HOL-CSP : des erreurs à rattraper

Benoît Ballenghien

HOL-CSP : des erreurs à rattraper

Throw et Interrupt en HOL-CSP

Benoît Ballenghien[1]

[1]Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, 91190 Gif-sur-Yvette, France,
benoit.ballenghien@universite-paris-saclay.fr

La théorie Communicating Sequential Processes (CSP) initialement introduite par Hoare est encore aujourd'hui une référence pour modéliser les systèmes concurrents. Récemment, avec l'essor des assistants de preuve, des implémentations notamment en Coq, Agda et Isabelle ont été réalisées. Ce travail est basé sur la troisième : `Isabelle/HOL-CSP`, un plongement superficiel qui généralise le modèle échec/divergence de la sémantique dénotationnelle proposée par Hoare, Roscoe et Brookes dans les années quatre-vingt. Nous décrivons dans cet article l'ajout de deux opérateurs qui n'étaient pas encore supportés, *Interrupt* et *Throw* ; depuis la correction des définitions de la littérature jusqu'aux monotonies et lois algébriques formellement démontrées.

1 Introduction

Communicating Sequential Processes (CSP) est un langage permettant de spécifier et vérifier des patrons d'interaction de systèmes concurrents. Avec LOTOS et CSS, il appartient à la famille des *algèbres de processus*. D'abord décrit par Hoare en 1978, puis détaillé dans son livre de 1985 [Hoa85], il a significativement évolué depuis [BHR84, BR85, Ros97] si bien que la sémantique comprend aujourd'hui trois facettes : dénotationnelle, opérationnelle et algébrique. La première consiste en un modèle abstrait qui se veut *compositionnel*. Un processus est donc décrit par tous ses comportements possibles, c'est-à-dire ses *traces* possiblement annotées d'informations permettant de raisonner sur les interblocages (*deadlocks*) : sémantique d'*échec* (failure) $\mathcal{F}$, et les interblocages actifs (*livelocks*) : sémantique d'*échec/divergence* $\mathcal{FD}$.

On peut citer plusieurs tentatives de formalisation de cette théorie, notamment [TW97, IR10, Noc16, IS16]. Le travail présenté ici s'appuie sur `Isabelle/HOL-CSP`, un plongement superficiel (*shallow embedding*) de la sémantique dénotationnelle de CSP dans l'assistant de preuve `Isabelle/HOL` dont les objectifs sont à la fois la vérification d'architectures de processus [TWY20, BW25] et l'étude de méta-propriétés sur les sémantiques [BW24].

Dans cette formalisation déjà conséquente, deux opérateurs n'étaient pas encore supportés : *Throw* et *Interrupt*. L'article que nous proposons ici décrit cet ajout ; depuis les motivations et justifications des définitions dénotationnelles choisies jusqu'aux esquisses de preuve des propriétés algébriques formellement établies.

Ce travail est original et complémentaire d'une part de [BW24] où ces deux nouveaux opérateurs ont récemment été utilisés et d'autre part de [Ros10] puisque nous motivons et proposons une correction de la définition donnée pour *Interrupt*.

Les fichiers sources peuvent être consultés [TYW19, BTW23, BW23] [1] sur l'AFP (*Archive of Formal Proofs*, une collection de bibliothèques de formalisations et preuves vérifiées mécaniquement dans Isabelle).

1. Version de développement : https://gitlab.lisn.upsaclay.fr/burkhart.wolff/hol-csp2.0

2 Contexte

2.1 Syntaxe de CSP

Voici en résumé le fragment du langage CSP déjà supporté par `HOL-CSP`.

$$P ::= SKIP \mid STOP \mid P \,\Box\, P' \mid P \sqcap P' \mid \Box a \in A \to P\ a$$
$$\mid P \,;\, P' \mid P \,[\![A]\!]\, P' \mid P \setminus A \mid Renaming\ P\ g \mid \mu\ X.\ f\ X$$

Les deux processus constants *SKIP* et *STOP* signalent respectivement la terminaison et un interblocage (*deadlock*). Ensuite deux opérateurs de choix sont à distinguer :

— $P \,\Box\, P'$, choix *externe*, forçant un processus à suivre le contexte

— $P \sqcap P'$, choix *interne*, imposant au contexte de suivre les choix non-déterministes faits par le processus. Une version non-bornée est aussi supportée : $\sqcap b \in B.\ P\ b$.

Intuitivement, le choix multi-préfixe déterministe $\Box a \in A \to P\ a$ choisit (choix externe) un événement a de l'ensemble A, le signale, puis se comporte ensuite comme $P\ a$. Quand A est un singleton, on abrège avec l'opérateur préfixe $a \to P$; et quand les événements sont structurés avec des canaux, les notations dérivées $c\,?x \to P\ x$ et $c\,!x \to P\ x$ signifient respectivement « x est lu depuis le canal c » et « x est envoyé dans le canal c ».

La composition séquentielle $P \,;\, P'$ se comporte comme P dans un premier temps puis, une fois que P a terminé normalement, comme P'. La synchronisation $P \,[\![A]\!]\, P'$ permet quant à elle de laisser les processus P et P' évoluer en même temps mais les force bien sûr à s'accorder lorsqu'ils souhaitent performer un événement de l'ensemble A.

Il est aussi possible de manipuler les événements d'un processus : cacher ceux de P présents dans un ensemble A via $P \setminus A$ ou renommer e en $g(e)$ via $Renaming\ P\ g$.

Enfin l'opérateur de point-fixe $\mu\ X.\ f\ X$ retourne une solution à l'équation $P = f\ P$ sous certaines conditions sur f (cf 2.3).

