MASTER EN SCIENCES INFORMATIQUES

Conception d'un atelier pour l'algèbre de processus mCRL2

Vander Auwera, Jeremy

Award date:
2021

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2020–2021

Conception d'un atelier pour l'algèbre de processus mCRL2

Vander Auwera Jeremy

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Jean-Marie Jacquet

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Remerciements

Au moment d'écrire ces quelques lignes, ma première pensée va au Professeur Jean-Marie Jacquet. Son œil critique et ses remarques constructives ont en grande partie contribué à faire de ce travail ce qu'il est aujourd'hui. Je lui exprime ici toute ma gratitude pour sa patience, sa disponibilité et sa motivation. Sa compétence dans de nombreux domaines et ses facultés pédagogiques ne sont pas étrangères à mon intérêt pour l'informatique.

Je tiens particulièrement à remercier l'ensemble des professeurs pour la qualité de leur enseignement qu'ils m'ont prodigué au cours de ces deux années passées à Unamur. Je souhaite également adresser ma reconnaissance envers tous les membres du personnel Unamur qui ont permis de près ou de loin à réaliser ce mémoire.

Ma gratitude s'adresse particulièrement à Madame Laurence Defort pour ses retours pertinents. Sa disponibilité, ses relectures et sa gentillesse ont permis d'améliorer fortement la qualité de ce mémoire.

Enfin, je voudrais témoigner ma reconnaissance à ma famille et mes proches, qui m'ont soutenu tout au long de ce travail.

Résumé

Le mCRL2 est une algèbre des processus. Elle possède des outils qui permettent d'aider au raisonnement et à l'analyse de spécifications. Cependant, ils ont tous tendance à modifier la spécification et à ne pas mentionner les processus mais plutôt de se concentrer uniquement sur les actions. Ces deux inconvénients, retrouvés sur tous les outils proposés par mCRL2, proviennent de la transformation de la spécification sous une forme linéaire. Ce mémoire a pour but de développer un outil plus dynamique et qui ne passera pas par cette forme linéaire. Une analyse d'une application qui a été créée sans passer par la forme linéaire est réalisée confirmant les motivations de ne pas utiliser cette forme. Cependant, l'analyse de cette application révèle un manque de dynamisme et se concentre fortement sur les actions en mettant de côté les processus. L'application créée affiche un ascenseur par processus possédant un étage par action possible. Les ascenseurs se déplacent pour amener du dynamisme. La trace des actions exécutées peut se faire par processus. Un retour en arrière est possible pendant l'exécution. Et bien d'autres fonctionnalités palliant les défauts retrouvés au sein des outils analysés. L'aspect technique comprenant un scindement entre le backend et le frontend repose sur les technologies : Scala, Typescript, Angular et Spring. Il offre des points d'entrée API qui permettent nottamment de récupérer toutes les actions exécutables possibles au fur et à mesure de l'exécution. Ce point d'entrée est créé pour permettre d'être réutilisé pour d'autres affichages dynamiques d'exécution de spécification. L'outil créé pousse à la réflexion de la forme linéaire en se demandant si les défauts qu'elle engendre ne valent pas le coup d'être évité au profit des avantages de ne pas l'utiliser. Conscient que cette dernière est au cœur de tous les outils proposés par mCRL2, le présent mémoire suggère une analyse plus poussée de l'utilité de la forme linéaire pour comprendre les tenants et les aboutissants de l'utilisation de cette forme.

Abstract

The mCRL2 is a process algebra. It has tools to help reason about and analyze specifications. However, they all tend to modify the specification and not to mention the processes but rather to focus only on the actions. These two drawbacks, found in all the tools proposed by mCRL2, come from the transformation of the specification into a linear form. The aim of this thesis is to develop a more dynamic tool that will not use this linear form. An analysis of an application that has been created without using the linear form is carried out confirming the motivations for not using this form. However, the analysis of this application reveals a lack of dynamism and focuses strongly on the actions while putting aside the processes. The created application displays one elevator per process with one floor per possible action. The elevators move to bring dynamism. The trace of the executed actions can be done by process. A rewind is possible during the execution. And many other features to overcome the defects found in the analyzed tools. The technical aspect including a split between the backend and the frontend is based on technologies : Scala, Typescript, Angular and Spring. It offers API entry points that allow to retrieve all possible executable actions as they are executed. This entry point is created to allow to be reused for other dynamic specification execution displays. The created tool pushes to the reflection of the linear form by asking if the defects that it generates are not worth to be avoided in favour of the advantages of not using it. Recognizing that the latter is at the heart of all the tools proposed by mCRL2, this thesis suggests further analysis of the utility of the linear form to understand the ins and outs of using it.

Table des matières

Chapitre 1

Introduction

Les programmes informatiques, dotés parfois d'une complexité élevée au niveau de leurs structures comme de leurs comportements, peuvent être analysés, optimisés et décrits à l'aide d'algèbres de processus[1]. Un de ses nombreux avantages est sa lisibilité et sa capacité à être utilisée pour les petits comme pour les grands systèmes [1]. Cette algèbre peut être exécutée pas à pas et ce, de manière dynamique, afin d'aider à l'analyse du comportement logiciel.

Cette exécution dynamique peut se faire grâce à une application. L'objectif de ce mémoire est la création de ce type de logiciel. La description de l'outil, créé dans le cadre de ce mémoire, est proposée au lecteur en trois chapitres.

Le premier chapitre permet, avant de pouvoir parler de la création de l'outil, de maitriser la spécification mCRL2 qui est à la base de l'application. Cependant, pour comprendre le mCRL2, il faut d'abord comprendre ce qu'est l'algèbre des processus. De ce fait, ce chapitre commence par introduire cette notion ainsi que sa relation avec mCRL2. Il constitue donc la base théorique. Il apporte en premier lieu la définition, la description et l'utilité de l'algèbre des processus. En second lieu, il présente les raisons de l'évolution de l'algèbre des processus décrite au moyen de la présentation de son histoire. Cette dernière est présentée en partant de sa première apparition jusqu'au mCRL2. Ensuite, dans le but de donner au lecteur des bases pratiques, le mCRL2 est décomposé en plusieurs versions évolutives. Chaque version permet de répondre à un exemple pratique qui est améliorée petit à petit démontrant l'avantage de la version suivante. Ces versions permettent de passer en revue les différents éléments du langage et de comprendre ce qui a amené leur ajout au sein de cette algèbre des processus. Fort de cet apprentissage, le lecteur possède les outils nécessaires à l'introduction de l'application créée dans le cadre de ce mémoire à la fin de ce chapitre.

Le deuxième chapitre apporte l'aspect fonctionnel du logiciel. La détermination des objectifs du nouvel outil demande de connaitre les avantages et inconvénients des logiciels existants. De ce fait, le chapitre commence par un parcours des outils de la boîte outils proposée par mCRL2. Pour chaque outil, elle décrit leurs fonctionnalités et leurs objectifs. A l'aide de ces diverses descriptions, les limites, les inconvénients ainsi que les qualités de ces différents outils peuvent être exprimés. Ensuite, la description de l'analyse d'un mémoire d'un ancien étudiant est fournie. L'intérêt est fort pour cet ancien mémoire étant donné qu'il propose une solution à un inconvénient retrouvé au sein de tous les outils analysés dans la boîte à outils. Sa présentation permet de comprendre pourquoi l'application créée est considérée comme une version deux de cet outil. Les descriptions d'outils ont permis la création des objectifs de l'application de ce mémoire. Elle a effectivement pour but d'amener une façon de visualiser, d'exécuter une spécification différente des autres outils vus précédemment en mettant au cœur du logiciel les actions et les processus sans changer la spécification mCRL2 et en palliant les défauts trouvés dans les autres outils.

Ces objectifs sont présentés au lecteur en listant les exigences fonctionnelles. Les exigences étant énoncées, une description des fonctionnalités est alors effectuée. Chaque fonctionnalité est décrite à l'aide d'une description textuelle et d'interfaces graphiques de l'application. Pour suivre, du recul est pris en décrivant les remises en question de l'application. Ce recul est fait en fournissant des pistes d'améliorations et des objectifs futurs de l'outil que le lecteur pourra parcourir.

Pour terminer, maintenant que le lecteur connait le "quoi" de l'application, le "comment" est décrit. Ce

dernier est l'objectif du dernier chapitre. Il commence par expliquer les motivations des choix des technologies utilisées au sein de l'application. Ensuite, il apporte la vue globale du fonctionnement technique du système. Cette vue est ensuite décrite de manière plus précise en expliquant les fonctionnalités du système de manière détaillée. Ces détails demandent des bases de programmation pour être comprise, mais pas spécialement d'être un expert des outils de programmation avec lesquels elles ont été créées. Effectivement, les langages et les Frameworks utilisés sont introduits avant de fournir les détails des fonctionnalités principales du système. Le chapitre se termine par une remise en question des décisions, une exposition des limites techniques et une explication des futurs défis techniques à résoudre.

Chapitre 2

Fondements des algèbres de processus

L'objectif de ce chapitre est de fournir aux lecteurs les bases théoriques à la compréhension de l'algèbre mCRL2. Dans un premier temps, nous nous efforcerons de décrire et de définir la notion d'algèbre de processus. Ensuite, nous en démontrerons son utilité. Enfin, partant du principe que cette notion ne peut pas être totalement comprise sans son histoire, nous en expliquerons les grandes étapes de son évolution.

La suite fournira aux lecteurs un tutoriel sur mCRL2 ainsi que sur ses multiples concepts. Pour ce faire, le mCRL2 sera décomposé en versions évolutives. La première version introduira les bases de la définition d'actions et de processus. Toutefois, elle n'est pas suffisante pour décrire des processus simples. Les versions intermédiaires ajouteront des concepts et viendront pallier les lacunes des versions précédentes jusqu'à arriver à une dernière version qui permettra la description de processus complexe.

2.1 Intuition

Il est possible de représenter nos interactions dans le monde sur base de transitions. Une transition peut être vue comme le passage d'un état de départ vers un état d'arrivée suite à l'exécution d'une action [2]. Les transitions peuvent être exprimées sous plusieurs formes. L'expression de ces transitions peut se faire à l'aide d'un schéma. La transition schématique peut se faire à l'aide d'une flèche et de deux états :

$$etat\_depart \xrightarrow{evenement} etat\_arrivee$$

.

En pratique, on pourrait décrire que lorsque l'utilisateur insère une pièce de 5 dans une machine, il peut choisir entre un café ou un thé que la machine préparera et lui fournira. En revanche, si l'utilisateur décide de mettre une pièce de 10, les mêmes transitions que celles avec la pièce de 5 seront exécutées mais en plus l'utilisateur recevra en retour une pièce de 5 étant donné que le café et le thé ne coûtent que 5.

Sous forme de transition on pourrait le représenter comme à la figure **??**.

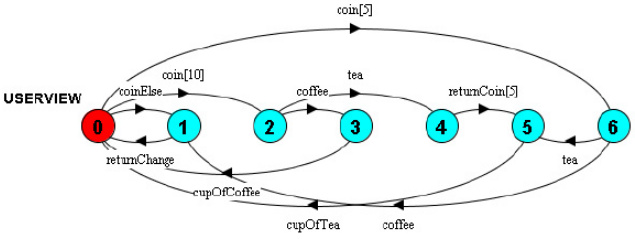


FIGURE 2.1 – Représentation transitionnelle d'une machine à café [3].

On se rend facilement compte que les deux formes sont plutôt intuitives. Cependant, la forme textuelle a

le défaut de comprendre trop de détails, d'être lourde et de ne se limiter qu'à l'observation des transitions. La forme de transition schématique permet de résoudre ces problèmes de détail et de pouvoir ainsi se concentrer sur les transitions. Elle reste toutefois assez lourde à écrire. De plus, elle manque de clarté à cause des flèches.

Pour pallier aux problèmes cités, une possibilité est de formaliser les transitions au moyen d'égalité algébrique. Pour cela, reprenons nos différentes actions possibles :

$$\boxed{\begin{array}{l} E0 \xrightarrow{c5} E1 \xrightarrow{prepareTea} E2 \xrightarrow{sendTea} E3; \\ E0 \xrightarrow{c5} E1 \xrightarrow{prepareCafee} E4 \xrightarrow{sendCafee} E5; \\ E0 \xrightarrow{c10} E6 \xrightarrow{prepareTea} E2 \xrightarrow{sendTea} E3 \xrightarrow{send5c} E7; \\ E0 \xrightarrow{c10} E6 \xrightarrow{prepareCafee} E4 \xrightarrow{sendCafee} E5 \xrightarrow{send5c} E7 \end{array}}$$

Dans ces représentations d'événements deux notions importantes ressortent : la notion de séquence ainsi que celle de choix d'événement [2]. On peut déterminer que le choix d'événement se représentera par un '+' et que la séquence se représentera par un '.'. Notre représentation pourra se simplifier de la façon suivante :

$$\boxed{\begin{array}{l} (c5.prepareTea + prepareCafee.sendCafee) + \\ ((c10.prepareTea + prepareCafee.sendCafee).send5c) \end{array}}$$

Cette représentation permet de voir directement où se trouvent les choix et les séquences. Il est possible à l'aide de syntaxe et de sémantique et de raisonnement mathématique de raisonner sur notre monde et ses interactions.

Ce besoin de plus en plus grand de devoir vérifier, représenter et comprendre des systèmes complexes nous amène à l'utilisation de l'algèbre des processus [2][4].

2.2 Description de l'algèbre des processus

Le terme "algèbre de processus" peut se comprendre de plusieurs façons. Dans le cadre de ce mémoire définissons d'abord le terme processus. Il désigne le comportement d'un système logiciel, les actions d'une machine, ou même les actions d'un être humain [1]. Le comportement est l'ensemble des événements ou des actions qu'un système peut effectuer, l'ordre dans lequel ils peuvent être exécutés et peut-être d'autres aspects de cette exécution tels que le temps ou les probabilités. L'accent est toujours mis sur certains aspects du comportement, sans tenir compte d'autres aspects. On peut plutôt dire qu'il y a une observation du comportement et qu'une action est l'unité d'observation choisie [1]. En général, on considère que les actions se déroulent à un moment donné et que l'on peut distinguer différentes actions dans le temps. C'est pourquoi un processus est parfois aussi appelé système d'événements discrets. Le terme d'algèbre fait référence au fait que l'approche adoptée pour raisonner sur le comportement est algébrique et axiomatique. C'est-à-dire que les opérations sur les processus sont définies et leurs lois équationnelles sont étudiées. En d'autres termes, les méthodes et techniques de l'algèbre universelle sont utilisées.

Pour établir une comparaison, considérons la définition d'un groupe en algèbre universelle. Un groupe est une structure (G, *, $^{-1}$, u) avec G l'univers des éléments, l'opérateur binaire *, l'opérateur unaire $^{-1}$ et la constante u$\in$ G. Pour tout a,b,c $\in$ G, les axiomes, sont :

$$a * (b * c) = (a * b) * c$$

$$u * a = a = a * u$$

$$a * a^{-1} = a^{-1} * a = u$$

[1]

Un groupe peut donc se définir comme une structure mathématique constituée d'un seul univers d'éléments, avec des opérateurs sur cet univers d'éléments qui satisfont aux axiomes du groupe. En d'autres termes, un groupe est tout modèle de la théorie équationnelle des groupes [1]. De même, il est possible de définir des

opérations sur l'univers des processus [1]. Une algèbre des processus est alors toute structure mathématique satisfaisant les axiomes donnés pour les opérateurs définis, et un processus est alors un élément de l'univers de cette algèbre des processus. Les axiomes permettent des calculs avec des processus, souvent appelés équations.

L'algèbre des processus trouve donc ses racines dans l'algèbre universelle. Cependant, le domaine d'étude que l'on appelle aujourd'hui l'algèbre des processus dépasse souvent les limites strictes de l'algèbre universelle. Parfois, la restriction à un seul univers d'éléments est assouplie et différents types d'éléments, différentes sortes, sont utilisés et parfois des opérateurs contraignants sont considérés [1].

Le modèle le plus simple du comportement d'un système consiste à considérer le comportement comme une fonction d'entrée/sortie [1]. Une valeur ou une entrée est donnée au début d'un processus et ensuite il y a une valeur en tant que résultat ou sortie. Ce modèle de comportement a été avantageusement utilisé comme modèle le plus simple du comportement d'un programme informatique, dès le début du sujet au milieu du XXe siècle. Il a joué un rôle déterminant dans le développement de la théorie des automates. Dans cette théorie, un processus est modélisé comme un automate [1]. Un automate a un certain nombre d'états et un certain nombre de transitions, allant d'un état à l'autre. Une transition dénote l'exécution d'une action, l'unité de base du comportement. De plus, l'automate a un état initial et un certain nombre d'états finaux. Étant donné cette abstraction comportementale de base, un aspect important est de savoir quand on peut considérer deux automates égaux, ceci étant exprimé par une notion d'équivalence, l'équivalence sémantique. Sur les automates, la notion de base de l'équivalence sémantique est l'équivalence de langage : un automate est caractérisé par l'ensemble de ses chemins d'exécution, et deux automates sont égaux quand ils ont le même ensemble de chemins. Une algèbre qui permet un raisonnement équationnel sur les automates est l'algèbre d'expressions régulières [1].

Par la suite, on a constaté que le modèle des automates faisait défaut dans certaines situations. Ce qui manque fondamentalement, c'est la notion d'interaction : pendant l'exécution de l'état initial vers l'état final, un système peut interagir avec un autre système [1]. Ceci est nécessaire pour décrire les systèmes parallèles ou distribués. Lorsqu'il s'agit de modèles et de raisonnements sur des systèmes en interaction, on utilise l'expression "théorie de la concurrence". La théorie de la concurrence est la théorie dite des systèmes en interaction, parallèles et/ou distribués. L'algèbre des processus est généralement considérée comme une approche de la théorie de la concurrence. Cela explique pourquoi une algèbre des processus aura généralement une composition parallèle comme opérateur de base. Dans ce contexte, les automates sont principalement appelés systèmes de transition [1]. La notion d'équivalence étudiée n'est généralement pas l'équivalence de langage. Parmi les équivalences étudiées, la notion de bisimilarité, qui considère que deux systèmes de transition sont égaux si et seulement s'ils peuvent imiter le comportement de l'autre dans n'importe quel état qu'ils peuvent atteindre, occupe une place importante.

L'algèbre des processus est défini par une syntaxe et une sémantique. La syntaxe fut simple au départ et ne comprenait qu'un nombre réduit d'opérateurs algébriques primitifs. Ces opérateurs algébriques permettent par assemblage de décrire des comportements complexes. L'évolution de la syntaxe a poussé progressivement l'évolution de ces derniers à devenir des langages à part entière. La manipulation de données ou la favorisation de la modularité des spécifications explique l'enrichissement de la syntaxe [1]. La sémantique est définie de manière axiomatique ou opérationnelle. La sémantique axiomatique est un ensemble de lois algébriques comme la commutativité, l'associativité ou la distributivité des opérateurs qui permettent de démontrer l'équivalence de termes. Une sémantique opérationnelle repose sur une relation de transition $A \xrightarrow{L} B$ qui exprime que A a le droit d'exécuter l'action L et devenir B. Cette règle de transition détermine une correspondance entre un terme A et un automate qui décrit les évolutions futures de A. Il est ainsi possible d'exécuter les termes algébriques et de vérifier leur correction en analysant l'automate qui leur correspond [5].

Une définition possible du domaine de l'algèbre des processus est le domaine qui étudie le comportement des systèmes parallèles ou distribués par des moyens algébriques [6]. Il offre des moyens pour décrire ou spécifier des systèmes, et pour cela il a des moyens pour parler de composition parallèle. En outre, il peut généralement aussi parler de composition alternative et séquentielle. De plus, il est possible de raisonner sur de tels systèmes en utilisant un raisonnement équationnel. Grâce à ce raisonnement équationnel, la vérification devient possible et il peut être établi qu'un système satisfait à une certaine propriété [1]. Souvent, l'étude des systèmes de transition, des moyens de les définir et des équivalences sur ceux-ci sont également considérés comme faisant partie de l'algèbre des processus, même en l'absence de théorie équationnelle. L'algèbre des processus a conduit à la définition de multiples langages comme CSP, CCS, TCSP, ACP, PSF ou mCRL2 et il en existe bien d'autres. [5].

2.3 Raison d'existence de l'algèbre des processus

Au-delà des programmes séquentiels capables de recevoir des données en entrée et de fournir des résultats en sortie, beaucoup de systèmes informatiques sont composés de plusieurs processus qui s'exécutent parallèlement tout en s'échangeant des informations. Ces systèmes sont appelés systèmes parallèles. Les formes de parallélisme peuvent être synchrone ou asynchrone [1]. Le parallélisme est considéré comme synchrone lorsque l'exécution des processus est pilotée à intervalles réguliers par une horloge globale interne au système [5]. A l'inverse, le parallélisme est considéré comme asynchrone lorsque l'exécution des processus est libre. Ces processus peuvent tout de même être synchronisés lors de l'accès à des ressources partagées dans le but de conserver la cohérence des données [1].

Les études de ces systèmes asynchrones sont nombreuses et importantes et servent d'applications dans les domaines du matériel, des télécommunications et du logiciel [5]. Ces recherches ont trouvé leurs origines dans l'apparition des systèmes distribués. Elles ont ensuite été alimentées et continuent de l'être aujourd'hui par les protocoles de télécommunication et de réseau. L'intérêt pour les systèmes asynchrones s'explique également par l'augmentation de la vitesse et de la consommation énergétique faible par rapport à leurs prédécesseurs synchrones [5].

Les programmes synchrones restent cependant plus faciles à concevoir et à mettre au point. L'amélioration continue des langages de programmation permet plus facilement la mise en place de systèmes séquentiels complexes. Les concepts informatiques de typage, de programmation structurée, d'architecture modulaire, d'orientée objet et bien d'autres font partie de cette simplification.

En revanche, les systèmes asynchrones restent moins avancés. Plusieurs raisons expliquent cette situation :

- Leur exécution est indéterminée. En d'autres termes, il peut être impossible de prédire ou de reproduire leur exécution ce qui complique la mise au point et les tests. La vérification automatisée sera limitée étant donné qu'un système asynchrone peut, à un ou plusieurs moments, avoir plusieurs évolutions possibles [5].

- La propriété évidente de compositionnalité n'existe pas dans ces systèmes. L'assemblage de sous-systèmes simple et correct ne permettra pas de s'assurer que le système englobant est correct. Prenons l'exemple d'un blocage : même si deux processus ne comportent pas de blocage séparément, il n'est pas prouvé que leur exécution asynchrone n'engendra pas un blocage[1].

- Le cerveau humain a plus de mal à appréhender les systèmes asynchrones. De ce fait, Il n'est pas rare d'observer des premières versions de protocoles buggés.

Pour faire face à ces difficultés, de multiples recherches ont été effectuées. Une des solutions est la modélisation correcte des processus asynchrones par assemblage de processus décrits séparément. Cette solution demande cependant des formalismes adaptés qui permettent d'exprimer le parallélisme [5].

Un des formalismes qui a connu un grand succès est l'algèbre des processus. L'algèbre des processus offre des moyens de décrire ou de spécifier ces systèmes, de parler de composition parallèle et peut exprimer des compositions alternatives ou séquentielles.

2.4 Évolution de l'algèbre des processus

L'algèbre des processus a commencé au début des années 1970. A cette époque, la seule partie de la théorie de la concurrence existante est la théorie des réseaux de Petri. On peut distinguer 3 styles de sémantique sur les programmes informatiques : la sémantique opérationnelle, la sémantique dénotationnelle et la sémantique axiomatique. Dans la sémantique opérationnelle, la modélisation se base surtout sur l'évaluation de variables et sur des changements d'états. Dans la sémantique dénotationnelle, la modélisation est généralement exprimée par une fonction transformant l'entrée en sortie. Dans la sémantique axiomatique, on met l'accent sur les méthodes de preuve de véracité des programmes.

Par la suite, il a été constaté qu'il était difficile de donner une sémantique aux programmes contenant un opérateur parallèle en utilisant une des 3 styles sémantiques de l'époque. Avant que la théorie des programmes

parallèles ne soit élaborée en termes d'algèbre des processus, deux changements de paradigme ont été nécessaires. Le premier est qu'il a fallu se rendre compte que l'interaction d'un processus entre l'entrée et la sortie peut influencer le résultat et peut même perturber son comportement fonctionnel. Le second est qu'il faut réaliser que l'exécution indépendante de processus parallèle rend difficile, voire impossible, la détermination des valeurs des variables globales à un moment donné. Il a fallu prendre conscience qu'il était plus simple de laisser chaque processus gérer ses propres variables locales et ainsi d'indiquer explicitement l'échange d'informations.

Cette section décrit brièvement l'histoire de l'algèbre des processus en partant des personnes ayant fortement influencé l'émergence de l'algèbre des processus jusqu'au mCRL2.

2.4.1 Bekic

Bekic est une de personne qui a contribué à l'émergence de l'algèbre des processus. Dans les années 1960, il a notamment travaillé sur la sémantique dénotationnelle ALGOL et PL/I. Il est à l'origine d'une loi portant sur la composition quasi parallèle, appelée l'entrelacement non spécifié des étapes élémentaires de deux processus [6, 7]. Cette loi est sans aucun doute à la base de ce que l'on appellera plus tard la loi d'expansion de l'algèbre des processus. Elle explique aussi pourquoi Bekic a opéré un premier changement de paradigme : l'étape suivante de l'entrelacement n'est pas déterminée, de sorte que l'idée d'un programme en tant que fonction a été abandonnée. Dans une conférence en 1974 à Newcastle, il présente un opérateur "parallèle gauche", avec des lois et avec la même signification que celle que Bergstra et Klop donneront plus tard à leur opérateur de gauche [6].

En conclusion, Bekic a apporté un certain nombre d'ingrédients de base à l'algèbre des processus, mais il ne fournit pas encore une théorie globale cohérente.

2.4.2 Calculus of Communication System

Calculus of Communication System (CCS) est un système qui permet à la fois de fournir des spécifications rigoureuses et des moyens de vérifier l'exactitude de ces spécifications [8]. Ce système a été développé par Robin Milner entre 1973 et 1980 [7]. Dans une de ses anciennes publications, il considère le problème d'un programme qui ne finit pas en se penchant sur ses effets de bord et sur le non-déterminisme. Dans cette publication, il utilise l'opérateur * pour les compositions séquentielles, ? pour les alternatives et $\|$ pour les compositions parallèles.

Plus tard, Milner écrira un article avec Matthew Hennessy où il formule un CCS de base avec une équivalence observationnelle et une équivalence forte définie intuitivement. Il introduit également une logique qui fournit une caractérisation logique de l'équivalence des processus.

Milner publiera ensuite le livre "A Calculus of Communicating Systems" qui deviendra la référence en matière d'algèbre des processus [9]. Milner parle de calcul de processus partout dans son travail, en insistant sur l'aspect calculatoire. Il présente les lois équationnelles comme des vérités dans le domaine sémantique qu'il a choisi, plutôt que de considérer les lois comme primaires, et d'étudier l'éventail de modèles dont elles disposent [9].

Pour la première fois dans l'histoire, on peut parler d'algèbre des processus.

2.4.3 Communicating Sequential Processes

Tony Hoare a également influencé le développement de l'algèbre des processus en contribuant à la création du langage Communication Sequential Processes (CSP) [10]. Le langage CSP (Communicating Sequential Processes) a pour but de décrire les interactions entre des processus concurrents [1]. Hoare utilise le calcul pour montrer qu'il est possible de travailler avec des interblocages et du non-déterminisme comme s'il s'agissait d'événements terminaux dans des processus ordinaires.

Le modèle de CSP s'est d'abord basé sur la théorie des traces, c'est-à-dire sur les séquences d'actions qu'un processus peut effectuer [10]. Cependant, ce modèle a été remplacé par un modèle sur les paires qui sont des chemins qui amènent à des échecs car le précédent ne conservait pas les comportements de "deadlock". Par la

suite, il a été conclu que le modèle d'échec est le modèle le moins discriminant qui préserve les comportements de "deadlock" [6]. L'importance de Tony Hoare est qu'il est l'auteur du deuxième changement de paradigme important afin d'arriver à la théorie des programmes parallèles qu'on connait de nos jours : la disparition des variables globales au sein de CSP.

2.4.4 Algebra of Communicating Processes

Algebra of Communicating Processes (ACP) fait référence au document écrit par Jan Bergstra et Jan Willem Klop en 1982 [1, 11, 12]. Dans ce document, il est cité : "Une algèbre de processus sur un ensemble d'actions atomiques A est une structure $\mathcal{A} = \langle \mathrm{A}, +, ., \between, a_i \ (\mathrm{i} \in \mathrm{I}) \rangle$où A est un ensemble contenant A, les a_i sont des symboles constants correspondant à l' $a_i \in A$, et + (union), . (concaténation ou composition, laissée de côté dans les axiomes), $\between$ (entrelacement gauche) satisfont pour tous les x, y, z $\in$ A et $a \in A$ les axiomes suivants :

PA1	x + y = y + x
PA2	(x + y) + z = x + (y + z)
PA3	x + x = x
PA4	(xy) z = x (yz)
PA5	(x + y) z = xz + yz
PA6	(x + y) $\between$ z = x $\between$ z + y $\between$ z
PA7	ax $\between$ y = a (x $\between$ y + y $\between$ x)
PA8	a $\between$ y = ay

On peut constater que l'expression "algèbre des processus" a été employée pour la première fois dans ce document [1, 11].

Par la suite, d'autres algèbres de processus ont été développées chacune faisant son propre ensemble de choix dans les différentes possibilités. Chaque algèbre des processus possède son lot d'avantages et d'inconvénients [1]. Lorsqu'une notion est utilisée dans deux algèbres de processus différentes avec la même intuition sous-jacente, mais avec un ensemble différent de lois équitables, certains plaident pour avoir la même notation, afin de montrer que nous parlons vraiment de la même chose, et d'autres plaident pour avoir différentes notations, afin de mettre l'accent sur les différents contextes sémantiques.

2.4.5 Micro CommonRepresentation Language (µCRL)

Dans le but de construire un langage vers laquelle tous les langages de spécification existantes pourraient être traduites, un langage de représentation commun (LRC) a été construit dans le cadre d'un projet financé par la CE appelé SPECS [13]. Ce langage est devenu tellement complexe qu'il était impossible d'élaborer une sémantique cohérente, et encore moins de l'utiliser comme base pour une théorie ou une construction d'outils supplémentaires. À la suite de ces découvertes, en 1990, un langage minimal appelé µCRL (micro CommonRepresentation Language) est apparu comme le langage le plus simple possible pour modéliser des systèmes réalistes. Ce langage est une algèbre de processus avec des données [4]. Il se base sur l'algèbre des processus de communication (ACP) en ajoutant la possibilité de définir et d'utiliser des types de données abstraits [4]. Le µCRL est un langage de spécification formel avec un ensemble d'outils associés [4]. La possibilité d'utiliser des données dans le cadre d'une algèbre de processus spécifique est une grande amélioration.

Le langage µCRL possède une syntaxe et une sémantique claire et bien définie qui simplifie les vérifications. De nombreux systèmes ont été vérifiés à l'aide de ces techniques, mais la vérification du protocole à l'aide de fenêtre coulissante est particulièrement remarquable. La vérification de ce protocole a conduit à la détection d'une impasse inconnue dans le protocole. Elle a montré que le comportement externe du protocole original était d'une complexité prohibitive et a catalysé le développement de nombreuses méthodologies de preuve [13].

Les grandes spécifications, comme les programmes ordinaires, se sont avérées contenir des défauts de telle façon que des outils ont été utilisés pour s'assurer de l'absence d'anomalie. Pour des vérifications simples, l'outil peut traiter des systèmes avec plus de 10^9 états. En utilisant la confluence, l'interprétation abstraite et le raisonnement symbolique, des systèmes beaucoup plus importants, contenant des centaines de composants, ont été vérifiés.

Au fil des années, divers outils ont été développés pour le µCRL, tous fondés sur des théories formelles.

L'outil a également joué un rôle essentiel dans l'enseignement de la conception de systèmes fiables dans diverses universités.

Les données sont spécifiées à l'aide d'une logique équationnelle du premier ordre, ce qui était la norme à l'époque. Les langages développés antérieurement, tels que LOTOS et PSF, contenaient également des types de données équationnelles. Cependant, la simplicité du µCRL lui a permis de se différencier des autres langages.

