RAPPORT DE STAGE

Connecting IMP Concur with Isabelle/HOL-CSP

Juillet 2025

Zineddine KERMADJ
LDD3 MI
Mention Magistère d'informatique
Université Paris-Saclay

Burkhart WOLFF
Encadrant de stage
Laboratoire de Méthodes Formelles (LMF)
Université Paris-Saclay

Contents

Résumé

Ce rapport présente les travaux réalisés au cours d'un stage de 10 semaines au Laboratoire de Méthodes Formelles (LMF) de l'Université Paris-Saclay. L'objectif principal de ce stage était de concevoir une interface simple et efficace reliant IMP Concur, un langage de programmation concurrent dérivé de IMP, avec Isabelle/HOL-CSP, un environnement de preuve puissant basé sur Isabelle/HOL et l'algèbre des processus Communicating Sequential Processes (CSP).

Isabelle/HOL-CSP permet de modéliser et de raisonner sur des processus complexes, incluant notamment des structures d'événements infinies, la gestion du temps dense, des états physiques et des structures algébriques. Grâce à cette base solide, l'intégration d'IMP Concur devait offrir une sémantique formelle rigoureuse à ce langage concurrent, facilitant ainsi la vérification et la preuve de propriétés pour des architectures de processus.

Le stage a consisté à étudier les fondements théoriques de CSP et HOL-CSP, à apprendre l'API fonctionnelle du système Isabelle, puis à concevoir et implémenter des schémas de traduction d'IMP Concur vers HOL-CSP et vers un formalisme orienté automate appelé ProcOmata, ce dernier étant particulièrement adapté aux méthodes de preuve automatisées.

Ce travail a permis de développer des compétences avancées en programmation fonctionnelle, en lambda-calcul typé, ainsi qu'une compréhension approfondie des algèbres de processus et de la théorie de la concurrence, tout en contribuant à la construction d'outils formels pour la vérification de programmes concurrents.

Les contributions principales de ce stage comprennent la conception d'une architecture modulaire, le développement d'un traducteur unidirectionnel entre IMP Concur et HOL-CSP, ainsi que la validation expérimentale sur des cas d'étude représentatifs. Ces résultats démontrent la faisabilité de l'approche proposée et ouvrent des perspectives prometteuses pour la vérification formelle automatisée de programmes concurrents.

Pour cela, j'ai développé plusieurs composants essentiels : un analyseur syntaxique (parser), un vérificateur de contexte, un contrôleur de sémantique statique, ainsi qu'un encodeur sémantique produisant des spécifications au format CSP. Ces éléments garantissent la cohérence et la correction des programmes IMP Concur avant leur traduction, assurant ainsi une base solide pour l'analyse formelle dans Isabelle/HOL-CSP.

Mots-clés : Vérification formelle, IMP Concur, Isabelle/HOL-CSP, programmes concurrents, traduction automatique

Remerciements

Je tiens à exprimer ma sincère gratitude à Dr. Burkhart Wolff pour son encadrement expert et ses conseils précieux tout au long de ce stage. Je remercie également M. Ballenghien pour son suivi attentif et ses orientations académiques.

Mes remerciements s'adressent aussi à l'ensemble de l'équipe du Laboratoire de Méthodes Formelles pour leur accueil chaleureux et les nombreuses discussions enrichissantes qui ont contribué à l'aboutissement de ce travail.

Enfin, je remercie l'Université Paris-Saclay pour m'avoir offert cette opportunité de stage dans un environnement de recherche d'excellence.

1 Introduction

Ce rapport présente les travaux réalisés durant mon stage de 10 semaines au Laboratoire de Méthodes Formelles (LMF) de l'Université Paris-Saclay, sous la direction de Burkhart Wolff. L'objectif principal de ce stage était de développer une connexion entre IMP Concur, un langage de programmation concurrent dérivé du langage IMP, et Isabelle/HOL-CSP, un environnement de preuve formelle puissant basé sur Isabelle/HOL et l'algèbre des processus Communicating Sequential Processes (CSP).

La vérification formelle est devenue un enjeu majeur dans le développement de systèmes critiques où la sûreté et la fiabilité sont indispensables. Les programmes concurrents posent des défis particuliers à cause de leur complexité intrinsèque, notamment les problèmes de synchronisation et les interactions non déterministes. IMP Concur et Isabelle/HOL-CSP sont deux outils complémentaires qui, une fois reliés, permettent de combiner la simplicité de modélisation d'IMP Concur avec la rigueur et la puissance des techniques de preuve d'Isabelle/HOL-CSP.

Cependant, ces outils fonctionnent généralement de façon isolée, limitant ainsi leur potentiel. La problématique du stage était donc d'élaborer une interface efficace permettant d'exploiter au mieux les forces de chacun, en fournissant une sémantique formelle rigoureuse à IMP Concur via HOL-CSP, et ainsi faciliter la vérification automatique de programmes concurrents.

