \Sigma \cup \{ \check{x} \mid x \in \mathcal{S} \}$ , 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 `Ret x` in the ITree, in which case the trace is augmented with  $\check{x}$  and the successor state is set to `stop`. We often use a condition of the form  $\text{set}(s) \subseteq \Sigma$  to mean that no  $\check{x}$  event is in  $s$ . We can now define the sets of traces, failures, and divergences [36]:

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

$$\text{traces}(P) \triangleq \{ s \mid \text{set}(s) \subseteq \Sigma^\check{} \wedge (\exists P' \bullet P \xRightarrow{s} P') \}$$

$$P \text{ ref } E \triangleq ((\exists F \bullet P = \text{Vis } F \wedge E \cap \text{dom}(F) = \emptyset) \vee (\exists x \bullet P = \text{Ret } x \wedge \check{x} \notin E))$$

$$\text{failures}(P) \triangleq \{ (s, X) \mid \text{set}(s) \subseteq \Sigma^\check{} \wedge (\exists Q \bullet P \xRightarrow{s} Q \wedge Q \text{ ref } X) \}$$


$$\text{divergences}(P) \triangleq \{ s @ t \mid \text{set}(s) \subseteq \Sigma \wedge \text{set}(t) \subseteq \Sigma \wedge (\exists Q \bullet P \xRightarrow{s} Q \wedge Q \uparrow) \}$$

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

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

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

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

408 ► **Theorem 17** (Semantic Model Properties). 

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

415 The first two are standard healthiness conditions of the failures-divergences model [36], called  
 416 **F3** and **D1**, respectively. **F3** states that if  $(s, X)$  is a failure of  $P$  then any event that cannot  
 417 subsequently occur after  $s$ , according to the  $traces$ , must also be refused. **D1** states that  
 418 the set of divergences is extension closed. We have also proved that two weakly bisimilar  
 419 processes have the same set of divergences and failures. The next result links the coinductive  
 420 definition of divergence freedom and the set of divergences. The final result demonstrates  
 421 that ITrees satisfy Roscoe's definition of determinism for CSP [36]: if an ITree  $P$  is divergence  
 422 free then there is no trace after which an event can be both accepted and also refused.

## 423 5 Simulation by Code Generation

424 The Isabelle code generator [19, 18] can be used to extract code from (co)datatypes, functions,  
 425 and other constructs, to functional languages like SML, Haskell, and Scala. Although ITrees  
 426 can be infinite, this is not a problem for languages with lazy evaluation, and so we can step  
 427 through the behaviour of an ITree. Code generation then allows us to support generation of  
 428 verified simulators, and provides a potential route to correct implementations.

429 The main complexity is a computable representation of partial functions. Whilst  $A \rightarrow B$   
 430 is partly computable, all that we can do is apply it to a value and see whether it yields an  
 431 output or not. For simulations and implementations, however, we typically want to determine  
 432 a menu of enabled events for the user to select from. Moreover, calculation of a semantics for

```

*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




433 CSP operators like  $\square$  and  $\parallel$  requires us to compute with partial functions. For this, we need  
 434 a way of calculating values for functions  $\text{dom}$ ,  $\triangleleft$ , and  $\oplus$ , which is not possible for arbitrary  
 435 partial functions. Instead, we need a concrete implementation and a data refinement [18].

436 We choose associative lists as an implementation,  $A \rightarrow B \approx (A \times B) \text{ list}$ , which limits  
 437 us to finite constructions. However, it has the benefit of being easily pretty printed and so  
 438 makes the simulator easier to implement. More sophisticated implementations are possible,  
 439 as the core theory of ITrees is separated from the code generation setup. To allow us to  
 440 represent partial functions by associative lists, we need to define a mapping function:

```

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

```

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

```

448 (pfun_alist f) ⊕ (pfun_alist g) = pfun_alist (g @ f)
449 A < (pfun_alist f) = pfun_alist (AList.restrict A m)
450 (λ x ∈ (set xs) • f(x)) = pfun_alist (map (λ k • (k, f k)) xs)
451

```

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

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

```

459 data Set a = Set [a] | Coset [a] deriving (Read, Show);
460

```

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

```

467 member :: forall a. (Eq a) => a -> Set a -> Bool;
468 member x (Coset xs) = not (x 'elem' xs); member x (Set xs) = xs 'elem' x;
469
470

```

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:  $\gg$

