Vers la vérification des langages de description d'interface utilisateur

Nicolas Nalpon

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)*

Présentée et soutenue le *13 Mars 2023* **par :**

Nicolas Nalpon

Vers la vérification des langages de description d'interface utilisateur

---

JURY

Timothy Bourke	Chargé de recherche	Membre du Jury
Frédéric Dabrowski	Maître de conférences	Membre du Jury
Célia Martinie	Maître de conférences	Membre du Jury
Xavier Thirioux	Professeur ERE	Président du Jury
Cyril Allignol	Enseignant-chercheur	Invité

École doctorale et spécialité :
EDSYS : Informatique 4200018

Unité de Recherche :
Équipe Informatique Interactive (ENAC, Université de Toulouse, France)

Directeur(s) de Thèse :
Pierre-Loïc Garoche et *Celia Picard*

Rapporteurs :
Christine Tasson et *Sylvain Conchon*

Abstract

Les UIDLs (User Interface Descrption Languages) sont des langages conçus pour faciliter la conception des interfaces utilisateurs. Ils permettent de se concentrer sur le développement de l'interface utilisateur sans se préoccuper du reste du programme tout en offrant une syntaxe adéquate à leur description. Cependant ces langages sont utilisés dans des domaines critiques, tels que l'aéronautique ou le domaine médical, alors qu'ils ne permettent pas, en l'état, d'apporter les garanties requises pour ce type d'applications critiques.

Dans cette thèse, nous nous questionnons sur les UIDLs spécialisés dans la description des interfaces graphiques et leur utilisation dans les contextes critiques. Notre approche porte sur l'étude de la sémantique de ces langages et de leur formalisation. Les sémantiques des UIDLs ont peut été étudiées dans la littérature et pourtant, leur formalisation, pourraient permettre de vérifier l'ensemble des interfaces descriptible. Nous présentons des propriétés communes aux UIDLs pour enfin nous questionner sur la façon de les formaliser.

Pour répondre à cette question, nous proposons d'utiliser les bigraphes de Robin Milner un formalisme mathématique permettant de modéliser un système évoluant en espace et en temps. Nous montrons que la théorie des bigraphes est adéquate pour la formalisation de la sémantique des UIDLs et définissons un UIDL ayant pour fondement théorique les bigraphes. La définition d'un tel UIDL permet de pouvoir l'utiliser en tant que langage intermédiaire pour la compilation d'autres UIDL et, de par son intermédiaire, de pouvoir vérifier des interfaces graphiques. Nous essayons notre approche en compilant le langage Smala, un UIDL utilisé dans le domaine de l'aviation, vers l'UIDL définit et en vérifiant certaines propriétés sur des exemples d'interfaces.

Remerciements

Tout d'abord, je tiens à exprimer ma profonde gratitude à tous les membres de mon jury. Je souhaite remercier chaleureusement Christine Tasson et Sylvain Conchon pour avoir accepté de relire mon manuscrit et pour leurs retours constructifs et pertinents qui m'ont ouvert de nouvelles perspectives pour la poursuite de mon travail. Un grand merci également à Timothy Bourke, Frédéric Dabrowski, Célia Martinie et Xavier Thirioux pour les retours qui ont été faits lors de ma soutenance. Ces retours ont contribué à faire évoluer mes pensées.

Je souhaite ensuite exprimer ma reconnaissance envers Sébastien Leriche, qui a dirigé le début de ma thèse. Je le remercie pour ses précieux conseils qui m'ont permis de m'organiser efficacement tout au long de celle-ci. Je remercie également Pierre-Loïc Garoche, qui a pris la relève en tant que directeur de thèse et qui m'a permis d'élargir mes horizons de recherche, ma thèse a pris un nouveau tournant grâce à toi.

Je tiens à exprimer ma gratitude envers mes co-directeurs, Celia Picard et Cyril Allignol, sans lesquels je n'aurais pas pu accomplir ce travail. Durant ces trois années de thèse, vous avez su tirer le meilleur de moi-même et vous m'avez permis de surmonter de nombreuses difficultés. J'ai passé ces trois années de thèse en étant épanoui, j'ai vraiment apprécié travailler avec vous. Merci infiniment pour tout cela.

Mes encadrants et moi-même ne sommes pas les seuls contributeurs de cette thèse. J'ai eu la chance d'être aidé par Michele Sevegnani et Blair Archibald de l'Université de Glasgow qui m'ont permis d'approfondir mes connaissances sur la théorie des bigraphes et qui m'ont beaucoup aidé sur la formalisation présentée dans ce manuscrit, merci pour cela. Enfin, je remercie Julien Saker, Cécile Marcon et Émilie Tortel trois étudiants exceptionnels qui ont contribué respectivement aux chapitres 5, 4 et 6 de cette thèse. J'ai eu le privilège d'encadrer ces trois étudiants lors de leur stage ou projet d'initiation à la recherche et ce fut un plaisir de travailler avec eux.

Une thèse n'est pas une aventure dans laquelle nos interactions se limitent seulement avec nos encadrants. Ces interactions s'étendent jusqu'aux membres de l'équipe dans laquelle nous appartenons. Je tiens donc à exprimer ma reconnaissance envers l'équipe Informatique Interactive de l'ENAC, au sein de laquelle j'ai trouvé un grand plaisir à travailler pendant ces trois ans. Je remercie tout particulièrement l'axe ISI, dont notamment Mathieu Magnaudet, Mathieu Poirier et Stéphane Conversy, pour m'avoir partagé leur vision et aidé à comprendre le framework DJNN/SMALA. Merci également à l'axe IHM, qui m'a fait découvrir une discipline jusqu'alors inconnue pour moi, les différentes présentations de vos travaux auxquelles j'ai pu assister ou bien les discussions que j'ai eues avec vous m'ont permis de développer mon esprit. Et bien sûr, un grand merci aux doctorants pour les moments de détente et de convivialité qu'on a pu vivre. Je remercie en particulier les collègues avec qui j'ai pu partager le bureau C117, merci à Pascal Béger pour sa bienveillance pour mes débuts à l'ENAC, merci à Alice Martin pour ces différents échanges autours de nos sujets de thèse et merci à Florine Simon pour ces moments de partage et de détente qu'on a pu passer. Je remercie aussi Nicolas Viot, Maxime Bardou et Balita Rakotonarivo pour les multiples pauses cafés sans fin qu'on a pu avoir.

La fin de ma thèse a été marquée par mon recrutement à l'EPITA qui m'a généreusement permis de terminer la rédaction de mon manuscrit ainsi que préparer ma soutenance. Je tiens donc à les remercier pour cela. Je remercie tout particulièrement Souheib Baarir, Hubert Gentil, Ghada Garbi, Thibault Lejemble et Quentin Peyras du campus de Toulouse pour leur soutien et leurs encouragements lors de la dernière ligne droite. Je remercie également les membres de l'équipe Automate et Application du

Laboratoire de recherche de l'EPITA, qui m'ont aidé à la préparation de ma soutenance.

Je remercie mes proches qui m'ont soutenu pendant la fin de ma scolarité en commençant par Quentin Garchery et Régis Titus, qui m'ont aidé à surmonter les moments difficiles du MPRI. Je remercie Soheib Biga qui a toujours les bons mots pour me redonner de la force. Merci à Jérôme, Jessica et Jennyka Arulpiragassam, ainsi que Didier Madavy, pour leur soutien régulier lors de nos fameuses tables rondes. Je souhaite remercier mon coach, mon ami et mon frère de cœur, Benjamin Kalita, ainsi que sa femme Manon, pour leurs précieux conseils. Un grand merci à Victoria Gounassegarane pour son soutien et son aide pour la préparation des auditions de l'EPITA. Et enfin, je voudrais remercier mes familles de cœur habitant à Mumbai, les familles Rathod et Makwana, pour leurs soutiens et multiples prières. Une attention particulière à Miral Makwana qui pendant ces trois dernières années m'a aidé à répéter toutes mes présentations qui devaient être effectuées en anglais, merci pour ta patience et ta précieuse aide.

Enfin, je termine en exprimant ma gratitude envers ma famille, qui m'a soutenu toute ma vie. Merci à mes parents, Caroline et Jérôme Nalpon, à ma sœur Delphine Nalpon, ainsi qu'à son conjoint Veechal Ramphal et leur fille Veeya et un immense merci à ma sœur et beau frère, Sophie et Pascal Phaboutdy.

Je dédis ce travail à mes parents qui ont toujours été une source de soutien.
Merci du fond du cœur.

Table des matières

Chapitre 1

Introduction

Les systèmes interactifs sont omniprésents dans notre société suite à la rapide numérisation qui a eu lieu ces dernières décennies. Il s'agit de systèmes qui sont en interaction avec des humains par exemple, des distributeurs automatiques d'argent ou bien des bornes permettant l'achat de billets dans les gares ferroviaires. La communication avec ce type de systèmes se fait au travers d'interfaces utilisateurs qui sont des programmes informatiques. Leur programmation a pendant longtemps été faite à l'aide de langages impératifs en utilisant des callbacks c'est-à-dire des fonctions que l'on associe au déclenchement de certains évènements par exemple l'appui d'une touche. Cependant, la complexité croissante des interfaces utilisateur a conduit à la critique des méthodes employées [28, 32]. En effet, le code généré par ces pratiques ne reflète pas clairement l'interface développée dont notamment la structure et les liens de causalités entre les différents composants de celle-ci.

Les langages de description d'interface utilisateur (UIDLs de l'anglais User Interface Description Langages) sont reconnus comme étant une solution à ce problème. Ces langages sont réputés pour faciliter le développement des interfaces utilisateurs en le rendant indépendant du reste du programme et en proposant une syntaxe adaptée à la description des interactions. Il existe de nombreux types d'UIDLs avec des caractéristiques différentes, comme MARIA qui propose de modéliser différentes perspectives d'une interface utilisateur tels que le modèle de donnée ou bien l'ensemble des évènements la composant [39], ou FXML [13], un UIDL spécialisé dans la description d'interfaces graphiques. Notre intérêt porte sur ce dernier type d'UIDLs permettant de décrire les interfaces graphiques utilisateurs.

Les systèmes interactifs sont aussi omniprésents dans des domaines critiques tels que le secteur de l'aviation ou de la santé. Nous pouvons notamment penser à l'usage des tableaux de bord dans les aéronefs ou bien les différentes interfaces graphiques mises à disposition des soignants permettant de contrôler des outils médicaux. Le développement de logiciels dans ces domaines requiert une attention particulière. En effet, un problème d'exécution dans ces interfaces pourrait entraîner des risques considérables pour la sécurité des vols dans le contexte de l'aviation, la vie des patients dans le contexte de la santé et aussi des coûts importants pour les différents acteurs dans un contexte général. Les incidents autour de l'instrument de radiothérapie Therac-25 [25] illustrent parfaitement les risquent pouvant être rencontrés dans le milieu médical. Entre 1985 et 1987, l'outil Therac-25 a été impliqué dans six accidents qui ont provoqué la mort de plusieurs patients. Ces accidents correspondent à un surdosage des radiations envoyées sur les patients à cause d'erreurs informatiques du système dont une étant une erreur d'implémentation de l'interface utilisateur.

Afin d'éviter des incidents logiciels dans ce domaine, le développement des logiciels est soumis à certaines normes, notamment la norme DO-178C et son complément la norme DO-333 [42]. Celles-ci encouragent l'utilisation de méthodes formelles pour la vérification des logiciels développés et ainsi

assurer leur fiabilité. Une des solutions de vérification formelle utilisée pour la vérification de logiciels, consiste à vérifier que les compilateurs des langages de programmation utilisés préservent la sémantique des programmes spécifiés par le développeur. Cela garantit que le code exécuté respecte bien la spécification attendue. Une façon d'apporter cette vérification à un compilateur est de formaliser chaque transformation le composant à l'aide d'un assistant de preuve et prouver que celles-ci préservent la sémantique de tout programme source compilé. Le projet dans lequel se situe cette thèse consiste à développer un UIDL formel ainsi qu'un compilateur vérifié par assistant de preuve. Actuellement les UIDLs facilitent le développement des interfaces graphiques mais la plupart d'entre eux ne permettent pas, en l'état, d'apporter les garanties requises, par exemple la préservation de la sémantique après compilation, pour la conception d'applications critiques. Un tel compilateur pour un UIDL permettrait de générer du code vérifié correspondant à l'interface graphique ce qui la rendrait plus fiable à l'utilisation dans ces contextes critiques.

Dans la prochaine section, nous passons en revue différents travaux de la littérature traitant de la compilation vérifiée.

1.1 Compilateurs vérifiés

Ces dernières décennies, les compilateurs vérifiés par des assistants de preuve ont su trouver leur place dans le monde de la vérification formelle. De nombreux projets ont traité le sujet de la compilation vérifiée, mais nous parlerons ici seulement de trois d'entre eux, relativement récents, qui se démarquent par le fait qu'ils proposent un compilateur qui a presque entièrement été vérifié et une méthode similaire qui est la construction d'un compilateur par composition. Nous parlerons des compilateurs :

1. CompCert
2. Vélus
3. CakeML

CompCert [24] est un compilateur du langage C, un langage impératif, qui a été formalisé avec l'assistant de preuve Coq[5]. Ce compilateur, composé de neuf passes, est le premier à avoir été vérifié presque entièrement. La méthode utilisée pour vérifier CompCert, reprise par les compilateurs Vélus et CakeML, consiste à décomposer le compilateur en plusieurs sous-compilateurs, selon le nombre de passes de compilation, et à vérifier chacun d'eux. Ainsi, si le compilateur C_1 correspondant au compilateur de la première passe est vérifié et que le compilateur C_2 correspondant au compilateur de la seconde passe l'est aussi alors leur composition est vérifiée. La vérification ne concerne pas seulement la compilation de code C mais concerne également de l'optimisation de code. CompCert comprend des optimisations de code qui ont elles aussi été vérifiées. À ce jour, l'ISO C 2011 est entièrement supporté par le compilateur à quelques exceptions près. Plus précisément, la première passe du compilateur, qui n'est pas vérifiée, se charge de transformer le code source C en du code Clight qui est le sous-ensemble de C entièrement vérifié par le compilateur.

Vélus [10] est un compilateur du langage Lustre, un langage synchrone, qui lui aussi a été formalisé avec l'assistant de preuve Coq. Ce compilateur a été vérifié de la même manière que CompCert c'est-à-dire en utilisant la propriété de préservation de la vérification par composition. Il est constitué de six passes de compilation compilant un programme Lustre en du code Clight et c'est ensuite CompCert qui se charge de terminer la compilation. De la même manière que CompCert, ces passes de compilation incluent des phases de normalisations ainsi que des optimisations elles aussi vérifiées. À ce jour, le langage Lustre est entièrement supporté par le compilateur

CakeML [47] est un écosystème de preuves et d'outils construit autour d'un langage fonctionnel reposant sur un fragment du langage Standard ML, lui aussi fonctionnel. Le compilateur CakeML

est constitué de plusieurs parties dont la plupart sont vérifiées avec l'assistant de preuve HOL4. Le compilateur est composé de neuf passes, dont certaines correspondent à des optimisations. Il a été vérifié de la même manière que les précédents compilateurs. Deux utilisations sont possibles au compilateur CakeML. La première consiste en un outil de production de preuves qui génère un AST CakeML vérifié depuis des fonctions de type ML, formalisées en logique d'ordre supérieur. La deuxième est un compilateur traditionnel, vérifié, compilant du code CakeML en du code machine.

Concernant les autres projets autour de la compilation vérifiée, les états de l'art des articles [24, 10, 47] en développent un bon nombre. Le premier article [24] aborde tous les travaux concernant les preuves mécanisées permettant de vérifier un compilateur publiés jusqu'en 2006, date de publication de l'article. De la même manière que le précédent article, le deuxième article [10] fait un état de l'art et rassemble tous les travaux traitant de vérification de compilation de langages synchrones à l'aide d'assistants de preuve. Enfin, l'article [47] présente quelques travaux autour de la vérification de compilateurs de langages fonctionnels.

De nombreux projets appliquant les méthodes formelles aux interfaces humain-machine existent [50] dans la littérature mais aucun d'entre eux ne traite de la compilation vérifiée. Nous présentons toutefois dans la prochaine section des travaux ayant un aspect formel et concernant les UIDLs ou des langages utilisés pour développer des interfaces graphiques.

1.2 Études formelles sur les interfaces graphiques

En 1996, Bumbulis et al [40] présentent un langage appelé Interconnection Language (IL) utilisé pour décrire comment différentes entités d'une interface graphique sont composées et inter-connectées. Ce langage est utilisé dans le but de générer l'implémentation des interfaces décrites ainsi que leur modèles formels en HOL [38], dans le but de raisonner sur celles-ci. Les auteurs décrivent dans un premier temps la sémantique du langage IL de manière informelle puis dans un second temps présentent sa formalisation au travers de prédicats définis en HOL. Cette formalisation du langage permet d'avoir une formalisation du comportement de toute interface utilisateur décrite avec IL. Les auteurs utilisent ainsi ces formalisations pour prouver que les interfaces respectent certains comportements désirés.

Toujours dans l'objectif de modéliser les interfaces utilisateurs mais cette fois-ci seulement dans le but de faciliter leur conception, Vanderdonckt et al [48] présentent en 2004 USer Interface eXtensible Markup Language (UsiXML). UsiXML est un markup-language, basé sur XML, ayant pour but de décrire des interfaces utilisateur pour différentes plateformes telles que les téléphones et les ordinateurs. UsiXML est défini en plusieurs méta-modèles décrivant différents aspects d'une interface utilisateur. Ainsi avec cet UIDL il est possible de définir différente perspectives d'une interface avec d'une part la possibilité de définir un modèle abstrait qui représente l'ensemble des interactions définissant les différents moyens d'interagir avec l'interface et d'autre part de décrire concrètement les objets qui vont permettre d'effectuer ces interactions. L'association des éléments d'une perspective aux éléments d'une autre perspective leur correspondant est formalisé par des graphes et des règles de réécriture. UsiXML n'est pas le seul UIDL proposant une description sur différent niveau d'abstraction.

Dans le même style Paterno et al [39] présentent en 2009, Model-based lAnguage foR Interactive Application (MARIA). MARIA est un langage de modélisation pour interface utilisateur. MARIA a été inventé pour répondre aux besoins de développement des interfaces utilisateur modernes qui aujourd'hui doivent être déployées sur différentes plate-formes (e.g. web, mobile) et qui ont besoin d'accéder à de multiples services web (e.g. bases de données, requêtes pour un service en particulier). Ces besoins ont été identifiés via une analyse de l'état de l'art, au moment de l'écriture de l'article, et de l'expérience de deux des trois auteurs acquis via un de leurs projets antérieurs, TERESA XML

[8]. MARIA hérite de l'aspect modulaire de TERESA XML i.e. le langage propose un langage permettant de décrire abstraitement l'interface ainsi que d'autres langages qui permettent de raffiner la description abstraite selon les interactions considérées e.g. graphiques, mobile graphique, vocal. Les langages permettant de faire des descriptions abstraites sont bénéfiques aux concepteurs d'interfaces multi-plate-forme. Leurs représentations abstraites, permettent aux concepteurs d'éviter d'apprendre les détails liés aux plate-formes sur lesquelles les interfaces sont déployées et les langages utilisés pour programmer ces plate-formes. Cela permet ainsi de définir une interface générale en se basant seulement sur les interactions que celle-ci doit contenir. MARIA propose ensuite de raffiner une description abstraite selon les plate-formes envisagées pour le déploiement de l'interface utilisateur développée. Le but de ces descriptions, nommées descriptions concrètes, est de fournir une description dépendante de la plate-forme cible mais toujours indépendante des langages utilisés pour l'implémentation de l'interface. UsiXML et MARIA proposent tous les deux le moyen de modéliser une interface utilisateur de manière non mathématiquement formelle.

ICO de Navarre et al [37] présenté en 2009, est un UIDL formel. ICO est fondé sur des concepts de la programmation orientée objet et les réseaux de Petri de haut-niveau, permettant de décrire les spécifications d'une interface utilisateur. ICO a pour but d'augmenter la fiabilité d'une interface développée et a été réfléchi de manière à intégrer le processus de développement d'une interface sans gêner l'utilisation des outils industriels déjà mis en place pour le développement de celle-ci. Les concepts empruntés à la programmation orientée objet tels que l'héritage, l'encapsulation, l'instanciation dynamique ou la classification sont utilisés afin de décrire les aspects structuraux et statiques d'une interface. Les réseaux de Petri de haut-niveau, sont eux utilisés pour décrire leur comportement. Le choix d'utiliser des concepts venant de la programmation objet n'a rien d'anodin. Cela découle du fait que la programmation objet est un paradigme assez répandu en industrie et par conséquent, facilite la prise en main du formalisme. La seule difficulté d'ICO, pour les concepteurs non habitués aux formalismes mathématiques, est de s'approprier les réseaux de Petri afin de spécifier le comportement d'une interface. Les réseaux de Petri permettent l'utilisation de tout outil qui leur est destiné mais c'est l'outil Petshop, accompagné d'ICO, qui est préconisé dans l'article, permettant de concevoir, prototyper et valider les logiciels interactifs.