2.4.6 Milli Common Representation Language (mCRL2)

Le langage mCRL2 (milli Common Representation Language) est le successeur de µCRL. Ses auteurs sont des chercheurs de l'université de technologie d'Eindhoven [13]. Le but de ces chercheurs était de fournir un langage et une interface plus conviviale que celle du µCRL en se basant sur les expériences des utilisateurs du langage µCRL. Le mCRL2 est utile pour la modélisation, la validation ainsi que la vérification de systèmes et de protocoles concurrents. Le changement de nom s'explique par le fait que le nom µCRL n'était pas vraiment adapté à Internet en raison de la lettre grecque initiale [13]. Les développeurs ont dès lors décider de lui donner un nouveau nom par l'ajout d'un numéro de version au nom. La suite sera donc mCRL3, mCRL4, ... [13]. Un exemple d'amélioration par rapport au µCRL est l'introduction des types de données prédéfinis et d'ordre supérieur, des expressions de calcul lambda et diverses autres constructions linguistiques qui sont conçues pour rendre le type de données des définitions plus courtes et plus faciles à lire et à écrire [4]. En ce qui concerne les processus, le changement le plus remarquable est l'introduction de la multiplicité d'actions qui permettent une conversion plus facile des réseaux de Pétri selon les spécifications mCRL2 [14]. En termes d'utilisateur, l'outil mCRL2 est complété par une interface utilisateur graphique ce qui améliore la convivialité de l'ensemble d'outils et peut être utilisé en parallèle avec l'interface de commande traditionnelle [14].

2.5 Présentation du mCRL2

Cette section a pour but de présenter la syntaxe et la sémantique de mCRL2. Elle permet de fournir des bases suffisantes pour la compréhension et l'utilisation de mCRL2. Afin de guider au mieux le lecteur, le mCRL2 est présenté en 5 versions. Chaque version comprendra toutes les fonctionnalités des versions précédentes tout en y ajoutant des nouvelles.

(a) mCRL2, modèle de base
(b) mCRL2, modèle avec composition alternative et séquentielle
(c) mCRL2, modèle avec composition parallèle
(d) mCRL2, modèle avec encapsulation
(e) mCRL2, modèle avec récursivité
(f) mCRL2, modèle avec opérateur d'abstraction

2.5.1 mCRL2 - Modèle de base

Cette version apporte les bases de mCRL2. Elle a pour but de décrire les différentes possibilités de déclaration d'actions et de processus. Elle permet également de comprendre la relation de composition qui existe entre les processus et les actions. Cette version comporte des limites exposées en fin de section. Ces limites démontrent que cette version n'est pas suffisante pour décrire des processus.

Les actions

Le fonctionnement primitif des processus est l'action. Cette dernière est une des notions les plus importantes dans le langage du processus mCRL2 [4, 15]. Une action représente un événement atomique et qui peut être de toute nature. Elle peut être de manière exhaustive : l'envoi d'un message, le dépôt d'argent sur un compte ou même le salut de quelqu'un [15]. Avec mCRL2, on déclare des actions après le mot clé 'act', en indiquant le nom de l'action en minuscule :

act *action*1;

Il est possible de définir plusieurs actions à l'aide du séparateur ','. Exemple :

$$\boxed{\textbf{act}\ action1, action2, action3;}$$

Les actions peuvent également être déclarées par un passage à la ligne :

$$\boxed{\begin{array}{ll}\textbf{act} & action1;\\ & action2;\\ & action3;\end{array}}$$

Une action peut avoir zéro, un ou plusieurs arguments [15]. Pour déclarer une action possédant un argument, on stipule l'action suivie de deux points et du type de l'argument :

$$\boxed{\textbf{act}\ action1 : Type;}$$

Lorsque l'action possède plusieurs arguments, il faut les séparer par le signe '#' :

$$\boxed{\textbf{act}\ action1 : Type1\ \#\ Type2;}$$

Les processus

Un processus exécute une ou plusieurs actions [15]. Pour les définir, mCRL2 utilise le mot clé 'proc' suivi du nom de processus en commençant le nom par une majuscule, d'un signe '=' et des différentes actions qu'il effectue :

$$\boxed{\begin{array}{l}\textbf{act}\ action1;\\ \textbf{proc}\ Processus1 = action1;\end{array}}$$

Il est possible de définir plusieurs processus. Comme pour les actions, le passage à ligne sans mot clé indique la définition de plusieurs processus :

$$\boxed{\begin{array}{ll}\textbf{act} & action1, action2;\\ \textbf{proc} & Processus1 = action1;\\ & Processus2 = action2;\\ & Processus3 = action2;\end{array}}$$

Un processus peut se composer d'action et/ou de processus. Dans l'exemple ci-dessous, le Processus1 fait appel au Processus2 qui exécute le Processus3. L'exécution de ce dernier réalise l'action3.

$$\boxed{\begin{array}{ll}\textbf{act} & action3;\\ \textbf{proc} & Processus1 = Processus2;\\ & Processus2 = Processus3;\\ & Processus3 = action3;\end{array}}$$

Le mot clé 'init' a une grande importance dans la spécification de processus. Il est le point de départ de l'exécution considérée. Il est le début de l'enchaînement d'actions ou de processus. Lorsqu'on déclare plusieurs processus, il est essentiel de pouvoir exprimer le premier processus exécuté. Cette fonctionnalité est gérée par le mot clé : 'init'. Un enchaînement d'actions et/ou de processus peut suivre le mot clé 'init'. Il est possible de déclarer une algèbre des processus en indiquant des actions après le mot clé 'init', sans déclarer aucun processus. Avec ces nouvelles connaissances, on pourrait réécrire notre exemple précédent de deux façons différentes. La première en plaçant l'action3 dans 'init' :

$$\boxed{\begin{array}{ll}\textbf{act} & action3;\\ \textbf{init} & action3;\end{array}}$$

La seconde en stipulant le Processus1 dans la clause 'init' :

$$\begin{array}{|ll|}\hline \textbf{act} & action3; \\ \textbf{proc} & Processus1 = Processus2; \\ & Processus2 = Processus3; \\ & Processus3 = action3; \\ \textbf{init} & Processus1; \\ \hline \end{array}$$

Une autre particularité des actions est le passage d'arguments. Les processus peuvent passer des arguments aux actions. Pour déclarer l'appel à une action avec des arguments, il faut faire suivre la déclaration de l'action d'une parenthèse contenant les différents arguments séparés par une virgule [15].

$$\begin{array}{|l|}\hline \textbf{act}\ action1 : Bool\ \#\ Nat; \\ \textbf{proc}\ Processus1 = action1(true, 1); \\ \hline \end{array}$$

Dans le cas ci-dessus, l'action1 est appelée avec un argument de type Bool (boolean) "true" et un argument de type Nat (natural) 1.

Limite

A l'aide de notre mCRL2 version 0, il est possible de déclarer des actions et des processus. Cette première version est très limitée. Elle permet de représenter des processus composés d'une seule action ou d'un seul processus. Cependant, les processus se composent de plusieurs actions et/ou de plusieurs processus qui s'enchaînent de manière particulière qu'il faut pouvoir exprimer sous forme algébrique.

Pour mettre en évidence les manquements de notre version 0, représentons l'exécution simplifiée du processus de commande. Dans ce processus de commande, l'utilisateur doit indiquer à la machine la commande souhaitée. Une fois la commande indiquée, la machine prévient le responsable du magasin afin qu'il prépare la commande du client. Lorsque la commande est prête, l'utilisateur peut procéder au paiement sur la machine par carte ou par liquide.

A l'aide de notre version 0, on peut indiquer les différentes actions qui ont lieu du point de vue de notre machine :

$$\begin{array}{|l|}\hline \textbf{act}\ transmettreCommande, attendrePreparationCommande, \\ \quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementLiquide; \\ \hline \end{array} \tag{2.1}$$

La deuxième étape consiste à stipuler les processus. On réalise tout de suite que nous ne sommes pas capables, avec cette version, de stipuler un enchaînement d'actions et/ou de processus.

$$\begin{array}{|l|}\hline \textbf{act}\ transmettreCommande, attendrePreparationCommande, \\ \quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementLiquide; \\ \textbf{proc}\ \ Machine = attendreCommande?????; \\ \hline \end{array}$$

La seule possibilité pour nous de définir ce processus est d'abstraire l'enchaînement d'action :

$$\begin{array}{|l|}\hline \textbf{act}\ traiterCommande; \\ \textbf{proc}\ \ Machine = traiterCommande; \\ \hline \end{array}$$

La plupart du temps, la représentation d'un ou plusieurs processus doit pouvoir aider les personnes à raisonner sur cet algorithme. Abstraire les étapes sur lesquelles le raisonnement doit se faire permet de rendre cette algèbre inutile.

On peut donc conclure que cette version ne permet pas de décrire des processus. En revanche, elle permet de préparer le terrain pour la version suivante qui ajoute les compositions alternatives et séquentielles à mcRL2.

2.5.2 mCRL2 - Modèle avec composition alternative et séquentielle

La version 1 décrit les compositions alternatives et séquentielles en mCRL2. Les compositions sont essentielles à la description d'un processus. Les deux compositions sont expliquées séparément pour ensuite être décrites ensemble. Les compositions comportent des axiomes qui sont listés en fin de section. Cette version est une amélioration de la précédente. Elle comporte cependant des lacunes que nous démontrerons à l'aide de notre exemple de commande.

Les compositions séquentielles

Pour décrire le comportement de processus, il est essentiel de pouvoir combiner des actions et/ou des processus de manière séquentielle. Cet enchaînement d'action se fait à l'aide de l'opérateur '.'. Cet opérateur peut se situer entre deux actions, deux processus ou entre une action et un processus. Il spécifie que le membre de gauche sera d'abord exécuté et ensuite son membre de droite.

L'exemple ci-dessous exprime en mCRL2 qu'une personne doit pour passer la commande d'un verre commencer par attendre le serveur. Ensuite, elle peut commander. Son algèbre des processus sera la suivante :

act	$attendreServeur, passerCommande;$
proc	$CommandeVerre = attendreServeur.passerCommande;$
init	$CommandeVerre;$

Schématiquement, on pourrait le dessiner de la manière suivante :

$$CommandeVerre = \xrightarrow{attendreServeur} \xrightarrow{passerCommande}$$

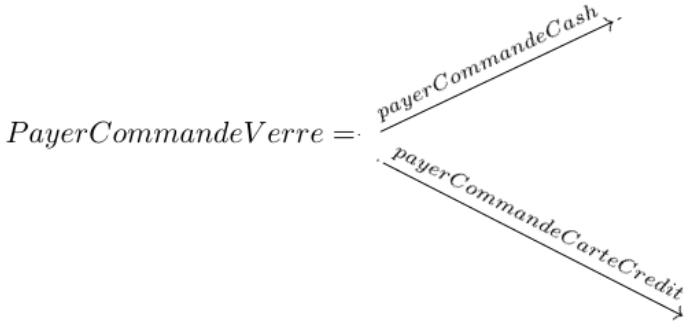
Les compositions alternatives

Une autre combinaison importante pour spécifier le comportement d'un processus est la composition alternative. Pour ce faire, on utilise l'opérateur '+'. Cet opérateur peut se situer entre deux actions, deux processus ou entre une action et un processus. Il spécifie que l'on peut exécuter soit son membre de gauche soit son membre de droite en faisant le choix d'une manière non déterministe sur base des premiers pas exécutables.

L'exemple ci-dessous spécifie en mCRL2 que lors du paiement de la commande d'un verre on peut soit payer la commande par carte de crédit soit en cash.

act	$payerCommandeCash, payerCommandeCarteCredit;$
proc	$PayerCommandeVerre = payerCommandeCash + payerCommandeCarteCredit;$
init	$PayerCommandeVerre;$

Schématiquement, on pourrait le dessiner de la manière suivante :



Combinaison des deux compositions

Il est possible de combiner la composition alternative et la composition séquentielle au sein d'une algèbre de processus. Souvenons-nous de nos deux exemples précédents : Une personne doit pour passer la commande d'un verre commencer par attendre le serveur. Ensuite, elle peut commander. Après la commande, le client peut choisir entre payer en cash ou par carte de crédit. En mixant nos compositions, on peut le représenter de la manière suivante :

act $attendreServeur, passerCommande, payerCash, payerCarte$;
proc $CommandeVerre = attendreServeur.passerCommande.(payerCash + payerCarte)$;
init $CommandeVerre$;

On constate qu'il nous est possible de représenter des combinaisons de processus et d'actions simples.

Axiomes

Les axiomes aident essentiellement à interpréter correctement l'algèbre. La sémantique de la langue est donnée en termes d'algèbre de données et de sémantique opérationnelle. Elle permet une compréhension plutôt opérationnelle. Une des raisons d'avoir ces axiomes est qu'ils constituent la base élémentaire du raisonnement axiomatique sur les processus [16]. La table 1.1 ci-dessous indique les axiomes utilisables pour cette version.

Table 2.1 – Les axiomes pour mCRL2 version 1

A1	x + y = y + x
A2	(x + y) + z = x + (y + z)
A3	x + x = x
A4	(xy) z = x (yz)
A5	(x + y) z = xz + yz

Pour bien comprendre les axiomes, il est nécessaire de parler de bisimilarité. La bisimilarité est une équivalence comportementale [17]. Elle signifie que deux comportements de processus sont identiques en termes d'actions mais aussi en termes de structure de branchements [17]. Pour bien le comprendre, prenons $P \xrightarrow{\mu} P'$, si dans une situation, P offre une interaction μ que ne peut fournir Q, il est naturel de considérer que le comportement de P est différent de celui de Q [17]. L'exemple de la machine à café dans la section intuition peut nous aider à le comprendre :

$$P1 = c5.(prepareTea + prepareCafee)$$
$$P2 = c5.prepareTea + c5.prepareCafee$$

Si c5 est exécuté dans P1 et dans P2, dans P1 il sera simplement exécuté nous laissant le choix entre prepareTea et prepareCafee. Par contre, dans P2 nous allons devoir faire un choix arbitraire entre prepareTea et prepareCafee. On en conclut donc que les processus P1 et P2 ne sont pas bisimilaires.

Comme défini dans le document [18] : "une relation $\mathcal{R}$ entre processus est une bisimulation si et seulement si $P\,\mathcal{R}\,Q$ et $P \xrightarrow{\mu} P'$, il existe Q' tel que $Q \xrightarrow{\mu} Q'$ et $P'\,\mathcal{R}\,Q'$, et symétriquement pour les transitions de Q" [18].

Les axiomes ci-dessus garantissent la bisimilarité.

On constate que la table reprise ci-dessus exprime les propriétés élémentaires des opérateurs de compositions séquentielles et alternatives. En analysant A1, A2 et A3, on en retire que l'opérateur '+' est commutatif, associatif et idempotent [16]. L'axiome A4 nous montre le côté associatif de '.' tandis que A5 démontre que la distributivité simple est d'application pour l'opérateur séquentiel.

Cette table peut bien entendu s'agrandir au fur et à mesure de nos versions successives, au fur et à mesure que des opérateurs seront présentés.

Limite

Cette version ajoute un plus à notre version mCRL0 : la représentation de processus simple. Pour mettre en évidence l'amélioration de notre version 1, représentons notre exemple qui limitait notre version 0. Pour rappel, cette représentation algébrique était celle d'une commande. Dans ce processus de commande, l'utilisateur doit indiquer à la machine la commande souhaitée. Une fois la commande indiquée, la machine prévient le responsable du magasin afin qu'il prépare la commande du client. Lorsque la commande est prête l'utilisateur peut procéder au paiement sur la machine par carte ou par liquide. Notre version 1 nous permet facilement de résoudre ce problème :

$$\begin{array}{l}\textbf{act}\ transmettreCommande, attendrePreparationCommande,\\ \quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementLiquide;\\ \textbf{proc}\ Machine = transmettreCommande.attendrePreparationCommande;\\ \quad demanderPaiement.(recevoirPaiementLiquide + recevoirPaiementLiquide);\\ \textbf{init}\ \ Machine;\end{array}$$

Améliorons notre principe de commande pour offrir au client la possibilité de payer pendant la préparation de la commande. Pour l'instant, le client paie une fois la commande préparée et sur demande de la machine. Cette fois, le client peut choisir de payer à tout moment après avoir passé commande. Du point de vue de la machine, elle doit pouvoir transmettre la commande, attendre la préparation de la commande et en parallèle être capable de gérer le paiement de cette commande.

On constate qu'il nous manque un opérateur. Un opérateur permettant de définir des comportements parallèles :

$$\begin{array}{l}\textbf{act}\ transmettreCommande, attendrePreparationCommande,\\ \quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementLiquide;\\ \textbf{proc}\ Machine = (transmettreCommande.attendrePreparationCommande)???\\ \quad (recevoirPaiementLiquide + recevoirPaiementLiquide);\\ \textbf{init}\ \ Machine;\end{array}$$

Le manque de possibilité de représentation de processus parallèles est une des faiblesses de notre version. Notre version 2 spécifie les comportements parallèles qui sont essentiels dans la spécification des processus.

2.5.3 mCRL2 - Modèle avec composition parallèle

Outre les opérateurs de base, qui sont généralement utilisés pour spécifier le comportement des composants principaux du système, notre langage doit pouvoir disposer d'une composition parallèle pour composer les processus et/ou les actions [19]. Dans le langage mCRL2, il en existe 3 types :

1. L'opérateur de composition parallèle (A ‖ B)
2. L'opérateur de synchronisation (A | B)
3. L'opérateur d'entrelacement gauche (A ‖ B)

Cette section présentera ces parallélismes. Ils seront illustrés d'exemples afin d'aider à leur compréhension. Ces nouvelles compositions nous permettent d'ajouter des axiomes supplémentaires par rapport à la version 1. Ils seront listés en fin de section. Cette version nous permet de décrire la plupart des processus même si des limites seront présentées en fin de chapitre.

L'opérateur de composition parallèle ‖

La composition parallèle entrelace et synchronise les actions ou processus de gauche avec ceux de droite. Utiliser cet opérateur implique qu'aucune notion d'ordre n'existe au sein de ses membres et donc qu'il n'y a aucun point de synchronisation entre eux pendant leur exécution [15]. Si on prend l'exemple de a ‖ b, on peut avoir l'exécution de l'action a suivie de celle de b, l'inverse : b puis a ou même l'exécution des deux actions en même temps [15, 2].

Par exemple, le tableau suivant représente une lecture et une écriture parallèle :

act	$ecrire, lire;$
proc	$Processus = ecrire \parallel lire;$
init	$Processus;$

L'exemple suivant est totalement équivalent au précédent :

act	$ecrire, lire;$
proc	$Processus = ecrire.lire + lire.ecrire + lire \mid ecrire;$
init	$Processus;$

On en déduit que ecrire$\parallel$ lire = ecrire . lire + lire . ecrire + lire | ecrire. L'opérateur | est présenté dans la section suivante.

L'opérateur de synchronisation |

Les actions qui peuvent se produire simultanément s'appellent multi-actions. Une multi-action est une séquence d'actions séparées par des barres. Par exemple, a | b | z. L'opérateur | synchronise la première action de son membre de gauche avec celle de son membre de droite tout en combinant le reste de son membre de gauche et de droite comme la composition parallèle [15]. Si on prend l'exemple A | B, la première action de a et b commenceront en même temps pour ensuite exécuter les actions suivantes comme l'opérateur $\parallel$.

Par exemple, pour définir qu'il faut obligatoirement lire et écrire en même temps il faut utiliser l'opérateur de synchronisation :

act	$ecrire, lire;$
proc	$Processus = ecrire \mid lire\ ;$
init	$Processus;$

L'opérateur d'entrelacement gauche $\lfloor\!\lfloor$

L'opérateur d'entrelacement de gauche permet à son membre de gauche d'exécuter une première action et d'ensuite combiner le reste de son membre de gauche avec son membre de droite comme l'opérateur parallèle [15]. Il est notamment utile pour permettre l'axiomatisation de la composition parallèle. Par exemple :

act	$ecrire, lire, enregistrer, envoyer;$
proc	$Processus = (ecrire.enregistrer) \lfloor\!\lfloor (lire.envoyer)\ ;$
init	$Processus;$

Indique que l'action écrire sera toujours exécutée en premier lieu et qu'enregistrer, lire et envoyer seront ensuite exécutés par l'opérateur parallèle [15].

Les axiomes

La figure 2.2 ci-dessous indique les axiomes supplémentaires par rapport à la version 1.

FIGURE 2.2 – Les axiomes supplémentaires pour la version 2 [2].

M1	$s\|\|t = (s\lfloor\lfloor t + t\lfloor\lfloor s) + s\|t$
LM2	$v\lfloor\lfloor y = v \cdot y$
LM3	$(v \cdot x)\lfloor\lfloor y = v \cdot (x\|\|y)$
LM4	$(x + y)\lfloor\lfloor z = x\lfloor\lfloor z + y\lfloor\lfloor z$
CM5	$v\|w = \gamma(v, w)$
CM6	$v\|(w \cdot y) = \gamma(v, w) \cdot y$
CM7	$(v \cdot x)\|w = \gamma(v, w) \cdot x$
CM8	$(v \cdot x)\|(w \cdot y) = \gamma(v, w) \cdot (x\|\|y)$
CM9	$(x + y)\|z = x\|z + y\|z$
CM10	$x\|(y + z) = x\|y + x\|z$

L'axiome M1 nous rappelle que l'opérateur parallèle peut être décomposé en s puis t ou en t puis s ou même les deux en même temps. Les axiomes LM2 et LM3 définissent l'opérateur d'entrelacement gauche en démontrant qu'il est égal à l'exécution de la première action de son membre de gauche et ensuite de son membre de droite comme l'opérateur parallèle. L'axiome LM4 démontre la distributivité simple droite de l'opérateur de l'entrelacement gauche [2]. Les axiomes CM5, CM6, CM7 et CM8 démontrent la synchronisation de l'opérateur de synchronisation. L'axiome CM9 illustre la distributivité à droite de z et l'axiome CM10 illustre la distributivité à gauche de l'opérateur | [2].

Limite

Dans la version mCRL2 version 2, l'exemple suivant a été mentionné : représentation d'une préparation de commande avec possibilité de paiement pendant la préparation. Pour rappel, on avait suggéré que le client puisse choisir de payer à tout moment après avoir passé commande. Du point de vue de la machine, elle doit pouvoir transmettre la commande, attendre la préparation de la commande et en parallèle être capable de gérer le paiement de cette commande. On constate qu'avec l'ajout du parallélisme, cette représentation n'est plus un problème :

act $transmettreCommande, attendrePreparationCommande,$
$demanderPaiement, recevoirPaiementLiquide, recevoirPaiementLiquide;$
proc $Machine = (transmettreCommande.attendrePreparationCommande)||$
$(recevoirPaiementLiquide + recevoirPaiementLiquide);$
init $Machine;$

Améliorons notre exemple précédent pour faire ressortir les limites de notre version 2. Jusqu'ici, on ne spécifiait que les actions de la machine. On aimerait également s'intéresser aux actions de l'utilisateur. Un utilisateur remplit un formulaire de commande et clique sur un bouton enclenchant l'envoi de la commande à la machine. Lorsqu'un utilisateur insère sa carte, la machine lit la carte et procède au retrait par carte. En partant du principe que la machine et l'utilisateur sont deux processus indépendants, on pourrait représenter naïvement cet enchaînement de la manière suivante :

act $transmettreCommande, attendrePreparationCommande,$
$demanderPaiement, recevoirPaiementLiquide, recevoirPaiementCarte$
$remplirFormulaire, envoyerCommande, insererCarte, lireCarte;$
proc $Machine = (transmettreCommande.attendrePreparationCommande)||$
$((lireCarte.recevoirPaiementCarte) + recevoirPaiementLiquide);$
$Utilisateur = remplirFormulaire.envoyerCommande.insererCarte;$
init $Utilisateur||Machine;$

Cette solution n'est pas correcte étant donné que l'opérateur $\|$ entre Utilisateur et Machine indique que le processus Machine pourrait être terminé avant le processus Utilisateur. Quand bien même, cette solution ne synchronise pas les actions de l'utilisateur avec celle de la machine. L'opération lireCarte ne peut se produire seule. Seule une carte insérée par l'utilisateur peut la déclencher. A nouveau cette solution ne spécifie pas cette contrainte ; l'action lire pourrait se produire avant l'insertion de la carte. Ces deux actions ne peuvent se produire qu'ensemble, il est impossible de les exécuter séparément.

Ces problèmes constituent la limite de notre version 2. Dans la version 3, nous pallierons ce problème à l'aide de l'opérateur communication et allow. Cette version présentera également un moyen de spécifier les interdits, les interruptions ou les blocages.

2.5.4 mCRL2 - Modèle avec encapsulation

La version 3 a pour but d'introduire les opérateurs communication, allow, delta et encapsulation. Elle décrit communication et allow d'abord pour ensuite expliquer ce qu'est delta et encapsulation. Les axiomes de communication, allow, delta et encapsulation sont listés et les limites de cette version sont également exposées.

Communication et allow

Dans la version précédente, on a constaté qu'une composition parallèle $a \parallel b$ stipule que l'action a pu se dérouler avant l'action b ou que l'action b peut s'exécuter avant l'action a ou même que les deux peuvent se dérouler en même temps. Sa représentation peut alors s'écrire de la façon suivante :

$$a.b + b.a + a|b$$

Le nombre d'entrelacements possibles s'accroit fortement en augmentant les termes et ce de manière combinatoire [19]. Prenons, par exemple $a.b \parallel c.d$ sa représentation est :

$$a.(b.c.d + b|c.d + c.(b.d + d.b + b|d)) + c.(d.a.b + a|d.b + a.(b.d + d.b + b|d)) + (a|c).(b.d + d.b + b|d)$$

[19]

La Communication nous permet de réduire ce nombre de possibilités en spécifiant que certaines actions ne peuvent s'exécuter seules [15]. Effectivement, la plupart du temps, lorsque deux actions sont prévues pour communiquer ensemble, il n'est pas prévu de les exécuter individuellement [19]. Si on prend l'exemple d'un processus de transfert de données qui envoie à l'aide de l'action "envoyer" et qui lit à l'aide de l'action "lire" [15]. Le monde extérieur pourrait le voir comme une et une seule action de communication, comme l'action transmettre. Ce monde extérieur simplifie nos actions en stipulant qu'une donnée envoyée est obligatoirement une donnée reçue.

Pour parvenir à synchroniser l'action "envoyer" avec l'action "lire", il est obligatoire d'interdire l'exécution individuelle des actions. Pour y parvenir, il existe deux opérateurs : comm et allow.

L'opérateur $comm(\{lire|envoyer \rightarrow transmettre\}, processus)$ indique que les multi-actions lire et envoyer sont renommées en une seule action. Il indique que les actions lire et envoyer doivent communiquer dans le processus "processus". Concrètement, dans toute multi-action de processus, toutes les occurrences de $lire|envoyer$ sont remplacées par transmettre.

L'opérateur "allow" indique les actions autorisées en son sein. Par exemple, $allow(\{a, b\}, a + b + c)$ n'autorise l'exécution que de 'a' ou 'b', 'c' n'étant pas indiqué dans "allow", il n'est pas exécutable. Cet opérateur est également utile lorsqu'on le mélange à communication. En spécifiant, $allow(\{transmettre\}, comm(\{lire|envoyer \rightarrow transmettre\}, lire||envoyer)$ indique que seules les multi-actions consistant en un seul "transmettre" sont autorisées dans "processus". Toutes les autres actions sont bloquées obligeant l'utilisation de lire et d'ensuite envoyer.

Il est donc possible de spécifier la transmission énoncée en début de section de la manière suivante :

act	$lire, envoyer;$
init	$allow(\{transmettre\}, comm(\{lire\|envoyer \rightarrow transmettre\}, lire\|\|envoyer));$

Delta et encapsulation

Il est possible d'interdire une communication à l'aide de l'opérateur delta. Delta exprime : une erreur, une interruption ou une communication interdite.

Un opérateur important souvent utilisé pour exprimer le blocage est l'opérateur d'encapsulation qui permet de bloquer des actions individuelles. Si une action unique bloquée se produit dans une action multiple, l'ensemble de l'action multiple est renommée delta. Par exemple, block({b}, a ‖ b) est égale à a. delta. Et l'expression

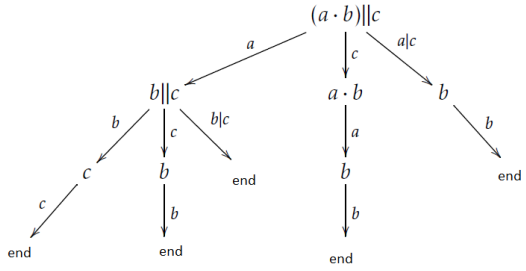
act	$a, b;$
init	$delta + a.b;$

est égale a :

act	$a, b;$
init	$a.b;$

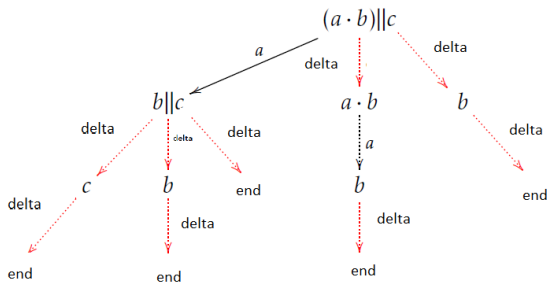
Pour bien comprendre ces deux opérateurs prenons l'exemple de l'expression (a . b)‖ c. Ce dernier peut être représenté selon un arbre comme à la figure : 2.3

FIGURE 2.3 – Tree representation of (a . b)‖ c [2].



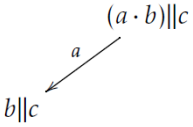
Si on décide maintenant d'utiliser l'expression : block({b, c}, a . b ‖ c) les actions b et c vont être remplacées par delta. L'arbre se représentera alors comme à la figure 2.4.

FIGURE 2.4 – Représentation en arbre de block({b, c}, a . b ‖ c) [2].



Delta signifiant des chemins bloquants, il est possible d'élaguer l'arbre en supprimant les chemins qui possèdent delta à leur racine. L'arbre simplifié se représentera alors comme à la figure 2.5.

FIGURE 2.5 – Représentation en arbre de block({b, c}, a . b ‖ c) simplifié [2].



Axiomes

La figure 2.6 ci-dessous indique les axiomes supplémentaires de la version 3 par rapport à la version 2. Pour bien comprendre la figure, il faut considérer que $\delta = delta$ et que ∂H = encapsulation

FIGURE 2.6 – Les axiomes supplémentaires introduits dans la version 3 [2].

TABLE 2.4 – Axiômes pour ACP_2

A6		$x + \delta = x$
A7		$\delta \cdot x = \delta$
D1	$v \notin H$	$\partial_H(v) = v$
D2	$v \in H$	$\partial_H(v) = \delta$
D3		$\partial_H(\delta) = \delta$
D4		$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
D5		$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
LM11		$\delta \lfloor\lfloor x = \delta$
CM12		$\delta \lfloor x = \delta$
CM13		$x \lfloor \delta = \delta$

Notre opérateur séquentiel et alternatif se voit enrichi de A6 et A7. L'axiome A6 démontre que delta n'est jamais une alternative en spécifiant que delta n'a aucun comportement. Il nous démontre le coté bloquant de delta en indiquant dans A7 que rien ne peut être exécuté après delta. Cette constatation est renforcée par LM11 qui stipule le comportement de delta avec l'opérateur d'entrelacement gauche. Les axiomes CM12 et CM13 ajoutent des précisions à la constatation que rien ne peut être exécuté en même temps et après delta. Les axiomes D1 et D3 nous montrent que l'opérateur d'encapsulation ne change ni delta, ni les actions non stipulées dans H. L'axiome D2 prouve le lien entre encapsulation et delta en confirmant qu'encapsulation est semblable à renommer toutes les actions stipulées dans H en delta.

Limite

L'exemple de la section précédente peut maintenant être résolu à l'aide de l'opérateur comm et allow :

$$
\begin{array}{l}
\textbf{act}\ transmettreCommande, attendrePreparationCommande, initierPayement, commander \\
\quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementCarte \\
\quad remplirFormulaire, envoyerCommande, insererCarte, lireCarte; \\
\textbf{proc}\ Machine = ((transmettreCommande.attendrePreparationCommande)|| \\
\quad ((lireCarte.recevoirPaiementCarte) + recevoirPaiementLiquide)); \\
\quad Utilisateur = remplirFormulaire.envoyerCommande.insererCarte; \\
\textbf{init}\ \ allow(\{commander, initierPayement, attendrePreparationCommande, initierPayement, \\
\quad commander, demanderPaiement, recevoirPaiementLiquide, recevoirPaiementCarte, \\
\quad remplirFormulaire, recevoirPaiementCarte\}, \\
\quad comm(\{envoyerCommande|transmettreCommande \rightarrow commander, \\
\quad insererCarte|lireCarte \rightarrow initierPayement\}, Utilisateur||Machine));
\end{array}
$$

On constate bien que, grâce à l'opérateur allow et comm, les actions insererCarte et lireCarte fonctionnent ensemble, les deux actions étant totalement synchronisées l'une avec l'autre. Il en va de même pour envoyerCommande et transmettreCommand.

