MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un interpréteur pour l'algèbre de processus µCRL2

Croegaert, Olivier

Award date:
2016

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2015–2016

Réalisation d'un interpréteur pour le langage d'algèbre de processus µCRL2

Olivier Croegaert

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.

Résumé

La complexité grandissante des technologies d'information et de communication fait l'objet de nombreuses recherches. Parmi les défis que cela représente, la communication entre différents systèmes s'avère être un point important à modéliser.

En effet, pour réaliser des systèmes interconnectés, il est primordial de prédire et maîtriser la manière dont ils s'influencent l'un l'autre. Apparaît dès lors la nécessité d'utiliser des outils formalisant ces comportements.

Pour faire face à ces difficultés, l'outil mathématique se révèle un puissant allié. Le développement de l'algèbre de processus s'inscrit notamment dans cette logique. En outre, la réalisation d'un outil favorisant une démarche intuitive et interactive sur ces concepts semble s'imposer naturellement.

Le travail présenté dans ce mémoire aborde la synthèse des concepts mathématiques en jeu et propose l'implémentation d'un outil permettant de raisonner sur les processus concurrents et communicants au moyen de l'algèbre µCRL2.

Mots-clés : algèbre de processus, µCRL2, scala, interpréteur, parsing, concurrence, modélisation.

Abstract

The growing complexity of information and communication technologies is subject to a wide range of research. Among all the challenges this complexity involves, the communication between several systems is an important point to modelize.

To set up interconnected systems, it is vital to predict and control the way these systems affect each other. Therefore it is necessary to use the right tools to formalize their behaviour.

To face these challenges, mathematics seem to be a powerful ally. Process algebra development fits into this logic. Moreover, the conception of a tool to support an intuitive and interactive approach on those concepts, seems to be essential.

The work presented in this final paper gives an overview on the mathematical concepts we are dealing with. It offers a tool implementation allowing to reason on concurrent communicating processes by using µCRL2 process algebra.

Keywords : process algebra, µCRL2, scala, interpretor, parsing, concurrency, modeling.

Avant-propos

Sans les conseils, la bienveillance et la disponibilité du Professeur Jean-Marie JACQUET, promoteur de ce mémoire, il ne m'aurait pas été possible d'entreprendre, de poursuivre et d'achever ce travail.
Son aide et ses encouragements m'ont permis d'évoluer dans mes connaissances et dans l'élaboration de ce mémoire.
Qu'il trouve ici l'expression de mon sentiment de reconnaissance.

Ma gratitude s'adresse également à mon employeur, Oniryx, qui m'a fourni les moyens matériels et financiers pour m'accompagner dans ce cursus.

Je souhaite en outre exprimer ma reconnaissance envers mon ancien employeur, Icoms Detections, en la personne de Monsieur Philippe de Visscher, qui a également sponsorisé cette volonté de reconversion.

La réalisation technique de ce travail n'aurait probablement pas été aussi fructueuse sans le regard d'expert de mon collègue et ami Jonathan.

Pour leurs relectures et avis précieux, je remercie mes parents qui m'ont soutenu tout au long de ce parcours.

Enfin, un merci s'adresse à tous ceux qui, de près ou de loin, m'ont encouragé à atteindre mon objectif.

Table des matières

Chapitre 1

Introduction

La multiplicité des moyens technologiques permet, chaque jour, d'augmenter notamment le confort de vie, la productivité au sein d'une entreprise, l'automatisation de tâches et les communications d'informations de tous types à travers le monde.

Derrière cette simplicité apparente se cachent bon nombre de recherches et d'outils dont la complexité de fonctionnement interne se traduit par une utilisation simple et intuitive. Ainsi, il n'est plus à prouver que l'avènement de l'outil informatique dans les entreprises, et ensuite son interconnexion au travers de réseaux internes et externes, a permis un gain considérable au niveau du rendement et une efficience qui, au jour le jour, fait l'objet d'optimisations.

Pour faire face à la complexité liée à l'élaboration de ces outils, il s'avère nécessaire d'en étudier le comportement attendu. Cette réflexion passe tout naturellement par l'élaboration de modèles. L'outil mathématique est tout indiqué pour donner un cadre de raisonnement formel à des concepts compliqués.

Il y a quelques décennies encore, un processus était considéré comme une séquence d'actions déterminant, selon les données fournies, un résultat unique. Les multiples interconnexions de systèmes et les influences qu'ils peuvent avoir l'un sur l'autre ont rendu ce type de raisonnement obsolète. L'illustration de la figure 1.1, donne l'intuition au lecteur qu'un système peut avoir une influence sur un autre.

Figure 1.1 – Influence entre processus [24]

Ainsi, la chute de dominos qui, en "vase clos", se poursuivrait jusqu'à son terme, est interrompue par un système externe qui interagit arbitrairement avec elle.

Dans cette optique, de nombreuses études scientifiques ont été menées pour étendre les outils mathématiques et permettre de raisonner sur les processus concurrents et communicants. L'algèbre s'est révélée un moyen puissant pour adresser cette complexité. Des recherches scientifiques ont progressivement affiné des outils de modélisation algébrique.

Dans ce mémoire, nous ciblerons l'algèbre de processus µCRL2.

"(...) µCRL2 is a formal specification language with an associated toolset. The toolset can be used for modelling, validation and verification of concurrent systems and protocols. (...)" [6]

Une algèbre repose sur des raisonnements mathématiques. Pour commencer, nous aborderons, dans ce travail, les bases mathématiques sur lesquelles repose la définition de l'algèbre de processus µCRL2. Cette partie, faisant référence notamment à un ouvrage de Wan Fokkink, *Introduction to Process Algebra*[7], aura pour but de donner au lecteur les éléments utiles pour raisonner sur les algèbres de processus.

La suite aura pour objectif la conception d'un outil simple et intuitif d'interprétation de l'algèbre envisagée. Ce dispositif viendra se positionner au regard de la complexité et du manque de convivialité des outils proposés par l'éditeur du langage µCRL2. L'idée générale consiste à révéler au lecteur les possibilités de développement d'un outil convivial au moyen d'un POC [1].

Ensuite, après la présentation de l'utilisation attendue de l'interpréteur, une proposition d'implémentation et de choix technologiques sera abordée. Cette partie constitue en soi un défi. En effet, entre la compréhension des concepts et l'implémentation d'un outil, les choix techniques impliquent un investissement d'apprentissage considérable. De plus, nous avons à coeur d'aborder des technologies actuelles dont la maîtrise à court terme n'est pas des plus évidentes.

Les principaux éléments du programme développé feront alors l'objet d'illustrations et de commentaires. En outre, une version complète des sources sera également proposée. Pour clôturer ce dernier chapitre, une présentation de l'outil final sera illustrée, décrite et critiquée.

Nous conclurons enfin par une réflexion sur ce travail dans sa globalité, un retour d'expérience en tant que développeur, une auto-critique et des pistes d'amélioration.

1. Proof Of Concept

Chapitre 2

Algèbre de processus concurrents et communicants

Ce chapitre a pour but de mettre l'ensemble de ce travail dans son contexte.

Premièrement, nous abordons, de manière intuitive, les concepts importants sous-jacents à notre travail.

Ensuite, nous procédons à un bref historique de l'évolution des recherches dans le domaine des langages d'algèbre de processus. L'utilité de ces dernières est plus que jamais démontrée de nos jours. En effet, à l'heure actuelle, il est rare qu'un système agisse de manière autonome sans jamais interagir avec d'autres systèmes, tant au sein même d'un ordinateur qui utilise plusieurs processeurs pour effectuer une tâche qu'à l'échelle macroscopique où l'on peut considérer que la plupart des appareils dotés d'un processeur interagissent avec le reste du monde.

Nous posons ensuite le cadre avec une définition des principaux concepts de l'algèbre de processus, qui constitue un outil indispensable pour comprendre et savoir comment maintenir une cohérence et une efficacité dans un système malgré la complexité de ses interconnexions.

Finalement, nous exposons, de manière formelle, l'algèbre ACP [1] sur laquelle nous nous reposons pour la réalisation de ce travail.

2.1 Intuition

Une grande partie du monde réel peut être modélisée par des transitions, c'est-à-dire le passage d'un état vers un autre état par l'observation d'un évènement. Cette observation générique peut se décrire par le système suivant $etat_1 \xrightarrow{evenement} etat_2$.

La figure 2.1 nous montre le fonctionnement d'un distributeur de boissons chaudes sous forme d'un ensemble d'états possibles (les cercles) et de transitions étiquetées

1. Algebra of Communicating Processes

par des évènements dont l'observation permet de passer d'un état source vers un autre état.

Nous y observons qu'à partir de l'état initial "0", nous pouvons exécuter trois actions (*coinElse*, *coin*[10], *coin*[5]) qui mènent à l'état "1", "2" ou "6". De ces différents états, d'autres transitions étiquetées par d'autres évènements peuvent faire passer dans d'autres états du système.

Un processus possible consiste, par exemple, à introduire 5 euros (passer de l'état 0 à l'état 6 par la transition *coin*[5]), sélectionner un thé (passer de l'état 6 à l'état 5 par la transition *tea*) et récupérer le thé demandé (passer de l'état 5 à l'état 0 par la transition *cupOfTea*).

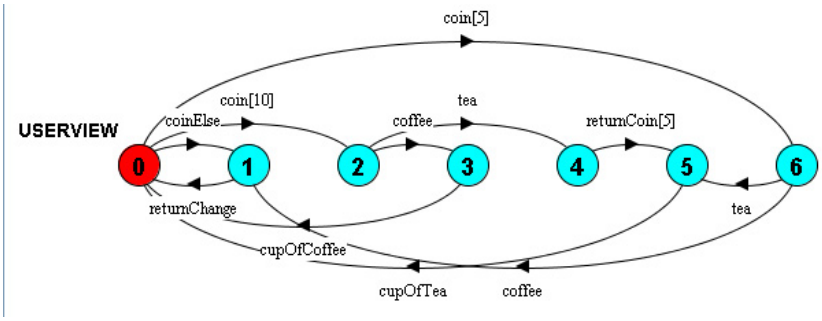


FIGURE 2.1 – Distributeur de café[5]

La figure 2.2 nous montre le fonctionnement d'un distributeur de billets qui met également en évidence différents états et transitions. Prenons par exemple l'état *Idle* dans lequel nous nous trouvons si l'automate a été démarré avec succès. Partant de cet état, l'évènement *cardInserted* nous amène dans l'état *ServingCustomer* depuis lequel il est possible à tout moment de repasser dans l'état *Idle* par un évènement *cancel*.

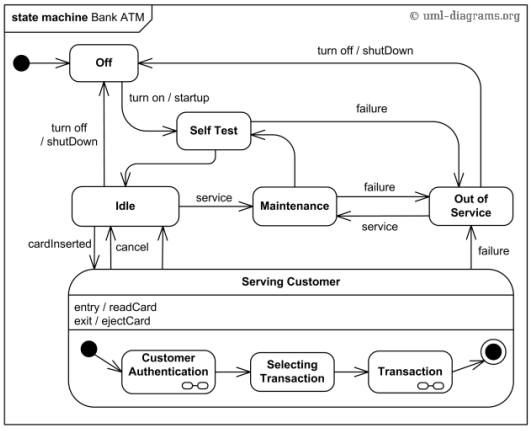


FIGURE 2.2 – Un autre exemple[22]

Ces exemples ont le mérite d'être intuitifs mais leur écriture est assez lourde et le résultat est peu abstrait. Une approche déclarative où l'on formalise textuellement la description d'un état source, d'un évènement et d'un état cible au moyen d'égalités nous permet une abstraction de cette complexité.

Reprenons notre machine à café de la figure 2.1. Par convention et pour alléger la lecture, nommons les états 0 - 6 $\rightarrow e_0 \cdots e_6$, les insertions de monnaie $coin[x] \rightarrow c_x$, les retours de monnaie $returnCoin[x] \rightarrow rc_x$. Si l'utilisateur a une pièce de 5 c_5 et une pièce de 10 c_{10} et qu'il souhaite un thé *tea*, nous aurons, schématiquement, ces deux systèmes de transitions possibles :

- $e_0 \xrightarrow{c_5} e_6 \xrightarrow{tea} e_5 \xrightarrow{cupOfTea} e_0$
- $e_0 \xrightarrow{c_{10}} e_2 \xrightarrow{tea} e_4 \xrightarrow{rc_5} e_5 \xrightarrow{cupOfTea} e_0$

On observe ici une notion de séquence d'évènements ainsi qu'une autre notion de choix. Nous pouvons imaginer une description textuelle dans laquelle nous pourrions modéliser ce choix par un opérateur et la séquence par un autre. Prenons + pour le choix et · pour la séquence :

$$e_0 \cdot (c_5 \cdot e_6 \cdot tea + c_{10} \cdot e_2 \cdot tea \cdot e_4 \cdot rc_5) \cdot e_5 \cdot cupOfTea \cdot e_0$$

On remarque un état initial commun, ensuite, comme explicité ci-avant, un choix entre la séquence de gauche ou de droite dans les parenthèses, puis une séquence commune de fin.

On peut aussi définir une règle d'égalité dans notre système qui, par exemple, considère que les deux choix sont équivalents car ils servent le même but final. Il en ressort une égalité du type :

$$e_0 \cdot c_5 \cdot e_6 \cdot tea \cdot e_5 \cdot cupOfTea \cdot e_0 = e_0 \cdot c_{10} \cdot e_2 \cdot tea \cdot e_4 \cdot rc_5 \cdot e_5 \cdot cupOfTea \cdot e_0$$

Cet exemple simple nous montre qu'il est possible, en définissant une syntaxe et des règles, de raisonner de manière mathématique sur des systèmes d'états-transitions au moyen de structures algébriques.

Ce besoin de représentation fait l'objet d'investigations depuis près d'un demi-siècle et nous abordons ci-après quelques principales avancées dans ces recherches qui nous mènent aujourd'hui à la réalisation de ce travail et de ce document.

2.2 Histoire des algèbres de processus

Au début des années 70, seuls les réseaux de Petri parlent de la concurrence. On peut à ce moment distinguer trois styles principaux de raisonnement formel sur les programmes informatiques. Le but premier consiste à donner un sens aux langages de programmation :

1. La sémantique opérationnelle : un programme est une machine abstraite. Un état de cette machine représente une certaine valeur pour chacune de ses variables. Les transitions entre les différents états sont des instructions élémentaires
2. La sémantique dénotationnelle : on abstrait d'avantage en définissant un programme comme étant une fonction qui transforme une entrée en une sortie (comme dans la figure 2.3).
3. La sémantique axiomatique (Hoare) : on se concentre sur des méthodes prouvant les programmes. Les notions principales sont les assertions, les triplets "précondition, instruction, postcondition" ainsi que les invariants.

Comment dès lors donner une sémantique à un programme contenant un opérateur parallèle ? Il s'est avéré, malgré les tentatives, qu'il était difficile de se contenter des trois méthodes ci-avant décrites. En effet, chaque style de raisonnement énoncé ci-dessus aborde un programme comme une suite d'instructions séquentielles.

Il y a deux importants concepts à revisiter avant de développer une théorie des programmes parallèles en termes d'algèbre de processus.

1. L'idée d'un comportement identique à une fonction prenant une entrée et produisant une sortie : on peut certes continuer à modéliser un programme comme un automate, mais en raison des interactions qu'un processus peut avoir entre l'entrée et la sortie, cette sortie peut être influencée (et donc modifier le comportement fonctionnel en cours de route). On ne peut donc plus considérer la notion d'équivalence de langage.
2. La notion de variable globale doit être éliminée : lorsque des processus parallèles s'exécutant indépendamment accèdent/modifient une variable globale, il est quasiment impossible de donner la valeur de cette variable à un moment donné. Il est beaucoup plus simple que chaque processus ait ses variables locales et échange explicitement des informations.

2.2.1 Bekič

Hans Bekič a, dans les années 60, travaillé sur la sémantique dénotationnelle de différents langages. Cela l'a amené à mettre en évidence un problème quant au fait de donner une sémantique pour la composition parallèle. Bekič propose une sémantique qu'il appelle "*exécution quasi-parallèle de processus*". L'opérateur choisi, //, deviendra l'opérateur || par après. Son travail a permis de donner des briques de base pour la suite dans la recherche sur l'algèbre de processus[2].

2.2.2 CCS

Dans les années 70, une théorie des processus, CCS [2], a été développée entre autres par Robin Milner et Matthew Hennessy. Cette théorie se focalise sur les problèmes causés par les programmes qui ne se terminent pas avec effets de bord et non-déterminisme. Aux opérateurs $*$ (composition séquentielle), ? (composition alternative) et $\|$ (composition parallèle) s'ajoutera ensuite un opérateur *préfixe* τ qui sera par après utilisé comme opérateur de synchronisation entre deux arguments.

Pour la première fois dans l'histoire, nous avons une algèbre de processus complète, avec un ensemble d'équations et un modèle sémantique[2].

2.2.3 CSP

CSP [3] est un langage au développement duquel a fortement contribué Tony Hoare. Il n'est plus question ici d'utiliser des variables globales mais plutôt de développer un moyen de communication par message entre les processus[2]. Le présent mémoire s'inspire d'un travail similaire réalisé avec CSP (voir [8]). Cette thèse de doctorat, effectuée par Marc Fontaine à l'université de Düsseldorf, a pour objet la réalisation d'un vérificateur de modèle pour le langage CSP. Ce vérificateur a été développé avec le langage Haskell[11] qui permet de développer de manière déclarative, ce qui est efficace lors de la définition d'une syntaxe et d'une sémantique. La manière de procéder pour la réalisation de notre interpréteur pour µCRL2 se rapproche technologiquement du langage Haskell car nous utilisons un langage, Scala, qui vient ajouter une couche fonctionnelle au-dessus de Java. Nous y reviendrons dans les chapitres suivants.

2.2.4 ACP

Le terme ACP [4] désigne l'algèbre de processus sur laquelle Jan Bergstra et Jan Willem Klop ont travaillé dans les années 80. La théorie qui va suivre se repose sur ACP tout en faisant des liens avec µCRL2 dont la définition est assez proche.

Une algèbre de processus sur un ensemble d'actions atomiques A est une structure $\mathcal{A} = \langle \mathsf{A}, +, \cdot, \between, a_i(i \in I)\rangle$ où A est un ensemble contenant A, les a_i sont des constantes qui correspondent aux $a_i \in A$ et $+$(*union*), $\cdot$(*concaténation*), $\between$(*combinaison à gauche*) satisfont pour tout $x, y, z \in \mathsf{A}$ et $a \in A$ suivant les axiomes[2] :

2. Calculus of Communicating Systems
3. Communicating Sequential Processes
4. Algebra of Communicating Processes

Table 2.1 – Axiômes pour ACP

PA1	$x + y = y + x$
PA2	$(x + y) + z = x + (y + z)$
PA3	$x + x = x$
PA4	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
PA5	$(x + y) \cdot z = x \cdot z + y \cdot z$
PA6	$(x + y) \between z = x \between z + y \between z$
PA7	$a \cdot x \between y = a \cdot (x \between y + y \between x)$
PA8	$a \between y = a \cdot y$

L'axiome PA8 nous permet de faire le lien entre la combinaison à gauche et l'opérateur de composition séquentielle (*concaténation*). Dans la suite de ce chapitre, nous verrons comment la composition parallèle $\|$ est liée à $\between$ et donc aux opérateurs de compositions alternative et séquentielle.

2.2.5 µCRL2

µCRL2 se base sur ACP et vient l'étendre avec la notion de donnée et de temps. Le concept fondamental est le processus. Un processus peut exécuter des actions qui peuvent être composées pour former d'autres processus en utilisant les opérateurs algébriques. Dans un système, on observe habituellement plusieurs processus qui s'exécutent en parallèle[6].

Dans µCRL2, une notion essentielle est la notion de processus linéaire. C'est un processus duquel tout le parallélisme a été retiré pour laisser place à une série de règles "condition-action-effet". Les systèmes complexes peuvent être traduits en un seul processus linéaire.

Les outils actuellement fournis pour µCRL2 transforment tout d'abord le code entré en processus linéaire avant de pouvoir le traiter ou l'analyser. Par exemple, un code µCRL2 simple :

```
act a, b;
proc P=a||b;
init P;
```

se traduira par le code suivant :

```
act  Terminate,a,b;
proc P(x1,x2: Pos) =
       (x1 == 1) -> a . P(x1 = 2)
     + (x2 == 1) -> b . P(x2 = 2)
     + (x1 == 2 && x2 == 2) -> Terminate . P(x1 = 3, x2 = 3)
     + (x1 == 1 && x2 == 1) -> a|b . P(x1 = 2, x2 = 2)
     + delta;
init P(1, 1);
```

Intuitivement, on démarre avec $x1$ et $x2 = 1$. Nous avons la possibilité d'emprunter la première, la deuxième ou la quatrième ligne. En prenant la première, on exécute d'abord a, puis on assigne 2 à la valeur $x1$ (implicitement, $x2$ garde sa valeur). On exécute ensuite $P(2,1)$, ce qui nous branche avec la deuxième ligne où l'on exécute b. On assigne ensuite 2 à $x2$ pour permettre d'exécuter $P(2,2)$ qui nous mène à la fin de l'exécution.

Comme le lecteur l'aura aisément observé, il découle de cette brève description qu'il est très difficile de comparer le comportement à étudier par rapport au code entré. Un des objectifs de ce travail est de pouvoir analyser le comportement à partir du code source sans recourir à la linéarisation.

La suite de ce chapitre a pour but de formaliser une base théorique sur laquelle reposera l'ensemble de ce travail.

2.3 Algèbre de processus

2.3.1 Processus

Un processus décrit le comportement d'un système. L'exécution d'une méthode dans un système informatique décrit un comportement de ce système. Par analogie, tourner la clé de contact de sa voiture, appuyer sur un interrupteur, demander son chemin, ... sont des actions qui démarrent un processus et déterminent le comportement d'un système.

La machine à café (figure 2.1) et le distributeur de billets (figure 2.2) mettent en évidence et de manière intuitive de tels systèmes de transitions. On peut considérer une vue du comportement qui peut être de différents niveaux d'abstraction et il importe de choisir le bon pour l'application envisagée. Par exemple, tenir compte des différents composants électroniques sollicités n'a pas de grande valeur dans le cadre d'une application de type "calculatrice".

Un processus sera aussi souvent décrit comme un système d'évènements discrets, en référence au fait que chaque action de ce système est considérée comme discrète (atomique ou non-décomposable). Si on prend le processus $AllumerLumiere$, une action incontournable est $PresserInterrupteur$. Compte tenu du niveau d'abstraction que nous pouvons ici aisément tirer du contexte, cette action est un *évène-*

ment discret, c'est à dire qu'il n'existe pas d'état possible entre l'interrupteur relevé et l'interrupteur baissé.

Par opposition, prenons un état initial *recette* et un état final *gateau*. Si on définit une action $Cuisiner$, il est évident que cette dernière pourra elle-même se décomposer en un système d'évènements discrets. Ce processus peut être interprété comme dans la figure 2.3 où $X = recette$ et $Y = gateau$. La boîte noire entre les deux représente tout un enchaînement d'évènements discrets prenant la recette en entrée et fournissant le gâteau en sortie.

FIGURE 2.3 – Système avec entrée-sortie

En descendant d'un niveau d'abstraction, nous verrions l'"intérieur" de cette boîte avec tous les états et transitions nécessaires à la production de la sortie.

2.3.2 Algèbre

Le modèle le plus abstrait est de voir un comportement comme une fonction prenant une entrée et fournissant une sortie. On donne une valeur en entrée, on reçoit une valeur en sortie (figure 2.3 ci-avant). Entre l'entrée et la sortie se déroule une suite d'actions qui permettent de produire la sortie désirée.

Un processus que nous pouvons imaginer à l'intérieur de la boîte noire peut ainsi être décrit comme une machine à états finis ou un automate. Un processus a un nombre d'états et un nombre de transitions, partant d'un état et allant vers un autre. Une transition représente l'exécution d'une action atomique.

Une algèbre qui permet de raisonner sur les automates est l'algèbre des expressions régulières (figure 2.4).

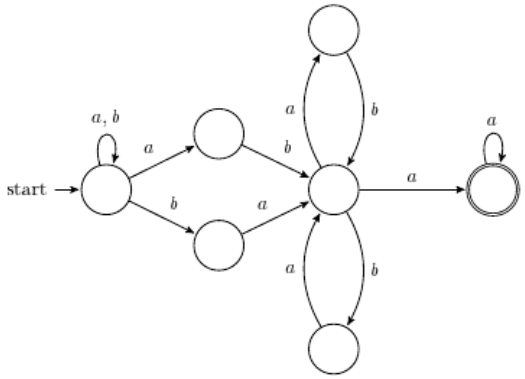


FIGURE 2.4 – Automate représentant l'expression régulière $(a+b)*(ab+ba)+a+$

Cette machine à état peut permettre de produire ou de valider une expression régulière (un élément d'un langage non contextuel). Nous apercevons, dans cet exemple, un parallèle entre une représentation schématique et une représentation algébrique qui ne perd pas de sémantique, est plus concise et, comme nous le voyons ci-dessous, permet de raisonner sur des règles définies dans le domaine d'application considéré.

"(...) L'idée principale en algèbre est d'utiliser des lettres pour représenter les relations entre les nombres sans spécifier quels sont ses nombres. On utilise des lettres ou des variables pour représenter les parties de l'expression qui peuvent prendre différentes valeurs, ou qui sont inconnues et que l'on veut résoudre (...)"[1].

Plus généralement, les structures mathématiques sont des ensembles qui comprennent une loi de composition associative, qui admettent un élément neutre et, pour chaque élément de l'ensemble, un élément symétrique [25].

Les règles suivantes, par exemple, doivent être respectées pour la multiplication arithmétique :

- $a*(b*c) = (a*b)*c$, (associativité)
- $u*a = a = a*u$, (élément neutre)
- $a*a^{-1} = a^{-1}*a = u$. (élément symétrique)

Ce type de syntaxe a une utilisation très répandue en arithmétique. Il est bien entendu possible de réutiliser ce paradigme en passant par une adaptation et une redéfinition dans le cadre, par exemple, de la définition de systèmes concurrents et communicants. C'est ce que nous appliquons dans la suite de ce travail.

L'interaction avec un autre système durant l'exécution doit être considérée dès lors qu'on introduit la notion de systèmes parallèles ou distribués. On modélise donc la concurrence par l'ajout de la composition parallèle dans les opérateurs de base.

L'algèbre de processus est donc l'étude des systèmes parallèles et offre la possibilité de décrire ces systèmes de manière algébrique au moyen des opérateurs de compositions alternative, séquentielle et parallèle. En raisonnant sur des équations, on peut vérifier que le système satisfait certaines propriétés (axiomes).

Dans le cadre de l'algèbre de processus, nous utilisons notamment les opérateurs suivants :

- $+$ pour la composition alternative.
- $\cdot$ pour la composition séquentielle.
- $\|$ pour la composition parallèle.

De la même manière que les axiomes pour la multiplication arithmétique, voici un exemple d'axiomes pour les algèbres de processus :

- $x + y = y + x$ (commutativité de $+$)
- $(x + y) \cdot z = x \cdot z + y \cdot z$ (distributivité de $\cdot$ par rapport à $+$)

En résumé, nous avons ici la définition d'une structure mathématique avec trois opérations présentant une algèbre de processus. Pour être une algèbre de processus, cette structure doit satisfaire aux axiomes qui lui sont définis.