2.2 Sémantique dénotationnelle

La sémantique dénotationnelle de CSP est un modèle abstrait compositionnel, autrement dit un processus est décrit par tous ses comportements possibles. Pour simplifier, elle se décline en trois variantes [Ros97] : *trace*, *échec* (stable) et *échec/divergence*.

Dans le modèle de trace, le comportement d'un processus est décrit par un ensemble de traces qui est préfixe-clos, noté $\mathcal{T}\ P$. Comme ce sont simplement des listes (finies), les comportements infinis sont représentés par l'ensemble des approximations et un événement spécial *tick* (noté ✓) est utilisé pour représenter la terminaison explicite. Bien sûr, ✓ ne peut apparaître qu'à la fin d'une trace t, ce qu'on abrège en *ftF* t (pour *front-tickFree* t).

Exemple 1 (*STOP* et *SKIP* dans la sémantique de trace).
STOP ne peut engager aucun événement, il est donc décrit par $\{[]\}$.
SKIP ne peut engager que ✓ et s'arrête ensuite, il est donc décrit par $\{[], [✓]\}$.

Pour distinguer le choix externe du choix interne (impossible avec seulement des traces), [BHR84] propose de les annoter avec un ensemble de *refus*. Ceci donne naissance à la notion d'échec $(t,\ X) \in \mathcal{F}\ P$: une paire formée d'une trace t et d'un ensemble de refus X.

Exemple 2 (*STOP* et *SKIP* dans la sémantique d'échec).
STOP doit tout refuser, il est donc décrit par $\{(t,\ X) \mid t = []\}$. *SKIP* ne peut pas refuser ✓ au début, il est donc décrit par $\{([],\ X) \mid ✓ \notin X\} \cup \{(t,\ X) \mid t = [✓]\}$.

Enfin, pour distinguer un interblocage d'un interblocage actif (*livelock*) [2], [BHR84] propose d'ajouter l'ensemble des divergences, noté $\mathcal{D}\ P$, qui sont simplement des traces amenant à une situation où un interblocage actif peut advenir. Il en résulte le modèle échec/divergence dans lequel le domaine sémantique consiste en une paire de deux ensembles éponymes : les échecs et les divergences.

2. comme dans le processus $\mu\ X.\ (a \to X) \setminus \{a\}$

Exemple 3 (*STOP* et *SKIP* dans la sémantique d'échec/divergence).
STOP et *SKIP* n'ont pas de divergence, et leurs échecs restent inchangés.

Notons que comme les processus CSP ne terminent pas forcément, établir des propriétés sur eux passe la plupart du temps par la notion de raffinement. Il en existe plusieurs mais le plus important pour nous (car étant trivialement un ordre partiel pour le troisième modèle) est la version échec/divergence :

$$P \sqsubseteq_{FD} Q \equiv \mathcal{F}\ P \supseteq \mathcal{F}\ Q \wedge \mathcal{D}\ P \supseteq \mathcal{D}\ Q$$

et outre évidemment le raffinement de protocole, des propriétés comme l'absence d'interblocage s'expriment avec : $\textit{deadlock-free}\ P \equiv (\mu\ X.\ \sqcap a \in \textit{UNIV}.\ a \rightarrow X) \sqsubseteq_{FD} P$.

2.3 Isabelle/HOL-CSP

`Isabelle/HOL-CSP` est un plongement superficiel du modèle échec/divergence de CSP en `Isabelle/HOL`. La construction commence par la définition des événements similairement à un type option afin de « rajouter » à un type arbitraire $'a$ la terminaison explicite ✓.

$$\textbf{datatype}\ 'a\ \textit{event} = \textit{ev}\ 'a \mid \textit{tick}\ (\langle\checkmark\rangle)$$

Viennent ensuite les définitions des traces, refus, échecs et divergences.

$$\begin{aligned}
&\textbf{type-synonym}\ 'a\ \textit{trace} &&= \langle 'a\ \textit{event list}\rangle\\
&\textbf{type-synonym}\ 'a\ \textit{refusal} &&= \langle 'a\ \textit{event set}\rangle\\
&\textbf{type-synonym}\ 'a\ \textit{failure} &&= \langle 'a\ \textit{trace} \times 'a\ \textit{refusal}\rangle\\
&\textbf{type-synonym}\ 'a\ \textit{divergence} &&= \langle 'a\ \textit{trace}\rangle\\
&\textbf{type-synonym}\ 'a\ \textit{process}_0 &&= \langle 'a\ \textit{failure set} \times 'a\ \textit{divergence set}\rangle
\end{aligned}$$

Etant donnée une trace t, on définit de plus le prédicat $tF\ t$ pour $\textit{tickFree}\ t$ (resp. $\textit{ftF}\ t$ pour $\textit{front-tickFree}\ t$) signifiant que ✓ n'apparaît pas dans t (resp. s'il apparaît, c'est à la fin).

Vient alors le moment de formaliser sur le type $'a\ \textit{process}_0$ qui constitue notre paire d'échecs et de divergences un invariant permettant de discriminer les processus bien formés.