Chatty et al [15] de l'équipe informatique interactive (II) de l'ENAC ont publié en 2015 des résultats portant sur la vérification de propriétés graphiques d'interfaces reposant sur de l'analyse statique de code. L'analyse est faite sur du code écrit via l'environnement de programmation Djnn. Djnn [27] est un environnement de programmation basé sur un modèle de logiciel interactif dans lequel tout programme peut être décrit comme un arbre de composants interactifs [14]. Dans cet environnement, les composants basiques tels que les structures de contrôle et les objets graphiques sont assemblés afin de produire de nouveaux composants. Une arborescence modélisant une interface utilisateur et créée à l'aide de Djnn peut alors être exploitée afin de s'assurer que l'interface modélisée respecte des propriétés graphiques la spécifiant. Les vérifications sur cette arborescence sont effectuées via des expressions XPath [1] [2] qui formalisent les propriétés à vérifier. Le code Djnn décrivant le modèle peut facilement être compilé en du code XML, cela a guidé les auteurs vers l'utilisation de XPath pour effectuer des vérifications préliminaires.

Béger et Prun de l'équipe II de l'ENAC poursuivent ces travaux dans une thèse [6], publiée en 2020, dont les résultats ont donné lieu à une publication [41] en 2022. Les résultats présentés consistent en la vérification de propriétés graphiques, toujours au travers de l'analyse statique, sur du code Smala. Smala est un UIDL développé par l'équipe II basé sur Djnn, et proposant un modèle conceptuel ainsi qu'une syntaxe permettant de décrire plus facilement les différentes hiérarchies et interactions

1. Langage, du consortium WWW, permettant l'exploration d'arbre XML
2. https ://www.w3.org/

composant l'interface modélisée. La thèse traite de deux problématiques. La première concerne la formalisation des éléments d'une scène graphique d'une interface. L'absence de résultat à ce sujet dans la littérature a mené l'auteur à développer un formalisme pour les scènes graphiques basé sur les variables de Bertin et Barbut [9] qui permettent de représenter des éléments graphiques. Ces variables définissent un élément selon ses coordonnées cartésiennes, sa taille, sa valeur, son grain, sa couleur, son orientation et sa forme. Plus précisément, le formalisme de Béger consiste en une logique de premier ordre utilisant les variables de Bertin et Barbut et mettant à disposition des opérateurs permettant de comparer des figures selon certains critères. Ces opérateurs peuvent être utilisés dans le but de déterminer si une figure en recouvre une autre ; si une figure en intersecte une autre ; si une figure est incluse dans une autre ; si une figure est égale à une autre ; et si la couleur d'une figure est égale à la couleur d'une autre. En deuxième problématique, l'auteur se questionne sur la possibilité de mécaniser les vérifications d'exigences graphiques des systèmes informatiques interactifs. Pour répondre à cette question, l'auteur propose l'outil GPCheck. GPCheck est un outil, écrit en Java, permettant de vérifier des exigences graphiques sur des modèles Smala. L'outil prend en entrée des exigences graphiques, formalisées avec le formalisme défini, ainsi qu'un modèle Smala. GPCheck utilise un algorithme permettant de générer une formule logique (en la logique présentée) depuis le modèle Smala et les exigences données qui vont ensuite être données à un SMT-solver. Si la formule est satisfiable alors les exigences sont respectées par le modèle Smala sinon elles ne le sont pas. D'autres travaux portant sur l'analyse statique de code d'interface graphique existent en dehors de l'ENAC tels que Verified React.

La création de la bibliothèque JavaScript React par Meta a permis d'améliorer le web-développement en permettant de concevoir des logiciels avec des architectures plus claires et d'écrire du code moins sujet aux erreurs. Dans cette lancée, Meta a conçu ReasonML qui est un langage basé sur le système de type d'OCaml et proposant une syntaxe familière aux utilisateurs de C et JavaScript. Reposant sur le fondement théorique d'OCaml, il est possible d'appliquer toutes les techniques de raisonnement automatiques possibles sur un programme OCaml sur du code ReasonML. Cette dynamique de Meta qui apporte un aspect formel autour du langage JavaScript, termine avec l'apparition de la bibliothèque ReasonReact. La bibliothèque permet de concevoir des composants React avec le langage ReasonML et a motivé l'émergence de nouveaux projets de vérification formelle autour de celle-ci. Un article de blog datant de 2019 [1] présente ainsi Verified React, un projet en cours d'évolution dont les contributeurs ont pour but de développer des outils de vérification pour le langage ReasonReact. Les composants React correspondent à des machines à état potentiellement infini. Ces outils permettent la vérification de propriétés sur ces composants ainsi que leur exploration.

Dans les deux sections précédentes, nous avons dressé un état de l'art des domaines de la compilation vérifié et des méthodes formelles appliquées aux interfaces humain-machine. La popularité des projets CompCert, Vélus et CakeML justifie l'utilisation des compilateurs vérifiés et l'absence de tels projets dans le domaine des interfaces humain-machine rend légitime le projet que nous commençons. Dans la prochaine section, nous détaillons notre approche et présentons les problématiques auxquelles nous nous confrontons.

1.3 Problématiques

Comme dit précédemment, notre projet consiste à la définition d'un UIDL formel et de son compilateur vérifié. Un UIDL supportant les modélisations formelles a pour avantage de permettre tout raisonnement sur les interfaces décrites. Ce qui permettrait de faire des vérifications formelles sur une interface, e.g. vérifier qu'elle respecte sa spécification, mais aussi nous aiderait à construire notre compilateur vérifié. Ce que nous attendons de notre compilateur, c'est qu'il préserve la sémantique

des interfaces graphiques décrites avec notre UIDL. La sémantique des interfaces graphiques est caractérisée par des caractéristiques spatiales et interactives. Nous aimerions, plus précisément, que notre compilateur conserve ces caractéristiques. L'idée est de définir ce compilateur avec l'assistant de preuve Coq pour ainsi prouver que chacune des transformations le constituant préservent bien ces caractéristiques.

Cette thèse, apporte sa première pierre à l'édifice, nous commençons ainsi par étudier la sémantique des UIDLs ainsi que leur formalisation afin de débuter ce projet en définissant cet UIDL formel. Les sémantiques des UIDLs spécialisés dans les interfaces graphiques ont été peu étudiées dans la littérature et pourtant leur formalisation permettrait la vérification de l'ensemble des interfaces qu'elles couvrent. Malgré leurs différences, ces UIDLs ont tous le même objectif de description des interfaces graphiques, et partagent de ce fait de nombreuses caractéristiques et propriétés qui se reflètent dans leur sémantique. Trouver et définir un tel ensemble de caractéristiques et de propriétés nous permettrait de définir un UIDL minutieusement. La problématique suivante se pose donc :

Quelles sont les caractéristiques et propriétés communes des sémantiques des UIDLs spécialisés dans la description des interfaces graphiques ?

Cette question en soulève une autre qui concerne la formalisation de ces caractéristiques et propriétés définissant les UIDLs et concernant ainsi indirectement leur sémantique. En effet, la description des interfaces graphiques repose sur la description du positionnement des composants graphiques ainsi que leurs interactions. Nous nous en doutons, ces caractéristiques communes concerneront les aspects spatiaux et interactifs des interfaces utilisateurs et nous nous questionnons sur le formalisme à utiliser pour les représenter. Ce choix n'est pas à prendre à la légère surtout lorsque l'utilisation d'assistants de preuve est envisagé. Effectivement, nous nous confrontons à la modélisation de langages pouvant exprimer des connaissances de nature spatiales et interactives et un mauvais choix concernant le formalisme utilisé pourrait entraîner des difficultés de raisonnement sur le modèle étudié.

Comment pouvons-nous représenter les aspects spatiaux et interactifs de la sémantique des UIDLs, de façon à pouvoir raisonner efficacement sur celle-ci ?

Enfin, nous nous posons une dernière question. Dans le but de faciliter le processus de vérification du compilateur de notre langage, nous aimerions que notre UIDL soit le plus simple possible, c'est-à-dire avec le moins d'instructions possibles. Pour limiter le nombre d'instructions, il faudrait que celles-ci soient atomique, i.e. qu'on ne puisse pas les redéfinir avec d'autres instructions du langage, et le plus expressive possible. Ici, lorsque nous parlons d'expressivité nous sous-entendons l'expressivité spatiale et interactif qu'une instruction possède. Donc :

Est-il possible de définir un UIDL aux instructions atomiques qui permettent de former des constructions interactives et spatiales plus complexes ?

1.4 Contributions

Afin de répondre aux trois questions précédentes, les contributions effectuées dans cette thèse sont les suivantes :

— Nous donnons une définition des caractéristiques et propriétés communes aux UIDLs spécialisés dans la description d'interfaces graphiques. Ces caractéristiques et propriétés sont obtenues après avoir comparé plusieurs UIDLs et s'avèrent être liées à la description du positionnement des éléments graphiques et à la description de leurs interactions.

— Nous mettons en lumière l'importance du choix d'un cadre mathématique pour formaliser la sémantique des UIDLs. Ce choix à un certain impact sur la représentation des connaissances de la sémantique des UIDLs et en conséquence sur les futurs raisonnements qu'on aura sur celle-ci. Cet argument nous a poussé à juger les bigraphes [31] comme une solution pertinente pour formaliser la sémantique de notre UIDL. La théorie des bigraphes, définie par Robin Milner, propose un cadre mathématique permettant de formaliser les aspects structurels et interactifs d'un système ainsi que son évolution dans le temps.
— Nous définissons un UIDL que nous nommons *Biguil* (Bigraphical User Interface Language), couvrant uniquement les aspects spatiaux et interactifs des interfaces graphiques, et donnons une formalisation de sa sémantique entièrement basée sur la théorie des bigraphes.
— Nous donnons la preuve que *Biguil* respecte une des propriétés essentielles des UIDLs et décrivons une démarche permettant de vérifier automatiquement que la sémantique d'une interface graphique décrite par du code *Biguil* respecte certaines de ces propriétés.
— Enfin, à titre d'illustration, nous avons compilé le langage Smala vers *Biguil* pour tester l'expressivité de notre UIDL et partager son aspect formel aux interfaces décrites avec Smala. Smala est un langage développé par l'équipe Informatique Interactive de l'ENAC que nous présentons dans le chapitre 2.

1.5 Organisation

Cette thèse est constituée de sept chapitres à lire de façon séquentielle pour avoir une meilleure compréhension du document. Le chapitre 2 introduit les concepts essentiels à la compréhension de la thèse. Il donne une vue d'ensemble sur les UIDLs, présente le langage Smala utilisé en tant que cas d'étude et présente la théorie des bigraphes. Le chapitre 3 compare deux formalisations d'un sous ensemble de Smala permettant ainsi de révéler l'impact du choix d'un cadre formel sur la représentation des connaissances d'une sémantique. Nous donnons la définition de *Biguil*, notre UIDL formel, ainsi que la formalisation, via les bigraphes, de sa sémantique dans le chapitre 4 . Le chapitre 5 explique comment nous avons compilé le langage Smala vers *Biguil*. Le chapitre 6 discute de la correction de la sémantique de *Biguil*. Une preuve permettant de justifier le respect d'une des propriétés communes par la sémantique de *Biguil* est donnée et nous montrons comment il est possible de vérifier automatiquement le respect de ces propriétés dans une interface utilisateur décrite avec *Biguil* et Smala.

Le chapitre 7 conclut cette thèse et présente des perspectives afin de poursuivre ce projet.

1.6 Publications

Certains résultats présentés dans cette thèse ont fait objet de publication :
— Les résultats présentés dans le chapitre 3 ont donné lieu à des publications [35, 36].
— La sémantique présentée dans le chapitre 4 a été initiée par les résultats de la publication [29].
— Enfin, les caractéristiques des UILDs ainsi que le choix des bigraphes pour leur formalisation sont des résultats de la publication [34].

Chapitre 2

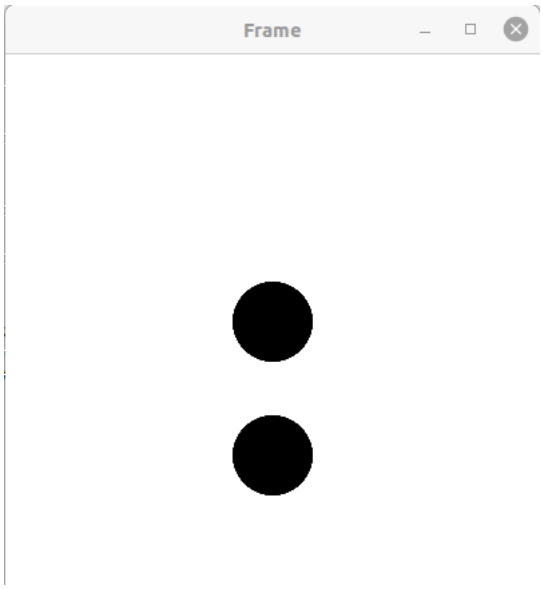
Introduction des concepts

Ce chapitre présente les concepts essentiels que nous utilisons dans cette thèse. Nous commençons par présenter dans la section 2.1 les langages de description d'interface graphique, que l'on nomme UIDLs venant de l'anglais User Interface Description Languages, ainsi que les caractéristiques primordiales les définissant et nous terminons par présenter les propriétés qui en découlent. Puis, la section 2.2 présente Smala, un langage développé par l'équipe informatique interactive de l'ENAC, que nous utilisons comme cas d'étude dans cette thèse. Nous finissons par présenter la théorie des bigraphes dans la section 2.3, le cadre formel que nous avons choisi pour formaliser la sémantique de notre UIDL.

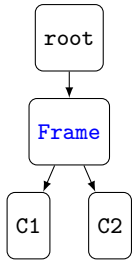
2.1 Les UIDLs

Les UIDLs (User Interface Description Languages) sont des langages de programmation généralement utilisés pour concevoir des interfaces utilisateur. Le développement des interfaces graphiques a pendant longtemps été fait en utilisant des langages impératifs accompagnés du mécanisme de callback. Les callbacks, sont des fonctions que nous définissons et rattachons à des évènements. De cette façon, à chaque fois que l'évènement est déclenché, le callback rattaché est exécuté. Comme exemple nous pouvons citer une simple fonction qui affiche un message sur le terminal, correspondant au callback, associée à l'évènement appui d'un bouton. Cependant, ces pratiques ont été critiquées [28, 32] car le code engendré par celles-ci ne reflète pas clairement l'interface utilisateur développée. En effet, des enchevêtrements complexes de code peuvent apparaître dès lors que la définition d'un callback entraîne la définition d'autres callbacks. Des aller-retours sont alors nécessaires afin de comprendre les liens de causalité entre les différentes entités composant l'interface.

En distinguant clairement le développement de l'interface utilisateur du reste du programme tout en offrant une syntaxe adéquate pour modéliser les composants graphiques d'une interface et leurs interactions (e.g. QML signals and slots [16]), les UIDLs sont reconnus comme une solution à ce problème. Dans cette section nous présentons plus en détails les UIDLs. La section 2.1.1 présente comment la syntaxe des UIDLs permet de structurer facilement une interface graphique et de définir des interactions entre différents composants. La section 2.1.2 présente les deux caractéristiques définissant les UIDLs ainsi que des propriétés sur leur sémantique, que nous avons trouvé par analyse de la sémantique de différents UIDLs.



(a) Exemple d'interface graphique.



(b) Graphe de scène représentant l'interface graphique.

FIGURE 2.1 – Interface graphique ainsi que son graphe de scène.

2.1.1 Syntaxe, structure et event handler

Il est commun de décrire l'aspect spatial d'une interface graphique avec une structure d'arbre appelée le graphe de scène. Chaque nœud du graphe de scène correspond à une entité graphique de l'interface et les arêtes nous donnent une indication sur l'imbrication des composants graphiques. Ainsi, un nœud ayant un enfant correspond à une entité graphique contenant une autre entité graphique. La figure 2.1 illustre une interface graphique ainsi que son graphe de scène. La figure 2.1a représente une interface graphique qui consiste en une fenêtre contenant deux cercles noirs l'un au-dessus de l'autre. La figure 2.1b représente le graphe de scène correspondant à cette interface graphique. Nous pouvons y voir un arbre ayant pour racine un nœud correspondant à la fenêtre, ayant pour enfant un nœud `c1` représentant le premier cercle noir et un autre noeud enfant `c2` représentant le deuxième cercle noir. Dans la suite, nous présentons un concept primordial des UIDLs qui consiste en la définition de graphes de scène que nous illustrons par des exemples FXML et QML, deux UIDLs spécialisé dans la description d'interface graphique.

Afin de décrire l'aspect spatial d'une interface graphique, les UIDLs offrent une syntaxe adéquate permettant de définir un graphe de scène. Cette syntaxe repose souvent sur des langages à balise tel que le XML ou le JSON qui permettent de facilement représenter une structure d'arbre par encapsulation de balise. La figure 2.2 illustre cela en présentant l'interface graphique de la figure 2.2a ainsi que du code QML, dans la figure 2.2b, du code FXML, dans la figure 2.2c et du code JavaFX, dans la figure 2.2d, la représentant. Le code JavaFX est présenté pour contraster avec la syntaxe que QML et FXML proposent pour décrire une interface. L'interface décrite par la figure 2.2a, correspond à une fenêtre contenant un carré rouge, de taille 50 et positionné aux coordonnées $(0, 0)$ de la fenêtre, qui contient un carré bleu, de taille 25 lui aussi aux coordonnées $(0, 0)$ de la fenêtre. Le code QML utilise seulement la capacité de la syntaxe JSON à imbriquer les éléments pour superposer des figures. En conséquence dans la figure 2.2b, le rectangle rouge est déclaré en premier puis vient la déclaration du rectangle bleu, imbriquée dans la définition du rectangle rouge. Le code FXML de la figure 2.2c,

déclare un `StackPane` qui est un layout, i.e. une structure de disposition, permettant de superposer tout ses enfants dans l'ordre de leur définition. Deux `Rectangle` sont ensuite déclarés comme enfant du `StackPane` et, par conséquence, le rectangle bleu est superposé sur le rectangle rouge. Le code JavaFX de la figure 2.2d correspond exactement au code FXML à l'exception des balises XML absentes. Ici, la définition de l'interface est séquentielle.

l1 Le layout `StackPane` est défini

l3 Le carré de taille 50 est défini

l4 Nous fixons la couleur du carré défini précédemment à rouge

l5 Le carré rouge devient le premier enfant du `StackPane`

l6 Nous définissons le deuxième carré de taille 25

l7 La couleur bleu est associé à ce carré

l8 Le carré bleu devient le deuxième enfant du `StackPane`

Bien que le code JavaFX ressemble au code FXML, ce dernier au travers de ses balises XML a une meilleure représentation du graphe de scène. Ce premier concept décrit, nous présentons maintenant un dernier concept des UIDLs consistant à définir des interactions dans une interface graphique que nous illustrons lui aussi par des exemples.

Les UIDLs permettent d'exprimer les interactions via deux mécanismes qui sont les gestionnaires d'événement et les bindings. Les gestionnaires d'événement sont des fonctions que des événements peuvent exécuter (dans l'esprit des callback) et les bindings permettent de faire communiquer les changements de valeur d'une variable à une autre. Dans les figures 2.2b et 2.2c nous pouvons voir un `Button` qui est défini et du code est associé respectivement aux attributs `onMousePressed` et `onClicked` permettant d'afficher le message `"rec2 clicked"` sur le terminal. Les gestionnaires d'événement ressemblent beaucoup aux callbacks, nous pouvons donc nous demander qu'est ce qui diffère par rapport aux anciennes pratiques de développement qui ont été critiquées et pourquoi utiliser encore des callbacks ? Ce qui est différent cette-fois, c'est que les gestionnaires d'évènements sont correctement ordonnés grâce à la structure d'arbre engendré par les UIDLs.

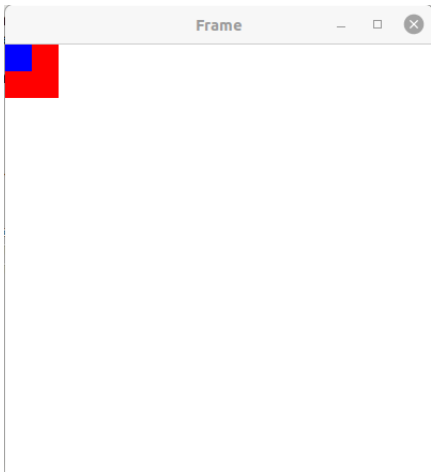
Dans la figure 2.2c, nous pouvons voir qu'un binding est défini, entre la hauteur du rectangle `rec2` et la hauteur du rectangle `rec1`, via la syntaxe `height="${rec1.height}"`. Le même binding est défini dans la figure 2.2b sauf que la syntaxe utilisée est `height : parent.height` qui indique que la hauteur du rectangle courant est connectée avec la hauteur de son parent. Par conséquence, à chaque fois que la hauteur du rectangle rouge est modifiée alors la hauteur du rectangle bleu va l'être aussi en conséquence et prendre la même valeur.

Nous venons de voir à l'aide d'exemples décris en FXML et QML les fonctions de base des UIDLs. Nous avons choisi FXML et QML de par leur popularité pour présenter les fonctions des UIDLs, cependant de nombreux UIDLs différents existent et une comparaison détaillée d'UIDLs basés sur XML peut-être trouvée dans la littérature[19]. Dans la prochaine section, nous présentons les caractéristiques qui définissent les UILDs et des propriétés sur leur sémantique.

2.1.2 Propriétés communes aux UIDLs spécialisés dans les interfaces graphiques

Malgré la diversité des UIDLs spécialisés dans les interfaces graphiques, deux caractéristiques [46] les définissent.