Habituellement, ces structures sont exprimées en termes de système de transitions, c'est à dire avec un nombre d'états et de transitions entre eux, un état initial et un ou plusieurs états finaux[2]. Schématiquement, le processus noté algébriquement par $a \cdot b \cdot c$ se décrira comme ceci : $(a \cdot b \cdot c) \xrightarrow{a} (b \cdot c) \xrightarrow{b} c \xrightarrow{c} \surd$

Cela permet d'introduire, de manière intuitive, une autre notion importante : la *bisimulation*. Il s'agit ici de déterminer l'équivalence entre deux systèmes.

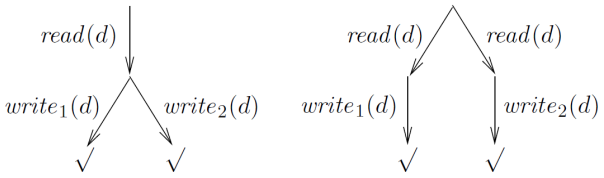


FIGURE 2.5 – Donnée lue et écrite sur deux disques 1 et 2[7]

Dans la figure 2.5, on observe à gauche qu'une donnée est lue, puis un choix est fait quant à la cible sur laquelle cette donnée sera écrite. Dans le deuxième cas, ce choix est fait avant la lecture de la donnée. L'ensemble des traces d'exécution est équivalent pour les deux cas mais si, par exemple, l'un des deux disques devient indisponible, le premier processus pourra choisir l'autre tandis que le second aura une chance sur deux de tomber dans un *deadlock*, c'est à dire que le système se retrouverait dans un état non-final depuis lequel aucune transition sortante ne pourrait être effectuée.

Intuitivement, l'inspection des transitions dans un système peu complexe comme celui schématiquement représenté dans la figure 2.5 peut faire ressortir qu'il n'y a donc pas de bisimulation entre ces deux processus. Il est aussi intuitivement évident que cette inspection peut devenir laborieuse dans un système complexe. Les axiomes viennent nous donner un moyen de raisonner rapidement au moyen d'équations algébriques sur des processus pour déterminer si, oui ou non, ceux-ci sont bisimilaires.

2.4 L'algèbre de processus ACP

Le but de cette section est de présenter l'algèbre de processus concurrents et communicants *ACP* en quatre étapes. La première sera de présenter une version d'ACP, que nous appellerons ACP_0, qui contient les opérateurs de composition de base. La deuxième étape présentera ACP_1 qui reprend toutes les possibilités d'ACP_0 en y incluant la composition parallèle. La troisième étape présentera ACP_2 où l'on vient enrichir l'algèbre précédente avec la constante deadlock δ et l'encapsulation ∂_H dont le but est de permettre de simplifier le graphe d'un processus parallèle. ACP_3, viendra ajouter la récursivité pour permettre les processus infinis. L'action silencieuse τ et l'abstraction font l'objet du dernier incrément ACP_4.

Ces algèbres sont illustrées par des exemples définis avec µCRL2.

Il est tout d'abord nécessaire de prévoir une section d'introduction d'éléments de syntaxe et leur sémantique car nous y faisons référence tout au long de ce chapitre. Une première sous-section a donc pour but de donner les définitions nécessaires à une compréhension commune.

2.4.1 Définitions

Signature

Une signature Σ consiste en un ensemble fini de symboles représentant des *fonctions* $f, g, \ldots$ où chaque fonction f a une arité $ar(f)$ qui donne son nombre d'arguments. Par exemple, si on a une fonction $f(p_1, p_2, p_3)$, alors $ar(f) = 3$.

Si une fonction g a une arité $ar(g) = 0$, il s'agit d'une constante. Si $ar(g) = 1$, c'est une fonction unaire. Si $ar(g) = 2$, il s'agit d'une fonction binaire.

D'autre part, nous émettons l'hypothèse qu'il existe un ensemble contenant une infinité de variables $x, y, z, \ldots$, ce dernier étant disjoint de Σ.

Terme

Soit Σ une signature (voir ci-avant). L'ensemble $\mathbb{T}(\Sigma)$ des termes $s, t, u, ...$ sur Σ est défini comme étant l'ensemble où :

- chaque variable est dans $\mathbb{T}(\Sigma)$;
- $si\ f \in \Sigma$ et $t_1, ..., t_{ar(f)} \in \mathbb{T}(\Sigma)$, alors $f(t_1, ..., t_{ar(f)}) \in \mathbb{T}(\Sigma)$.

Un terme est dit *fermé* s'il ne contient pas de variables. L'ensemble des termes fermés se dénote $\mathcal{T}(\Sigma)$.

La définition ci-dessus pose qu'un *terme* peut être l'argument d'un autre terme.

Dans un langage d'algèbre de processus, un terme est la représentation algébrique d'un processus. Soit, par exemple, un terme $t1$ qui représente un processus $p1$ et un terme $t2$ qui représente un processus $p2$. L'expression $t1 + t2$ est également un terme car elle représente un processus, celui qui exécute soit $p1$, soit $p2$. Le terme $t1 + t2$ représente un processus pour lequel l'état initial de $p1$ et l'état initial de $p2$ sont confondus (il s'agit du point de choix dans la composition alternative).

Transition

La machine à café (figure 2.1) et le distributeur de billets (figure 2.2) sont un exemple d'un système de transitions. Dans la même logique, la figure 2.6 illustre la notion de transition :

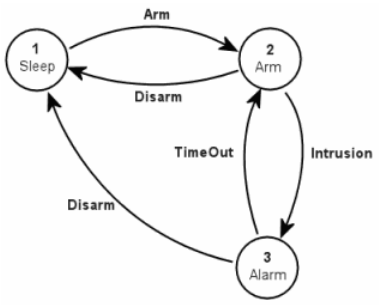


FIGURE 2.6 – Machine à états représentant une alarme[14]

La transition est matérialisée par la flèche qui mène d'un état à un autre par l'observation d'un évènement. Par exemple, la transition qui fait passer le système de l'état "armé" à l'état "alarme" s'exécute par l'observation de l'évènement "intrusion".

Une transition est une construction du type (s, a, s') où s, s' sont des états et a une étiquette de transition.

Système de transitions labellisées *LTS*

Pour reprendre formellement la définition de la transition, supposons disposer d'un ensemble non vide S d'états, d'un ensemble non vide A d'actions et d'un ensemble non vide $\mathcal{P}$ de *prédicats*.

Une transition est un triplet (s, a, s') avec $a \in A$, ou une paire (s, P) où $P \in \mathcal{P}$ et où $s, s' \in S$. Un LTS [5] est un ensemble de transitions (ensemble potentiellement infini). Un LTS se ramifie *de manière finie* si chacun de ses états a un nombre fini de *transitions* sortantes.

Dans la notation qui est utilisée dans ce document, les transitions (s, a, s') sont également notées $s \xrightarrow{a} s'$ (l'état s peut évoluer vers l'état s' par l'exécution de l'action a). De plus, une transition (s, P) est habituellement notée sP (le prédicat P est retenu dans l'état s).

Spécification d'un système de transitions *TSS*

Une règle de transition ρ est une expression de la forme $\frac{H}{\pi}$.

H représente les *prémices* de ρ. C'est un ensemble d'expressions de la forme $t \xrightarrow{a} t'$ et tP, avec $t, t' \in \mathbb{T}(\Sigma)$.

π représente la *conclusion* de ρ. C'est une expression de la forme $t \xrightarrow{a} t'$ ou tP, avec $t, t' \in \mathbb{T}(\Sigma)$.

La partie gauche de π est appelée la *source* de ρ. Une règle de transition est *fermée* si elle ne contient pas de variable.

Un TSS est un ensemble, fini ou infini, de règles de transition $\frac{H}{\pi}$.

Par exemple, une règle de transition valide serait :

$$\text{REGLE A} : \frac{}{x \xrightarrow{x} \surd}$$

On voit que la prémice est vide et donc on déduit la conclusion automatiquement. En d'autres termes, **rien** ($H = \emptyset$) nous permet de déduire que l'observation de l'action x fait passer de l'état x à l'état $\surd$ (état terminé) .

Un autre exemple ayant une prémice H non-vide serait

$$\text{REGLE B} : \frac{x \xrightarrow{x} \surd}{x + y \xrightarrow{x} \surd}$$

5. Labelled Transition System

Dans ce cas-ci, la règle dit que, sachant que l'observation de l'action x fait passer de l'état x à l'état $\surd$ (=H), on en déduit que l'observation de x sur un choix entre x et y fait également passer le système dans l'état $\surd$.

Un TSS génère un LTS (voir 2.4.1). Pour cela, on utilise la notion de *preuve* d'une règle de transition fermée d'un TSS (voir 2.4.1).

Preuve d'un TSS contenant une règle

Imaginons que, sur base des deux règles énoncées ci-avant, on nous demande de vérifier la cohérence de la transition suivante

$$a + b \xrightarrow{a} \surd$$

Voyons comment, ci-après, le vérifier de manière formelle.

La preuve d'une règle de transition fermée $\frac{H}{\pi}$ d'un TSS T est un arbre avec des transitions étiquetées vers le haut. Dans cet arbre, chaque chemin est fini, et les noeuds sont labellisés par des transitions, tel que :

- la racine a le label π
- si un noeud a le label ℓ, et que K est l'ensemble des labels des noeuds directement connectés au dessus de lui :
  - soit $K = \emptyset$ et $\ell \in H$
  - soit $\frac{K}{l}$ est une substitution d'une règle du TSS considéré.

Voici comment nous pouvons représenter cet arbre au moyen de nos règles définies plus haut (A et B) :

$$\dfrac{\dfrac{(\emptyset)}{a \xrightarrow{a} \surd}}{a + b \xrightarrow{a} \surd}$$

On peut observer, de bas en haut, la déduction de la preuve. Cet arbre a été obtenu par application des règles comme ceci :

$$\overline{x \xrightarrow{x} \surd} \quad \rightarrow \quad x := a$$

$$\frac{x \xrightarrow{x} \surd}{x + y \xrightarrow{x} \surd} \quad \rightarrow \quad x := a, y := b$$

Cette preuve va nous permettre par après d'une part, de déterminer les règles de transition dans un langage d'algèbre de processus, d'autre part de générer le LTS du processus.

Nous abordons plus loin d'autres applications concrètes de cette preuve.

2.4.2 Algèbre ACP_0

Syntaxe

Un processus est une composition d'actions. L'*action* est l'opération primitive d'un processus. Elle représente un évènement atomique et s'exécute elle-même avec succès.

$$a \xrightarrow{a} \surd$$

L'exécution de cette dernière est conditionnée par les dépendances qui lui sont définies par la composition dans laquelle elle se trouve.

$$a \cdot b \xrightarrow{a} b \xrightarrow{b} \surd$$

Cet exemple illustre une composition séquentielle des actions "a" et "b". Cette composition ne se termine avec succès que si "b" s'exécute et "b" ne peut s'exécuter que si "a" a pu s'exécuter avec succès.

Dans le langage µCRL2, l'action est représentée par une chaîne de caractères commençant par une lettre minuscule et est définie comme suit grâce au mot-clé `act` :

```
act action;
```

Il est également permis en µCRL2 de définir plusieurs actions en les séparant par des virgules :

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

Les processus sont quant à eux déclarés avec le mot-clé `proc` et définissent des compositions.

```
act a, b;
proc P = a . b;
```

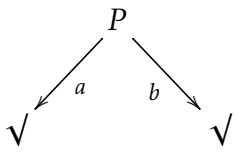
Dans le cas de ACP_0, nous allons aborder les compositions alternative et séquentielle.

Composition alternative

Cette composition, représentée algébriquement par l'opérateur binaire "+", définit un point de choix entre l'exécution de son membre de gauche et celle de son membre de droite. Ainsi, un *processus* "P" décrivant une composition alternative de deux *actions* "a" et "b" se décrit algébriquement par l'équation suivante :

$$P = a + b$$

Schématiquement, le processus P se représente comme suit :



En μCRL2, il se décrit par :

```
act a,b;
proc P = a + b;
```

Composition séquentielle

Cette composition, que nous représenterons ici par l'opérateur binaire "·", définit une exécution en séquence de son membre de gauche et ensuite de son membre de droite. Un *processus* Q décrivant une composition séquentielle de deux *actions* a et b se décrit algébriquement par l'équation suivante :

$$Q = a \cdot d$$

En μCRL2, il se décrit par :

```
act a,b;
proc Q = a . b;
```

Une représentation schématique figure déjà dans la section 2.4.2.

Combinaison de ces deux compositions

Il est bien entendu permis de combiner les opérateurs. Prenons l'exemple d'un automate bancaire qui permet soit de faire un retrait d'argent, soit de consulter son solde. L'automate correspondant est représenté dans la figure 2.7.

D'un état initial, l'évènement *inserCarte* nous amène dans un état où nous avons deux options, soit observer un évènement *retirer*20 qui nous amène dans un état,

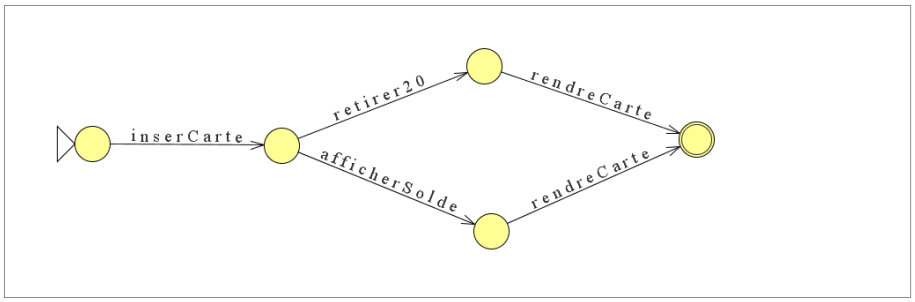


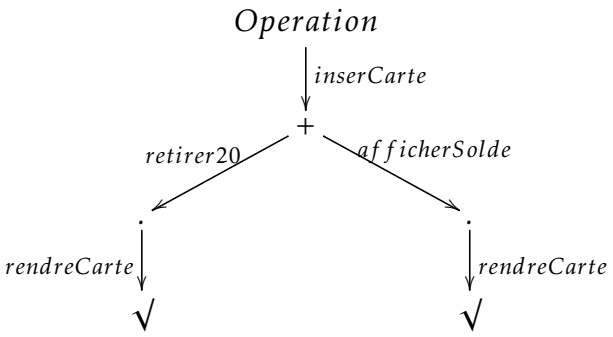
FIGURE 2.7 – Automate représentant un ATM simple.

soit observer un évènement *afficherSolde* nous menant à un autre état. De ces deux états, un évènement *retirerCarte* nous mène dans l'état *terminé*.

Voici la modélisation en µCRL2 :

```
act inserCarte, retirer20, afficherSolde, rendreCarte;
proc Operation = inserCarte.(retirer20.rendreCarte
                           + afficherSolde.rendreCarte);
```

Avec le mot-clé `act`, nous définissons les actions atomiques pouvant être exécutées dans notre système. Ensuite, nous définissons le processus `Operation` qui représente une composition séquentielle de l'action `inserCarte` et du groupe entre parenthèses. Ce groupe représente une composition alternative dont les deux termes sont des compositions séquentielles. Une exécution du processus `Operation` mène donc à une séquence de `inserCarte` suivie d'une des deux compositions séquentielles dans le groupe entre parenthèses. En voici le graphe du processus :



Notons la ressemblance entre ce graphe et l'automate de la figure 2.7.

Nous verrons, à la fin de cette section sur ACP_0 qu'il est possible d'avoir plusieurs représentations de ce processus (voir 2.4.2).

En référence au point 2.4.1, les deux fonctions binaires s (composition séquentielle) et a (composition alternative) appartiennent donc à la signature du langage ACP_0 (Σ_{ACP_0}).

Règles de transition

Nous avons jusqu'ici introduit la syntaxe du langage que nous appelons ACP_0 dans le cadre de ce travail. Nous avons également évoqué, au moyen d'exemples et de graphiques, un lien avec la sémantique qui y est associée. Nous allons ici formaliser ce lien pour lui donner du sens et en avoir une compréhension commune. Nous allons dans ce cadre appliquer la *sémantique opérationnelle structurelle* que nous introduisons ci-après.

Règles pour ACP_0

En posant que $\surd$ dénote l'état *terminé* :

- La première règle de transition formalise que chaque action atomique v se termine avec succès en s'exécutant :

Règle 1 $\dfrac{}{v \xrightarrow{v} \surd}$ Dans ce cas, H = $\emptyset$. Cela signifie qu'il n'y a pas prérequis pour valider π (ici = $v \xrightarrow{v} \surd$).

- Les quatre règles suivantes formalisent que chaque composition $t + t'$ exécute soit t, soit t', avec un choix sur base du premier pas :

Règle 2 $\dfrac{x \xrightarrow{v} \surd}{x + y \xrightarrow{v} \surd}$ Cette règle indique que si x se termine avec succès après l'exécution de v, alors le terme $x + y$ se termine également avec succès après l'exécution de v.

Règle 3 $\dfrac{y \xrightarrow{v} \surd}{x + y \xrightarrow{v} \surd}$ De manière symétrique, cette règle illustre le même cas que la précédente avec comme précondition que y se termine avec succès après l'exécution de l'action v.

Règle 4 $\dfrac{x \xrightarrow{v} x'}{x + y \xrightarrow{v} x'}$ On indique ici que, si l'exécution d'une action v sur un terme x mène à un terme x', alors l'exécution de cette action v sur le terme $x + y$ mènera également au terme x'.

Règle 5 $\dfrac{y \xrightarrow{v} y'}{x + y \xrightarrow{v} y'}$ Symétriquement, on observe un comportement identique dans le cas où l'action v sur y mène dans un état y'.

- Les deux règles de transition qui suivent formalisent que chaque composition $t.t'$ va exécuter t et, lorsque t aura terminé avec succès,t' pourra s'exécuter.

Règle 6 $\dfrac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y}$ Cette règle indique que si x se termine avec succès après l'exécution de l'action v, alors l'exécution de v sur $x.y$ mène au terme y.

Règle 7 $\dfrac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y}$ Dans le cas où l'action v mène du terme x au terme x', alors l'exécution de v sur $x.y$ mènera tout naturellement au terme $x'.y$. En d'autres termes, on obtient $x \xrightarrow{v} x' \cdot y$.

Pour mieux comprendre la *preuve d'un TSS* (expliquée au point 2.4.1), nous pouvons appliquer ces règles de transition à l'expression "$((a+b)\cdot c)\cdot d$" pour vérifier qu'elle se termine avec succès :

$$\dfrac{\dfrac{\dfrac{b \xrightarrow{b} \surd}{a+b \xrightarrow{b} \surd}}{(a+b)\cdot c \xrightarrow{b} c}}{((a+b)\cdot c)\cdot d \xrightarrow{b} c \cdot d}$$

On obtient cette preuve par application des règles de transition énoncées plus haut en les appliquant respectivement comme suit :

$$\dfrac{}{v \xrightarrow{v} \surd} \quad \rightarrow \quad v := b$$

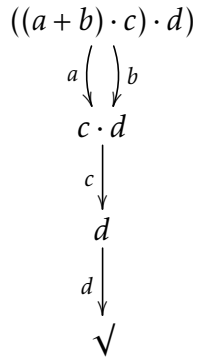
$$\dfrac{y \xrightarrow{v} \surd}{x+y \xrightarrow{v} \surd} \quad \rightarrow \quad v := b, x := a, y := b$$

$$\dfrac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y} \quad \rightarrow \quad v := b, x := a+b, y := c$$

$$\dfrac{x \xrightarrow{v} x'}{x \cdot y \xrightarrow{v} x' \cdot y} \quad \rightarrow \quad v := b, x := (a+b)\cdot c, x' := c, y := d$$

Une autre représentation que nous pouvons en tirer est le graphe du processus (ci-après) où l'on voit l'expression algébrique se réduire à chaque étape. On peut fa-

cilement la traduire en langage naturel par *"Après avoir exécuté l'action "a" ou l'action "b", on exécute l'action "c" et, ensuite, l'action "d""*.



Axiomes algébriques

Processus bisimilaires

Pour bien comprendre les axiomes, il est nécessaire d'approfondir le concept de bisimilarité entre processus. Comme expliqué dans la première section, l'équivalence de deux processus (c'est-à-dire le fait qu'ils puissent tous les deux exécuter la même séquence d'actions), n'est pas suffisante [6].

Si deux processus sont bisimilaires, ils peuvent non seulement exécuter la même séquence d'actions, mais ils ont également la même structure de branchement [7] :

Une relation $\mathcal{B}$ de bisimulation est une relation binaire sur les processus, telle que :

1. si $p\mathcal{B}q$ et $p \xrightarrow{a} p'$, alors $q \xrightarrow{a} q'$ avec $p'\mathcal{B}q'$;
2. si $p\mathcal{B}q$ et $q \xrightarrow{a} q'$, alors $p \xrightarrow{a} p'$ avec $p'\mathcal{B}q'$;
3. si $p\mathcal{B}q$ et $p \xrightarrow{a} \surd$, alors $q \xrightarrow{a} \surd$;
4. si $p\mathcal{B}q$ et $q \xrightarrow{a} \surd$, alors $p \xrightarrow{a} \surd$.

Deux processus p et q sont donc bisimilaires s'il existe une relation $\mathcal{B}$ telle que $p\mathcal{B}q$.

Ceci est noté par une relation d'équivalence $p \leftrightarrows q$, c'est à dire :

- $\leftrightarrows$ est réflexive : $p \leftrightarrows p$;
- $\leftrightarrows$ est symétrique : si $p \leftrightarrows q$, alors $q \leftrightarrows p$;
- $\leftrightarrows$ est transitive : si $p \leftrightarrows q$ et $q \leftrightarrows r$, alors $p \leftrightarrows r$.

A titre d'exemple, la figure 2.5 (introduction) montre qu'il n'y a pas de bisimulation entre deux processus d'écriture sur un disque qui, pourtant, ont l'air similaire à première vue.

6. Exemple dans la figure 2.5
7. Intuitivement, à tout moment, toute *action* que l'un peut exécuter peut être initiée par l'autre

Voici une notation algébrique de chacun de ces deux processus :

P_1 $read \cdot (write_1 + write_2)$
P_2 $read \cdot write_1 + read \cdot write_2$

On remarque que la règle 1. n'est pas respectée : imaginons l'exécution de *read* sur P_1 et l'exécution de *read* sur P_2 (cette dernière va nous faire choisir arbitrairement une des deux possibilités et nous prendrons la première pour l'exemple). Nous obtenons les processus résultants P_1' et P_2' :

P_1' $write_1 + write_2$
P_2' $write_1$

Nous constatons qu'il n'y a pas d'équivalence entre P_1' et P_2'. Sur base de la règle 1., nous pouvons donc dire que $P_1 \not\leftrightarrows P_2$.

Congruence

La congruence est, avec l'équivalence, une propriété de la bisimulation. Nous l'introduisons ici : Soit Σ une signature [8]. Une relation d'équivalence $\mathcal{B}$ sur $\mathcal{T}(\Sigma)$ est une congruence si pour chaque symbole de fonction $f \in \Sigma$,

$$\text{si } s_i \mathcal{B} t_i \text{ pour } i \in 1,...,ar(f), \text{ alors } f(s_1,...,s_{ar(f)}) \mathcal{B} f(t_1,...,t_{ar(f)})$$

En d'autres termes, si chaque argument d'une fonction a un terme équivalent, alors cette fonction est équivalente à la fonction dont chaque argument est remplacé par son équivalent.

L'intérêt est ici de mettre en évidence qu'il est possible de raisonner de manière compositionelle et donc de réfléchir sur des problèmes simples pour résoudre un problème complexe.

Axiomes pour ACP_0

Les axiomes sont des règles admises et permettent, dans le cadre de l'algèbre considérée, de donner un cadre formel de raisonnement. Ils ont ici été définis pour pouvoir rapidement établir la bisimilarité entre deux termes et, de cette manière, éviter l'inspection de chaque transition de chaque spécification à comparer. Voici les axiomes pour l'algèbre considéré dans cette section :

Table 2.2 – Axiomes pour ACP_0

A1	$x + y = y + x$
A2	$(x + y) + z = x + (y + z)$
A3	$x + x = x$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$

8. voir 2.4.1

Les règles ci-avant représentent une axiomatisation $\mathcal{E}_{\mathrm{ACP}_0}$ pour ACP_0 qui garantit la bisimilarité.

La relation d'égalité entre les termes dans $\mathcal{E}_{\mathrm{ACP}_0}$ pour ACP_0 s'obtient en prenant l'ensemble des instances de *substitutions fermées* des axiomes dans $\mathcal{E}_{\mathrm{ACP}_0}$ pour ACP_0 et en fermant cet ensemble par l'*équivalence* et le *contexte* (voir ci-après).

Substitution Posons une signature Σ. Une substitution est une fonction σ des variables vers l'ensemble des termes *ouverts* $\mathbb{T}(\Sigma)$. Une substitution s'étend vers une fonction des termes *ouverts* vers des termes *ouverts* : le terme $\sigma(t)$ s'obtient en remplaçant toutes les occurences des variables x dans t par $\sigma(x)$. Une substitution σ est *fermée* si $\sigma(x) \in \mathcal{T}(\Sigma)$ pour toutes les variables x.

Si "$s = t$" est un axiome et σ est une substitution, alors $\sigma(s) = \sigma(t)$.

Par exemple, posons l'hypothèse suivante dans le langage ACP_0 :

- $(a \cdot b + c) + (d \cdot e) = (d \cdot e) + (a \cdot b + c)$

La représentation algébrique ci-dessus est faite au moyen de la notation infixe. Si nous posons une fonction binaire f représentant la *composition alternative* $(+)$ et une fonction binaire g représentant la *composition séquentielle* $(\cdot)$, l'équation précédente peut être réécrite comme suit :

- $f(f(g(a,b),c),g(d,e)) = f(g(d,e),f(g(a,b),c))$

Cette représentation permet de faire ressortir la *signature* (voir le point 2.4.1) de notre langage ACP_0, Σ_{ACP_0}. Cette dernière représente l'ensemble (f,g) avec $ar(f) = 2$ et $ar(g) = 2$.

On voit l'usage des variables a,b,c,d,e qui sont utilisées avec Σ_{ACP_0} pour former des *termes*. Ces termes ne sont rien de plus que les symboles de l'ensemble Σ_{ACP_0} avec, comme arguments, des *variables* ou des symboles de fonctions appartenant à l'ensemble $\mathbb{T}$ des termes composés avec la signature Σ_{ACP_0} $(\mathbb{T}(\Sigma_{\mathrm{ACP}_0}))$.

Prenons l'axiome *A1*, nous y voyons deux variables : x et y. Reprenant la notation utilisée ci-avant, l'axiome peut se réécrire comme ceci :

A1. $f(x,y) = f(y,x)$

Posons les substitutions suivantes :

- $\sigma(f(x,y)) = f(f(g(a,b),c),g(d,e))$
- $\sigma(f(y,x)) = f(g(d,e),f(g(a,b),c))$

Des relations de substitution de l'axiome A1 vers nos deux fonctions, on tire les relations de substitution suivantes :

- $\sigma(x) = f(g(a,b),c)$
- $\sigma(y) = f(g(d,e))$

Nous pouvons donc réécrire notre égalité comme ceci :

- $f(\sigma(x),\sigma(y)) = f(\sigma(y),\sigma(x))$

Etant donné que, si $f(\sigma(x),\sigma(y)) = f(\sigma(y),\sigma(x))$, alors $f(x,y) = f(y,x)$, nous venons de montrer que l'égalité ci-dessus respectait l'axiome A1 au moyen de la substitution.