Définition 1. P est bien formé si les neuf conditions [TW97, Ros10] suivantes sont vérifiées.

$$\begin{aligned}
&\text{T1}: && [] \in \mathcal{T}\ P\\
&\text{T2}: && \forall\, t\ X.\ (t, X) \in \mathcal{F}\ P \longrightarrow \textit{ftF}\ t\\
&\text{T3}: && \forall\, t\ u.\ t\ @\ u \in \mathcal{T}\ P \longrightarrow t \in \mathcal{T}\ P\\
&\text{T4}: && \forall\, t\ X\ Y.\ (t, Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (t, X) \in \mathcal{F}\ P\\
&\text{T5}: && \forall\, t\ X\ Y.\ (t, X) \in \mathcal{F}\ P \wedge (\forall\, e.\ e \in Y \longrightarrow t\ @\ [e] \notin \mathcal{T}\ P) \longrightarrow (t, X \cup Y) \in \mathcal{F}\ P\\
&\text{T6}: && \forall\, t\ X.\ t\ @\ [\checkmark] \in \mathcal{T}\ P \longrightarrow (t, X - \{\checkmark\}) \in \mathcal{F}\ P\\
&\text{T7}: && \forall\, t\ u.\ t \in \mathcal{D}\ P \wedge tF\ t \wedge \textit{ftF}\ u \longrightarrow t\ @\ u \in \mathcal{D}\ P\\
&\text{T8}: && \forall\, t\ X.\ t \in \mathcal{D}\ P \longrightarrow (t, X) \in \mathcal{F}\ P\\
&\text{T9}: && \forall\, t.\ t\ @\ [\checkmark] \in \mathcal{D}\ P \longrightarrow t \in \mathcal{D}\ P
\end{aligned}$$

Cet invariant est capturé via la spécification

$$\textbf{typedef}\ 'a\ \textit{process} = \langle\{P :: 'a\ \textit{process}_0.\ \textit{is-process}\ P\}\rangle$$

qui axiomatise un nouveau type $'a\ \textit{process}$ isomorphe au sous-ensemble des éléments du type $'a\ \textit{process}_0$ vérifiant le prédicat *is-process*. Chaque opérateur est alors défini par ses échecs et divergences au niveau $'a\ \textit{process}_0$, et la machinerie d'Isabelle relève automatiquement cette définition au niveau $'a\ \textit{process}$ (sous réserve bien sûr qu'elle préserve l'invariant *is-process*).

La sémantique de l'opérateur de point-fixe requiert d'ajouter de la structure.

Définition 2 (Ordre partiel proposé par [Ros92] pour le modèle échec/divergence)**.**
Pour deux processus P et Q, on définit $P \sqsubseteq Q$ par $\mathcal{D}\ Q \subseteq \mathcal{D}\ P$, $\textit{min-elems}\ (\mathcal{D}\ P) \subseteq \mathcal{T}\ Q$ et $\forall\, t\ X.\ t \notin \mathcal{D}\ P \longrightarrow ((t, X) \in \mathcal{F}\ P \longleftrightarrow (t, X) \in \mathcal{F}\ Q)$.

En plus d'être un ordre partiel, ($\sqsubseteq$) est complet, ce qui signifie que chaque chaîne (application $Y :: nat \Rightarrow 'a\ process$ telle que $\forall\, i.\ Y\ i \sqsubseteq Y\ (i + 1)$) possède un supremum. Autrement dit il existe X tel que $\forall\, i.\ Y\ i \sqsubseteq X$ et $\forall\, X'.\ (\forall\, i.\ Y\ i \sqsubseteq X') \longrightarrow X \sqsubseteq X'$. La preuve consiste à définir les échecs et divergences de X par respectivement l'intersection des échecs et l'intersection des divergences des $Y\ i$ puis prouver que l'invariant est satisfait et enfin vérifier que c'est bien un supremum pour Y. Un tel supremum étant nécessairement unique, on le note $\bigsqcup i.\ Y\ i$. L'ordre partiel ($\sqsubseteq$) est de plus pointé : un certain processus $\bot$ (caractérisé par $[] \in \mathcal{D}\ \bot$) vérifie $\forall\, P.\ \bot \sqsubseteq P$.

Finalement on a prouvé que $'a\ process$ est une instance de la classe *pcpo* (*pointed complete partial order*) et on hérite alors de la bibliothèque `HOLCF` qui implémente la logique de Scott pour les fonctions calculables. Notons que cette théorie est compatible avec le produit et l'ordre supérieur ($'a\ process \times 'b\ process$ et $'b \Rightarrow 'a\ process$ sont encore des *pcpo*) et qu'elle introduit plusieurs concepts, notamment celui de *continuité*. Sans grande surprise, il s'agit de pouvoir échanger la limite des images avec l'image de la limite, limite étant à comprendre au sens de supremum. Formellement, f est continue lorsque pour toute chaîne Y, l'application $f \circ Y$ admet encore un supremum qui vérifie $(\bigsqcup i.\ f\ (Y\ i)) = f\ (\bigsqcup i.\ Y\ i)$. Sous cette hypothèse, l'opérateur de plus petit (au sens de ($\sqsubseteq$)) point-fixe $\mu\ X.\ f\ X$ est bien défini.

En ce qui nous concerne, les opérateurs de `HOL-CSP` sont tous continus (selon toutes leurs variables s'ils en ont plusieurs) avec éventuellement une hypothèse supplémentaire.

— Pour $\sqcap b \in B.\ P\ b$, l'ensemble B est supposé fini.
— Pour $P \setminus A$, l'ensemble A est supposé fini.
— Pour $Renaming\ P\ g$, on suppose que la préimage par g de tout singleton est finie. Ceci est bien sûr équivalent à dire que la préimage par g de tout ensemble fini est finie.