Caractéristique 2.1.1 (Représentation du graphe de scène) *Cette caractéristique des UIDLs spécialisés dans les interfaces graphiques, concerne la capacité du langage à décrire le graphe de scène*



(a) Exemple d'interface graphique.

```
Rectangle {
 id: rec1
 width : 50
 height : 50
 color : "#ff00000"
 Rectangle {
  id: rec2
  width :25
  height : parent.height
  color : "#0000ff"
  MouseArea {
     anchors.fill: parent
     onClicked: {
     console.log("rec2
         clicked")
     }
  }
 }
}
```

(b) Code QML correspondant à l'exemple.

```
<StackPane fx:controller="javafxml.Sample"
           xmlns:fx="http://javafx.com/fxml"
           prefHeight="200"
           prefWidth="400">
  <Rectangle fx:id="rec1" width="50" height="50" fill="#ff0000"/>
  <Rectangle
     fx:id="rec2" width="25"
     height="${rec1.height}" fill="#0000ff"
     onMousePressed="java.lang.System.out.println('rec2 clicked');"/>
</StackPane>
```

(c) Code FXML correspondant à l'exemple.

```
StackPane s = new StackPane();
s.setMaxSize(400,200);
Rectangle rec1 = new Rectangle(50,50);
rec1.setFill(Color.RED);
s.getChildren().add(rec1);
Rectangle rec2 = new Rectangle(25,25);
rec2.setFill(Color.BLUE);
s.getChildren().add(rec2);
rec2.heightProperty().bind(rec1.heightProperty());
rec2.setOnMousePressed(
        new EventHandler<MouseEvent>() {
            public void handle(MouseEvent t) {
             System.out.println("rec2 clicked");
      }});
```

(d) Code JavaFX correspondant à l'exemple.

FIGURE 2.2 – Exemple de code décrivant une interface graphique.

d'une interface graphique. Pour faire usage de cette caractéristique, les UIDLs fournissent généralement un mécanisme d'encapsulation permettant de former des hiérarchies sur les entités graphiques, ce qui permet ainsi de créer un graphe de scène.

Caractéristique 2.1.2 (Représentation des interactions) *Cette caractéristique des UIDLs spécialisés dans les interfaces graphiques, concerne la capacité du langage à décrire les interactions entre les différents composants d'une interface graphique. Pour faire usage de cette caractéristique, les UIDLs fournissent généralement des gestionnaires d'événement et des bindings, permettant de gérer les interactions d'une interface.*

Bien que ces UIDLs soient implémentés différemment, nous avons trouvé des propriétés communes aux sémantiques de FXML, QML et Smala après les avoir analysés. Ces propriétés sont étroitement liées aux deux caractéristiques ci-dessus et sont fondamentales au bon fonctionnement des interfaces définies. Nous présentons ces propriétés dans la suite.

Propriété 2.1.1 (Lois de composition) *Une loi de composition est respectée parmi les éléments constituant le langage afin d'éviter les compositions d'éléments graphiques impossibles, e.g. une fenêtre contenue dans un rectangle, ou bien les connexions entre évènement impossible, e.g. un clic de bouton qui active un autre clic. Dans les UIDLs étudiés, les lois de composition sont gérées par les systèmes de type des langages de programmation sous-jacent aux UIDLs. Par exemple avec FXML, les formes géométriques sont de type Shape, type défini dans le langage Java, qui n'ont pas la propriété d'avoir des enfants. Au contraire les layout en FXML sont de type Java Layout et possèdent cette propriété. Ainsi il n'est pas possible d'imbriquer quelconque élément dans un élément de type Shape. Plus généralement une loi de composition peut être vue comme un ensemble de règles de typage sur les éléments composant l'UIDL.*

Propriété 2.1.2 (Affichage parent-enfant) *Un composant enfant n'est jamais activé sans que son parent soit activé.*

Propriété 2.1.3 (Entités persistantes et transitoires) *Deux types d'entités peuvent être décrites à l'aide d'un UIDL.*

1. *Le premier type que nous appelons persistant, correspond aux composants qui ont une exécution longue. Parmi ces composants nous retrouvons les composants fenêtres ou bien les composants géométriques qui une fois affichés restent affichés jusqu'à la fin du programme ou jusqu'à qu'une interaction mette fin à leur exécution.*
2. *Le deuxième type que nous appelons transitoire, correspond aux composants qui ont une exécution éphémère. Parmi ces composants nous retrouvons les composants acteurs des interactions tels que l'action appui d'un bouton ou bien le clic d'une souris.*

Propriété 2.1.4 (Propagation des interactions) *Les interactions sont utilisées afin d'interagir avec les interfaces graphiques et les mettre à jour e.g. appuyer sur un bouton d'une interface affiche une nouvelle vue. En généralisant, si une interaction a lieu, celle-ci peut déclencher le fonctionnement de certains composants de l'interface graphique qui vont à leur tour activer d'autres composants, etc. L'interaction déclenche une réaction en chaîne au sein du programme. Cette réaction en chaîne est définie en cohérence de l'ordre de causalité des composants i.e. l'ordre d'activation des composants de l'interface est déterminé par les relations de causalité de ses composants.*

Propriété 2.1.5 (Terminaison et confluence d'une réaction) *Les réactions en chaîne déclenchées par les interactions terminent toutes et sont confluentes si jamais des activations concurrentes ont lieu dans la réaction. Cela permet à l'interface de rester réactive et déterministe.*

Dans cette section, une présentation générale des UIDLs a été donnée. Nous avons vu la raison de leur invention, qui peut être illustrée avec les figures 2.2c et 2.2d, leur fonctionnement et les deux caractéristiques les définissant, ainsi que les propriétés définissant leur sémantique. Dans la prochaine section, nous nous attarderons sur SMALA, un langage conçu pour développer des interfaces, utilisées dans l'aviation, et qui est le langage d'étude de cette thèse.

2.2 Smala

SMALA est un UIDL conçu par l'équipe Informatique Interactive de l'ENAC pour développer des interfaces utilisées dans l'aviation. Ce langage se démarque par les trois caractéristiques suivantes [27] :

1. SMALA propose de modéliser une interface à l'aide d'une abstraction de base nommée le PROCESSUS, qui sera présentée dans la section 2.2.1. Les PROCESSUS dans SMALA ont pour avantage d'unifier les concepts de programmation. Ce choix est fondé sur l'hypothèse que moins il y a de concepts de base dans un langage de programmation, plus il est simple de l'apprendre.
2. SMALA est un langage orienté interaction. Par son modèle conceptuel ainsi que sa syntaxe, le langage permet de spécifier efficacement les interactions d'une interface dont notamment les liens de causalité entre PROCESSUS. De plus, les expressions les définissants ont été conçues pour être courtes et intuitives.
3. Enfin, SMALA répond au besoin de la conception itérative, qui est une méthode de conception en cycle dans lequel le produit est prototypé, testé et amélioré jusqu'à ce qu'il réponde à nos attentes, en permettant de changer le graphisme ou les interactions d'une interface sans une refonte complète du code.

Les fondements du modèle de SMALA reposent sur la notion de PROCESSUS présentée dans la section 2.2.1, la notion de TOPOLOGIE présentée dans la section 2.2.2 et la notion de DYNAMICITÉ présentée dans la section 2.2.3.

2.2.1 Les processus

Les processus sont l'abstraction de base du langage SMALA. Dans le paradigme fonctionnel, tous les éléments du langage sont décrits comme des fonctions, dans le paradigme orienté objet les éléments sont décrits comme des objets et de la même manière dans SMALA tout les éléments sont des processus. Un processus est défini par les deux éléments suivants [27] :

— un *identifiant* unique
— un *état* (soit Activé ou Désactivé), indiquant si le processus est allumé et peut exécuter sa sémantique, ou éteint et ne peut exécuter sa sémantique tant qu'il n'est pas allumé.

Les PROCESSUS sont fondamentaux dans SMALA. Étant utilisés en tant que structures de contrôle, composants graphiques et interactifs, pour interagir avec la mémoire ou bien communiquer, ils permettent de constituer un programme. La section suivante les présente et explique comment les utiliser pour modéliser une interface graphique.

2.2.2 Topologie

La figure 2.3 présente la grammaire de SMALA et en même temps comment il est possible de définir un programme. Un Component est un PROCESSUS pouvant contenir des processus enfants et les activer dans l'ordre de leur définition lors de son activation. Les PROCESSUS pouvant être définis dans ce PROCESSUS `root` sont :

Programme	Prog : :=	`Component root` $\{C\}$
Expression	e	Classical unary and binary expressions
Identifiant		id
Types [1]	$t_c \in$	{`Int`,`Double`,`String` ,`Component`,...}
Liste de processus	$C ::=$	$C\ c \mid \epsilon$
Processus	$c ::=$	$t_c\ id\ P\ \{\ C\ \} \mid t\ id\ P$ $\mid FSM\ id\ \{\ S\ T\ \}$ $\mid id\ Cop\ id \mid e\ CopE\ id$ $\mid$ `move` $id\ <\ id$ $\mid$ `move` $id\ >\ id$ $\mid$ `move` $id\ >>$ $\mid$ `move` $id\ <<$ $\mid$ `addChildrenTo` $id\ \{\ C\ \}$ $\mid$ `remove` id `from` id
États	$S ::=$	`State` $id\ \{\ C\ \}$
Transitions	$T ::=$	$id\ \rightarrow\ id\ (id)$ $\mid id\ \rightarrow\ id\ (id, id)$
Liste de paramètre	$P ::=$	$P\ p \mid \epsilon$
Paramètre	$p :=$	$id \mid t \mid e$
Opérateur de couplage	$Cop :=$	$\rightarrow\ \mid !\rightarrow\ \mid\ \rightarrow! \mid !\rightarrow!$
Opérateur de couplage avec expression	$CopE :=$	$=:\ \mid\ =:>\ \mid\ =>$

FIGURE 2.3 – Grammaire de SMALA.

— Les PROCESSUS conteneurs $t_c\ id\ P\ \{\ C\ \}$, ayant un type t_c e.g. Component (`Component`), un id, une liste de paramètre optionnel P et une liste de processus enfant C. Les machines à états (`FSM`) et leurs états (`State`) sont aussi des PROCESSUS conteneurs.

— Les PROCESSUS préalablement prédéfinis $t_c\ id\ P$ qui doivent être nommés et instanciés avec les bons paramètres comme par exemple le PROCESSUS graphique cercle, `Circle c (0,0,10)`, qui a trois paramètres de type DOUBLE correspondant respectivement aux coordonnées de son centre et à son rayon.

— Les PROCESSUS de couplage $id\ Cop\ id$, de couplage avec expression $e\ CopE\ id$, de suppression de PROCESSUS `remove` id `from` id , de re-localisation de PROCESSUS `move` id ... et d'ajout de PROCESSUS `addChildrenTo` $id\ \{\ C\ \}$ que nous présentons dans la prochaine section et qui permettent de mettre en relation les PROCESSUS et modifier la topologie du programme.

La définition d'un programme SMALA induit la création de hiérarchies et de connexions entre les PROCESSUS. Ces hiérarchies et ces connexions peuvent être respectivement représentées par des structures mathématiques d'arbre et de graphe.

Pour décrire ces hiérarchies, nous nous appuyons sur l'exemple de la figure 2.4a à des fins de les illustrer. Dans ce programme,

l3 une fenêtre de taille 400×400 et qui a pour titre `"Circle"` est définie

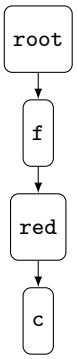
l4 puis vient la définition d'un PROCESSUS permettant de colorer tous les prochains PROCESSUS graphique en rouge

l5 un PROCESSUS cercle positionner aux coordonnées $(200, 200)$ de la fenêtre et de rayon 50 est défini

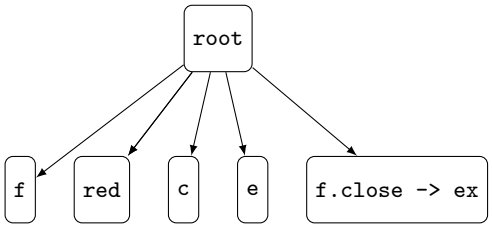
l6 un PROCESSUS permettant de quitter le programme lors de son activation est définie

```
1  _main_
2  Component root {
3    Frame f ("Circle", 0, 0, 400,
       400)
4    FillColor red (255,0,0)
5    Circle c (200,200,50)
6    Exit e (0,1)
7    f.close -> e
8  }
```

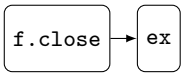
(a) Programme SMALA – Programme 1.



(b) Graphe de scène du Programme 1.



(c) Arbre des composants du Programme 1.



(d) Graphe de lien du Programme 1.

FIGURE 2.4 – Topologie d'un programme SMALA.

l7 enfin un couplage est défini entre le PROCESSUS `Exit` et le PROCESSUS `f.close`, qui représente le bouton fermer de la fenêtre.

Deux types de hiérarchie existent dans un programme SMALA. La première, induite par les PROCESSUS conteneurs, forme un arbre comme illustré dans la figure 2.4c. Les PROCESSUS conteneurs forment des nœuds (e.g. `root`) de l'arbre et les PROCESSUS contenus sont des feuilles (e.g. `f`, `red`) ou bien des nœuds enfants dans le cas où le processus contenu est lui-même un conteneur. L'ordre des enfants dans l'arbre est analogue à l'ordre de définition des PROCESSUS dans un programme, l'enfant le plus à gauche correspondant au premier PROCESSUS défini et celui le plus à droite au dernier. La seconde hiérarchie, reflétée par l'arbre de la figure 2.4b et correspondant au graphe de scène du programme, est induite par l'ordre de définition des PROCESSUS graphiques. Parmi les PROCESSUS graphiques trois types de PROCESSUS se distinguent : les PROCESSUS fenêtres, les PROCESSUS représentant des propriétés graphiques et ceux représentant les formes géométriques. Certaines règles basées sur ces trois types de PROCESSUS sont définies pour former la seconde hiérarchie. Ainsi, dans cette hiérarchie, les PROCESSUS fenêtre sont toujours des enfants de la racine de l'arbre et les PROCESSUS représentant les propriétés géométriques et ceux représentant les formes géométriques sont toujours enfants de la dernière fenêtre ou de la dernière propriété géométrique définie. Les connexions, représentées par un graphe, sont issues des communications entre PROCESSUS définis dans le programme. Ces communications sont issues de la définition des PROCESSUS suivant : BINDING, CONNECTOR, ASSIGNMENT et FSM présentés dans la prochaine section. Une seule communication est définie dans la figure 2.4a (ligne 7) et la figure 2.4d la représente.

SMALA est composé de plusieurs bibliothèques qui proposent l'utilisation de tout type de PROCESSUS. Le fonctionnement de ces PROCESSUS est expliqué, via des tutoriels, sur le site internet de SMALA [2]. Maintenant que nous avons décrit la topologie d'un programme SMALA, nous allons voir

2. www.smala.io/reference.html

comment il est possible de l'exploiter ou bien de la modifier à l'aide des PROCESSUS.

2.2.3 La dynamicité

La notion de DYNAMICITÉ couvre trois différents aspects : 1) la propagation de changement d'état d'un PROCESSUS, 2) la gestion de la mémoire et 3) la création dynamique de PROCESSUS.

Propagation de changement d'état

Comme avec les autres UIDLs, la communication entre PROCESSUS, qui est unidirectionnelle dans SMALA, se trouve au cœur du langage. À son lancement, une application SMALA est inactive jusqu'à ce qu'elle détecte un évènement produit dans son environnement (e.g. une touche appuyée, un clic de souris). Chaque évènement, peut enclencher une activation ou une désactivation de PROCESSUS qui à son tour va propager son changement d'état à d'autre PROCESSUS, une réaction en chaîne se produit. Dans l'exemple figure 2.4a, le PROCESSUS `f.close` représente le bouton permettant de fermer une fenêtre. La ligne `f.close -> ex` indique que lorsque ce PROCESSUS est activé alors son activation va être propagée à `ex` qui correspond au PROCESSUS `Exit` qui une fois activé va mettre fin à l'exécution du programme. Les structures de graphe et d'arbre formant la topologie d'un programme SMALA ont un impact majeur dans la propagation de ces changements d'état.

1. D'une part, il est possible de propager les changements d'états en s'appuyant sur l'arborescence (telle celle de la figure 2.4c) induite par les Component définis dans le programme. Cela est possible seulement lorsqu'un `Component` s'active (respectivement se désactive) et doit activer (respectivement désactiver) ses PROCESSUS enfants. La propagation va s'effectuer en parcourant en profondeur de gauche à droite le sous-arbre ayant pour racine le `Component` dont l'état a été mis à jour. Ainsi, à chaque fois qu'un nœud du sous-arbre va être parcouru, alors l'état du PROCESSUS correspondant au nœud va être mis à jour en conséquence.
2. D'autre part, il est possible de propager les changements d'état en s'appuyant sur la structure de graphe (telle celle de la figure 2.4d) induite par les communications définies dans le programme. Cette structure est induite par quatre PROCESSUS utilisés dans le but de diffuser des changements d'état ou de donnée dans le programme.

   1) Le PROCESSUS $p_1 \rightarrow p_2$ appelé BINDING met en relation deux PROCESSUS p_1 et p_2. Celui-ci, permet de changer l'état d'activation de p_2 de désactivé à activé lorsque l'état de p_1 passe lui-même de désactivé à activé. En plus du BINDING $p_1 \rightarrow p_2$ qui vient d'être présenté, trois autres BINDING existent dans SMALA les BINDING $p_1! \rightarrow p_2$, $p_1 \rightarrow !p_2$ et $p_1! \rightarrow !p_2$. Si le symbole ! est situé en amont du BINDING alors cela indique que le BINDING modifie l'état de sa cible lorsque la source passe de l'état activé à désactivé. Si le symbole ! est placé en aval du BINDING alors cela indique que le BINDING passe sa cible de l'état activé à désactivé au moment où sa source aura subi un changement d'état. Par exemple, le BINDING $p_1! \rightarrow p_2$ permet de passer l'état de p_2 de désactivé à activé lorsque l'état de p_1 passe d'activé à désactivé.

   2) La machine à état est un processus permettant de structurer le code en plusieurs états et de passer d'un état vers un autre lorsque certaines actions se produisent. Les états sont des PROCESSUS conteneurs spécifiques à la machine à état et un seul état est activé à la fois. Une transition consiste en la dés-activation de l'état activé (ainsi que de ses enfants) pour activer un nouvel état.

   3) + 4) L'opérateur => appelé CONNECTOR ainsi que l'opérateur =: appelé ASSIGNMENT permettent de propager les changements d'état d'un certain PROCESSUS appelé la PROPERTY.

Ayant un impact sur la mémoire du programme, ces PROCESSUS sont présentés en détail dans la section suivante.

Afin que l'interface décrite soit réactive à tout instant de son exécution, les communications dans un programme SMALA sont acycliques et en conséquence le graphe induit par les PROCESSUS présentés précédemment est un graphe acyclique orienté. Les communications inter-PROCESSUS peuvent être vues comme un chemin entre un nœud source et un nœud cible dans ce graphe. Ce chemin est déterminé par un tri topologique car ce tri permet d'explorer chaque nœud du graphe acyclique orienté après avoir exploré tous ses prédécesseurs. La propagation des changements d'état dans ce graphe se fait donc en suivant un des ordres induit par ce tri.

Lors d'une réaction en chaîne, la propagation des changements se fait en alternant d'une structure à une autre. Par exemple, selon le contexte, la propagation peut se faire suivant la structure d'arbre, puis suivant la structure de graphe et revenir à la structure d'arbre. Par exemple si l'instruction `f.press ->! red` était présente dans le programme de la figure 2.4a et que le PROCESSUS f.press, modélisant l'action d'appui sur la fenêtre `f`, était activé alors typiquement une propagation se ferait en suivant la structure de graphe pour désactiver le PROCESSUS `red` puis une autre propagation se ferait en suivant les structures d'arbre pour vérifier si le PROCESSUS `red` a des enfants à désactiver. En plus de respecter les contraintes de propagation imposées par la topologie du programme, la propagation des changements respecte les propriétés de la section 2.1.2.

Gestion de la mémoire

Le modèle mémoire de SMALA est dépendant du PROCESSUS PROPERTY du langage. Ce PROCESSUS possède un unique espace mémoire dans lequel il est possible de stocker une valeur. Chaque PROPERTY a un type et c'est celui-ci qui définit la taille de l'espace mémoire et le type de valeur que l'on peut y stocker. De ce fait, une PROPERTY peut être de type INT, DOUBLE, STRING ou BOOL. Dans un programme SMALA, l'identifiant de la PROPERTY est l'unique pointeur vers cet espace et l'aliasing n'est pas permis. Deux PROCESSUS permettent d'interagir avec ces espaces mémoires :

1. Le processus ASSIGNMENT noté $p_1 =: p_2$ permet, lors de son activation, de copier la valeur de la PROPERTY p_1 dans la PROPERTY p_2. La mise à jour d'une PROPERTY est considérée comme un changement d'état qui peut être propagé via des BINDINGS.
2. Le processus CONNECTOR noté $p_1 => p_2$ permet de copier la valeur de la PROPERTY p_1 dans la PROPERTY p_2 lorsque la valeur de p_1 a été mise à jour.

Création dynamique de processus

Le dernier aspect de la dynamicité concerne la création dynamique de processus. SMALA permet à l'aide des opérateurs `addChildrenTo`, `move` et `remove` (figure 2.3) respectivement d'ajouter, de déplacer et de supprimer un PROCESSUS lors de l'exécution d'un programme et donc de changer sa topologie dynamiquement.

Dans la prochaine section nous allons voir à travers un exemple comment il est possible de mettre ces concepts en pratique.