Les objectifs du stage ont été les suivants :

- Étudier les fondements théoriques et les architectures d'IMP Concur et d'Isabelle/HOL-CSP
- Concevoir une architecture modulaire pour la traduction unidirectionnelle d'IMP Concur vers HOL-CSP
- Implémenter un analyseur syntaxique (parser), un vérificateur de contexte, un contrôleur de sémantique statique et un encodeur sémantique générant des spécifications CSP
- Valider l'approche à travers des cas d'étude représentatifs

2 État de l'art

2.1 IMP Concur

IMP Concur est une extension concurrente du langage IMP, un langage de programmation impératif simple et bien connu. Il intègre des primitives spécifiques à la programmation concurrente, telles que les mécanismes de verrouillage (lock/unlock) et les communications synchrones (send/receive). Sa syntaxe abstraite et concrète a été formalisée dans Isabelle/HOL, ce qui permet de raisonner précisément sur les programmes écrits en IMP Concur. Ce langage sert ainsi de front-end simple pour modéliser des processus concurrents avant leur traduction vers des modèles formels.

2.2 Isabelle/HOL-CSP

Isabelle/HOL-CSP est une extension du système de preuve Isabelle/HOL, intégrant le calcul des processus Communicating Sequential Processes (CSP), une théorie développée par Tony Hoare et Bill Roscoe dans les années 1980. HOL-CSP permet de modéliser et de raisonner sur des processus complexes, incluant notamment des structures d'événements infinies, la gestion de temps dense, des états physiques et des structures algébriques. Cette richesse permet d'effectuer des preuves générales sur des architectures de processus et des propriétés abstraites de patrons de processus.

Une sémantique rigoureuse pour IMP Concur au travers de HOL-CSP permet d'utiliser cette infrastructure pour vérifier formellement des programmes concurrents avec une grande expressivité et puissance.

3 Méthodologie et Approche

Pour mener à bien ce projet, ma première étape a été une immersion approfondie dans la documentation et les articles scientifiques relatifs à Isabelle/HOL et à l'algèbre des processus CSP. Cette phase d'étude intensive m'a permis de mieux comprendre les capacités du système Isabelle, ses méthodes de preuve, ainsi que les fondements théoriques nécessaires à la modélisation et à la vérification formelle.

Parallèlement, afin de me familiariser concrètement avec Isabelle et son langage de preuve, j'ai réalisé plusieurs exercices de programmation formelle, bien qu'indépendants du sujet principal. Parmi ceux-ci

figurent l'implémentation des nombres de Church, l'étude de la confluence du lambda-calcul, ainsi que la génération de documents PDF via Isabelle. Ces travaux m'ont permis de gagner en aisance avec les outils et les techniques utilisés, renforçant ainsi mes compétences en programmation fonctionnelle et en logique formelle.

Cette phase préparatoire a été essentielle avant d'aborder le développement des composants spécifiques au projet, comme le parser, les contrôles sémantiques et l'encodeur CSP.

4 Développement et Implémentation

Après cette phase d'appropriation théorique, j'ai commencé le développement des différents composants du projet. La première étape a été la réalisation du parser pour IMP Concur. Ce travail a nécessité une adaptation à une bibliothèque peu documentée, ce qui a demandé patience et exploration pour comprendre son fonctionnement.

Ensuite, j'ai implémenté le vérificateur de contexte (context checker), puis le contrôleur de sémantique statique (semantic checker), garantissant la cohérence et la validité des programmes avant leur traduction.

Bien que la génération de code ne fasse pas partie des objectifs initiaux du stage, je me suis lancé dans cette étape, cherchant à produire directement des spécifications en CSP. Cette phase s'est révélée plus complexe que prévu : mes premières tentatives manquaient de rigueur, ce qui a conduit à des résultats insatisfaisants. Après une période de réflexion, j'ai recentré mes efforts pour développer un encodeur sémantique adapté, capable de produire des spécifications conformes et exploitables dans Isabelle/HOL-CSP.

Cette expérience m'a permis de mieux cerner les défis liés à la traduction formelle et à la génération de code dans un contexte complexe, tout en consolidant mes compétences techniques.

5 Implémentation

Le traducteur convertit les constructions IMP Concur en théories Isabelle/HOL-CSP équivalentes.