3 L'opérateur Throw

3.1 Intuition et définition dénotationnelle

Comme son nom le laisse présager, *Throw* permet de gérer les exceptions. Il est défini pour un ensemble A et deux processus P et Q : l'idée est de se comporter d'abord comme P puis, dès lors qu'il performe un élément présent dans l'ensemble $ev\ `\ A$, se comporter comme $Q$ en abandonnant $P$. Dans une certaine mesure, cet opérateur peut être vu comme une généralisation de la composition séquentielle $P\ ;\ Q$ où le signal de terminaison pour $P$ (donc de début pour $Q$) n'est plus ✓ mais n'importe quel événement $ev\ a \in ev\ `\ A$. Toutefois $ev\ a$ ne sera pas caché comme c'est le cas avec ✓ dans $P\ ;\ Q$, il faudrait pour cela utiliser additionnellement l'opérateur *Hiding*.

Afin de permettre au processus de droite de dépendre de l'exception attrapée, nous généralisons la définition pour $P :: 'a\ process$ et $Q :: 'a \Rightarrow 'a\ process$, ce que l'on met en exergue avec la notation suivante : $P\ \Theta\ a{\in}A.\ Q\ a$.

Exemple 4 (Utilisation de l'opérateur *Throw*).
Imaginons que l'on dispose d'un processus P qui, s'il performe un certain élément ✗, peut malheureusement aboutir à une situation d'interblocage. Pour l'éviter, on peut « réparer » P en utilisant plutôt $P\ ' \equiv P\ \Theta\ a{\in}\{✗\}.\ Q\ ✗$, où $Q\ ✗$ dépend bien sûr de ce que l'on souhaite faire lorsque l'on attrape l'exception ✗.

Comme on souhaite avoir $Q :: 'a \Rightarrow 'a\ process$, il convient d'adapter très légèrement (remplacer Q par $Q\ a$) les définitions fournies dans [Ros10].

Définition 3. Voici les divergences et échecs de l'opérateur *Throw*.

$$\mathcal{D}\ (P\ \Theta\ a{\in}A.\ Q\ a) =$$
$$\{t\ @\ u \mid t \in \mathcal{D}\ P \wedge tF\ t \wedge set\ t \cap ev\ `\ A = \emptyset \wedge ftF\ u\} \cup$$
$$\{t\ @\ ev\ a \cdot u \mid t\ @\ [ev\ a] \in \mathcal{T}\ P \wedge set\ t \cap ev\ `\ A = \emptyset \wedge a \in A \wedge u \in \mathcal{D}\ (Q\ a)\}$$

$$\begin{array}{l}\mathcal{F}\ (P\ \Theta\ a{\in}A.\ Q\ a) = \\ \{(t,\ X) \mid (t,\ X) \in \mathcal{F}\ P \wedge set\ t \cap ev\ `\ A = \emptyset\} \cup \\ \{(t\ @\ u,\ X) \mid t \in \mathcal{D}\ P \wedge tF\ t \wedge set\ t \cap ev\ `\ A = \emptyset \wedge ftF\ u\} \cup \\ \{(t\ @\ ev\ a \cdot u, \\ \quad X) \mid t\ @\ [ev\ a] \in \mathcal{T}\ P \wedge set\ t \cap ev\ `\ A = \emptyset \wedge a \in A \wedge (u,\ X) \in \mathcal{F}\ (Q\ a)\}\end{array}$$

La première chose que l'on peut faire est de regarder ses traces (dans le modèle d'échec/divergence, ces dernières sont définies par $t \in \mathcal{T}\ P$ si et seulement si $(t,\ \emptyset) \in \mathcal{F}\ P$).

Corollaire 1. *Les traces de l'opérateur Throw s'expriment comme suit.*

$$\begin{array}{l}\mathcal{T}\ (P\ \Theta\ a{\in}A.\ Q\ a) = \\ \{t \in \mathcal{T}\ P \mid set\ t \cap ev\ `\ A = \emptyset\} \cup \\ \{t\ @\ u \mid t \in \mathcal{D}\ P \wedge tF\ t \wedge set\ t \cap ev\ `\ A = \emptyset \wedge ftF\ u\} \cup \\ \{t\ @\ ev\ a \cdot u \mid t\ @\ [ev\ a] \in \mathcal{T}\ P \wedge set\ t \cap ev\ `\ A = \emptyset \wedge a \in A \wedge u \in \mathcal{T}\ (Q\ a)\}\end{array}$$

Sans trop de difficulté, on vérifie que les échecs et divergences satisfont à l'invariant *is-process*, autrement dit que notre opérateur est bien défini. On vérifie aussi que *Throw* est monotone en ses deux variables pour les raffinements ($\sqsubseteq_{FD}$) et ($\sqsubseteq_{DT}$) ainsi que pour l'ordre de continuité ($\sqsubseteq$). Les preuves ne sont pas captivantes : elles consistent à déplier les définitions et vérifier que tout fonctionne. Le résultat énoncé ci-dessous est en revanche plus intéressant, nous allons nous y attarder.

Théorème 1. *L'opérateur Throw est continu. Plus formellement, on a le résultat suivant.*

$$[\![cont\ f;\ \forall\ a.\ cont\ (g\ a)]\!] \Longrightarrow cont\ (\lambda x.\ f\ x\ \Theta\ a{\in}A.\ g\ a\ x)$$