2.2.4 Exemple

Dans cette section nous allons décrire le programme de la figure 2.5 illustré par la figure 2.6. Concrètement, ce programme a le comportement suivant. À l'initialisation, le programme ouvre une fenêtre ayant pour titre `"Circle"` et de dimension 400×400. Par défaut le premier état de la machine à

```
1  _main_
2  Component root {
3    Frame f ("Circle", 0, 0, 400, 400)
4    Circle c (200, 200, 0)
5    FSM fsm {
6      State resizeByWidth {
7        Double radius (50)
8        radius =: c.r
9        (f.width/10) => c.r
10     }
11     State resizeByHeight {
12       Double radius (50)
13       radius =: c.r
14       (f.height/10) => c.r
15     }
16     resizeByWidth -> resizeByHeight (f.press)
17     resizeByHeight -> resizeByWidth (f.press)
18
19   }
20   Exit ex (0,0)
21   f.close -> ex
22 }
```

FIGURE 2.5 – Exemple d'un programme SMALA.

état est activé et donc un cercle noir (couleur par défaut) de rayon 50 est affiché à la position $(200, 200)$ de la fenêtre. Dans cet état, si la largeur de la fenêtre est re-dimensionnée alors le rayon du cercle est égal à la largeur de la fenêtre divisée par 10. Si le bouton de la souris est appuyé sur le canevas de la fenêtre alors le programme transite vers le deuxième état, le rayon du cercle sera réinitialisé à 50 et cette fois le cercle pourra être re-dimensionnée en rapport avec la hauteur de la fenêtre.

Comme indiqué dans section 2.2.2, le programme décrit dans la figure 2.5 consiste en un `Component` nommé `root` contenant une liste de PROCESSUS enfant. Le premier PROCESSUS déclaré est un PROCESSUS fenêtre (`Frame`) nommé `f` de taille 400×400, positionné aux coordonnées $(0, 0)$ de l'écran et qui a pour titre `"Circle"`. Puis, un processus cercle `Circle c` dont le centre est positionné aux coordonnées $(200, 200)$ de la fenêtre et dont le rayon initialisé à 0 est défini. En suite, une machine à état notée `FSM fsm` est déclarée. Cette machine à état possède deux états `State resizeByWidth` et `State resizeByHeight` et deux transitions. L'état `State resizeByWidth` déclare :

l7 : Une propriété Double `radius` qui a pour valeur initiale 50.

l8 : Une assignation, de la valeur de la propriété `radius` à la valeur de la propriété `c.r` correspondant au rayon du cercle.

l9 : Un CONNECTOR qui relie la propriété double `f.width` à la propriété `c.r` du cercle. Ainsi, par le biais de cet opérateur, la valeur de `c.r` est mise à jour à chaque fois que la largeur de la fenêtre, `f.width`, est re-dimensionnée. Il existe un autre CONNECTOR dénoté =:> qui est un variant du CONNECTOR dénoté =>. Le CONNECTOR =:> diffère en permettant de copier la valeur de sa source dans sa PROPERTY cible lorsque celle-ci est activée. Le CONNECTOR => n'agit pas de cette manière lors de son activation.

L'état `State resizeByHeight` est similaire à l'état `resizeByWidth` à l'exception de sa dernière instruction `f.height =:> c.r` qui re-dimensionne `c.r` à chaque fois que la hauteur de la fenêtre est re-dimensionnée.

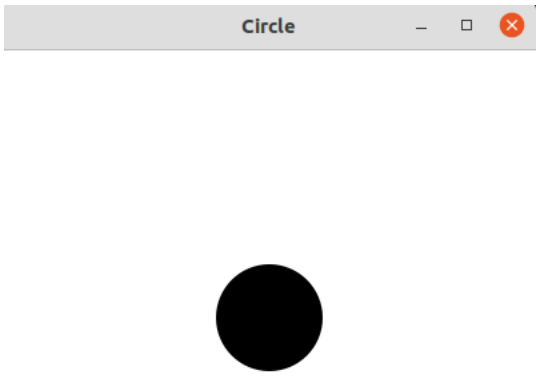


FIGURE 2.6 – Exécution du programme figure 2.5.

Concernant les transitions de la machine à état :

l16 : La transition `resizeByWidth -> resizeByHeight (f.press)` permet de faire une transition de l'état resizeByWidth à l'état resizeByHeight lorsque le bouton de la souris est appuyé sur la fenêtre `f`.

l17 : La transition `resizeByHeight -> resizeByWidth (f.press)` est la transition inverse à la précédente.

Pour terminer, **l20** déclare le PROCESSUS `Exit` nommé `ex` et permettant de mettre fin à l'exécution du programme lorsqu'il est activé. Ce PROCESSUS a deux paramètres le premier étant la valeur de retour du programme, ici `0`, et le deuxième permettant de préciser si le PROCESSUS doit être activé ou pas lorsque son parent est activé. **l21** déclare un BINDING permettant d'activer `ex` lorsque le bouton pour fermer la fenêtre, dénoté par `f.close`, est appuyé.

La topologie de ce programme est représentée au travers de la figure 2.7 pour l'arbre des composants, de la figure 2.8 pour le graphe de lien et de la figure 2.9 pour le graphe de scène.

La prochaine section présente la théorie des bigraphes, le formalisme choisi pour définir la sémantique de l'UIDL que nous définissons dans cette thèse.

2.3 Les bigraphes

La théorie des bigraphes, formellement définie par Robin Milner [31], permet de modéliser des systèmes qui évoluent en espace et en temps. Les bigraphes consistent en un ensemble d'entités (les nœuds) partagé par deux structures de graphes orthogonales. Le graphe de places, qui est une forêt, représente l'aspect spatial du système en termes d'imbrication et le graphe de liens, qui est un hypergraphe, représente les interactions présentes dans le système par des hyper-arêtes. Afin de faire évoluer un bigraphe, la théorie nous permet de définir des règles de réaction permettant de modifier une certaine partie de celui-ci. Nous présentons plus en détails dans la section 2.3.1 la structure des bigraphes et dans la section 2.3.2 les règles de réaction permettant de les modifier. Dans la section 2.3.2

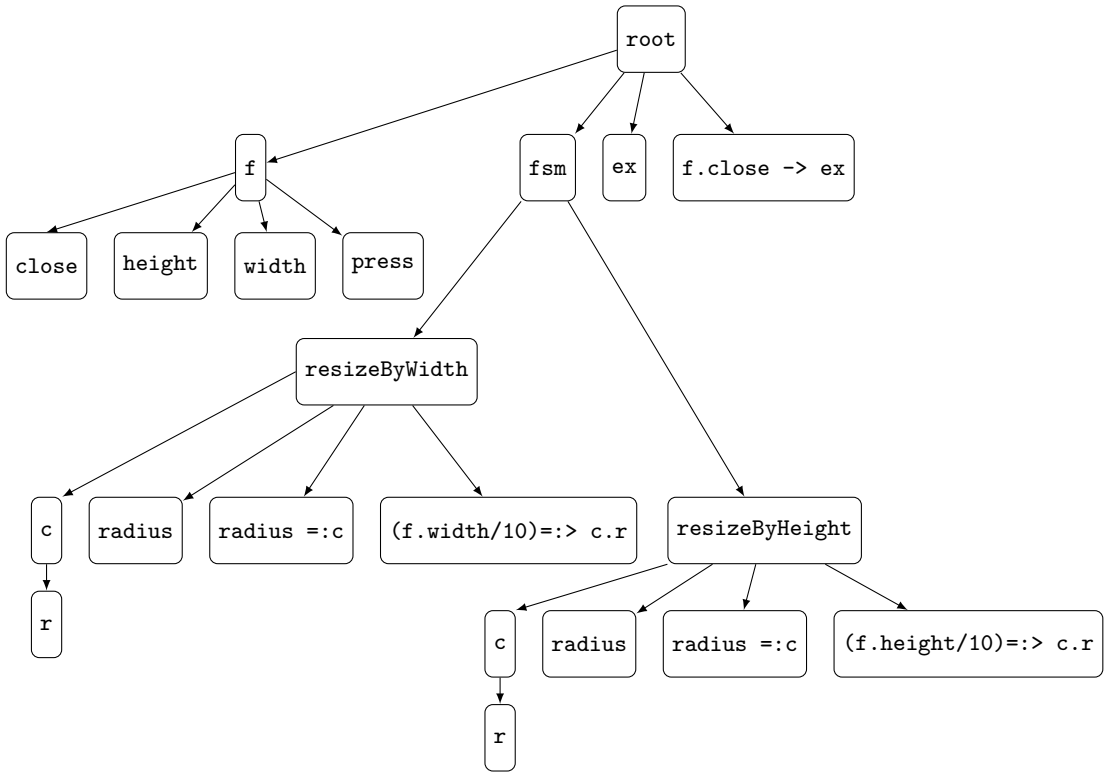


FIGURE 2.7 – Arbre des composants du programme figure 2.5.

nous parlons de l'outil BigraphER permettant de simuler un système représenté par un bigraphe et des règles de réaction.

2.3.1 Structure des bigraphes

Un bigraphe, comme illustré par l'exemple de la figure 2.10a, est un ensemble d'entités partagées par deux structures orthogonales appelé graphe de places, illustré par l'exemple de la figure 2.10b et graphe de liens, illustré par l'exemple de la figure 2.10c. Chaque bigraphe a une signature basique qui correspond à un ensemble d'éléments appelés les contrôles que l'on associe à chacune des entités du bigraphe et permettant de définir leur arité i.e. le nombre d'arêtes pouvant être connectées sur l'entité. Par exemple dans la figure 2.10a, les contrôles A d'arité deux, B d'arité un et C d'arité un, forment la signature du bigraphe illustré. Nous utilisons une version des bigraphes qui est non standard [2] et qui offre la possibilité d'utiliser des contrôles paramétriques e.g. $\mathsf{A}(x)$, généralisables sur l'ensemble de variables x qui peut être typé (int,float ou string) e.g. si la valeur x du contrôle A est un int alors x peut prendre la valeur de n'importe quel entier. Les entités composant les bigraphes peuvent être imbriquées e.g. l'entité de contrôle A qui est imbriquée dans l'entité de contrôle B, et liés aux travers des hyper-arêtes, e.g. l'entité A et l'entité C. Une entité est dite atomique si celle-ci ne peut pas contenir d'autres entités.

Un graphe de places permet de représenter l'agencement spatial des entités à travers l'imbrication et la juxtaposition. Par exemple dans la figure 2.10b, l'entité A est contenue dans l'entité B et l'entité C est placée à coté de l'entité B. Des constructions spéciales, telles que les sites et les régions, existent et permettent de composer les graphes de places et en conséquence les bigraphes. Les sites, représentés

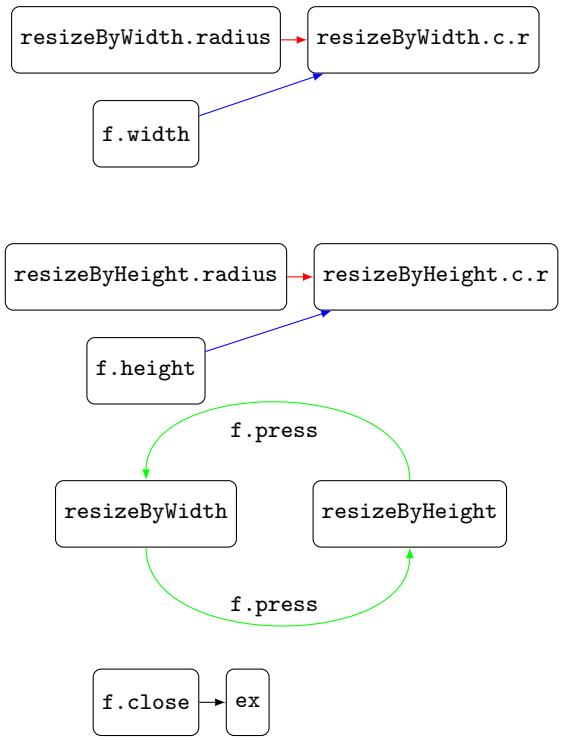


FIGURE 2.8 – Graphe de lien du programme figure 2.5. Les flèches noires, rouges et bleues correspondent respectivement aux BINDINGS, ASSIGNMENTS et CONNECTORS. Les flèches vertes étiquetées correspondent aux transitions des machines à états avec l'action les activant.

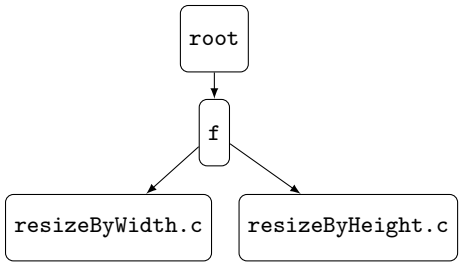
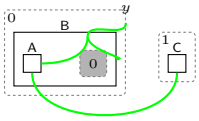
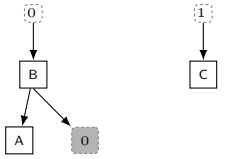


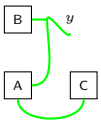
FIGURE 2.9 – Graphe de scène du programme figure 2.5.



(a) Bigraphe.



(b) Graphe de places.



(c) Graphe de liens.

FIGURE 2.10 – Représentation d'un bigraphe avec son graphe de liens et son graphe de places.

par des rectangle gris, abstraient une partie du graphe de places et les régions, représenté par des rectangles en pointillés, sont les conteneurs de base dans un graphe de places. Il est possible de composer les graphes de places, et donc les bigraphes, en plaçant une région dans un site.

Un graphe de liens représente les interactions des entités via les arêtes qui sont représentées par des liens de couleur verts. Par exemple, dans la figure 2.10c, l'entité A est liée à l'entité C. Comme pour les graphes de places, il est possible de composer les graphes de liens mais cette-fois ci en utilisant les noms des arêtes. Il existe deux types d'arêtes dans un graphe de liens, les arêtes fermées e.g. l'arête reliant A et C et les arêtes ouvertes possédant un nom appelé NOM EXTERNE, e.g. l'arête reliant A et B qui a pour nom externe y, ou NOM INTERNE qui ne sont pas utilisés dans cette thèse. Ce sont les noms externes et internes des arêtes qui permettent la composition des graphes de liens par égalité des NOMS. Ainsi il est possible de composer un graphe de liens ayant une arête ouverte de NOM INTERNE y avec un autre graphe de liens ayant une arête ouverte de NOM EXTERNE y. Il est important de comprendre qu'un NOM ne spécifie pas un lien i.e. dans la figure 2.10c il n'y a pas d'arête appelée y et on aurait pu substituer le nom y par z sans que la sémantique du graphe de liens ait changé.

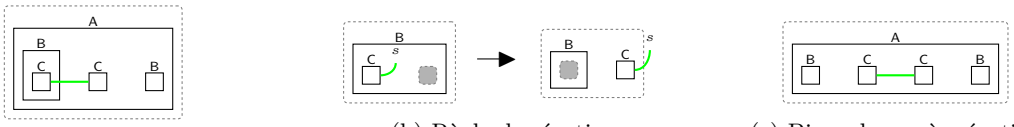
Nous terminons cette section par une présentation superficielle des sortings. Les sortings permettent de définir des lois de composition, sur les graphes de places et les graphes de liens, sous conditions qu'elles soient préservées par composition et juxtaposition des bigraphes. Les sortings sont généralement utilisés pour filtrer des compostions non désirées dans une instance de bigraphes définie.

Dans la prochaine section nous présentons les systèmes réactifs bigraphiques qui vont nous aider à formaliser la sémantique des UIDLs.

2.3.2 Système réactif bigraphique (BRS)

Le côté dynamique de la théorie est défini à travers les systèmes réactifs bigraphiques (BRS) qui correspondent à un bigraphe assorti d'un ensemble de règle de réaction.

Une règle de réaction est composée de deux parties comme l'illustre l'exemple de la figure 2.11b. La partie gauche d'une règle de réaction, nommée redex, représente la partie de bigraphe qui va être sujette à modification. La partie droite, nommée reactum, indique comment le bigraphe correspondant à la partie gauche de la règle est modifié. Ainsi, une règle est applicable sur un bigraphe si le redex de celle-ci correspond à une partie du bigraphe concerné. Cette partie est ensuite remplacée par le reactum de la règle. La figure 2.11 illustre comment une règle de réaction est applicable sur un bigraphe ; nous appliquons la règle représentée par la figure 2.11b sur le bigraphe représenté par la figure 2.11a, le résultat de la transformation est représenté par la figure 2.11c. Cette règle de réaction a pour but de



(a) Bigraphe initial. (b) Règle de réaction. (c) Bigraphe après réaction.

FIGURE 2.11 – Exemple de réaction sur un bigraphe.

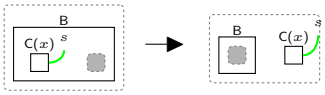


FIGURE 2.12 – Règle paramétrique.

déplacer toutes les entités C incluses dans une entité B à l'extérieur de celle-ci. Une règle de réaction permet de déplacer, supprimer, dupliquer et permuter des entités, des connexions ou des sites formant le bigraphe cible et même d'ajouter de nouvelles entités. Les règles de réactions ne font apparaître que les entités nécessaires à leur redex et reactum, le reste est soit omis car ces entités se trouvent en contexte soit abstrait par des sites.

Afin de gérer des entités sur lesquelles des contrôles paramétriques seraient associés, il est possible de définir des règles paramétriques comme présenté dans la figure 2.12

Un BRS a pour but de générer un système à transitions représentant l'exécution d'un système représenté par un bigraphe. Le bigraphe donné à la définition du BRS correspond à l'état initial du système. Chaque nouvel état du système est le résultat d'une application de règle de réaction. Pour savoir quelle règle de réaction appliquer sur un état, il est possible de définir un ordre partiel sur celles-ci afin de donner une indication sur leurs priorités. Ainsi, les règles applicables sur un état à un instant t sont les règles les plus prioritaires (selon l'ordre partiel défini) dont le redex correspond à une partie du bigraphe représentant l'état.

2.3.3 L'outil Bigraph Evaluator & Rewriting (BigraphER)

L'outil BigraphER [44] est un outil open-source qui implémente les systèmes réactifs bigraphiques. Cet outil permet de simuler les BRS, d'explorer les états d'un système à transitions et de faire de la vérification de propriétés. La syntaxe mise à disposition par BigraphER pour définir des bigraphes correspond à la représentation algébrique des bigraphes définie par Robin Milner et présentée dans le tableau 2.1. La figure 2.13 correspond à la définition via BigraphER du bigraphe représenté dans la figure 2.11a et la figure 2.14 à celle de la règle de réaction représentée dans la figure 2.11b.

La définition d'un BRS, ayant comme bigraphe initial celui de la figure 2.11a, est illustré par la figure 2.15a. Les trois premières lignes du code définissent les contrôles indispensables au BRS, i.e. les contrôles A et B d'arité zéro et le contrôle atomique C d'arité une. La ligne 6 définit un bigraphe qui correspond à une entité de contrôle A contenant une entité de contrôle B qui contient trois entités de contrôle C. Ensuite à la ligne 8, la règle de réaction correspondant à la figure 2.11b est définie et à la ligne 10 une règle de réaction, permettant de supprimer une entité B qui ne contient plus aucune entité, est définie. La définition du BRS vient à la ligne 12 avec le bigraphe initial défini à la ligne 13

```
1 big fig313 = A.(B.C{x} | C{x} | B.1);
```

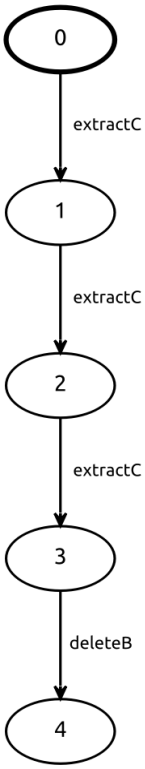
FIGURE 2.13 – Code bigraphER représentant le bigraph de la figure 2.11a.

```
react extractC = B.(C{x} | id)  --> B.id | C{x} ;
```

FIGURE 2.14 – Code bigraphER représentant la règle de réaction de la figure 2.11b.

```
ctrl A = 0;
ctrl B = 0;
atomic ctrl C = 1;


big test = A.B.(C{x} | C{x} | C{x
    }) ;

react extractC = B.(C{x} | id)
    --> B.id | C{x} ;

react deleteB = B.1 --> 1 ;

begin brs
init test;
rules = [
          {extractC,deleteB}
        ];
end
```

(a) Code bigraphER représentant un BRS.



(b) Simulation d'un BRS via bigraphER.

FIGURE 2.15 – Définition d'un BRS et sa simulation.

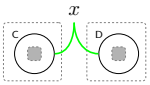
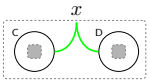
Composantt/Operation	Forme algébrique	Forme diagrammatique
Entity of arity 1	K_a	
Name closure	$/a\ \mathsf{K}_a$	
Site	id	
Region	1	
Nesting	A.B.id	
Parallel product	C_x.id ‖ D_x.id	
Merge product	C_x.id | D_x.id	

TABLE 2.1 – Équivalence entre la notation en diagramme et la notation algébrique des bigraphes.

et les règles que nous allons utiliser définies à la ligne 15. Ces règles sont contenues dans un ensemble qui est contenu dans une liste. Ces structures de données permettent de définir un ordre partiel sur les règles de réaction et en conséquence les règles d'un ensemble sont prioritaires aux règles contenues dans les ensembles suivants enfin les règles d'un même ensemble ne sont pas ordonnées.