```

475 data Pfun a b = Pfun_alist [(a, b)];
476
477
478 dom :: forall a b. Pfun a b -> Set a;
479 dom (Pfun_alist xs) = Set (map fst xs);
480

```

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:  $\gg$

```

485 data Itree a b = Ret b | Sil (Itree a b) | Vis (Pfun a (Itree a b));
486
487

```

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

```

490 extchoice :: (Eq a, Eq b) => Itree a b -> Itree a b -> Itree a b;
491 extchoice p q = (case (p, q) of {
492   (Ret r, Ret y) -> (if r == y then Ret r else Vis zero_pfun);
493   (Ret _, Sil qa) -> Sil (extchoice p qa); (Ret r, Vis _) -> Ret r;
494   (Sil pa, _) -> Sil (extchoice pa q); (Vis _, Ret a) -> Ret a;
495   (Vis _, Sil qa) -> Sil (extchoice p qa);
496   (Vis f, Vis g) -> Vis (map_prod f g); });
497
498

```

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 ( $\square$ ) 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, *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. 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:  $\gg$

```

509 sim_cnt :: (Eq e, Show e, Read e, Show s) => Int -> Itree e s -> IO ();
510 sim_cnt n (Ret x) = putStrLn ("Terminated:\_ " ++ show x);
511 sim_cnt n (Sil p) =
512   do { if (n == 0) then putStrLn "Internal\_Activity..." else return ();
513       if (n >= 20)
514         then do { putStr "Many\_steps\_(>\_20);\_Continue?"; q <- getLine;
515                 if (q=="Y") then sim_cnt 0 p else putStrLn "Ended."; }
516         else sim_cnt (n + 1) p };
517 sim_cnt n (Vis (Pfun_alist [])) = putStrLn "Deadlocked.";
518 sim_cnt n t@(Vis (Pfun_alist m)) =
519   do { putStrLn ("Events:\_" ++ show (map fst m)); e <- getLine;
520       case (reads e) of
521

```