Esquisse de preuve : Par composition, il suffit de prouver que l'opérateur est continu selon sa variable de gauche, puis selon sa variable de droite. Intéressons-nous ici au premier cas. Grâce à la monotonie de *Throw* pour l'ordre de continuité ($\sqsubseteq$), il suffit de démontrer le lemme suivant, dont nous allons un peu détailler la preuve.

```
lemma cont-left-prem-Throw :
  fixes Y A Q assumes ‹chain Y›
  defines ‹imageSup ≡ (⨆ i. Y i) Θ a∈A. Q a› and ‹supImages ≡ (⨆ i. Y i Θ a∈A. Q a)›
  shows ‹imageSup = supImages›
proof (subst Process-eq-spec-optimized, safe)
  — 1 show ‹s ∈ 𝒟 imageSup ⟹ s ∈ 𝒟 supImages› for s by ...
next
  — 2 fix s assume ‹s ∈ 𝒟 supImages›
  define S where ‹S i ≡ {t. ∃ u. s = t @ u ∧ t ∈ 𝒟 (Y i) ∧ tF t ∧
                                  set t ∩ ev ` A = {} ∧ ftF u} ∪
                        {t. ∃ a u. s = t @ ev a # u ∧ t @ [ev a] ∈ 𝒯 (Y i) ∧ tF t ∧
                                  set t ∩ ev ` A = {} ∧ a ∈ A ∧ u ∈ 𝒟 (Q a)}› for i
  have ‹∀ i. S i ≠ {}› by ...
  moreover have ‹finite (S 0)› by ...
  moreover have ‹∀ i. S (Suc i) ⊆ S i› by ...
  ultimately have ‹(⋂ i. S i) ≠ {}› by (fact Inter-nonempty-finite-chained-sets)
  then obtain t where ‹∀ i. t ∈ S i› by ...
  then show ‹s ∈ 𝒟 imageSup› by ...
next
  — 3 show ‹(s, X) ∈ ℱ imageSup ⟹ (s, X) ∈ ℱ supImages› for s X by ...
next
  — 4 show ‹𝒟 imageSup = 𝒟 supImages ⟹ (s, X) ∈ ℱ supImages ⟹
            (s, X) ∈ ℱ imageSup› for s X by ...
qed
```

Les premier et troisième sous-objectifs sont directement résolus par **`sledgehammer`**, un outil faisant appel à des solveurs externes pour automatiquement reconstruire des preuves en Isabelle. Le quatrième est aussi relativement facile en jouant avec la monotonie de *Throw* pour ($\sqsubseteq$), la définition de ($\sqsubseteq$) et le fait que les divergences de *imageSup* et *supImages* sont égales. Finalement l'effort de preuve réside dans le deuxième, pour lequel une nouvelle méthode a été développée. Par définition des divergences du supremum d'une chaîne pour les processus (cf 2.3), l'hypothèse $s \in \mathcal{D}$ *supImages* se traduit par $\forall\, i.\ s \in \mathcal{D}\ (Y\ i)$.
Avec la définition de *Throw*, on a donc pour tout i une disjonction :

— il existe t, u vérifiant $s = t\ @\ u$, $t \in \mathcal{D}\ (Y\ i)$, $tF\ t$, $set\ t \cap ev\ `\ A = \emptyset$ et $ftF\ u$
— ou il existe t, a, u vérifiant $s = t\ @\ ev\ a \cdot u$, $t\ @\ [ev\ a] \in \mathcal{T}\ (Y\ i)$, $set\ t \cap ev\ `\ A = \emptyset$, $a \in A$ et $u \in \mathcal{D}\ (Q\ a)$.

L'intuition de la méthode que nous proposons se résume simplement : échanger les quantificateurs, c'est-à-dire passer de « pour tout i il existe ... tels que ... » à « il existe ... tels que pour tout i ... ». Pour ce faire, nous allons chercher à appliquer le lemme suivant.

$$[\![\forall\, i.\ S\ i \neq \emptyset;\ finite\ (S\ 0);\ \forall\, i.\ S\ (i + 1) \subseteq S\ i]\!] \Longrightarrow (\bigcap_i S\ i) \neq \emptyset$$

La preuve consiste alors à poser une suite d'ensembles $S :: nat \Rightarrow 'a\ trace\ set$ (judicieusement choisie à partir des divergences de *Throw* bien sûr) et montrer que le lemme est applicable (là encore, **`sledgehammer`** est très efficace). L'intersection des $S\ i$ est donc non vide, autrement dit on peut trouver une trace t qui appartient à chacun des $S\ i$, et par définition de ces ensembles on obtient conséquemment deux alternatives : $s = t\ @\ u$ ou $s = t\ @\ ev\ a \cdot u$, avec les contraintes sur t, a et u correspondantes. Le reste de la preuve est immédiat, la difficulté résidant essentiellement dans le fait d'obtenir t et u ou t, a et u qui ne dépendent plus de i.

Nous avons aussi appliqué avec succès cette méthode à *Interrupt* et à d'autres opérateurs déjà définis dans `HOL-CSP`. Pour la synchronisation par exemple, nous passons de 350 à 75 lignes de code tout en rendant la preuve beaucoup plus lisible.

3.2 Sémantique algébrique

En pratique, lors de la définition d'un processus, on ne spécifie pas directement ses échecs et divergences. On préfère travailler à plus haut niveau, sur l'algèbre de processus, en utilisant des opérateurs pour le décrire puis utiliser les lois algébriques (en plus des monotonies) pour prouver des propriétés sur notre système. Dériver ces règles pour *Throw* est donc essentiel ! Les résultats mentionnés ci-après sont démontrés en dépliant les définitions dénotationnelles pour prouver que les échecs et divergences des processus de gauche et droite sont égaux.

Proposition 1. *L'opérateur Throw vérifie les propriétés élémentaires ci-dessous.*

$$STOP\ \Theta\ a{\in}A.\ Q\ a = STOP \qquad SKIP\ \Theta\ a{\in}A.\ Q\ a = SKIP$$
$$\bot\ \Theta\ a{\in}A.\ Q\ a = \bot \qquad P\ \Theta\ a{\in}\emptyset.\ Q\ a = P$$

Quand l'alphabet ne contient pas d'exception, on établit la généralisation suivante.

$$A \cap events\text{-}of\ P = \emptyset \Longrightarrow P\ \Theta\ a{\in}A.\ Q\ a = P$$

Une autre propriété attendue des opérateurs CSP est la distributivité du non-déterminisme.