Listing 1: Code avant encodage

```
SYSTEM WellTypedSys
    globals
        x:<int> = <3 :: int>
        var1:<int> = <4::int>
        var4:<int> = <4::int>
    locks
        l:<()>
        l2:<()>
        l3:<()>

    thread t1 :
        any
            y : <int> = <36 :: int>
            var2 : <int>
        actions
            SKIP;
            LOCK l;
            y <- var1;
            UNLOCK l;
            IF <var1<(3:: int)> THEN
                WHILE <var1<(3:: int)>
                     DO
                    LOCK l2;
                    SKIP;
                    var2 = <var2 + 1
                        ::int>;
                    var1 -> <var2 ::
                        int>;
                    UNLOCK l2;
                DONE
            ELSE
                SKIP;
            DONE
            y  = <y+2 :: int>;
            y  = <y+2 :: int>;
            var1 -> <y :: int>;

    thread t2 :
        any
            var2:<int> = <(4+6) :: int
                >
            y : <int> = <28 :: int>
        actions
            SKIP;
            LOCK l;
            UNLOCK l;
            IF <3 = (5:: int)> THEN
                WHILE <x -(8 ::int) <
                    x > DO
                    LOCK l;
                    UNLOCK l;
                DONE
            ...

    thread t3:
        any
            test :int = <-3 ::int>
            var_local :int = <4 :: int
                >
        actions
            var_local = <(4+42) :: int
                >;
             ...
end;
```

Listing 2: Définition post encodage

```
MultiInter (mset [''x'', ''var1'', ''
    var4''])
    (GLOBALVARS
        ((λ_. 0)
         (''x'' := 3,
          ''var1'' := 4,
          ''var4'' := 4))) ||
    (Sem t1_cmd ||
     MultiInter (mset [''y'', ''var2
         ''])
        (LOCALVARS
            ((λ_. 0)
             (''y'' := 36))) ||
     (Sem t2_cmd ||
      MultiInter (mset [''var2'', ''y
          ''])
         (LOCALVARS
             ((λ_. 0)
              (''y'' := 28,
               ''var2'' := 4 + 6))) ||
      Sem t3_cmd ||
      MultiInter (mset [''test'', ''
          var_local''])
         (LOCALVARS
             ((λ_. 0)
              (''test'' := - 3,
               ''var_local'' := 4)))))
                  ||
    MultiInter (mset [1, 2, 3])
       SEMAPHORES
```

6 Architecture du Traducteur

6.1 Analyse syntaxique (parsing)

Afin de permettre l'interprétation et la traduction des programmes écrits dans le langage IMP Concur, un analyseur syntaxique (ou *parser*) a été développé. Celui-ci lit un programme textuel et le transforme en une structure de données interne appelée *arbre syntaxique abstrait* (AST), utilisée par la suite pour la vérification et la génération de code Isabelle/HOL-CSP.

Le langage repose sur une syntaxe structurée, et le parser est capable de reconnaître les éléments suivants :

- les déclarations de variables globales, éventuellement initialisées
- les verrous (locks), utilisés pour la synchronisation
- les threads, chacun pouvant déclarer des variables locales et contenir une séquence d'actions
- les instructions élémentaires : `SKIP`, `LOCK`, `UNLOCK`, assignations, envois (`->`) et réceptions (`<-`) de messages
- les structures de contrôle : conditions (`IF ... THEN ... ELSE ... DONE`) et boucles (`WHILE ... DO ... DONE`)

L'analyse est modulaire : chaque construction est associée à un petit parser indépendant, et ces parsers sont combinés pour former des blocs plus complexes. Cette approche rend l'ensemble facilement maintenable et extensible.

Un programme typique, bien formé, s'écrit de la manière suivante :

```
SYSTEM WellParsedSys
  globals var1:<int> = <initialisation::int>
          var2:<int>
          ...
          varN:<int> = <initialisation::int>
  locks   l:<()>
          l2:<()>
          ...
          lN:<()>
  thread t1 :
         any lvar1:<int> = <initialisation::int>
             lvar2:<int>
             ...
             lvarN:<int>
         actions
         {Suite d'actions}
  thread t2 :
         ...
end;
```

Chaque thread débute par le mot-clé `thread`, peut déclarer des variables locales via le mot-clé `any`, puis énumère une suite d'actions séparées par des points-virgules. Les blocs de contrôle sont toujours délimités par le mot-clé `DONE`, garantissant une terminaison explicite.

Le parser a été écrit à l'aide de la bibliothèque `Parse` fournie par Isabelle. Celle-ci repose sur des combinateurs comme `|--`, `--|` et `»`, qui facilitent la composition des règles de syntaxe tout en rendant l'analyse syntaxique plus lisible et modulaire.

Les actions du langage sont représentées par un type de données récursif, ce qui permet de modéliser facilement des structures imbriquées. Par exemple, une assignation est encodée via le constructeur `Assign`, une réception par `Receive`, et une boucle par `While`.

L'approche adoptée rend le langage évolutif : l'ajout d'une nouvelle instruction se fait simplement en ajoutant un nouveau constructeur dans le type des actions, accompagné d'un parser associé.

Enfin, l'utilisation d'une bibliothèque interne peu documentée a demandé un effort d'adaptation important, notamment pour comprendre le fonctionnement des combinateurs et le format attendu des productions syntaxiques.

6.2 Vérification du contexte (*Context Checker*)

Une fois l'analyse syntaxique effectuée, une phase essentielle consiste à valider le bon usage des identifiants, des types et des déclarations dans le programme. Cette étape de vérification contextuelle s'assure que l'ensemble des entités (variables, verrous, threads) respecte les règles de portée, de typage et de déclaration imposées par le langage IMP Concur. Elle est entièrement implémentée en ML dans l'environnement d'Isabelle/HOL.