Equivalence La relation "=" est fermée par la réflexivité, la symétrie et la transitivité :

- $t = t$ pour tout t ;
- si $s = t$, alors $t = s$;
- si $s = t$ et $t = u$, alors $s = u$.

Contexte La relation "=" est fermée par le contexte : si $t = u$ et f est une fonction dont le nombre d'argument ar(f) > 0, alors

$$f(s_1,...,s_{i-1},t,s_{i+1},...,s_{ar(f)}) = f(s_1,...,s_{i-1},u,s_{i+1},...,s_{ar(f)})$$

En d'autres termes, si on remplace un argument d'une fonction f par un autre argument *équivalent*, alors f se comporte de la même manière.

Par exemple, si $b = y$ alors $a + b + c = a + y + c$

2.4.3 Algèbre ACP_1

Dans la pratique, un système est souvent composé de plusieurs processus qui s'exécutent en parallèle. Un processus peut dès lors influencer l'exécution d'un autre processus. Un système peut être considéré comme un ensemble de processus qui communiquent entre eux pour exécuter des tâches.

Pour illustrer l'influence qu'un processus peut avoir sur un autre, nous pouvons observer un automate qui vend un produit pour un montant donné (exemple tiré du site mcrl2[6]).

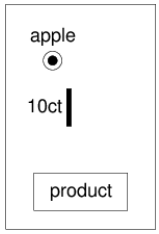


FIGURE 2.8 – Machine simple pour vendre un produit

La figure 2.8[6] illustre une machine qui permet l'insertion d'une pièce et la sélection d'un produit.

Le fonctionnement de cette machine peut être vu comme deux processus a priori indépendants. D'une part, l'utilisateur qui est capable d'insérer 10 centimes ($ins10$)

et de presser le bouton de son choix ($optA$). D'autre part, la machine qui est en mesure d'accepter 10 centimes ($acc10$) et de distribuer le produit souhaité ($putA$).

Une première représentation naïve de ces deux processus en µCRL2 est la suivante :

```
act ins10, optA, acc10, putA ;
init ins10 . optA || acc10 . putA;
```

Avec cette implémentation, nous pouvons avoir des entrelacements arbitraires entre le membre de gauche et le membre de droite. En effet, un entrelacement possible serait celui-ci :

$$acc10 \cdot ins10 \cdot putA \cdot optA$$

Intuitivement, on voit qu'il y a là un problème possible dans la séquence des évènements. Ceci est lié au contexte du problème envisagé. En l'occurrence, la machine ne peut accepter une pièce que si l'utilisateur l'a insérée, et elle ne peut distribuer un produit que si l'utilisateur a appuyé sur le bouton correspondant. La solution consiste à élaborer une communication entre les deux processus pour imposer, par exemple, que $ins10$ et $acc10$ ne puissent pas être dissociées. Il est ainsi nécessaire d'introduire un opérateur supplémentaire pour la communication.

Syntaxe

Pour modéliser cette communication, nous ajoutons à la signature d'ACP_0 (voir 2.4.2) trois nouvelles fonctions binaires de *combinaison* qui exécutent les deux porcessus des termes en argument en parallèle. En notation infixe, nous représentons ces opérateurs par $\|$ pour la *combinaison*, $\lfloor\!\lfloor$ pour la *combinaison gauche*, $|$ pour la *combinaison communicante*.

- $s\|t$: Ce terme peut choisir d'exécuter d'abord la transition initiale de s ou la transition initiale de t, voire les deux en même temps (c'est-à-dire se comporter comme $|$ (décrit ci-après).
- $s\lfloor\!\lfloor t$: Ce terme va d'abord exécuter la première transition de s (son membre de gauche) avant de continuer à se comporter comme $\|$.
- $s|t$: Ce terme va exécuter de manière synchrone la première action de s et la première action de t avant de continuer à se comporter comme $\|$.

Un exemple simple de combinaison en µCRL2 est :

```
act a, b;
P = a || b;
```

Comme décrit ci-dessus, l'action a ou l'action b peuvent se dérouler chacune en premier lieu ou elles peuvent se dérouler en même temps, ce qui se traduit en termes des opérateurs d'ACP_0 comme :

```
P = a.b + b.a + a | b;
```

On remarque ici qu'une composition parallèle de deux actions amène déjà à trois choix d'entrelacement, le dernier représentant simplement l'exécution simultanée de deux actions au moyen de l'opérateur |. S'il y a plusieurs actions dans chaque membre, le nombre de combinaisons possibles augmente davantage, et même de manière combinatoire, comme le montre l'exemple ci-dessous :

```
P = a.b || c.d
```

Cette composition s'exprime en termes des opérateurs d'ACP_0 comme ceci :

```
Q = 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)
```

L'exécution d'une *action* fait passer un système d'un *état* à un autre. Dans le cas du premier exemple $a||b$, on a la possibilité de partir de l'état initial en exécutant a vers un second état, en exécutant b vers un autre état ou en exécutant les deux actions simultanément pour passer dans l'état final. Ceci est illustré à la figure 2.9.

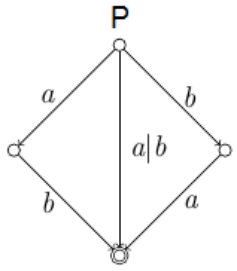


FIGURE 2.9 – Représentation des états possibles pour l'expression $a||b$

On remarque quatre états possibles du système. Dans une composition parallèle, si le membre de gauche a `m` états possibles et le membre de droite a `n` états possibles, le nombre d'états de la composition est de `m*n`.

On observe donc une explosion du nombre d'états lors de la définition d'une composition parallèle dont les deux membres sont eux-mêmes des compositions.

Nous aborderons plus loin (2.4.4) un moyen efficace de limiter le nombre de combinaisons possibles.

Règles de transition

Nous allons ici enrichir les règles de transition d'ACP_0 vues au point 2.4.2 avec cette nouvelle composition.

- Quatre règles de transition viennent s'ajouter :

Règle 8 $\dfrac{x \xrightarrow{v} \surd}{x \| y \xrightarrow{v} y}$ Si x peut arriver dans l'état terminé par l'exécution de v, alors $x \| y$ peut se retrouver dans l'état y par l'exécution de ce même v.

Règle 9 $\dfrac{x \xrightarrow{v} x'}{x \| y \xrightarrow{v} x' \| y}$ Variante de la règle précédente : si x peut évoluer vers x' par l'exécution de v, alors $x \| y$ peut se retrouver dans l'état $x' \| y$ par l'exécution de ce même v.

Règle 10 $\dfrac{y \xrightarrow{v} \surd}{x \| y \xrightarrow{v} x}$ Si y peut arriver dans l'état terminé par l'exécution de v, alors $x \| y$ peut se retrouver dans l'état x par l'exécution de ce même v.

Règle 11 $\dfrac{y \xrightarrow{v} y'}{x \| y \xrightarrow{v} x \| y'}$ Si y peut évoluer vers y' par l'exécution de v, alors $x \| y$ peut se retrouver dans l'état $x \| y'$ par l'exécution de ce même v.

- $s \| t$ peut également choisir d'exécuter une communication entre les transitions initiales de s et de t. Nous définissons donc ici une *fonction de communication*, notée $\gamma : A \times A \to A$, c'est-à-dire une fonction binaire qui représente une action résultant de la synchronisation de deux actions passées en argument.

  Autrement dit, si a et b sont des actions atomiques, $\gamma(a, b)$ formalise leur communication. Cette fonction de communication nécessite d'être commutative et associative, ce qui se traduit par :

  $$\gamma(a, b) \equiv \gamma(b, a)$$
  $$\gamma(\gamma(a, b), c) \equiv \gamma(a, \gamma(b, c))$$

  pour tout a,b,$c \in A$

Les quatre règles suivantes expriment que $s\|t$ peut choisir d'exécuter une communication (ou exécution synchrone) des transitions initiales de s et t.

Règle 12 $\dfrac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} \surd}$

Si x se termine avec succès par l'exécution de v et que y se termine avec succès par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x\|y$ se termine avec succès.

Règle 13 $\dfrac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} y'}$

Si x se termine avec succès par l'exécution de v et que y se retrouve dans l'état y' par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x\|y$ mène à l'état y'.

Règle 14 $\dfrac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x\|y \xrightarrow{\gamma(v,w)} x'}$

Symétriquement par rapport à la règle précédente, si x se retrouve dans l'état x' par l'exécution de v et que y se termine avec succès par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x\|y$ mène à l'état x'.

Règle 15 $\dfrac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x\|y \xrightarrow{\gamma(v,w)} x'\|y'}$

Les règles précédentes nous amènent intuitivement à cette règle : si x se retrouve dans l'état x' par l'exécution de v et que y se retrouve dans l'état y' par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x\|y$ mène à l'état $x'\|y'$.

- Les deux règles suivantes viennent s'ajouter aux précédentes avec l'opérateur $\lfloor\!\lfloor$.

Règle 16 $\dfrac{x \xrightarrow{v} \surd}{x\lfloor\!\lfloor y \xrightarrow{v} y}$

Si x se termine avec succès par l'exécution de l'action v, alors l'exécution de v sur $x\lfloor\!\lfloor y$ va mener à l'état y. En effet, l'opérateur $\lfloor\!\lfloor$ impose tout d'abord l'exécution de la première action de son membre de gauche.

Règle 17 $\dfrac{x \xrightarrow{v} x'}{x \between y \xrightarrow{v} x' \| y}$

De même, si l'exécution de v sur x mène dans un état x', alors l'exécution de v sur $x \between y$ va mener dans un état $x' \| y$. $\between$ a exécuté la première action de son membre de gauche, ensuite le résultat se comporte comme $\|$.

- Les quatre règles suivantes viennent compléter l'ensemble avec l'opérateur $|$.

Règle 18 $\dfrac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} \surd}{x|y \xrightarrow{\gamma(v,w)} \surd}$

Si x se termine avec succès par l'exécution de v et que y se termine avec succès par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x|y$ se termine avec succès. Intuitivement, nous pouvons factoriser $x|y$ comme étant un état e dont l'action initiale serait $\gamma(v,w)$.

Règle 19 $\dfrac{x \xrightarrow{v} \surd \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} y'}$

Si x se termine avec succès par l'exécution de v et que y se retrouve dans l'état y' par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x|y$ mène à l'état y'. Effectivement, les deux transitions des prémices H s'exécutent en même temps dans la conclusion π (rappel : 2.4.1).

Règle 20 $\dfrac{x \xrightarrow{v} x' \quad y \xrightarrow{w} \surd}{x|y \xrightarrow{\gamma(v,w)} x'}$

Symétriquement par rapport à la règle précédente, si x se retrouve dans l'état x' par l'exécution de v et que y se termine avec succès par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x|y$ mène à l'état x'.

Règle 21 $\dfrac{x \xrightarrow{v} x' \quad y \xrightarrow{w} y'}{x|y \xrightarrow{\gamma(v,w)} x'\|y'}$

Si x se retrouve dans l'état x' par l'exécution de v et que y se retrouve dans l'état y' par l'exécution de w, alors l'exécution synchronisée $\gamma(v,w)$ sur $x|y$ mène à l'état $x'\|y'$. En effet, comme décrit plus haut, $x|y$ va exécuter de manière synchronisée la première action de x et la première action de y, afin qu'ensuite le système se comporte comme une exécution parallèle sans les contraintes imposées par $\lfloor\!\lfloor$ ou $|$.

On remarque que la combinaison $\|$ peut être couverte par $|$ et $\lfloor\!\lfloor$. La relation de bisimilarité suivante vient intuitivement : $s\|t \leftrightarrows (s\lfloor\!\lfloor t + t\lfloor\!\lfloor s) + s|t$. En effet, la lecture de cette relation vient simplement nous dire que $\|$ est équivalent au choix entre commencer à gauche, commencer à droite, ou synchroniser l'exécution des deux actions.

Nous pouvons donc dès à présent améliorer notre exemple de l'automate qui vend un produit. Notre implémentation naïve en µCRL2 était la suivante :

```
act ins10, optA, acc10, putA ;
init ins10 . optA || acc10 . putA;
```

Nous devons donc imposer une communication entre $ins10$ et $acc10$, ainsi qu'entre $optA$ et $putA$. Dans la syntaxe de notre algèbre ACP, nous créons deux nouvelles actions $\gamma(ins10, acc10)$ et $\gamma(optA, putA)$. En µCRL2, l'implémentation devient la suivante :

```
act ins10, optA, acc10, putA, coin, ready ;

init comm(
      { ins10|acc10 -> coin, optA|putA -> ready },
      ins10 . optA || acc10 . putA);
```

Nous avons ici créé deux nouvelles actions *coin* et *ready* et nous les avons associées, grâce à l'opérateur `comm`, respectivement aux combinaisons communicantes $ins10|acc10$ et $optA|putA$. Cela reste néanmoins du renommage. Dans cette configuration, tous les entrelacements restent possibles. Il nous faut donc un moyen d'imposer que seuls *coin* et *ready* s'exécutent.

Pour l'instant, avec nos connaissances, la seule manière d'imposer ce comportement est de le coder "en dur" comme le montre l'exemple qui suit : Nous constatons qu'un workaround est possible avec nos connaissances dans ce cas-ci :

```
act ins10, optA, acc10, putA, coin, supply ;

init comm(
      { ins10|acc10 -> coin, optA|putA -> supply },
      coin.supply);
```

Nous "trichons" en imposant explicitement la séquence des deux fonctions de communication $coin \cdot supply$. Cela fonctionne, mais il n'y a plus de parallélisme et ce n'est pas ce que nous recherchons. En effet, dans un système plus complexe, nous pourrions obtenir plus de finesse en permettant/interdisant certaines actions ou communications. Cela fait l'objet de notre langage d'algèbre de processus ACP_2.

Axiomes algébriques

Cette section présente une axiomatisation $\mathcal{E}_{\mathrm{ACP}_1}$ pour ACP_1. Une relation d'égalité décrit une bisimilarité dans ACP_1 tenant compte de ceci :

- si $s = t$ peut être dérivé des axiomes de $\mathcal{E}_{\mathrm{ACP}_1}$ et que $s, t \in \mathrm{ACP}_1$, alors $s \leftrightarroom t$.
- si $s \leftrightarroom t$ est valable pour les termes $s, t \in \mathrm{ACP}_1$, alors $s = t$ peut forcément être dérivé des axiomes dans $\mathcal{E}_{\mathrm{ACP}_1}$.

Table 2.3 – Axiômes pour ACP_1

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	$x\|(y + z) = x\|y + x\|z$

Preuve de $\mathcal{E}_{\mathrm{ACP}_1}$

Les axiomes sont des règles permettant de raisonner sur un problème avec efficacité. Bien qu'ils soient considérés comme étant admis, ils sont bien entendu prou-

vables. Nous allons ici montrer intuitivement que $\mathcal{E}_{\mathrm{ACP}_1}$ est cohérent pour ACP_1 selon l'équivalence bisimilaire.

- M1 a été intuitivement démontrée à la fin de la section 2.4.3.
- LM2 et 3 définissent des axiomes pour la combinaison à gauche où l'on exprime simplement que $s \between t$ prend comme transition initiale la transition initiale de s.
- LM4 illustre la *distributivité à droite* de $\between$: dans le terme $(s+t) \between u$, un choix pour transition de s ou de t est un choix entre $s \between u$ ou $t \between u$, respectivement.
- CM5 à 8 sont les axiomes pour l'opérateur de communication. $s|t$ prend comme transition initiale une communication γ des transitions initiales de s et t.
- CM9 illustre la *distributivité à droite* de $|$: dans le terme $(s+t)|u$, un choix pour la transition de s ou de t est un choix entre $s|u$ ou $t|u$.
- CM10 illustre la *distributivité à gauche* de $|$: dans le terme $s|(t+u)$, un choix pour la transition de t ou de u est un choix entre $s|t$ ou $s|u$.

2.4.4 Algèbre ACP_2

Lorsque deux *actions* atomiques sont prévues pour communiquer, la plupart du temps, il n'est pas prévu qu'elles s'exécutent chacune individuellement. Prenons l'exemple d'un processus qui envoie une donnée par l'action *send* et qui la lit par l'action *read*. Cela représente une communication au travers d'un canal de communication. Pour le monde extérieur, il s'agit d'une et une seule action de communication *comm*. En d'autres termes, il n'est pas prévu qu'une donnée envoyée ne soit pas reçue [9].

Si nous voulons que l'action *send* soit systématiquement exécutée en *communication* avec *read*, nous devons interdire l'exécution individuelle de chacune de ces actions pour garantir le comportement de *comm*.

Par analogie, imaginons une action $actionnerInterrupteur$ et une autre action $activerLampe$. D'un point de vue extérieur, ces deux *actions* peuvent être perçues comme une seule $allumerLumire$.

Dans notre cas du distributeur de la figure 2.8, nous percevons deux actions qui peuvent être vues comme $coin$ pour l'action d'insérer la pièce de monnaie et comme $supply$ pour l'action de distribution du produit.

Nous allons voir dans cette section comment enrichir ACP_1 pour permettre d'exprimer cette nouvelle exigence.

Syntaxe

Pour forcer la communication dans les cas précités, il est nécessaire d'introduire une constante spéciale *deadlock*, notée δ, qui ne représente pas de comportement.

9. Nous faisons abstraction ici de l'éventuelle perte de données durant la communication

La fonction de communication γ est étendue pour permettre que la communication de deux actions atomiques mène à δ, c'est à dire :

$$\gamma : A \times A \to A \cup \{\delta\}$$

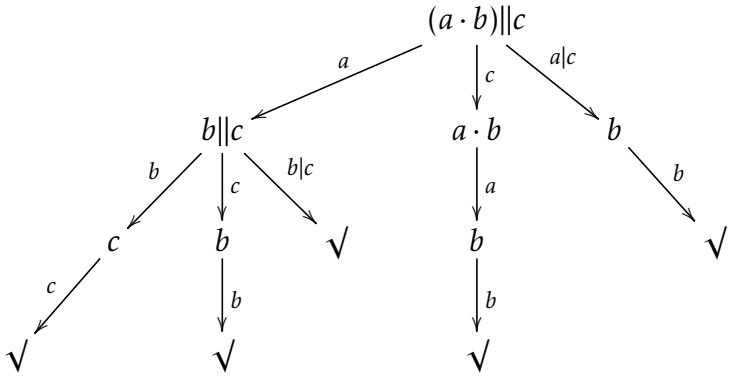
En d'autres termes, la communication de deux actions peut se traduire par un *deadlock* (autrement dit, on interdit cette communication), cela se ce note :

$$\gamma(a, b) \triangleq \delta$$

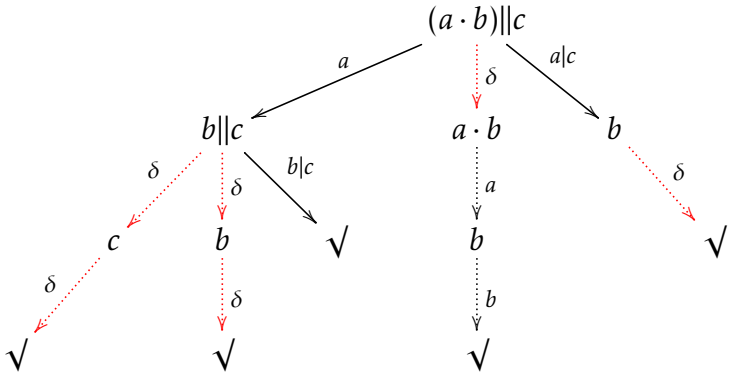
Cette notation peut également être utilisée pour exprimer qu'une communication entre deux actions peut se traduire par une autre action : $\gamma(a, b) \triangleq a$.

Un autre opérateur à introduire est l'opérateur d'*encapsulation* unaire ∂_H, H représentant un ensemble d'actions atomiques. Cet opérateur prend un terme en argument et remplace chaque action de l'ensemble H par δ.

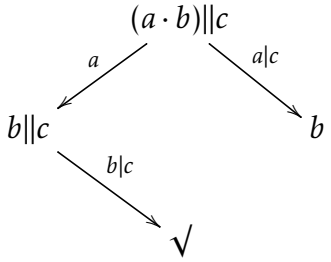
Prenons par exemple le processus $(a \cdot b)\|c$. Ce dernier peut être représenté, selon les règles de transition d'ACP$_1$(2.4.3), par le schéma suivant :



Reprenons ce processus en l'*encapsulant* de cette manière : $\partial_{\{b,c\}}((a \cdot b)\|c)$. Cela revient à "remplacer" les labels b et c par δ dans le graphe précédent.



Etant donné que δ empêche de parcourir un chemin jusqu'au bout, le graphe peut être simplifié en "élaguant" toutes les branches ayant une transition labellisée δ à leur racine :



On remarque dans le graphe résultant que, si on choisit la transition labellisée $a|c$, le système se retrouve dans un état "bloqué" ou en *deadlock*. Pour éviter ce cas, il faut remplacer $a|c$ par δ, c'est à dire rajouter la contrainte $\gamma(a,c) \triangleq \delta$. Notre graphe devient donc :

$$(a \cdot b)||c \xrightarrow{a} b||c \xrightarrow{b|c} \surd$$

Deadlock et *encapsultation* constituent un moyen puissant pour éviter le phénomène d'explosion combinatoire dans une composition parallèle (évoquée au point 2.4.3). Il s'agit également d'un moyen d'ajouter du sens à un comportement, comme dans le cas de notre distributeur de la figure 2.8.

Lors de notre dernière itération, nous avions manifesté le besoin de donner une autorisation à certaines actions de s'exécuter, et pas d'autres. En µCRL2, un opérateur similaire à ∂ existe mais, à la différence qu'au lieu de bloquer les actions qui lui sont associées, il les permet à la manière d'une "whitelist". Cette logique inversée ne nous handicape pas dans notre démarche :

```
act
  ins10, optA, acc10, putA, coin, supply ;

init
  allow(
    { coin, supply },
    comm(
      { ins10|acc10 -> coin, optA|putA -> supply },
      ins10 . optA || acc10 . putA
  ) ) ;
```

L'opérateur `allow` prend deux arguments : d'une part, un ensemble d'actions permises (ici *coin* et *supply*), d'autre part, un processus sur lequel cette restriction s'applique.

La constante δ existe également en µCRL2 et se note `delta`. C'est par cette constante que seront remplacées toutes les actions non permises.

Un programme tel que :

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

sera ainsi équivalent à

```
act a, b;
init a.b;
```

Les axiomes nous le confirment par après.

Règles de transition

Etant donné que δ n'a pas de comportement [10], il n'y a pas de règles de transition supplémentaires pour cette constante. Cela étant dit, le fait d'avoir étendu la fonction de communication en formalisant qu'une communication peut mener à δ, certaines règles de la section 2.4.3 doivent être adaptées. Plus précisément, les quatre règles sur la combinaison [11] ainsi que les quatre règles sur la communication [12] nécessitent d'expliciter l'exigence que $\gamma(v,w) \not\equiv \delta$. Autrement dit, il est maintenant nécessaire d'indiquer que ces règles ne seront valables que si $\gamma(v,w)$ peut s'exécuter.

En ce qui concerne l'opérateur d'encapsulation, on obtient les deux règles suivantes :

Règle 22 $\dfrac{x \xrightarrow{v} \surd}{\partial_H(x) \xrightarrow{v} \surd} \quad v \notin H$

L'explication de ces deux règles est assez triviale :

Règle 23 $\dfrac{x \xrightarrow{v} x'}{\partial_H(x) \xrightarrow{v} \partial_H(x')} \quad v \notin H$

$\partial_H(x)$ peut exécuter toutes les transitions de x (ici v) pour lesquelles les labels (ou *actions*) ne sont pas dans l'ensemble H.

10. Entendons par là que δ empêche de prendre la transition qu'il étiquette
11. Règles 12-15 de la section 2.4.3
12. Règles 18-21 de la section 2.4.3

Axiomes algébriques

Nous présentons ici les axiomes $\mathcal{E}_{\mathrm{ACP}_2}$ pour le deadlock et l'encapsulation. Nous donnons ensuite quelques intuitions pour vérifier leur cohérence.

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		$\delta \lvert x = \delta$
CM13		$x \lvert \delta = \delta$

- A6 nous montre que le deadlock δ n'a pas de comportement, donc l'opérande δ est redondante. On a ici un choix entre x et "rien".
- A7, LM11, CM12 et 13 montrent que δ bloque les comportements.
- D1,3 montrent que l'opérateur d'encapsulation ∂_H ne change ni δ, ni les actions atomiques qui n'appartiennent pas à H.
- D2 montre que ∂_H renomme toutes les actions qui appartiennent à H en δ.
- D4 et D5 montrent que $\partial_H(t)$ renomme toutes les transitions du terme t étiquetées avec des actions atomiques qui appartiennent à H.

2.4.5 Algèbre ACP_3

Les itérations précédentes de notre langage *ACP* ne prennent en compte que les processus finis. Nous allons voir dans cette section que des équations récursives permettent d'exprimer un comportement illimité.

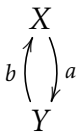
Pour reprendre le problème du distributeur, les spécifications µCRL2 proposées jusqu'à maintenant envisagent un usage unique. Nous souhaitons, dans un but de rentabilité de notre machine, qu'une fois le processus terminé, il puisse se répéter indéfiniment.

Nous allons, pour ce faire, introduire la récursion et revenir sur notre problème du distributeur.

Syntaxe

Prenons un processus qui exécute à l'infini alternativement les actions a et b. Sa représentation schématique serait la suivante :

Dans nos langages précédents, il n'existe aucune équation algébrique se traduisant par ce type de graphe. Renommons les deux états possibles de ce processus X et Y :



Intuitivement, on voit que pour quitter l'état X, on exécute l'action a, ce qui nous fait arriver dans l'état Y. De manière symétrique, pour quitter l'état Y, on exécute l'action b, ce qui nous fait arriver dans l'état X. On retrouve donc deux équations récursives :

$$X = a \cdot Y$$
$$Y = b \cdot X$$

Spécification récursive

Pour enrichir notre langage avec la récursivité, nous ajoutons la *spécification récursive*, qui est un ensemble *fini* d'équations récursives :

$$\begin{aligned} X_1 &= t_1(X_1, ..., X_n) \\ &\vdots \\ X_n &= t_n(X_1, ..., X_n) \end{aligned}$$

A gauche des équations, les X_i (avec $i \in \{1, ..., n\}$), sont appelées les *variables de récursion* ; du côté droit, les $t_i(X_1, ..., X_n)$ sont des *termes* ayant d'éventuelles occurences des variables de récursion X_i.

Pour reprendre les deux équations précédentes, et en considérant la fonction binaire f qui représente la *composition séquentielle*, nous avons, dans la notation préfixe :

$$X = f(a, Y)$$
$$Y = f(b, X)$$

Solution d'une spécification

Les processus $p_1,...,p_n$ sont une solution pour une *spécification récursive* $\{X_i = t_i(X_1,...,X_n) | i \in \{1,...,n\}\}$ **SSI** $p_i \leftrightarrows t_i(p_1,...,p_n)$ pour $i \in \{1,...,n\}$.