En termes de représentation algébrique de processus fini, cette version est assez complète. Pour l'instant, notre processus de commande ne traite qu'une seule commande. Après cette première commande la machine est arrêtée. Cependant, le souhait n'est pas qu'elle gère une seule commande mais qu'elle puisse reproduire cette procédure plusieurs fois. Cette fois-ci, la limite se pose à cause d'un manque de comportement de récursivité. La version suivante essaye d'amener ce comportement.

2.5.5 mCRL2 - Modèle avec récursivité

La version précédente ne parle que des processus finis. Dans cette section, nous allons exposer que la récursivité peut nous permettre d'exprimer des comportements illimités. Ensuite, on parlera de sa représentation en mCRL2. Pour clôturer, les axiomes de la récursion sont présentés.

Recursion

Un processus pourrait répéter des actions à l'infini. Par exemple, prenons un processus A qui exécute une action y et que l'exécution de cette action y le fait tomber dans un état B, qui lui-même exécute un action z qui le fait retomber dans l'état A, ... Cet exemple pourrait être représenté de la manière suivante :

$$A \xrightarrow{y} B \xrightarrow{z} A \xrightarrow{y} B \xrightarrow{z} A...$$

De manière intuitive, on constate que pour sortir de l'état A, il faut exécuter l'action y qui nous place dans l'état B. On constate également que pour quitter l'état B, il faut exécuter l'action z qui nous replace dans l'état A. Sur base de ces constations, on peut dresser les deux équations récursives suivantes :

$$A = y.B$$

$$B = z.A$$

Exemple en mCRL2

En reprenant l'exemple de la machine à café, prenons le processus simple où la machine reçoit une pièce et distribue alors un café. Ensuite, elle attend qu'une nouvelle pièce soit introduite pour fournir à nouveau un café

et ce de manière infinie. Avec les bases exprimées dans les versions précédentes, on le représenterait de la façon suivante :

act	$piece, cafee;$
proc	$Process = piece.cafee$;
init	$Process;$

Si on traduit cette algèbre de manière textuelle on dirait que la machine reçoit une pièce, ensuite distribue un café puis ne fait plus rien. Cette traduction n'est pas vraiment celle que l'on souhaite exprimer. Pour pallier ce problème, on peut utiliser nos connaissances de la section précédente et transformer cette algèbre en :

act	$piece, cafee;$
proc	$ProcessCafee = piece.ProcessPiece$
proc	$ProcessPiece = piece.ProcessCafee$
init	$ProcessPiece;$

Cependant, ce n'est pas la seule manière d'exprimer cette récursivité. Une autre façon de l'exprimer est de rappeler son propre processus :

act	$piece, cafee;$
proc	$Process = piece.cafee.Process$
init	$Process;$

Dans nos deux exemples précédents on constate que la signification voulue est bien respectée.

Axiomes

La récursion nous permet d'ajouter des axiomes supplémentaires à notre liste. Un concept important de ces axiomes est celui d'équation gardée. Une équation est gardée si chaque occurrence d'une variable de récursion dans la partie droite est précédée d'une occurrence d'une action. Par exemple, l'équation X = a + b . Y est gardée car, lorsque l'on accède à ce X, il faut passer soit par le garde a, soit par le garde b. Au contraire, l'équation : X = X + a.X n'est pas gardée.

Ces axiomes sont représentés sur la figure 2.7

FIGURE 2.7 – Axiomes récursifs

RDP	$\langle X_i \mid E\rangle = t_i(\langle X_1 \mid E\rangle, ..., \langle X_n \mid E\rangle)$	$(i \in \{1, ..., n\})$
RSP	Si $y_i = t_i(y_1, ..., y_n)$, alors $y_i = \langle X_i \mid E\rangle$	$(i \in \{1, ..., n\})$

Le principe de définition récursive restreinte (RDP) exprime l'hypothèse selon laquelle toute spécification récursive gardée a minimum une solution [20]. Le principe de spécification récursive (RSP) est l'hypothèse selon laquelle chaque spécification récursive gardée a maximum une solution [20].

Limite

Cette version nous ouvre des possibilités supplémentaires pour représentation de nos processus. Notre processus de commande peut maintenant être peaufiné pour affirmer que notre processus se répète de manière infinie.

$$
\begin{array}{l}
\textbf{act}\ transmettreCommande, attendrePreparationCommande, initierPayement, commander \\
\quad demanderPaiement, recevoirPaiementLiquide, recevoirPaiementCarte \\
\quad remplirFormulaire, envoyerCommande, insererCarte, lireCarte; \\
\textbf{proc}\ Machine = ((transmettreCommande.attendrePreparationCommande)|| \\
\quad ((lireCarte.recevoirPaiementCarte) + recevoirPaiementLiquide)).Machine; \\
\quad Utilisateur = remplirFormulaire.envoyerCommande.insererCarte.Utilisateur; \\
\textbf{init}\ \ allow(\{commander, initierPayement, attendrePreparationCommande, initierPayement, \\
\quad commander, demanderPaiement, recevoirPaiementLiquide, recevoirPaiementCarte, \\
\quad remplirFormulaire, recevoirPaiementCarte\}, \\
\quad comm(\{envoyerCommande|transmettreCommande \rightarrow commander, \\
\quad insererCarte|lireCarte \rightarrow initierPayement\}, Utilisateur||Machine));
\end{array}
$$

Cependant, on constate que plus notre processus évolue plus il est difficile à comprendre. Cette augmentation du nombre d'actions et du nombre de processus a un impact direct sur la lisibilité et la compréhension. Dans notre exemple, il pourrait être utile d'abstraire, de cacher certaines actions pour pouvoir se concentrer sur d'autres. Cette faiblesse de notre version pourrait nous poser des problèmes lors de l'analyse de flux massif. La version suivante vient introduire la notion d'abstraction par l'utilisation de l'action silencieuse pour répondre à cette problématique.

2.5.6 mCRL2 - Modèle avec opérateur d'abstraction

Cette section permet de comprendre la notion d'abstraction en introduisant l'action silencieuse. Un exemple concret de l'abstraction en langage mCRL2 est présenté. Cette nouvelle notion d'abstraction nous permet d'intégrer de nouveaux axiomes et de clôturer nos sous-versions de mCRL2.

Action silencieuse

Une question fondamentale dans la conception et la spécification des systèmes hiérarchiques de processus de communication est celle de l'abstraction. Sans ce mécanisme d'abstraction nous permettant de faire abstraction du fonctionnement interne des modules à composer pour obtenir des systèmes plus grands, la spécification de tous les systèmes, sauf les très petits, serait pratiquement impossible [1].

Si on prend l'exemple de notre commande et que l'on décide d'y ajouter que la carte de paiement est une carte du magasin sur laquelle on peut verser de l'argent pour payer, la puce de cette carte enfoncée dans notre machine, va être lue et dans un premier temps un appel va être fait pour vérifier que le solde du compte de la carte est suffisant. Ensuite, une seconde vérification sera faite pour vérifier l'état des points collectés par le client ; à partir de 500 points, une remise de 20% sur la commande est octroyée. Une fois toutes ces vérifications effectuées, le retrait de l'argent est effectué sur la carte. Si je m'intéresse par exemple au flux exécuté par la commande, ces détails peuvent vite rendre illisible la spécification. Cependant, comment pouvons-nous faire abstraction de tels détails internes si nous ne nous intéressons qu'au comportement externe. La première étape pour obtenir une telle abstraction est de supprimer l'identité distinctive des actions à abstraire, c'est-à-dire de les renommer toutes en une seule action dissociée que nous appelons, d'après Milner, τ : l'action silencieuse [1].

La deuxième étape consiste à tenter de concevoir des axiomes pour l'étape silencieuse τ, au moyen desquels τ peut être retirée de l'expression, comme dans l'équation $a\tau b = ab$.

Cependant, il n'est pas possible de supprimer tous les τ dans une expression si l'on s'intéresse à une description fidèle du comportement de blocage des processus [1]. Milner a conçu quelques axiomes simples qui peuvent

être utilisés pour donner une description complète des propriétés du pas silencieux (voir la section axiomes) [1].

Abstraction en mCRL2

Le mot clé *hide* permet d'abstraire des actions en mCRL2. Par exemple, $hide(\{a\}, a|b)$ est égale à b étant donné que a est "caché". Ce masquage, cette abstraction, permet d'indiquer que certaines actions ne peuvent plus être observées par le monde extérieur.

Pour bien comprendre, on peut prendre la spécification suivante :

act	$creer, recevoir, transmettre, lire, envoyer;$
init	$creer.lire.envoyer.recevoir;$

On pourrait ne pas voir d'intérêt à montrer l'enchainement d'action lire et envoyer à l'utilisateur. L'utilisateur ne verrait alors que créer puis recevoir. On peut cacher ces actions en modifiant la spécification de la manière suivante :

act	$creer, recevoir, transmettre, lire, envoyer;$
init	$hide(\{lire, envoyer\}, creer.lire.envoyer.recevoir);$

Axiomes

L'abstraction nous permet d'ajouter des axiomes supplémentaires à notre liste. Ces axiomes sont représentés à la figure 2.8

FIGURE 2.8 – Axiomes pour la version 5

B1	$v \cdot \tau = v$
B2	$v \cdot (\tau \cdot (x+y)+x) = v \cdot (x+y)$

2.6 Conclusion

Ce chapitre a permis de définir l'algèbre des processus comme le domaine qui étudie le comportement des systèmes parallèles ou distribués par des moyens algébriques [6]. On a constaté que l'algèbre des processus est un bon moyen pour exprimer des comportements parallèle, séquentiel et alternatif [5]. De plus, il constitue une approche avec une syntaxe et une sémantique simple pour la formalisation, la réflexion et l'analyse de processus asynchrones. Son histoire nous montre une évolution rapide de l'intérêt des personnes envers ce domaine. Elle nous apprend également que deux changements ont été importants dans son développement : se rendre compte des effets de bord des processus et se rendre compte de la difficulté de détermination des valeurs des variables globales des processus parallèles. Elle démontre une envie d'amélioration du mcrl en créant le mCRL2. Le mCRL2 a été explicité au travers ce chapitre. On a pu voir sa puissance dans la spécification de divers comportements : alternatif, parallèle et bien d'autres.

Nous avons maintenant une bonne image de ce qu'est le mCRL2, de sa puissance, son histoire, sa syntaxe ainsi que de sa sémantique. Ces apprentissages permettent de comprendre les chapitres suivants qui concernent la mise en place d'un atelier d'algèbre des processus en mCRL2.

Chapitre 3

Aspect fonctionnel de ProCalg V2

3.1 Introduction

Ce chapitre commence par présenter différents outils de visualisation existants dans mCRL2. La présentation de ces outils comporte pour chacun leur fonctionnement, leur limite et la description des problèmes rencontrés durant leur exécution. La compréhension de ces outils a permis de tracer les différents objectifs de l'outil créé dans le cadre de ce mémoire. Un des problèmes récurrents des applications mCRL2 est le passage par la forme linéaire qui modifie la spécification entrée par l'utilisateur. Ce problème a été résolu par un étudiant sous la responsabilité de Mr Jacquet promoteur également de ce mémoire. Son application, au même titre que les autres, est décrit avec une recherche des limites et des problèmes durant l'exécution dans le but de fixer de nouveaux objectifs pour l'application de ce mémoire. La présentation de son travail nous réconforte dans l'idée que la forme linéaire apporte des inconvénients pour la visualisation et l'exécution malgré sa facilité de parcours.

Après ces diverses constatations des outils existants, les objectifs de ProCalg v2 sont mis à plat. Ses fonctionnalités sont ensuite présentées en l'état de l'application du 11-05-2021. L'outil est ensuite vérifié afin de s'assurer qu'il vient pallier les problèmes rencontrés dans les autres applications. Une réflexion par rapport au logiciel est présentée en fin de chapitre. Cette réflexion décrit les manquements, les faiblesses et exprime des idées d'améliorations.

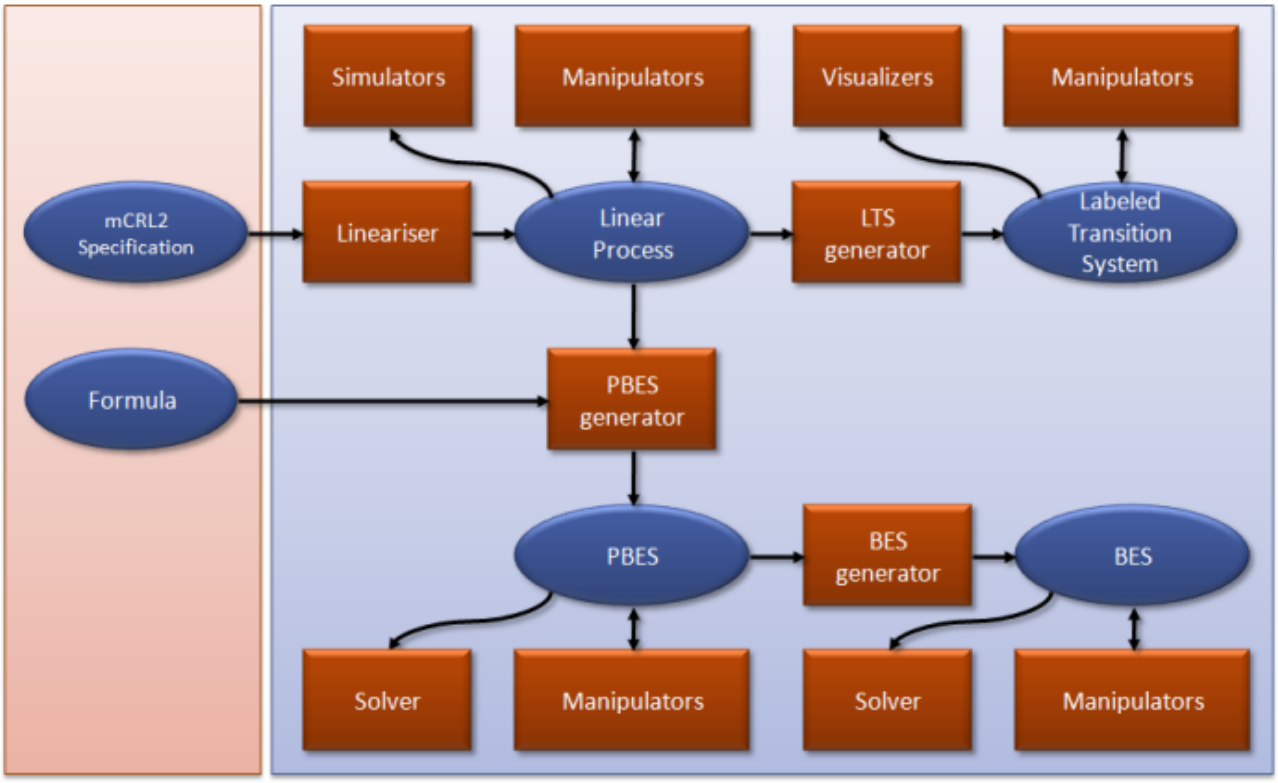
3.2 Présentation des outils existants mCRL2

3.2.1 Vue d'ensemble

Le mCRL2 possède une boite à outils téléchargeable sur le site officiel [21]. Ces outils aident à la lecture, à la manipulation, à l'analyse, à la visualisation et aux traitements de spécifications mCRL2.

Le schéma 3.1 récapitule les concepts principaux ainsi que les opérations que l'on peut faire à l'aide de cette boite à outils. En bleu, on retrouve les principaux concepts. En rouge, on remarque les opérations.

FIGURE 3.1 – Schéma récapitulatif des outils mCRL2 [22].



Le schéma nous indique la possibilité sur base d'une spécification mCRL2 de la transformer sous forme linéaire. En partant de cette forme linéaire, il est possible de faire de la simulation et de la manipulation de processus.

Elle peut être utilisée pour créer un système de transition labellisée. Ce système de transition met en avant les transitions du système. Elle est utile pour visualiser les transitions possibles entre chaque action d'une spécification.

En repartant de la forme linéaire, il est possible de générer un système d'équations booléennes paramétrées (PBES). Ce PBES est décrit dans la sous-section "PBES". Il est utile pour coder les problèmes de vérification de modèles mais aussi pour coder l'équivalence sur les processus avec données.

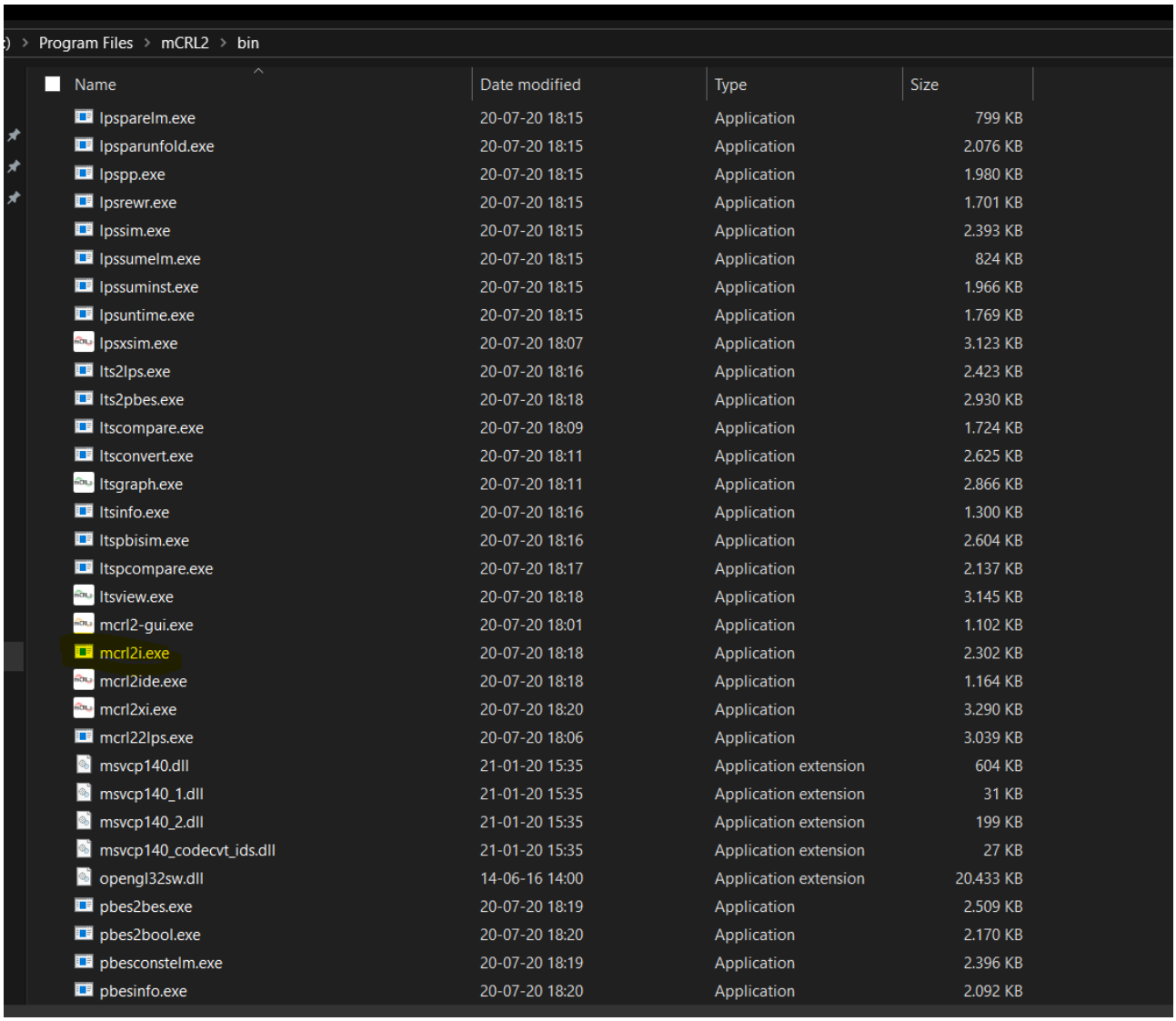
Il est possible de les mettre sous forme de système d'équation booléenne, qui est un sous-ensemble des systèmes PBES contenant des caractéristiques propres.

Les outils permettant de faire de la forme linéaire, libellé et d'équation booléenne sont présentés en détail dans les sections suivantes. Ces sections permettent de faire un zoom sur les différents éléments de cette vue d'ensemble.

3.2.2 Tutoriel de lancement des outils mCRL2

Mcrl2 offre deux possibilités pour lancer ses outils. La première est via un terminal de commande. Pour ce faire, il faut se rendre dans mCRL2/bin. Sa localisation dépend du système d'exploitation et de l'installation. La figure 3.2 montre en jaune l'exécutable mcrl2i.exe à lancer sur Windows.

FIGURE 3.2 – Executable du terminal de commande mCRL2 [22].



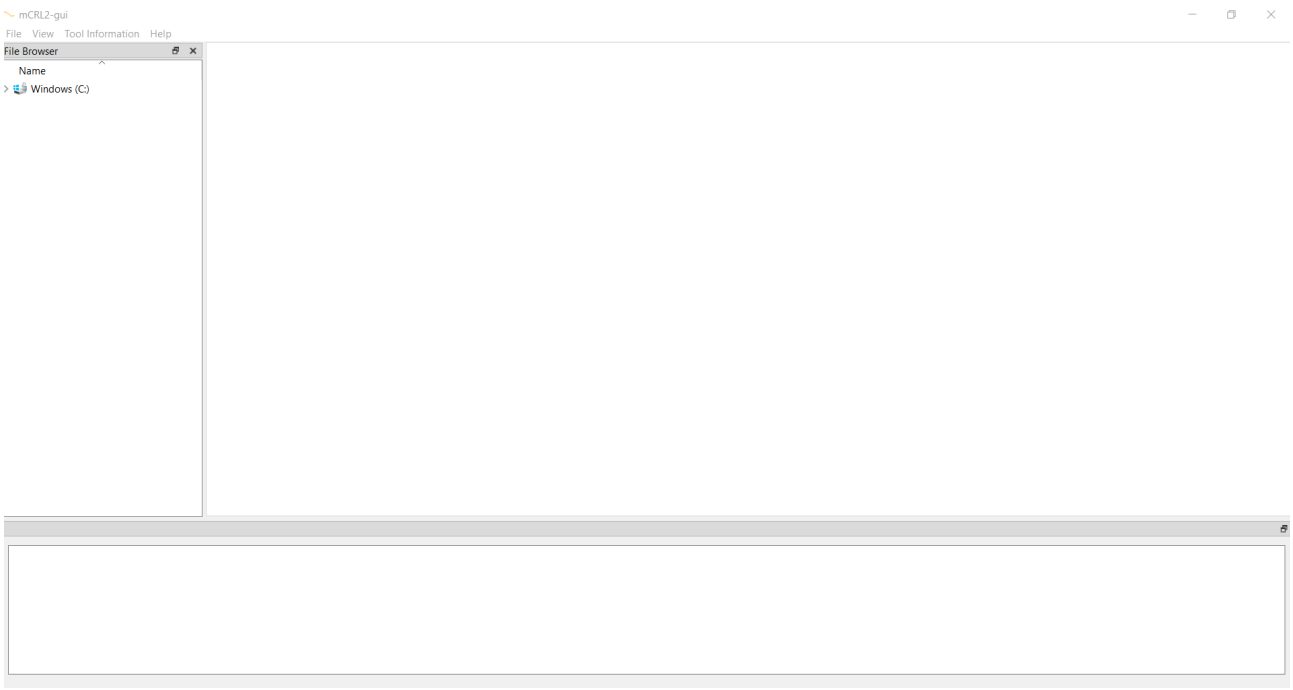
De l'aide est fournie en insérant h dans la console. Le message indiqué par le terminal est visible sur la figure 3.3.

FIGURE 3.3 – Terminal mcrl2i.

Ce terminal est difficile à prendre en main et n'est pas super agréable d'utilisation.

Une seconde possibilité est d'utiliser l'interface graphique fournie par mCRL2. Pour le lancer, il suffit de lancer l'exécutable mcrl2-gui.exe situé dans le même répertoire que mcrl2i.exe. La figure 3.4 montre le premier écran visible lors du lancement de l'interface graphique.

FIGURE 3.4 – Executable du terminal de commande mCRL2.



La section file browser liste les fichiers du système d'exploitation. Lorsqu'un fichier possède une extension compatible avec un des outils mCRL2, l'interface graphique donne la possibilité d'exécuter différents outils au moyen d'un clic droit sur le fichier. Par exemple, si on crée un fichier avec l'extension .mcrl2, en effectuant un clic droit sur le fichier, un menu apparaît. Ce menu propose notamment l'édition du fichier à l'aide de l'outil mcrl2xi (voir figure 3.5). Le lancement de cet outil permet d'obtenir un éditeur de mCRL2 affiché à la figure 3.6.

FIGURE 3.5 – Menu pour le lancement d'outil mCRL2.

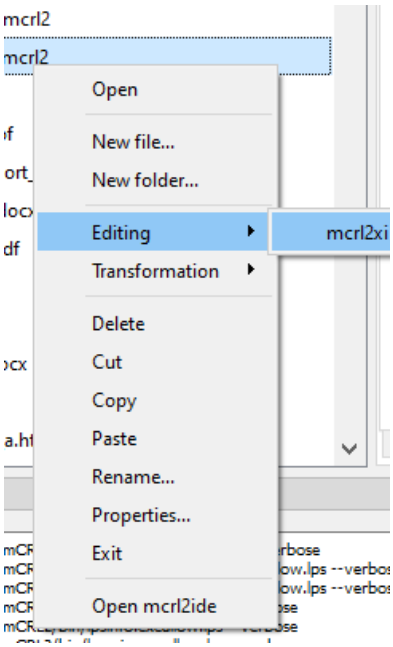
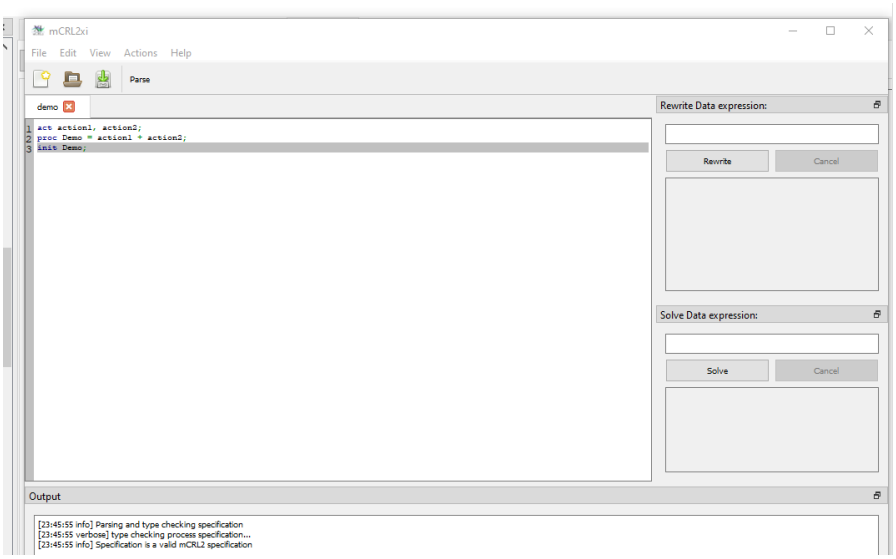


FIGURE 3.6 – Fenêtre de l'outil Mcrl2xi.



Cette interface est bien plus conviviale que le terminal et permet fortement de simplifier la prise en main. Dans les sous-sections suivantes, cette interface homme-machine est utilisée pour lancer les outils.

3.2.3 La linéarisation

Qu'est-ce que la linéarisation ?

A la figure 3.1, on constate l'importance de la linéarisation : elle est au centre de tout. Ce schéma prouve que la plupart des outils de la boîte à outils mCRL2 se basent sur des spécifications linéaires de processus. La spécification d'un système contient souvent plusieurs processus pouvant être exécutés en parallèle. Il est plus facile de construire des outils sous une forme linéaire que sous forme brute. Elle possède une forme plus facile à traiter pour l'analyse automatisée. Effectivement, la linéarisation est finalement le résultat de la suppression du parallélisme de notre spécification algébrique. Le traitement d'un enchaînement linéaire est plus simple à traiter que l'enchaînement de processus parallèle. Elle contient des règles condition - action - effet qui indiquent au lecteur comment le système réagit dans son ensemble à un certain stimuli.

Le principal outil de linéarisation est mcrl22lps. Il permet, en prenant en entrée une spécification mCRL2, de fournir la spécification de processus linéaire (LPS) correspondant.

Il supporte les actions, les conditions, les opérateurs de somme, les opérateurs de synchronisation, de séquence et de composition alternative en entrée [23]. La partie de gauche de l'égalité de la définition d'un processus linéaire correspond à un seul processus possédant un état courant. La partie droite de l'égalité possède une série de conditions. Cette série de conditions est suivie d'actions qui peuvent être exécutées si la condition est remplie. Ces actions peuvent être suivies ou non d'un état suivant. On lit une équation de linéarisation de la manière suivante : "Étant donné l'état actuel X, si la condition est remplie, l'action Y peut être exécutée ce qui peut optionnellement donner un état suivant Z"[23]. On comprend pourquoi le nom de règle condition-action-effet est souvent donné à la linéarisation.

Pour bien comprendre, prenons l'exemple de la spécification suivante :

act	$action1, action2;$
proc	$Process1 = action1 + action2$;
init	$Process1;$

[23]

Une fois linéarisée, elle prend la forme spécifiée à la figure 3.7

FIGURE 3.7 – LPS linéaire de processus

```
act  Terminate,action1,action2;

proc P(s1_Process: Pos) =
       (s1_Process == 2) ->
         Terminate .
         P(s1_Process = 3)
     + (s1_Process == 1) ->
         action2 .
         P(s1_Process = 2)
     + (s1_Process == 1) ->
         action1 .
         P(s1_Process = 2)
     + delta;

init P(1);
```

Cette forme linéaire est difficile à lire même lorsque les spécifications sont simples. Sa complexité augmente fortement lorsqu'on intègre de la composition parallèle. Par exemple, prenons la spécification simple suivante :

act	$action1, action2, action3;$
proc	$Process1 = action1.(action2 + action3)$;
	$Process2 = action1.(action2 + action3)$;
init	$Process1 \|\| Process2;$

[23]

Cette spécification, bien que simple et composée de peu d'actions, donne le LPS spécifié à la figure 3.8

FIGURE 3.8 – LPS linéaire de processus parallèles.

```
act  Terminate,action1,action2,action3;

proc P(s1_Process,s2_Process: Pos) =
       (s1_Process == 4) ->
         action3 .
         P(s1_Process = 2)
     + (s1_Process == 4) ->
         action2 .
         P(s1_Process = 2)
     + (s1_Process == 1) ->
         action1 .
         P(s1_Process = 4)
     + (s2_Process == 4) ->
         action3 .
         P(s2_Process = 2)
     + (s2_Process == 4) ->
         action2 .
         P(s2_Process = 2)
     + (s2_Process == 1) ->
         action1 .
         P(s2_Process = 4)
     + (s1_Process == 4 && s2_Process == 4) ->
         action3|action3 .
         P(s1_Process = 2, s2_Process = 2)
     + (s1_Process == 4 && s2_Process == 4) ->
         action2|action3 .
         P(s1_Process = 2, s2_Process = 2)
     + (s1_Process == 4 && s2_Process == 1) ->
         action1|action3 .
         P(s1_Process = 2, s2_Process = 4)
     + (s1_Process == 4 && s2_Process == 4) ->
         action2|action3 .
         P(s1_Process = 2, s2_Process = 2)
     + (s1_Process == 4 && s2_Process == 4) ->
         action2|action2 .
         P(s1_Process = 2, s2_Process = 2)
     + (s1_Process == 4 && s2_Process == 1) ->
         action1|action2 .
         P(s1_Process = 2, s2_Process = 4)
     + (s1_Process == 2 && s2_Process == 2) ->
         Terminate .
         P(s1_Process = 3, s2_Process = 3)
     + (s1_Process == 1 && s2_Process == 4) ->
         action1|action3 .
         P(s1_Process = 4, s2_Process = 2)
     + (s1_Process == 1 && s2_Process == 4) ->
         action1|action2 .
         P(s1_Process = 4, s2_Process = 2)
     + (s1_Process == 1 && s2_Process == 1) ->
         action1|action1 .
         P(s1_Process = 4, s2_Process = 4)
     + delta;

init P(1, 1);
```

Plusieurs constatations peuvent se faire. La première est qu'il est difficile pour un être humain de la lire et de la comprendre à cause de ses multiples états. Dans cet exemple, nous n'avions que deux processus exécutés en parallèle et si nous avions pris 10 processus en parallèle, nous aurions eu pour résultat plusieurs centaines d'alternatives dans le LPS. La lecture ainsi que la compréhension de centaines d'alternatives sont hors de portée d'un être humain. La deuxième est qu'on ne sait plus vraiment dans quel processus on se trouve ce qui semble essentiel à la compréhension d'une spécification. La troisième est que le LPS est fortement différent de la spécification mCRL2 entrée ce qui rend difficilement possible de débugger cette forme linéaire. Cette différence entre les deux spécifications résulte à un manque de clarté et demande une conversion complexe pour l'utilisateur qui la plupart du temps souhaite réfléchir en termes de mCRL2.