Proposition 2. *Le choix non-déterministe se distribue sur l'opérateur Throw.*

$$P \sqcap P'\ \Theta\ a{\in}A.\ Q\ a = (P\ \Theta\ a{\in}A.\ Q\ a) \sqcap (P'\ \Theta\ a{\in}A.\ Q\ a)$$
$$P\ \Theta\ a{\in}A.\ Q\ a \sqcap Q'\ a = (P\ \Theta\ a{\in}A.\ Q\ a) \sqcap (P\ \Theta\ a{\in}A.\ Q'\ a)$$

Enfin, pour chaque opérateur, on établit la loi de *pas* : son comportement un pas en avant, ou autrement dit comment se distribue le choix multi-préfixe déterministe. Ici, à chaque étape, on engage un événement puis si c'est une exception on passe à l'argument de droite, sinon on continue avec celui de gauche en conservant la possibilité d'en attraper plus tard.

Théorème 2. *L'opérateur Throw vérifie la loi de pas suivante.*

$$(\Box a{\in}A \rightarrow P\ a)\ \Theta\ b{\in}B.\ Q\ b = \Box a{\in}A \rightarrow (\mathit{if}\ a \in B\ \mathit{then}\ Q\ a\ \mathit{else}\ P\ a\ \Theta\ b{\in}B.\ Q\ b)$$

D'autres propriétés formellement prouvées sont à retrouver dans l'AFP.

4 L'opérateur Interrupt

4.1 Intuition et définition dénotationnelle

Là encore, le nom est assez explicite : on cherche à définir un opérateur binaire *Interrupt*, noté $P \triangle Q$. Il est censé se comporter d'abord comme P, sauf qu'à n'importe quel moment Q peut démarrer si l'environnement lui en offre la possibilité et P est alors abandonné.

Exemple 5 (Utilisation de l'opérateur *Interrupt*).
Considérons P une modélisation d'une interface graphique synchonisée avec un contexte C sur les événements de l'ensemble A. Le système global est donc $S \equiv C\ [\![A]\!]\ P$. On aimerait pouvoir ajouter la possibilité de fermer la fenêtre à n'importe quel moment si l'événement ✗ est proposé par le contexte (quand l'utilisateur clique sur la croix rouge). Sans l'opérateur *Interrupt*, cela demanderait de réécrire complètement l'implémentation de P pour proposer à chaque étape la possibilité de s'engager dans ✗ et, le cas échéant, de terminer. Avec, il suffit de considérer le système modifié $S' \equiv C\ [\![A]\!]\ P \triangle (✗ \rightarrow Q)$, où Q sera la démarche à suivre pour fermer la fenêtre (enregistrer les données avant de terminer par exemple).

Définition 4. Dans [Ros10], les définitions (naïves) suivantes sont proposées pour les divergences et échecs de l'opérateur *Interrupt*.

$$\mathcal{D}\ (P \triangle Q) = \mathcal{D}\ P \cup \{t\ @\ u \mid t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{D}\ Q\}$$

$$\begin{aligned}&\mathcal{F}\ (P \triangle Q) =\\ &\{(t, X) \mid (t, X) \in \mathcal{F}\ P \wedge ([], X) \in \mathcal{F}\ Q\} \cup\\ &\{(t\ @\ u, X) \mid t \in \mathcal{T}\ P \wedge tF\ t \wedge (u, X) \in \mathcal{F}\ Q \wedge u \neq []\} \cup\\ &\{(t, X) \mid t \in \mathcal{D}\ P\} \cup\\ &\{(t\ @\ u, X) \mid t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{D}\ Q\}\end{aligned}$$

Bien que raisonnables à première vue, les échecs transgressent l'invariant *is-process* (définition 1). En effet, avec l'aide de `sledgehammer`, on prouve très facilement qu'avec cette définition de *Interrupt*, pour e et P quelconques :

- le processus $e \rightarrow P \triangle SKIP$ ne vérifie pas la condition T6, la correction nécessite d'ajouter $\{(t, X - \{✓\}) \mid t \in \mathcal{T}\ P \wedge tF\ t \wedge [✓] \in \mathcal{T}\ Q\}$ aux échecs
- le processus $SKIP \triangle e \rightarrow P$ ne vérifie pas la condition T6, la correction nécessite d'ajouter $\{(t, X - \{✓\}) \mid t\ @\ [✓] \in \mathcal{T}\ P\}$ aux échecs
- le processus $SKIP \triangle e \rightarrow P$ ne vérifie pas la condition T5, la correction nécessite d'ajouter $\{(t\ @\ [✓], X) \mid t\ @\ [✓] \in \mathcal{T}\ P\}$ aux échecs.