L'exécution de ce BRS est illustrée par la figure 2.15b qui correspond à son système à transition généré par BigraphER. Dans les trois premières transitions la règle `extractC` est appliquée permettant ainsi d'extraire toutes les entités de contrôle C de l'entité B. L'exécution se termine par l'application de la règle `deleteB` qui supprime l'entité B. Si plusieurs exécutions étaient possibles, dû à la possibilité d'appliquer différentes règles ayant la même priorité sur un état, alors l'outil aurait simulé ces différentes exécutions.

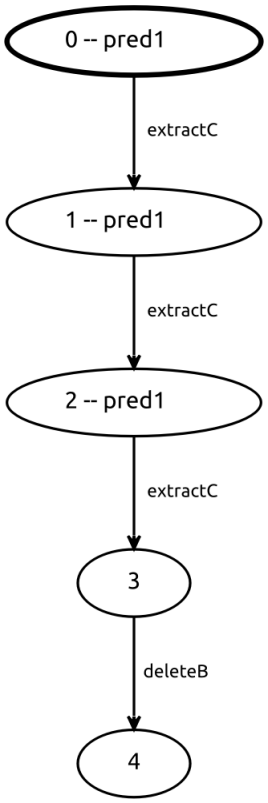
Les bigraphes permettent de modéliser les aspects spatiaux d'un système ainsi que les dépendances ou les connexions entre les composants qui existent dans celui-ci. Les modélisations de systèmes sont motivées par le fait de vouloir mieux le comprendre et de vérifier leur bon fonctionnement. Pour cela BigraphER permet de définir des prédicats et de vérifier qu'ils sont satisfaits sur un BRS. Ces prédicats consistent en un bigraphe et leur satisfaisabilité consiste à vérifier leur présence dans les états générés par le BRS. Si un état satisfait le prédicat (i.e. contient le bigraphe représentant le prédicat) alors celui-ci est étiqueté par son nom. Les figures 2.16a et 2.16b illustrent la définition et l'utilisation de prédicats. Les lignes 12 et 19 de la première figure, montrent respectivement comment définir et utiliser un prédicat et la deuxième figure, illustre le système à transition correspondant à l'exécution du BRS avec les états satisfaisant le prédicat étiquetés. Dans cet exemple nous vérifions l'existence d'une entité B contenant une entité C et seuls les trois premiers états satisfont ce prédicat. Dans la prochaine section, nous montrons comment ces étiquettes sont utilisées pour vérifier des propriétés CTL sur les BRS.

2.3.4 Vérification de propriétés CTL sur les systèmes de transitions par PRISM

PRISM est un vérificateur de modèle probabiliste et un outil de formalisation et d'analyse de systèmes qui présentent des comportements probabilistes. PRISM permet de modéliser des systèmes

```
ctrl A = 0;
ctrl B = 0;
atomic ctrl C = 1;


big big_init = A.B.(C{x} | C{x} |
    C{x}) ;

react extractC = B.(C{x} | id)
    --> B.id | C{x} ;

react deleteB = B.1 --> 1 ;

big pred1 = B.(C{x} | id);

begin brs
init big_init;
rules = [
          {extractC,deleteB}
        ];
preds = {pred1};
end
```

(a) Définition d'un prédicat avec bigraphER.



(b) Simulation du BRS contenant une vérification de prédicat via bigraphER.

FIGURE 2.16 – Définition d'un BRS avec prédicat et sa simulation.

sous forme d'automate et de vérifier des propriétés comme, "Quelle est la probabilité que le système s'arrête dans 4 heures à cause d'une erreur ?" ou "Quel est le pire temps d'exécution attendu pour que l'algorithme termine ?". Pour cela, PRISM propose un langage permettant de décrire des modèles et un autre permettant de spécifier des propriétés. Ce dernier langage, inclut des logiques temporelles telles que PCTL, PCTL*, CSL et LTL. Pour cette thèse nous utilisons PRISM, qui est préconisé par les créateurs de BigraphER, pour vérifier des propriétés sur des machines à états correspondant aux exécutions des BRS. Dans notre cas, nous utilisons la logique CTL, dont un sous-ensemble est couvert par PRISM, pour spécifier nos propriétés.

Pour rappel la logique CTL, de l'anglais Computation Tree Logic, est une logique temporelle qui permet de vérifier des formules sur des modèles correspondant à des graphes dirigé. Chaque chemin du modèle, représente un futur possible qui peut se réaliser. En conséquence, pour vérifier une formule CTL sur un modèle, il est nécessaire de préciser un nœud du graphe, qui correspond à un point de départ, à partir du quel nous allons vérifier la formule. Afin de raisonner sur cette structure, CTL met à disposition les opérateurs logiques usuels ($\neg, \wedge, \Rightarrow$ et $\vee$) et les opérateurs temporels suivants :

- **A** ψ (All), qui permet de vérifier si la formule ψ est valide sur tous les chemins partant du nœud du graphe préalablement fixé comme point de départ ;
- **E** ψ (Exists), qui permet de vérifier si la formule ψ est valide sur au moins un chemin partant du nœud du graphe sélectionné comme point de départ ;
- **X** ψ (Next), qui permet de vérifier si la formule ψ est valide au nœud succédant le nœud de départ ;
- **G** ψ (Globally), qui permet de vérifier si la formule ψ est valide sur l'ensemble de la sous-séquence de nœuds partant du nœud de départ ;
- **F** ψ (Eventually), qui permet de vérifier si la formule ψ est valide sur au moins un nœud du chemin commençant au nœud sélectionné comme point de départ ;
- ϕ **U** ψ (Until), qui permet de vérifier si la formule ϕ est valide à partir du nœud de départ jusqu'à un nœud vérifiant la formule ψ.

Dans notre contexte, les formules CTL sont vérifiables seulement sur des systèmes de transitions étiquetés par des prédicats BigraphER car ces étiquettes correspondent aux variables atomiques de la logique. La première étape à effectuer afin de vérifier des formules CTL sur un système de transitions est de générer le modèle PRISM correspondant à celui-ci ainsi que ses étiquettes. BigraphER permet de générer un fichier, d'extension **.tra**, contenant une matrice à transition représentant le système produit et un fichier, d'extension **.csl**, contenant les étiquettes ainsi que les états auxquels celles-ci sont associées. La figure 2.17a illustre une telle matrice pour la machine à état de la figure 2.16b et la figure 2.17b présente ses états étiquetés.

Il est ensuite possible de compléter le fichier **.csl** avec les formules CTL que nous voulons vérifier sur ce système de transitions comme présenté dans la figure 2.18. La matrice à transition ainsi que les spécifications des formules CTL sont finalement données à PRISM pour effectuer la vérification.

2.4 Conclusion

Dans ce chapitre nous avons présenté les concepts essentiels à la compréhension de cette thèse.

La section 2.1 présente les langages de description d'interface utilisateur utilisés pour faciliter le développement d'interfaces graphiques. Cette section présente les deux principales caractéristiques définissant la plupart des UIDLs qui sont la capacité à décrire 1) un graphe de scène et 2) des interactions. Nous avons aussi présenté les propriétés que ces deux caractéristiques engendrent.

La section 2.2 présente l'UIDL Smala développé par l'équipe informatique interactive de l'ENAC. Cet UIDL est conçu pour développer des interfaces utilisateur dans l'aviation et se démarque par ses

```
5 4
0 1
1 2
2 3
3 4
```

(a) Matrice qui correspond au système de transitions de la figure 2.16b. La première ligne indique le nombre d'états et le nombre de transitions composant le système. Les états de la première colonne sont les points de départ des transitions. Les états de la deuxième colonne sont les points d'arrivée des transitions. Ainsi la deuxième ligne de la matrice est lue, il existe une transition de l'état 0 à l'état 1.

```
label "pred1" = x = 2 | x = 1 | x
    = 0;
```

(b) Étiquettes de la machine à état de la figure 2.16b. Ici, les états 0,1 et 2 sont étiquetés par "pred1".

```
label "pred1" = x = 2 | x = 1 | x = 0;
E [(F !("pred1"))]
```

FIGURE 2.18 – Spécification de la formule CTL à vérifier. La formule spécifie qu'il existe un chemin du système sur lequel il y a finalement un état qui ne vérifie pas "pred1".

PROCESSUS qui sont l'abstraction de base du langage facilitant la prise en main de celui-ci, par sa syntaxe permettant de spécifier efficacement les interactions d'une interface et par la prise en charge de la conception itérative.

Enfin, la section 2.3 présente la théorie des bigraphes et l'outil BigraphER qui implémente celle-ci. Nous avons décrit la structure des bigraphes et présenté comment, avec les règles de réaction, il est possible de décrire l'aspect dynamique d'un système. Nous avons aussi présenté brièvement l'outil BigraphER permettant de simuler les BRS et de vérifier des prédicats sur ces derniers.

Dans le prochain chapitre nous discutons des motivations qui nous ont poussées à utiliser la théorie des bigraphes pour représenter la sémantique des UIDLs.

Chapitre 3

Représentation des connaissances

Différentes formalisations d'une sémantique de langage défini avec des cadres mathématiques différents peuvent être équivalentes mais divergentes en terme de représentation des connaissances [49, 21, 23]. Nous souhaitons réfléchir sur cette question et montrer que cela nécessite une attention particulière dans le cas de langages tels que les UIDLs présentant des caractéristiques spatiales et interactives. L'objectif de ce chapitre, est de montrer que différentes représentations d'une sémantique ne conduisent pas à la même compréhension épistémique d'un langage et donc par conséquence que le choix du cadre mathématique pour leur définition a un certain impact sur le sujet. Afin de montrer la différence épistémique entre deux représentations équivalentes d'une sémantique, nous décidons d'étudier deux représentations de la sémantique d'un sous-ensemble du langage Smala, présenté dans la section 3.1, qui couvre seulement la propagation d'activation des PROPERTIES. L'équivalence entre les deux sémantiques étudiées est admise. La première sémantique, donnée dans la section 3.2, consiste en une sémantique opérationnelle exprimée via la logique de premier ordre et des règles d'inférence. Nous l'avons mentionné dans les chapitres précédents, nous comptons utiliser la théorie des bigraphes comme cadre mathématique pour présenter la sémantique de l'UIDL que nous voulons définir. C'est d'ailleurs les bigraphes qui ont été choisis comme cadre mathématique de la deuxième sémantique que nous allons étudier dans ce chapitre. Avant de présenter cette sémantique, nous comparons différentes algèbres de processus, dans la section 3.3, afin de comprendre pourquoi les bigraphes sont pertinents dans notre contexte. Nous présentons enfin dans la section 3.4 la deuxième sémantique reposant sur la théorie des bigraphes.

3.1 Sous-ensemble de Smala

Dans ce chapitre, nous travaillons avec un sous-ensemble de Smala, illustré dans le tableau 3.1, qui se focalise uniquement sur l'aspect réactif du langage, ce qui a pour avantage de rendre l'analyse plus simple à comprendre. Ce sous-ensemble couvre seulement les propagations d'activations du langage et ne couvre pas son aspect topologique. Le sous-ensemble est composé des CONNECTORS, des BINDINGS et des PROPERTIES. Les PROPERTIES sont utilisées pour abstraire les interactions utilisateur. Ainsi, une PROPERTY qui abstrait un clic souris s'active à chaque fois que l'évènement se produit. Les CONNECTORS et les BINDINGS ont la même sémantique que dans le langage Smala i.e. le CONNECTOR copie la valeur de sa PROPERTY source dans sa PROPERTY cible tout en l'activant et le BINDING se charge d'activer sa PROPERTY cible lorsque sa PROPERTY source est activée.

Un programme de ce sous-ensemble consiste en la définition de PROPERTIES, de BINDINGS et de CONNECTORS. La figure 3.1 donne un exemple de programme que le sous-ensemble permet de

Instructions du sous-ensemble
BINDING _ → _
CONNECTOR _ ⇒ _
PROPERTY Int x (2)

TABLE 3.1 – Instructions du sous-ensemble de SMALA. Le tiret du bas remplace des expressions (e.g. opérations arithmétiques, variables) des PROCESSUS.

```
1    Int p1 (0)
2    Int p2 (0)
3    Int p3 (0)
4    p1 => p2
5    p2 => p3
```

FIGURE 3.1 – Exemple issu du sous ensemble.

concevoir.

Ce programme consiste en la définition de trois `Int` PROPERTIES nommées `p1`, `p2` et `p3` toutes initialisées à `0`. Ensuite, deux CONNECTORS y sont définis. Le premier CONNECTOR met en relation `p1` et `p2`. Le deuxième, met en relation `p2` et `p3`. Ainsi, si la valeur de `p1` est modifiée alors celle-ci va être propagée, par les deux CONNECTORS déclarés, à `p2` et à `p3`.

Dans la prochaine section, nous donnons une sémantique opérationnelle pour ce sous-ensemble de SMALA et discutons sur sa représentation des connaissances.

3.2 Une sémantique opérationnelle

La sémantique opérationnelle donnée pour le sous-ensemble décrit dans la section 3.1 se décompose en trois phases. SMALA étant un langage déclaratif, la première phase, présentée dans la section 3.2.1, consiste en la construction du modèle représentant le programme, incluant l'initialisation des PROPERTIES et la création d'interaction entre celles-ci. Enfin, la phase de propagation et d'exécution, décrite dans la section 3.2.2, définit comment les activations des PROCESSUS se propagent dans le programme et comment les PROCESSUS du sous-ensemble sont exécutés.

3.2.1 Phase d'initialisation

Les déclarations pouvant être effectuées à travers le sous-ensemble de SMALA sélectionné, permettent de créer des PROPERTIES en mémoire et des dépendances entre elles en utilisant les opérateurs $\rightarrow$ et $\Rightarrow$.Ainsi pour représenter un programme de ce sous-ensemble, nous créons un environnement E, qui associe à chaque PROPERTY sa valeur, et une fonction S, qui a pour domaine l'ensemble des PROPERTIES définies dans le programme et qui associe à chacune d'elles l'ensemble des instructions qui peut être exécutées lors de leur activation. Les règles d'inférence présentées ci-dessous permettent la création de l'environnement E et de la fonction S

La règle **Init** permet de construire l'environnement E et S en parcourant chaque instruction du programme. Un programme est représenté par une liste d'instruction que nous notons $i :: Ins$ avec i la première instruction du programme et Ins la liste des instructions restantes. L'hypothèse $E, S \vdash i \rightsquigarrow E', S'$ dénote un changement d'état qui consiste en la modification de E et S en conséquence de l'initialisation de l'instruction i et $E', S' \vdash Ins \rightsquigarrow E'', S''$ dénote un changement sur E et S en conséquence de l'initialisation de la liste d'instructions Ins. L'idée générale de cette règle, est d'itérer

la règle **Init** sur la liste d'instruction afin de modifier E et S en fonction des instructions pour finalement avoir un environnement E'' représentant la mémoire initiale du programme et une fonction S'' permettant d'exécuter les instructions nécessaires dues à l'activation de leur PROPERTIES associée. Pour chaque instruction parcourue la règle d'initialisation correspondante à celle-ci est appliquée i.e. **InitBinding**, **InitConnector**, **InitAssignment** et **InitProperty**.

$$\textbf{Init}\ \frac{E, S\ \vdash i \rightsquigarrow E', S' \qquad E', S'\ \vdash Ins \rightsquigarrow E'', S''}{E, S\ \vdash i :: Ins \rightsquigarrow E'', S''}$$

La règle **InitProperty** permet d'initialiser une PROPERTY. Ainsi, si l'expression e devant initialiser la PROPERTY s'évalue en une valeur v, dénotée par $E \vdash e \Downarrow v$, alors l'environnement E est modifié par l'ajout de la PROPERTY x associé à sa valeur v.

$$\textbf{InitProperty}\ \frac{E\ \vdash e\ \Downarrow\ v}{E, S\ \vdash Int\ x(e)\ \rightsquigarrow E[x/v], S'}$$

Notre sous-ensemble étant restreint à l'utilisation des entiers et des PROPERTIES seules deux règles d'évaluation sont définies.La règle **EvalInt** qui est triviale et la règle **EvalVar** qui évalue une PROPERTY à sa valeur actuelle dans l'environnement E.

$$\textbf{EvalInt}\ \frac{}{E\ \vdash i \Downarrow i}$$

$$\textbf{EvalVar}\ \frac{}{E\ \vdash x \Downarrow E(x)}$$

La règle **InitBinding** initialise le BINDING en ajoutant à la fonction S le couple $(x1, x1 \rightarrow x2)$ ce qui permettra à la fonction S de retourner un ensemble d'instruction contenant le BINDING $x1 \rightarrow x2$ lorsque la PROPERTY $x1$ sera activée.

$$\textbf{InitBinding}\ \frac{}{E, S\ \vdash x_1 \rightarrow x_2 \rightsquigarrow E, \{(x_1, x_1 \rightarrow x_2)\} \cup S}$$

La règle **InitConnector** est similaire à la règle **InitBinding** à l'exception qu'elle rajoute à la fonction S l'ensemble des couples $(y, e \rightarrow x2)$ avec y correspondant aux PROPERTIES libres de e.

$$\textbf{InitConnector}\ \frac{}{E, S\ \vdash e \Rightarrow x \rightsquigarrow E, \{(y, e \Rightarrow x) \mid y \in FV(e)\} \cup S}$$

La règle **EndInit** permet de terminer la phase d'initialisation lorsque il n'y a plus d'instruction à traiter en renvoyant l'environnement E et la fonction S construits après avoir initialisé toutes les instructions du programme.

$$\textbf{EndInit}\ \frac{}{E, S\ \vdash [] \rightsquigarrow \text{E,S}}$$

Dans la prochaine section nous présentons comment les activations des PROPERTIES se propagent via les BINDINGS et les CONNECTORS ainsi que le mécanisme d'exécution des PROCESSUS.

3.2.2 Phase de propagation et d'exécution

Le but de la phase de propagation et d'exécution est de définir le mécanisme de propagation d'activation et d'exécution des PROCESSUS lorsqu'une interaction a lieu. Dans notre sous-ensemble, une interaction, e.g. un clic de souris, est abstraite par une PROPERTY. Lorsqu'une interaction avec le système exécutant le programme a lieu, alors la PROPERTY l'abstrayant voit sa valeur modifiée et son activation va être propagée à ses dépendances et ainsi enclencher les instructions les contenant et pouvant correspondre à des instructions constituées de CONNECTOR ou bien de BINDING.

La règle **propagEv** définit ce mécanisme. Dans cette règle, si un évènement change la valeur de la PROPERTY e par v, alors un appel à la fonction S avec pour paramètre e est fait afin d'exécuter toutes les instructions exécutables sur l'activation de e. L'ensemble d'instructions renvoyé par S est ensuite traité par les autres règles d'inférence, ce qui donnera un nouvel environnement E' correspondant à l'environnement $E[e/v]$ ayant été modifié en fonction des instructions exécutées.

$$\textbf{PropagEv}\ \frac{E[e/v], S \ \vdash\ [S(e)] \rightsquigarrow E', S}{E, S \ \vdash\ Event(e, v) \rightsquigarrow E', S}$$

Les instructions contenues dans la liste d'ensembles, qui sont soit des instructions contenant des BINDINGS soit des CONNECTORS sont traitées avec les règles d'inférence suivantes.

La règle **PropagBinding** permet de traiter les instructions contenant des BINDINGS. Les BINDINGS ont pour unique fonction de propager l'activation de leur PROPERTY source à leur PROPERTY cible. Dans cette règle, une liste d'ensembles d'instructions ayant pour première instruction un BINDING et s'exécutant dans un environnement E donne un nouvel environnement E'', si l'exécution de la liste d'ensemble d'instructions sans sa première instruction concaténé à la liste d'ensemble d'instructions $[S(x_2)]$, qui correspond aux dépendances de x_2 à activer, donne l'environnement E''. L'activation de la cible x_2 est représentée par la concaténation, représenté par l'opérateur $++$, de la liste $[S(x_2)]$ à la liste $(ins :: I)$.

$$\textbf{PropagBinding}\ \frac{E, S \ \vdash\ (ins :: I) + +[S(x_2)] \rightsquigarrow E'', S}{E, S \ \vdash\ (\{x_1 \to x_2\} \cup ins) :: I \rightsquigarrow E'', S}$$

La règle **PropagConnector** permet d'exécuter les instructions contenant un CONNECTOR en modifiant la valeur de sa PROPERTY cible est en retirant de la liste d'ensembles d'instructions l'instruction traitée afin d'exécuter les instructions suivantes. Dans la règle **PropagConnector**, une liste d'ensembles d'instructions ayant pour première instruction un CONNECTOR et s'exécutant dans un environnement E, donne un nouvel environnement E'', si l'exécution du CONNECTOR dans l'environnement E donne un environnement E' et que le reste de la liste d'ensemble d'instructions sans sa première instruction concaténé avec la liste d'ensemble $[S(x)]$, correspondant aux dépendances de x à activer, exécuté dans l'environnement E' donne l'environnement E''. Comme pour la règle **PropagBinding**, la concaténation de $[S(x)]$ correspond à la propagation de l'activation de x.

$$\textbf{PropagConnector}\ \frac{\begin{array}{c} E \ \vdash e \Rightarrow x \rightsquigarrow E' \\ E', S \ \vdash\ (ins :: I) + +[S(x)] \rightsquigarrow E'', S \end{array}}{E, S \ \vdash\ (\{e \Rightarrow x\} \cup ins) :: I \rightsquigarrow E'', S}$$

L'exécution du CONNECTOR est formalisée par la règle **ExecConnector** qui déclare que si l'expression e est évaluée à v dans l'environnement E alors l'exécution d'un CONNECTOR dans l'environnement E donne un nouvel environnement $E[e/v]$.

$$\textbf{ExecConnector}\ \frac{E \ \vdash e \Downarrow v}{E \ \vdash e \Rightarrow x \longrightarrow E[x/v]}$$

$$E = \{p1 \mapsto 0, p2 \mapsto 0, p3 \mapsto 0\}$$
$$S = \{p1 \mapsto \{p1 \Rightarrow p2\}, p2 \mapsto \{p2 \Rightarrow p3\}, p3 \mapsto \{\}\}$$

FIGURE 3.2 – Sémantique opérationnelle du programme de la figure 3.1.

$$\textbf{propagEnv}\ \dfrac{\textbf{ExecConnector propagConnector}\ \dfrac{\textbf{EvalVar}\ \dfrac{}{E[p1/42] \vdash p1 \Downarrow 42}}{E[p1/42] \vdash p1 \Rightarrow p2 \longrightarrow E[p1/42, p2/42]} \quad \dfrac{\dfrac{\dfrac{}{E[p1/42, p2/42] \vdash p2 \Downarrow 42}\ \textbf{EvalVar}}{E[p1/42, p2/42] \vdash p2 \Rightarrow p3 \longrightarrow E^T, S}\ \textbf{ExecConnector} \quad \dfrac{}{E^T, S \vdash [] \rightsquigarrow E^T, S}\ \textbf{Endpropag}}{E[p1/42, p2/42], S \vdash [\{p2 \Rightarrow p3\}] \rightsquigarrow E^T, S}\ \textbf{propagConnector}}{E[p1/42], S \vdash [\{p1 \Rightarrow p2\}] \rightsquigarrow E^T, S}$$
$$E, S \vdash Event(p1, 42) \rightsquigarrow E^T, S$$

FIGURE 3.3 – Execution du programme de la figure 3.1.