On peut tirer la conclusion que malgré qu'elle apporte une simplification dans la création d'outils, cette forme linéaire apporte son lot d'inconvénients.

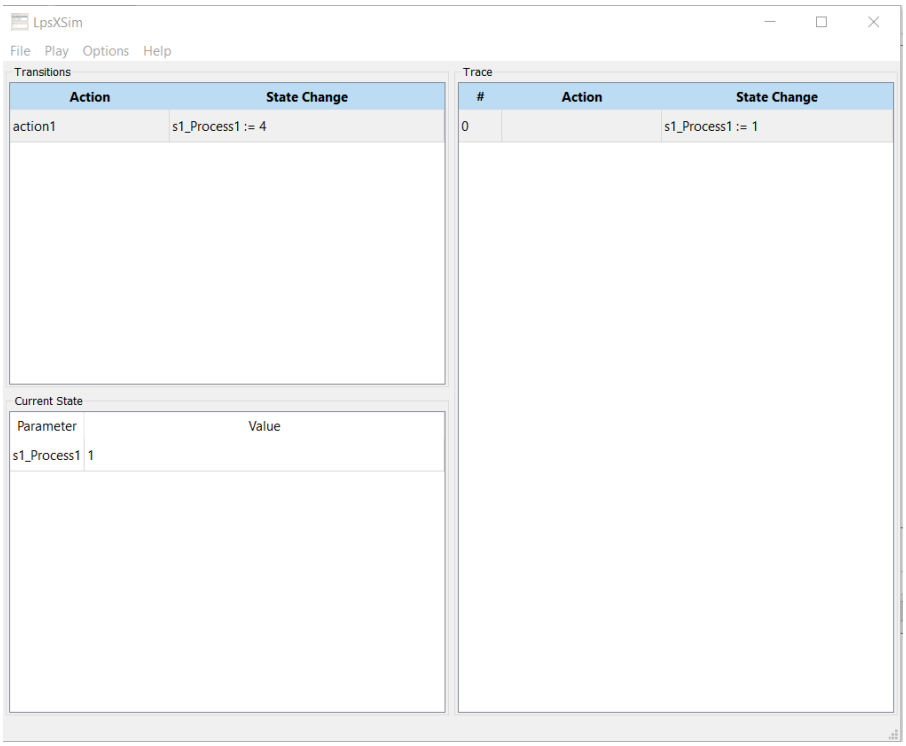
Outils de simulation : lpsxsim

Pour raisonner sur la création de notre nouvel outil, il est essentiel de comprendre les autres outils mcrl2. Un outil intéressant se basant sur LPS est lpsxsim. Il permet de simuler une spécification linéaire à l'aide d'une interface graphique. Pour comprendre son fonctionnement, prenons la spécification suivante :

$$\begin{array}{|l} \textbf{act} \quad action1, action2, action3; \\ \textbf{proc}\ Process = action1.(action2 + action3)\ ; \\ \textbf{init} \quad Process; \end{array}$$

Après sa transformation linéaire par l'outil mcrl22lps, il est possible sur ce fichier d'exécuter l'outil : lpsxsim qui présente la fenêtre graphique de la 3.9 :

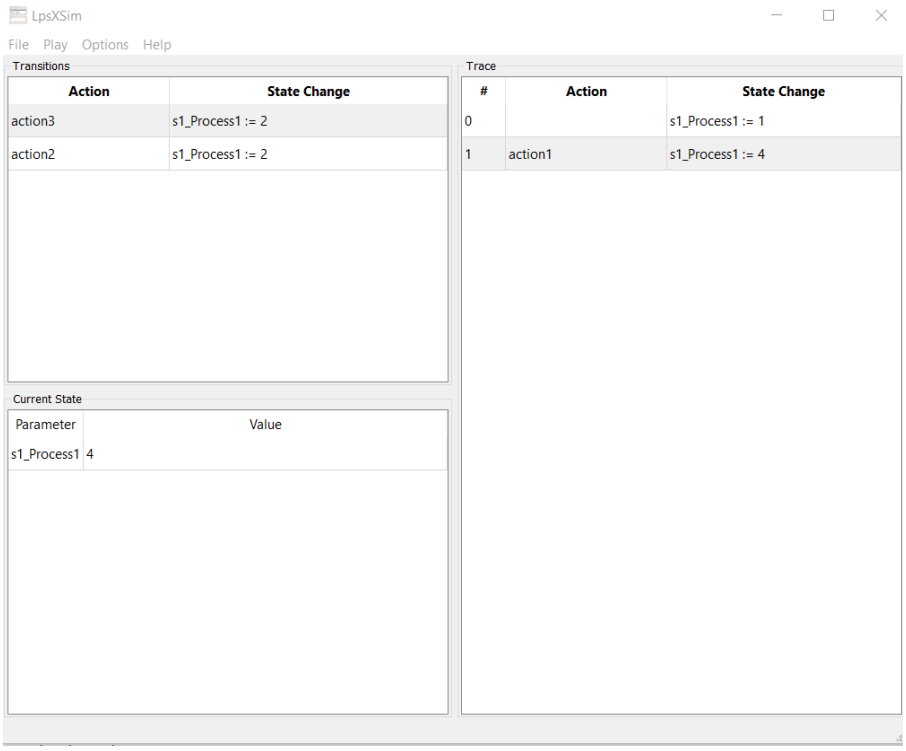
FIGURE 3.9 – Fenêtre de l'outil Mcrl2xi.



Cette fenêtre possède trois cadres. Le cadran supérieur à gauche permet de visualiser une liste de toutes les transitions possibles à partir de l'état actuel. Ce dernier contient deux parties, l'action exécutable ainsi que l'état résultant de cette action. Pour réaliser l'action, il suffit d'effectuer un double clic sur l'action à réaliser. Le cadran inférieur gauche montre l'état actuel tandis que le dernier cadran liste les actions et changements d'état correspondant effectués.

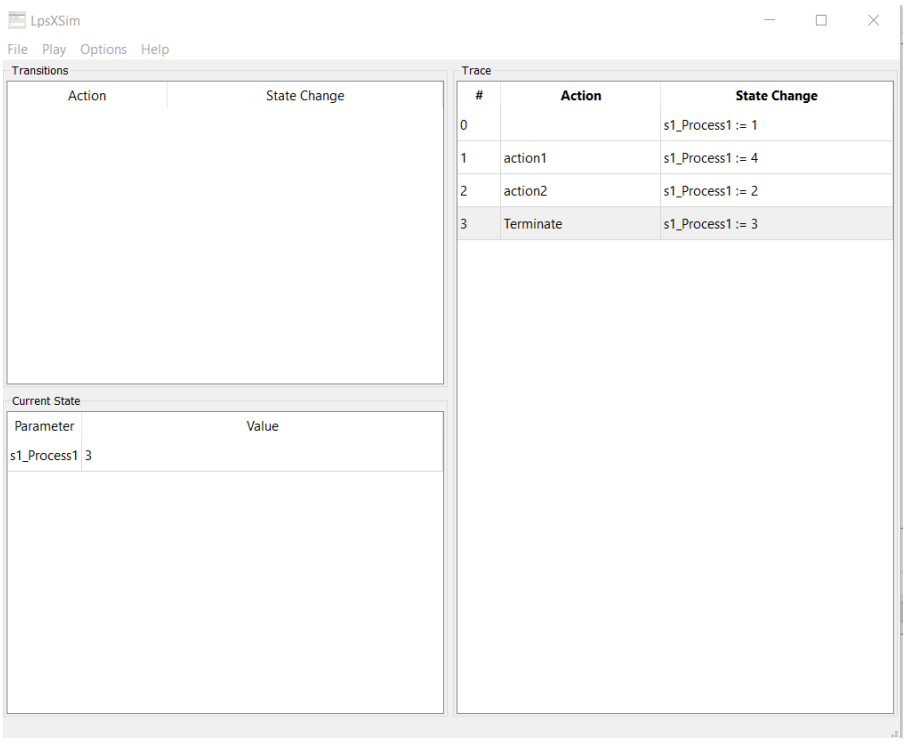
La spécification précise qu'il faut d'abord exécuter l'action1. Après son exécution, les 3 cadrans se mettent à jour comme montré à la figure 3.11.

FIGURE 3.10 – Fenêtre lpsxsim après execution de l'action1



Il nous reste deux choix possibles : exécuter l'action1 ou l'action2. En lançant l'action2, on constate que lpsxsim nous propose l'action terminate qui stipule que l'exécution est finie. A la fin de l'exécution, l'état de la fenêtre est le suivant :

FIGURE 3.11 – Feneêtre lpsxsim après execution de l'action1



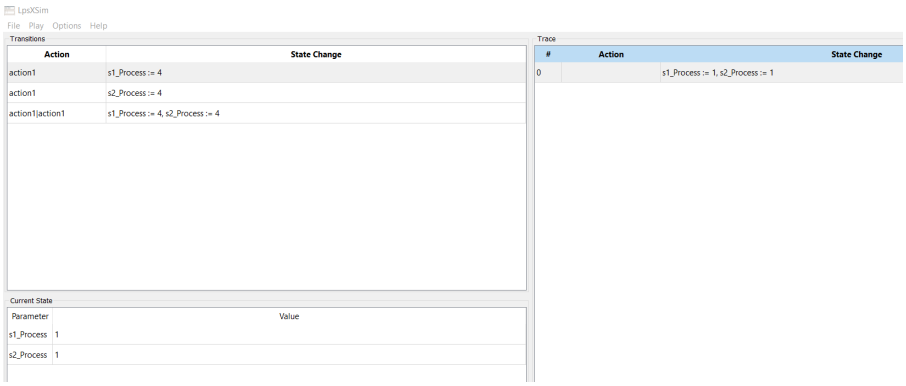
Pour démontrer les désavantages de la forme linéaire, reprenons notre exemple de la section linéarisation composé de deux processus et comportant de la composition parallèle. Pour rappel, l'exemple était le suivant :

act	$action1, action2, action3;$
proc	$Process1 = action1.(action2 + action3)$;
proc	$Process2 = action1.(action2 + action3)$;
init	$Process1 \| \| Process2;$

[23]

A l'ouverture de la fenêtre Lpsxsim, on a trois choix possibles : action1, action1 et action1 — action1 (voir figure 3.12).

FIGURE 3.12 – Lpsxsim exemple parallèle et multi processus



Si je souhaite exécuter l'action1 du processus : Process2, comment savoir quelle action choisir ? Le fait de ne pas avoir le processus stipulé dans la spécification mCRL2 demande d'ouvrir le LPS et de trouver le processus correspondant à s1_Process et s2_Process. Cependant, dans la section précédente, on a observé qu'il est difficile et peu agréable de faire la traduction entre LPS et mCRL2. Ce premier constat montre une limite de l'outil : le manque d'indication du processus.

Une deuxième limite est qu'il est difficile de s'y retrouver dans la colonne state, en dehors du nom des processus qui ont changé. Par exemple, à la figure 3.12, l'utilisateur pourrait se questionner sur ce que sont les := 4 dans la colonne state change, ce qu'est le current state et bien d'autres. La transformation de la spécification mCRL2 pousse à des questionnements et à des informations non nécessaires qui éloigne l'utilisateur de son objectif de lancement d'un outil de visualisation de spécifications mcrl2. Pour bien comprendre cet écran, il est nécessaire de connaître la linéarisation et de pouvoir la lire. Cette demande de connaissance supplémentaire pourrait être aisément évitée en ne passant pas par cette forme linéaire. Eviter la forme linéaire aiderait l'utilisateur à ne pas convertir, à ne pas comprendre, à ne pas maîtriser la forme linéaire et à se rapprocher de son objectif : visualiser sa spécification mCRL2.

3.2.4 Le système de transition labellisé

LPS est une description symbolique du comportement d'un système. Le système de transition labellisé, en abrégé LTS, rend ce comportement explicite [24]. Sous d'autres termes, il indique que dans n'importe quel état du système, un certain nombre d'actions peuvent être effectuées, chacune d'entre elles menant à un nouvel état. Le LTS peut être vu comme un ensemble d'étiquettes d'actions qui se compose d'un ensemble d'états, d'un état initial et d'une relation de transition entre les états où chaque transition est étiquetée par une action [24, 22]. De manière plus formelle, on peut dire que LTS est un tuple (S, A,→, S0). S est l'ensemble fini d'états, A est l'ensemble d'actions, → est la relation de transition et S0 est l'état initial. On conclut qu'il s'appuie sur la linéarisation et donne les états atteignables et la relation de transition qui définit les actions possibles pour chaque état d'un LPS. Pour convertir notre LPS en LTS, l'outil lps2lt est à notre disposition.

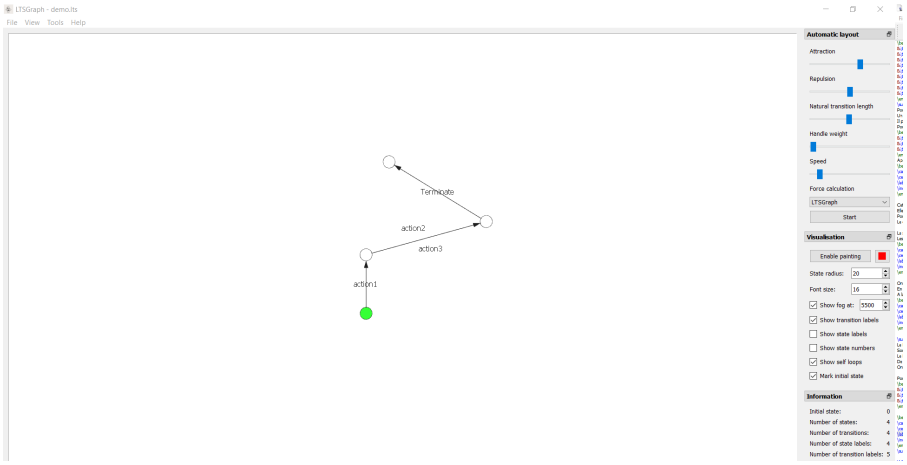
Outils de simulation : LTSView

Le lts est visualisable sous forme de graphe à l'aide de l'outil LTSGraph. L'exemple ci-dessous nous donne le résultat à la figure 3.13 :

$$\begin{array}{|l} \textbf{act} \quad action1, action2, action3; \\ \textbf{proc}\ Process = action1.(action2 + action3)\ ; \\ \textbf{init} \quad Process; \end{array}$$

[23]

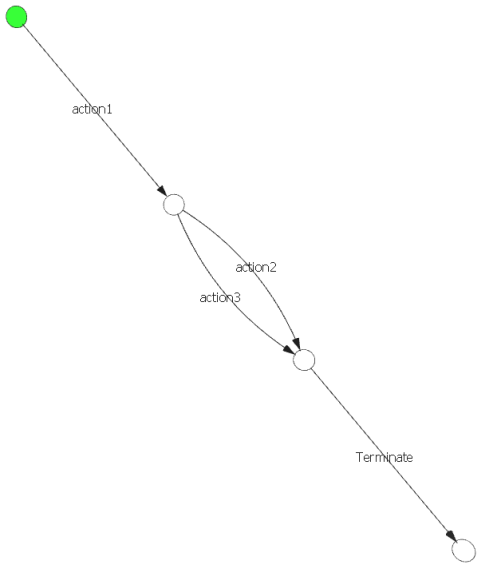
FIGURE 3.13 – LTSgraph de l'exemple



Cette figure nous montre le point de départ en vert et l'état de fin indiqué par un rond blanc avec l'exécution de l'action Terminate. On constate que l'action 1 doit obligatoirement être prise. Les actions 2 ou 3 viennent ensuite. Pour finir, l'action Terminate doit être effectuée.

Cependant, cet outil est peu convivial et ne semble pas efficace à la détection d'erreurs. Cette observation provient du manque de possibilité d'exécution pas à pas des différentes actions comme il était possible de faire dans l'outil de simulation du LPS. En revanche, il permet d'avoir une vue d'ensemble de l'enchaînement d'actions. La figure 3.14 nous montre ce qu'il se passe lors du clic sur start.

FIGURE 3.14 – LTSgraph exemple

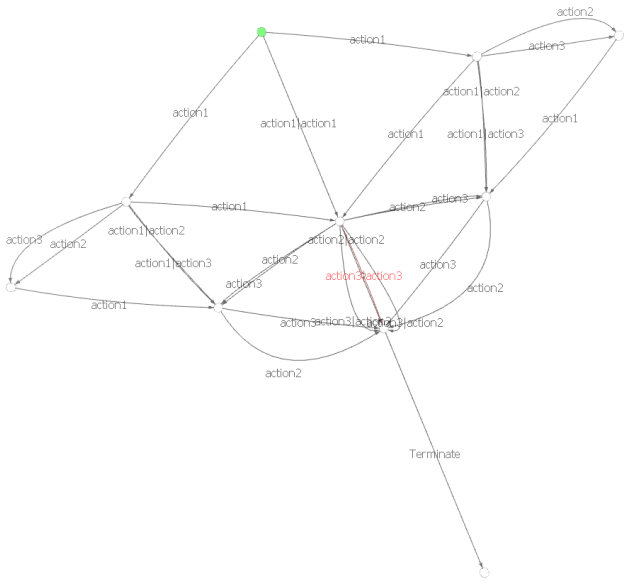


Dans ce premier cas, l'exemple était simple. Pour vérifier la capacité de visualisation reprenons notre exemple :

act $action1, action2, action3;$
proc $Process = action1.(action2 + action3)$;
$Process = action1.(action2 + action3)$;
init $Process||Process2;$

Le résultat est affiché à la figure 3.15. Cette figure démontre clairement les limites de l'outil. Sur des processus simples, avoir une vue d'ensemble peut s'avérer fort efficace mais dès que l'enchaînement des processus se complexifie, on arrive à des graphes illisibles. Si le souhait de l'utilisateur est de suivre un chemin d'exécution, il est difficile de ne pas s'y perdre. De plus, il est de nouveau difficile de savoir dans quel processus on se trouve. La question de l'utilisateur pourrait être : dans quel processus puis-je exécuter l'action1 ? Est-ce l'action du processus Process1 ou Process2 ? Dans ce cas précis, il n'est pas possible de répondre à la question.

FIGURE 3.15 – LTSgraph de l'exemple complexifié



3.2.5 Le système d'équations booléennes paramétrées (PBES)

Un PBES permet de faire de la vérification de modèle en fournissant une formule modale qui énonce une exigence fonctionnelle sur la spécification mCRL2 [15]. L'analyse des systèmes PBS et PBES n'a que peu d'intérêt, le nouvel outil concerne la visualisation ainsi que l'exécution de spécification, et non, la vérification de modèle sur base d'une formule. Nous retenons juste que ce système s'appuie également sur la linéarisation.

3.3 Réflexion par rapport aux outils existants

La section précédente nous présente plusieurs outils intéressants de visualisation et d'exécution. La première réflexion concerne la linéarisation. Tous les outils précités se basent sur la forme linéaire. Le souci de cette forme linéaire est que le programme de base est fortement transformé ce qui ne facilite pas son exécution du point de vue de l'utilisateur. Le risque de s'y perdre est grand, par exemple en prenant un programme contenant une composition de plus de 10 composants, nous obtenons plusieurs centaines d'alternatives. L'utilisateur doit alors être capable de faire la conversion entre l'état de la forme linéaire et son programme de base d'exécution. Nous pouvons dire que cette conversion est hors de portée d'un être humain.

La deuxième réflexion porte sur le manque d'outils de visualisation et d'exécution. Les outils qui fournissent de la visualisation ne permettent pas une visualisation de l'exécution mais plutôt une visualisation des chemins possibles. Comme on peut le voir dans l'outil LTSGraphe, dès que le programme utilise de la composition parallèle, le graphe devient vite illisible.

La troisième réflexion est que les processus dans lesquels les actions sont exécutées ne sont jamais stipulés. Probablement à cause de la transformation en LPS qui ne contient plus les noms des processus. Pourtant, dans la spécification suivante, il semble essentiel de connaître le processus et de ne pas attendre l'exécution de l'action4 pour savoir si nous etions dans le processus Process1 ou Process2.

$$
\begin{array}{|l}
\textbf{act} \quad action1, action2, action3; \\
\textbf{proc}\ Process1 = action1.action2.action3.action4\ ; \\
\quad Process2 = action1.action2.action3.action4.action5\ ; \\
\textbf{init} \quad Process1 + Process2;
\end{array}
$$

Olivier Croegaert a créé un outil se basant non pas sur la linéarisation mais sur la spécification mcrl2. Cet outil se nomme : ProCalg. Cette visualisation se fait à l'aide de règles de transition permettant à l'utilisateur de choisir l'action à effectuer de son algèbre entrée et non de son algèbre transformée. L'analyse de cet outil est présentée dans la section suivante.

3.4 Présentation de l'outil : ProCalg

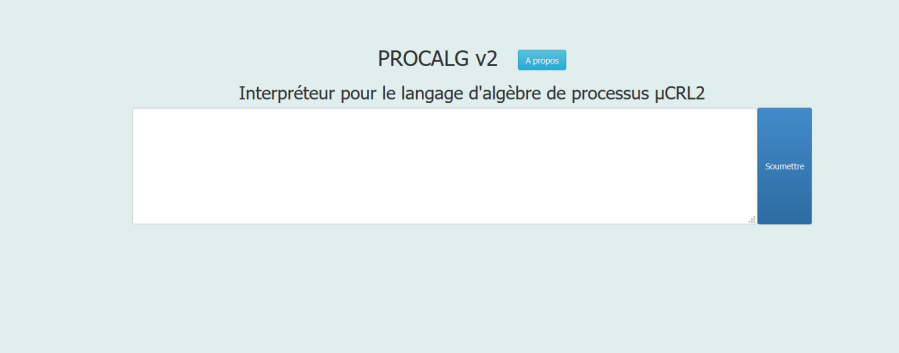
Olivier Croegaert a créé un outil pour donner suite à l'observation que la linéarisation possède le désavantage de transformer la spécification entrée [2]. Pour ce faire, il évite la linéarisation en utilisant un parseur sur le mCRL2 [2]. La description dans son mémoire ne correspond pas à la version finale de l'outil. Probablement que l'étudiant a-t-il continué son développement après l'écriture du mémoire. Cette section présente la version finale de son outil dans le but de nourrir les objectifs de l'application créée dans le cadre de ce mémoire. Comme pour les autres, nous vérifions son utilisation en insérant une spécification mono-processus simple et multi-processus parallèle afin d'en détecter les limites. La section se finit par les différentes remises en question de l'outil.

3.4.1 Présentation

En 2016, Olivier Croegaert, guidé également par Mr Jacquet, a créé un outil permettant de pallier les défauts du passage à la forme linéaire exprimée dans la section précédente [2]. L'outil ne passe donc pas par la forme linéaire mais se base sur des règles de transition afin d'exprimer l'exécution du processus. Il permet également d'avoir une visualisation et une exécution de spécification mCRL2 au moyen d'un parseur.

La figure 3.16 montre l'écran de départ de cet outil.

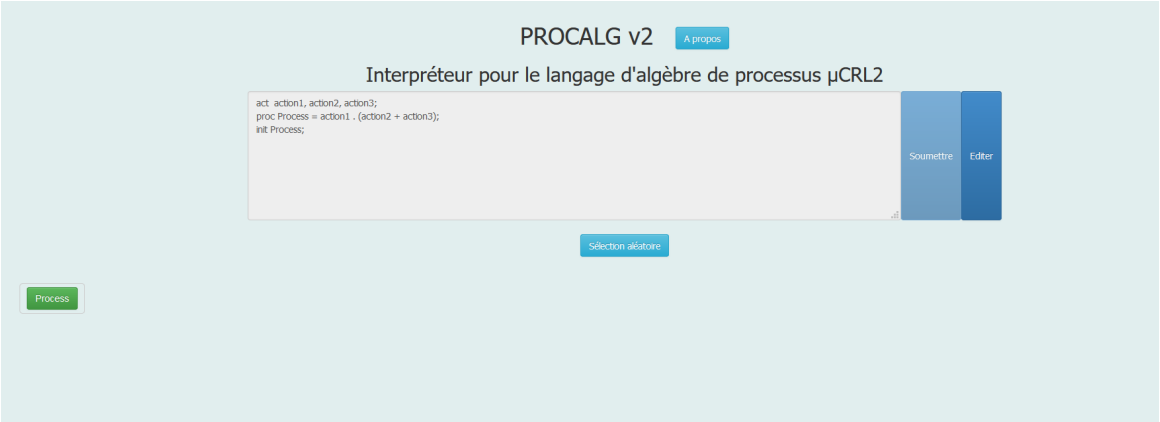
FIGURE 3.16 – Écran de départ de l'outil de Mr Croegaert [2].



On constate que l'utilisateur peut entrer une spécification mcrl2. L'outil lui offre la possibilité de la soumettre. Cette soumission peut soit contenir des erreurs soit ne pas en contenir. Dans le cas d'erreur, un pop-up apparaît à l'écran en indiquant où se trouve l'erreur. Dans le cas où il n'y pas d'erreur, la visualisation et l'exécution commencent.

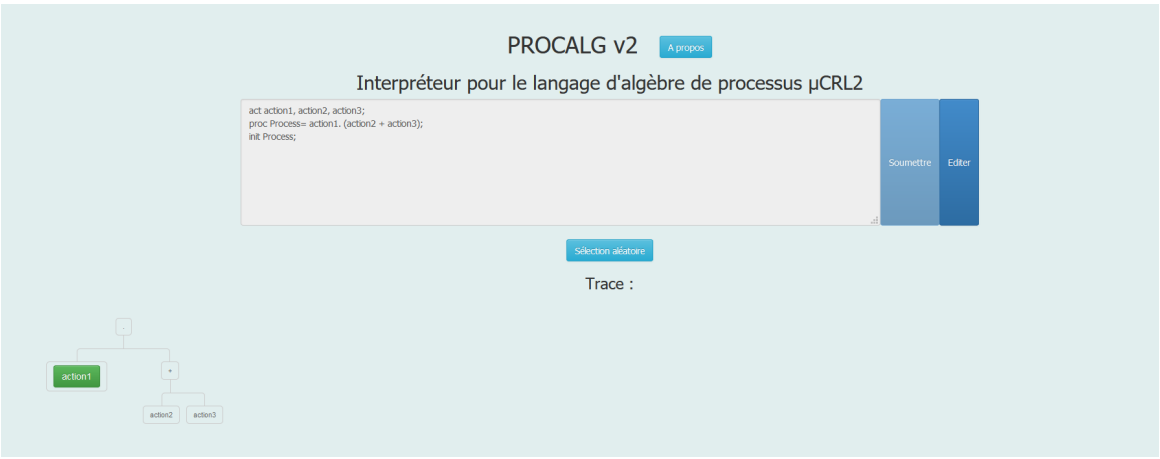
Lors de la visualisation et de l'exécution, l'utilisateur peut petit à petit sélectionner les actions à exécuter ou demander au logiciel de choisir de manière aléatoire l'action suivante. Comme nous pouvons voir sur la figure 3.17, après soumission d'une spécification, nous observons une case verte contenant : Process.

FIGURE 3.17 – Écran de soumission de l'exécution [2].



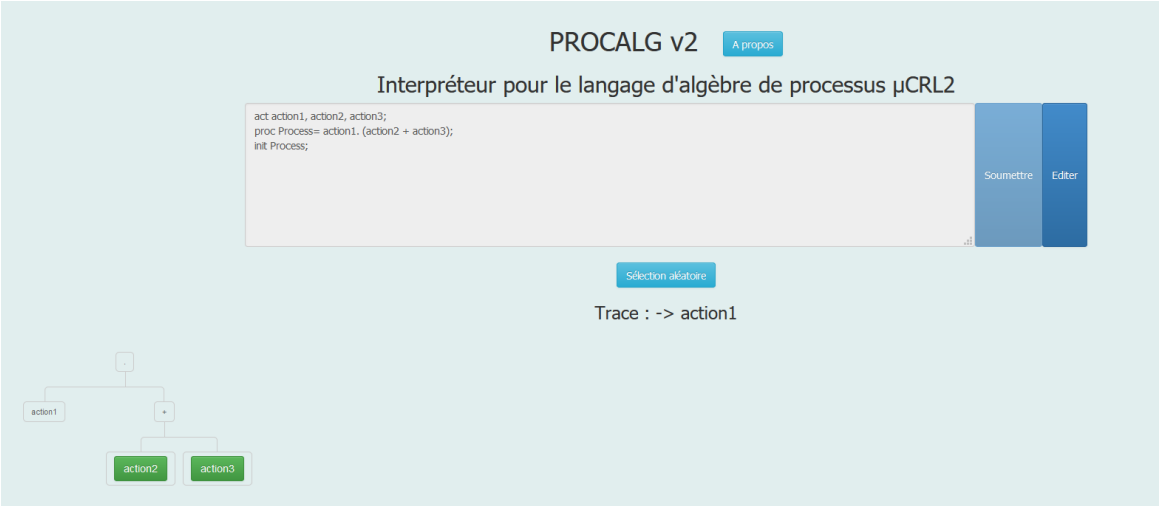
Lors du clic sur la case verte qui contient le premier processus exécutable, on aperçoit les actions exécutables suivantes 3.18. On se retrouve dans la possibilité de lancer l'action : action1 et on repère que les action2 et action3 ne sont pas encore disponibles mais qu'un choix devra être effectué à l'aide de l'arbre et de l'opérateur : +.

FIGURE 3.18 – Écran d'exécution du processus : Process [2].



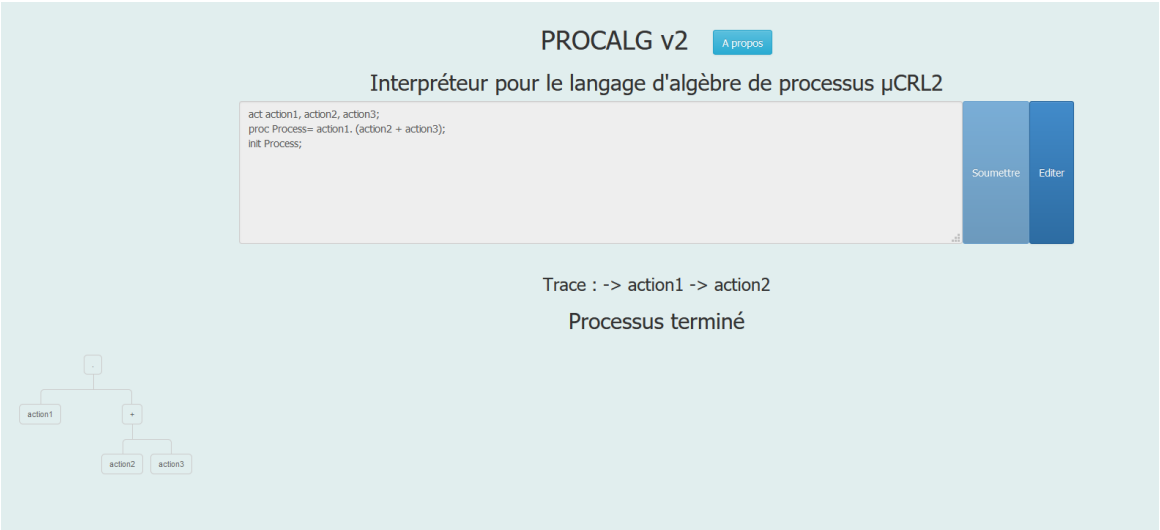
Au clic sur l'action : action1, la trace d'exécution se met à jour en indiquant : Trace : → action1 (voir figure 3.19) ;

FIGURE 3.19 – Écran d'exécution de l'action : action1 [2].