6.2.1 Représentation intermédiaire

Avant la traduction finale en logique HOL ou en CSP, le programme est converti dans une forme intermédiaire plus structurée. Les instructions élémentaires d'un thread sont représentées à l'aide du type récursif `a_term`, qui distingue plusieurs formes d'actions :

- des instructions simples comme `SkipA`, `AssignA`, `LockA`, `UnlockA`, `SendA` ou `ReceiveA`
- des structures de contrôle comme `IfelseA` ou `WhileA`
- la composition séquentielle via `SeqA`

Chaque thread est représenté par une structure `thread_absy` contenant :

- un nom de thread
- une liste de déclarations locales (avec types et initialisations optionnelles)
- la liste des actions
- une table des symboles locale associant des noms à leurs éventuelles valeurs

L'ensemble du programme est encapsulé dans une structure `absy`, qui regroupe : le nom du système, les variables globales, les verrous, les threads, ainsi que la table des symboles globale.

6.2.2 Lecture des expressions et vérification des types

Pour transformer les chaînes de caractères issues du parsing en termes internes d'Isabelle, on utilise la fonction `read_term_err`. Celle-ci lit un terme tout en signalant clairement, en cas d'erreur, la position exacte dans le code source. Une variante `get_term_and_vars` permet en outre d'extraire les variables libres apparaissant dans un terme.

Lors de la lecture d'une déclaration, si une initialisation est présente, son type est inféré et comparé au type explicitement indiqué. Une erreur claire et informative est générée si une incohérence est détectée (typage erroné ou absence de déclaration).

6.2.3 Résolution des identifiants

La fonction `resolve_var` est responsable de la résolution des noms de variables. Elle cherche d'abord dans la table locale puis, en fonction d'un paramètre `prefer_global`, dans la table globale. Une version stricte, `resolve_var_strict`, lève une exception si l'identifiant n'est trouvé dans aucune des portées, assurant ainsi qu'aucune variable non déclarée n'est utilisée dans le code.

6.2.4 Vérification des threads

La fonction `check_thread` applique les vérifications sur chacun des threads. Elle s'assure notamment :

- que toutes les variables locales sont bien typées et déclarées avant usage
- que toutes les actions manipulant des identifiants (assignation, envoi, réception) utilisent des noms valides dans la portée adéquate
- que les structures de contrôle contiennent des branches correctement formées

Chaque action est analysée individuellement. Par exemple, une instruction d'assignation est rejetée si la variable à gauche de l'affectation n'est pas déclarée. De même, une instruction de réception nécessite que la variable de réception soit déclarée localement, et la source globalement.

Une fois la vérification terminée, un thread est reconstruit sous forme d'un `thread_absy`, enrichi des déclarations locales résolues et d'une table des symboles locale.

6.2.5 Vérification du programme global

La fonction `context_check` orchestre la vérification contextuelle de l'ensemble du programme :

- **Déclarations globales** : chaque variable est analysée et typée, les constantes sont enregistrées dans le contexte Isabelle
- **Déclarations de verrous** : elles sont transformées en constantes unaires (représentant des événements) typées correctement
- **Déclarations de threads** : chaque thread est analysé via `check_thread`, en héritant de la table des symboles globale

Une fois cette vérification réussie, la représentation intermédiaire complète du programme est stockée dans une référence mutable `SPY`, pour être réutilisée lors des étapes ultérieures (traduction en logique, génération de code CSP, etc.).

6.2.6 Rôle et importance de cette phase

Le *context checker* assure une détection précoce des erreurs statiques : identifiants non déclarés, conflits de noms, incohérences de types, etc. Il repose sur une gestion fine des tables de symboles, mises à jour dynamiquement lors de la traversée de l'AST. Cette vérification contextuelle garantit que le programme est bien formé avant toute sémantique formelle ou exécution symbolique.

6.3 Contrôleur de sémantique statique (Semantic Checker)

Après la validation contextuelle, une phase cruciale de vérification sémantique analyse le comportement dynamique du programme concurrent. Cette étape, entièrement implémentée en ML dans l'environnement d'Isabelle/HOL, détecte les erreurs potentielles liées à la concurrence et garantit la cohérence sémantique du programme IMP Concur.

6.3.1 Évaluation des expressions et états

La vérification sémantique s'appuie sur un système d'évaluation des expressions Isabelle/HOL dans des états concrets. Un état est représenté par une fonction `state = string -> int` qui associe à chaque nom de variable sa valeur entière.