Définition 5. Finalement, la spécification corrigée en Isabelle se présente comme suit.

```
lift-definition Interrupt :: ⟨['a process, 'a process] ⇒ 'a process⟩ (infixl ⟨△⟩ 75)
  is ⟨λP Q. ({(t @ [✓], X)| t X. t @ [✓] ∈ T P} ∪
             {(t, X - {✓})| t X. t @ [✓] ∈ T P} ∪
             {(t, X) ∈ F P. tF t ∧ ([], X) ∈ F Q} ∪
             {(t @ u, X)| t u X. t ∈ T P ∧ tF t ∧ (u, X) ∈ F Q ∧ u ≠ []} ∪
             {(t, X - {✓})| t X. t ∈ T P ∧ tF t ∧ [✓] ∈ T Q} ∪
             {(t, X). t ∈ D P} ∪
             {(t @ u, X)| t u X. t ∈ T P ∧ tF t ∧ u ∈ D Q},
             D P ∪ {t @ u| t u. t ∈ T P ∧ tF t ∧ u ∈ D Q})⟩
```

```
proof −
  show ‹ ?thesis P Q› (is ‹is-process ( ?fail, ?div)›) for P Q
  proof (unfold is-process-def, intro conjI allI impI)
    — 1 show ‹([], {}) ∈ ?fail› by ...
  next
    — 2 show ‹(t, X) ∈ ?fail ⟹ ftF t› for t X by ...
  next
    — 3 show ‹(t @ u, {}) ∈ ?fail ⟹ (t, {}) ∈ ?fail› for t u by ...
  next
    — 4 show ‹(t, Y) ∈ ?fail ∧ X ⊆ Y ⟹ (t, X) ∈ ?fail› for t X Y by ...
  next
    — 5 show ‹(t, X) ∈ ?fail ∧ (∀ e. e ∈ Y ⟶ (t @ [e], {}) ∉ ?fail) ⟹
                (t, X ∪ Y) ∈ ?fail› for t X Y by ...
  next
    — 6 show ‹(t @ [✓], {}) ∈ ?fail ⟹ (t, X − {✓}) ∈ ?fail› for t X by ...
  next
    — 7 show ‹t ∈ ?div ∧ tF t ∧ ftF u ⟹ t @ u ∈ ?div› for t u by ...
  next
    — 8 show ‹t ∈ ?div ⟹ (t, X) ∈ ?fail› for t X by ...
  next
    — 9 show ‹t @ [✓] ∈ ?div ⟹ t ∈ ?div› for t by ...
  qed
qed
```

On voit ici comment s'utilise `lift-definition` : *Interrupt* est défini en fonction de ses échecs et divergences, donc au niveau $'a\ process_0$, et une obligation de preuve est alors générée pour s'assurer que l'invariant *is-process* est bien satisfait. Notons au passage la facilité d'utilisation d'Isabelle, depuis le filtrage par motif (*pattern matching*) avec des variables schématiques pour abréger les échecs et divergences jusqu'à la lisibilité de la structure de la preuve, même pour quelqu'un qui ne serait pas familier avec cet assistant de preuve. En ce qui nous concerne, quasiment tous les sous-objectifs sont résolus par disjonction de cas puis dépliage des définitions, seul le troisième (condition T3) nécessite une induction. Finalement notre version corrigée de l'opérateur *Interrupt* est bien définie, et on peut argumenter qu'elle est optimale car les ajouts apportés à la définition naïve de [Ros10] se limitent au strict nécessaire.

On établit aussi la monotonie de l'opérateur *Interrupt* en ses deux variables pour les raffinements ($\sqsubseteq_{FD}$) et ($\sqsubseteq_{DT}$) ainsi que pour l'ordre de continuité ($\sqsubseteq$). Avec la même méthode que pour *Throw*, on prouve sa continuité.

Pour finir, avant de quitter la sémantique dénotationnelle, mentionnons ses traces.

Corollaire 2. *Les traces de l'opérateur Interrupt s'expriment comme suit.*

$$\mathcal{T}\ (P \triangle Q) = \mathcal{T}\ P \cup \{t\ @\ u \mid t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{T}\ Q\}$$

4.2 Sémantique algébrique

En première propriété, on prouve que l'opérateur binaire *Interrupt* est associatif (comme $P \Box Q$, $P \sqcap Q$, $P \llbracket A \rrbracket Q$ et $P ; Q$). On établit aussi que *STOP* est élément neutre. Rien d'étonnant : puisque ce dernier ne peut s'engager dans aucun événement, il ne pourra jamais interrompre et ne fera jamais rien avant d'être interrompu. Sans surprise non plus, $\bot$ est élément absorbant : comme pour les autres opérateurs, on ne revient pas de s'être engagé sur une divergence. Enfin le non-déterminisme se distribue.

Sa loi de pas se comprend aisément : à chaque étape, le processus $P \triangle Q$ a le choix entre s'engager dans Q et abandonner P ou faire un pas supplémentaire dans P en gardant l'option d'interruption, ce choix étant dépendant du contexte (externe donc).

Théorème 3. *L'opérateur Interrupt vérifie la loi de pas suivante.*

$$(\Box a{\in}A \rightarrow P\ a) \triangle Q = Q \Box (\Box a{\in}A \rightarrow P\ a \triangle Q)$$

Encore une fois, d'autres propriétés formellement prouvées sont à retrouver dans l'AFP.

5 Travaux connexes et perspectives

Bien sûr les opérateurs *Throw* et *Interrupt* ne concernent pas que CSP, nous nous limitons cependant à ce cadre pour rester succinct. Evoquons d'abord rapidement le model checking : l'outil référence pour CSP, FDR [fdr19], supporte les deux tandis que PAT [SLDP09] ne supporte que l'interruption alphabétisée.

Il serait cependant plus juste de mettre en relation notre travail avec des implémentations similaires, à savoir sur assistant de preuve (et si possible vérifiées). Nous en avons référencé trois autres : CSP-Prover [IR10] (aussi en Isabelle), CSP-Coq [dSCdF20] et CSP-Agda [IS16]. Aucune d'elle ne supporte *Interrupt* ou *Throw* pour l'instant.

Si la littérature n'est pas avare en définitions et lois, le travail présenté ici est à notre connaissance la première vérification formelle des lois algébriques de ces deux opérateurs à partir de leurs définitions dénotationnelles. Pour *Interrupt* il s'agit même de la première définition dénotationnelle correcte : celle donnée par la référence du domaine [Ros10] transgresse l'invariant *is-process*.