On voudrait qu'une spécification récursive ne représente qu'un seul processus et donc qu'il n'y ait, pour cette spécification, qu'une et une seule solution. Cependant, il en existe qui acceptent plus d'une solution. Quelques exemples de telles spécifications sont :

1. La spécification récursive $X = X$ accepte tout processus comme solution (on peut substituer X par n'importe quel processus, l'égalité restera valable).
2. La spécification $X = a + X$ accepte comme solution tout processus p qui est capable d'exécuter a comme transition initiale.
3. La spécification $X = X \cdot a$ accepte comme solution tout processus qui ne se termine pas avec succès.

Récursion gardée

La notion de récursion gardée nous permet d'introduire une forme de spécification récursive qui n'accepte qu'une solution.

La spécification récursive

$$\begin{aligned} X_1 &= t_1(X_1,...,X_n) \\ &\vdots \\ X_n &= t_n(X_1,...,X_n) \end{aligned}$$

est *gardée* si et seulement si la partie droite des équations récursives peut être adaptée sous la forme suivante :

$$a_1 \cdot s_1(X_1,...,X_n) + \cdots + a_k \cdot s_k(X_1,...,X_n) + b_1 + \cdots + b_\ell$$

avec $a_1,...,a_k,b_1,...,b_\ell \in A$ (ensemble des actions atomiques), par application des axiomes dans $\mathcal{E}_{ACP_2}$ (2.4) et en remplaçant les variables de récursion par la partie droite de leur équation récursive. Si k et ℓ sont tous deux = 0, la somme représente δ.

Par exemple, la spécification $X = (a \| b) \cdot X$ n'a, à première vue, pas la forme gardée. Elle peut cependant être réécrite comme ceci : $X = a \cdot b \cdot X + b \cdot a \cdot X + c \cdot X$, avec $c = \gamma(a,b)$. Nous constatons qu'il s'agit bien ici d'une récursion gardée.

Plus simplement, l'ensemble d'équations $\{X = a \cdot Y, Y = b \cdot X\}$ est aussi sous forme gardée. Observons que les équations suivantes offrent une solution :

$$\begin{aligned} X &= a \cdot b \cdot a \cdot b \cdot \cdots \\ Y &= b \cdot a \cdot b \cdot a \cdot \cdots \end{aligned}$$

Le lecteur n'aura aucun mal à se convaincre qu'il s'agit de l'unique solution du système.

Le distributeur revisité

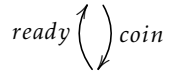
Voici ci-dessous la spécification µCRL2 du distributeur de la figure 2.8.

```
act
  ins10, optA, acc10, putA, coin, ready ;
proc
  User = ins10 . optA . User ;
  Mach = acc10 . putA . Mach ;
init
  allow(
    { coin, ready },
    comm(
      { ins10|acc10 -> coin, optA|putA -> ready },
      User || Mach
  ) ) ;
```

Nous voyons bien ici le mot-clé `proc` servant à définir des processus. Le processus $User$ est représenté algébriquement par une spécification récursive gardée et n'accepte donc qu'une solution$\langle ins10 \cdot optA \cdot ins10 \cdot optA \cdots$. De manière symétrique, le processus $Mach$ est aussi représenté par une spécification récursive gardée.

Ne permettant que $coin$ et $ready$, nous obtenons comme résultat un processus infini où $coin$ et $ready$ se succèdent indéfiniment.

Schématiquement, on obtient, tout simplement, le graphe suivant :



On remarque qu'en étant de plus en plus précis dans notre spécification, l'interprétation est de plus en plus simple. Nous sommes loin de l'explosion combinatoire d'états que donne l'usage naïf de l'opérateur ||.

Règles de transition

Soit E une *spécification récursive gardée* et X une variable de récursion $\in E$. Notons le *processus* $\langle X|E\rangle$ comme étant le processus qui doit être substitué à X dans la solution pour E. Par exemple, si E est l'ensemble $\{X = a \cdot Y, Y = b \cdot X\}$, alors $\langle X|E\rangle$ représente le processus $a \cdot b \cdot a \cdot b \cdots$ et $\langle Y|E\rangle$ représente $b \cdot a \cdot b \cdot a \cdots$.

Prenons la spécification récursive gardée E de la forme :

$$\begin{aligned} X_1 &= t_1(X_1, ..., X_n) \\ &\vdots \\ X_n &= t_n(X_1, ..., X_n) \end{aligned}$$

Pour étendre notre langage ACP_2, il nous suffit d'ajouter les deux règles de transition suivantes :

Règle 24 $\dfrac{t_i(\langle X_1|E\rangle,...,\langle X_n|E\rangle) \xrightarrow{v} \surd}{\langle X_i|E\rangle \xrightarrow{v} \surd}$

Si un des processus [13] dont on remplace chaque variable de récursion par sa *solution* peut exécuter une action v, alors la variable récursive que ce processus représente donne, une fois substituée dans son équation, un processus qui peut également exécuter cette action v.

Règle 25 $\dfrac{t_i(\langle X_1|E\rangle,...,\langle X_n|E\rangle) \xrightarrow{v} y}{\langle X_i|E\rangle \xrightarrow{v} y}$

Similaire à la règle précédente avec l'arrivée dans l'état y au lieu de l'état *terminé*.

Si on prend la spécification récursive gardée $E \triangleq \{X = a \cdot Y, Y = b \cdot X\}$, on obtient le graphe de processus suivant :

Comme expliqué plus haut, ces deux graphes représentent une notation différente du même processus. La transition $\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle$ peut être dérivée des règles de transition comme ceci :

$$\dfrac{\dfrac{a \xrightarrow{a} \surd}{a \cdot \langle Y|E\rangle \xrightarrow{a} \langle Y|E\rangle}}{\langle X|E\rangle \xrightarrow{a} \langle Y|E\rangle}$$

13. Une des équations du système

On obtient cette preuve par application des règles de transition en opérant respectivement comme suit :

$$\frac{}{v \xrightarrow{v} \surd} \quad \rightarrow \quad v := a$$

$$\frac{x \xrightarrow{v} \surd}{x \cdot y \xrightarrow{v} y} \quad \rightarrow \quad v := a, x := a, y := \langle Y|E\rangle$$

$$\frac{a \cdot \langle Y|E\rangle \xrightarrow{v} y}{\langle X|E\rangle \xrightarrow{v} y} \quad \rightarrow \quad v := a, y := \langle Y|E\rangle$$

Axiomes algébriques

Supposons que la spécification récursive gardée E pour les axiomes est de la forme :

$$\begin{aligned} X_1 &= t_1(X_1, ..., X_n) \\ &\vdots \\ X_n &= t_n(X_1, ..., X_n) \end{aligned}$$

On obtient les axiomes suivants :

Table 2.5 – Axiômes pour ACP_3

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

Intuitivement RDP [14] nous dit que $\langle X_1|E\rangle, ..., \langle X_n|E\rangle$ est une solution pour E, alors que RSP [15] nous montre que c'est la seule solution pour E, en tenant compte de l'équivalence en termes de bisimulation.

14. Principe de définition récursive
15. Principe de spécification récursive

2.4.6 Algèbre ACP_4

Lorsqu'un programmeur réalise un produit pour un client, il le fait sur base des exigences du client. Le client attend un certain comportement du programme lorsqu'il effectue une opération. Lorsque le programmeur implémente le programme, une question se pose : le comportement externe attendu est-il rencontré par l'implémentation ? Pour s'en rendre compte, il est nécessaire de s'abstraire des étapes de calcul interne.

Le but de cette section est de proposer un élément qui permet d'effectuer cette abstraction, en extension de notre langage ACP_3.

Syntaxe

Nous introduisons ici une constante spéciale, τ, appelée *action silencieuse*. Une transition τ représente une séquence d'actions internes pour pouvoir les éliminer d'un graphe de processus. Nous adaptons donc le niveau d'abstraction. Le choix du bon niveau d'abstraction a été évoqué dans la section 2.3.1. La constante τ peut s'exécuter d'elle-même, comme toute action atomique.

Nous étendons donc l'ensemble A des actions atomiques comme ceci : $A \cup \{\tau\}$. Le domaine de la fonction de communication γ est également étendu avec cette *action silencieuse* :

$$\gamma : A \cup \{\tau\} \times A \cup \{\tau\} \to A \cup \{\delta\}$$

en définissant que chaque communication comprenant τ mène à δ.

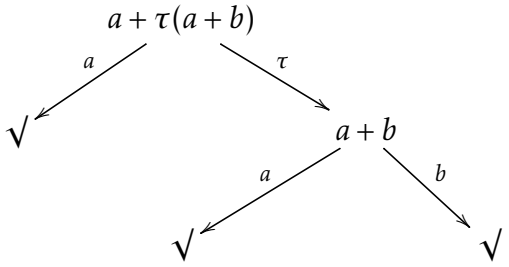
Un point d'attention est à porter au fait que, si deux processus p et q sont équivalents, et que p peut exécuter l'action silencieuse τ, alors q n'a pas besoin de simuler cette transition τ par l'exécution d'une action τ.

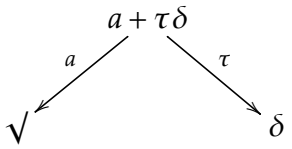
Si deux processus sont équivalents et que l'un d'eux comprend τ et l'autre pas, on en vient à l'évidence que l'action τ n'est pas tout à fait silencieuse.

Par exemple, si l'on considère le processus $a + \tau\delta$ et le processus a, ces deux termes seraient équivalents si τ était vraiment silencieuse. On remarque ici, dans le premier, terme que l'on peut choisir la transition τ qui nous mène à un deadlock.

La transition τ ci-dessus nous fait perdre la possibilité d'exécuter l'action a. Dans cette situation, τ n'est pas silencieuse. Pour avoir une transition τ "réellement" silencieuse, l'intuition est que cette transition ne perde aucun comportement possible après avoir été empruntée.

Prenons comme exemple les termes $a+\tau(a+b)$ et $a+\tau\delta$. Voici leurs représentations schématiques respectives :





Dans cette représentation, on observe qu'il y a tout d'abord un choix entre a et τ. Mais le fait de choisir τ conserve la possibilité d'exécuter a. Exécuter τ n'a aucune influence sur les choix possibles initialement. τ est donc bel et bien une action silencieuse.

Dans cette représentation en revanche, emprunter la branche de droite nous prive de la possibilité d'exécuter l'un des choix possibles initialement.

Bisimulation faible

Pour formaliser l'intuition abordée ci-dessus, nous introduisons la notion de *bisimulation faible* qui vient décrire une relation binaire sur un ensemble de processus, tel que :

1. si $p\mathcal{B}q$ et $p \xrightarrow{a} p'$, alors
   - soit $a \equiv \tau$ et $p'\mathcal{B}q$
   - soit il existe une séquence de transitions $\tau\ q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_0$ tel que $p\mathcal{B}q_0$ et $q_0 \xrightarrow{a} q'$ avec $p'\mathcal{B}q'$.
2. si $p\mathcal{B}q$ et $q \xrightarrow{a} q'$, alors
   - soit $a \equiv \tau$ et $p\mathcal{B}q'$
   - ou soit il existe une sequence de transitions $\tau\ p \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_0$ tel que $p_0\mathcal{B}q$ et $p_0 \xrightarrow{a} p'$ avec $p'\mathcal{B}q'$.
3. si $p\mathcal{B}q$ et $p \cdot P$, alors il y a une séquence de transitions $\tau\ q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_0$ tel que $p\mathcal{B}q_0$ et $q_0 \cdot P$
4. si $p\mathcal{B}q$ et $q \cdot P$, alors il y a une séquence de transitions $\tau\ p \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_0$ tel que $p_0\mathcal{B}q$ et $p_0 \cdot P$

Deux processus sont faiblement bisimilaires $p \leftrightarrows_b q$ s'il existe une bisimulation faible $\mathcal{B}$, tel que $p\mathcal{B}q$.

Plus intuitivement, si un terme bisimule un autre et que l'un d'eux a une transition initiale τ, alors le terme résultant après la transition τ est toujours bisimilaire à l'autre terme :

$$p\mathcal{B}(\tau q') \Longrightarrow p\mathcal{B}q'$$

De même, si un terme à partir duquel il est possible d'effectuer une série de transitions τ, suivie de l'action initiale de l'autre terme, alors les termes résultants après l'exécution de cette action sont faiblement bisimilaires :

$$(p \cdot a \cdot p')\mathcal{B}(p \cdot \tau \cdots \tau \cdot a \cdot q') \Longrightarrow p'\mathcal{B}q'.$$

Enfin, si un terme composé d'une séquence d'une action suivie d'une variable de récursion est bisimilaire à un autre terme, alors cet autre terme peut passer, après une série de transitions τ, dans un état où il bisimule toujours ce terme :

$$(p \cdot P)\mathcal{B}(q \cdot \tau \cdots q_0 \cdot P) \Longrightarrow (p \cdot P)\mathcal{B}(q_0 \cdot P)$$

Bisimulation faible à la racine

On observe néanmoins que la bisimulation faible n'est pas une congruence pour ACP_0. Par exemple, $\tau a + b$ et $a + b$ ne sont pas faiblement bisimilaires (si on exécute τ, on perd la possibilité d'exécuter a). Il est nécessaire de rajouter une condition à la racine pour restaurer cette congruence [16]. On remarque en fait qu'une transition *initiale* τ n'est pas réellement silencieuse.

Deux processus sont équivalents s'ils peuvent l'un et l'autre simuler leurs transitions initiales et s'ils sont tels que leurs processus résultants sont également bisimilaires. Dans l'exemple ci-dessus, $a + b$ n'est pas en mesure de simuler τ.

Pour exprimer cette nouvelle forme de bisimilarité, on introduit tout d'abord un *prédicat de terminaison* $\downarrow$ qui signifie simplement que le terme qu'il représente se termine. Une relation de bisimulation faible à la racine $\mathcal{B}$ est une relation binaire sur les processus, tel que :

1. si $p\mathcal{B}q$ et $p \xrightarrow{a} p'$, alors $q \xrightarrow{a} q'$ avec $p' \leftrightarrows_b q'$
2. si $p\mathcal{B}q$ et $q \xrightarrow{a} q'$, alors $p \xrightarrow{a} p'$ avec $p' \leftrightarrows_b q'$
3. si $p\mathcal{B}q$ et $p \downarrow$, alors $q \downarrow$
4. si $p\mathcal{B}q$ et $p \downarrow$, alors $q \downarrow$

Réexprimé intuitivement, si deux processus sont bisimilaires et que l'un est capable d'exécuter a comme action initiale, alors l'autre en est également capable et les termes résultants sont également bisimilaires.

16. Expliquée au point 2.4.2

De même, si deux processus sont bisimilaires et que l'un se termine, alors l'autre se termine également.

Deux processus sont faiblement bisimilaires à la racine $p \leftrightarrows_{rb} q$ s'il existe une bisimulation faible à la racine $\mathcal{B}$ tel que $p\mathcal{B}q$.

Abstraction : cas pratique

Le mécanisme d'abstraction et l'action silencieuse τ qui y est liée est en fait un problème bien connu des programmeurs. Prenons un exemple de code java :

```
public class UtilityClassImpl implements UtilityClass{
        public static void doSomething(){
                //realise des operations
                //jette eventuellement une exception non geree
                //si pas d'exception, se termine normalement
        }
}
```

Prenons une autre classe Java qui utilise la méthode statique ci-dessus :

```
import UtilityClass;
public class DoSomethingClass{
        public void a(){
                System.out.println("J'execute l'action \"a\"");
        }
        public void b(){
                System.out.println("J'execute l'action \"b\"");
        }
        public void c(){
                System.out.println("J'execute l'action \"c\"");
        }
        public static void main(String args[]){
                a(); b(); UtilityClass.doSomething(); c();
        }
}
```

Reprenons la classe ci-dessus et modifions la méthode `main` :

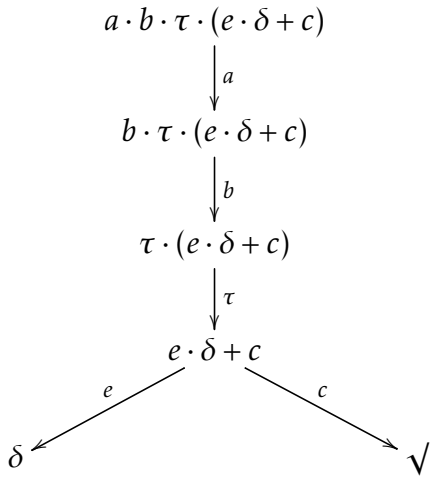
```
        public static void main(String args[]){
                a(); b(); c();
        }
```

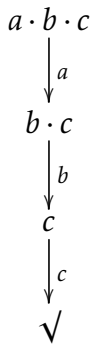
L'utilisateur de la méthode `doSomething` illustre bien le mécanisme d'abstraction que propose Java. Le programmeur n'a pas besoin de savoir ce qu'il se passe à l'intérieur de celle-ci. Il sait que c'est une méthode qui "fait quelque chose" et qui se termine. Le résultat visible pour l'utilisateur sera l'impression des trois lignes sur la console.

Si nous modélisons sous forme de processus les deux méthodes main et si nous les appelons respectivement $M1$ et $M2$, le code suivant formalise cet exemple dans le langage ACP_4.

$M1$ $a \cdot b \cdot \tau(e \cdot \delta + c)$
$M2$ $a \cdot b \cdot c$

Naïvement, on pourrait penser que $M1 \leftrightarrows_b M2$. Cependant, τ, qui représente le processus abstrait dans `UtilityClass`, n'est pas réellement transparent car la méthode `doSomething` pourrait ne pas se terminer correctement et générer une exception, que nous nommons ici e, menant à un deadlock et la non-exécution de la méthode c. Les schémas suivants nous le montrent.





Pour cette raison, le mécanisme de gestion d'exception de Java permet de gérer explicitement ces cas et donc de s'assurer que l'on ait la trace souhaitée.

```
a(); b();
try{
        UtilityClass.doSomething();
}catch(Exception e){
        //gestion de l'exception
}finally{
        c();
}
```

On devine qu'une telle implémentation aurait restauré la bisimilarité.

On constate ici que le raisonnement sur les algèbres de processus amène à des utilisations concrètes (et très souvent inconscientes) au jour le jour par des millions de développeurs.

Règles de transition

La spécification récursive gardée (2.4.5) demande une attention particulière dès lors qu'on introduit la notion d'action silencieuse. En effet, si une spécification récursive gardée, par exemple $X = a \cdot X$, admet une et une seule *solution*, il n'en est pas de même pour la spécification suivante $X = \tau \cdot X$ ou $X = \tau \cdot \tau \cdot X$, qui accepte n'importe quelle solution.

Il est donc nécessaire de redéfinir cette spécification gardée en tenant compte de τ.

Spécification récursive linéaire gardée

Une spécification récursive est linéaire si ses équations récursives sont de la forme

$$X = a_1 \cdot X_1 + \cdots + a_k \cdot X_k + b_1 + \cdots + b_\ell$$

avec $a_1, ..., a_k, b_1, ..., b_\ell \in A \cup \{\tau\}$

Une spécification linéaire récursive E est gardée s'il n'existe pas une séquence infinie de transitions $\tau\ \langle X|E\rangle \xrightarrow{\tau} \langle X'|E\rangle \xrightarrow{\tau} \langle X''|E\rangle \xrightarrow{\tau} \cdots$

Règles

Notre langage ACP, avec l'action silencieuse et la récursion linéaire gardée, conserve les règles de transition énoncées dans les pages précédentes car :

1. Les règles de transition qui n'incluent pas τ sont dépendantes de la *source* [17].
2. La source pour la règle de transition pour l'*action silencieuse* est la constante τ. ($\overline{\tau \xrightarrow{\tau} \surd}$)
3. La source d'une règle de transition pour une spécification récursive linéaire gardée E qui comprend τ est la constante $\langle X|E\rangle$
4. Chaque règle de transition pour la composition alternative, la composition séquentielle ou la récursion linéaire gardée qui comprend des transitions τ comme

$$\frac{x \xrightarrow{\tau} x'}{x + y \xrightarrow{\tau} x'}$$

inclut une prémice [18] qui contient
   - une relation du type $\xrightarrow{\tau}$ ou
   - un prédicat du type $\xrightarrow{\tau} \surd$
   - toutes les variables de la partie gauche de la prémice (ici x), reprises dans la source de la règle (ici $x + y$).

17. Le terme à gauche dans π (voir *spécification d'un système de transitions* à la section 2.4.1)
18. La partie H dans $\frac{H}{\pi}$

Exemple

Améliorons notre distributeur : il est maintenant capable de rendre la monnaie insérée. La figure 2.10 représente ce distributeur.

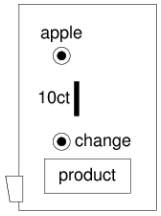


FIGURE 2.10 – Machine simple pour vendre un produit et rendre la monnaie.

En µCRL2, nous pouvons donc enrichir le programme comme ceci :

```
act
  ins10, acc10, coin10,ret10,
  optA, chg10, putA,
  readyA,out10 ;

proc
  User =
    ins10.(optA + chg10 ).User;

  Mach =
    acc10.( putA + ret10 ).Mach;

init
  hide(
    {readyA, out10},
    allow(
      { coin10,  readyA,  out10 },
      comm(
        { ins10|acc10 -> coin10, chg10|ret10 -> out10,
          optA|putA -> readyA },
        User || Mach
    ) ) ) ;
```

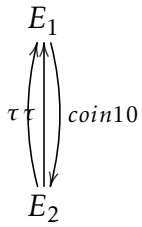
Au regard de ce code, nous pouvons déduire que nous imposons les communications suivantes :

*coin*10 (ins10|acc10) : insertion d'une pièce de 10 centimes dans la machine
readyA (optA|putA) : choix et distribution d'une pomme
*out*10 (chg10|ret10) : retour de 10 centimes

Une nouveauté est que nous "emballons" le processus `init` dans un bloc `hide` dont le premier argument reprend un ensemble d'actions à *abstraire*.

Nous faisons, dans cet exemple, *abstraction* des actions readyA et out10. Cela signifie que, pour un observateur extérieur, seule l'action coin10 est visible, les autres étant *silencieuses*. En d'autres termes, ces actions sont remplacées par τ dans leurs transitions.

Le schéma ci-dessous représente la vue du système à deux états correspondant.



De l'état E_1 du distributeur, nous pouvons passer dans l'état E_2 par l'observation de l'évènement $coin10$. Depuis cet état E_2, nous observons deux actions silencieuses τ pouvant nous mener à l'état E_1.

Axiomes algébriques

La table 2.6 nous donne les axiomes pour l'action silencieuse en tenant compte de la bisimulation faible à la racine.

TABLE 2.6 – Axiomes pour ACP_4

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

Les axiomes B1, 2 disent qu'une transition τ qui n'est pas initiale et qui ne perd aucun comportement est "réellement silencieuse".

2.5 Conclusion

Dans ce chapitre, nous avons posé le cadre théorique sur lequel nous nous appuyons lors du développement de notre interpréteur. De manière itérative, nous avons introduit des éléments-clés utilisés dans les langages d'algèbre de processus et, plus particulièrement, dans µCRL2.

Fort de cette base théorique, nous pouvons aborder la réalisation d'un interpréteur qui a pour vocation d'être plus ergonomique que les outils existants. Le but est d'obtenir une représentation du langage qui soit plus proche du code µCRL2, en évitant une étape de linéarisation, ainsi que de posséder un outil plus convivial à l'utilisation et évolutif pour permettre des extensions dans le futur.

Dans les chapitres suivants, nous présenterons les technologies utilisées, leurs atouts et nous proposerons une implémentation de *Procalg*, un outil de validation, de représentation et d'interprétation pour le langage d'algèbre de processus µCRL2.

Chapitre 3

Réflexions pour le développement de l'outil

Les algèbres de processus ACP_0 à ACP_4 et le langage de spécification µCRL2 ont donné naissance à une série de logiciels, regroupés dans une boîte à outils. Malheureusement, ces outils reposent sur une traduction préalable du code µCRL2 en une forme linéaire. Ainsi qu'illustré à la page 17, celle-ci est relativement éloignée du code original et rend ainsi la simulation du code très délicate. En outre, l'interface prend la forme archaïque d'une commande en ligne et, par suite, est relativement peu conviviale. Ceci ouvre la porte au développement d'une application de simulation, qui est le sujet de ce mémoire. Nous nous tournons dans ce chapitre vers la conception de cette application après avoir mis en lumière l'outil existant de simulation µCRL2.

3.1 Outil de simulation de la boîte à outils de µCRL2

Comme évoqué, µCRL2 propose un vaste panel d'outils. La figure 3.1 nous montre les différentes options qui s'offrent à l'utilisateur. Dans cet exemple, nous voyons une interface de commande en ligne ainsi qu'une interface graphique rudimentaire et peu intuitive. Etant donné leur multitude, il est difficile de présenter, dans ce travail, un tour complet des outils proposés. Il serait néanmoins intéressant de connaître ceux qui sont réellement exploités, leurs avantages et leurs inconvénients.

La commande en ligne qui, dans l'exemple de la figure 3.1, affiche l'aide, n'induit pas intuitivement la manière de l'utiliser.

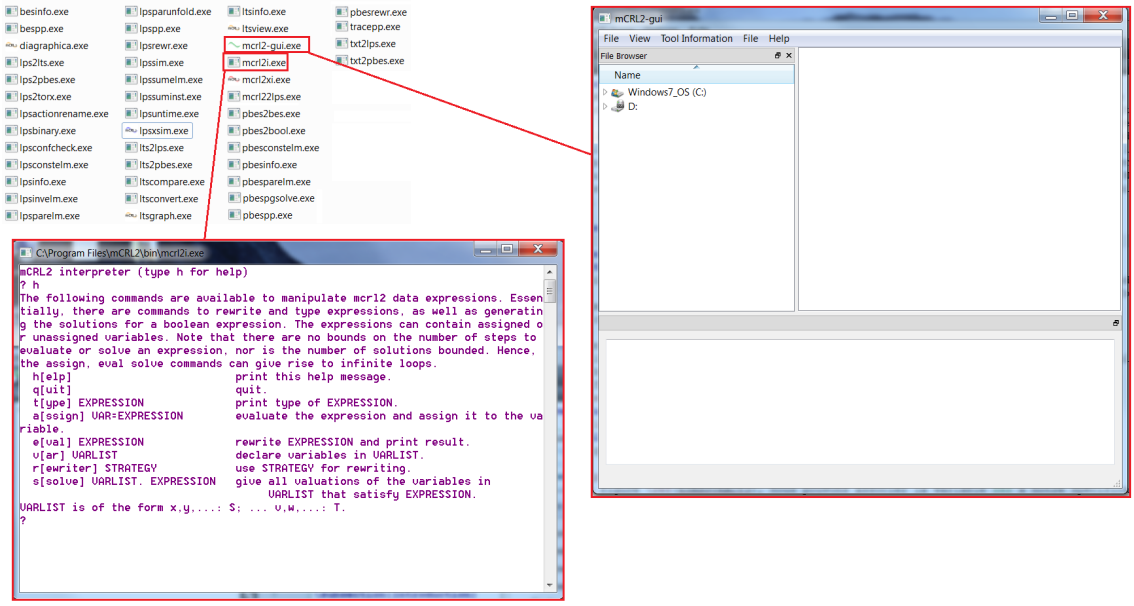


FIGURE 3.1 – Boîte à outils µCRL2

Nous suggérons de nous attarder avant tout sur l'interface graphique "mcrl2-gui". Cette dernière ajoute un peu de convivialité. Comme l'indique la figure 3.2, cette interface nécessite, au préalable, la création d'un fichier contenant la spécification que nous souhaitons analyser. Le clic droit de la souris nous dévoile une série d'options. La représentation graphique à laquelle nous désirons arriver passe par une première transformation qui aboutit à la création d'un fichier LPS [1]. Celui-ci traduit la spécification, contenant ici des processus parallèles, en spécification linéaire dont nous avons fourni l'exemple dans le chapitre précédent (page 17).