Suite à l'exécution de l'action : action1, une alternative nous est imposée : choisir l'action : action2 ou choisir l'action : action3. Dans cet exemple, le choix de prendre l'action : action2 comme action suivante est pris. Nous arrivons ensuite à la fin de l'exécution dans l'état indiqué sur la figure 3.20.

FIGURE 3.20 – Écran de fin d'exécution de l'outil de Mr Croegaert [2].



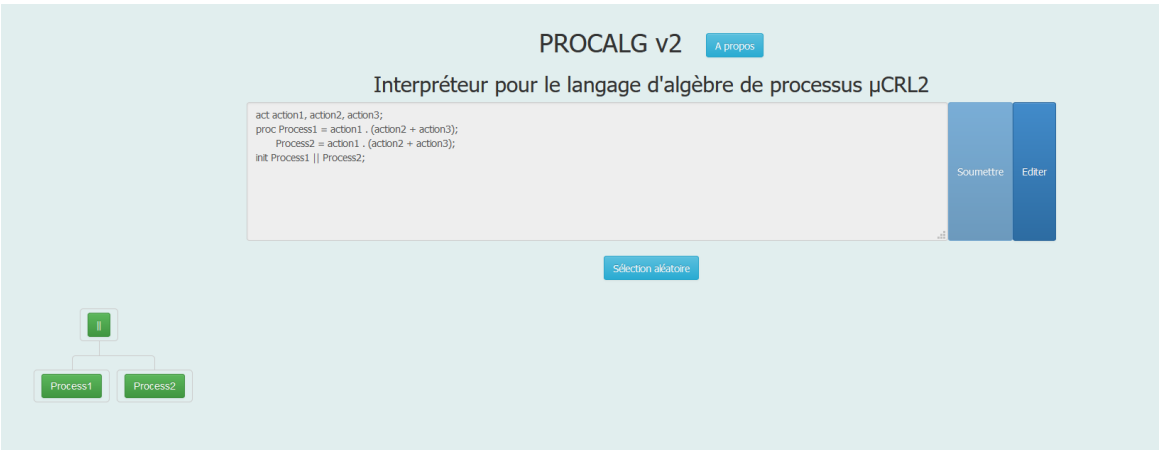
Une constatation du programme peut déjà être réalisée : ne pas passer par la forme linéaire permet d'exécuter réellement la spécification encodée et non pas la conversion linéaire. Les actions restent bien les actions encodées dans la spécification et aucun processus d'un autre nom que ceux encodés n'apparaît.

Tout comme pour les autres outils, testons maintenant avec la spécification suivante :

$$\begin{array}{l} \textbf{act} \quad action1, action2, action3; \\ \textbf{proc}\ Process1 = action1.(action2 + action3)\ ; \\ \quad Process2 = action1.(action2 + action3)\ ; \\ \textbf{init} \quad Process1 || Process2; \end{array}$$

Le résultat est indiqué à la figure 3.21.

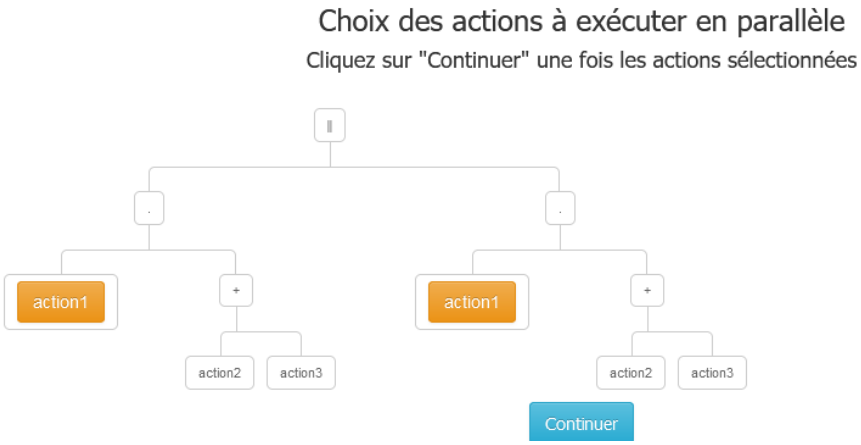
FIGURE 3.21 – Écran de Process —— Process2 [2].



On constate tout de suite qu'il offre une meilleure compréhension que les autres outils. On peut choisir le processus ce qui nous indique dans quel processus se déroulera l'action suivante. Plusieurs choix s'offrent à nous. Le premier est de cliquer sur un des deux processus ; ce qui ouvrira, comme dans l'exemple précédent, l'arbre avec les possibilités d'actions de ce processus. La deuxième est de cliquer sur le bouton || qui permet l'exécution

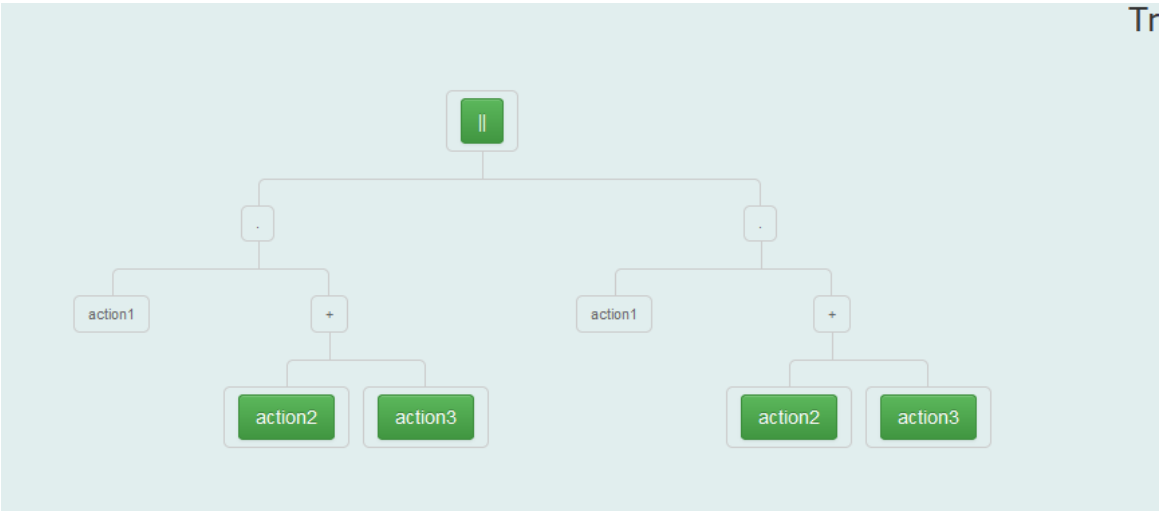
parallèle des deux processus. Lors du clic sur le bouton parallèle, on se retrouve dans le pop-up montré sur la figure 3.22

FIGURE 3.22 – Écran de traitement parallèle de processus de l'outil [2].



En sélectionnant les actions action1 et action1, on arrive à l'état indiqué sur la figure 3.23.

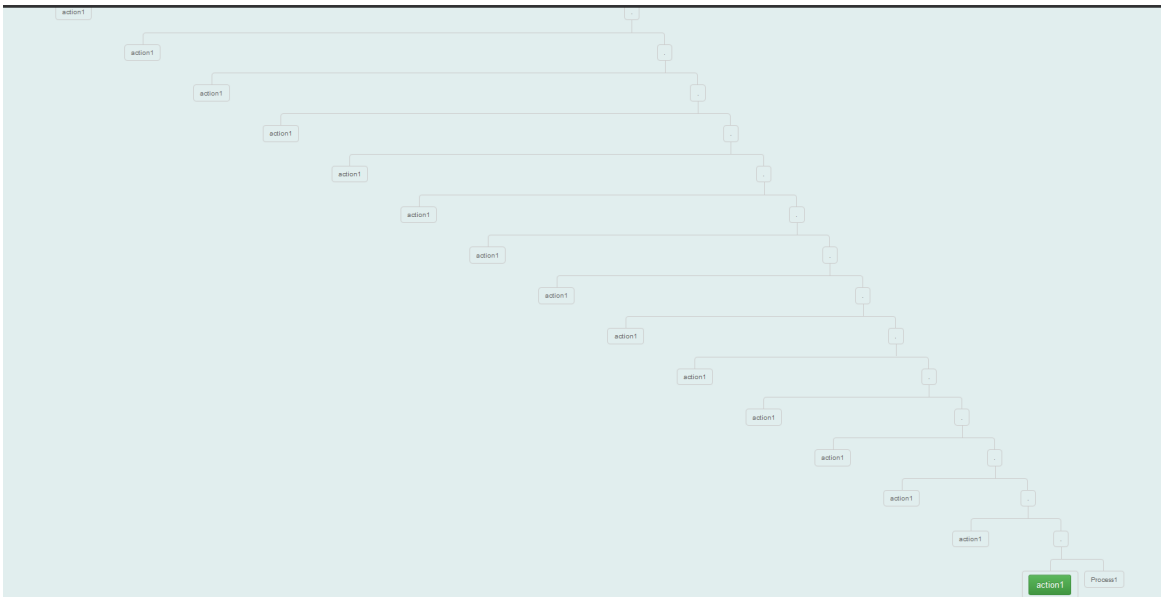
FIGURE 3.23 – Etat après exécution parallèle de action1 et action1 [2].



Une limite observée ici est qu'on se perd vite dans les processus. Il faut retenir que le sous-arbre de gauche représente le processus : Process1 et celui de droite Process2. Ce problème s'accentue lorsqu'on exécute plus de 2 processus en parallèle. Le remplacement du nom du processus par les actions exécutables rend difficile la compréhension de l'historique. Même si la trace d'exécution stipule les actions exécutées, elle ne stipule pas dans quel processus. Par exemple, si on sélectionne action1 puis action2 et action2 en parallèle, la trace d'exécution sera la suivante : Trace : → action1→ action1 → action2 → action2. Nous pouvons clairement dire que cette trace d'exécution n'indique pas la réalité de l'exécution parallèle de action1 dans Process1 avec l'exécution parallèle de action1 dans Process2. Il en va de même pour les actions : action2 lancée en parallèle.

Une autre limite est celle que l'arbre peut vite devenir grand et rendre difficile sa lecture. Par exemple, si on exécute un processus récursif, l'arbre peut vite faire plusieurs pages même avec un seul processus. Une démonstration de ce problème se trouve sur la figure 3.24.

FIGURE 3.24 – Etat après exécution de processus recursif [2].



3.4.2 Réflexion

Cet outil permet une visualisation plus élégante et agréable de l'algèbre mCRL2. De plus, il vient pallier les problèmes du passage par la forme linéaire. Il permet de réaliser une visualisation et une exécution avec des noms d'action et de processus correspondants à la spécification entrée. On constate clairement que ne pas passer par la forme linéaire amène son lot d'avantages et rend l'expérience de l'utilisateur plus agréable en ne modifiant pas sa spécification. Nous restons conscients tout de même que la forme linéaire est plus simple à gérer techniquement. Le parallélisme par exemple en serait simplifié. Cependant, certaines améliorations peuvent être faites afin de le rendre encore plus agréable et plus visuel.

La première est qu'obtenir la possibilité de sauvegarder une spécification permettra de simplifier la vie de l'utilisateur. La deuxième est le problème stipulé dans la section précédente par rapport à l'indication des processus. Il serait bien de ne pas avoir à retenir que la branche de gauche de l'arbre est tel processus et celle de droite un autre. La troisième est celui de la trace d'exécution qui ne permet pas de voir rapidement quel processus a effectué quelle action et les exécutions parallèles qu'il y a eu. La quatrième est l'arbre en lui-même. Il n'est pas très agréable de sélectionner ces actions dans un arbre qui grandit au fur et à mesure. Cet arbre exige de devoir scroller vers la spécification pour la relire et faire la correspondance avec les branches de l'arbre. Il en va de même pour la trace d'exécution qui disparaît de notre vue au fur et à mesure du grossissement de l'arbre. De plus, l'arbre n'est pas vraiment la solution idéale pour imager une spécification car il manque de dynamisme. Avoir un objet qui se déplace en fonction des actions pourrait rendre l'expérience de l'utilisateur plus agréable.

Ces réflexions supplémentaires sont prises en compte pour la création de l'application de ce mémoire.

3.5 Analyse fonctionnelle du nouvel outil : ProCalg V2

Cette section a pour but de présenter l'outil ProCalg V2 créé dans le cadre de ce mémoire. Elle commence par définir les objectifs de cet outil en indiquant les contraintes ainsi que les fonctionnalités offertes à l'utilisateur. Ensuite, elle propose une description des différentes fonctionnalités tout en indiquant l'interface graphique pour chaque fonctionnalité. Pour vérifier qu'elle correspond bien à une amélioration, une spécification ayant rendu la vie compliquée à d'autre outil est exécutée pas à pas. Pour finir, elle indique les faiblesses, les limites ainsi que les améliorations fonctionnelles possibles de l'application.

3.5.1 Objectif

L'objectif de l'outil est de pouvoir exécuter une spécification en mCRL2 tout en visionnant les actions exécutées pas à pas et ce de manière dynamique sans passer par la linéarisation. Cet objectif aidera l'utilisateur à comprendre, visualiser et débugger son algèbre de processus. La visualisation doit être imagée à l'aide d'objet dynamique. Il est important que cette représentation visuelle permette d'interpréter efficacement les actions exécutées au sein des différents processus spécifiés. On peut définir la visualisation comme "l'utilisation de représentations visuelles interactives de données assistées par ordinateur pour renforcer la cognition"[25]. Il faut comprendre le mot cognition comme le pouvoir de la perception humaine dans cette phrase précédente [25].

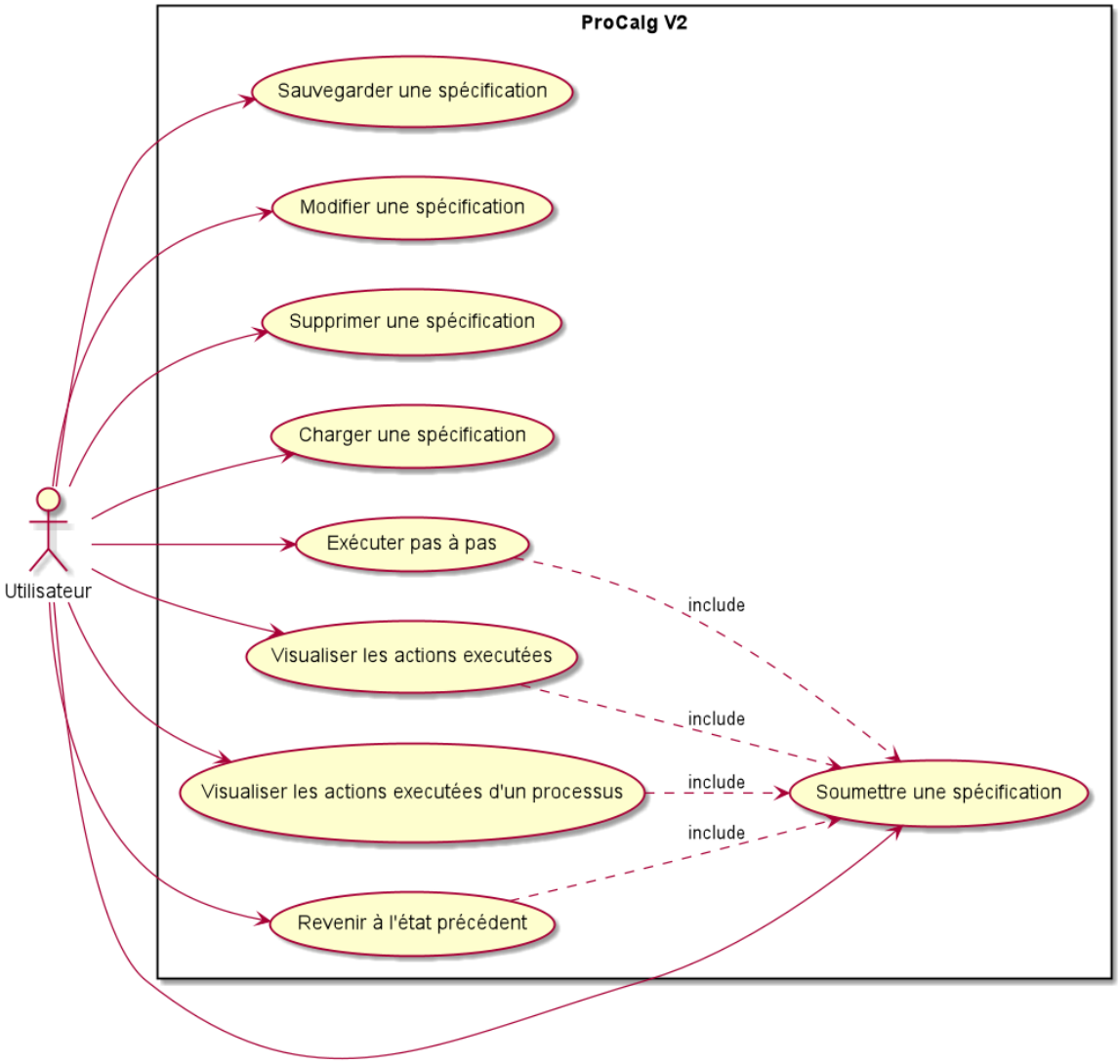
Il est important également que l'utilisateur puisse distinguer les processus dans lesquels il décide d'exécuter l'action. Si un processus A exécute l'action b et qu'un autre processus B exécute l'action b, il doit savoir dans quel processus est exécuté b pour ne pas avoir à le deviner (ce qui n'est pas toujours possible). Il doit également lors de la sélection savoir dans quel processus il exécute l'action.

Les différentes constatations précédentes doivent être résolues. La sauvegarde d'une spécification qui demande également d'avoir les fonctionnalités de suppression et de modification d'une ancienne spécification est également un objectif. La trace d'exécution doit permettre à l'utilisateur de connaître rapidement : les actions réalisées ainsi que les processus dans lesquels l'action a été lancée et les actions réalisées par processus. La possibilité de revenir en arrière semble être une bonne manière de débugger sa spécification. Si l'utilisateur à un choix à faire entre l'action a et l'action b, lorsqu'il choisit l'action a, il doit pouvoir revenir en arrière et pouvoir voir ce qu'il se passe lors du choix de l'action b.

L'exécution doit se concentrer sur les actions réalisables et ne pas fournir des informations non essentielles comme les opérateurs qui séparent les actions. En d'autres termes, la visualisation doit contenir les processus et les actions réalisables. Cette contrainte permettra de rendre la visualisation simple et intuitive.

Pour finir, il ne doit pas y avoir de perte de fonctionnalité entre l'application ProCalg et celle-ci. Effectivement, l'idée est que cette version soit une amélioration de la version de ProCalg. L'idée également de ne pas avoir de base de données comme dans ProCalg est conservée pour permettre à l'outil de rester le plus simple possible en termes de fonctionnement, de sécurité et de déploiement. La figure 3.25 exprime à l'aide d'un diagramme des cas d'utilisation les diverses fonctionnalités de l'application.

FIGURE 3.25 – Diagramme des cas d'utilisation de l'application StepByStepVisualisation.

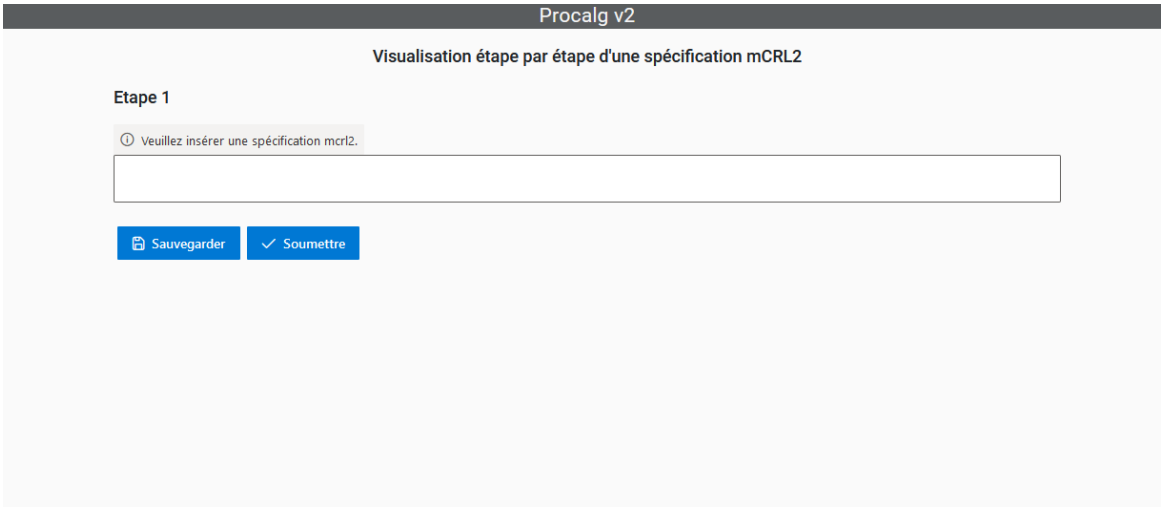


3.5.2 Présentation de l'application

Écran d'accueil

Lorsque l'utilisateur arrive sur le site web, la première page visible est disponible sur la figure 3.26. Cette page invite à insérer une spécification mCRL2. On observe deux options possibles : celle de sauvegarder ou celle de soumettre. Il est bien entendu possible de sauvegarder après la soumission ou de soumettre après la sauvegarde.

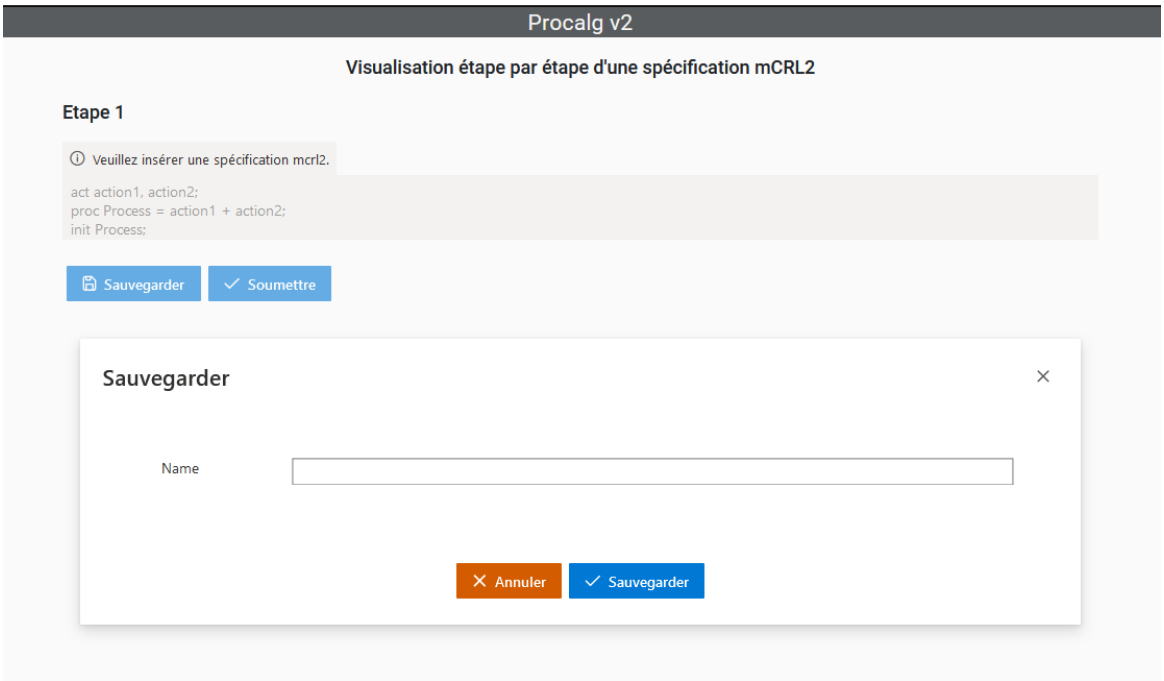
FIGURE 3.26 – Écran d'accueil de ProCalg v2.



Sauvegarder, modifier, charger et supprimer une spécification

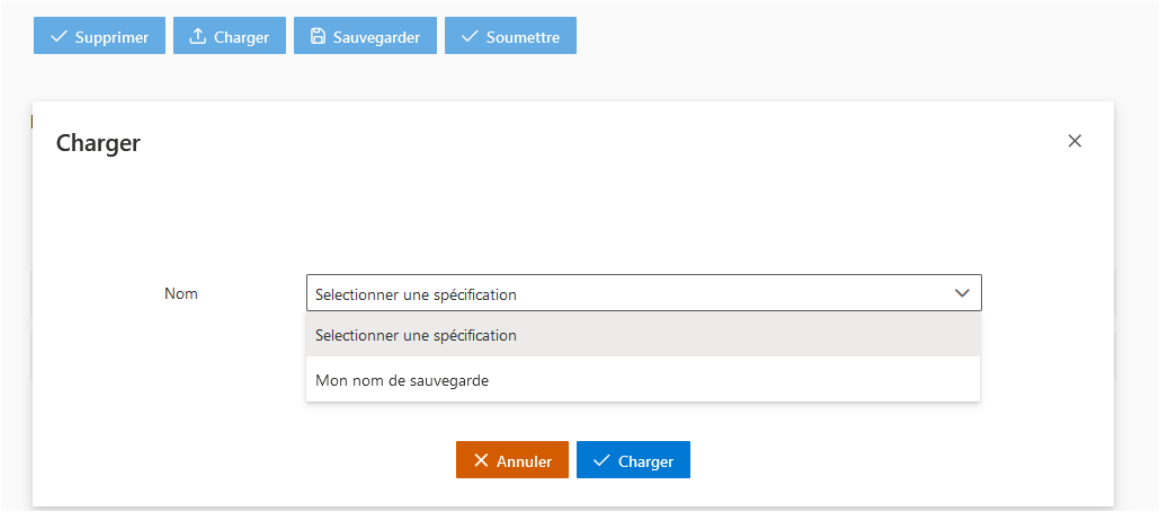
Lors du clic sur le bouton Sauvegarder de l'écran d'accueil, l'utilisateur a la possibilité de sauvegarder la spécification encodée. Ce bouton ouvre le pop-up disponible sur la figure 3.27. Ce pop-up demande à l'utilisateur d'entrer le nom de la spécification.

FIGURE 3.27 – Sauvegarder une spécification dans ProCalg v2.



Si la sauvegarde d'une spécification existe, l'utilisateur voit apparaître le bouton Charger sur l'écran d'accueil. Lors du clic sur ce bouton, le pop-up de la figure 3.28 apparaît lui permettant ainsi de choisir dans la dropdown la spécification à charger. Une fois sélectionné, le champ texte de spécification contient la spécification sauvegardée.

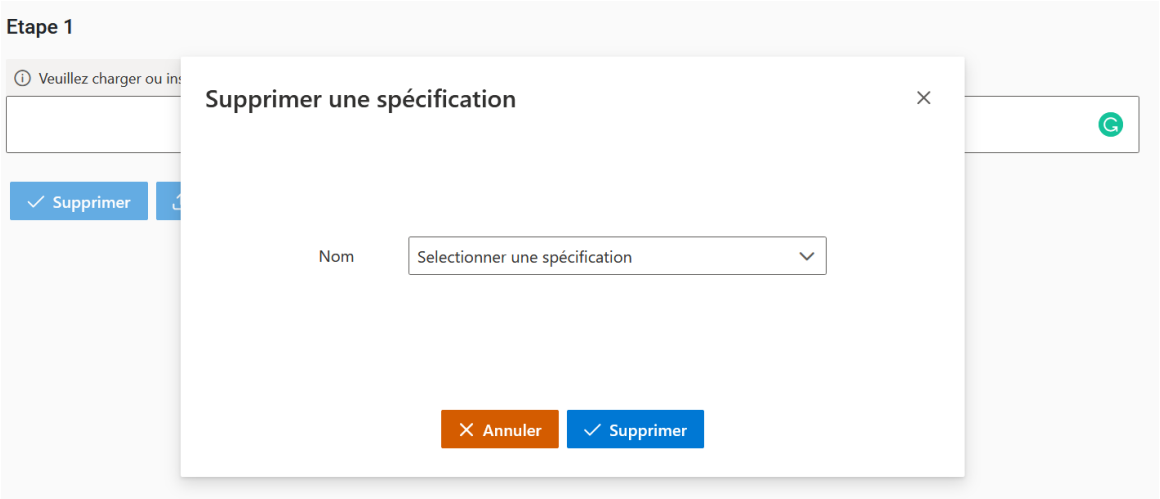
FIGURE 3.28 – Charger une spécification dans ProCalg v2.



Pour modifier une spécification, l'utilisateur peut cliquer sur le bouton Sauvegarder et indique le nom de la spécification à écraser. La sauvegarde aura pour effet de modifier l'ancienne par la nouvelle spécification insérée dans le champ texte.

Pour supprimer une spécification, l'utilisateur doit utiliser le bouton Supprimer. Ce bouton fait apparaître le pop-up affiché à la figure 3.29. Dans la dropdown, il peut sélectionner le nom de la spécification à supprimer et cliquer sur Supprimer pour la faire disparaître.

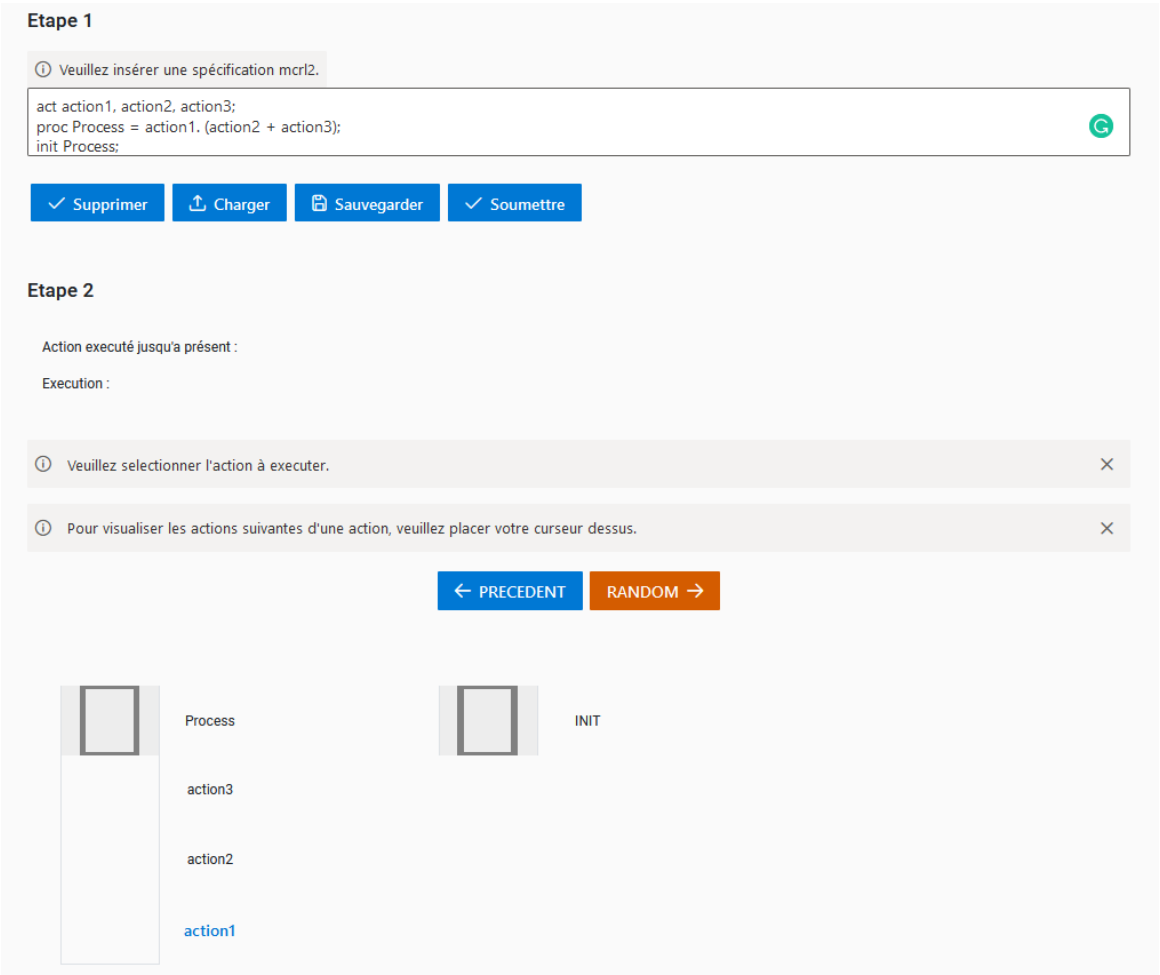
FIGURE 3.29 – Supprimer une spécification dans ProCalg v2.