```

522         []          -> do { putStrLn "No_parse"; sim_cnt n t }
523         [(v, _)] -> case (lookup v m) of
524             Nothing -> do { putStrLn "Rejected"; sim_cnt n t }
525             Just k   -> sim_cnt 0 k };
526 simulate = sim_cnt 0;
527

```

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


535 In order to apply the simulator, we need only augment the generated code for a particular  
536 ITree with the simulator code. Figure 1 shows a simulation of the CSP buffer in §3, with the  
537 possible inputs limited to  $\{0..3\}$ . We provide an empty list as a parameter for the initial  
538 state. The simulator tells us the events enabled, and allows us to pick one. If we try and  
539 pick a value not enabled, the simulator rejects this. Since lenses and expressions can also be  
540 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: 

$$(Controller \llbracket \{rd.c, wrt.c \mid c \in \mathbb{N}\} \rrbracket (\llbracket i \in \{0..maxbuff\} \bullet Cell(i) \rrbracket) \setminus \{rd.c, wrt.c \mid c \in \mathbb{N}\})$$

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

$$Cell(i) \triangleq wrt?c \rightarrow val := v \ ; \ loop(wrt?c \rightarrow val := v \ \square \ rd!val \rightarrow Skip)$$

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

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

## 551 **6** Related Work

552 Infinite trees are a ubiquitous model for concurrency [40]. In particular, ITrees can be seen  
553 as a restricted encoding of Milner's synchronisation trees [27, 41, 28]. In contrast to ITrees,  
554 synchronisation trees allow multiple events from each node, including both visible and  $\tau$   
555 events. They have seen several generalisations, most recently by Ferlez et al. [10], who  
556 formalise Generalized Synchronisation Trees based on partial orders, define bisimulation  
557 relations [11], and apply them to hybrid systems. Our work is different, because ITrees use  
558 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  $a \rightarrow P \mid b \rightarrow Q$  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  $k : \mathcal{A} \rightarrow \text{itree } \mathcal{E} \mathcal{R}$ , 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 *Vis*  $e k$  as  $\llbracket x \in \text{dom}(\text{match}_e) \rightarrow k(\text{match}_e(x)) \rrbracket$  with  $e : A \xrightarrow{\Delta} E$ . 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  $\tau$ ; (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  $\checkmark$ -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.

## 605 — References

- 606 1 R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
- 607 2 J. Baxter, P. Ribeiro, and A. Cavalcanti. Sound reasoning in tock-CSP. *Acta Informatica*,  
608 April 2021. doi:[10.1007/s00236-020-00394-3](https://doi.org/10.1007/s00236-020-00394-3).
- 609 3 J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with Benefits:  
610 Implementing Corecursion in Foundational Proof Assistants. In *Programming Languages and*  
611 *Systems, 26th European Symposium on Programming (ESOP)*, April 2017.
- 612 4 J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular  
613 (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *5th Intl. Conf.*  
614 *on Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.
- 615 5 J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal*  
616 *of Formalized Reasoning*, 9(1), 2016. doi:[10.6092/issn.1972-5787/4593](https://doi.org/10.6092/issn.1972-5787/4593).
- 617 6 J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof  
618 assistant perspective. In *20th Intl. Conf. on Functional Programming (ICFP)*, pages 192–204.  
619 ACM, August 2015. doi:[10.1145/2858949.2784732](https://doi.org/10.1145/2858949.2784732).
- 620 7 J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by  
621 coinductive methods. *Journal of Automated Reasoning*, 58:149–179, 2017. doi:[10.1007/](https://doi.org/10.1007/s10817-016-9391-3)  
622 [s10817-016-9391-3](https://doi.org/10.1007/s10817-016-9391-3).
- 623 8 S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential  
624 processes. *Journal of the ACM*, 31(3):560–599, 1984. doi:[10.1145/828.833](https://doi.org/10.1145/828.833).
- 625 9 A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. Filho, A. Didier, W. Li, and  
626 J. Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37,  
627 2019. doi:[10.1016/j.scico.2019.01.004](https://doi.org/10.1016/j.scico.2019.01.004).
- 628 10 J. Ferlez, R. Cleaveland, and S. Marcus. Generalized synchronization trees. In *Proc. 17th Intl.*  
629 *Conf. on Foundations of Software Science and Computation Structures (FOSSACS)*, volume  
630 8412 of *LNCS*, pages 304–319. Springer, 2014. doi:[10.1007/978-3-642-54830-7\\_20](https://doi.org/10.1007/978-3-642-54830-7_20).
- 631 11 J. Ferlez, R. Cleaveland, and S. I. Marcus. Bisimulation in behavioral dynamical systems and  
632 generalized synchronization trees. In *Proc. 2018 IEEE Conf. on Decision and Control (CDC)*,  
633 pages 751–758. IEEE, 2018. doi:[10.1109/CDC.2018.8619607](https://doi.org/10.1109/CDC.2018.8619607).
- 634 12 J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
- 635 13 S. Foster. Hybrid relations in Isabelle/UTP. In *7th Intl. Symp. on Unifying Theories of*  
636 *Programming (UTP)*, volume 11885 of *LNCS*, pages 130–153. Springer, 2019.
- 637 14 S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations  
638 for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197,  
639 October 2020. doi:[10.1016/j.scico.2020.102510](https://doi.org/10.1016/j.scico.2020.102510).
- 640 15 S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. Unifying theories of  
641 reactive design contracts. *Theoretical Computer Science*, 802:105–140, January 2020. doi:  