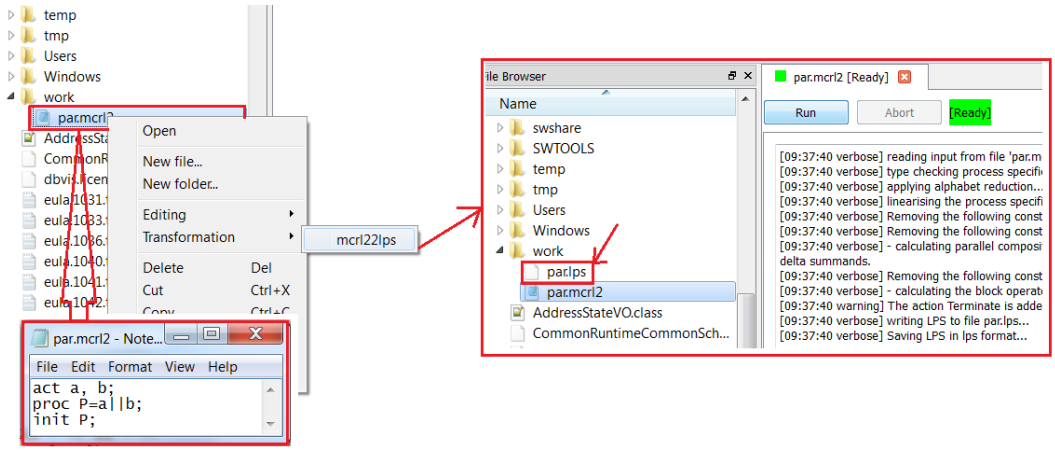


FIGURE 3.2 – Traduction en processus linéaire

1. Linear Process Specification

Ce processus linéaire nous permet alors de raisonner avec l'outil de simulation "lpssim" dont nous montrons l'interface dans la figure 3.3.

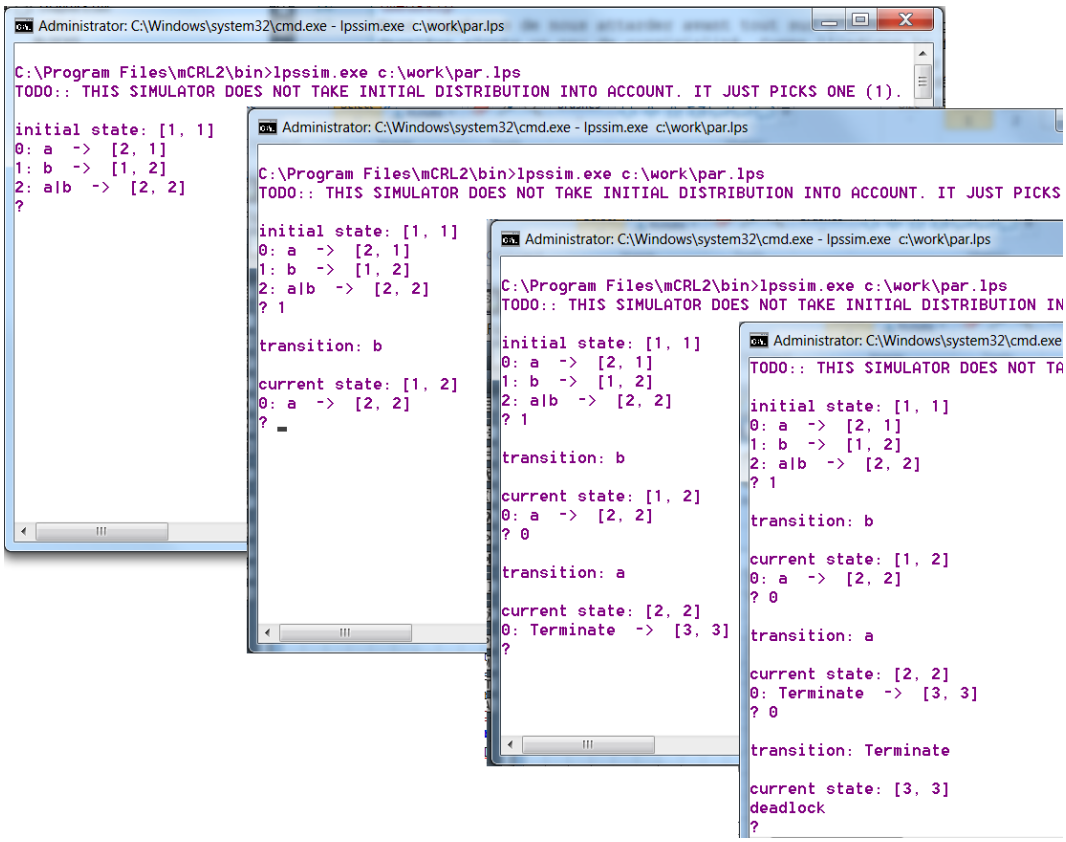


FIGURE 3.3 – Outil de simulation "lpssim"

Cette interface, des plus rudimentaires, nous propose une manière de raisonner très éloignée de la spécification parallèle entrée. En effet, seuls des éléments du fichier linéarisé (celui de la page 17) apparaissent. Dans des modélisations complexes, l'utilisateur risque fort d'y perdre son chemin. La motivation du présent travail vient notamment de ce constat.

La section suivante montre une analyse pour la réalisation d'un outil intuitif, la boîte à outils µCRL2 s'écartant un tant soit peu, il faut le reconnaître, de cet objectif.

3.2 Un outil de simulation convivial

3.2.1 Introduction

D'un point de vue "haut niveau", le lecteur peut s'attendre à une application qui prend, en entrée, une spécification µCRL2, qui la traite et propose ensuite une vue interactive. Comme nous l'exprimions au chapitre précédent,

"(...) Une grande partie du monde réel peut être modélisée par des transitions (...)"

La réalisation attendue n'échappe pas à ce concept, comme nous l'illustrons dans l'exemple suivant qui se base sur une figure utilisée précédemment.

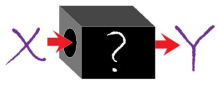


FIGURE 3.4 – Système avec entrée-sortie

Dans la figure 3.4, nous pouvons associer la variable *X* à notre spécification textuelle conforme à la syntaxe µCRL2. La boîte noire représente l'application proprement dite et la variable *Y* représente la sortie qui sera la vue interactive proposée à l'utilisateur.

La présentation de ce que nous attendons de notre outil nous mènera, dans les chapitres suivants, à décrire comment nous implémentons le contenu de cette boîte noire.

3.2.2 Actions de l'utilisateur

L'application que nous envisageons ici donne à l'utilisateur la possibilité d'effectuer les opérations décrites dans le diagramme des cas d'utilisation de la figure 3.5.

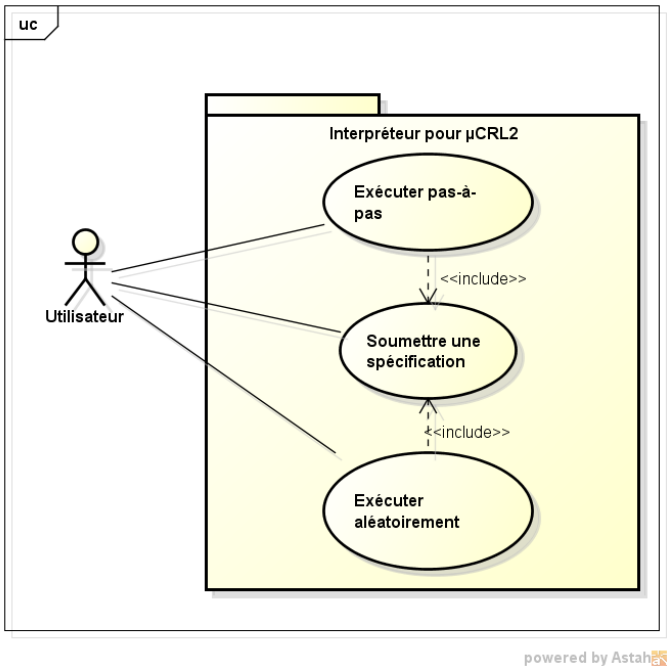


FIGURE 3.5 – Diagramme de cas d'utilisation pour l'interpréteur

- **Soumettre une spécification :** il est évident que soumettre une spécification est un prérequis à toute forme d'exécution. Cette dernière est entrée conformément à la syntaxe µCRL2.
- **Exécuter pas à pas :** l'exécution pas à pas consiste à donner à l'utilisateur le contrôle dans l'exécution de chaque action atomique atteignable depuis chaque état du système.
- **Exécuter aléatoirement :** l'exécution aléatoire donne la possibilité à l'utilisateur de demander à l'application d'effectuer au hasard une transition possible à chaque état du système jusqu'à l'état terminé, ou jusqu'à un certain niveau de la structure au cas où, par exemple, cette dernière représenterait une récursion infinie.

3.2.3 Interface utilisateur

Ces cas d'utilisation nous donnent déjà une représentation intuitive des possibilités que nous offre la vue "utilisateur". Nous présentons ici une vue intuitive de ce que nous allons "matérialiser" par après.

Dans cette section, nous allons concevoir une idée des écrans que nous souhaitons obtenir. Dans la suite de ce mémoire, nous expliquerons la manière de les implémenter et nous établirons une présentation du rendu final de l'application.

Écran de départ

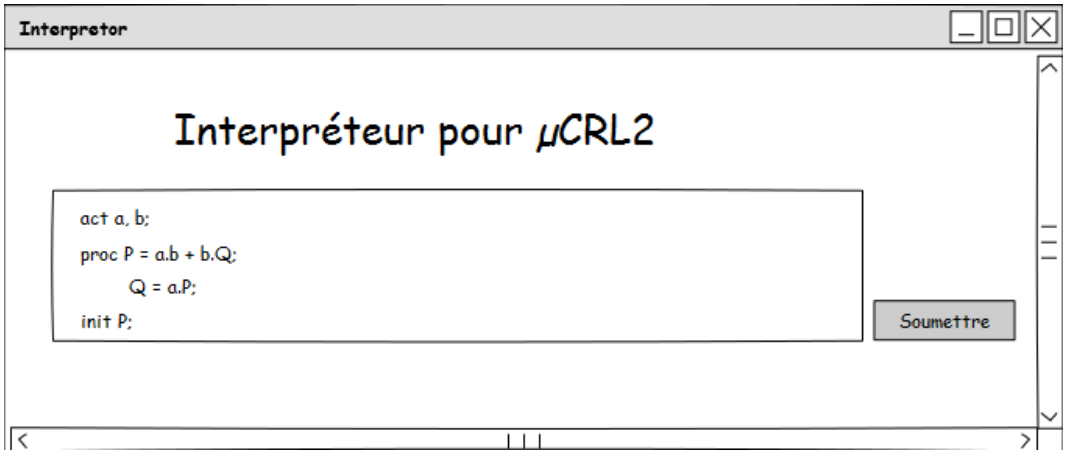


FIGURE 3.6 – Ecran de départ

L'écran de la figure 3.6 se veut intuitif. L'avantage réside dans le fait que, en l'absence de points de choix pour l'utilisateur, cette interface suggère elle-même la manière dont elle doit être utilisée. On y voit la possibilité d'insérer du texte et ensuite de le soumettre avec le bouton à droite. L'utilisateur déduira que l'on s'attend à avoir, dans cette zone de texte, une spécification µCRL2 valide.

Écran d'erreur

Dans le cas d'une spécification non valide [2], nous aboutissons sur un écran d'erreur qui tente de guider l'utilisateur dans la résolution de cette erreur, comme le montre la figure 3.7.

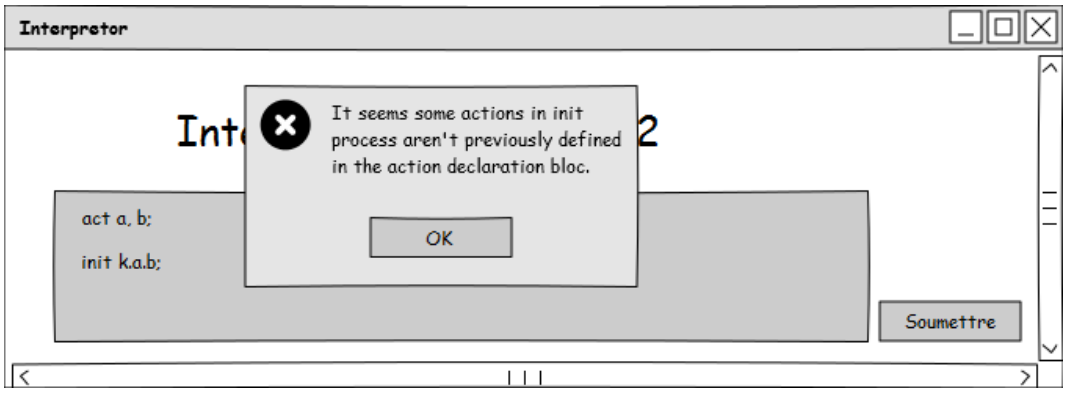


FIGURE 3.7 – Ecran d'erreur

2. Par exemple une mauvaise syntaxe ou une référence à des actions non déclarées

Il s'agit d'une fenêtre modale. De cette manière, l'utilisateur n'a d'autre choix que d'accuser réception du message d'erreur avant de continuer. Nous évitons ainsi que cette information ne soit ignorée et que l'utilisateur, en l'absence d'information, ne perde du temps à essayer de comprendre ce qui ne fonctionne pas.

Écran d'évolution du processus

La figure 3.8 nous montre le comportement attendu lorsqu'une spécification valide est entrée.

Nous pouvons observer une zone "Resultat" qui affiche la trace courante des actions exécutées par l'utilisateur. Sur la droite de l'écran, une information nous indique que le processus est en cours, c'est à dire qu'il est toujours possible d'exécuter des actions. Nous y voyons également la possibilité d'un choix aléatoire dans ce qui est cliquable.

Nous pouvons ensuite voir une série de boutons qui représentent les actions et, entre ces actions, leur opérateur de composition. Les actions "exécutables" dans l'état courant du système sont cliquables et également mises en évidence par leur couleur verte.

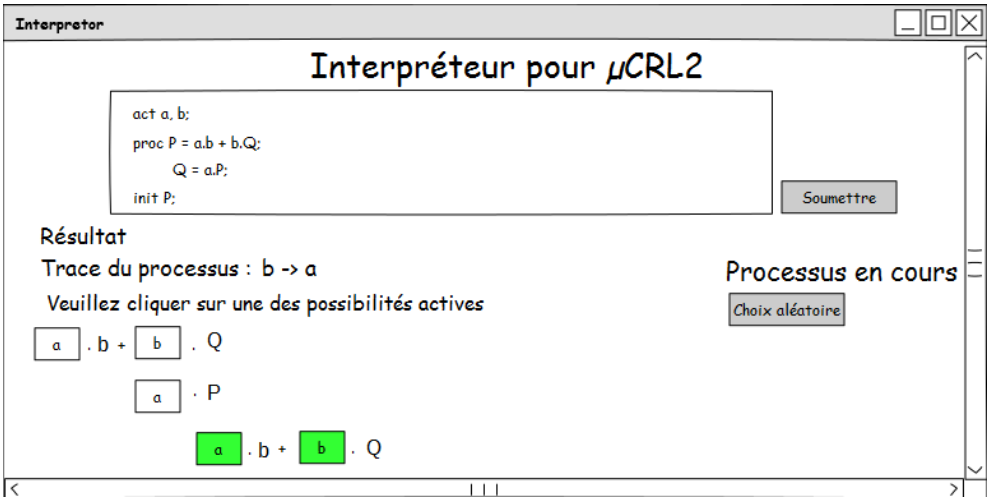


FIGURE 3.8 – Ecran d'analyse du processus µCRL2

Écran de terminaison d'un processus

La figure 3.9 nous donne une vue de l'écran lorsqu'un processus se termine. Nous pouvons observer ici la trace complète effectuée par l'utilisateur et l'information que le processus est terminé [3].

3. Potentiellement, avec la récursion, nous pouvons ne jamais arriver dans cet état

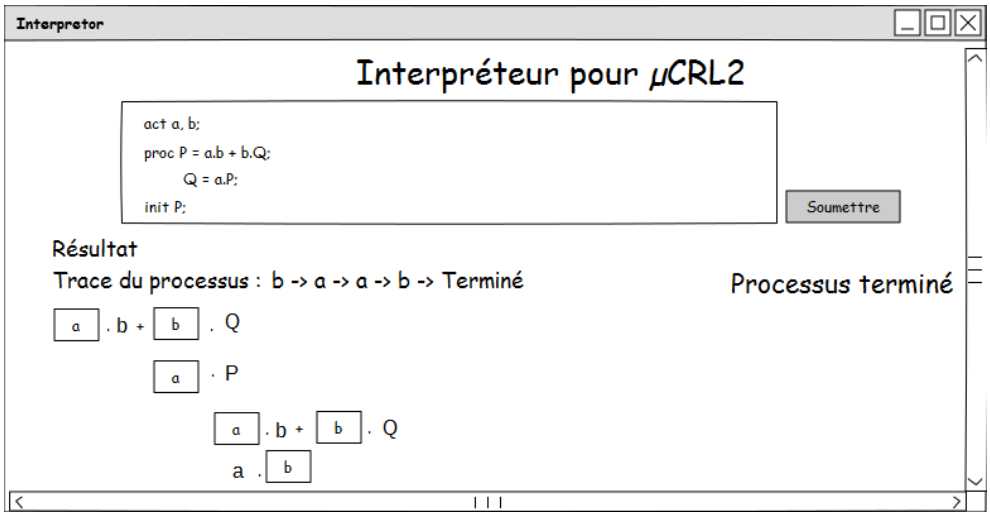


FIGURE 3.9 – Ecran de fin de processus

3.3 Retour des utilisateurs

L'analyse ainsi présentée se conforme principalement à notre perception des éléments que nous jugeons nécessaires. Dans le cadre d'une démarche visant à satisfaire les besoins de différents types d'utilisateurs, il conviendrait de définir des profils et leurs attentes.

Dans l'état actuel, il serait intéressant de faire tester l'application, et de collecter ensuite, à l'aide de questionnaires, l'avis des utilisateurs. Nous pensons notamment à des profils de chercheurs, d'ingénieurs, de techniciens et d'informaticiens.

Voici un ensemble de questions qui pourraient nous retourner des éléments pertinents :

- quel type de profession exercez-vous ?
- comment qualifieriez-vous votre maîtrise de l'outil informatique ?
- comment qualifieriez-vous vos connaissances en algèbre ?
- préférez-vous une interface simple et intuitive ou complète et documentée ?
- la plupart du temps, êtes-vous en déplacement ou au bureau ?
- quels types d'appareils informatiques utilisez-vous à des fins professionnelles ?
- y a-t-il des éléments superflus dans l'application ?
- constatez-vous que des éléments manquent dans l'application ?

En ce qui concerne les quelques réactions que nous avons récoltées par rapport à notre première analyse, nous pourrions rendre l'application plus agréable par l'utilisation de couleurs et d'arbres binaires au lieu d'une série de boutons. Pour certains, il s'avérerait aussi intéressant que l'application propose de charger un exemple de spécification μCRL2. Un besoin potentiel consiste également à pouvoir revenir sur un choix afin d'explorer d'autres possibilités.

Cette section met en évidence qu'il existe de nombreuses pistes d'amélioration, tant au niveau fonctionnel qu'ergonomique.

3.4 Conclusion

Ce chapitre a présenté les principales exigences dans le cadre du développement de l'outil que nous souhaitons produire.

Le type d'application que nous envisageons ici, et le lecteur en aura sans doute l'intuition, est d'apparence basique. Notre but premier, dans cette partie du travail, consistait à nous familiariser avec de nouvelles technologies et à montrer une interface avant tout utilisable. C'est donc dans l'esprit d'un POC [4] que nous réaliserons l'application demandée.

Il est bien entendu que d'éventuelles améliorations dans le confort d'utilisation et l'ajout de fonctionnalités pourraient voir le jour. Néanmoins, nous ne sommes pas resté sur des bases théoriques. Notre analyse a fait l'objet d'une implémentation dont le lecteur pourra explorer les sources sur le dépôt git à l'adresse : `https://forge.info.unamur.be/scm/git/ocroegae_memoire_jacquejm`.

Les chapitres qui suivent abordent la réalisation concrète et la présentation de l'outil dans sa version du 30 mai 2016.

4. Proof Of Concept

Chapitre 4

Choix des techniques et implémentation

La réflexion pour le développement s'est déroulée principalement sur base de plusieurs axes. Le lecteur aura déduit que le processus menant d'une entrée en texte brut vers un affichage ergonomique et interactif passe tout d'abord par une interprétation et une validation du langage µCRL2.

Comme tout langage de programmation, et en opposition au langage naturel, µCRL2 est un langage non-contextuel. Cela signifie que la définition d'une spécification peut être interprétée de manière non-ambigüe [1]. De cette façon, il est aisé de faire interpréter ce langage par une machine. Dans ce cadre, nous nous aventurons en terrain connu, l'analyse syntaxique faisant l'objet d'un enseignement appliqué à la réalisation d'un compilateur à la Faculté d'informatique [2].

Une bonne partie des langages de programmation permet l'interprétation d'un code textuel sur base d'une analyse d'expressions régulières et de définitions de règles. Il en est néanmoins qui s'avèrent plus en adéquation avec notre objet. Notre choix s'est porté sur *Scala*. Dans ce chapitre, nous expliquerons en quoi la couche fonctionnelle qu'apporte Scala au-dessus de Java se révèle un outil puissant pour un parsing efficace d'une chaîne de caractères, procurant un gain considérable au niveau du temps et diminuant de manière drastique le nombre de lignes de code nécessaire.

Notre code Scala nous permet de valider rapidement une spécification entrée et également de générer un arbre représentant la structure de données qui sera utilisée pour livrer une représentation sur l'interface "utilisateur".

Au niveau du développement d'une interface graphique, la réflexion s'est portée sur une application de type "monolithique" où nous exploitons la sous-couche Java

1. Toute interprétation du langage mène à une et une seule représentation possible
2. Le cours "Syntaxe et Sémantique"

et les librairies graphiques qu'elle offre. Ce type de solution est certes éprouvé depuis des décennies mais présente l'inconvénient de la nécessité d'un client "lourd" sur sa machine avec les risques de compatibilité que cela implique. Par ailleurs, l'évolution du web nous montre une tendance au découplage des applications, qui sépare le traitement des données et la vue que l'on peut proposer dans un navigateur. Contrairement à la première philosophie de programmation, ces méthodes ne nécessitent pas l'installation d'un client lourd et sont directement utilisables sur la majorité des plateformes existantes.

Ce découplage mène inévitablement à une réflexion sur la technologie à utiliser, d'une part pour l'affichage et d'autre part pour les possibilités de communication entre notre back-end en Scala et notre front-end. Parmi la jungle des frameworks existants, notre choix s'est porté sur une application Spring Boot Scala avec notre front-end réalisé en AngularJs. Cette technologie est particulièrement intéressante dans le contexte d'une application interactive. En effet, AngularJs offre une réactivité au niveau du navigateur qui nous permet, de manière fluide, d'enrichir la page web à chaque action de l'utilisateur.

Nous développerons, dans les sections qui suivent, les raisons de nos différents choix ainsi que leurs avantages/inconvénients au regard d'autres technologies qui nous donnent également des pistes intéressantes. Nous illustrerons également les éléments-clés de notre implémentation de l'interpréteur.

4.1 Le framework utilisé

Comme évoqué ci-avant, nous nous sommes dirigé vers une application Spring Boot que nous introduisons dans cette section.

4.1.1 Spring

Spring est un framework Java open source qui a été créé pour faire face à la complexité liée au développement d'applications d'entreprise.

Il est possible avec Spring de construire des applications en permettant de lier des POJO [3] aux différents services existants (bases de données, webservices, etc...). Ce mapping est en outre réalisé de manière non invasive (au moyen d'annotations sur les classes, leurs attributs et méthodes). Par exemple, dans le cas de notre application, nous pouvons aisément déduire que nous aurons besoin que notre back-end offre un service pour lui envoyer le code µCRL2. Nous pouvons naïvement déclarer une classe Scala comme suit :

3. Plain Old Java Object : un objet tel que nous le codons en Java.

```
class IOController{

  def send(input: Input, bindingResult: BindingResult, servletRequest :
      HttpServletRequest) = {
        /**
          * Use the input and provide it as a parameter for the service's
              method
          * which gives us the structured response we need.
          */
    Mcrl2Service.getRoot(input)

  }
}
```

Listing 4.1 – Code Scala sans annotations

Le code du listing 4.1 nous donne une intuition du but recherché. Le paramètre "input" va contenir le code µCRL2 et la méthode "send" renvoie le résultat du service. Il s'agit d'un point d'entrée pour notre back-end Scala. Mais comment, dès lors, communiquer avec des applications extérieures ? La puissance de Spring nous permet, avec quelques annotations dans le code, de décrire la manière d'interagir avec le contrôleur ci-dessus :

```
@RestController
@RequestMapping(Array("/analyze"))
class IOController{

  @RequestMapping(value = Array("/send"), method = Array(RequestMethod.POST) ,
      consumes = Array(MediaType.APPLICATION_JSON_VALUE))
  @ResponseStatus(HttpStatus.OK)
   def send(@RequestBody input: Input, bindingResult: BindingResult,
      servletRequest : HttpServletRequest) = {

    Mcrl2Service.getRoot(input)

  }
}
```

Listing 4.2 – Code Scala avec annotations

Dans le listing 4.2, nous voyons apparaître des éléments précédés du caractère "@". Ces annotations réfèrent toujours à l'élément du langage avant lequel elles sont insérées. Dans l'exemple ci-dessus, la classe `IOController` est enrichie par deux annotations :

- la première, `@RestController`, désigne le type `IOController` comme étant un contrôleur REST [4] dont toutes les méthodes annotées avec `@RequestMapping` sont automatiquement considérées comme renvoyant une réponse.