Soumettre une spécification

Pour soumettre une spécification, l'utilisateur doit cliquer sur le bouton Soumettre. Si des erreurs apparaîssent, un message d'erreur sera visible au-dessus du champ texte. Dans le cas contraire, un ascenseur par processus est affiché. Le premier étage représente le nom du processus et les étages suivants les actions réalisables au sein de ce processus. La figure 3.30 montre l'écran affiché à l'utilisateur lorsque la spécification ne contient pas d'erreur.

FIGURE 3.30 – Après la soumission d'une spécification valide dans ProCalg v2.



Exécuter pas à pas

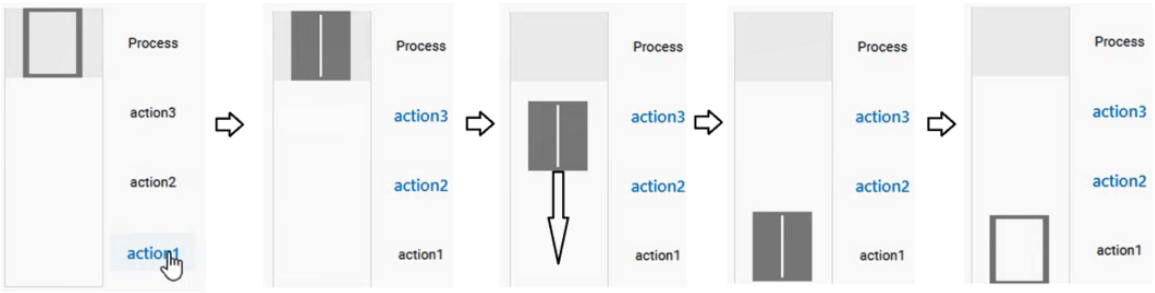
Les actions exécutables sont indiquées en gras et en bleu. Pour sélectionner une action, l'utilisateur doit cliquer sur l'action correspondante. Lors du clic sur l'action, les actions sélectionnables suivantes sont proposées, et de manière asynchrone, l'ascenseur ferme ses portes et se déplace jusqu'à la case de l'action. Voir les actions suivantes en parallèle du déplacement de l'ascenseur permet de ne pas devoir attendre le déplacement de l'ascenseur pour déjà sélectionner l'action suivante. Il permettra à l'utilisateur de ne pas être ralenti par les mouvements de l'ascenseur dans ses choix suivants. Il est également possible de cliquer sur le bouton RANDOM qui exécute une ou plusieurs actions au hasard.

La figure 3.31 montre les étapes de déplacement de l'ascenseur pour le début de la spécification suivante :

$$\begin{array}{|l} \textbf{act} \quad action1, action2, action3; \\ \textbf{proc}\ Process = action1.(action2 + action3)\ ; \\ \textbf{init} \quad Process; \end{array}$$

On constate que l'ascenseur ferme ses portes, descend à l'étage de l'action sélectionnée et les réouvre tout en indiquant déjà les actions suivantes.

FIGURE 3.31 – Après la soumission d'une spécification valide dans ProCalg v2.



On distingue deux comportements parallèles affichés par les ascenseurs : les comportements parallèles inter-processus et intra-processus. Lors d'un comportement parallèle intra-processus, on renomme les deux actions parallèles en une seule sous la forme : action1 | action2. La figure 3.32 suivante montre l'ascenseur de la spécification qui contient un comportement parallèle intra-processus :

act $action1, action2;$
proc $Process = action1||action2$;
init $Process;$

FIGURE 3.32 – Comportement intra-processus de l'ascenseur dans ProCalg v2.

On peut observer que l'action : action1 en parallèle de l'action : action2 se fait par le biais de l'action : action1 | action2 de l'ascenseur.

L'affichage de l'ascenseur d'un comportement inter-processus est différent. Prenons la spécification suivante pour l'expliquer :

act $action1, action2;$
proc $Process1 = Process2||Process3$;
proc $Process2 = action1$;
proc $Process3 = action2$;
init $Process1;$

La figure 3.33 indique les ascenseurs créés suite à la soumission de cette spécification. On constate qu'un message apparaît stipulant qu'il faut sélectionner les actions à exécuter en parallèle en cliquant sur le nom de l'action et qu'il faut ensuite cliquer sur confirmer parallèles actions pour les exécuter. Les actions sélectionnables en parallèle possèdent le signe ‖ dans la zone de l'ascenseur. L'utilisateur peut sélectionner l'action : action1 seule, l'action : action2 seule ou sélectionner les deux. Lors de la sélection, le signe ‖ se transforme en un signe : v indiquant qu'il est sélectionné et le nom de l'action devient noir. La figure 3.34 montre l'affichage lors de la sélection de l'action : action1. Si action1 et action2 sont sélectionnées et que l'utilisateur clique sur confirmer parallèles actions, les deux ascenseurs se déplacent simultanément et arrivent dans la case correspondante en même temps quelle que soit la distance.

FIGURE 3.33 – Comportement inter-processus de l'ascenseur dans ProCalg v2.

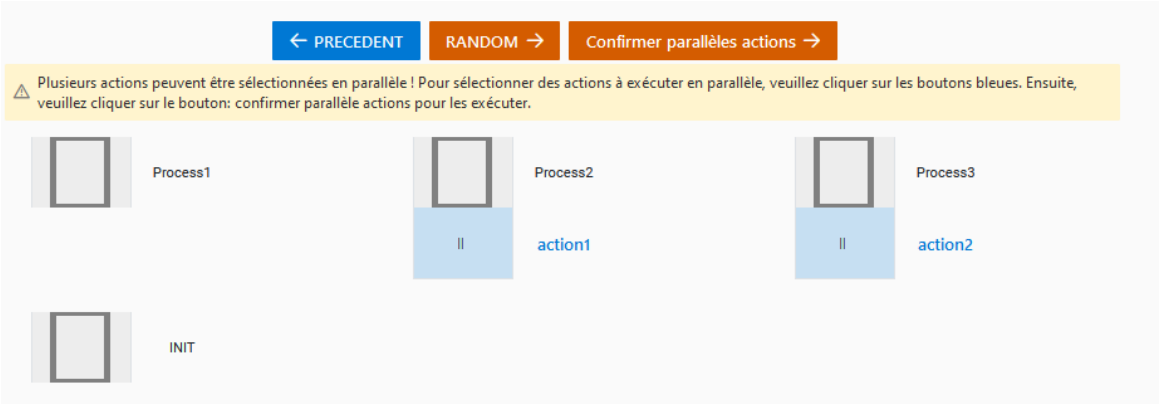
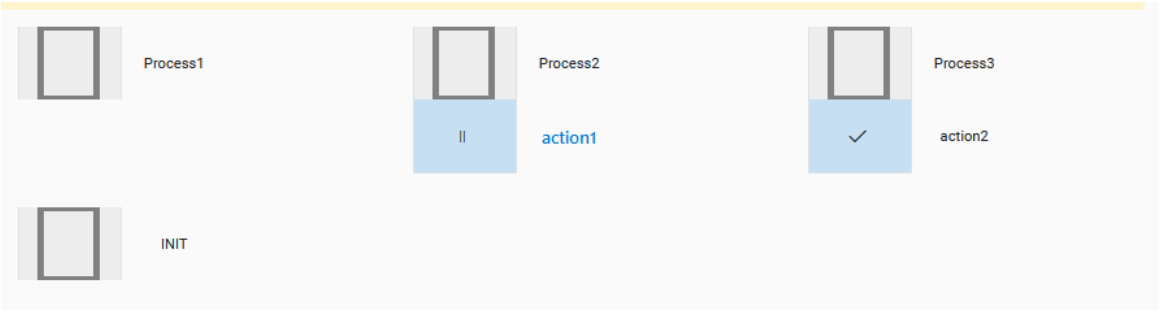


FIGURE 3.34 – Comportement de la sélection d'une action parallèle intra-processus dans ProCalg v2.

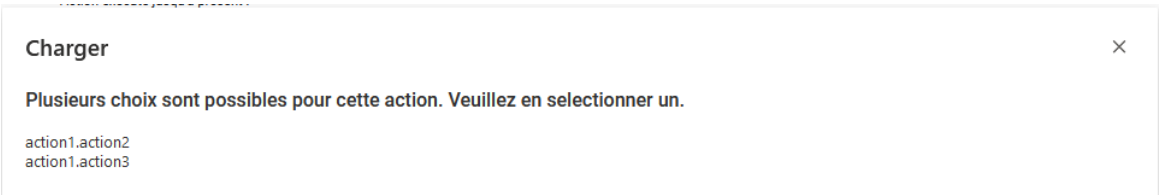


Dans certains cas, la sélection d'une action implique de décider du choix suivant au moment de sa sélection. Comme stipulé dans la partie théorique, la spécification : $action1.(action2 + action3)$ n'est pas identique en terme de comportement que la spécification : $(action1.action2) + (action1.action3)$. Dans le premier cas quand on lance action1, on peut décider ensuite d'effectuer l'action b ou c, tandis que dans le deuxième cas choisir l'action action1 implique d'avoir déjà choisi l'action action2 ou action3. Pour pallier ce problème dans ProCalg V2, un pop-up s'ouvre et demande à l'utilisateur d'effectuer un choix entre les deux. La figure 3.35 montre le pop-up qui apparaît lors de la sélection de a dans la spécification :

act $action1, action2, action3;$
proc $Process = action1.action2 + action1.action3$;
init $Process;$

Il faut cliquer sur des deux propositions afin d'indiquer l'action1 à sélectionner : celle qui est suivie de action2 ou celle suivie de action3.

FIGURE 3.35 – Comportement de la sélection d'une action impliquant le choix de l'action suivante dans ProCalg v2.

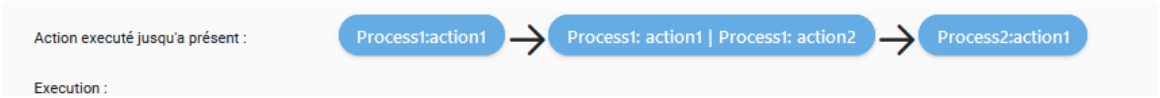


Visualiser les actions executées

Une trace d'exécution est indiquée à l'utilisateur afin de lui montrer toutes les actions exécutées ainsi que le processus dans lequel les actions ont été réalisées. Pour ce faire, le logiciel indique à côté de la mention "Action

exécutée jusqu'à présent", le nom du processus suivi de ' :', suivie du nom de l'action réalisée. Dans le cas d'actions parallèles inter-processus, le logiciel indique de la même façon la première action exécutée en parallèle, suivie de '|', suivie de l'action suivante exécutée en parallèle, suivie '|', ... Dans le cas d'actions intra-processus, les actions sont renommées en indiquant le nom du processus suivie de ' :', suivie du nom la première action suivie du signe : '|', suivie de l'action suivante exécutée en parallèle... Si une autre action a eu lieu après, le logiciel les ajoute une flèche droite suivie de l'action. La figure 3.36 montre l'exécution d'une action action1 du processus Process1 suivie d'action1 en parallèle de l'action2 du processus Processus1 suivie d'une exécution parallèle inter processus de l'action action1 du processus Process1 et Process2

FIGURE 3.36 – Trace d'exécution de ProCalg v2.



Il est également possible de visionner les actions exécutées par processus. Par exemple à la figure 3.37, on constate à coté de l'ascenseur du processus Process1 une trace d'exécution. L'utilisateur prend rapidement connaissance que le Process1 a exécuté les actions suivantes dans l'ordre : action1 puis action1 en parallèle de l'action2.

FIGURE 3.37 – Trace d'exécution par processus de ProCalg v2.

Revenir à l'état précédent

La possibilité de revenir en arrière est une fonctionnalité importante pour parcourir facilement plusieurs choix possibles. Il permet notamment dans le cas d'une composition alternative de visualiser les actions suivantes des deux alternatives. Pour ce faire, l'application dispose d'un bouton PRECEDENT. En cliquant dessus, la dernière ou les dernières actions sélectionnées précédemment sont désélectionnées et les ascenseurs sont repositionnés à leurs positions préalables à la sélection courante. Il est possible de revenir en arrière jusqu'au début de l'exécution de la spécification.

3.5.3 Applicabilité

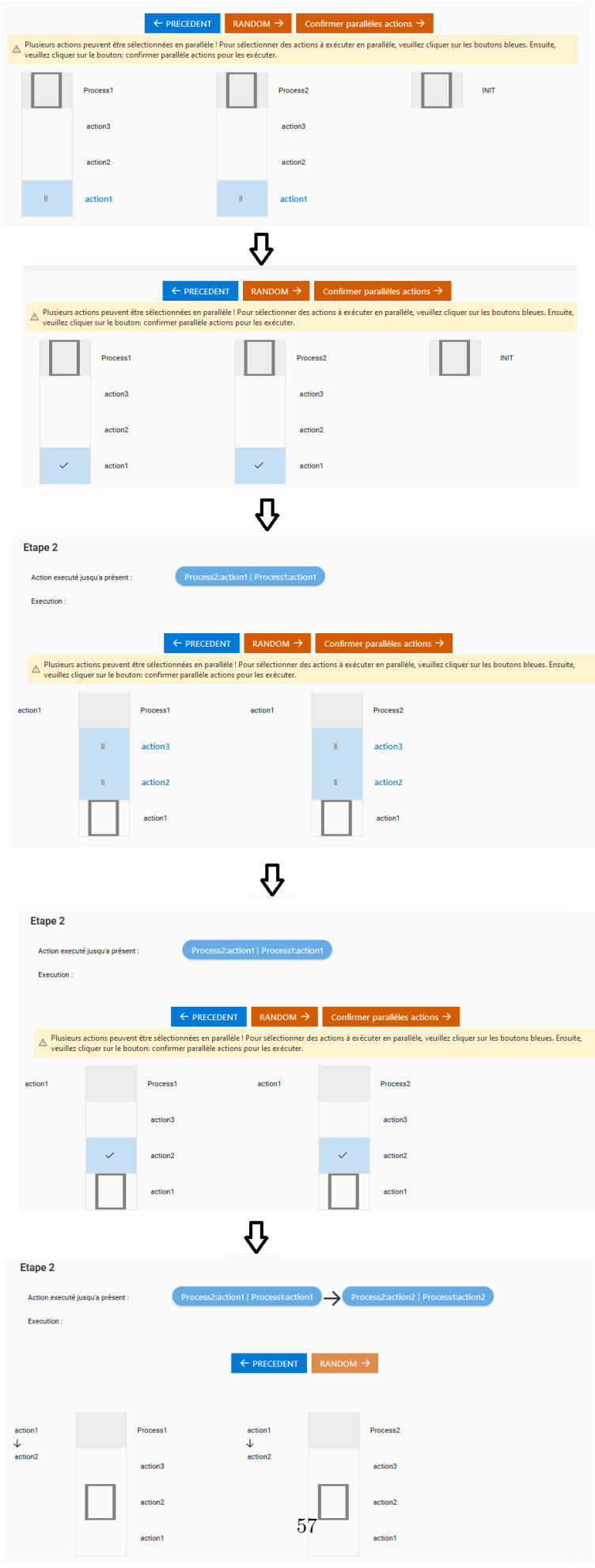
Lors de la découverte des outils existants, une spécification a été utilisée systématiquement pour détecter les limites de chacun. Pour rappel, la spécification est la suivante :

$$
\begin{array}{|l}
\textbf{act} \quad action1, action2, action3; \\
\textbf{proc}\ Process1 = action1.(action2 + action3)\ ; \\
\quad Process2 = action1.(action2 + action3)\ ; \\
\textbf{init} \quad Process1 || Process2;
\end{array}
$$

Cette dernière avait dans le cadre de l'analyse de lpsxsim démontré qu'on se perdait dans les actions à exécuter par manque de stipulation des processus (ou à cause du passage à la forme linéaire qui les renomme) dans lesquels on choisissait d'exécuter l'action. Elle avait également créé des problèmes lors de l'analyse du LTSView car elle le faisait grossir et le rendait complètement illisible. A nouveau, on ne savait plus suivre l'enchaînement des actions et on ne savait pas non plus si en prenant le chemin de droite on exécutait action1 du processus Process1 ou celui de Process2. Lors de son utilisation dans ProCalg V1, des manques de clarté sont également apparus dans la trace d'exécution ainsi que pour savoir si le Process1 était dans la branche de gauche ou de droite de l'arbre. Cette spécification bien que simple et ne comportant que peu d'action et de processus semble donner des difficultés aux outils existants. Pour vérifier l'efficacité du nouvel outil, nous allons tester cette spécification dans ProCalg V2.

La figure 3.38 montre son exécution pas à pas. La visualisation de cette spécification ne semble pas créer de problème à cet outil. L'utilisateur se rend facilement compte du processus dans lequel il exécute action1. La trace ne semble pas manquer de clarté et les états n'ont pas grossi à en devenir illisibles. Cet exemple nous prouve bien l'importance d'avoir le nom du processus dans lequel est exécutée l'action tout en confirmant que ce nouvel outil pallie des problèmes rencontrés dans d'autres logiciels.

FIGURE 3.38 – Exécution pas à pas d'une spécification donnant des difficultés aux autres logiciels dans ProCalg v2.



3.5.4 Limite

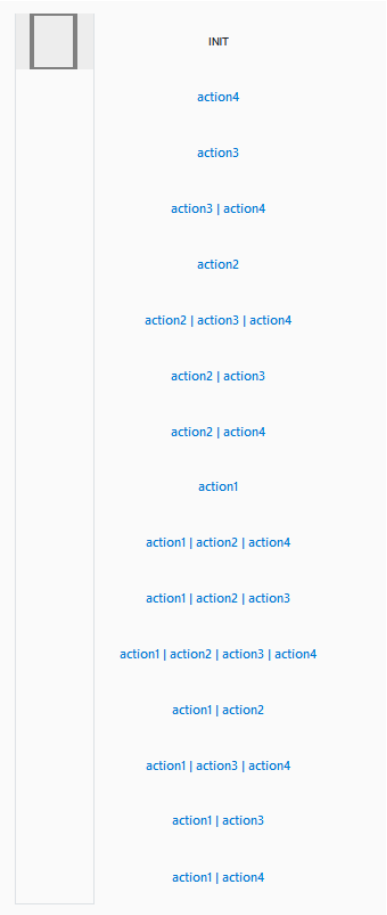
En date du 11-05-2021, l'outil possède la composition alternative, séquentielle, parallèle et accepte la récursivité. Il contient de bonnes bases pour la visualisation, l'exécution et la détection d'erreurs d'une spécification. Cependant, il présente des manques tels que l'opérateur block, communication et abstraction qui ont de l'importance dans une spécification. Dans le cas d'une communication de type a | b $\rightarrow$ c, le logiciel pourrait forcer l'utilisateur à exécuter a et b en même temps et le renommer en c lorsque la communication est intra-processus. Abstraction pourrait d'un point de vue fonctionnel, renommer un groupe d'actions en une seule. Block pourrait interdire l'exécution d'action résultant à delta. Ce manquement n'est pas dû à un souci technique mais plutôt à un manque de temps. Elles constituent les objectifs suivants en termes de fonctionnalité du logiciel.

Une autre limite est celle des communications intra-processus. Ces dernières peuvent vite grossir lorsque les possibilités augmentent. Par exemple, l'exécution de la spécification suivante donne un ascenseur d'une taille qui réduit la lisibilité et la facilité de choix d'action :

act	$action1, action2, action3, action4;$
init	$action1 \| \| action2 \| \| action3 \| \| action4;$

Son exécution est visible sur la figure 3.39. Ce nombre d'états demande à l'utilisateur de rechercher l'action exécutable et dans la plupart des cas, exiger un scroll pour visualiser l'ensemble des actions du processus. Ce nombre d'états pousse à se questionner sur l'image d'un ascenseur. Un ascenseur ne peut se dupliquer afin de permettre visuellement de faire deux actions en même temps. Cette décision serait en dehors des réalités. Une optimisation possible est d'afficher les actions parallèles intra processus exécutables et de les retirer lorsqu'elles ne le sont plus. Cependant, cette optimisation ne résout pas le problème étant donné que le nombre d'état Une autre possibilité est d'avoir plusieurs ascenseurs pour un même processus. Le nombre dépendrait du nombre d'actions parallèles possibles. Seul un ascenseur se déplacerait si aucune action parallèle au sein du processus n'est nécessaire et plusieurs lors de la sélection de plusieurs actions. Cependant, cette solution est peu élégante car un ascenseur prend beaucoup de place. L'objectif est de garder de la clarté. Ce dernier peut cependant être conservé en plaçant un objet mobile de plus petite taille tel un bonhomme. On placerait alors un bonhomme sur la case du processus par nombre d'actions exécutables simultanément au sein du processus. . On ne déplacerait qu'un seul bonhomme si c'est une action simple et plusieurs pour montrer une composition parallèle intra-processus. Cette solution prendrait moins de place et serait tout aussi performante visuellement parlant.

FIGURE 3.39 – Exécution intra processus réduisant la lisibilité.



Une limite est observée également lorsque le même processus est déclaré deux fois et que ces deux processus s'exécutent en parallèle. Lorsqu'une spécification déclare un processus X avec des actions, un ascenseur X est créé avec les actions déclarées. Cependant, si ce processus X est exécuté en parallèle avec lui-même, l'affichage peut manquer de clarté étant donné que les actions exécutables seront affichées sur le même ascenseur. L'utilisateur peut, lors de l'exécution, se perdre à ne plus savoir si l'action disponible est une action exécutable du processus X de gauche ou de droite du signe ||. Une première solution a ce problème est de créer un ascenseur par processus X déclaré. Elle permettrait de régler le problème de clarté si la spécification ne contient de pas récursivité. Dans le cas contraire, le nombre d'ascenseurs augmenterait de manière infinie. Une meilleure solution est de renommer le premier X en X1 et le deuxième en X2, et de déclarer ces deux processus avec les mêmes actions que le processus X. Dans ce cas, deux ascenseurs distincts seraient créés par l'outil permettant de fournir à l'utilisateur toute la clarté nécessaire. Cette solution semble adéquate si et seulement si la modification de la spécification est effectuée par l'utilisateur et non par le logiciel. La modification par le logiciel apporterait les mêmes problèmes rencontrés lors de l'analyse mCRL2 se basant sur la transformation linéaire de la spécification.

Plusieurs fonctionnalités peuvent être ajoutées à l'application. Par exemple, la possibilité d'avoir un visionnage de l'exécution. Ce visionnage demanderait d'indiquer un nombre de secondes entre chaque action et diffuserait à l'utilisateur le mouvement et l'enchaînement des actions exécutées pendant la spécification. Une autre optimisation est celle du bouton précédent. L'amélioration pourrait être de sélectionner dans l'historique des actions une action et de se retrouver à l'état de l'exécution de cette ou ces actions. L'idée de pouvoir avoir un correcteur automatique de spécification peut améliorer les problèmes de validations de spécification. Ce correcteur soulignerait les erreurs et proposerait une correction. Prenons l'exemple d'un utilisateur qui taperait *actab*; , le correcteur soulignerait en rouge a b et indiquerait la possibilité de sélectionner la correction : $acta, b$. De multiples améliorations peuvent encore être effectuées sur la base créée jusqu'à présent.

3.6 Conclusion

L'analyse des outils existants a permis de fixer les objectifs de ProCalg v2. Le passage par la forme linéaire, le manque d'information par rapport au processus dans lequel est exécuté l'action, le manque de clarté et le manque de représentation dynamique des outils existants sont les principaux problèmes retrouvés dans ces applications. La création d'un outil qui permet de voir des ascenseurs se déplacer par rapport aux actions exécutées apporte un réel plus en termes d'imagination, de convivialité et de visualisation de l'exécution d'une spécification. Le passage par un parcours de la spécification plutôt que par la forme linéaire permet de voir des informations concernant nos processus ainsi que nos actions spécifiées et non des informations sur une transformation. L'outil permet de remettre en question le manque d'information concernant le processus dans lequel l'action est exécutée et démontre l'utilité de réfléchir les outils en termes de processus et d'actions au lieu de le réfléchir juste en termes d'action. L'état actuel de l'outil peut être imagé par la construction d'une maison. Il ne reste que les finitions à faire mais on peut déjà y habiter. Son développement continuera avec en priorité l'ajout de block, communication et abstraction pour se poursuivre avec les fonctionnalités supplémentaires tel que le visionnage de l'exécution passée ou le correcteur de faute dans la spécification. L'application, dans son état actuel, répond aux objectifs fixés au début du chapitre. Nous pouvons conclure qu'elle pallie les problèmes des applications rencontrées notamment en passant le test d'une spécification qui a posé des problèmes à tous les autres outils existants.

Le côté fonctionnel n'est que la partie émergée de l'iceberg, le chapitre suivant exprime les améliorations techniques ainsi que les moyens techniques utilisés qui ont permis la création de l'outil.

Chapitre 4

Aspect technique de ProCalg V2

4.1 Introduction

Ce chapitre a pour but de présenter la partie non visible du programme, à savoir l'implémentation. Il commence par exposer les motivations technologiques qui expriment principalement les choix de langage et de Framework. Elle permet de comprendre ce qui a poussé ProCalg V2 dans ses technologies. Ensuite, elle présente une vue globale du fonctionnement de l'application sans rentrer dans les détails. Cette vue apporte au lecteur le cheminement de l'exécution d'une spécification. Cette vue d'ensemble est ensuite détaillée au sein de deux sections : celle du backend et celle de frontend. Pour chacune d'entre elles, une introduction aux technologies est fournie dans le but d'assurer des connaissances suffisantes pour les explications de fonctionnalités qui suivent l'introduction. Les explications de fonctionnalité donnent les solutions techniques mises en place pour la réalisation de cas d'utilisation. Conscient de la non-perfection de l'application ou des solutions choisies, la fin de ce chapitre donne une remise en question du point de vue technique de l'application. Elle met en avant les objectifs techniques futurs de ProCalg V2 tout en exposant les interrogations quant à la solution technique de leurs réalisations

4.2 Motivation technologique

De nombreuses solutions technologiques conviennent à la création de cette application. Le défi est de trouver dans ces solutions celle qui répondra le mieux aux diverses contraintes ainsi qu'au contexte de l'application.

Une des premières décisions prises est celle de scinder ou ne pas scinder le backend et le frontend. L'ancienne application scindait le backend et le frontend utilisant HTTP comme protocole de communication. L'avantage de ce scindement est qu'il sépare totalement le frontend du backend permettant à chacun de jouir d'une technologie répondant le mieux à leurs rôles. De plus, la séparation permet d'avoir plusieurs frontends pour un même backend. Ces avantages ouvrent des possibilités d'évolution telles que la création d'une application mobile, d'une application web et d'une application de bureau jouissant chacune de leurs propres choix technologiques en utilisant le même backend. Un des désavantages est le déploiement de deux applications au lieu d'une. En consultant la balance des avantages et des désavantages, tout semble confirmer que le scindement est un bon choix dans le cadre de ProCalg V2.

La décision suivante a été celle du langage de programmation. Une connaissance importante avant de prendre cette décision est celle des langages employés par ProCalg. Effectivement, la réutilisation est plus facile en restant dans le même langage. La première version de l'application utilise JavaScript au niveau du frontend et Scala 2.11.7 au niveau du backend. JavaScript semble être une bonne idée au vu de sa grande popularité et de son utilisation grandissante en entreprise. Cependant, il possède le désavantage de manquer de typage, ce à quoi vient pallier TypeScript. Ce langage semble être une bonne solution, il possède les avantages de JavaScript avec quelques défauts en moins.

Concernant le backend, la réutilisation du parseur pousse l'envie de conserver le langage de programmation

Scala. Cet argument peut faire la différence lorsque le choix entre deux langages est équivalent, mais ne peut être une raison suffisante à lui seul. D'autres avantages nous amènent au choix de Scala. Un des avantages est que le parsing de Scala est simple et capable en un faible nombre de lignes de codes de fournir une conversion d'un texte en un objet correspondant. Ce constat est visible dans l'application ProCalg dont le mémoire [2] démontre avec quelle facilité le parseur a pu être créé. Ces deux arguments, additionnés au fait de ne pas voir plus d'intérêt à prendre Java ou C# ou dans un autre langage fortement utilisé, nous dirige vers le choix de prendre Scala comme langage backend.

Les Frameworks permettent de donner de la structure, aident à la réutilisation et fournissent un gain de temps considérable au développement d'application. Certaines contraintes ont été décidé dans le choix des Frameworks. La première contrainte est que les Frameworks doivent être reconnus. La deuxième est qu'ils doivent posséder une documentation riche et facile d'accès. Ces deux contraintes permettront une prise en main rapide et facile par des personnes souhaitant lire ou améliorer l'outil.

ProCalg utilise le Framework AngularJS pour le frontend. Dans le cas de ProCalg V2, le choix a été de réécrire le frontend dans une technologie plus récente que celle proposée dans ProCalg. Cette décision a été prise parce qu'AngularJS ne sera plus supporté à partir du 31 Décembre 2021. Effectivement, on constate qu'à la date du 12-05-2021, le dernier support long terme est celui d'Angular 10. L'application a donc un retard considérable sur la technologie frontend. Angular est un bon choix de Framework pour ce type d'outil. Il a la force d'être réactif, asynchrone et permet de ne pas rafraîchir la page pour en changer son contenu. Il possède du typage fort grâce à l'utilisation du TypeScript ce qui correspond au langage de programmation choisi. D'autres Frameworks le permettent aussi, mais la maîtrise du créateur dans ce langage pousse à se diriger vers ce Framework plutôt qu'un autre. Il est également facile de le déployer, le résultat du packaging de l'application contient des fichiers JavaScript, des fichiers html et des fichiers css qui sont considérés comme des fichiers plats. Ces fichiers peuvent donc être facilement déployés sur un serveur web.

ProCalg utilise le Framework Spring boot pour le backend. Ce Framework ne manque pas de popularité à la date du 12-05-2021. Ses avantages sont multiples, pour en citer quelques-un : il facilite les tests, la création d'api, le lancement de l'application et apporte de l'injection de dépendance légère. Ces avantages sont une réelle aide dans la création de l'outil. De plus, la réutilisation de bout de code de l'ancienne application se fera plus facilement en conservant ce Framework. Tous les feux sont verts pour conserver l'utilisation de Spring boot comme Framework.

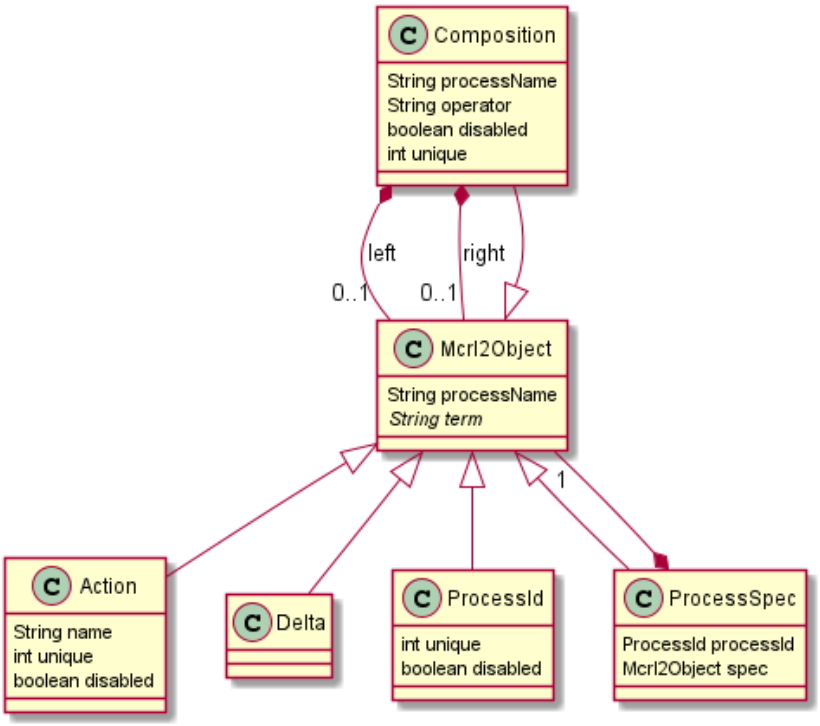
En une phrase, l'application a été créée dans le langage TypeScript en utilisant le Angular 10 comme Framework et Scala avec le Framework Spring boot.

4.3 Vue d'ensemble

Cette section a pour but de donner une vue d'ensemble de l'application d'un point de vue technique. Des détails de conception sont fournis dans la section backend et frontend.