642 [10.1016/j.tcs.2019.09.017](https://doi.org/10.1016/j.tcs.2019.09.017).
- 643 16 S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Automated verification of reactive and  
644 concurrent programs by calculation. *Journal of Logical and Algebraic Methods in Programming*,  
645 121, June 2021. doi:[10.1016/j.jlamp.2021.100681](https://doi.org/10.1016/j.jlamp.2021.100681).
- 646 17 T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 — A Modern  
647 Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and*  
648 *Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 187–201,  
649 2014.
- 650 18 F. Haftmann, A. Krauss, O. Kuncar, and T. Nipkow. Data refinement in Isabelle/HOL. In  
651 *Proc. 4th Intl. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages  
652 100–115. Springer, 2013.
- 653 19 F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *10th Intl.*  
654 *Symp. on Functional and Logic Programming (FLOPS)*, volume 6009 of *LNCS*, pages 103–117.  
655 Springer, 2010.

- 656 20 Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Information and*  
657 *Computation*, 117(2):221–239, 1995.
- 658 21 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- 659 22 C. A. R. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and  
660 B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August  
661 1987.
- 662 23 Nicolas Koh, Yao Li, Yishuai Li, Li yao Xia, Lennart Beringer, Wolf Honoré, William Mansky,  
663 Benjamin C. Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying,  
664 and Testing a Networked Server. In *Proc. 8th ACM SIGPLAN International Conference on*  
665 *Certified Programs and Proofs (CPP)*, 2019. doi:10.1145/3293880.3294106.
- 666 24 M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int J*  
667 *Softw Tools Technol Transf*, 10:185–203, 2008. doi:10.1007/s10009-007-0063-9.
- 668 25 Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked  
669 applications. In *Proc. 30th ACM SIGSOFT International Symposium on Software Testing*  
670 *and Analysis (ISSTA)*, 2021.
- 671 26 William Mansky, Wolf Honoré, and Andrew W. Appel. Connecting higher-order separation  
672 logic to a first-order outside world. In *Proc. 29th European Symposium on Programming*  
673 *(ESOP)*, 2020.
- 674 27 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer*  
675 *Science*. Springer, 1980.
- 676 28 Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- 677 29 A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart:  
678 modelling and verification of the functional behaviour of robotic applications. *Software and*  
679 *Systems Modelling*, January 2019. doi:10.1007/s10270-018-00710-z.
- 680 30 C. Morgan. *Programming from Specifications*. Prentice-Hall, January 1996.
- 681 31 J. H. Y. Munive, G. Struth, and S. Foster. Differential Hoare logics and refinement calculi for  
682 hybrid systems with Isabelle/HOL. In *RAMiCS*, volume 12062 of *LNCS*. Springer, April 2020.  
683 doi:10.1007/978-3-030-43520-2\_11.
- 684 32 M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of*  
685 *Computing*, 21:3–32, 2009. doi:10.1007/s00165-007-0052-5.
- 686 33 M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *The Art,*  
687 *Science, and Engineering of Programming*, 1(2), 2017. doi:10.22152/programming-journal.  
688 org/2017/1/7.
- 689 34 A. W. Roscoe. Denotational semantics for Occam. In *Intl. Seminar on Concurrency*, volume  
690 197 of *LNCS*, pages 306–329. Springer, 1984.
- 691 35 A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 2005.
- 692 36 A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
- 693 37 Lucas Silver and Steve Zdancewic. Dijkstra monads forever: Termination-sensitive specifica-  
694 tions for Interaction Trees. *Proceedings of the ACM on Programming Languages*, 5(POPL),  
695 January 2021. doi:10.1145/3434307.
- 696 38 M. Spivey. *The Z-Notation - A Reference Manual*. Prentice Hall, Englewood Cliffs, N. J.,  
697 1989.
- 698 39 S. Taha, B. Wolff, and L. Ye. Philosophers may dine – definitively! In *Proc. 16th Intl. Conf.*  
699 *on Integrated Formal Methods*, LNCS. Springer, 2020. doi:10.1007/978-3-030-63461-2\_23.
- 700 40 R. J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer*  
701 *Science*, 1997.
- 702 41 G. Winsel. Synchronisation trees. *Theoretical Computer Science*, 34(1-2):33–82, 1984.
- 703 42 J. Woodcock and A. Cavalcanti. A concurrent language for refinement. In A. Butterfield,  
704 G. Strong, and C. Pahl, editors, *Proc. 5th Irish Workshop on Formal Methods (IWFm)*,  
705 Workshops in Computing. BCS, July 2001.

- 706 43 L.-Y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic.  
707 Interaction trees: Representing recursive and impure programs in Coq. In *Proc. 47th ACM*  
708 *SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2020. doi:  
709 [10.1145/3371119](https://doi.org/10.1145/3371119).
- 710 44 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for  
711 weak bisimulation via generalized parameterized coinduction. In *Proc. 9th ACM SIGPLAN*  
712 *International Conference on Certified Programs and Proofs (CPP)*, 2020. doi:[10.1145/](https://doi.org/10.1145/3372885.3373813)  
713 [3372885.3373813](https://doi.org/10.1145/3372885.3373813).
- 714 45 Vadim Zaliva, Iliia Zaichuk, and Franz Franchetti. Verified translation between purely functional  
715 and imperative domain specific languages in HELIX. In *Proc. 12th International Conference*  
716 *on Verified Software: Theories, Tools, Experiments (VSTTE)*, 2020.
- 717 46 Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer,  
718 William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value  
719 server with Interaction Trees and VST. In *Proc. 12th International Conference on Interactive*  
720 *Theorem Proving (ITP)*, 2021. doi:[10.4230/LIPIcs.ITP.2021.32](https://doi.org/10.4230/LIPIcs.ITP.2021.32).