4. Representational State Transfer : L'architecture REST permet une communication entre producteur et consommateur [20]

- la seconde, `@RequestMapping` sur la classe, donne le chemin de base pour appeler le service depuis le client (dans notre cas, le navigateur). L'annotation `@RequestMapping` sur la méthode `send` permet de définir le chemin pour pouvoir appeler le service depuis le chemin de base. Cela signifie que, depuis le navigateur, un appel à cette méthode devra contenir l'information du chemin `/analyze/send`, comme nous le montre le fragment de code Javascript du listing 4.3.

```
$scope.submit = function(){
        $http({url:'analyze/send',method:'POST',data:{"code":$scope.value}}).then
            (function(data){
            $rootScope.result = data.data.initialSpec;
        })
    };
```

Listing 4.3 – Code JavaScript appelant un service REST

Nous pouvons, sans trop de difficultés, établir le lien entre les fragments des listings 4.2 et 4.3. Dans l'annotation `@RequestMapping`, nous définissons le type de méthode utilisé pour appeler le service (http POST dans notre cas). Dans le code Javascript du listing 4.3, nous définissons également cette méthode POST. Dans ce code, la valeur `data` est une structure JSON [5] contenant des champs structurés comme une série de valeurs associées à des clés. Ces clés correspondent aux noms des champs de l'objet `Input` que l'on retrouve dans le code de notre contrôleur (listing 4.2). Le mapping est assuré en définissant le type de média utilisé comme étant le type JSON (`MediaType.APPLICATION_JSON_VALUE`). L'annotation `@RequestBody` présente sur l'attribut `Input` permet de définir le mapping avec la donnée `data` transmise depuis le code JavaScript.

Enfin, l'annotation `@RestController` dont nous avons parlé ci-dessus nous fournit une traduction de l'objet Scala retourné par la méthode `send` en un objet JSON qui sera envoyé en réponse vers notre front-end JavaScript.

Nous constatons donc que ces annotations nous permettent de définir facilement et lisiblement la manière dont notre service communique, dans le but de simplifier le développement. Spring consiste en un ensemble de fonctionnalités qui sont organisées en une vingtaine de modules ayant chacun leur domaine spécifique (persistance, web, sécurité, test, ...). De cette manière, on évite d'écrire du code réutilisable, ce qui permet de se focaliser sur les fonctionnalités business.

4.1.2 Spring Boot

Springboot est un projet qui a été introduit depuis la version 4 de Spring dans le but de réduire le temps de configuration d'un nouveau projet.

5. JavaScript Object Notation

Une fonctionnalité très intéressante s'observe dans le fait qu'il embarque des serveurs pouvant être activés facilement. Dans notre application, l'exécution de la méthode *main* démarre un serveur Apache Tomcat [9] et rend l'application fonctionnelle et disponible depuis un navigateur en quelques secondes.

Ce projet s'inspire d'un template d'application Spring Boot Scala disponible publiquement dans un dépôt git à l'adresse suivante :
`https://github.com/bijukunjummen/spring-boot-scala-web` [13].

4.2 Le back-end

Dans cette section, nous exposons la partie back-end de notre application qui est développée en Scala. Cette dernière est composée d'une définition du langage µCRL2 au moyen de la librairie *scala-parsers-combinators* ainsi que de la structure permettant de représenter, sous forme d'arbre, la spécification entrée. Outre la syntaxe, qui est entièrement validée par les parseurs que nous définirons dans cette section, une étape de validation sémantique est assurée par différentes méthodes rassemblées dans une classe utilitaire. Le back-end expose le service dont nous avons besoin au moyen du contrôleur introduit dans la section précédente.

4.2.1 Le langage

Nous proposons avant toute chose une vue des principales caractéristiques et des éléments de syntaxe du langage Scala.

"(...) Le langage Scala propose trois paradigmes de programmation :

- le paradigme de programmation orientée objet, qui reprend la plupart des concepts de fonctionnement du langage Java.
- le paradigme de programmation impérative.
- le paradigme de programmation fonctionnelle.

Bien que ce dernier paradigme s'oppose en tout point à la programmation impérative, Scala propose de travailler soit avec l'un, soit avec l'autre, soit avec les deux. (...)"[4]

"(...) Scala est un langage à typage statique, c'est-à-dire que toutes les variables manipulées ont un type. Ce choix a été fait pour que les erreurs de programmation soient détectées à la compilation. Le langage Scala offre cependant les avantages du typage dynamique : lorsque le compilateur peut deviner le type d'une variable, il n'est pas nécessaire de la spécifier. (...)"[4]

"(...) En Scala, tout est vraiment objet contrairement au langage Java. En effet, les types primitifs du langage comme les entiers ou les booléens sont des classes (Int et Boolean, respectivement). Le langage Scala étant un langage objet, une variable

d'un type peut accepter soit une instance de ce type, soit une instance d'une classe qui hérite de ce type.

Enfin, la syntaxe du langage Scala se veut la plus légère possible. Par exemple, le point-virgule à la fin d'une instruction n'est pas obligatoire. (...)"[4]

Prenons pour exemple le code du listing 4.4, qui représente un *objet* contenant la méthode principale pour le parsing de notre spécification µCRL2.

```
object Mcrl2Service {
  var initialSpec = new Output
  def getRoot(input: Input): Output ={

    println("Parsing \""+input.getCode+"\"")


    val parser = new Mcrl2Parser
    val parsingResult = parser.parseAll(parser.mcrl2Specification, input.getCode)
        .get

    initialSpec.setInitialSpec(parsingResult.initialSpec)
    initialSpec.setActions(parsingResult.actions)
    initialSpec.setInitialCode(input.getCode)
    initialSpec.setProcesses(parsingResult.processes)

    initialSpec
  }
}
```

Listing 4.4 – Service pour µCRL12

Le mot-clé "`object`" déclare un *singleton*, c'est-à-dire l'équivalent d'une classe statique en Java. Nous nous en servons ici car nous souhaitons nous assurer qu'il n'existe qu'une et une seule instance durant notre utilisation de l'interpréteur. La raison est que cet objet va contenir, dans la variable `initialSpec`, l'ensemble de la spécification, auquel nous souhaitons pouvoir accéder tout au long d'une session "utilisateur".

Nous apercevons également le mot-clé "`def`" qui nous permet de définir une méthode, dans ce cas-ci, la méthode "`getRoot`". En entrée, elle prend un objet de type "`Input`"[6] et renvoie un objet `Output` qui contient toutes les structures nécessaires pour être manipulé dans l'interface.

Dans cette méthode, nous pouvons apercevoir la déclaration de constantes (l'équivalent d'une statique en Java) au moyen du mot-clé "`val`". Nous déduisons par ce code que le compilateur va automatiquement associer aux valeurs déclarées le type de l'objet qui leur est assigné.

L'objet "`parser`" déclaré possède une méthode "`parseAll`". Cette dernière prend en argument le code µCRL2 reçu ainsi que le parser désiré. Elle renvoie, dans l'objet

6. L'objet reçu par notre contrôleur et contenant la spécification textuelle µCRL2

parsingResult, une structure contenant l'arbre désiré pour pouvoir commencer à raisonner sur l'algèbre ainsi fournie.

Nous constatons également l'absence d'un mot-clé "`return`" car la dernière ligne révèle un objet du même type que celui retourné par la méthode. Le compilateur déduira donc qu'il s'agit de la valeur à retourner.

Nous introduirons, dans la suite, d'autres particularités du langage.

4.2.2 Le parsing avec Scala

Dans le listing 4.4, nous pouvons observer l'instanciation d'un objet du type "`Mcrl2Parser`". Pour comprendre ce qui s'opère dans sa méthode "`parseAll`", nous allons explorer la définition de sa classe au sein de l'objet Scala "`dsl`"[7].

```
import scala.util.parsing.combinator.JavaTokenParsers

/**
  * Created by Olivier Croegaert on 8/11/2015.
  * This scala class describes the syntax for mcrl2
 */
object dsl {
  /**
    * Storing a technical name chosen for the process "init" which embed all the
      mcrl2
    * application logic
    */
  val INIT_PROC_ID = "PROC20160502"

  class Mcrl2Parser extends JavaTokenParsers{

    /**
      * Defining a parser to describe what a mcrl2 specification is
      *
      * A mcrl2 specification is either one clause "init" or either a definition
        of actions eventually followed
      * by some process definitions and a (mandatory) clause init.
      *
      */
    def mcrl2Specification = opt(actions~opt(processes))~init ^^{
      case None ~ i => MCRL2Factory.buildTree(i)
      case Some(a ~ None)~ i => MCRL2Factory.buildTree(a, i)
      case Some(a~Some(p))~ i => MCRL2Factory.buildTree(a, p, i)
    }
    (...)
```

Listing 4.5 – Parser d'une spécification µCRL2

Nous pouvons tout d'abord observer dans ce listing que la classe `Mcrl2Parser` étend la classe JavaTokenParsers, disponible dans la librairie "scala.util.parsing.combinator". `JavaTokeParsers` étend à son tour la classe `RegexParser` qui contient la méthode `parseAll` que nous employons dans le listing 4.4.

7. Le singleton "dsl", ou Domain Specific Language

Le but est de venir ajouter aux parsers déjà existants dans ces classes, des parsers spécifiques à notre langage µCRL2. Dans l'exemple 4.5, nous définissons le parser `mcrl2Specification`. Focalisons-nous sur cet élément :

```
def mcrl2Specification = opt(actions~opt(processes))~init
```

Nous définissons ici une spécification µCRL2 comme étant un ensemble de parsers. Le sous-ensemble de µCRL2 que nous envisageons peut comporter une seule instruction `init`, ou alors une définition d'actions et éventuellement de processus. Le mot-clé `opt` marque le caractère optionnel du bloc entre parenthèses tandis que le mot-clé "~" marque la lecture en séquence des différents éléments considérés. "`actions`, "`processes`" et "`init`" sont simplement d'autres parsers décrivant respectivement une déclaration d'actions, de processus et d'initialisation.

Nous venons ici, en toute simplicité, de déclarer ce qu'est une spécification µCRL2. Il s'agit néanmoins d'une analyse syntaxique et nous souhaitons obtenir un résultat sous forme d'objet. Observons la suite du code :

```
def mcrl2Specification = opt(actions~opt(processes))~init ^^{
  case None ~ i => MCRL2Factory.buildTree(i)
  case Some(a ~ None)~ i => MCRL2Factory.buildTree(a, i)
  case Some(a~Some(p))~ i => MCRL2Factory.buildTree(a, p, i)
}
```

Le symbole "^^" pourrait être interprété par "que fait-on ensuite ?". En réalité, il s'agit d'une composition de fonctions où ce qui est retourné par la fonction de gauche devient un argument de la fonction de droite. Nous l'avons évoqué ci-dessus et formalisé dans notre parser : il y a trois cas possibles. Nous devons donc traiter ces trois cas en les matérialisant par trois séquences possibles.

La première séquence identifiée par le "pattern" suivant :

```
case None ~ i
```

Ce dernier sera retenu (matching) si le groupe optionnel englobant est vide et qu'il détecte le parser `init`. Nous en déduisons donc ici que la variable "`i`" fait référence à `init`.

La deuxième séquence possible est identifiée par le "pattern" :

```
case Some(a ~ None) ~ i
```

Nous en tirons intuitivement qu'il s'agit d'une déclaration d'actions, qui correspond à la variable "a", suivie d'une initialisation correspondant à "`i`".

La troisième séquence est le "pattern" d'une spécification comprenant actions, processus et initialisation.

Le symbole “=>” vient enfin indiquer, dans chaque cas, ce que doit retourner le parser sur base des éléments détectés. Il fait appel ici aux méthodes d’un objet Scala, MCRL2Factory, qui permettent de construire l’arbre que nous recherchons. La méthode “buildTree” retourne un objet de type “Specification”.

La manière dont est déclaré le parser “mcrl2Specification” ne nécessite pas de déclarer explicitement son type car il prendra implicitement le type Specification retourné par la méthode buildTree.

Focalisons-nous maintenant sur le parser “actions” et le parser “action” qui, respectivement, décrivent ce qu’est une déclaration d’actions (une liste d’actions) et ce qu’est une action au sens syntaxique du terme.

```
  def actions: Parser[List[Action]] = "act"~> action ~ rep(","~> action) <~";
      "^^{
  case n1~n2 => n1::n2
}
    def action: Parser[Action] = """[a-z][a-zA-Z0-9]*""".r^^{
  case a => {
    new Action(a)
  }
}
```

Nous remarquons ici que la définition du type de retour du parser est explicite, contrairement au cas de “mcrl2Specification”. Il est en effet parfois plus facile d’effectuer ce choix lorsqu’il y a beaucoup de déclarations de types différents, ne fût-ce que pour faciliter la tâche du développeur en cas de bug.

Le parser “action” est assez simple à comprendre. On y définit, sous forme d’une expression régulière, qu’une action µCRL2 est une chaîne de caractères commençant par une minuscule, suivie d’un nombre aléatoire de lettres minuscules, majuscules et de chiffres. L’expression régulière est déclarée dans un bloc """<REGEX>""".r. Lorsque cette dernière est détectée, le parser renvoie un objet de type “Action” contenant la chaîne de caractères trouvée.

Le parser “actions” fait apparaître d’autres éléments intéressants. Si, dans le langage µCRL2, le mot-clé “act” est important pour annoncer une déclaration d’actions, il n’est plus d’aucune utilité une fois que nous détectons ces actions pour les stocker dans une structure adéquate. Là où la séquence "act"~action conserve le mot-clé, l’ajout du caractère “>” vient donner l’information de ne pas tenir compte du mot détecté à sa gauche lors du traitement du résultat du parsing. On voit, dans ce parser, que nous utilisons ce caractère pour nous libérer de toutes les informations "parasites" comme les virgules et points-virgules.

Le mot-clé “rep” déclare une répétition de 0 à N fois son contenu, ce qui est très pratique dans le cas d’une déclaration de multiples actions. Le résultat du parser nous donne donc deux variables, n1 et n2, n1 étant le résultat du parser “action” et n2 étant une liste de résultats du même parser. Il reste donc à retourner une

liste unique comprenant toutes ces actions. Scala nous offre là une syntaxe légère : `n1::n2` nous retournant la liste désirée. Cette liste sera au final utilisée comme paramètre de la méthode `buildTree` utilisée dans notre parser mCRL2Specification.

Cette section nous a donné une idée de la puissance de la programmation déclarative dans le cadre du parsing. En maîtrisant les parsers de Scala, il nous reste à bien formaliser le langage que nous analysons, sans penser à la manière dont nous allons le déclarer. Le lecteur admettra peut-être, selon sa sensibilité, que les différentes déclarations expliquées ci-dessus peuvent se déduire assez rapidement de la lecture du code Scala, étant donné que ces dernières sont proches d'une explication en langage naturel de notre spécification.

4.2.3 La construction de l'arbre

Pour obtenir une représentation non-ambigüe de notre spécification, nous souhaitons construire un arbre dont la racine sera le processus "`init`". C'est ce que fournit notre parser dans l'objet de type "`Specification`" retourné par la méthode buildTree. Nous allons, dans cette section, aborder les structures développées pour mener à bien cette construction.

Le code ci-dessous nous dévoile les principaux éléments créés lors de l'interprétation de la spécification µCRL2.

```
import scala.beans._
@BeanProperty
abstract class Mcrl2Object
trait Leaf
@BeanProperty
case class Composition(@BeanProperty op:String, @BeanProperty left:Mcrl2Object, @
    BeanProperty right:Mcrl2Object) extends Mcrl2Object {
  @BeanProperty
  val term = "composition"}
@BeanProperty
case class Action(@BeanProperty name:String) extends Mcrl2Object with Leaf{
  @BeanProperty
  val term = "action"}
@BeanProperty
case class Delta() extends Mcrl2Object with Leaf{
  val term = "delta"}
@BeanProperty
case class ProcessId(@BeanProperty name:String) extends Mcrl2Object with Leaf{
  @BeanProperty
  val readable = toString}
@BeanProperty
case class ProcessSpec(@BeanProperty id:ProcessId, @BeanProperty process:
    Mcrl2Object) extends Mcrl2Object {
  @BeanProperty
  val term = "processSpec"}
```

Listing 4.6 – Eléments du langage µCRL2

Nous remarquons une nouvelle annotation, “@BeanProperty”. Celle-ci permet, lors de la compilation, de générer les accesseurs nécessaires pour extraire les propriétés désirées. Elles pourront alors être utilisées par le contrôleur pour générer le format de données voulu et le communiquer vers un client. En l’occurence, le format que nous avons défini dans le contrôleur est le format JSON.

Prenons, par exemple, la classe “Composition”. Cette dernière sera renvoyée vers le front-end dans le format suivant :

```
{
op : "<valeur>",
left : {<autre objet JSON>},
right : {<autre objet JSON>},
term:"composition"
}
```

Analysons de plus près la classe du listing 4.6 : nous remarquons que toutes les propriétés déclarées en argument n’ont pas besoin d’être explicitement déclarées à l’intérieur de la structure. Les méthodes permettant d’y accéder sont aussi implicitement disponibles. Nous avons rajouté une propriété “term” qui nous permettra d’identifier le type de structure une fois l’objet JSON reçu dans le front-end.

Nous voyons qu’une classe abstraite “Mcrl2Object” est étendue par chacun des éléments du langage, présentés ci-dessus. Les “case class” qui les étendent sont des classes optimisées pour être utilisées dans des patterns. Etant donné que tous nos objets ont pour supertype “Mcrl2Object”, l’emploi de ces “case class”, nous permettra d’utiliser un mécanisme intéressant qu’offre Scala : le pattern matching. Découvrons-en l’intérêt au moyen de l’exemple suivant, tiré de notre classe utilitaire.

```
def getActionsFromProcessSpec(p:Mcrl2Object):List[Action] = {
  var answer:List[Action]=List.empty
  p match{
    case comp: Composition => {
      answer:::=getActionsFromProcessSpec(comp.left)
      answer:::=getActionsFromProcessSpec(comp.right)
    }
    case act: Action => answer = answer:+act
    case proc: ProcessId => answer = answer
  }
  answer
}
```

La méthode getActionsFromProcessSpec nous permet de parcourir l’arbre syntaxique et de collecter toutes les actions atomiques.

Cette méthode est appelée pour permettre de vérifier que toute action utilisée dans une spécification se retrouve dans l’ensemble des actions définies au moyen du mot-clé act.

Il s'agit donc d'une méthode récursive qui, à l'origine, prend une spécification pouvant être une composition, une action ou un identifiant de processus. Nous définissons tout d'abord une liste qui contiendra uniquement des actions. Nous observons ensuite l'objet p. Pour connaître son type, nous utilisons le mot-clé `match` et ensuite, pour chaque type d'instance possible, le mot-clé `case`. Examinons le fragment suivant :

```
p match{
  case comp: Composition =>
```

Ce code est à interpréter comme suit : "si p est une composition comp => traiter comp". Dans le cas d'une composition, on ajoute à la liste courante le résultat de la même méthode appliquée à chaque noeud "enfant" de l'arbre. L'extension d'une liste par une autre en Scala s'effectue avec l'opérateur " : : :=".

Le pattern matching que propose Scala nous permet, de manière intuitive, de raisonner sur des patterns d'objets.

4.2.4 La priorité des opérations

Avec la construction de l'arbre, qu'en est-il de la gestion de la priorité des opérateurs ? En observant le code de plus près, nous pouvons en déduire que cette gestion s'opère lors du parsing. Pour faciliter la compréhension, rappelons que, lorsqu'on parle d'opérateur, on parle de composition.

Comme nous l'avons déjà soulevé, une composition, quel que soit son opérateur, ainsi qu'un terme, héritent de la classe Mcrl2Object. Il est donc possible de définir des parsers récursifs où une composition se décrit comme une composition de moindre priorité ou un terme (bloc entre parenthèses ou processus) suivi ou non d'un opérateur et d'une composition du même type. L'arbre se construira de facto en respectant les règles de priorité. Le code ci-après donne simplement l'importance la plus haute à la composition alternative et la plus basse à la composition séquentielle.

Une fois de plus, nous exposons ici que Scala, avec très peu de lignes de code, permet de décrire une partie importante et indispensable de notre langage µCRL2.

```
/**
  * Alternative composition definition
  */
def compositionAlt: Parser[Mcrl2Object] =
    (compositionPar|term) ~ opt("+" ~ compositionAlt) ^^ {
  case a ~ None => a
  case a ~ Some(b ~ c)  => {
    new Composition(b, a, c)
  }
}
/**
  * Parallel composition definition
  */
def compositionPar: Parser[Mcrl2Object] =
    (compositionSeq|term) ~ opt("||" ~ compositionPar) ^^ {
  case a ~ None => a
  case a ~ Some(b ~ c) => {
    new Composition(b, a, c)
  }
}
/**
  * Sequential composition definition
  */
def compositionSeq: Parser[Mcrl2Object] =
    term ~ opt("." ~ compositionSeq) ^^ {
  case a ~ None => a
  case a ~ Some(b ~ c) => {
    new Composition(b, a, c)
  }
}
```

4.2.5 La validation

Dès que le parsing aura retourné un résultat sans erreur, la validation reste une étape indispensable. Notre classe utilitaire “Util”, définie comme objet Scala (classe statique), nous donne des méthodes permettant de générer les exceptions adéquates en cas d’erreurs dans le code. Observons, dans le fragment suivant, comment nous pouvons détecter le cas d’une action utilisée et non-définie dans la liste d’actions `act`.

```
object Util {
          (...)
  def getActionsFromProcessSpec(p:Mcrl2Object):List[Action] = {
    var answer:List[Action]=List.empty
    p match{
      case comp: Composition => {
        answer:::=getActionsFromProcessSpec(comp.left)
        answer:::=getActionsFromProcessSpec(comp.right)
      }
      case act: Action => answer = answer:+act
      case proc: ProcessId => answer = answer
    }
    answer
  }
  def actionListToStringList(actions:List[Action]): List[String] ={
    var answer:List[String]=List.empty
    for(i<-actions.indices){
      answer = answer:+actions(i).name
    }
    answer
  }
  def areActionsDefined(defined:List[Action], toCheck:List[Action]): Boolean ={
    for(i<-toCheck.indices){
      if(!actionListToStringList(defined).contains(actionListToStringList(
          toCheck)(i))){
        return false
      }
    }
    true
  }
}
```

La première méthode, `getActionsFromProcessSpec`, dont nous avons parlé précédemment, permet de générer la liste de toutes les actions utilisées dans une spécification donnée.

Ensuite, nous voyons une méthode `actionListToSTringList` grâce à laquelle il nous est possible de transformer une liste d'actions en une liste de chaînes de caractères représentant chacune le nom des actions de la liste fournie en argument. Au passage, nous remarquons l'économie que nous offre Scala pour la boucle "for". En effet, la méthode `indices` que procure la classe "Liste" nous permet d'itérer sur tous les indices de cette liste.

La méthode `areActionsDefined` est employée lors de la construction de l'arbre pour générer l'exception nécessaire lorsqu'une action utilisée n'est pas définie au préalable. Elle prend comme premier argument la liste des actions réellement définies, et comme second, la liste des actions retrouvées dans les processus définis et obtenue grâce à la première méthode ci-dessus.

Le morceau de code suivant montre que, lors de la génération de l'arbre, le résultat de cette méthode provoque ou non le renvoi d'une exception.

```
for(i<-proc.indices){
  initActions:::=Util.getActionsFromProcessSpec(proc(i).process)
}
if(!Util.areActionsDefined(act,initActions))
  throw new ValidationException("It seems some actions in one or more
      processes aren't previously defined")
```

Nous voyons ici que nous collectons toutes les actions de tous les processus définis et que nous vérifions leur existence dans la clause `act`.

4.2.6 Gestion des erreurs

Les erreurs que nous gérons du côté serveur sont principalement de deux types : tout d'abord les erreurs de parsing, qui provoquent une exception, mais également les exceptions générées lors de la validation sémantique.

En ce qui concerne le parsing, nous venons enrichir notre appel à la méthode parseAll pour récupérer les erreurs qu'il génère et les renvoyer dans la structure de retour dans une propriété "errorMessage".

```
val parsingResult = parser.parseAll(parser.mcrl2Specification, input.getCode)

parsingResult match{
  case parser.Success(r, n) =>{
    initialSpec.setInitialSpec(parsingResult.get.initialSpec)
    initialSpec.setActions(parsingResult.get.actions)
    initialSpec.setInitialCode(input.getCode)
    initialSpec.setProcesses(parsingResult.get.processes)
  }
  case parser.Failure(msg, n) => initialSpec.setErrorMessage(msg)
  case parser.Error(msg,n) => initialSpec.setErrorMessage(msg)

}
```

A nouveau grâce au pattern matching, nous évaluons la réponse du parser. En cas de succès, nous enrichissons la structure `initialSpec` et , en cas d'erreur ou d'échec, nous retournons le message d'erreur correspondant.

Nos méthodes utilitaires permettant de valider le code renvoient des exceptions en cas de non-conformité de la spécification µCRL2. Il convient donc d'encapsuler le parsing dans un bloc try-catch, similaire à celui que l'on peut trouver dans Java :

```
try{
val parsingResult = parser.parseAll(parser.mcrl2Specification, input.getCode)
parsingResult match{
   ( ..... )
}catch{
  case ve:ValidationException => initialSpec.setErrorMessage(ve.getMessage)
}
```

Nous renvoyons ainsi toutes les informations sur les erreurs vers le client qui pourra ainsi les gérer.

4.2.7 Modélisation du back-end

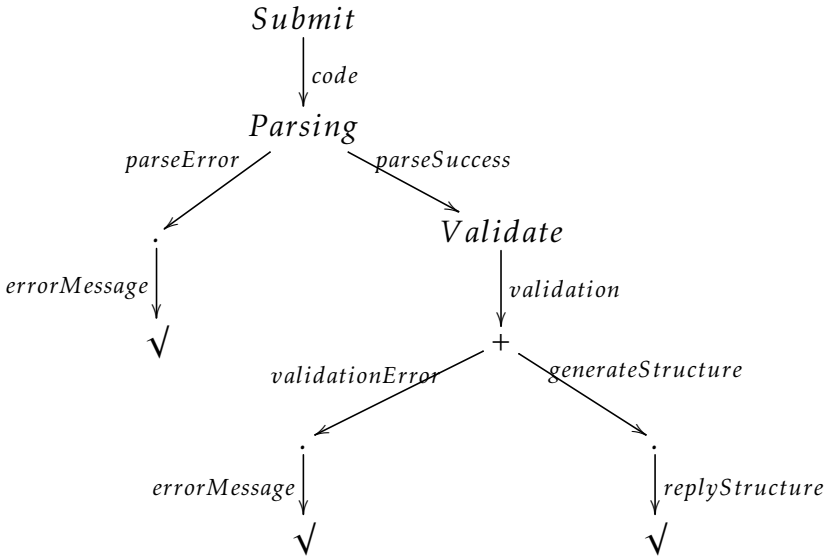
Pour terminer cette section, nous proposons une modélisation abstraite µCRL2 du fonctionnement de notre back-end :

```
act code, parseError, parseSuccess,
    errorMessage, validation, validationError,
    generateStructure, replyStructure;

proc Submit = code.Parsing;
     Parsing = parseError.errorMessage + parseSuccess.Validate;
     Validate = validation.(validationError.errorMessage
                          + ProduceStructure);
     ProduceStructure = generateStructure.replyStructure;

init Submit;
```

En voici la représentation schématique :



Lorsqu'on soumet un code µCRL2, il est transmis au service Mcrl2Service où il est analysé syntaxiquement par les parsers. La sortie du parser est soit une erreur, soit une structure qui est transmise à MCRLFactory pour être validée et produire soit une erreur de validation, soit un arbre syntaxique correspondant à la spécification µCRL2 valide fournie en entrée.