L'évaluation des expressions arithmétiques se fait via la fonction `eval_term_to_int`, qui traverse récursivement les termes Isabelle en gérant les opérations de base (addition, soustraction, multiplication) et les variables libres. Une fonction similaire `eval_term_to_bool` traite les expressions booléennes en supportant les connecteurs logiques et les opérateurs de comparaison.

```
fun eval_term_to_int (thy : theory) (term : term) (state : state) : int =
  let
    fun eval t = case t of
        Const (@{const_name "Groups.plus_class.plus"}, _) $ t1 $ t2 =>
          (eval t1) + (eval t2)
      | Free (name, _) => safe_lookup name state
      | _ => (* gestion des autres cas *)
  in eval term end
```

6.3.2 Conversion vers une représentation exécutable

Le code ML transforme l'AST en une représentation exécutable via le type `com`, qui encode les instructions élémentaires :

- `SKIP` pour l'instruction vide
- `Assign` pour l'affectation locale
- `Store` et `Load` pour les accès aux variables partagées
- `Lock` et `Unlock` pour la synchronisation
- `Seq`, `Cond`, `While` pour les structures de contrôle

La fonction `convert_to_com` effectue cette traduction en distinguant automatiquement les variables locales des variables globales grâce aux tables de symboles construites lors de la phase contextuelle.

6.3.3 Sémantique à événements CSP

L'implémentation adopte une approche par événements inspirée du formalisme CSP (Communicating Sequential Processes). Chaque opération génère un événement typé :

- `LockEvent` et `UnlockEvent` pour la synchronisation
- `ReadEvent` et `UpdateEvent` pour les accès mémoire
- `AssignEvent` pour les affectations locales

La sémantique CPS (Continuation-Passing Style) permet de capturer précisément l'ordre d'exécution et les dépendances entre opérations :

```
fun sem_cps machine (Lock id) cont =
  (fn state => LockEvent (id, get_lock_id id machine) :: cont state)
```

6.3.4 Détection statique des problèmes de concurrence

Analyse des interblocages (Deadlocks) La fonction `detect_potential_deadlock` analyse les dépendances entre verrous utilisés par différents threads. Elle identifie les situations où plusieurs threads acquièrent les mêmes verrous, créant un risque d'interblocage mutuel.

Détection des conditions de course (Race Conditions) L'analyse `detect_race_conditions` examine les accès concurrents aux variables partagées. Elle repère les situations où plusieurs threads accèdent à la même variable globale avec au moins une opération d'écriture, signalant un risque de corruption de données.

Vérification de l'ownership des verrous La fonction `check_lock_ownership` vérifie statiquement que chaque thread respecte la discipline d'acquisition et de libération des verrous. Elle détecte les tentatives de libération d'un verrou non possédé ou les acquisitions multiples du même verrou.

6.3.5 Vérification de cohérence des types

La vérification sémantique inclut une phase de contrôle de types approfondie via `type_check_thread`. Cette fonction :

- Vérifie la cohérence entre les types déclarés et les types inférés des expressions
- Contrôle la compatibilité des types dans les opérations d'affectation et de communication
- Valide que les conditions des structures de contrôle sont bien de type booléen

Les erreurs de typage sont rapportées avec des messages précis indiquant la nature du conflit et la position dans le code source.

6.3.6 Simulation d'exécution

Pour chaque thread, une simulation d'exécution est réalisée via `sem_step`. Cette fonction :

- Construit un état initial basé sur les déclarations locales et globales
- Exécute symboliquement le code du thread
- Génère une trace d'événements CSP
- Détecte les erreurs d'exécution (variables non déclarées, divisions par zéro, etc.)

6.3.7 Vérification globale du système

La fonction `semantic_check` orchestre l'ensemble des vérifications :

- Validation de la cohérence des déclarations de verrous et variables
- Analyse croisée des interactions entre threads
- Génération d'un rapport détaillé des problèmes détectés
- Construction d'un modèle sémantique prêt pour la traduction formelle

6.3.8 Importance de cette phase

La vérification sémantique constitue une étape critique qui garantit la robustesse du programme concurrent avant sa traduction en logique HOL ou CSP. Elle permet :

- La détection précoce des erreurs de concurrence difficiles à identifier lors des tests
- La génération d'un modèle sémantique fiable pour les analyses formelles ultérieures
- La production de diagnostics précis pour guider le développeur dans la correction des problèmes

Cette phase assure ainsi la transition entre la représentation syntaxique validée et le modèle sémantique formel, constituant le fondement des preuves de correction et des analyses de propriétés concurrentes.

6.4 Encodage Sémantique

6.4.1 Architecture du module d'encodage

L'encodage sémantique constitue une étape fondamentale dans la chaîne de traduction d'IMP Concur vers le formalisme Isabelle/HOL. Cette phase transforme l'AST validé en représentations exécutables, permettant l'analyse formelle et la vérification de propriétés concurrentes.

Le module d'encodage s'appuie sur une architecture ML native intégrée à Isabelle/HOL, garantissant une cohérence parfaite entre la représentation syntaxique et la sémantique formelle. Cette approche évite les problèmes de désynchronisation entre différents formalismes et assure la fiabilité des traductions.