La règle **EndPropag** permet de terminer la phase de propagation lorsque il n'y a plus d'instruction à traiter en renvoyant l'environnement E mis à jour par toutes les propagations qui ont été effectuées.

$$\textbf{EndPropag}\ \frac{}{E, S \vdash [] \rightsquigarrow E, S}$$

3.2.3 Exécution d'un exemple dans la sémantique opérationnelle

Le programme de la figure 3.1 est représenté par la figure 3.2. Initialement, dans ce programme, toutes les PROPERTIES ont leur valeur à 0.

La figure 3.3 représente l'exécution du programme de la figure 3.1. Si la valeur de $p1$ passe à 42 suite à une interaction, alors la règle **propagEnv** est applicable. Pour vérifier l'hypothèse engendrée par **propagEnv** nous appliquons la règle **propagConnector**. Cette application engendre deux nouvelles hypothèses. Focalisons nous sur l'hypothèse de gauche. Pour la prouver, il suffit d'appliquer la règle **ExecConnector** puis de terminer la preuve avec la règle **EvalVar**. Si nous revenons à l'hypothèse de droite engendré par l'application de **propagConnector**, il suffit d'appliquer une nouvelle fois la règle **propagConnector** pour continuer le raisonnement. Nous nous retrouvons encore une fois avec deux cas à traiter. Le cas de gauche est prouvé en applicant successivement les règles **ExecConnector** et **EvalVar** et le cas de droite est prouvé en utilisant la règle **EndPropag**. Ainsi l'exécution de la sémantique justifie bien que le programme termine avec $E[p1/42, p2/42, p3/42]$ si une interaction change la valeur de $p1$ en 42.

3.2.4 Discussion

Dans cette section nous avons présenté une première sémantique qui repose sur des règles d'inférence exprimée en logique du premier ordre. Cette sémantique présente un programme comme un environnement E associant à chaque PROPERTY définie dans le programme sa valeur et une fonction S associant à chaque PROPERTY les instructions qu'elle active. Une sémantique opérationnelle, décrit l'interprétation d'un programme valide comme des séquences d'exécution d'instructions. Ce type de sémantique, dans notre contexte, a pour avantage d'avoir une description de l'exécution d'un programme précise mais rend difficile la représentation des connaissances venant du langage. En effet, le sous-ensemble de SMALA étudié, est restreint à la définition des PROPERTIES et à l'expression de leurs interactions qui sont exprimées de manière séquentielle par la sémantique opérationnelle. Ainsi, pour définir l'ordre d'exécution des instructions d'un programme lorsque une propagation d'activation d'une PROPERTY a lieu, nous avons eu besoin d'utiliser une liste d'ensembles pour l'ordonnancement.

Contrairement à un cadre mathématique proposant des constructions primitives adaptées à la définition de ce type d'instruction, nous nous retrouvons avec une sémantique plus complexe à comprendre notamment à cause de sa verbosité et donc une représentation des interactions peu intuitives. De plus dans le cadre de preuves formelles, ces structures de données pourraient rendre plus ardu le raisonnement et cela en devient plus compliqué avec les assistants de preuve demandant une description minutieuse de tout élément formalisé. Nous l'avons vu dans le chapitre précédent : les UIDLs ne se contentent pas d'exprimer les interactions, ils décrivent aussi les aspects spatiaux d'une interface graphique. L'idéal serait d'avoir un cadre mathématique ayant des constructions primitives permettant d'exprimer des interactions tout en pouvant spécifier leur ordre d'exécution et le fait qu'elles puissent être exécutées en concurrence. Nous aimerions également avoir des constructions primitives permettant d'exprimer la topologie d'une interface.

Pour cela, nous explorons dans la section suivante des algèbres de processus assez populaires dans la littérature et comparons leur expressivité, en terme spatio-interactif, à celle des bigraphes afin de rendre plus clair notre choix.

3.3 Algèbre de Processus

Les deux caractèristiques définissant les UIDLs, la description du graphe de scène et des interactions d'une interface, que nous avons présentés en section 2.1.2, nous ont naturellement guidé vers les algèbres de processus pour formaliser leur sémantique. Les algèbres de processus sont généralement utilisées pour décrire les communications entre différents processus et/ou leur espaces d'activités. Différentes algèbres de processus existent dans la littérature et chacune ont leur particularités. Dans la suite nous présentons une liste non exhaustive d'algèbres de processus spécialisées dans la communication et les aspects spatiaux de processus qui ont eu un impact majeur sur la recherche.

Nous commençons par présenter les populaires Réseaux de Petri [12] qui sont apparus en 1962. Les Réseaux de Petri sont connus pour décrire les comportements à caractère discret des systèmes. Durant les années 80, des évolutions de la théorie [22] qui ont été appelées Réseau de Haut Niveau sont apparues. Parmi ces Réseaux de Haut Niveaux nous trouvons les Réseaux Prédicat/Transition et les Réseaux de Petri colorés. Ces Réseaux de Haut Niveau apportent une couche d'abstraction aux Réseaux de Petri traditionnels permettant ainsi de faciliter la modélisation des systèmes étudiés. Ils facilitent la modélisation des systèmes en offrant la possibilité de manipuler des variables et des formules logiques mais aussi des structures de données.

Toujours dans les année 80, le CCS (Calculus of Communicating System), de Robin Milner, et le π-calculus, de Robin Milner, Joachim Parrow et David Walker, font leur apparition [30]. Le CCS propose des constructions afin de définir des interactions, communications et des moyens de synchronisation entre des processus indépendants. Ce calcul a servi de base au π-calculus qui reprend toutes les constructions du CCS et ajoute à cela des constructions permettant de modifier les inter-connections entre processus durant leur exécution.

L'ambient calculus [11] apparu en 1997, est un calcul qui se focalise sur la mobilité des processus. L'ambient calculus propose pour abstraction de base les ambients qui correspondent à des espaces bornés et des constructions permettant de les déplacer, les répliquer et les composer.

Le tableau 3.2 fait un comparatif de ces algèbres de processus en incluant la théorie des Bigraphes présentée dans la section 2.3 du chapitre 2. Nous comparons ces algèbres de processus sur leur capacité à représenter les communications et les mobilités de processus et nous constatons que seule la théorie des bigraphes possède les critères recherchés. Les réseaux de Petri et ceux de Haut Niveau décrivent les comportements discrets d'un système par le biais d'échange de jetons ou de transition d'un état à un autre qui correspond bien à des communications. Quant aux CCS et au π-calculus, ce sont à

Algèbre de Processus	Mobilité	Communication
Réseaux de Petri/ Réseaux de Petri de Haut niveau		X
CCS / π-calculus		X
Ambient Calculus	X	
Bigraphes	X	X

TABLE 3.2 – Comparaison d'algèbres de processus sur leur capacité à représenter les communications entre processus et leur mobilité.

travers des opérateurs (e.g. synchronisation, communication) ou bien des échanges qu'ils décrivent le comportement d'entités d'un système, ce qui correspond clairement à des méthodes de communication. À propos du calcul des ambients, ce sont notamment ces opérateurs permettant aux ambients d'interagir avec leur espace qui ont permis de le catégoriser en tant qu'algèbre pouvant représenter des mobilités.

La théorie des bigraphes n'est pas le seul formalisme à pouvoir exprimer les mobilités et communications à la fois, il en existe d'autres comme par exemple des extensions du π-calculus [45, 18]. Mais les bigraphes se démarquent des autres algèbres de processus par le fait que c'est le formalisme qui permet de représenter d'autre algèbre de processus, notamment le π-calculus et le calcul des ambiants, dans un cadre uniforme.

Dans la section suivante, nous présentons une sémantique de notre sous-ensemble formalisée avec la théorie des bigraphes.

3.4 Une sémantique bigraphique

La sémantique donnée dans cette section repose sur les bigraphes. La sémantique consiste en un BRS et associe un bigraphe à chaque PROCESSUS. Les bigraphes associés aux PROCESSUS du sous-ensemble peuvent être composés et ainsi former un programme. Cet aspect de la sémantique est appelé l'aspect structurel et est présenté dans la section 3.4.1. Les règles de réactions définissant le BRS permettent de définir l'aspect dynamique de la sémantique e.g. la propagation d'activation. Cet aspect appelé l'aspect dynamique est présenté dans la section 3.4.2.

3.4.1 Aspect structurel de la sémantique

Le tableau 4.1 présente les bigraphes que nous avons associés à chaque PROCESSUS du sous-ensemble. Nous commençons par présenter les bigraphes correspondant aux BINDINGS et aux CONNECTORS. Le BINDING (respectivement CONNECTOR) est composé d'une entité de contrôle Binding (respectivement Binding) qui contient deux entités de contôle Source et Target. Si une PROPERTY est connectée à l'entité Source (respectivement Target) alors elle sera la source (respectivement la cible) du BINDING (respectivement CONNECTOR).

Ensuite le bigraphe associé à la PROPERTY est présenté. Celui-ci est composé d'une entité de contrôle Property qui contient deux entités de contrôle Id et Value. L'entité Id encapsule une chaîne de caractères représentant l'identifiant de la PROPERTY et l'entité Value contient une autre entité de contrôle Val qui encapsule sa valeur.

En composant les bigraphes présentés précédemment, par lien ou placement, il est possible de constituer tous les programmes que notre sous-ensemble permet de définir. Les figures 3.1 et 3.4 illustrent cela en présentant un programme ainsi que son bigraphe correspondant.

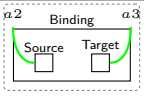
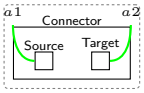
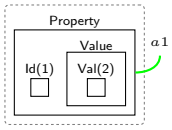
Instruction du langage	Bigraphe correspondant
_ → _	
_ ⇒ _	
`Int x (2)`	

TABLE 3.3 – Bigraphes correspondants aux instructions du sous-ensemble de SMALA — Le tiret du bas remplace des expressions des PROCESSUS.

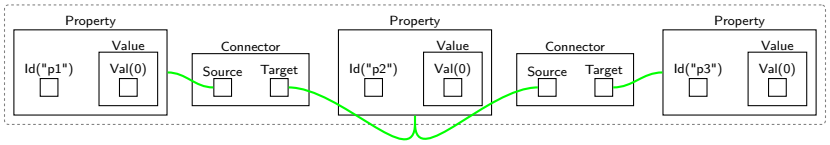


FIGURE 3.4 – Bigraphe correspondant à l'exemple de la figure 3.1.

L'aspect structurel d'un programme maintenant défini, nous présentons dans la section suivante les règles de réaction définissant l'aspect dynamique d'un programme qui définit le comportement de celui-ci lorsqu'une activation a lieu.

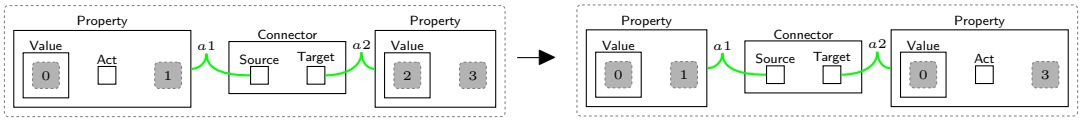
3.4.2 L'aspect dynamique de la sémantique

Trois règles de réactions définissent l'aspect dynamique de la sémantique. Les règles de réactions présentées dans les figures figure 3.5a et figure 3.5b définissent respectivement le comportement du CONNECTOR et du BINDING lors de l'activation de leur PROCESSUS source.

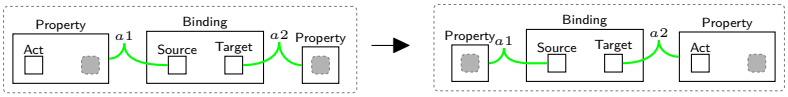
La figure 3.5a définit comment un CONNECTOR change la valeur de son PROCESSUS cible et l'active lorsque son PROCESSUS source est activé. Dans la partie gauche de la règle, l'activation d'une PROPERTY est représentée par une entité de contrôle Act qui est déplacée à la PROPERTY cible du CONNECTOR, ce qui indique la propagation. La copie de la source vers la cible se fait en remplaçant le site encapsulé dans l'entité Value de la cible par celui encapsulé par l'entité Value de la source.

La figure 3.5b définit comment un BINDING propage l'activation de son PROCESSUS source à son PROCESSUS cible. Dans la partie gauche de la règle, l'activation du PROCESSUS source se voit avec l'entité Act encapsulé par celui-ci. Dans la partie droite de la règle, l'entité Act est déplacée au PROCESSUS cible ce qui indique que l'activation a bien été propagée à la PROPERTY cible.

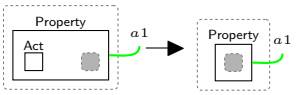
La figure 3.5c définit une règle permettant de traiter les PROPERTIES activées qui ne sont source d'aucun PROCESSUS de communication. Cette règle retire simplement l'entité Act du PROCESSUS.



(a) Règle de réaction pour la propagation d'une activation à travers un CONNECTOR.



(b) Règle de réaction pour la propagation d'une activation à travers un BINDING.



(c) Règle de ré-initialisation d'une PROPERTY.

FIGURE 3.5 – Règles de réaction pour le sous-ensemble étudié.

3.4.3 Exécution d'un exemple dans la sémantique bigraphique

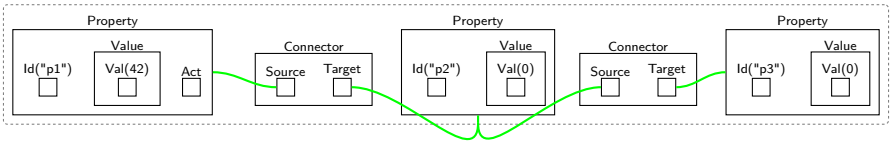
La figure 3.4 présente le bigraphe correspondant au programme de la figure 3.1. Ce bigraphe contient trois bigraphes représentant les PROPERTIES $p1$, $p2$ et $p3$ et deux bigraphes représentant les CONNECTOR liant la PROPERTY $p1$ (la source) à la PROPERTY $p2$ (la cible) et la PROPERTY $p2$ (la source) à la PROPERTY $p3$ (la cible).

La figure 3.6 présente l'exécution du programme lorsque la PROPERTY $p1$ est modifiée. La figure 3.6a montre le programme lorsque la PROPERTY $p1$ est activée et que sa valeur a été changée en 42. La seule règle applicable à ce programme est la règle de la figure 3.5a qui va propager l'activation à $p2$ et modifier sa valeur en même temps. La figure 3.6b illustre le bigraphe représentant cela. La figure 3.6c représente la prochaine étape de l'exécution qui applique au bigraphe courant la règle de la figure 3.5a qui propage l'activation de $p2$ à $p3$ et change sa valeur en 42. La dernière étape de l'exécution représentée par la figure 3.6d consiste à ré-initialiser la PROPERTY $p3$, en retirant l'entité Act, pour une éventuelle nouvelle interaction dans le futur.

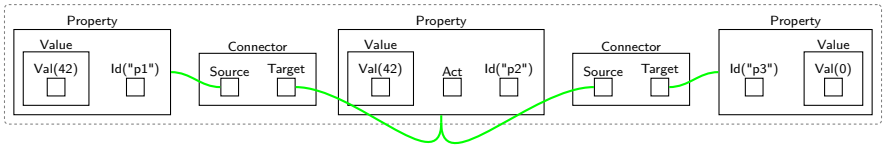
3.4.4 Discussion

Dans cette section nous avons présenté une deuxième sémantique pour notre sous-ensemble basée sur les bigraphes. Cette sémantique représente un programme comme un bigraphe qui grâce à sa notation en diagramme, permet de visualiser toutes les PROPERTIES et interactions définies.

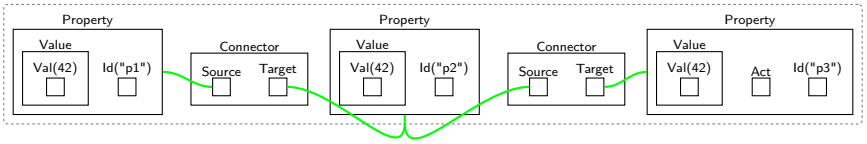
Si nous comparons la représentation bigraphique de l'exemple présenté dans la figure 3.1 à la représentation de la figure 3.2, le programme est clairement représenté, nous pouvons voir toutes les PROPERTIES et tous les CONNECTORS défini grâce à leur diagramme correspondant ainsi que chaque connexion grâce aux liens. Concernant l'exécution du programme, présenté dans la figure 3.6, il est possible de suivre clairement la propagation d'activation de la PROPERTY $p1$. L'activation qui est représentée par le contrôle Act peut être vue et comparée à la précédente exécution du programme et en conséquence il est possible de suivre sa propagation d'une étape à une autre jusqu'à que la règle en figure 3.5c soit appliquée. Enfin, l'ordre de propagation est clairement représenté par les liens qui sont orientés par les contrôles Source et Target.



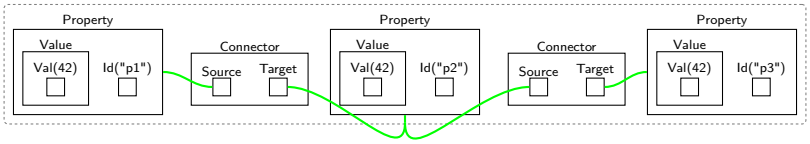
(a) Bigraphe représentant l'Exemple2 après que l'interaction ait changé la valeur de $p1$ en 42.



(b) Bigraphe (a) après avoir appliqué la règle figure 3.5a.



(c) Bigraphe (b) après avoir appliqué la règle figure 3.5a.



(d) Bigraphe (c) après avoir appliqué la règle figure 3.5c.

FIGURE 3.6 – Exécution de l'Exemple2 selon le BRS qu'il définit.

3.5 Conclusion

Dans cette section nous avons présenté un sous-ensemble du langage Smala et donné sa sémantique dans deux cadres mathématiques différents afin de comparer les différences liées à leur représentation des connaissances. La première sémantique donnée consiste en une sémantique opérationnelle exprimée en logique du premier ordre et la deuxième consiste en une sémantique dénotationnelle utilisant les bigraphes. Grâce à cette comparaison nous pouvons conclure, dans le contexte de l'étude formelle des UIDLs, que malgré que deux cadres mathématiques différents aient la même expressivité alors les formalisations de sémantiques d'UIDL qu'ils peuvent engendrer peuvent différer en terme de représentation des connaissances. L'une des raisons observées est liée à la nature du cadre utilisé dont notamment la mise à disposition d'opérateurs primitifs spécifiques aux interactions et aux mobilités.

Nous terminons ce chapitre en mettant l'accent sur une observation pertinente concernant les preuves formelles sur la sémantique des UIDLs et ainsi les interfaces graphiques. Le choix d'un cadre mathématique selon la compréhension épistémique d'un système pourrait s'avérer judicieux dans le cadre d'une preuve formelle. Nous l'avons vu précédemment, un cadre mathématique non adapté peut rendre la représentation du système verbeuse et la complexifier à cause de la définition de concept non existant dans le formalisme choisi. Ces complexités pourraient donc se répercuter dans des preuves à effectuer sur le système formalisé.

Dans le chapitre suivant nous définissons notre UIDL formel basé sur la théorie des bigraphes et présentons sa sémantique.

Chapitre 4

Biguil, Bigraphical User Interface Language

Dans ce chapitre nous définissons *Biguil*, Bigraphical User Interface Language. *Biguil* est la réponse que nous apportons à l'interrogation portant sur l'existence d'un langage aux instructions élémentaires qui permettent de définir des constructions interactives et spatiales complexes. *Biguil* est un UIDL formel que nous définissons pour avoir :

- à court terme, des modèles formels des interfaces utilisateurs décrites que nous pourrons utiliser pour effectuer des vérifications ;
- à long terme, un langage dont la compilation préservera toutes propriétés interactives et spatiales définissant une interface utilisateur.