4.3 Le front-end

4.3.1 AngularJs

De la multitude des technologies web disponibles, c'est vers AngularJs que nous nous sommes dirigé. Avant de présenter quelques détails de notre implémentation, il est important, tout d'abord, de donner un aperçu sur AngularJs, ses avantages et inconvénients, dans le cadre de notre travail et, ensuite, de préciser la raison de ce choix au regard des autres technologies disponibles actuellement.

AngularJs se présente sous la forme d'un framework JavaScript. JavaScript est un langage de programmation orienté objet. Il peut s'insérer dans le code d'une page web pour en augmenter le spectre de possibilités et il est exécuté par le navigateur. L'idée principale consiste à rendre les pages web interactives (ouvrir des pop-ups, insérer un menu dynamique, provoquer des actions au passage ou au clic de la souris, ...)[16].

AngularJs vient proposer une architecture MVC [8] qui impose une structure dans le code où toute la logique doit être englobée dans des *contrôleurs* et des *directives*[12].

Notre fichier racine HTML se décrit comme suit :

```
<html>
...
<body ng-app="interpretor">
...
</body>
</html>
```

Nous relions de cette manière notre HTML à une application "interpretor" pour manipuler le DOM [9].[12]

Dans notre fichier Js, nous déclarons une variable comme étant un module `angular` et se présentant comme suit :

```
var interpretor = angular.module('interpretor',[]);
```

Nous observons que le deuxième argument est un tableau vide auquel nous pouvons ajouter autant de contrôleurs que nécessaire, comme le montre l'exemple ci-dessous :

8. Model-View-Controller : Angular est aussi souvent considéré comme un framework MVW : Model-View-Whatever[12]

9. Document Object Model : le modèle contenant les données et méthodes que nous construisons pour interagir avec l'application

```
interpretor.controller('restController',function($scope){
        $scope.name = "interpretor";
});
```

Dans le fichier template (le fichier html), nous relions ce contrôleur soit dans une balise qui se trouve au même niveau que l'application, soit dans une balise "enfant".

```
<html>
...
<body ng-app="interpretor" ng-controller="restController">
...
</body>
</html>
```

De cette manière tout le corps de notre balise <body> pourra manipuler les valeurs du modèle qui est défini dans l'objet $scope. $scope est un objet JavaScript auquel peuvent être ajoutées toutes les variables et fonctions nécessaires à notre application.

La figure 4.1, nous donne un aperçu du fonctionnement d'AngularJs.

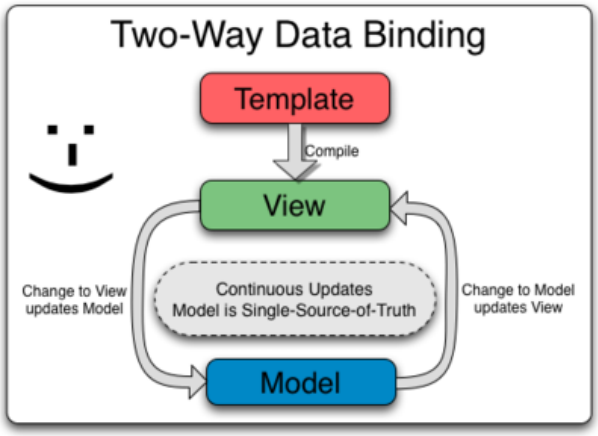


FIGURE 4.1 – Two way data binding[12]

Cette vue schématique montre qu'une valeur du DOM peut être affichée dans le template (view) et qu'elle peut également instantanément être modifiée depuis le template.

Un simple exemple est le cas d'une balise input :

```
<div ng-controller="MyCtrl">
    <input type="text" ng-model="name" />
    <p>Hello {{ name }}</p>
</div>
```

La *directive* ng-model permet ce “two way data binding”. La variable “name” est définie dans l’objet $scope et affichée dans la page html.

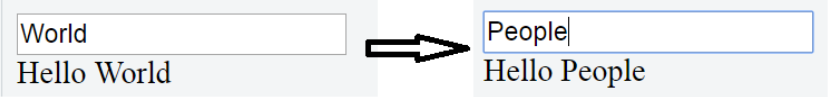


FIGURE 4.2 – Two way data binding : hello world

La figure 4.2 illustre le fait que changer le contenu de la balise input modifie immédiatement la valeur “name” du DOM et l’affiche instantanément à l’écran, sans devoir recharger la page.

Cette réactivité nous intéresse dans le cadre d’une application interactive telle que celle que nous souhaitons développer. En effet, nous devons progressivement rafraîchir un arbre au fur et à mesure des actions de l’utilisateur dans les choix qui lui sont proposés.

Dans cette optique, nous introduisons encore un élément intéressant d’AngularJs : la *directive*. Cette dernière peut être rajoutée à un module angular, à la manière d’un contrôleur. Son objectif est de créer un élément ayant son propre DOM et son contrôleur dans lequel les données nécessaires sont injectées. Nous la lions à un template HTML qui peut être ajouté et réutilisé à la demande dans une page web principale.

```
interpretor.directive('resultTree', function(){
    return {
        restrict: "E",
        templateUrl: '/directives/result-tree.html',
        scope:{
            result:'='
        }
    }
});
```

Le fichier “result-tree.html” contiendra le fragment html réutilisable et supervisé par un contrôleur qu’on lui définit et qui reçoit une valeur de type “objet” dans la variable “result” de son DOM (ici, il s’agit de la structure renvoyée par notre back-end).

Ce fragment html sera intégré à notre page web de cette manière :

```
<result-tree result="result"></result-tree>
```

Nous voyons que le nom de la directive est typographié en camelcase dans le fichier JavaScript alors qu’il l’est en “hyphen-separated” dans le fichier html. Cette convention est adoptée pour la simple raison que le langage html est “case insensitive”.

Nous pouvons voir l'injection de la valeur "result" provenant du DOM de l'appelant dans le DOM lié au contrôleur de la directive.

Cette brève introduction à AngularJs va nous permettre de mieux comprendre le fonctionnement de notre front-end que nous abordons dans la suite de cette section.

4.3.2 L'application front-end

Le choix s'est tourné vers une interface intuitive pour l'utilisateur, comme nous le décrivons dans le chapitre précédent. Nous en présentons ici les éléments principaux.

La page d'accueil propose une zone de texte pour entrer le code µCRL2 ; un bouton se présente à côté pour soumettre ce code. Dans cette section nous proposons un scénario pour cette application, agrémenté d'exemples du code JavaScript utilisé pour réaliser les exigences.

```
<button type="button" ng-click="submit()" ng-disabled=" showResult">Submit</
    button>
```

```
$scope.submit = function(){
    $http({url:'analyze/send',method:'POST',data:{"code":$scope.value}}).then
        (function(data){
        if(data.data.errorMessage == null){
            $scope.result = data.data.initialSpec;
            $scope.showResult = true;
        }else{
            $scope.showResult = false;
            alert(data.data.errorMessage);
        }
    })
};
```

Le fragment de code ci-dessus nous donne la méthode "submit" qui est une fonction du DOM. Lorsque l'utilisateur clique sur le bouton "Submit", la fonction est appelée et exécute la requête http avec l'url du service du contrôleur back-end. Le mot-clé "then" permet de traiter ensuite la réponse du service à laquelle nous donnons le nom générique "data".

L'objet "data" contient la structure de données correspondant à l'arbre de syntaxe produit par notre back-end. Nous vérifions tout d'abord l'absence de message d'erreur, auquel cas nous populons la variable "result" avec le contenu que nous souhaitons manipuler. Le booleen "showResult" nous permet de donner, à la page, l'information que nous pouvons afficher la section contenant le résultat, comme nous le montre le code html ci-dessous :

```
<div ng-if="showResult">
    <result-tree result="result"></result-tree>
</div>
```

La balise “div” contient la directive “ng-if” qui prend notre booleen en paramètre. Cette directive permet de créer la partie du DOM qu’elle englobe. Il est à noter qu’il existe une directive “ng-show” très semblable, mais cette dernière n’a pour effet que de cacher/montrer cette partie du DOM, ce qui ne nous intéresse pas. En effet, au premier chargement de la page, l’élément “result” n’existe pas encore, ce qui provoquerait une erreur du côté client, la directive “result-tree” essayant d’utiliser une variable qui n’existe pas.

La directive “resultTree” reçoit donc la variable “result” à traiter. Son contrôleur “treeController” se charge de manipuler les données. Voici tout d’abord un aperçu, édulcoré pour l’explication, du template html utilisé :

```
<script type="text/ng-template"  id="tree_result.html">
    <button ng-click="getNext(data)" ng-disabled="data.disabled">
                {{data.value}}
    </button>{{data.op}}
    <table>
        <td valign="top" ng-repeat="data in data.left"
                                        ng-include="'tree_result.html'"></td>
        <td valign="top" ng-repeat="data in data.right"
                                        ng-include="'tree_result.html'"></td>
    </table>
</script>

<div ng-app="Application" ng-controller="treeController">
    <span ng-repeat="data in resultTree" ng-include="'tree_result.html'"></span>
</div>
```

La partie entre les balises “script” décrit un template html dans lequel nous trouvons un bouton contenant le processus initial de notre spécification stockée dans “data”. Le clic entraîne l’exécution de la méthode “getNext” du contrôleur que nous aborderons ci-après. Nous y observons ensuite une table avec deux colonnes où ce template s’inclut de manière récursive pour les “enfants” droite et gauche de la racine. Sur le bouton, nous pouvons voir la directive “disabled” qui permet de rendre le clic impossible si l’action qu’il contient n’est pas atteignable dans l’état du système. Cette information est mise à jour à chaque changement d’état par le contrôleur.

La partie entre les balises “div” vient simplement décrire notre directive et vient y inclure le template défini au préalable.

Il n’en faut pas plus pour décrire notre arbre. Voyons maintenant le contrôleur qui gère cette partie et, en particulier, sa méthode “getNext”.

```
$scope.getNext = function(node){

    if(node.data.term == "composition"){

        var leftNode = {
            data : node.data.left,
            done : false,
            disabled : false,
            op   : node.data.op,
            value: node.data.left.readable,
            left:[],
            right:[]
        };

        var rightNode = {
            data : node.data.right,
            done : false,
            disabled : false,
            op   : "   ",
            value: node.data.right.readable,
            left:[],
            right:[]
        };
        node.left.push(leftNode);
        node.right.push(rightNode);
        $scope.processComposition(node);
    }
    if(node.data.term == "processId"){
        $scope.processProcessId(node);
    };
    if(node.data.term == "action"){
        $scope.processAction(node)
    };
        (...)
};
```

Cette fonction prend la donnée node et réalise une modification du DOM en fonction du type de noeud dont il s'agit. Si la donnée qu'il contient est une composition, alors elle contient un membre de droite et un membre de gauche. Elle crée donc deux nouveaux noeuds qu'elle insère dans les enfants du noeud actuel. Elle appellera ensuite la méthode "processComposition" pour ajouter la sémantique nécessaire à ces noeuds en fonction du type de composition (quel noeud est atteignable, quel noeud ne l'est pas).

Dans le cas où le noeud est un "ProcessId", nous ne possédons pas directement l'information du processus qu'il représente et nous devons réinterroger le back-end pour obtenir le processus associé. C'est ce que fait la méthode "processProcessId", que nous développons ici.

```
$scope.processProcessId = function(node){
                (...)
        var myPromise = $scope.getNode(node.data.name);
                (...)
}
```

Dans cette méthode, nous appelons “getNode” qui, à la manière de notre méthode “submit”, soumet un identifiant de processus µCRL2 au back-end dans le but de récupérer la spécification qui se cache derrière :

```
$scope.getNode = function(pid){

    (...)

    $http({url: 'analyze/getNode',method: 'POST',data:{"proc":pid}}).then(
        function(data){

            (...)

    return process;}
};
```

Nous pouvons en effet observer une construction qui appelle une méthode du contrôleur back-end à l’url “analyze/getNode”.

Un autre élément est à retenir : notre structure ainsi construite contrôle en permanence l’état terminé ou non de chaque noeud pour donner à la vue l’information au cas où un processus se termine. Pour cette raison, il est nécessaire de surveiller les modifications qui influencent récursivement tout l’arbre, et donc l’affichage. La méthode “$watch” permet d’effectuer cette surveillance. Observons le fragment suivant :

```
            (...)
$scope.$watch(function(){
    return node.left[0].done;
},function(){
    if(node.left[0].done){
        node.right[1].disabled = true;
    }
});
 (...)
```

Nous apercevons la méthode $watch qui prend deux fonctions sans paramètre en argument : la première retourne la valeur à surveiller et la seconde donne les instructions à exécuter lorsque cette valeur est modifiée. En l’occurrence, nous devinons ici une partie de la logique de la composition alternative qui dit que, si le membre de gauche est sélectionné, alors le membre de droite devient automatiquement inaccessible.

Nous avons ci-dessus brièvement exposé le fonctionnement de notre front-end, Le lecteur pourra explorer la totalité du code dans le dépôt git dont nous donnons l’adresse à la fin de ce chapitre.

4.4 Un peu de recul

L'apprentissage d'un nouveau langage et son utilisation au sein d'un framework que nous ne maîtrisons pas encore est, en soi, un challenge. Après avoir passé quelque temps dans le développement de l'application, certains éléments nous sautent aux yeux.

Nous avons exploité la puissance de Scala pour effectuer un parsing efficace de notre code µCRL2 et une validation des structures générées. Toute la logique en revanche a été confiée aux contrôleurs AngularJs. Il est évident que, dans ce cas, nous sous-utilisons Scala et nous poussons Angular dans ses retranchements.

Un développement plus approprié consisterait à gérer la logique µCRL2 du côté du back-end, et à mettre à jour la structure par appels successifs, ce que ne fait pas la version de notre application à la date du dépôt de ce mémoire. En gérant l'application de cette manière, nous nous permettons une implémentation beaucoup plus légère du côté front-end.

Il serait donc intéressant d'appliquer une réflexion visant à faire glisser la logique µCRL2 vers le back-end. Cela offrirait davantage de services au front-end et le déchargerait d'une bonne partie du code qu'il embarque actuellement.

Les pistes, dans ce cadre, seraient d'enrichir l'arbre généré pour connaître les noeuds visités, et, à chaque requête, de renvoyer l'état de la trace actuelle avec les possibilités qui s'offrent à l'utilisateur dans l'état courant du système.

4.5 Les autres technologies

Différentes raisons nous ont amené au choix de Sring Boot et AngularJs. Il ne s'agit cependant pas de la seule technologie capable de nous procurer les moyens de développer notre application.

Nous avons mis de côté la possibilité de nous diriger vers une application de type monolithique, comme le propose le package Swing de Scala [15]. Cette possibilité n'ayant pas été étudiée en profondeur, nous ne pouvons pas en juger la pertinence dans le cadre de notre développement. Néanmoins, et comme déjà annoncé, la principale raison réside dans le fait de notre souhait de rendre notre application portable.

En ce qui concerne l'architecture adoptée, nous pourrions débattre sur de nombreuses technologies. C'est pourquoi nous proposons ici de poser notre choix au regard du framework Play que nous introduisons.

Play est, à l'instar de Spring, un framework permettant le développement d'applications Web. Depuis sa version 2.0, il a été entièrement redéveloppé en Scala et

il offre la possibilité de développer en Java et en Scala. Notre back-end étant développé en Scala, il pourrait sembler séduisant d'utiliser un tel framework. En voici les principales caractéristiques [10] :

- play est Stateless, cela signifie qu'il n'y a pas de session pour une connexion. Chaque requête qui doit être traitée par le serveur doit entièrement être traitée sur base des seules informations qu'elle contient.
- tous les contrôleurs sont déclarés comme étant statiques (Objets Scala).
- les entrées/sorties sont asynchrones. Contrairement à Spring, cela permet de faire face à une évolution du web vers de plus en plus de traitement de données concurrentes.
- l'architecture est modulaire : play propose le concept de modules.
- le support est natif de Scala : play utilise Scala et expose une API Scala. Il expose également une API Java qui est légèrement adaptée pour respecter les conventions de Java. Cela permet d'interopérer également avec Java.

Play se revendique comme étant une framework permettant d'optimiser la productivité en utilisant des conventions plutôt que de la configuration. Cela permet de très rapidement se consacrer à la réalisation du contenu.[23]

Dans le cadre de ce travail, la connaissance limitée de ce framework nous a amené à compiler certaines informations pour juger de l'adéquation avec nos exigences. De ce que nous avons pu retenir, Play est très adéquat dans la réalisation de sites web (applications web, e-commerce, ...). A contrario, la valeur ajoutée de Spring est de proposer un large panel de composants qui, combinés ensemble, nous donnent beaucoup de possibilités. Lorsqu'il s'agit de construire une application qui requiert notamment des intégrations d'autres logiciels, Spring semble se révéler être l'option la plus adéquate.[18]

Une motivation qui nous a mené au choix de Spring est qu'il s'agit d'une technologie très utilisée dans l'entreprise qui nous emploie. Ayant eu l'occasion d'aborder brièvement, lors de formations et de conférences, les avantages de Spring et d'AngularJs, nous nous sommes dirigé vers une technologie permettant à la fois d'ajouter une valeur à ce travail et de tenter d'apporter des compétences utiles à l'employeur, Oniryx, qui sponsorise ces études.

Le temps nécessaire à l'apprentissage d'un nouveau framework a également influencé notre décision. Dans une telle démarche, faire machine arrière en raison de mauvais choix techniques n'est pas exclu. Bien que les erreurs favorisent l'apprentissage, elles consomment néanmoins beaucoup d'heures, ce qui peut s'avérer compliqué à gérer pour un étudiant à horaire décalé.

Il est cependant certain, dans un souci de veille technologique, que prendre le temps d'explorer les possibilités de Play et de bien d'autres outils s'inscrit, pour notre part, d'ores et déjà à l'ordre du jour pour la suite.

10. Tiré de [21]

4.6 Conclusion

Ce chapitre a exposé les principaux éléments qui permettent de comprendre le fonctionnement de notre interpréteur. Nous avons également pris du recul par le biais d'une auto-critique de notre méthode d'implémentation.

Pour développer une telle application, nous nous appuyons également sur le fait qu'il n'y a pas qu'un seul bon choix. Les nombreux frameworks pourraient être explorés en profondeur. Cependant, il existe toujours un choix arbitraire, souvent guidé par les affinités du développeur.

Le lecteur pourra consulter la version complète, et la plus actuelle, du code de notre application sur le dépôt git suivant :

```
https://forge.info.unamur.be/scm/git/ocroegae_memoire_jacquejm
```

Ayant introduit l'aspect technique de notre application, nous présentons, dans le chapitre suivant, l'aspect fonctionnel de notre outil.

Chapitre 5

Présentation de Procalg : un outil d'interprétation et de représentation pour le langage µCRL2

Les chapitres précédents nous guident tout naturellement vers une présentation de l'outil que nous avons développé. Nous verrons ici que les écrans sont en adéquation avec ce qui a été proposé dans le chapitre 3.

5.1 Démarrage depuis l'IDE

L'IDE [1] choisi pour le développement est IntelliJ. Nous soulignons principalement trois raisons pour cette option :

- IntelliJ est chaudement recommandé par la communauté Scala.
- nous sommes habitué à son utilisation car...
- ...l'entreprise qui nous emploie nous fournit une licence pour son utilisation.

Pour démarrer l'application sous windows, la manière la plus simple est la suivante :

1. Installer et ouvrir IntelliJ.
2. Ensuite, retourner dans le menu "file", aller dans "settings".
3. Dans la fenêtre "settings", sélectionner "plugins" et cliquer ensuite sur "Install JetBrains Plugin".
4. Rechercher et installer le plugin "Scala".
5. Dans le menu "file", sélectionner "new -> project from version control -> git".
6. Utiliser l'adresse fournie à la fin du chapitre précédent pour récupérer toutes les sources sur sa machine.

1. Integrated Developpment Environment

7. Autoriser, lorsqu'il le demande, l'outil de build "gradle" à télécharger les dépendances nécessaires.
8. Dans la section de droite, sélectionner le dossier "src -> main -> scala -> interpretorBackend".
9. Avec le bouton de droite de la souris, cliquer sur "interpretorWebApplication" pour ouvrir un menu contextuel.
10. Cliquer sur "run 'interpretorWebApplication". L'application démarre.
11. Dans le cas où il n'est pas spécifié, IntelliJ invite, dans une fenêtre, à sélectionner le jdk à utiliser. Nous utilisons sa version 1.7 dans le cadre de ce travail.
12. Ouvrir un navigateur et taper l'adresse "http ://localhost :8080/".

5.2 L'interface de l'application

5.2.1 Écran de départ

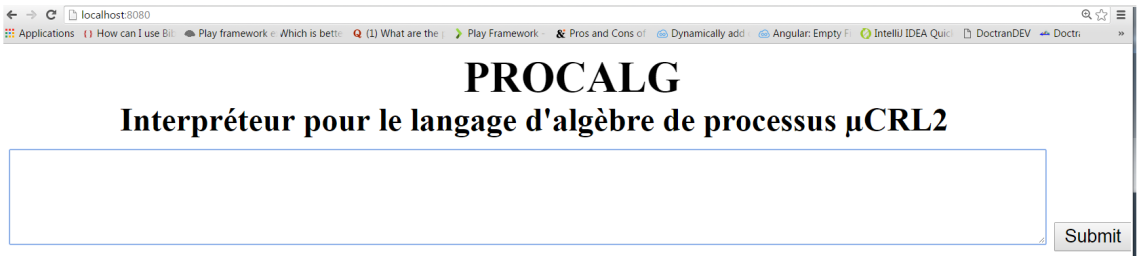


FIGURE 5.1 – Ecran de départ

La figure 5.1 nous donne l'aperçu que nous avons décrit au chapitre 3. Nous pouvons y entrer une spécification µCRL2 comme l'illustre la figure 5.2.

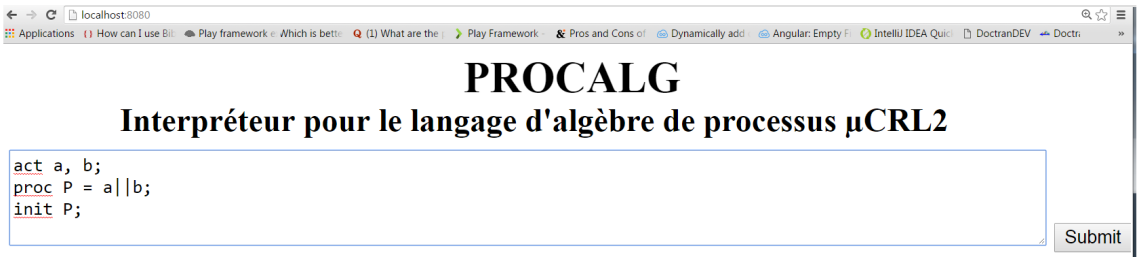


FIGURE 5.2 – Ecran où du code a été inséré

Lorsque le code est soumis au moyen du bouton "submit", l'écran propose la première action détectée. Il s'agit ici de développer le processus P, comme nous pouvons le constater dans la figure 5.3.

FIGURE 5.3 – Ecran où du code a été soumis au back-end

Nous voyons également un bouton "Random" qui permet, à tout moment, d'opérer un choix aléatoire lorsque plusieurs actions sont possibles.

5.2.2 Déroulement du processus

L'écran de la figure 5.4 nous montre un état possible pour le code entré, une fois que nous avons cliqué sur le processus P.

FIGURE 5.4 – Etat du système à un moment donné

Nous observons ici une composition parallèle et donc trois choix initiaux possibles : l'action "a", l'action "b" et la fonction de communication $\gamma(a, b)$, représentée par "a|b". Pour l'exemple, nous optons arbitrairement pour l'action "b", ce qui a pour effet de rendre impossible un clic sur la fonction de communication, étant donné qu'une des deux actions qui la composent vient d'être consommée, comme le montre la figure 5.5.

FIGURE 5.5 – Etat du système à un moment donné

Le seul choix qu'il nous reste consiste à cliquer sur l'action "b", conformément à la définition de la composition parallèle. Cela nous mène à l'état terminé que nous pouvons observer dans la figure 5.6

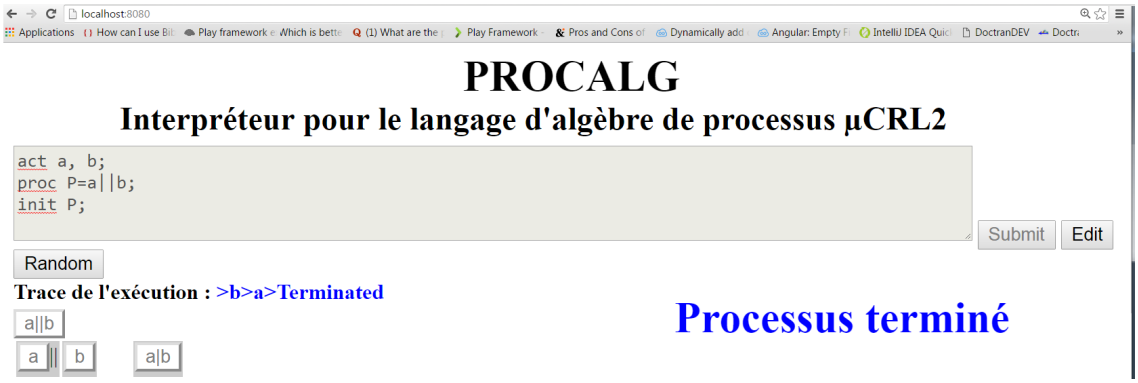


FIGURE 5.6 – Etat du système terminé

Nous recevons ici l'information que le processus s'est bien terminé. En outre, nous possédons une vue sur sa trace qui a été enrichie à chaque action.

Nous apercevons également un bouton d'édition qui a pour effet de rendre la fenêtre d'entrée à nouveau éditable tout en faisant disparaître les informations du processus en cours. Il s'agit d'un retour à l'écran de départ.

5.2.3 Écrans d'erreur

Dans la figure 5.7, nous pouvons observer le résultat d'une soumission qui ne respecte pas la syntaxe µCRL2. Un message apparaît pour donner la raison de l'erreur et guider l'utilisateur. Il s'agit du message de l'exception renvoyée par le parser Scala.

FIGURE 5.7 – Erreur de syntaxe

Enfin, la figure 5.8 nous donne le résultat d'une erreur qui ne respecte pas la spécification µCRL2. Dans l'exemple, est utilisée une action qui n'est pas définie au préalable. Le message provient de l'exception que nous générons dans notre back-end à l'étape de validation.

FIGURE 5.8 – Erreur de syntaxe

5.3 Extensions

L'application ainsi présentée conduit à la réflexion sur des points de développements ultérieurs intéressants. Nous en citons quelques-uns ci-après.

Actuellement, notre version n'implémente pas l'abstraction (τ : `hide` en µCRL2), le deadlock (δ : `delta` en µCRL2), et l'encapsulation (∂_H : `allow` en µCRL2). Il conviendrait de poursuivre le développement pour ajouter ces fonctionnalités.

Offrir le moyen de parcourir toutes les traces d'une spécification (avec la possibilité de stopper ce parcours à un certain niveau) semble pertinent.