6.4.2 Transformation des actions en commandos logiques

Conversion des actions abstraites La fonction `quote_com` effectue la traduction systématique des actions abstraites (type `a_term`) vers des termes logiques de type `com`. Cette transformation respecte la hiérarchie sémantique établie dans la théorie Isabelle/HOL-CSP :

```
fun quote_com _ SkipA = @{term "SKIP :: com"}
  | quote_com _ (AssignA (b, e)) =
    let
      val var_str = HOLogic.mk_string (Binding.name_of b)
      val expr = subst_with_sigma e
      val lam = make_lambda _sigma @{typ "string => int"} "σ" expr
    in
      @{term assign} $ var_str $ lam
    end
```

Cette approche garantit que chaque construction syntaxique d'IMP Concur possède un équivalent sémantique précis dans la logique HOL, préservant les invariants du langage source.

Gestion des environnements et substitutions L'encodage met en œuvre un système sophistiqué de gestion des environnements via les fonctions `subst_with_sigma` et `make_lambda_sigma`. Ces mécanismes permettent :

- La transformation des variables libres en applications d'environnement (σ "x")
- La création d'abstractions lambda correctement typées
- La préservation de la portée des variables dans les expressions complexes

```
fun subst_with_sigma (t : term) : term =
  let
    fun subst tm =
      case tm of
        Free (name, _) => sigma $ HOLogic.mk_string name
      | Abs (n, T, body) => Abs (n, T, subst body)
      | tm1 $ tm2 => subst tm1 $ subst tm2
      | _ => tm
  in
    subst t
  end
```

Cette approche assure une traduction fidèle des contextes d'exécution, élément crucial pour la vérification de propriétés dans les systèmes concurrents.

6.4.3 Génération des termes sémantiques

Déclaration automatique des termes de threads La fonction `declare_thread_terms` automatise la création des définitions HOL pour chaque thread du système concurrent. Cette génération produit deux artefacts complémentaires :

1. **Terme commande** : représentation directe de la séquence d'actions du thread
2. **Terme sémantique** : application de la sémantique CSP via l'opérateur Sem_0

```
fun declare_thread_terms (thy: theory) (machine : absy)
                         (threads: thread_absy list) =
  fold (fn {nom_thread, actions, ...} => fn thy' =>
    let
      val com_ast = sequence_actions actions
      val com_term = quote_com machine com_ast
      val sem_term = @{constSem₀} $ com_term $ cont
    in
      thy'''
    end)
```

Cette approche systématique garantit que chaque thread possède une représentation formelle complète, facilitant les analyses ultérieures de propriétés concurrentes.

Séquençage intelligent des actions La fonction `sequence_actions` optimise la représentation des séquences d'actions en :

- Éliminant les instructions vides redondantes
- Construisant des arbres de séquençage équilibrés
- Préservant la sémantique d'ordre d'exécution

Cette optimisation améliore significativement les performances des preuves formelles en réduisant la complexité des termes générés.

6.4.4 Intégration avec la théorie Isabelle/HOL

Typage et contraintes sémantiques L'encodage respecte rigoureusement le système de types d'Isabelle/HOL, notamment :

- Les contraintes de type sur les environnements (`string => int`)
- La cohérence entre les types d'actions et leurs représentations HOL
- La préservation des propriétés de well-formedness des termes

Gestion des identifiants de verrous L'encodage intègre une résolution automatique des identifiants de verrous via `get_lock_id`, assurant la cohérence entre :

- Les déclarations symboliques dans IMP Concur
- Les identifiants numériques utilisés dans la sémantique formelle
- La correspondance univoque entre verrous et threads

6.4.5 Évaluation et simulation

Système d'évaluation des expressions Le module inclut un évaluateur symbolique pour les expressions arithmétiques et booléennes, permettant :

- L'évaluation partielle des expressions constantes
- La détection précoce d'erreurs de type
- La simulation d'exécution pour la validation sémantique

Représentation des états concrets La fonction `update_state` maintient une représentation cohérente des états d'exécution, gérant :

- Les modifications des variables locales et globales
- La propagation des changements d'état entre actions
- La préservation de l'invariant de cohérence des environnements

6.4.6 Robustesse et validation

Gestion des erreurs L'encodage implémente une gestion d'erreurs exhaustive, détectant :

- Les références à des variables non déclarées
- Les incohérences de typage dans les expressions
- Les utilisations incorrectes de verrous et ressources partagées

Validation de la cohérence sémantique Chaque étape de l'encodage inclut des vérifications de cohérence garantissant :

- La préservation des propriétés sémantiques du langage source
- La compatibilité avec les axiomes de la théorie HOL-CSP
- La génération de termes well-formed pour les analyses formelles

6.4.7 Impact sur la chaîne de vérification

Cette phase d'encodage sémantique constitue le pont essentiel entre la représentation syntaxique d'IMP Concur et les capacités de vérification formelle d'Isabelle/HOL. Elle garantit :

- La fidélité de la traduction sémantique
- La préservation des propriétés concurrentes
- La génération d'un modèle formel exploitable pour les preuves de correction
- La facilitation des analyses de propriétés complexes (absence d'interblocage, correction des protocoles de synchronisation)

L'architecture modulaire et la robustesse de cet encodeur constituent des atouts majeurs pour la fiabilité de l'ensemble de la chaîne de vérification formelle d'IMP Concur.