L'application commence par l'entrée textuelle de la spécification par l'utilisateur. Ce texte est envoyé au backend via le protocole HTTP lors de la soumission. Dans le but de convertir la spécification textuelle en un objet la représentant, le backend utilise le parseur de ProCalg. Ce parseur permet de valider la spécification entrée et de générer sa représentation objet. Cette représentation est une structure en arbre. Cet arbre possède comme feuille une action ou un processus et comme branche des compositions. Le diagramme de classe de cet objet se trouve sur la figure 4.1. Il a été légèrement modifié par rapport à la version de ProCalg afin d'y ajouter notamment le nom du processus au sein des actions.

FIGURE 4.1 – Objet de résultat du parseur

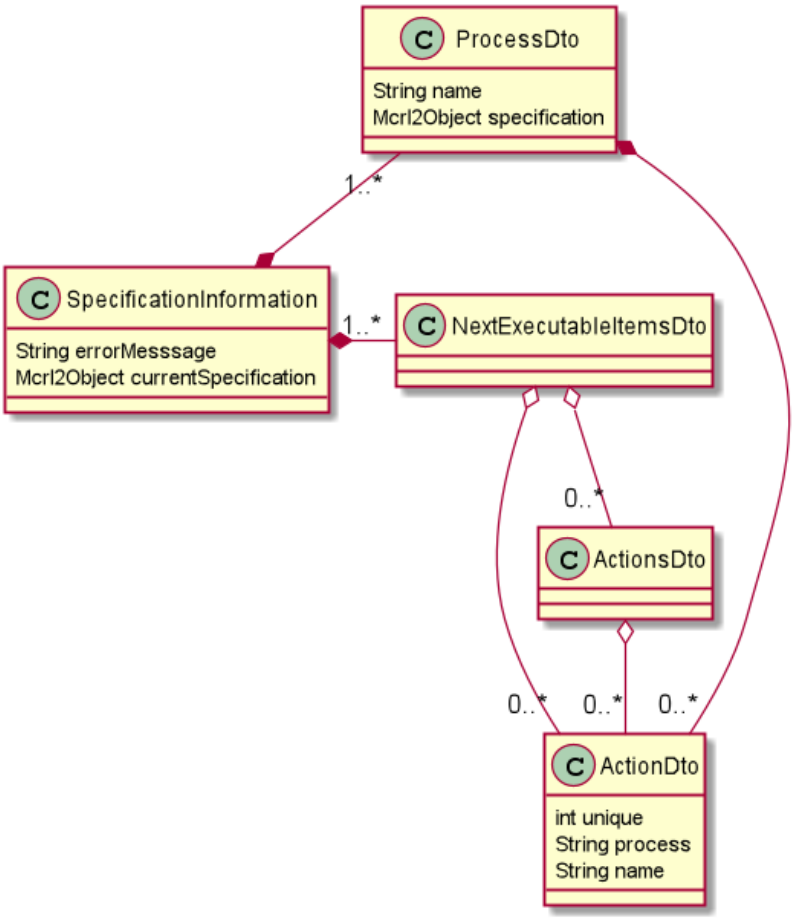


Cet objet est ensuite parcouru à l'aide d'une méthode récursive qui modifie le champ disabled à false des éléments pouvant être exécutés et à true pour ceux où l'exécution n'est pas autorisée. Dans le cas de la spécification suivante, les champs disabled des actions action1 et action2 seront modifiés à false lors de la première exécution de la méthode :

act	$action1, action2;$
init	$action1 + action2;$

Dans la première version, l'affichage est un arbre qui correspond exactement à la structure de l'objet renvoyé. Il suffisait de parcourir cet objet en fonction des types de compositions et d'afficher les nœuds petit à petit en les rendant sélectionnables ou non selon la valeur du champ disabled. Cependant, ProCalg V2 n'affiche pas la sélection à l'utilisateur sous forme d'arbre. Garder cet objet de retour du backend demande des efforts en terme de parcours de structure au frontend. Ce parcours est nécessaire pour lister les actions exécutables. Pour conserver seulement de la logique d'affichage dans le frontend, ProCalg V2 n'utilise pas cet objet comme résultat d'appel backend. Il garde cet objet comme état courant de parcours de spécification et le parcours pour en tirer des informations. Pour retirer des informations de cet objet, ce dernier est envoyé en input à une méthode récursive qui fournit en sortie les actions sélectionnables suivants. Ce résultat couplé au résultat du parseur, nous donne la classe permettant de créer l'instance de retour. Cette classe est spécifiée à la figure 4.2. Cet objet fournit toutes les informations utiles à l'affichage comme les processus, les actions possibles par processus et les actions exécutables à l'étape d'exécution courant.

FIGURE 4.2 – Objet renvoyé lors de la soumission de la spécification



Le frontend reçoit alors cet objet et l'affiche. C'est le moment pour l'utilisateur de sélectionner une action. Une fois cette sélection effectuée, le contenu du champ unique de l'action, renvoyé dans l'objet précédent, est envoyé au backend. Le backend peut alors exécuter à nouveau les deux méthodes précédentes, celle qui met à jour la structure en arbre et celle qui en calcule les actions possibles. L'output de cet appel est différent du premier. Il ne contient que les actions exécutables suivantes. Le frontend, sur base de l'objet, se met à jour. Ces enchainements continuent jusqu'à arriver à la fin de la spécification.

4.4 Backend

Cette section a pour but de donner des détails par rapport au backend. Elle commence par introduire le langage de programmation et le Framework de l'application pour comprendre les codes présentés dans les sous-sections suivantes. Elle propose de parcourir rapidement les concepts d'API, de point d'entrée et de format d'échange JSON pour donner au lecteur les clés qui permettent de comprendre les explications du fonctionnement du backend exprimées par la suite. Ensuite, elle décrit les améliorations qui ont été faite par rapport au code de ProCalg. Afin d'être complet, la description des différents points d'entrée du système est présentée dans le but de comprendre les communications possibles entre le frontend et le backend. Pour finir, l'objet utilisé pour décrire les éléments exécutables suivants est expliqué en détail afin de pouvoir l'interpréter correctement.

4.4.1 Introduction à Scala

Cette introduction demande d'avoir au préalable des connaissances en programmation orientée objet. Le nom Scala veut dire langage évolutif [26]. Ce nom lui est donné car il permet de résoudre des tâches de programmation basique comme un script tout ou des tâches plus complexes comme la création de gros système [26]. Scala est

un mélange d'orienté objet et de programmation fonctionnelle [26]. Scala est basé sur la JVM tout comme Java. Une grosse différence entre les deux est que Scala est plus moderne que Java. Il est facilement interopérable avec Java. Par exemple, il est possible de faire hériter des classes et des objets Scala d'une classe Java[27].

Pour comprendre la définition d'une classe en Scala, basons-nous sur l'exemple à la figure 4.3. On repère le mot clé "class" suivie du nom de la classe. Dans ce cas-ci, la classe est NextExecutableItemsDto. Au sein des accolades, trois attributs de classe sont spécifiés. On en déduit que pour déclarer un attribut de classe, il faut indiquer le mot clé "var". Un deuxième mot clé possible est "val" au lieu de "var". Le mot "val" exprime un champ immutable tandis que "var" exprime un champ mutable. Ce mot clé est suivi du nom de l'attribut, suivi de sa valeur précédé par une égalité. On observe également le mot clé "Array", ce mot clé exprime un tableau de type spécifié au sein des crochets. Array.empty permet de déclarer un tableau vide. Pour information, la déclaration d'une variable locale possède la même syntaxe et la même sémantique.

Une annotation @BeanProperty est visible au-dessus de la déclaration des attributs. Cette annotation permet de générer le getter et setter des attributs sur lesquels ils se trouvent.

Pour créer une méthode, il faut commencer par le mot clé "def", suivi du nom de la méthode, suivi d'une parenthèse possédant ou non des paramètres. Les paramètres d'une méthode s'expriment comme les attributs de classe sans le mot clé "var" ou "val".

FIGURE 4.3 – Exemple de déclaration de classe en Scala

```
class NextExecutableItemsDto {

  @BeanProperty
  var single: Array[ActionDto] = Array.empty;
  @BeanProperty
  var multi: Array[ActionsDto] = Array.empty
  @BeanProperty
  var processes: Array[ProcessIdDto] = Array.empty


}
```

On constate vite que ce langage de programmation n'est pas fort différent des autres langages de programmation. Scala possède en revanche quelques particularités frappantes comme celle de ne pas devoir finir la ligne par un ';' qui existe également en Python. Il n'est également pas obligatoire d'utiliser le mot clé return pour exprimer le retour d'une méthode, il suffit d'indiquer l'objet. Par exemple, la méthode back à la figure 4.4 renvoie backStates.last qui est équivalente à : return backStates.last.

FIGURE 4.4 – Retour d'une methode

```
def back(): BackState ={
  backStates = backStates.dropRight(1)
  backStates.last
}
```

La déclaration d'un constructeur se fait par le biais de parenthèses après le nom de la classe. Ces parenthèses peuvent contenir des variables de classe. La figure 4.5 montre la déclaration d'un constructeur pour la classe Mcrl2Controller qui comporte le champ mcrl2Facade de type Mcrl2Facade. Ce champ est considéré comme un attribut de la classe dès sa déclaration dans le constructeur.

FIGURE 4.5 – Constructeur en Scala

```
class Mcrl2Controller  (mcrl2Facade: Mcrl2Facade){
```

Ces informations sont suffisantes à la compréhension des sections suivantes.

4.4.2 Quelques concepts clés

Quelques concepts sont importants à comprendre afin de poursuivre la lecture. Cette section se propose de les définir. REST est l'abréviation de "Representational State Transfer" et définit une architecture de communication client/serveur sans état [28]. Dans une API RESTful, HTTP est le protocole de communication et les services disponibles sont définis comme des localisateurs de ressources unifiées (URL) [28]. Les entrées sont définies en construisant une URL avec des paramètres de requête définis par l'interface de programme d'application (API). Les données de sortie sont généralement renvoyées dans une structure définie [28]. ProCalg V2 utilise ce type d'architecture. Il possède donc des API qui proposent d'utiliser des points d'entrées permettant d'exécuter un service disponible.

Un autre concept à bien comprendre est le JavaScript Object Notation (JSON) [29, 30]. Le JSON est un format d'échange de données au même titre que le XML [31]. Il possède un format de clé valeur. Le plus simple pour le comprendre est de prendre un exemple. Parcourons la donnée de la figure 4.6, elle indique qu'une thèse nommée algèbre des processus dans l'année 2021 appartient à l'étudiant Vander Auwera qui est âgé de 27 ans. On comprend vite que les clés et les valeurs permettent de transmettre tout type de données. Ce format étant assez intuitif, nous n'allons pas plus loin dans l'explication de ce format d'échange.

FIGURE 4.6 – Exemple de JSON

```
{
  "thesis": "Process algebra",
  "year": 2021,
  "student": {
    "name": "Vander Auwera",
    "age": 27
  }
}
```

4.4.3 Introduction à Spring boot

Spring est une alternative légère au Java Enterprise Edition (JEE). Il propose une approche simple pour l'injection de dépendance et fournit entre autres la capacité des EJB sans passer par le serveur [32]. Spring peut s'avérer difficile à configurer. Initialement le XML était le moyen de le configurer, petit à petit Spring recourt à des annotations qui évitent ces déclarations XML. Malgré cela, une application Spring reste périlleuse à configurer. Spring boot est venu pallier les différentes difficultés de configuration. Il se base sur Spring et permet de gérer toute la logistique d'exécution de l'application et offre la possibilité de réduire le temps de configuration d'un nouveau projet. Un point intéressant de Spring boot est la possibilité d'avoir un serveur Tomcat embarqué. Ce serveur embarqué est empaqueté avec l'application dans un Jar. En une ligne de commande, l'application est déployée. Au vu de ces avantages, l'application ProCalg V2 s'appuie sur Spring boot.

La figure 4.7 montre la déclaration d'une entrée API atteignable en HTTP via la méthode POST sur l'url "/v2/analyse/send". Cet appel doit contenir un body convertissable en une instance de classe Input. Pour repérer ces informations, il faut comprendre les annotations. L'annotation RestController permet de spécifier que cette classe est un contrôleur REST. L'annotation RequestMapping contient la base de l'url de tous les points d'entrée déclarés au sein de la classe. L'annotation PostMapping sur la méthode send stipule que cette

méthode est exécutée lorsque l'utilisateur fait un appel HTTP de type POST à l'url "RequestMapping de la classe + /send". L'annotation ResponseStatus indique le statut HTTP retourné en cas de succès, c'est-à-dire, si aucune exception n'a eu lieu. Pour finir, l'annotation RequestBody stipule que l'objet JSON est représenté par la classe Input. Ce bout de code démontre la simplicité avec laquelle Spring permet de spécifier un point d'entrée. En moins de 10 lignes de codes, nous pouvons lancer l'application et faire un appel à notre API.

Une autre annotation importante visible sur la figure 4.7 est @Autowired. Elle se situe à côté de la déclaration de Mcrl2Facade dans le constructeur. Cette annotation permet d'injecter une instance de Mcrl2Facade sans avoir à effectuer de new. Par défaut, elle injecte la même instance à toutes les classes demandant une instance de ce type. Cette possibilité exige que la classe à injecter possède une annotation @Service, @Component, @Repository ou @Controller au-dessus de sa déclaration de classe. Elle exige également que la classe qui demande l'injection soit elle-même annotée d'une de ces annotations. Notons ici que l'annotation @RestController déclare automatiquement l'annotation @Controller. Cette fonctionnalité de Spring permet d'avoir un couplage faible entre les différentes classes en évitant le mot clé new et en passant par le constructeur.

FIGURE 4.7 – Controlleur Spring

```
@RestController
@RequestMapping(Array("/v2/analyze"))
class Mcrl2Controller(@Autowired mcrl2Facade: Mcrl2Facade) {
  /**
   * Allow to submit a spécification.
   * @param input the specification
   * @return processes, actions, next actions executables and the currentSpecification
   */
  @PostMapping(value = Array("/send"))
  @ResponseStatus(HttpStatus.OK)
  def load(@RequestBody input: Input): SpecificationInformationDto = {
    mcrl2Facade.loadSpecification(input)
  }
```

Cette introduction permet de comprendre les concepts clés de Spring boot utilisés dans l'application. Elle reste une introduction et n'explique pas toute la puissance de Spring boot. Elle est cependant suffisante pour comprendre le code Spring utilisé dans l'application.

4.4.4 Amélioration technique par rapport à ProCalg

De multiples améliorations ont été faites dans ProCalg V2. L'une d'entre elles est que ProCalg ne comportait aucun test. La nouvelle version apporte des tests unitaires et d'intégration permettant de vérifier et d'éviter les régressions de l'application.

La nouvelle version apporte aussi l'injection de dépendance qui facilite les tests et apporte une diminution du couplage. Une autre amélioration concerne la division en sous-méthode privée amenant une meilleure lisibilité du code. Le code de ProCalg possédait souvent des méthodes d'une grande taille qui rendaient particulièrement difficile la lecture.

Le backend de ProCalg permettait de récupérer un objet facilement utilisable pour construire une vue d'arbre. Grâce à un nouvel objet de type de retour, le frontend peut se concentrer sur sa tâche principale : contenir la logique d'affichage. Cet objet est plus adapté à tout type de frontend.

On constate que ProCalg V2 n'a pas fait qu'optimiser la partie visible du programme, il a également amélioré la partie technique de l'application ProCalg

4.4.5 Description de la classe SpecificationInformation

Une instance de la classe SpecificationInformation est envoyée sous le format JSON au frontend lors de la soumission d'une spécification textuelle. Cet objet est le cœur de la communication du frontend et backend. Dans la section vue d'ensemble, la classe a été décrite par le diagramme de classes affiché à la figure 4.2.

Pour bien comprendre les informations de cette classe, prenons l'exemple du retour JSON envoyé lors de la soumission de la spécification suivante :

act	$action1, action2, action3;$
proc	$Process1 = action1.(action2 + action3)$;
init	$Process1;$

Le résultat JSON renvoyé par le point d'entrée "/v2/analyze/send" est affiché à la figure 4.8.

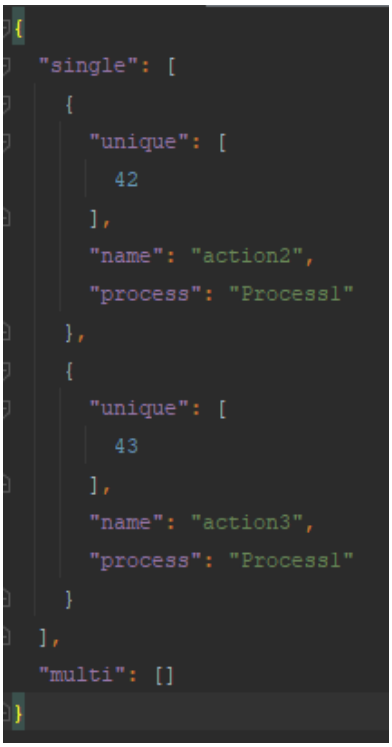
FIGURE 4.8 – Exemple de spécification information envoyé sous format JSON


```
{
  "processes": [
    {
      "name": "Process1",
      "actions": [
        {
          "unique": [],
          "name": "action3",
          "process": "Process1"
        },
        {
          "unique": [],
          "name": "action2",
          "process": "Process1"
        },
        {
          "unique": [],
          "name": "action1",
          "process": "Process1"
        }
      ],
      "specification": {"unique": 20...}
    },
    {
      "name": "INIT",
      "actions": [],
      "specification": {"name": "Process1"...}
    }
  ],
  "nextExecutableItems": {
    "single": [
      {
        "unique": [
          24
        ],
        "name": "action1",
        "process": "Process1"
      }
    ],
    "multi": []
  },
  "errorMessage": ""
}
```

On constate que les informations de cet objet permettent de créer aisément une vue de notre spécification. La clé “processes” contient le nom du processus de la spécification, à savoir Process1. A l’intérieur, la clé “actions” permet de donner toutes les actions qu’il est possible d’exécuter au sein du processus. On observe également la clé “nextExecutableItems” qui possède de deux clés à savoir “single” et “multi” indiquant les actions suivantes exécutables. Single indique les actions exécutables seules et celle parallèle intra-processus tandis que multi indique les actions exécutables en parallèle inter-processus. Dans le cas présent, l’action : action1 est exécutable et porte l’identifiant : 24. On peut voir également qu’elle appartient au processus : Process1.

Lors de la sélection de cette action identifiée sous le nombre 24, une alternative est indiquée dans le JSON de retour (voir figure 4.9). Il faut choisir entre deux “single” actions.

FIGURE 4.9 – Exemple de JSON retourné lors d’un choix alternatif d’actions



Dans tout cet enchainement backend frontend aucune action n’a été exécutée en parallèle. Dans le but d’expliquer la clé “multi”, modifions légèrement notre exemple en ajoutant un processus exécuté en parallèle dans la spécification. Nous obtenons la spécification suivante :

$$\begin{array}{|l|}\hline \textbf{act} \quad action1, action2, action3; \\ \textbf{proc}\ Process1 = action1.(action2 + action3)\ ; \\ \quad Process2 = action1.(action2 + action3)\ ; \\ \textbf{init} \quad Process1 || Process2; \\ \hline \end{array}$$

On analyse ensuite les nextExecutableItems renvoyés lors de la soumission de la spécification à la figure 4.10. On peut y observer deux “singles” actions correspondants à l’action : action1 de chaque processus. On peut donc les exécuter individuellement. On constate également que la clé multi contient elle-même une clé action composée de deux actions : action1 du Process1 et action1 du Process2. La clé multi indique que les actions peuvent être exécutées simultanément.

FIGURE 4.10 – Exemple de JSON retourné lors d'un choix parallèle inter processus d'actions

On en conclut que cet objet rassemble bien l'information d'exécution demandée. Il fournit l'ensemble des comportements possibles pour une certaine spécification à une certaine étape de son exécution afin d'être réutilisable par d'autres applications.

4.4.6 Le cas particulier du retour en arrière

Pour permettre le retour en arrière, il est obligatoire de conserver tous les états précédents. Pour ce faire, lors de la soumission de la spécification, on retient l'arbre courant ainsi que les actions sélectionnables suivantes dans une pile. Cette pile est également alimentée à chaque sélection. Lors d'une demande de retour en arrière, le backend dépile le dernier élément de la pile et renvoie à l'appelant les actions sélectionnables suivantes contenus dans l'élément dépilé.

4.4.7 Description des points d'entrée disponibles

Quatre points d'entrée d'API sont disponibles sur l'application.

Le premier permet de soumettre une spécification. Il prend en entrée la spécification sous forme de texte. En retour, il fournit les informations sur les processus et les actions exécutables à l'initialisation de la spécification. Les informations des processus comprennent : le nom, l'ensemble des actions réalisables au sein de ce processus et la spécification en arbre de ce processus.

Le second permet de sélectionner un élément d'une spécification soumise. Il demande en entrée de recevoir un tableau des identifiants à sélectionner. Il fournit en retour les éléments exécutables suivants.

Le troisième point d'entrée permet de recevoir sur base d'un identifiant la spécification sous forme de texte venant après cette action. Pour bien le comprendre, prenons l'exemple de la spécification suivante : $A = b.c + b.d$. Si ce point d'entrée est appelé pour l'action b qui se situe à gauche du '+' de la spécification, il renverra $b.c$. Au contraire, s'il est appelé pour l'action b se situant à droite du '+', il renverra : $b.d$. Ce point d'entrée trouve tout son intérêt lorsque deux actions différentes sont disponibles sur le même Processus. Il faut demander à l'utilisateur de choisir quelle action 'b' il souhaite sélectionner.

Le dernier point d'entrée permet de revenir en arrière. Il ne prend aucun paramètre en entrée et revient à la sélection précédente. Il fournit les actions séléctionnables courante en sortie.

A l'aide de ces quatre points d'entrée, l'application frontend peut se nourrir d'informations qui lui suffisent pour gérer l'affichage et la sélection d'actions.

4.5 Frontend

Cette section présente les détails de la partie frontend. Elle commence par une introduction à son langage ainsi qu'à son Framework. Ces introductions permettent de remplir le sac des connaissances du lecteur pour pouvoir se retrouver facilement dans les explications des autres sections. Ensuite, une section permet de comprendre les différentes améliorations techniques du logiciel par rapport à sa première version. Plusieurs fonctionnalités telles que la sauvegarde, la modification et le déplacement des ascenseurs sont présentées d'un point de vu technique. Pour terminer, elle indique les divers questionnements par rapport au choix pris pour cette application. Elle permet entre autres de définir les défis à réaliser par la suite pour cette application.

4.5.1 Introduction à Typescript

TypeScript est une extension de JavaScript destinée à faciliter le développement d'applications JavaScript à grande échelle [33]. Son avantage réside dans l'apport d'un système de modules de classe, d'interface mais surtout un riche système de types [33]. Il supporte également la plupart des pratiques utilisées en JavaScript [33].

Pour déclarer une classe en TypeScript, il faut utiliser le mot clé "class" et pour déclarer une interface, on utilise le mot "interface". Dans le but de fournir un exemple, la figure 4.11 déclare une classe possédant le nom "Action". Elle possède trois attributs, à savoir : "unique" de type tableau de number (un nombre), "nom" de type string (attention avec une minuscule) et "process" de type string. Les tableaux sont déclarés au moyen du symbole : "[]" après le type. Cette annotation est courte et intuitive. Un mot clé moins commun est "export". Ce mot clé permet de rendre disponible la classe dans tous les fichiers du projet. Par défaut, si on ne met rien la classe n'est disponible que dans son fichier. Il en va de même pour les variables, les interfaces, ... L'extension de classe, abstraite ou non, n'est pas visible sur la figure mais est possible en TypeScript.

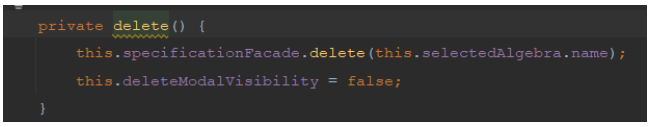
FIGURE 4.11 – Exemple de classe TypeScript

```
export class Action {
    unique: number[];
    name: string;
    process: string;
}
```

En ce qui concerne les méthodes, la syntaxe dépend de l'endroit de sa déclaration. Lorsque la méthode se trouve au sein d'une classe, il faut indiquer le nom de la méthode suivie de parenthèses ouvrante et fermante dans lesquels les paramètres de la méthode peuvent ou non être indiqués suivis de ' :' et de son type. Comme dans beaucoup d'autres langages, le corps de la méthode se délimite par des accolades et se situe après la déclaration. Typescript permet également d'ajouter de la visibilité à la méthode via les mots clés : private, public ou protected. Cette visibilité est disponible pour les attributs de classe. La figure 4.12 montre la déclaration d'une

méthode privée en TypeScript.

FIGURE 4.12 – Exemple de méthode privée en TypeScript



En revanche, lorsque la méthode est déclarée en-dehors d'une classe, elle se déclare au moyen du mot clé "function" précédé du mot clé "export" ou non. Dans ce cas particulier, aucune visibilité ne peut être ajoutée.

La déclaration de variable locale se fait au moyen du mot clé "const" ou du mot : "let'". Le mot "const" déclare une variable immutable tandis que "let" déclare un type mutable.

Il possède également les mots clés bien connus des programmeurs tels que "if', "else", "for", "while", "switch"... Ces mots-clés étant intuitifs nous ne les expliquons pas en profondeur dans ce chapitre.

4.5.2 Introduction à Angular

Angular est un Framework JavaScript qui a connu une évolution importante au cours des dernières années [34]. Le meilleur moyen de s'en rendre compte est de regarder son prédécesseur AngularJS et de les comparer [34]. Angular utilise TypeScript comme langage de programmation [34]. Un de ses nombreux objectifs est de permettre le développement unifié d'application mobile, web et de bureau.

L'architecture d'une application Angular repose sur certains concepts fondamentaux. Les éléments de base d'Angular sont organisés en module nommé NgModules. Une application possède systématiquement au moins un NgModule permettant de le démarrer. Ce dernier permet de rassembler le code connexe en ensembles fonctionnels. Il permet de déclarer des composants, d'autres NgModules et des services. On comprend tout de suite qu'Angular possède une structure modulaire.

Les composants définissent la vue. Cette classe est associée à un modèle HTML qui définit une vue à afficher dans un environnement cible [35] Il peut également être associé à un fichier css. Toutes les applications Angular possèdent au moins un composant généralement nommé "app.component". Il représente le point d'entrée de l'application. La figure 4.13 montre les constituants d'un composant. Le fichier elevator.html contient du html, le fichier .scss du css et le fichier .ts est la classe qui les associe. Le fichier elevator.spec.ts est le fichier test du composant. Pour les associer, le fichier ts contient une annotation précisant la localisation des différents fichiers. Cette annotation est visible sur la figure 4.14 et se déclare au-dessus de la classe. Un point important est le nom stipulé après la métadonnée "selector". Ce nom permet de déclarer le composant à l'intérieur d'un autre composant au moyen d'une balise html [35]. Par exemple, pour déclarer ce composant dans un autre composant, il faut dans le fichier HTML déclarer $< app - elevator >< /app - elevator >$.

FIGURE 4.13 – Fichiers constituant un composant Angular

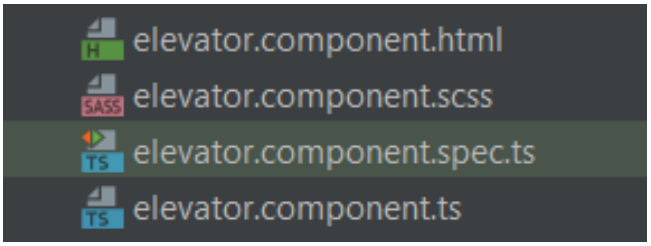


FIGURE 4.14 – Annotation permettant d'associer les différents fichiers au composant en Angular

```
@Component({
    selector: 'app-elevator',
    templateUrl: './elevator.component.html',
    styleUrls: ['./elevator.component.scss'],
```

Les composants s'emboitent les uns dans les autres. Il est essentiel que le composant parent puisse communiquer avec l'enfant et inversément. Pour ce faire, Angular possède deux annotations. La première est @Input() qu'il suffit de placer au-dessus d'un attribut de classe d'un composant enfant. Cette annotation permet au parent de communiquer avec l'enfant. Pour ce faire, lorsque le parent déclare l'enfant dans son html, il ajoute au sein de la balise de déclaration du composant enfant ouvrante, l'expression suivante : [nom de l'attribut annoté avec @Input() de l'enfant] = "valeur a envoyer". Le parent peut également passer un objet qui permettra aux deux composants de faire du partage par référence. Le deuxième est @*Output*() qui permet l'envoi d'événement de l'enfant vers le parent. Elle demande de déclarer dans l'enfant une attribut de type EventEmitter et d'annoter l'attribut avec @*Output*(). Le parent peut s'abonner sur cet envoyeur d'événement en indiquant dans la déclaration de la balise ouvrante de l'enfant des parenthèses avec l'expression suivante : (nom de l'attribut annoté avec @Output() de l'enfant) = "traitement à effectuer". Dans le traitement, il peut utiliser la variable $event qui est l'événement émis par l'enfant. L'enfant doit utiliser la méthode ".emit" de l'eventEmitter en passant en paramètre une valeur, une variable, un objet, rien ou autre à émettre au parent.

D'autres possibilités existent pour communiquer, par exemple celle d'utiliser un service. La définition d'un service est une classe TypeScript annotée par @Injectable(). Il permet de faire de l'injection de dépendance. Cette dernière est possible en déclarant le service au sein d'un constructeur. La figure 4.15 nous montre l'annotation permettant de déclarer un service. On constate que cette annotation possède une donnée "providedIn : 'root'". Celle-ci permet de spécifier que le service est un singleton et permet de ne pas le déclarer au sein d'un NgModule. Il peut être injecté dans n'importe quel composant ou service.

FIGURE 4.15 – Annotation permettant d'associer les différents fichiers au composant en Angular

```
@Injectable({
    providedIn: 'root'
})
export class SpecificationFacade {
```

Pour bien comprendre la déclaration des différents composants, services et autre module, expliquons le fichier "app.module.ts" de l'application ProCalg V2. Une partie du contenu du fichier se trouve sur la figure 4.16. On constate que l'annotation NgModule possède trois compartiments. Le premier est la déclaration des composants. Il suffit d'indiquer le nom des composants dans le tableau précédant la clé "déclaration". Le deuxième permet l'import d'autres NgModule. Son nom de clé est "imports". Le troisième spécifie le composant d'entrée de l'application. Dans notre exemple, c'est le composant AppComponent. Le composant d'entrée est le composant chargé et affiché lors du chargement de l'application. Il est le composant dans lequel tout commence. Une quatrième métadonnée est possible mais ne figure pas sur la figure 4.16, elle déclare un tableau après la métadonnée "providers". Il permet de spécifier les services lorsque ces derniers ne sont pas annotés avec la métadonnée "providedIn : X ". A chaque déclaration du module une instance est fournie à la différence du provdedIn root qui indique un singleton.