Pour réaliser ces objectifs, nous définissons *Biguil* comme un UIDL standard, i.e. un langage qui permet de décrire la topologie et les interactions d'une interface graphique et qui respecte les propriétés qui définissent la sémantique des UIDLs, présentées dans la section 2.1.2 du chapitre 2. Pour répondre à ces attentes, nous avons défini le langage de sorte qu'il soit formé d'un ensemble d'instructions restreint dont chacune d'elle a le pouvoir d'exprimer des concepts interactifs et spatiaux élémentaires. Ainsi, le langage a un nombre minimal d'instructions qui permettent de définir des concepts spatio-interactifs plus complexes et a pour avantage de faciliter le raisonnement sur sa sémantique.

Nous avons souligné dans le chapitre précédent l'importance du choix du formalisme concernant la représentation des connaissances exprimés par sémantique d'un langage. Nous avons donc décidé de formaliser notre langage en utilisant une instance de la théorie des bigraphes. La sémantique de *Biguil* est alors décrite par des règles de réaction et chaque interface décrite est modélisée par un bigraphe de cette instance. Formaliser la sémantique d'un langage informatique avec de la réécriture de graphe est courant, la section 4.1 présente des travaux sur cette thématique dont un utilisant la théorie des bigraphes. Nous présentons dans la section 4.2 les conceptes de bases du langage et la section 4.3 donne la définition de la sémantique de *Biguil* à travers des règles de réécritures.

4.1 Sémantique de langages informatique et réécritures de graphe

La réécriture de graphe a une certaine popularité et est utilisée dans de nombreuses disciplines, autre que l'informatique, e.g. la biologie [33] et la chimie [7]. Autour de l'Interaction Humains-Machine, la flexibilité et l'expressivité que nous avons à définir ces règles de réécriture ainsi que sa pragmatique

représentation en diagramme a convaincu des chercheurs à utiliser la réécriture de graphe pour formaliser des transformations d'interface utilisateur [50, 48]. Dans la suite nous présentons deux travaux utilisant la réécriture de graphe pour formaliser la sémantique de langage informatique.

Kappa [17] est un outil permettant de modéliser, simuler ainsi qu'analyser des systèmes biochimiques. Les modélisations se font à travers de règles qui décrivent les modifications à effectuer sur les liens entre les différents sites des agents, i.e. les entités d'une réaction. Dans un contexte biologique, les agents correspondent à des protéines et les sites correspondent à des domaines protéiques. Les régles de ré-écriture définies par l'utilisateur pour modéliser une réaction chimique sont basées sur les Σ-graphes. Les Σ-graphes, sont des graphes qui ont :

— des nœuds typés correspondant aux agents des réactions,
— des sites qui sont identifiés et associés à des agents, correspondant aux sites protéiques,
— une relation de lien qui est symétrique entre les nœuds, et
— un ensemble de propriétés sur les agents et les sites.

Le langage mis à disposition aux utilisateurs permet de définir des règles de ré-écriture sur ces Σ-graphes. La sémantique de ce langage est basée sur des ré-écritures de graphe en simple push-out, qui est un cadre mathématique pour la ré-écriture de graphe développé dans [26].

Des résultats autour de la modélisation et la vérification de l'architecture Beliefs-Desires-Intentions (BDI) publiés dans [3] utilisent la théorie des bigraphes comme formalisme. BDI est un cadre populaire pour les agents rationnels dans lequel ceux-ci sont caractérisés par leur croyances (B), désirs (D) et intentions (I). La croyance d'un agent correspond à ce qu'il sait, le désir correspond à ce qu'il veut apporter au système et l'intention correspond aux désirs que l'agent va réaliser. De nombreux langages de spécification d'agent BDI ont émergé grâce à la popularité de ce cadre, dont notamment CAN [43]. L'article contenant ces résultats, présente une instanciation de la théorie des bigraphes comme sémantique du langage CAN et donne sa preuve de correction.

Dans la section suivante, nous définissons les concepts de base de *Biguil* et dans la suite, comme les auteurs du dernier article présenté, nous allons donner une sémantique de notre langage par une instance de la théorie des bigraphes.

4.2 Concepts de base

Biguil (Bigraphical user interface language) est un langage conçu pour la compilation vérifiée d'interfaces utilisateur. Plus exactement, c'est un UIDL avec des fondements théoriques basés sur les bigraphes et permettant de vérifier les interfaces utilisateur qu'il décrit. C'est un langage déclaratif permettant de décrire des entités graphiques et des interactions qui constituent un graphe de scène. Ainsi, le langage permet de définir des entités graphiques telles que des figures géométrique et des layouts (mécanisme d'agencement pour composants graphiques) permettant de les positionner, ou bien des interactions telles qu'un clic de souris. Ces entités sont regroupées sous la notion de PROCESSUS, qui est l'abstraction de base du langage et qui est présenté dans la section 4.2.1. Leur manipulation à travers le langage permettant ainsi la création d'un graphe de scène est expliquée dans la section 4.2.2 qui décrit la grammaire de *Biguil*.

4.2.1 Les processus

Notre but est de définir un langage formel et pour cela nous avons besoin d'une abstraction de base qui sera le fondement théorique du langage. L'utilisation d'une abstraction de base permet de raisonner de façon générale sur les concepts du langage. Par exemple, utiliser une abstraction permettrait de définir une propriété couvrant l'ensemble des éléments graphique d'un langage. L'abstraction

Programme	Prog ::=	`root : Component` $\{C\}$
Expression	e	Identifiant, constante, expressions arithmétiques etc
Identifiant		id
Liste de processus	C ::=	$C\ c \mid \epsilon$
Processus	c ::=	id : `Component <i>` $[f]$ { C }
		$\mid id$: `GComponent <i>` $[f_g]$ { C }
		$\mid id$: `Spike <i>` $[ev]$
		$\mid id$: id `b<i>` id
		$\mid id$: e `=:<i>` id
Paramètre	p ::=	$id \mid e$
Opérateur de BINDING	b ::=	$\rightarrow \mid !\rightarrow \mid \rightarrow! \mid !\rightarrow!$
Fonction	$f \in$	{`Adder`, `FSM`...}
Fonction graphique	$f_g \in$	{`Rectangle`, `Frame`...}
Fonction	$ev \in$	{`Exit`, `Click`...}
Activation Initiale	`i` $\in$	{`a`,`d`}

FIGURE 4.1 – Grammaire *Biguil*.

que nous définissons pour *Biguil* est inspiré des PROCESSUS de SMALA (section 2.2.1) car c'est une abstraction avec laquelle nous avons l'habitude de travailler. Donc, tout comme SMALA, le PROCESSUS est l'abstraction de base du langage *Biguil*, i.e. chaque composant de l'interface graphique est un PROCESSUS. Ils peuvent représenter des composants graphiques, des structures de données ou même des structures de contrôle. Nous définissons un PROCESSUS comme en section 2.2.1 en ajoutant deux éléments supplémentaires que nous avons différencié en les sous-lignant :

- un *identifiant* unique
- un *état* (Activé ou Désactivé), indiquant si le PROCESSUS est activé et peut s'exécuter selon sa sémantique, ou désactivé et ne peut pas s'exécuter jusqu'à qu'il soit activé.
- une *sémantique*, décrivant le fonctionnement du PROCESSUS. En particulier, nous pouvons avoir des PROCESSUS gérant la communication inter-processus ou d'autres représentant des composants graphiques
- un *type* (Persistant ou Transitoire), décrivant l'activation du PROCESSUS. Un PROCESSUS Persistant est un PROCESSUS restant Activé jusqu'à la fin de l'exécution du programme ou jusqu'à ce qu'il soit désactivé par un autre PROCESSUS (e.g. un composant graphique) tandis qu'un PROCESSUS Transitoire est un processus qui s'exécute selon sa sémantique lors de son activation puis retourne à l'état Désactivé (e.g. l'activation d'un signal).

Les PROCESSUS permettent de constituer un programme. La section 4.2.2, au travers de la grammaire décrite dans la figure 4.1, présente la manière de définir des PROCESSUS et de les utiliser pour créer un graphe de scène *Biguil*.

4.2.2 Grammaire

La figure 4.1 présente la grammaire de *Biguil*.
Dans *Biguil* il est possible de définir :

- Des `Component` qui sont des PROCESSUS conteneurs, de type Persistant, pouvant contenir d'autres PROCESSUS. Ces derniers sont appelés les PROCESSUS enfants du `Component` et ils sont activés (respectivement désactivés) lorsque leur parent s'active (respectivement se désactive). Il est possible d'attribuer aux `Component` une fonction qui précise leur sémantique. Par exemple, il est

possible d'attribuer la fonction de PROPERTY qui est un PROCESSUS encapsulant une valeur de type `Int`, `Double`, `Bool` ou `String` et qui émet une activation transitoire à chaque mise à jour de sa valeur, à un `Component`

— Des `GComponent` qui sont similaires aux `Component`, mis à part qu'ils représentent une entité graphique du graphe de scène. Nous distinguons les `GComponent`, des `Component`, car cela va nous permettre de distinguer la partie logique de l'interface graphique de sa partie graphique. Comme les `Component`, les `GComponent` ont une fonction qui précise leur sémantique e.g. un `GComponent` pourrait correspondre à une figure géométrique tel qu'un `Rectangle`.

— Des `Spike` qui ont pour fonction de base de représenter une activation transitoire. Généralement ils sont utilisés pour représenter des évènements tels qu'un clic de souris, l'appui d'une touche de clavier ou bien le signal de fin d'un programme.

— Des BINDINGS $id \rightarrow id$ qui sont de type Persistant et permettent à un PROCESSUS source de communiquer son état d'activation à un PROCESSUS cible qui changera son état d'activation en conséquence. Il existe quatre BINDINGS différents : le BINDING $\rightarrow$ active sa cible lorsque sa source s'active ; le BINDING $! \rightarrow$ active sa cible lorsque sa source se désactive ; le BINDING $\rightarrow!$ désactive sa cible lorsque sa source s'active ; le BINDING $! \rightarrow!$ désactive sa cible lorsque sa source se désactive. Les BINDINGS $! \rightarrow$, $\rightarrow!$ et $! \rightarrow!$ sont contraints de manipuler uniquement la désactivation des PROCESSUS Persistant.

— Des PROCESSUS d'ASSIGNMENTS $e \ =: \ id$, de type Transitoire, permettant de copier la valeur de l'expression e et de l'écrire dans la propriété qui a pour identifiant id.

Il est possible de nommer tout les PROCESSUS par un identifiant id et d'indiquer par les valeurs `a` (activé) et `d` (désactivé) si ils doivent être activés lors de l'activation de leur parent. Nous nommons l'action d'activation d'un PROCESSUS par son parent, *activation initiale*.

Un programme *Biguil* consiste en un `Component` `root` contenant des PROCESSUS enfants. Nous donnons dans la figure 4.2, un exemple d'un programme *Biguil*.

Le programme décrit en figure 4.2 consiste en un `Component` nommé `root` contenant comme PROCESSUS enfant :

l2 un PROCESSUS fenêtre (`Frame`) nommé `f` de taille 500×400, positionné aux coordonnées $(0, 0)$ de l'écran, qui a pour titre `"Circle"` et qui est Persistant ;

l20 un `GComponent` `FillColor` nommé `red` qui a trois enfants qui sont des propriétés de type double et qui correspondent aux valeurs de rouge, de vert et de bleu définissant la couleur ;

l30 un `GComponent` `Circle` qui a pour centre $(200, 200)$ (sur le canvas de la fenêtre) et un rayon de 50 ;

l43 un PROCESSUS `Spike` `Exit` nommé `e` permettant de mettre fin à l'exécution du programme lors de son activation ;

l44 un BINDING qui permet d'activer le PROCESSUS `e` lorsque le bouton `f.close` de la fenêtre est appuyé.

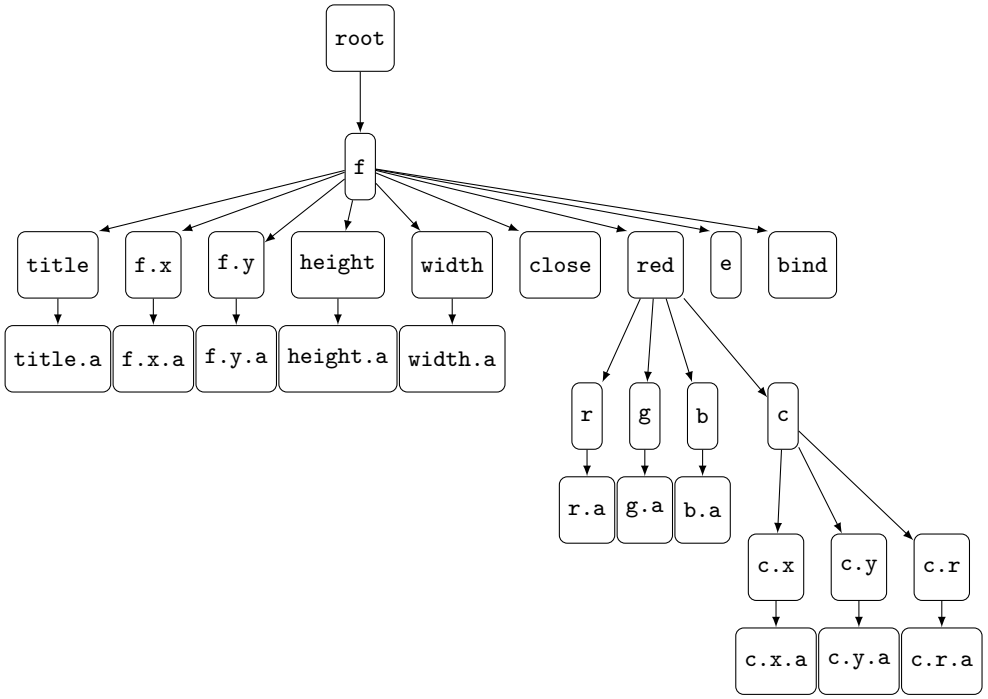
Concrètement, ce programme décrit une fenêtre de taille 500×400 contenant un cercle rouge de rayon 50 à la position $(200, 200)$. Lorsque le bouton FERMER de la fenêtre est cliqué par l'utilisateur, le programme prend fin.

4.2.3 Graphe de scène

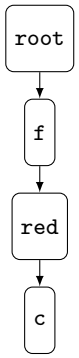
La définition d'un programme *Biguil* induit la création d'un graphe de scène, la figure 4.3 montre le graphe de scène de l'exemple précédent. Ce graphe de scène est constitué de deux structures d'arbre ainsi qu'une structure de graphe.

```
root : Component<a> {
  f : GComponent<a> Frame {
    title : Component <a> String("Circle") {
      title.a : Spike<d>
    }
    f.x : Component <a> Double(0) {
      f.x.a : Spike<d>
    }
    f.y : Component <a> Double(0) {
      f.y.a : Spike<d>
    }
    height : Component <a> Double(500) {
      height.a : Spike<d>
    }
    width : Component <a> Double(400) {
      width.a : Spike<d>
    }
    f.close : Spike<d> Close

    red : GComponent<a> FillColor {
      r : Component <a> Double(255) {
        r.a : Spike<d>
      }
      g : Component <a> Double(0) {
        g.a : Spike<d>
      }
      b : Component <a> Double(0) {
        b.a : Spike<d>
      }
      c : GComponent<a>  Circle {
        c.x : Component <a> Double(200) {
          c.x.a : Spike<d>
        }
        c.y : Component <a> Double(200) {
          c.y.a : Spike<d>
        }
        c.r : Component <a> Double(50) {
          c.r.a : Spike<d>
        }
      }
    }
  }
  e : Spike<d> Exit
  bind : f.close -><a> e
}
```

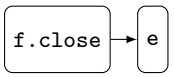
FIGURE 4.2 – Exemple d'un programme *Biguil*.



(a) Arbre des PROCESSUS.



(b) Arbre des composants graphiques.



(c) Graphe de Communication.

FIGURE 4.3 – Graphe de scène du programme *Biguil* de la figure 4.2.

La structure d'arbre (figure 4.3a) est un arbre formé par l'ensemble des PROCESSUS défini dans le programme. Les nœuds de l'arbre correspondent à la définition de `Component` ou de `GComponent` et les feuilles correspondent à la définition des autres PROCESSUS. Lorsqu'un PROCESSUS conteneur est activé, ses descendants vont être activés en cohérence de cette arborescence.

Le deuxième arbre (figure 4.3b), qui est induit par la définition des PROCESSUS graphiques, est une abstraction de l'interface graphique du programme. Cet arbre a pour racine le `Component root` qui a automatiquement pour enfant tous les PROCESSUS fenêtres définis dans le programme. Le `GComponent` fenêtre, de par sa fonction, se distingue des autres `GComponent` car chaque nœud de l'arbre graphique désigne un canvas dans lequel une figure peut être dessiné. Le PROCESSUS fenêtre ne pouvant pas être dessiné sur un canvas est automatiquement enfant du `Component root`. Les autres PROCESSUS graphiques ont pour parent le premier `GComponent` dans lequel ils sont inclus et un `GComponent a` encapsulé (en tant que enfant où descendant) dans un autre `GComponent b` a pour repère géométrique son parent `b`. De façon générale, cet arbre donne une information sur le repère géométrique de chaque PROCESSUS graphique.

La structure de graphe (figure 4.3c) représente les communications entre PROCESSUS définis dans le programme. Ces communications sont issues de l'utilisation des quatre BINDINGS et du PROCESSUS d'assignation.

Dans la prochaine section nous donnons une définition formelle de la sémantique de *Biguil*.

4.3 Sémantique

Nous avons vu dans le chapitre 3 que l'une des motivations de l'utilisation des bigraphes est que la sémantique des UIDLs est basée sur l'aspect structurel et les interactions des interfaces graphiques. Ainsi, nous avons décidé de fonder *Biguil* sur les bigraphes, en associant à chaque interface graphique décrite un système réactif bigraphique. Ce système a pour état initial un programme, qui est différent pour chaque interface décrite, et pour règles de réaction un ensemble de règles, qui reste fixe pour tout programme, définissant la sémantique du langage. Ce bigraphe représente en deux perspectives le graphe de scène d'un programme. La première perspective, appelée perspective logique, consiste en la description de l'arbre des processus ainsi que le graphe de communications. Cette perspective est centrée sur la représentation de la structure logique du programme ainsi que les communications entre les différentes entités de l'interface. La seconde, appelée perspective graphique, consiste en la description de l'arbre des composants graphiques et donne une abstraction de l'interface graphique du programme. La section 4.3.1, présente la construction du bigraphe initial à partir d'un programme induit par la syntaxe présentée en section 4.2.2. La section 4.3.2 présente comment une interaction est abstraite et comment elle impacte l'interface graphique définie. Enfin, les section 4.3.3 et 4.3.4 présentent respectivement, via des règles de réaction, la phase d'activation et de propagation d'un PROCESSUS qui sont au cœur de la sémantique de *Biguil*. Toutes les notions de bigraphes nécessaires à la compréhension de cette section sont disponibles dans la section 2.3 du chapitre 2.

4.3.1 Sémantique structurelle

La notion de PROCESSUS étant au coeur du langage *Biguil* nous commençons par donner en premier lieu, dans figure 4.4, la représentation d'un PROCESSUS en bigraphe. Un PROCESSUS consiste en une entité de contrôle Process d'arité un. Ce lien permet au PROCESSUS, par exemple, d'être lié à d'autre PROCESSUS via un BINDING ou un ASSIGNMENT. Une entité Process contient six autres entités.

1. L'entité Type a pour rôle d'encapsuler l'information au sujet du type du PROCESSUS i.e. une entité de contrôle Persistent ou Transient.

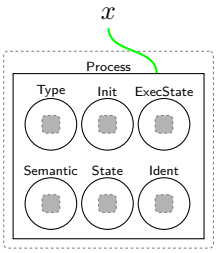


FIGURE 4.4 – Bigraphe correspondant à la sémantique structurelle d'un PROCESSUS.

2. L'entité Init encapsule l'information concernant le comportement que le PROCESSUS doit adopter lorsqu'il est activé par son parent. L'entité peut soit contenir une entité de contrôle On ou bien une entité Off symbolisant respectivement l'activation et la désactivation du PROCESSUS.
3. L'entité de contrôle Semantic contient un bigraphe qui a pour but de définir la sémantique du PROCESSUS.
4. L'entité State contient l'information concernant l'état d'activation d'un PROCESSUS. L'entité State peut soit contenir une entité On dans le cas où le PROCESSUS est activé ou bien une entité Off dans le cas où celui-ci est désactivé.
5. L'entité Ident encapsule l'identifiant unique du PROCESSUS.
6. L'entité ExecState correspond à l'état d'exécution d'un PROCESSUS et son utilité sera détaillée dans section 4.3.2.

Cette formalisation des PROCESSUS permet d'associer un bigraphe à chaque PROCESSUS du langage (tableau 4.1). Ainsi, un `Component` est représenté par un bigraphe PROCESSUS dont l'entité Type contient une entité de contrôle Persistent et l'entité Semantic contient une entité de contrôle Component. L'entité Component permet de contenir les enfants du `Component` ainsi que sa fonctionnalité attribuée, le cas échéant.

Un `GComponent` est représenté de la même façon qu'un `Component` à une seul exception près. Une entité de contrôle GProcess, faisant partie de la perspective graphique, est liée à l'entité Process. L'entité GProcess est l'abstraction du composants graphique défini à travers le `GComponent`.