7 Implémentation de la commande SYSTEM

7.1 Architecture générale

La commande `SYSTEM` constitue le point culminant de notre chaîne de compilation, orchestrant la transformation d'un programme IMP Concur validé en un modèle CSP complet et exécutable dans Isabelle/HOL. Cette commande intègre l'ensemble des composants développés précédemment pour produire une spécification formelle cohérente du système concurrent.

L'implémentation s'articule autour de trois phases principales : la génération des définitions de threads, la construction des réseaux de processus communicants, et la composition globale du système. Chaque phase exploite les structures de données validées lors des étapes précédentes pour garantir la cohérence sémantique du modèle final.

7.2 Génération des définitions de threads

La première étape consiste à transformer chaque thread déclaré en une définition HOL formelle. Cette transformation s'effectue via la fonction `define_cmd` qui :

- Génère un terme de type `com` représentant la séquence d'actions du thread
- Crée une définition HOL avec un nom unique (suffixé par `_cmd`)
- Vérifie la well-formedness du terme généré
- Enregistre la définition dans la théorie locale

Cette approche garantit que chaque thread possède une représentation formelle complète, facilitant les analyses ultérieures de propriétés concurrentes. La génération automatique des noms évite les conflits de nommage tout en maintenant une correspondance claire entre les threads source et leurs représentations formelles.

7.3 Construction du réseau de variables globales

Le réseau `GLOBALVARS` modélise l'ensemble des variables partagées du système. Notre implémentation adopte une approche compactifiée utilisant l'opérateur `MultiSync` avec un ensemble d'événements vide, optimisant ainsi la représentation des interactions entre processus :

```
val globals_net_term : term option =
  (case globals_names of
     [] => NONE
   | _ =>
     let
       val empty_evs_set = Const (@{const_name bot}, @{typ "evs set"})
       val multisync_const = Const (@{const_name MultiSync}, ...)
       val compact_term = multisync_const $ empty_evs_set $
                          globals_mset_term $ lam_GLOBALVARS
     in
       SOME compact_term
     end)
```

Cette construction permet une synchronisation efficace entre tous les processus accédant aux variables globales, tout en maintenant la sémantique de partage de mémoire d'IMP Concur.

7.4 Modélisation des sémaphores

Le réseau `SEMAPHORES` encapsule la logique de synchronisation par verrous. Chaque verrou déclaré dans le programme source est associé à un identifiant unique et modélisé comme un processus CSP indépendant :

```
val semaphores_net_term : term option =
  (case lock_ids_terms of
     [] => NONE
   | _ =>
```

```
    let
      val empty_evs_set = Const (@{const_name bot}, @{typ "evs set"})
      val compact_term = multisync_const $ empty_evs_set $
                         locks_mset_term $ lam_SEMAPHORES
    in
      SOME compact_term
    end)
```

Cette modélisation respecte la sémantique des verrous binaires tout en permettant une analyse formelle des propriétés de synchronisation (absence d'interblocage, exclusion mutuelle, etc.).

7.5 Gestion des variables locales

Un défi particulier réside dans la gestion des variables locales à chaque thread. Notre implémentation crée dynamiquement un réseau `LOCALVARS` pour chaque thread possédant des déclarations locales, utilisant l'environnement σ spécifique au thread :

```
fun localvars_net ({locals_decl, locstab, ...} : thread_absy) : term option =
  if null locals_decl then NONE
  else
    let
      val sigma_thread_term = make_sigma_term locstab
      val LOCALVARS_partial = Const (@{const_name LOCALVARS}, ...)
      val compact_term = multisync_const $ empty_evs_set $
                         mset $ lam
    in
      SOME compact_term
    end
```

Cette approche préserve l'isolation des variables locales tout en permettant leur intégration dans le modèle CSP global.

7.6 Composition du système complet

La phase finale assemble tous les composants via une composition séquentielle utilisant l'opérateur de synchronisation `Sync` sur l'ensemble universel d'événements :

1. **Intégration des threads** : Chaque thread est d'abord composé avec son réseau de variables locales (si applicable)
2. **Réseau de threads** : Les threads individuels sont combinés via l'opérateur d'entrelacement
3. **Composition globale** : Le réseau de threads est synchronisé avec les variables globales et les sémaphores

Cette composition respecte la hiérarchie sémantique d'IMP Concur : les variables locales sont privées à chaque thread, les variables globales sont partagées entre tous les threads, et les sémaphores régulent l'accès aux ressources critiques.