Nous pourrions proposer l'analyse de toutes les exécutions possibles au regard de certaines propriétés. L'idée serait de décrire des propriétés attendues d'une spécification (deadlock, récursion infinie, ...) pour que l'analyse détecte si elles sont rencontrées par le système. La logique de Hennessy-Milner [17] ajouterait une valeur dans ce cadre pour définir la bisimilarité entre deux processus.

Afin d'apporter de la convivialité et de rendre l'utilisation plus intuitive, nous pourrions également définir un langage graphique qui permettrait d'associer des images en fonction des actions effectuées. Les images seraient affichées dans un canevas. Par exemple, la modélisation d'un passage à niveaux qui s'ouvre après le franchissement du train pourrait illustrer cette action avec l'image d'un train qui traverse le passage à niveaux.

5.4 Conclusion

Cette section a présenté une application simple et fonctionnelle permettant de soumettre du code µCRL2 dans le but de raisonner sur les processus communicants qu'il représente.

Il s'agit ici d'un sous-ensemble des fonctionnalités qui constitueraient un outil complet pour raisonner au moyen de l'algèbre de processus µCRL2. Il n'est nul doute qu'un travail ultérieur pourrait délivrer une version aboutie, et, pourquoi pas, utilisable en temps qu'application disponible depuis le site µCRL2 lui-même.

Chapitre 6

Conclusion

Ce travail a pu mettre en évidence l'importance d'une bonne compréhension des processus concurrents et communicants dans le cadre du développement de systèmes complexes.

Dans un premier temps, nous avons abordé les bases théoriques sur lesquelles repose notre étude et dont la compréhension permet d'appréhender la raison de l'existence du langage d'algèbre de processus µCRL2.

Puis, nous nous sommes appliqué sur l'objet de ce mémoire : la réalisation d'un interpréteur permettant un raisonnement intuitif sur une spécification algébrique respectant la syntaxe de µCRL2. Nous avons notamment fourni un premier croquis des écrans que nous projetons d'obtenir après le développement de notre outil.

Les choix techniques pour le développement ont ensuite été mis en exergue, accompagnés d'une critique au regard d'éventuelles autres solutions possibles. Les principaux détails d'implémentation ont été expliqués au moyen de fragments choisis dans le code source, dont la totalité est disponible dans le dépôt git à l'adresse `https://forge.info.unamur.be/scm/git/ocroegae_memoire_jacquejm`.

Le rendu final de l'interpréteur ainsi conçu a alors été illustré, avec l'ajout d'une explication des principaux écrans.

Comme nous l'avons déjà souligné, il n'est nul doute qu'à la lumière des connaissances et de l'expérience acquises, notre cheminement ouvre de nouvelles perspectives pour des recherches ultérieures. Les pistes de réflexion qui suivent en témoignent.

Il serait tout à fait pertinent d'effectuer une analyse et une formalisation des besoins auprès des utilisateurs potentiels. Notre expérience en milieu professionnel nous démontre qu'il est important de se connecter régulièrement aux attentes du client afin d'éviter qu'un produit soit peu, pas ou mal utilisé.

Ainsi, par exemple, l'usage d'une tablette pour effectuer des modélisations sur le terrain justifierait amplement l'architecture développée dans ce travail du fait de sa portabilité. En revanche, l'utilisation du programme uniquement sur station de travail poserait la question d'un développement d'une application monolithique entièrement écrite en Scala.

En outre, dans l'état actuel, notre programme effectue peu d'appels vers le back-end en raison de la logique que notre front-end embarque. La limitation de JavaScript pose la question d'un transfert de cette logique vers le back-end, où Scala serait en mesure de fournir les services adéquats. En contrepartie, cela se ferait au prix de nombreux appels depuis la machine "client".

Par ailleurs, nous pourrions envisager un moyen de parcourir toutes les traces d'un processus jusqu'à une certaine profondeur mais aussi d'écrire des propriétés à détecter dans une spécification donnée (par exemple : deadlocks ou processus infinis).

La définition d'un langage graphique permettrait également de rendre notre interface plus conviviale par l'association, dans un canevas, d'images présentées en fonction des actions effectuées.

De plus, il semble aussi opportun de mener une réflexion approfondie en ce qui concerne le choix de la technologie front-end. Nous pensons, par exemple, à sa compatibilité avec le type et la version des principaux navigateurs ainsi qu'à son adéquation avec les types d'interactions que nous souhaitons fournir à l'utilisateur. Il est clair qu'AngularJs offre une réactivité idéale pour nos besoins, mais n'y a-t-il pas d'autres options ? Prenons par exemple Google, Facebook, LinkedIn : ces trois entreprises développent leurs front-end respectivement avec AngularJs, ReactJs et Play. Il serait intéressant de connaître leurs motivations, ainsi que celles d'autres organismes, afin de nous forger une opinion pour nos choix de développement.

Et d'un point de vue plus horizontal, nous serions tenté d'envisager une extension de cet interpréteur pour d'autres langages comme CSP et le π-calculus. L'architecture choisie permet sans aucun doute une telle tâche.

Annexe A

Mise en route de l'interpréteur dans IntelliJ

A.1 Remarques générales

Cette annexe décrit la mise en service de l'application. Cette procédure a été testée avec la version 15.04 d'IntelliJ IDEA [3]. L'installation présuppose la présence du jdk1.7 ou supérieur.

Cette installation a été effectuée avec succès sur Windows 7 pro 64 bits ainsi que Windows 10 pro 64 bits.

A.2 La procédure

- Installer et ouvrir IntelliJ.
- Ensuite, dans le menu "file", aller dans "settings".
- Dans la fenêtre "settings", sélectionner "plugins" et cliquer ensuite sur "Install JetBrains Plugin".

Ces trois premiers points sont illustrés dans la figure A.1

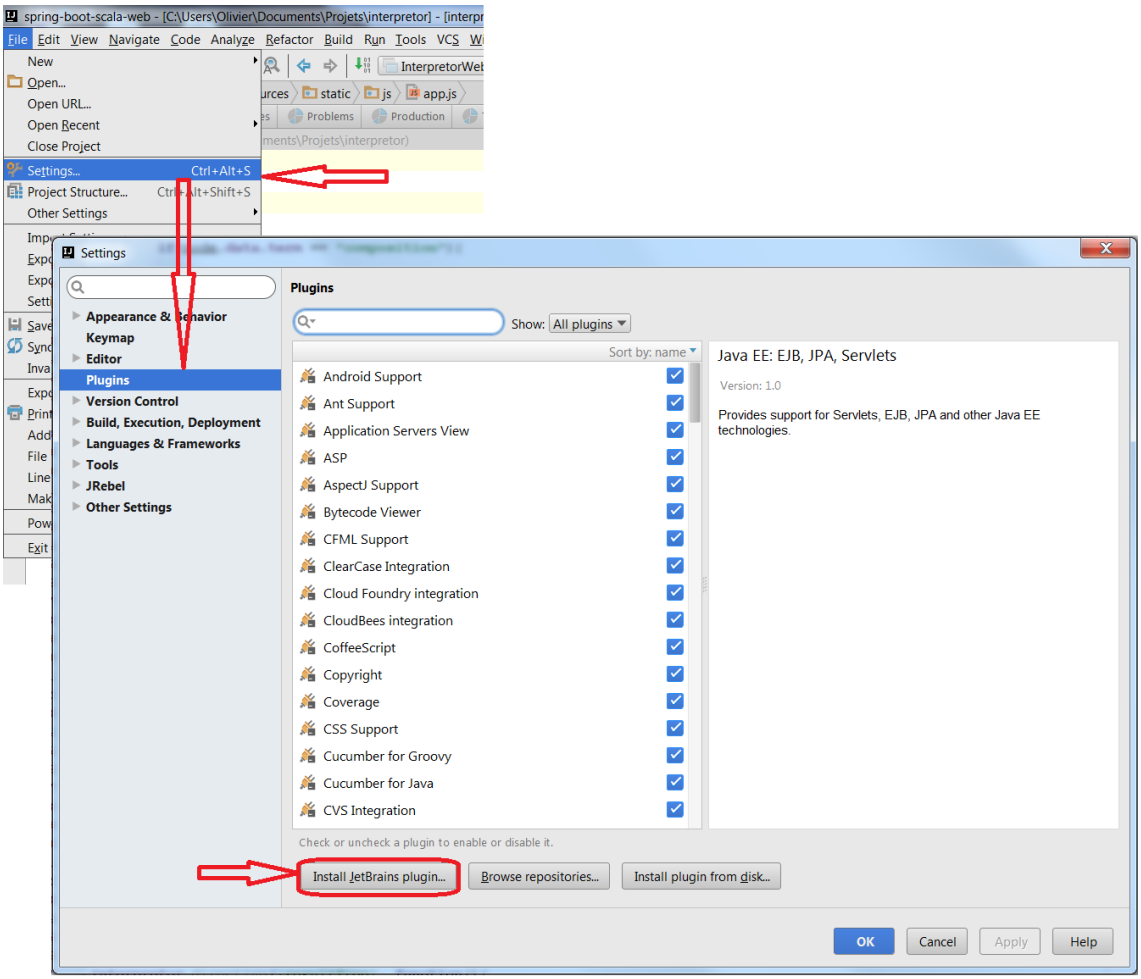


FIGURE A.1 – Configuration étape 1

- Rechercher et installer le plugin “Scala” (figure A.2).

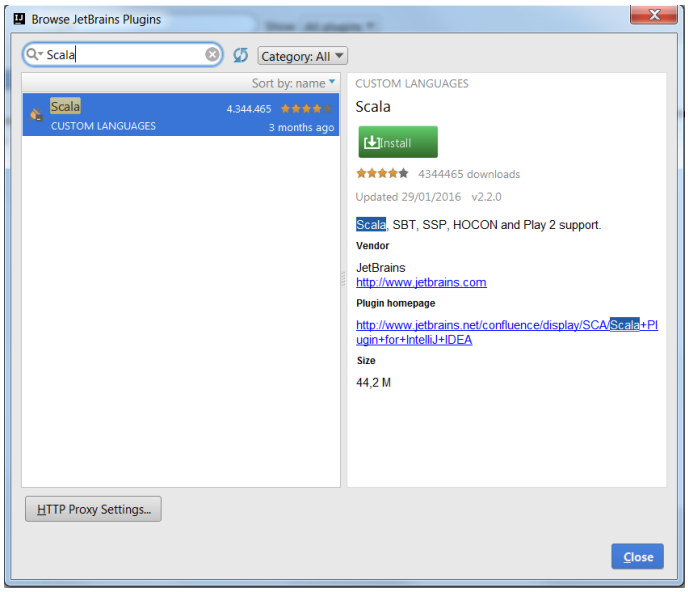


FIGURE A.2 – Configuration étape 2

- Dans le menu “file”, sélectionner “new -> project from version control -> git”.
- Utiliser l’adresse fournie à la fin du chapitre précédent pour récupérer toutes les sources (figure A.3).

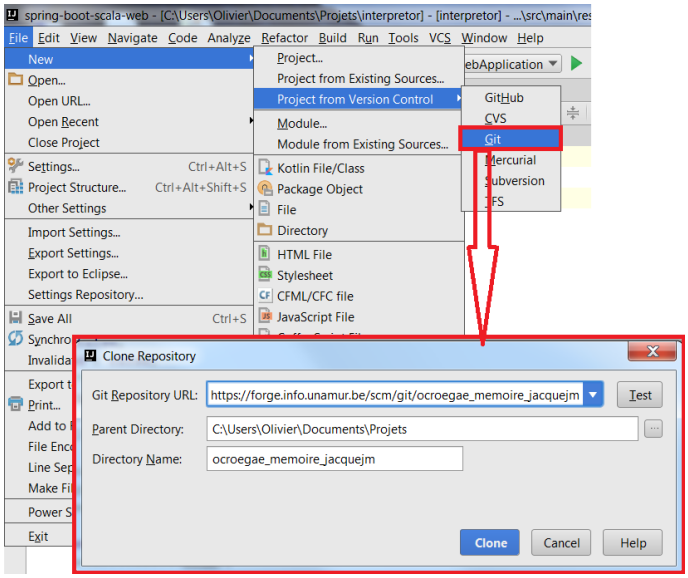


FIGURE A.3 – Configuration étape 3

- Autoriser l'outil de build "gradle", lorsqu'il le demande, à télécharger les dépendances nécessaires.
- Dans la section de droite, sélectionner le dossier "src -> main -> scala -> interpretorBackend" (figure A.4).

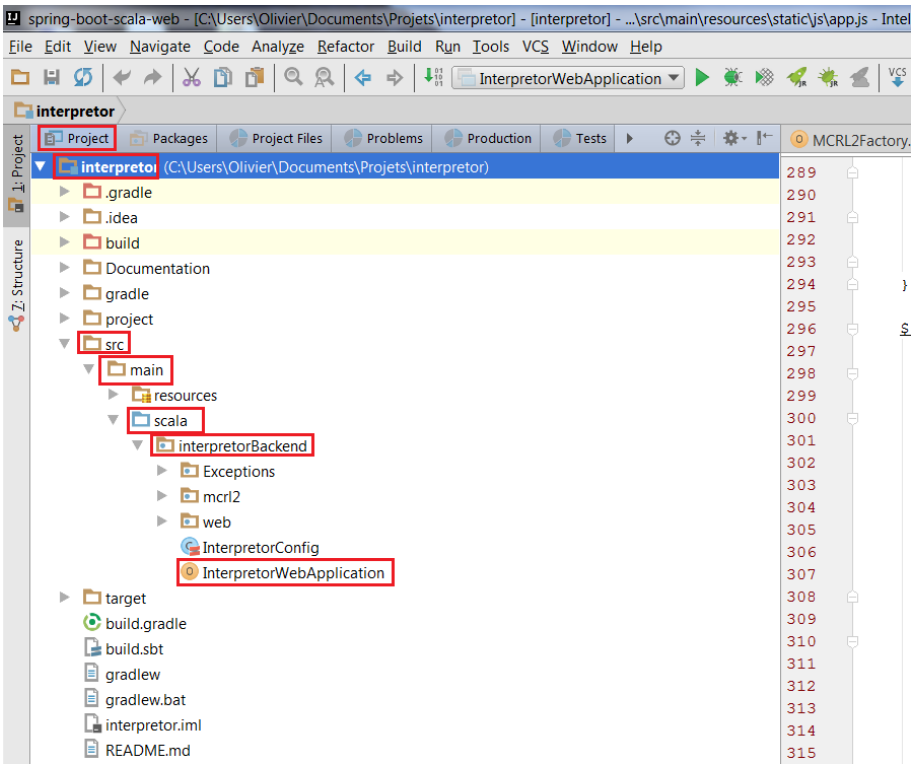


FIGURE A.4 – Configuration étape 4

- Avec le bouton de droite de la souris, cliquer sur "interpretorWebApplication" pour ouvrir un menu contextuel.
- Cliquer sur "run 'interpretorWebApplication" (figure A.5). L'application démarre (figure A.6).

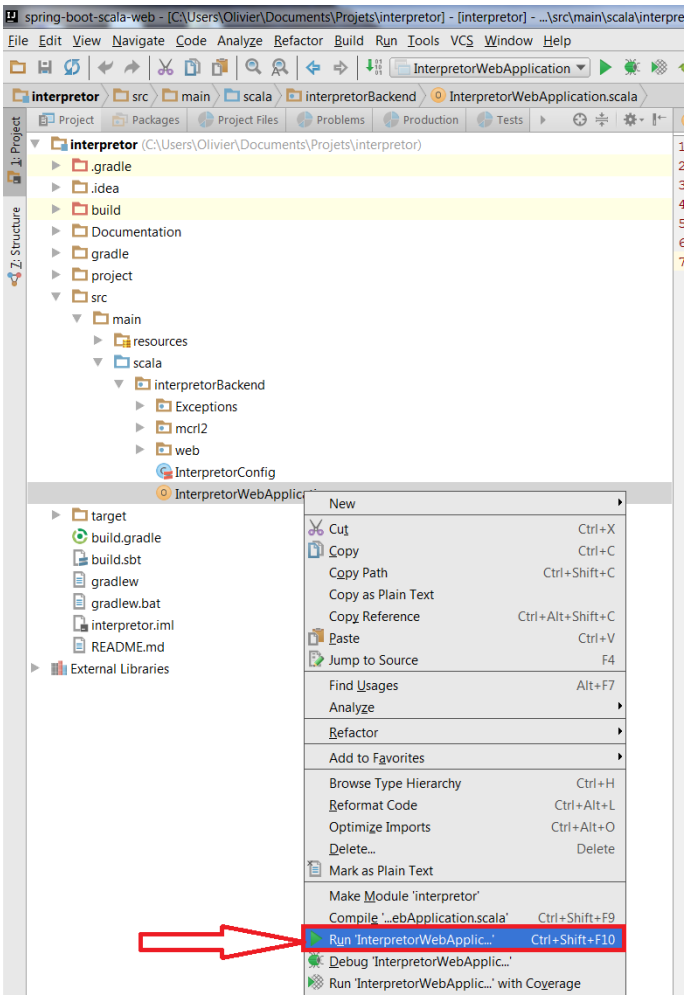


FIGURE A.5 – Démarrage de l'application

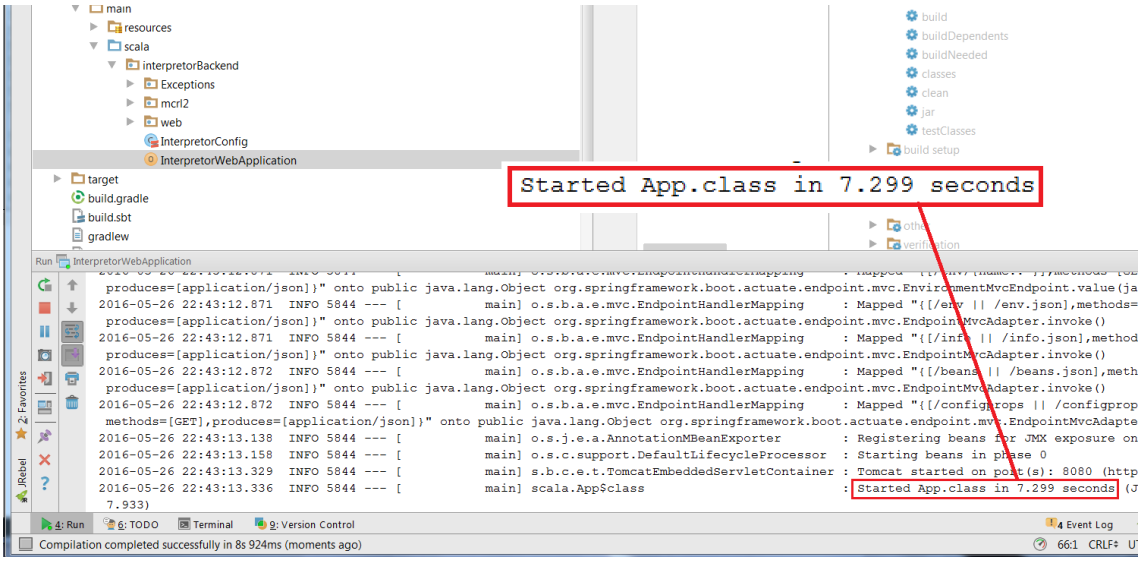


FIGURE A.6 – Application démarrée

- Dans le cas où il n'est pas spécifié, IntelliJ invite, dans une fenêtre, à sélectionner le jdk à utiliser. Nous employons sa version 1.7 dans le cadre de ce travail.
- Ouvrir un navigateur et taper l'adresse "http ://localhost :8080/" (figure A.7).

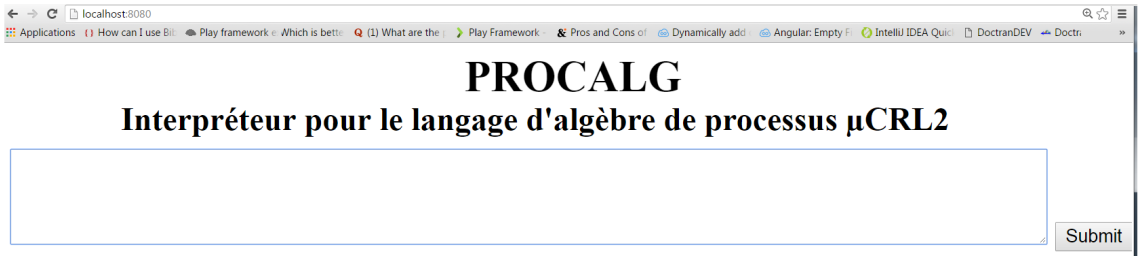


FIGURE A.7 – Application PROCALG

Annexe B

Localisation des sources

L'illustration de la figure B.1 donne un aperçu de la localisation des sources dans l'arborescence du projet au 30 mai 2016.

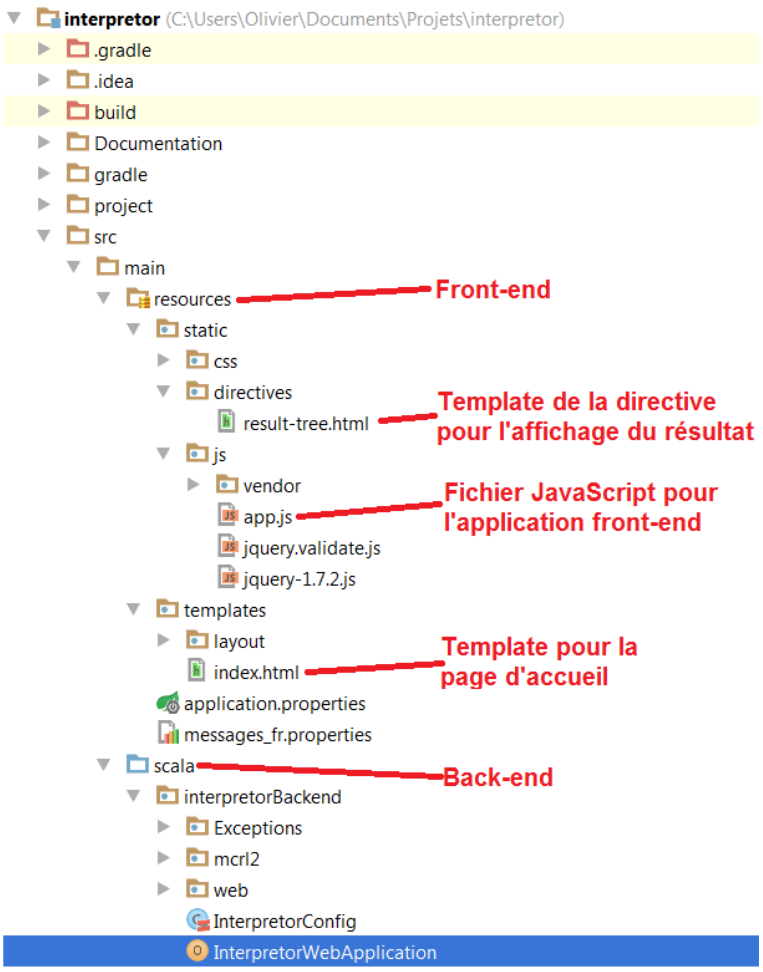


Figure B.1 – Sources du projet

Nous pouvons voir dans le répertoire source “src”, le répertoire “main” qui contient l’entièreté de l’application. Dans ce dossier “main”, le dossier “ressources” correspond à la partie front-end du programme et le dossier “scala” correspond à la partie back-end.

Dans ce dernier, le dossier “interpretorBackend” contient trois dossiers. L’un rassemble les exceptions. Un autre embarque toute la logique µCRL2 ayant été implémentée. Enfin, un troisième contient le contrôleur qui se charge de la communication avec l’application front-end.

Au même niveau que “src” se trouve un dossier “Documentation”. Celui-ci contient les sources du présent document.

Bibliographie

[1] Khan academy. Expressions algébriques. https://fr.khanacademy.org/math/algebra-basics/core-algebra-expressions. Accédé : 2016-04-12.

[2] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science 335 - Elsevier*, pages 131 – 146, 2005.

[3] Jet Brains. Intellij idea. https://www.jetbrains.com/idea/. Accédé : 2016-05-26.

[4] Institut d'électronique et d'informatique Gaspard-Monge UPEM. Le langage scala. http://www-igm.univ-mlv.fr/~dr/XPOSE2011/le_langage_scala/basis.html. Accédé : 2016-05-19.

[5] Université d'Ottawa. Example coffee machine. http://www.site.uottawa.ca/~bochmann/SEG-2106-2506/Notes/M1-2-StateMachines/example-coffeeMachine/. Accédé : 2016-04-15.

[6] Technische Universiteit Eindhoven. Mcrl2 : Analysing system behaviour. http://www.mcrl2.org/. Accédé : 2015-12-30.

[7] Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2nd edition, April 2007.

[8] Marc Fontaine. *A model checker for CSP*. PhD thesis, Heinrich-Heine-Universität, Düsseldorf, 2011.

[9] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/. Accédé : 2016-05-18.

[10] Play Framework. Play framework, the main concepts. https://www.playframework.com/documentation/1.0/main. Accédé : 2016-05-19.

[11] Haskell.org. Haskell - an advanced purely-functional programming language. https://www.haskell.org/. Accédé : 2016-04-15.

[12] Kaliop. Comprendre les bases d'angularjs et savoir quand l'utiliser. http://blog.kaliop.com/blog/2014/01/13/angularjs-presentation/. Accédé : 2016-05-19.

[13] Biju Kunjummen. Spring boot application using scala. https://github.com/bijukunjummen/spring-boot-scala-web. Accédé : 2016-05-18.

[14] Logiflash. Étude de cas : Machine d'état d'alarme. http://fr.logiflash.com/basics/10.php. Accédé : 2016-04-15.

[15] Ingo Maier. *The scala.swing package*, November 2009. http://www.scala-lang.org/old/sites/default/files/sids/imaier/Mon,%202009-11-02,%2008:55/scala-swing-design.pdf. Accédé : 2016-05-26.

[16] OpenClassrooms. Tout sur le javascript. https://openclassrooms.com/courses/tout-sur-le-javascript/presentation-32. Accédé : 2016-05-19.

[17] UK Paweł Sobocinski Univeristy of Southampton. Bisimulation, games & hennessy milner logic. http://users.ecs.soton.ac.uk/ps/teaching/bisimulation.pdf. Accédé : 2016-05-26.

[18] Pietro Bonanno Quora. Which is better, play framework or spring mvc? https://www.quora.com/. Accédé : 2016-05-19.

[19] Stackexchange.com. Simplification of regular expression and conversion into finite automata. http://cs.stackexchange.com/questions/6443/. Accédé : 2016-02-24.

[20] Tech Target. What is rest (representational state transfer)? http://searchsoa.techtarget.com/definition/REST. Accédé : 2016-05-13.

[21] Explained today. Play framework explained. http://everything.explained.today/Play_framework/. Accédé : 2016-05-19.

[22] uml diagrams.org. Uml state machine diagram example. http://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html. Accédé : 2016-02-24.

[23] Antoine Chauvin UPEM. Le framework play 2.0. http://www-igm.univ-mlv.fr/~dr/XPOSE2012/FrameworkPlay2.0/. Accédé : 2016-05-19.

[24] 360 Business Ventures. Gestion des risques. http://www.360businessventures.com/gestion-des-risques/. Accédé : 2016-05-23.

[25] Wikipedia. Groupe. https://fr.wikipedia.org/wiki/Groupe_(mathématiques). Accédé : 2016-04-13.