FIGURE 4.16 – App module de ProCalg V2

```
@NgModule({
    declarations: [
        AppComponent,
        ElevatorComponent,
        Step1Component,
        Step2Component,
        ChooseNextComponent,
        ProcessTraceComponent,
    ],
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        InputTextareaModule,
        ButtonModule,
        MessagesModule,
        MessageModule,
        SkeletonModule,
        KnobModule,
        ChartModule,
        DialogModule,
        DropdownModule,
        FormsModule,
        HttpClientModule
    ],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

On observe qu'Angular apporte une structure et de la modularité au code. Cette structure permet de comprendre facilement un autre projet Angular. Cette facilité est accentuée par sa documentation qui propose des bonnes pratiques, des tutoriels complets et des explications détaillées de son fonctionnement. Sa documentation est accessible sur son site officiel à l'adresse : https ://angular.io/guide/architecture. La lecture de son tutoriel permet une bonne maîtrise d'Angular.

4.5.3 Les observables

Les observables permettent de faire passer des messages entre les différentes parties d'une application [36]. Elles sont fréquemment utilisées dans Angular et constituent une technique de gestion des événements, de programmation asynchrone et de traitement des valeurs multiples [36]. Le modèle se base sur le pattern observateur. Ce pattern permet à des observateurs de s'abonner à un sujet qui maintient une liste de ses dépendances et un état, d'être notifié en cas de changement de cet état. De manière imagée, l'observable est comme un tunnel dans

lequel passent des événements que l'on peut écouter.

Pour le déclarer en Angular, il faut créer une instance de la classe Observable qui définit une fonction d'abonnement *subscibe*() [36]. Cette fonction peut prendre en paramètre un objet de type observator. Cet observator est un objet qui définit des méthodes pour trois types d'événements reçus. Le premier est le cas où une nouvelle valeur est entrée dans l'observable. Le deuxième est le cas où l'événement est une notification d'erreur. Le dernier est le cas où l'événement est une notification de fin d'exécution. Grâce à cet observator passé en paramètre de la fonction subscribe, l'écouteur peut à chaque nouvel événement inséré dans l'observable, effectuer un certain traitement.

Cette fonction subscribe renvoie une instance de la classe Subscription qui possède notamment la méthode unsubscribe [36]. Cette méthode permet d'arrêter l'écoute sur l'observable. Au vu des fonctionnalités de l'application et de ce qu'apportent les observables, on comprend que l'application en utilise énormément surtout dans la gestion de son état présenté dans la section "Gestion de l'état de l'application".

4.5.4 Amélioration de ProCalg

ProCalg est écrit en AngularJs. La différence entre AngularJS et Angular est énorme au point qu'on pourrait dire que ce sont deux langages différents et non une évolution. Il est difficile de comparer ces deux technologies pour définir les améliorations effectuées. Cependant, il est facile d'exprimer que la version deux apporte une mise à jour technologique.

D'un point de vu du code, ProCalg nécessitait beaucoup moins de lignes de codes. Cette expansion demande d'avoir une structure claire et bien définie qu'apporte Angular. On peut dire que la version 2 a su adapter ProCalg à cette expansion en changeant de technologie.

Etant donné que ProCalg V2 est plus une réécriture qu'une amélioration de l'existant au niveau frontend, aucune amélioration n'a été apportée à ce niveau.

4.5.5 Gestion de l'état de l'application

Afin de bien séparer les responsabilités, chaque élément doit avoir son propre rôle. Dans l'application, les composants s'occupent de la vue. Deux types de services existent ; ceux qui s'occupent de garder et de modifier l'état courant et ceux qui gèrent les appels à l'API du backend. Les composants ne peuvent communiquer qu'avec les services qui gèrent l'état. Cette architecture permet une grande lisibilité et une séparation des rôles qui est une bonne pratique de développement. Cette section explique le cœur de l'application à savoir la gestion de son état qui est affiché par les composants. L'état de l'application est scindé en deux. Le premier est l'état des spécifications géré par le service specification-loader. Le deuxième est l'état de la spécification en cours d'exécution. Ce dernier est géré par le service specification-executor. Les deux services permettent aux composants de venir s'abonner au moyen d'un observable sur leur état. Lorsqu'un changement d'état arrive, les composants sont avertis et peuvent faire le nécessaire pour gérer la vue de ce nouvel état.

Prenons un exemple simple pour bien comprendre le principe. Dans le cadre de la création d'un compteur, il est possible d'avoir l'état du compteur dans un service et deux composants permettant d'afficher cet état. Un composant counter-view qui affiche la valeur du compteur. Un autre, que l'on nomme counter-view-plus-one, qui affiche la valeur du compteur plus un. Ces deux composants s'abonnent à l'état du service. La valeur du compteur dans l'état commence à 0. Le premier compteur affiche donc 0 et le deuxième affiche 1. Un autre composant injectant le service pourrait afficher un bouton d'incrémentation du compteur. Lorsque l'utilisateur clique sur le bouton d'incrémentation, le composant demande au service de modifier son état interne en augmentant le compteur. Après la modification de l'état, les deux composants abonnés sur la valeur du compteur de cet état reçoivent un nouvel événement contenant la nouvelle valeur du compteur. A la réception de l'événement les deux compteurs mettent à jour leur vue pour indiquer la valeur 1 et l'autre la valeur 2. Cet exemple simplifié représente bien la manière dont ProCalg V2 gère ses états.

Le loader de spécification retient plusieurs informations. La première est l'étape courante qui permet de savoir si une spécification a déjà été chargée ou non. La deuxième comporte les spécifications de l'utilisateur sauvegardées ou non. La troisième est un message d'erreur qui est modifié lorsque le backend renvoie une erreur

sur une spécification. En plus de devoir conserver ces trois informations qui forment l'état de spécification, il est de son ressort de manipuler le service permettant de faire des appels au backend. Il représente donc le patron de conception façade pour les composants étant donné qu'il cache de la complexité derrière de simples méthodes. Lors de son initialisation, il va chercher les spécifications enregistrées de l'utilisateur, se place à l'étape 1 qui est celle qui stipule qu'aucune spécification n'a encore été soumise et place le message d'erreur à null. A chaque changement, les composants abonnés à l'état sont avertis.

Le service d'exécution est plus complexe. L'interface de son état se trouve sur la figure 4.17. On constate qu'il a la charge de retenir et de permettre la modification de toutes les informations utiles aux ascenseurs. De plus, il est également le point d'entrée des composants pour la communication avec le backend même s'il le délègue à un autre service.

FIGURE 4.17 – Etat du service d'exécution

```
export interface State {
    processes: ProcessDefinition[];
    actionPossible: Action[];
    actionsPossible: Actions[];
    currentMulti: Action[];
    liftPosition: Map<string, number>;
    oldState: State[];
    executedActionsPerProcess: Map<string, string[]>;
    executedActions: string[];
}
```

Il retient diverses informations. Les plus importantes sont les processus et leurs actions, les actions exécutables suivantes seules, les actions exécutables suivantes exécutables en parallèle, les positions des ascenseurs et les actions exécutées depuis le début de l'exécution. Les sous-sections suivantes expliquent en détail certaines fonctionnalités qui permettront de mettre en lumière l'utilité de ces diverses informations.

4.5.6 Sauvegarde, chargement, modification et suppression d'une spécification

Un des objectifs est de ne pas avoir de base de données. Pour ce faire, les spécifications sont placées dans le Local storage du browser de l'utilisateur. Cette solution permet d'avoir cette fonctionnalité sans demander d'authentification ou autres de l'utilisateur et ne pas avoir à le gérer dans le backend.

Lors d'une sauvegarde de spécification, le local storage est mis à jour et l'information de l'état de l'application contenant les spécifications sauvegardées est mise à jour. Cette mise à jour permet, lors de la demande de chargement d'une spécification, de proposer la nouvelle spécification enregistrée. La suppression et la modification fonctionnent de la même manière en supprimant et en écrasant dans le local storage et en mettant à jour l'état de l'application. Lors du chargement d'une spécification, la spécification est insérée dans la zone de texte correspondante.

4.5.7 Déplacement des ascenseurs

Les positions des différents ascenseurs sont maintenues dans l'état de l'application à l'aide d'une map. Cette map contient comme clé, le nom du processus et comme valeur, l'étage de l'ascenseur. Lorsqu'une action est sélectionnée, cet état est mis à jour prévenant le processus abonné sur sa position d'ascenseur de cette nouvelle valeur. La figure 4.18 indique la méthode appelée par chaque composant ascenseur, qui représente un processus, sur le service. Cette méthode prend l'observable states$ représentant l'état. Ensuite, elle récupère seulement la position des ascenseurs au sein des diverses informations de l'état. Pour ne pas recevoir les évènements des autres ascenseurs, elle récupère seulement la position de l'ascenseur du processus qu'elle représente. La ligne *distinctUntilChanged*() permet de ne pas recevoir d'événement si un événement arrive dans l'observable et que la dernière position envoyée n'est pas différente de la nouvelle.

FIGURE 4.18 – Etat du service d'exécution

```
getCurrentLiftPosition(processName: string): Observable<number> {
    return this.state$.pipe(
        map( project: state => state.liftPosition),
        map( project: liftPosition => liftPosition.get(processName)),
        distinctUntilChanged()
    );
}
```

A la réception d'un nouvel événement, elle actionne la transition de l'ascenseur. La fermeture et l'ouverture des portes en sont une. Pour la comprendre, il faut d'abord comprendre comment l'ascenseur est affiché. Ce dernier se présente en 3 parties. La partie de gauche représentant la porte de gauche est noire, la partie centrale est blanche et la partie de droite représentant la porte de droite est noire. Lorsque la partie centrale grossit, les deux autres parties rétrécissent. A l'inverse, si la partie centrale maigrit, les deux autres grossissent. Techniquement, on demande aux parties situées à l'extrémité de prendre cent pour cent de la place disponible grâce à la propriété css *width* : 100%. La partie centrale quant à elle possède une largeur fixe. En modifiant la place prise par la partie blanche, le logiciel offre une illusion d'ouverture ou de fermeture de porte d'ascenseur. Pour ce faire, le logiciel modifie la largeur de la partie centrale en modifiant la valeur en pixel de sa propriété css *width* Cette propriété est à 5px lorsque les portes sont fermées et à 1000px lorsque les portes sont ouvertes. Angular animation permet, sur base d'un changement de valeur d'une variable, d'effectuer un changement css. En déclarant une variable *doorState*, il est possible de placer la propriété css à 1000 px lorsqu'elle possède la valeur : "open" et à 5px lorsqu'elle possède la valeur"close". Pour ce faire, Angular incrémente ou décrémente la valeur jusqu'à atteindre celle spécifiée. La vitesse de cette incrémentation dépend de la configuration de l'utilisateur. En indiquant, 700ms comme temps de fermeture, il mettra 700ms pour incrémenter ou décrémenter la valeur width de la partie centrale jusqu'à la valeur spécifiée. Grâce à cela, le logiciel donne l'impression à l'utilisateur que des portes s'ouvrent ou se ferment.

En ce qui concerne le déplacement de l'ascenseur, il faut savoir que les étages de l'ascenseur sont placés en "position : relative" tandis que l'ascenseur est placé en "position : absolue" au sein des différents étages de l'ascenseur. Ceci permet de placer l'ascenseur au-dessus des cases et de le déplacer sans impacter la cage de l'ascenseur. En sachant qu'une case dans le logiciel possède toujours une hauteur de 70 pixels, il suffit de multiplier l'étage demandé par 70 pour obtenir la hauteur à laquelle l'ascenseur doit se placer pour arriver à la case désirée. De la même façon, Angular animation est utilisé pour augmenter la propriété css *top* indiquant la hauteur à laquelle est placée l'ascenseur.

La figure 4.19 montre la méthode qui, sur base du nouvel étage, effectue la transition de l'ascenseur. De manière textuelle, elle commence par demander la fermeture des portes, elle attend 700 millisecondes ce qui correspond au temps de fermeture des portes. Ensuite, elle demande de commencer le déplacement de l'étage courant jusqu'au nouvel étage passé en paramètre de la méthode. Après une attente de 700 millisecondes, elle referme ses portes. Ce temps correspond au temps de déplacement de l'ascenseur. Grâce à ce bout de code, l'illusion d'un ascenseur est donnée à l'utilisateur.

FIGURE 4.19 – Methode déplaçant un ascenseur

```
private moveElevator(newFloor: number) {
    of( args: '').pipe(
        tap( next: () => this.doorState = 'close'),
        delay( delay: 700),
        tap( next: () => {
            this.oldPosition = this.currentFloor * HEIGHT_OF_ONE_FLOOR;
            this.currentFloor = newFloor;
            this.state = 'startMove';
        }),
        delay( delay: 200),
        tap( next: () => this.state = 'endMove'),
        delay( delay: 500),
        tap( next: () => this.doorState = 'open'),
        first()
    ).subscribe();
}
```

4.5.8 Retour arrière

Pour le retour arrière, il est important de conserver l'état précédent. Effectivement, lorsque l'utilisateur clique sur précédent, l'état doit retenir les anciens éléments sélectionnables ainsi que les positions des ascenseurs à chaque nouvelle action réalisée. Il faut également communiquer au backend qu'un retour en arrière est effectué afin qu'il replace la spécification en cours à son état précédent. Pour ce faire, à chaque fois que l'application interroge le backend pour sélectionner une action, le service d'exécution de spécification conserve l'état précédent dans l'ordre au sein d'un tableau. Lors du clic de l'utilisateur, l'état est remplacé par l'ancien état ce qui permet de revenir à la configuration précédente et le backend est prévenu par HTTP. Les composants des ascenseurs abonnés sur l'état sont informés et modifient la vue. L'utilisateur se retrouve alors dans la configuration précédente.

Bien que le backend renvoie l'information des anciennes actions sélectionnables, il faut de toute façon retenir la dernière position des ascenseurs qui n'est pas renvoyée par ce dernier. Effectivement, le backend ne doit pas retenir de la logique du frontend. Malgré qu'il soit possible de ne retenir que cette information, pour des questions de rapidité d'affichage et en cas de lenteur de communication avec le backend, l'application retient tout l'état précédent et ignore le retour du backend. L'état n'étant pas très grand, le retenir complètement permet d'être ouvert à l'extension et fermé à la modification dans le cas où une autre information devrait être conservée pour effectuer un retour en arrière.

4.6 Questionnement

La création d'une application n'est jamais parfaite. ProCalg V2 ne l'est pas encore après la deuxième version. Le plus important est de remettre en question et de garder en perspective des solutions d'améliorations. Cette section se propose d'en lister quelques-unes.

Le choix d'utiliser le local storage comme "base de données" possède des désavantages tels que la perte des données à la suite d'un changement de navigateur. Elle comporte également des avantages tels que la simplicité, la non-demande d'une authentification et permet de rendre l'outil rapide d'utilisation. Ce dernier argument provient de l'envie d'arriver directement sur l'écran d'accueil demandant la spécification qui permet au programme de répondre rapidement à la demande d'exécution de spécification. Le premier questionnement à faire est de remettre en question le choix de ne pas prendre de base de données. Par rapport à la solution du local storage, elle possède l'avantage de pouvoir effectuer un changement de navigateur tout en conservant ses spécifications enregistrées. Comme toute solution n'apporte pas que son lot d'avantages, elle aurait l'inconvénient de devoir intégrer de la sécurité, de l'authentification afin de s'assurer qu'un utilisateur ne puisse modifier les spécifications de quelqu'un d'autre. Un autre désavantage est que la boite à outil mCRL2 ne possède pas d'authentification. Cette décision rendrait difficile son intégration à la boite à outils existante. Dans le cadre

de ProCalg V2, en vérifiant la balance des avantages et des inconvénients de ces deux options, la décision a été d'utiliser le local storage. Il faut cependant trouver des solutions aux problèmes de la perte des données suite à un changement de machine ou de navigateur. Peut-être qu'avoir un mécanisme d'export et d'import pourraient pallier le problème du changement de navigateur tout en conservant la simplicité de l'outil en permettant son intégration à la boite à outil mCRL2. Trouver une solution à ce problème fait partie des objectifs futurs au niveau technique de ProCalg V2.

Un autre défi technique de ProCalg V2 est qu'il est mono utilisateur. Il ne permet donc pas d'avoir plusieurs utilisateurs en même temps sur un même backend. Cette contrainte provient de l'ancienne application ProCalg qui ne le permettait pas et qui retient dans un singleton la spécification courante. Si un autre utilisateur venait à utiliser l'application sur un même backend, il écraserait la spécification en cours du premier utilisateur. Il demande à l'utilisateur de lancer le backend sur sa machine. Un idéal serait de pouvoir déployer le backend à un endroit et le laisser accessible à tous les utilisateurs. Pourtant, il serait plus confortable pour l'utilisateur de ne pas avoir à faire ce lancement. La réécriture a déjà fait les premiers pas dans cette direction en conservant le choix de scindement frontend-backend. Une solution à analyser pour la suite de ProCalg V2 est de ne pas retenir la spécification courante dans le backend mais dans le frontend. Cela permettrait au backend de recevoir en input une spécification sous forme d'arbre ainsi que l'identifiant unique de sélection dans le but de renvoyer l'arbre mis à jour ainsi que les actions exécutables suivantes. Cependant, il faut faire des tests pour vérifier que lorsque la spécification grandit le délai d'envoi et de réponse ne devient pas trop long. De plus, les navigateurs et les serveurs web ont tendance à imposer une limite de taille sur le body d'une requête. Une autre solution pourrait également être d'attribuer un numéro unique par utilisateur et de demander son envoi à chaque appel. On retiendrait dans une map le numéro unique de l'utilisateur et sa spécification courante. A nouveau, il faut peser le pour et le contre, ceci demanderait de vérifier l'identité de l'appelant pour être sûr qu'il ne modifie pas la spécification de quelqu'un d'autre. Un autre point important est de savoir quand on supprime les données de cette map. On peut dire que cette solution demande de se poser la question du temps de conservation des données. Une solution intéressante est celle de Scala JS. Le compileur Scala JS permet de compiler les sources de scala en son équivalent Javascript [37]. L'avantage réside dans la possibilité d'embarquer le code backend dans une librairie qui serait téléchargée et exécutée du coté du client. Dans ce cas, le soucis de mono utilisateur ne serait plus un problème étant donné que chaque utilisateur aurait sa propre version du backend. Il faudrait remplacer les api par des appels à la librairie pour pouvoir profiter de la force du dynamisme d'Angular et de la mettre en relation avec Scala JS. Une analyse devrait être effectuée pour voir les limites de cette version javascript de scala. Par exemple, il faudrait remettre en question l'application backend et savoir si l'application ne devrait pas être totalement faite en Scala JS. Il existe d'autres solutions qui devront être comparées à ces trois dernières pour en choisir la meilleure. Cette problématique est la plus haute priorité pour la suite du développement de ProCalg V2.

Une autre remise en question est celle de l'état d'exécution conservé dans le front end. Cet état pourrait être allégé en délégant une partie au backend. Ce dernier pourrait, par exemple, mémoriser en plus de la spécification courante, les actions exécutées par processus. Elle permettrait une réutilisation de la fonctionnalité si d'autres applications décident d'afficher différemment la spécification. Cependant, il est nécessaire de sélectionner les parties qui devraient l'être et celles qui ne le doivent pas. Par exemple, les positions des ascenseurs n'ont aucun intérêt à être placées dans le backend.

Une des envies futures pour cette application est de trouver les solutions qui correspondent le mieux à ces différents questionnements. Elle demande cependant une analyse approfondie.

Pour conclure cette section, on peut dire que le logiciel n'est pas encore dans une version suffisante pour une utilisation optimale de l'application. Elle reste cependant, suffisante à l'utilisation. Il est possible de lancer le backend et le front end et d'effectuer un parcours dynamique et d'en retirer beaucoup d'informations. Il faut maintenant se pencher sur la partie déploiement et réduire un maximum les contraintes actuelles précitées pour favoriser le confort de l'utilisateur.

4.7 Conclusion

L'utilisation des langages Scala et TypeScript ainsi que des Framework Spring boot et Angular semblent avoir été une bonne décision. Il reste cependant impossible de dire s'ils ont été la meilleure.

Concernant le backend, la réutilisation du parseur a permis de ne pas recommencer à zéro l'application. Sa capacité a être interrogée par HTTP aide grandement à sa réutilisation et fournit une possibilité de créer d'autres applications d'exécution de spécification s'appuyant sur les informations de ce dernier.

Concernant le frontend, la mise à jour technologique permet d'éviter de tomber dans une application dans laquelle le Framework n'est plus maintenu. La puissance d'Angular a permis de simplifier le développement. En quelques lignes de codes, il est possible de fournir l'illusion d'un ascenseur et de gérer la réactivité de fonctionnalité tels que le retour en arrière, le déplacement des ascenseurs et bien d'autres. Il faut rester conscient que la mise en place d'une application de ce type reste un défi. Le défi est tout autant de la rendre fonctionnelle que de parvenir à prendre de bonnes décisions techniques. Ce dernier est aussi expliqué par le temps disponible pour cette application. La présence de cette contrainte augmente l'importance de ces décisions techniques. Un mauvais choix peut se révéler fatal pour la mise en œuvre des fonctionnalités désirées si le temps ne permet pas de tout recommencer.

On peut dire que l'intérieur de l'application possède de bonnes bases. Cette nouvelle version apporte une belle mise à jour au niveau technologique. Elle doit cependant encore connaître des améliorations avant de parler de version de production

Chapitre 5

Conclusion

5.1 Contribution

Le mCRL2 permet de décrire les processus sous forme de spécification. Il a la force d'être court et intuitif tout en permettant de décrire n'importe quel enchainement de processus qu'ils soient parallèles, séquentiels, alternatifs ou même les deux avec des contraintes telles que la communication ou le blocage selon le choix d'en abstraire ou non des parties. Les outils permettent de raisonner, de tester, d'exécuter ou même de visualiser l'ensemble des chemins possibles.

Les outils disponibles possèdent cependant le problème de modifier la spécification mCRL2 entrée. Ce problème est introduit à cause d'une transformation qui est au cœur des différents outils : la forme linéaire. Pourtant, il est possible de ne pas passer par cette forme à condition d'en accepter une légère complexité supplémentaire au niveau du traitement. Un autre inconvénient existe comme le manque d'information concernant les processus. La vision est celle de voir un enchainement d'actions sans penser que l'utilisateur a souvent besoin de connaitre le processus dans lequel les actions sont enchainées. Le manque de dynamisme des outils limite également les utilisateurs dans l'imagination de l'exécution de leur spécification.

Toutes ces contraintes amènent à la réalisation du logiciel créé dans ce mémoire. Les objectifs de ce logiciel sont de ne pas passer par la forme linéaire, d'être dynamique et de fournir un maximum d'informations sur la spécification exécutée. De plus, l'application permet de raisonner sur l'exécution sans abstraire les processus. Les objectifs sont atteints, l'application amène même la possibilité de débugger de la spécification grâce à son option de retour au choix d'actions précédentes. Le résultat amène une solution qui n'avait pas encore vu le jour. Elle donnera peut-être l'envie de créer d'autres outils qui ne passent pas par la forme linéaire. Sa réalisation a permis de faire ressortir de nouvelles réflexions telles que celle sur l'utilisation du dynamisme des ascenseurs qui amène à de nombreux étages lorsque les compositions parallèles intra processus existent. Fonctionnellement, sa capacité d'évolution est encore grande.

Au niveau technique, elle se dirige vers une application web de production tout en ouvrant la possibilité, grâce à Angular, d'en créer une application de bureau ou de téléphone. Ces derniers demanderaient tout de même une revisite des composants pour les adapter à ce type de vue. Cependant, il est encore tôt pour la considérer comme une application de production. Il faut effectuer le changement vers une application multi-utilisateur avant de déployer le backend sur un serveur. Sans oublier que block, abstraction et communication n'ont pas encore été implémentés. De plus, il faudrait la faire passer par une période intensive de tests dans laquelle on ressortirait différents bugs restants. L'avantage de l'architecture existante est la description JSON complète des éléments exécutables suivants. Il permettra peut-être la naissance d'une autre application apportant une visualisation différente de celle proposée.

5.2 Perspectives

A titre de réflexion, la possibilité de lier une action et les processus à des objets dynamiques pourrait accroitre fortement le dynamisme. Par exemple, on pourrait avoir un processus représenté par un train, un autre par le passage à niveau et un autre par un ascenseur. Le premier exécuté est l'ascenseur qui possède comme dernière exécution le lancement du processus train qui exécute des actions avec le lancement du processus de passage à niveau au milieu. Lors de l'exécution, on visualiserait des personnes rentrées dans l'ascenseur exécuter des actions sous forme d'étage comme dans le présent mémoire. Ensuite, à la fin des exécutions d'action, on verrait des personnes prendre le train, ce qui représente le processus train. Le train tournerait en rond exécutant des actions à chaque gare se stoppant lors de l'exécution du processus passage à niveau et qui laisserait passer des voitures représentant ses actions. Cette idée demanderait à l'utilisateur d'associer son processus à un de ces trois types. Elle possède un dynamisme supérieur à celui de ProCalg V2. Cependant, est-elle réellement meilleure pour les utilisateurs ? Est-ce que l'augmentation du dynamisme ne diminuera pas le côté intuitif, rapide et simple du logiciel à cause des diverses configurations ?

Une perspective importante est celle de la mise en place de block, communication et abstraction. L'opérateur block demandera, lors du parsing, de créer un objet de la classe delta pour chaque action bloquée. Lors des deux parcours récursifs de la spécification sous forme d'arbre, la rencontre de delta stoppera le parcours de la branche. Concernant abstraction, il faudra déclarer, au même titre que delta, une classe Tau. Lors du parsing, un objet de la classe Tau devra être créé au lieu des objets actions. Lors des parcours récursifs, cette dernière sera simplement ignorée. L'opérateur de communication est plus complexe à implémenter. Il demandera de retenir, lors du parcours, toutes les communications possibles à l'aide d'un objet contenant les actions de la communication, les processus de ces dernières et le nom utilisé pour la communication. Ces communications devront être spécifiées dans les items exécutables suivants. Pour ce faire, une nouvelle entrée devra être ajoutée dans l'objet items exécutables suivants. Cette dernière permettra au front end de créer un étage avec le nom des actions renommées si elles sont intra-processus et de suggérer l'exécution, à l'aide d'une pop up, les actions qui communiquent entre processus.

En guise de conclusion, nous suggérons une remise en question de la linéarisation qui est au cœur des outils mCRL2. Au vu des avantages démontrés par cette application de ne pas utiliser la linéarisation, ne devrait-on pas trouver une autre façon de linéariser ? Apporte-t-elle réellement plus d'avantages que d'inconvénients ? Dans le cas où elle amène plus d'inconvénients, ne devrait-on pas simplement parcourir la spécification et la traiter sous forme d'arbre comme dans le présent mémoire ? Des réponses se situent probablement au niveau des créateurs de la forme linéaire dont nous espérons pouvoir un jour avoir leur éclaircissement.

Annexe 1 : procédure d'installation de l'application

Cette annexe décrit la procédure d'installation. Quelques prérequis sont nécessaires. Le premier est la nécessité d'avoir Docker installé et lancé sur sa machine. Le deuxième est que les ports 80 et 8080 de la machine doivent être disponibles.

Dans le but de simplifier l'installation pour le lecteur, une image Docker contenant le frontend ainsi que le backend sont téléchargeables depuis le registre de Docker.

Pour ce faire, il suffit de télécharger le fichier docker-compose.yml situé à la racine git de l'url :

$$\boxed{https://github.com/vdauwerj/memoire/blob/main/docker/docker-compose.yml}$$

Une fois téléchargé, il faut ouvrir un terminal et taper la commande suivante en remplaçant chemin_du_fichier par le chemin absolu du fichier docker-compose.yml téléchargé précédemment : "cd chemin_du_fichier". Ensuite, pour lancer l'application, il faut exécuter la commande "docker-compose up -d". A l'introduction de la commande, les deux images sont automatiquement téléchargées et lancées. A la fin du téléchargement et du lancement, l'application est joignable via un navigateur en introduisant l'url :

$$\boxed{http://localhost}$$

Information concernant les sources

Les sources de l'application sont disponibles via le git suivant : https ://github.com/vdauwerj/memoire

Bibliographie

[1] J. C. Baeten, T. Basten, T. Basten, and M. Reniers, *Process algebra : equational theories of communicating processes*, vol. 50. Cambridge university press, 2010.

[2] O. Croegaert, "Thesis : Réalisation d'un interpréteur pour le langage d'algèbre de processus µcrl2," 2016.

[3] D. I. F. Oppacher, "Université d'ottawa university of ottawa,"

[4] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. Van Weerdenburg, "The formal specification language mcrl2," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[5] D. Austry and G. Boudol, "Algebre de processus et synchronisation," *Theoretical Computer Science*, vol. 30, no. 1, pp. 91–131, 1984.

[6] J. C. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131–146, 2005.

[7] H. Bekić, "Towards a mathematical theory of processes," in *Programming Languages and Their Definition*, pp. 168–206, Springer, 1984.

[8] S. Kulick, "Process algebra, ccs, and bisimulation decidability," 1994.

[9] R. Milner *et al.*, "A calculus of communicating systems," 1980.

[10] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[11] J. A. Bergstra and J. W. Klop, "Algebra of communicating processes with abstraction," *Theoretical computer science*, vol. 37, pp. 77–121, 1985.

[12] J. A. Bergstra and J. W. Klop, "Process algebra for synchronous communication," *Information and control*, vol. 60, no. 1-3, pp. 109–137, 1984.

[13] J. F. Groote, A. Mathijssen, M. Van Weerdenburg, and Y. Usenko, "From μcrl to mcrl2 : Motivation and outline," *Electronic Notes in Theoretical Computer Science*, vol. 162, pp. 191–196, 2006.

[14] J. F. Groote, A. Mathijssen, B. Ploeger, M. Reniers, M. van Weerdenburg, and J. van der Wulp, "Process algebra and mcrl2," *IPA Basic Course on Formal Methods*, 2006.

[15] "BWorld Robot Control Software." `https://www.mcrl2.org`, 2011-2021. [Online ; accessed 06-02-2021].

[16] J. F. Groote, "The syntax and semantics of timed μcrl," in *CWI, PO BOX 94079, 1090 GB*, Citeseer, 1997.

[17] D. Sangiorgi, *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[18] D. Hirschkoff, *Calculs de processus : observations et inspections*. PhD thesis, Ecole normale supérieure de lyon-ENS LYON, 2009.

[19] J. F. Groote, A. H. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. J. van Weerdenburg, "Analysis of distributed systems with mcrl2," *Process Algebra for Parallel and Distributed Processing*, vol. 1, pp. 99–128, 2009.

[20] P. R. D'Argenio and J.-P. Katoen, "A theory of stochastic systems. part ii : Process algebra," *Information and Computation*, vol. 203, no. 1, pp. 39–74, 2005.

[21] `https://www.mcrl2.org/web/user_manual/download.html`. [Online ; accessed 08-Mars-2020].

[22] `https://www.mcrl2.org/web/user_manual/introduction.html`. [Online ; accessed 08-Mars-2020].

[23] Aad Mathijssen, Bas Ploeger, Frank Stappers, Tim Willemse, "Behavioural Analysis using mCRL2." `https://www.aadmathijssen.nl/files/ipacfm2008-06-26.pdf`, 2008. [Online ; accessed 16-03-2021].

[24] O. Bunte, J. F. Groote, J. J. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. Willemse, "The mcrl2 toolset for analysing concurrent systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 21–39, Springer, 2019.

[25] M. Khan and S. S. Khan, "Data and information visualization methods, and interactive mechanisms : A survey," *International Journal of Computer Applications*, vol. 34, no. 1, pp. 1–14, 2011.

[26] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*. Artima Inc, 2008.

[27] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 2004.

[28] P. Selby, R. Abbeloos, J. E. Backlund, M. Basterrechea Salido, G. Bauchet, O. E. Benites-Alfaro, C. Birkett, V. C. Calaminos, P. Carceller, G. Cornut, B. Vasques Costa, J. D. Edwards, R. Finkers, S. Yanxin Gao, M. Ghaffar, P. Glaser, V. Guignon, P. Hok, A. Kilian, P. König, J. E. B. Lagare, M. Lange, M.-A. Laporte, P. Larmande, D. S. LeBauer, D. A. Lyon, D. S. Marshall, D. Matthews, I. Milne, N. Mistry, N. Morales, L. A. Mueller, P. Neveu, E. Papoutsoglou, B. Pearce, I. Perez-Masias, C. Pommier, R. H. Ramírez-González, A. Rathore, A. M. Raquel, S. Raubach, T. Rife, K. Robbins, M. Rouard, C. Sarma, U. Scholz, G. Sempéré, P. D. Shaw, R. Simon, N. Soldevilla, G. Stephen, Q. Sun, C. Tovar, G. Uszynski, M. Verouden, and T. B. consortium, "BrAPI—an application programming interface for plant breeding applications," *Bioinformatics*, vol. 35, pp. 4147–4155, 03 2019.

[29] K. Afsari, C. M. Eastman, and D. Castro-Lacouture, "Javascript object notation (json) data serialization for ifc schema in web-based bim data exchange," *Automation in Construction*, vol. 77, pp. 24–51, 2017.

[30] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, "Json : data model, query languages and schema specification," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pp. 123–135, 2017.

[31] L. Bassett, *Introduction to JavaScript object notation : a to-the-point guide to JSON.* " O'Reilly Media, Inc.", 2015.

[32] C. Walls, *Spring Boot in action*. Manning Publications, 2016.

[33] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *European Conference on Object-Oriented Programming*, pp. 257–281, Springer, 2014.

[34] P. So, "Angular," in *Decoupled Drupal in Practice*, pp. 355–380, Springer, 2018.

[35] K. Schmiedehausen, "Single page application architecture with angular," 2018.

[36] B. Joshi, "Angular," in *Beginning Database Programming Using ASP. NET Core 3*, pp. 279–335, Springer, 2019.

[37] S. Doeraene, "Scala. js : Type-directed interoperability with dynamically typed languages," tech. rep., 2013.