7.7 Vérification de cohérence

L'implémentation inclut plusieurs mécanismes de vérification garantissant la cohérence du modèle généré :

- **Vérification de typage** : Chaque terme généré est validé via `Thm.cterm_of`
- **Cohérence des environnements** : Les fonctions respectent les déclarations de variables
- **Unicité des identifiants** : Les verrous et variables possèdent des identifiants uniques
- **Well-formedness** : Tous les termes HOL générés respectent les contraintes syntaxiques

7.8 Optimisations et robustesse

Notre implémentation intègre plusieurs optimisations pour améliorer les performances et la robustesse :

- **Génération à la demande** : Les réseaux de variables et sémaphores ne sont créés que si nécessaire
- **Composition optimisée** : L'utilisation de `MultiSync` réduit la complexité des termes générés
- **Gestion d'erreurs** : Les cas dégénérés (absence de variables, threads vides) sont traités explicitement
- **Préservation des noms** : La correspondance entre noms source et définitions HOL est maintenue

7.9 Intégration avec Isabelle/HOL

La commande `SYSTEM` s'intègre parfaitement dans l'écosystème Isabelle/HOL via :

- **Syntaxe outer** : Utilisation de `Outer_Syntax.command` pour l'enregistrement
- **Gestion de théories** : Manipulation correcte des contextes locaux et globaux
- **Définitions formelles** : Utilisation de `Specification.definition` pour la robustesse
- **Typage rigoureux** : Respect des contraintes de type HOL

Cette intégration garantit que les modèles générés peuvent être directement utilisés pour les preuves formelles et les analyses de propriétés concurrentes dans l'environnement Isabelle/HOL.

7.10 Validation et débogage

L'implémentation inclut un système de traces détaillé facilitant le débogage et la validation :

```
val _ = writeln ("Defined constant: " ^ Binding.name_of const_bnd)
val _ = writeln ("[SYSTEM]  reseau complet :\n  " ^
                 Syntax.string_of_term_global thy' full_system_term)
```

Ces traces permettent de suivre précisément le processus de génération et d'identifier rapidement les problèmes éventuels.

8 Perspectives d'évolution

Cette implémentation constitue une base solide pour des extensions futures, notamment :

- **Optimisations supplémentaires** : Analyse statique pour réduire la complexité des modèles
- **Support d'autres primitives** : Extension vers des constructions concurrentes plus avancées
- **Génération de preuves** : Automatisation partielle des preuves de propriétés courantes
- **Intégration avec d'autres outils** : Export vers des vérificateurs de modèles externes

Par ailleurs, j'ai commencé à me documenter sur les *antiquotations*, avec pour objectif de les utiliser pour générer automatiquement une représentation graphique de l'AST pouvant être directement intégrée dans des documents PDF.

L'architecture modulaire et la robustesse de cette implémentation facilitent ces évolutions tout en préservant la compatibilité avec l'écosystème Isabelle/HOL existant.

9 Conclusion

Ce stage m'a permis de développer une solution complète pour connecter IMP Concur avec Isabelle/HOL-CSP. L'architecture modulaire développée garantit la qualité de la traduction à travers plusieurs étapes de vérification et de validation.

Les principaux apports de ce travail sont :

- Une compréhension approfondie des méthodes formelles et des systèmes de preuve
- L'application concrète de concepts mathématiques avancés tels que le λ-calcul typé pour modéliser des mécanismes de programmation et de preuve
- Le développement d'un traducteur robuste et modulaire
- L'acquisition de compétences avancées en programmation fonctionnelle et en logique formelle
- Une contribution concrète aux outils de vérification formelle de programmes concurrents

Cette expérience ouvre de nombreuses perspectives pour l'extension du langage IMP Concur et l'amélioration des outils de vérification formelle.

References

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

[2] T. Nipkow, L. C. Paulson, M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Springer, 2002.

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

[4] Tobias Nipkow and Robert Sandner, *IMP in HOLCF*, February 20, 2021.

[5] Bryan Scattergood and Philip Armstrong, *CSPM: A Reference Manual*, January 24, 2011.

[6] Benoît Ballenghien and Burkhart Wolff, *A Theory of Proc-Omata and a Proof-technique for Parameterized Process-Architectures*, Theoretical Computer Science, 33 (1984), 279–304.

[7] N. Soundararajan, *Denotational Semantics of CSP*, Computer and Information Science, The Ohio State University, Columbus, OH 43210, USA. Communicated by M. Nivat. Received September 1983; Revised May 1984.

[8] Arshad Beg and Andrew Butterfield, *Development of a Prototype Translator from Circus to CSPm*, Proceedings of the 9th IEEE International Conference on Open Source Systems and Technologies (ICOSST), Lahore, Pakistan, December 2015. DOI: 10.1109/ICOSST.2015.7396396

[9] Nissim Francez, C. A. R. Hoare, Daniel J. Lehmann, Willem P. de Roever, *Semantics of Nondeterminism*, 1978.

[10] Makarius Wenzel With Contributions by Fabian Huch, *The Isabelle System Manual*, 2025.

[11] Diverses théories développées autour d'Isabelle/HOL.