Le prochain objectif à court terme est de formaliser et vérifier de gros exemples (si possible hors de portée des model checkers) faisant intervenir ces opérateurs. Plus généralement, les perspectives de recherche pour `Isabelle/HOL-CSP` ne manquent pas. Citons-en trois.

— Connecter avec FDR dans le cas de modèles finis : « finitiser » le type *'a*, traduire le processus, le donner à FDR, récupérer la sortie et la reconstruire dans `HOL-CSP`.
— Travailler sur l'exécutabilité des processus pour pouvoir générer des scénarios.
— Créer une bibliothèque rassemblant des briques élémentaires réutilisables modulairement pour qu'un utilisateur puisse construire son processus à la carte en bénéficiant de propriétés déjà établies.

6 Conclusion

Après un résumé de CSP et de la formalisation `HOL-CSP` en `Isabelle/HOL`, nous avons pour *Throw* et *Interrupt* présenté et justifié les définitions dénotationnelles retenues puis exposé les monotonies et lois algébriques dérivées. Ces lois formellement démontrées sont bien celles attendues dans la littérature et nécessitent environ deux mille lignes de preuve. Pour en donner quelques bribes, nous nous sommes attardés sur le concept de continuité en développant une nouvelle méthode que nous avons par la suite aussi appliquée à des opérateurs préexistants de `HOL-CSP` afin d'en améliorer la preuve.

En terme d'applications, *Interrupt* et *Throw* sont dores et déjà utilisés dans une dérivation formelle de la sémantique opérationnelle à partir de l'algébrique (donc de la dénotationnelle sous-jacente) [BW24]. Qui plus est, en plus de l'intérêt pratique d'ajouter ces deux opérateurs à la formalisation, ce travail illustre l'importance du rôle qu'ont à jouer les assistants de preuve dans la correction des incomplétudes/erreurs des modèles. En effet la définition donnée par Roscoe dans [Ros10] transgresse l'invariant *is-process* qu'il donne dans le même livre. Il est d'ailleurs intéressant de mentionner que la première version de `Isabelle/HOL-CSP` [TW97] visait déjà à corriger une inconsistance de la théorie dûe à la gestion de ✓, cette fois-ci dans la composition séquentielle.

Références

[BHR84] S. D. Brookes, C. A. R. Hoare et A. W. Roscoe : A theory of communicating sequential processes. *J. ACM*, 31(3) :560–599, 1984.

[BR85] S. D. Brookes et A. W. Roscoe : An improved failures model for communicating sequential processes. *In* Stephen D. Brookes, Andrew William Roscoe et Glynn Winskel, éditeurs : *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer.

[BTW23] Benoît Ballenghien, Safouan Taha et Burkhart Wolff : HOL-CSPM - architectural operators for HOL-CSP. *Archive of Formal Proofs*, 2023, 2023.

[BW23] Benoît Ballenghien et Burkhart Wolff : Operational semantics formally proven in hol-csp. *Archive of Formal Proofs*, December 2023.

[BW24] Benoit Ballenghien et Burkhart Wolff : An Operational Semantics in Isabelle/HOL-CSP. *In* Yves Bertot, Temur Kutsia et Michael Norrish, éditeurs : *15th International Conference on Interactive Theorem Proving, ITP 2024*, volume 309 de *LIPIcs*, pages 29:1–29:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[BW25] Benoît Ballenghien et Burkhart Wolff : A theory of proc-omata—and proof methods for process architectures. *In* Chutiporn Anutariya et Marcello M. Bonsangue, éditeurs : *Theoretical Aspects of Computing – ICTAC 2024*, pages 272–289, Cham, 2025. Springer Nature Switzerland.

[dSCdF20] Carlos Alberto da Silva Carvalho de Freitas : A theory for communicating, sequential processes in coq. 2020.

[fdr19] FDR4 - The CSP Refinement Checker. https://www.cs.ox.ac.uk/projects/fdr/, 2019.

[Hoa85] C. A. R. Hoare : *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[IR10] Yoshinao Isobe et Markus Roggenbach : Csp-prover : a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1) :32–39, 2010.

[IS16] Bashar Igried et Anton Setzer : Programming with monadic csp-style processes in dependent type theory. *In Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 2838, New York, NY, USA, 2016. Association for Computing Machinery.

[Noc16] Pasquale Noce : Conservation of CSP noninterference security under sequential composition. *Archive of Formal Proofs*, 2016.

[Ros92] A. W. Roscoe : An alternative order for the failures model. *J. Log. Comput.*, 2 :557–577, 1992.

[Ros97] A.W. Roscoe : *Theory and Practice of Concurrency*. Prentice Hall, 1997.

[Ros10] A.W. Roscoe : *Understanding Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg, 1st édition, 2010.

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong et Jun Pang : Pat : Towards flexible verification under fairness. volume 5643 de *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[TW97] Haykal Tej et Burkhart Wolff : A corrected failure divergence model for CSP in Isabelle/HOL. *In* John S. Fitzgerald, Cliff B. Jones et Peter Lucas, éditeurs : *Formal Methods Europe (FME)*, volume 1313 de *LNCS*, pages 318–337. Springer, 1997.

[TWY20] Safouan Taha, Burkhart Wolff et Lina Ye : The HOL-CSP refinement toolkit. *Arch. Formal Proofs*, 2020, 2020.

[TYW19] Safouan Taha, Lina Ye et Burkhart Wolff : HOL-CSP Version 2.0. *Archive of Formal Proofs*, avril 2019.