Un `Spike` est représenté par un bigraphe PROCESSUS dont l'entité Type contient une entité de contrôle Transient et l'entité Semantic contient une entité de contrôle Spike. L'entité Spike peut contenir une entité représentant l'évènement qu'un `Spike` abstrait, le cas échéant.

Un BINDING est représenté par un bigraphe PROCESSUS de type Persistant et l'entité Semantic contient une entité de contrôle Binding. L'entité Binding contient trois autres entités.

— L'entité de contrôle Target d'arité un, représentant un port de connexion pour les PROCESSUS cible,
— une entité de contrôle Source d'arité un, représentant un port de connexion pour les PROCESSUS source
— une entité de contrôle II, IO, OI ou OO représentant le type du BINDING (e.g. OI pour le binding $\rightarrow$!).

Enfin un ASSIGNMENT est représenté par un bigraphe PROCESSUS de type Transitoire et dont l'entité Semantic contient une entité de contrôle Assignment. L'entité Assignment contient deux autres entités de contrôle Target et Source ayant la même utilité que les entités Source et Target du bigraphe BINDING.

Chacun des bigraphes défini précédemment, contient les entités Init et Ident ainsi que les entités ExecState et State (se référer à la description des PROCESSUS dans section 4.2.1). Ces entités ne sont pas représentées car elles dépendent respectivement du choix de l'utilisateur et de l'état d'exécution

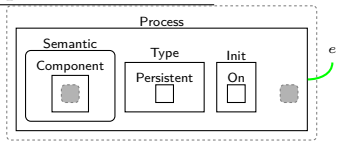
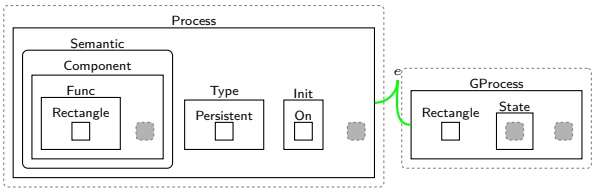
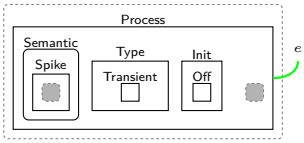
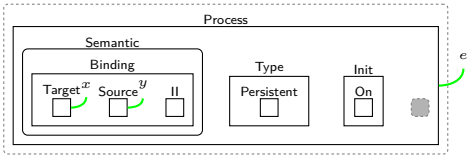
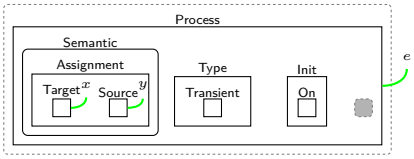
Instruction du langage	Bigraphe correspondant
`c : Component <_>`	
`r : GComponent <_> Rectangle`	
`s : Spike <_>`	
`b : _ -><_> _`	
`a : _ =:<_> _`	

TABLE 4.1 – Bigraphes correspondants aux instructions de *Biguil*. Le tiret du bas remplace des expressions ou le paramètre de l'activation initiale du PROCESSUS.

du programme. Par composition, de lien et de place, les bigraphes décrits dans tableau 4.1 peuvent former des bigraphes à deux perspectives, logique et graphique, représentant un programme *Biguil*. Cependant ces compositions sont réglementées par certaines règles de la sémantique présentées ci-dessous :

- par défaut, le PROCESSUS `root` a pour unique fonction d'encapsuler d'autres PROCESSUS et a toujours son paramètre d'activation initiale à `a` ;
- les seuls PROCESSUS pouvant avoir des enfants sont les PROCESSUS conteneurs i.e. les Components et les GComponents ;
- les sources et les cibles des ASSIGNMENTS sont uniquement des PROPERTIES i.e. des Components qui ont pour fonction DOUBLE(X), INT(X), STRING(X) ou BOOL(X) ;
- les PROCESSUS ne peuvent pas être connectés à leur parent ou enfant par BINDING ;
- un PROCESSUS ne peux pas être rattaché, en tant que source ou cible, à un BINDING enfant ;
- la désactivation des PROCESSUS Transitoire ne peut pas être exploité par les BINDINGS.

Afin d'éviter de former des modèles d'interface *Biguil* incorrectes, il est possible de définir un sorting (de placement et de lien) sur ces bigraphes afin de définir minutieusement l'ensemble des programmes valides par composition de *Biguil*. Ce sorting permettrait aussi de définir les compositions correctes sur les méta-structures du langage, par exemple avec un tel sorting il serait possible d'éviter qu'une entité State contiennent une entité Semantics.

Propriété 4.3.1 (Loi de composition) *Il existe un sorting de placement et de lien sur les bigraphes définis précédemment définissant toutes les représentations* valide *de programmes* Biguil.

La section 4.3.2 étant centrée sur la sémantique des interactions de *Biguil*, nous verrons que la formalisation donnée des PROCESSUS facilite sa définition car l'aspect générique venant des PROCESSUS est utilisé afin de définir des règles de réaction également génériques.

4.3.2 Sémantique des interactions

Les UIDLs permettent de définir des réactions de l'interface graphique suite à une interaction utilisateur. Ces réactions peuvent correspondre à l'exécution d'un script ou des changements visuels sur l'interface tels que l'apparition de nouveaux composants graphiques. Dans notre sémantique, les `Spike` sont utilisés pour représenter ces interactions. Par exemple, ils sont utilisés pour modéliser un clic de souris ou bien un appui sur une touche de clavier. En conséquence, une interaction avec l'interface graphique est représentée par le changement de l'état d'exécution du `Spike` modélisant l'interaction en question. Ce changement d'état permet d'activer le `Spike` concerné et de propager son activation à ses dépendances, des PROCESSUS qui vont réagir à l'interaction et qui peuvent aussi avoir des dépendances. Ainsi les changements d'une interface graphique suite à une interaction sont dus à l'activation ou la désactivation en chaîne de PROCESSUS. Les mécanismes d'activation/desactivation d'un PROCESSUS ainsi que ceux de propagation de changement d'état à d'autre PROCESSUS respectent les propriétés essentielles aux UIDLs présentées dans la section 2.1.2 du chapitre 2. Dans la section suivante, nous redéfinissons ces propriétés en fonction de notre formalisme.

Propriétés de *Biguil*

Biguil a été défini de sorte à couvrir les propriétés essentielles aux UIDLs suivantes.

La propriété 4.3.2 permet aux PROCESSUS d'être activés seulement si leur parent est activé. Cette propriété assure que chaque `GComponent` est représenté sur un canvas qui existe à ce moment là.

Propriété 4.3.2 (Cohérence d'activation parent-enfant) *Un PROCESSUS, différent du PROCESSUS `root`, est activé seulement si son PROCESSUS parent est activé.*

Les propriétés 4.3.1 et 4.3.2 permettent de définir les deux types de PROCESSUS existant dans le langage, les PROCESSUS Persistant et les PROCESSUS Transitoire.

Propriété 4.3.3 (Activation persistante) *Lors d'une réaction en chaîne, l'état d'un PROCESSUS Persistant peut changer au plus une fois. Son état d'activation est initialement activé ou désactivé et peut changer respectivement en désactivé ou activé.*

Propriété 4.3.4 (Activation transitoire) *Lors d'une réaction en chaîne, l'état d'un PROCESSUS Transitoire peut changer au plus deux fois. Son état d'activation est initialement désactivé et dans le cas de son activation celui-ci va s'activer puis se désactiver.*

La propriété 4.3.5 permet de faire le lien entre les perspectives logique et graphique du programme. Cette propriété garantit que si un PROCESSUS graphique est activé (respectivement désactivé) dans la partie logique alors il sera activé (respectivement désactivé) dans la partie graphique.

Propriété 4.3.5 (Cohérence d'activation entre perspectives logique et graphique) *Les parties graphique et logique d'un PROCESSUS graphique ont toujours le même état d'activation.*

Les propriétés 4.3.4, 4.3.5, 4.3.6 et 4.3.7 garantissent que la propagation via les BINDINGS et l'activation des ASSIGNMENTS se font en fonction de leurs dépendances qui ont été préalablement définies. Plus précisément, un PROCESSUS ne doit pas propager son changement d'état s'il a un PROCESSUS prédécesseur qui n'a pas propagé son changement d'état. Dans la même idée un ASSIGNMENT ne doit pas s'activer si il en existe un le précédant et n'ayant pas été activé.

Propriété 4.3.6 (Ordre de propagation des bindings) *La propagation des changements d'état se fait en respectant les dépendances.*

Propriété 4.3.7 (Ordre d'activation des assignments) *L'activation des ASSIGNMENTS se fait en respectant les dépendances.*

Propriété 4.3.8 (Unique activation d'un processus à multiples prédécesseurs (bindings)) *Lors d'une réaction en chaîne, un PROCESSUS ayant de multiples prédécesseurs par des BINDINGS propage une seule fois son changement d'état*

Propriété 4.3.9 (Unique activation d'un processus à multiple prédécesseurs (assignments)) *Lors d'une réaction en chaîne, un PROCESSUS ayant de multiples prédécesseurs par des ASSIGNMENTS propage une seule fois son changement d'état.*

La propriété 4.3.10 garantit que chaque réaction de l'interface utilisateur suite à une interaction est finie.

Propriété 4.3.10 (Terminaison) *Toutes les réactions en chaînes terminent*

La propriété 4.3.11 garantit le déterminisme de chaque réaction de l'interface utilisateur.

Propriété 4.3.11 (Confluence) *Toutes les réactions en chaîne confluent.*

Les mécanismes d'activation/désactivation ainsi que celui de propagation sont formalisés dans la suite via des règles de réaction toutes énumérées dans la figure 4.5 donnant leur ordre d'application. La figure se lit de la manière suivante : l'ensemble de règles le plus prioritaire est le premier et celui le moins prioritaire le dernier. Les règles contenues dans un même ensemble ne sont pas comparables et peuvent donc être utilisées dans un ordre quelconque. Ainsi, les changements qu'une interaction

```
{gProcess_act, gProcess_deact},
{parent_deact},
{init_act_children,
 act_children,
 deact_children_init,
 deact_children},
{act_Pr,
temporise_assignment,
act_component,
deact_Pr,
deact_component},
{calc_prop_II,
 calc_prop_IO,
 calc_prop_OI,
 calc_prop_OO,
 calc_sync_II,
 calc_sync_OI,
 calc_sync_IO,
 calc_sync_OO},
{calc_sync_restore_old},
{src_propagating},
{multi_trg_propagating},
{trg_propagating},
{synch_to_activate(v), synch_to_deactivate},
(check_pred),
(check_pred_multi_targ_1),
(check_pred_multi_targ_2),
(no_pred),
{assign_act_targ},
(clean_check),
{propagatingP_to_idle, propagatingT_to_idle},
{idle_to_1}
```

FIGURE 4.5 – Ordre de priorité des règles de réaction. Les règles de réaction dans un même ensemble ne sont pas comparable. L'ordre de priorité sur les ensembles de règles décroît de haut en bas.

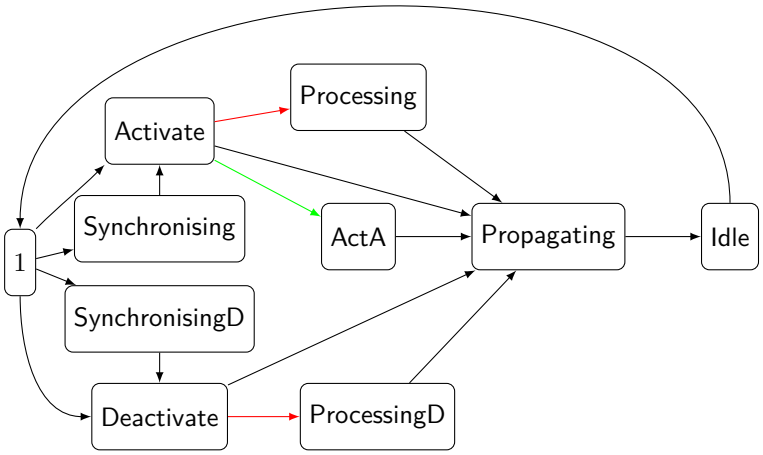


FIGURE 4.6 – Machine à état résumant les états d'exécution possibles d'un PROCESSUS. Les états reliés par les arêtes rouges sont accessibles seulement par les `Component` et ceux reliés par les arêtes vertes sont accessibles seulement par les ASSIGNMENTS.

peut produire sur une interface sont représentés dans la sémantique par l'application de ces règles de réaction sur le bigraphe la représentant. À chaque interaction utilisateur, des règles de réaction s'appliquent sur ce bigraphe afin de le modifier en cohérence.

Afin de constituer ces réactions en chaîne lorsqu'une interaction a lieu, les PROCESSUS passent par deux phases différentes décrites par les règles de réaction. La première phase, nommée phase d'activation/désactivation, est présentée dans la section 4.3.3. Cette phase se concentre sur le traitement de l'activation/désactivation d'un PROCESSUS qui diffère selon sa sémantique. La deuxième, nommée phase de propagation et qui se concentre sur la propagation d'un changement d'état de PROCESSUS, est présentée dans la section 4.3.4. Ces deux phases ne sont pas globales à la réaction i.e. il est possible que deux PROCESSUS soient dans différentes phases à un certain pas de la réaction. Dans les deux prochaines sections, l'état d'exécution d'un PROCESSUS (i.e. entité ExecState) sera au centre des intérêts car celui-ci indique dans quel phase un PROCESSUS se trouve. Les différents états d'exécution possibles pour un PROCESSUS sont énumérés par la figure 4.6 et seront expliqués dans la section 4.3.4 et la section 4.3.3. En générale, lorsqu'un PROCESSUS est enclenché par une activation (respectivement une désactivation) son état d'exécution passe toujours par les états Activate (respectivement Deactivate), pour indiquer qu'il doit être activé (respectivement désactivé), Propagating pour indiquer qu'il doit propager son changement d'état et Idle pour indiquer qu'il a fini de procéder.

Plus précisément, l'état d'exécution d'un PROCESSUS est vide initialement. Selon ses dépendances, celui-ci peut passer dans un état Activate indiquant que le PROCESSUS doit être activé, ou Deactivate, indiquant que le PROCESSUS doit être désactivé. Le PROCESSUS peut passer dans un état Syncrhonising, indiquant que le PROCESSUS est synchronisé pour une activation, ou SynchronisingD, indiquant que le PROCESSUS est synchronisé pour une désactivation, si il y a plusieurs prédécesseurs qui essayent de l'activer/désactiver. Une fois les activations/désactivations concurrentes gérées, le PROCESSUS passe dans un état Activate ou Deactivate. Dans le cas où le PROCESSUS a un état d'exécution Activate, si le PROCESSUS est un `Component` alors celui-ci va passer dans l'état Processing, indiquant que le `Component` est en train d'activer ses enfants, si c'est un ASSIGNMENT alors celui-ci va passer dans l'état TempA (comme Activate ASSIGNMENT), indiquant que l'activation de l'ASSIGNMENT est temporisée, et quant

aux autres PROCESSUS ils vont directement dans l'état Propagating. Dans le cas où le PROCESSUS est dans un état d'exécution Deactivate alors celui-ci va dans état ProcessingD, indiquant que le Component est en train de désactiver ses enfants, si c'est un Component et dans un état Propagating pour les autres cas. L'état d'exécution Propagating indique que le PROCESSUS est prêt à propager son changement d'état. Une fois la propagation faite celui-ci passe dans un état d'exécution Idle indiquant que le PROCESSUS n'a plus rien à faire jusqu'à la fin de la réaction. Une fois la réaction finie, l'état d'exécution du PROCESSUS est réinitialisée au bigraphe vide.

4.3.3 Phase d'activation/désactivation

La phase d'activation/désactivation consiste en l'activation ou la désactivation d'un PROCESSUS. L'activation (resp. la désactivation) d'un PROCESSUS est représentée par une entité de contrôle Activate (resp. Deactivate) encapsulée dans l'entité StateExec. L'activation et la désactivation dépendent de la sémantique d'un PROCESSUS. Par la suite, seules les règles concernant l'activation sont détaillées, celles concernant la désactivation sont similaires et sont données dans la figures 4.8 et 4.9.

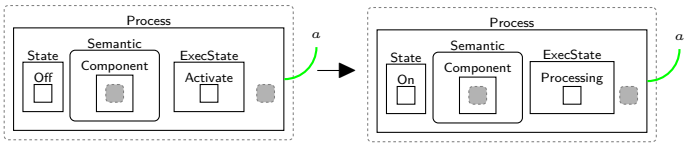
Phase d'activation : Component

Lorsque un Component s'active, celui-ci change d'état d'activation et active tout les enfants qui ont leur contrôle Init à On. Ceci se produit dans la phase d'activation d'un Component qui est donnée dans la figure 4.7. L'activation d'un Component se fait en trois étapes.

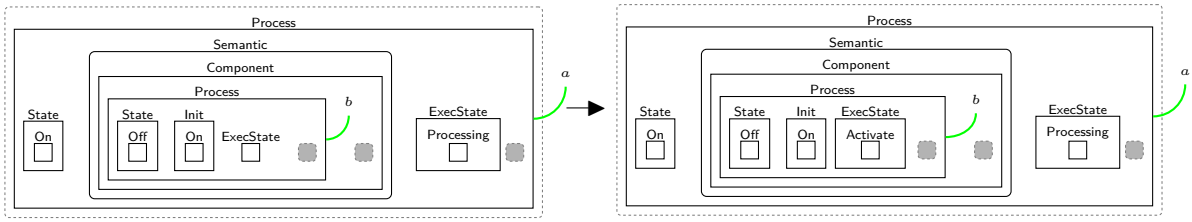
1. La première étape décrite par la figure 4.7a consiste à activer un Component éteint et qui doit être activé (état d'exécution à Activate). Pour cela son état d'activation passe de Off à On et son état d'exécution passe à Processing. L'état d'exécution Processing signifie que le Component doit activer ses PROCESSUS enfants avant de propager son changement d'état.
2. La figure 4.7b décrit la deuxième étape qui correspond à l'activation des enfants d'un Component. Cette règle s'applique seulement si un PROCESSUS Component est allumé, son état d'exécution fixé à Processing et son PROCESSUS enfant éteint, a un état d'exécution vide et peut être activé par son parent, ce qui est indiqué par son entité Init qui encapsule une entité On. L'état d'exécution de l'enfant vide assure l'activation de PROCESSUS qui n'ont jamais été activés pendant la courante réaction en chaîne. Plus d'informations seront donnés à ce sujet dans la section 4.3.4. Si toutes les conditions sont satisfaites, la règle de réaction change l'état d'exécution de l'enfant en Activate. L'ordre de priorité sur les règles de réaction défini dans la figure 4.5, permet d'appliquer cette règle jusqu'à ce que tout les enfants du Component soient activés. Une fois que tous les enfants d'un Component ont été activés, le Component peut enfin propager son activation.
3. Par l'ordre de priorité défini sur les règles de réaction, la règle illustrée dans la figure 4.7c permet de faire passer l'état d'exécution d'un Component allumé de Processing à Propagating une fois que tous ses enfants ont été traités. L'état d'exécution Propagating d'un PROCESSUS intervient après son exécution et signifie qu'il est prêt à propager son changement d'état.

Phase d'activation : Assignment

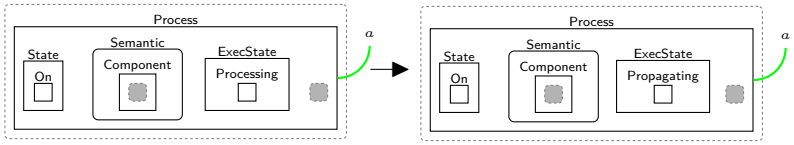
La phase d'activation d'un ASSIGNMENT nécessite un traitement particulier comparée à celle des autres PROCESSUS. En effet, l'ASSIGNMENT est l'unique PROCESSUS qui interagit avec la mémoire allouée au programme. Afin d'assurer un ordre de lecture-écriture en mémoire cohérent, l'ASSIGNMENT



(a) Initialisation de l'activation d'un `Component` et de ses enfants (init_act_children).

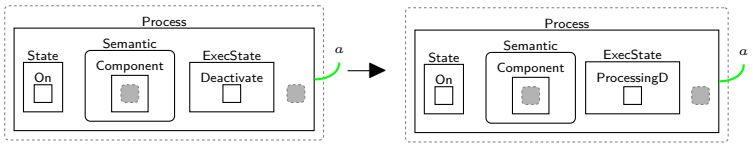


(b) Activation des enfants d'un `Component` (act_children).

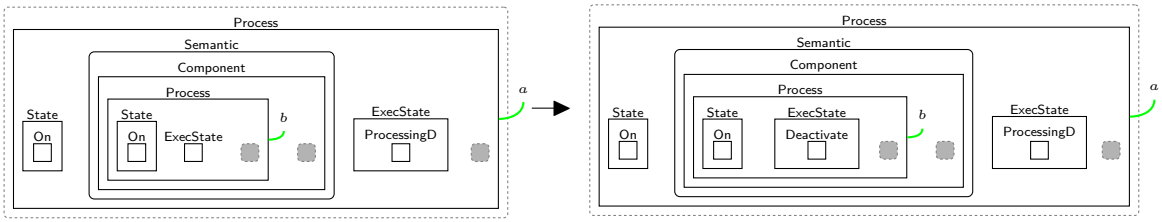


(c) Activation d'un `Component` act_component.

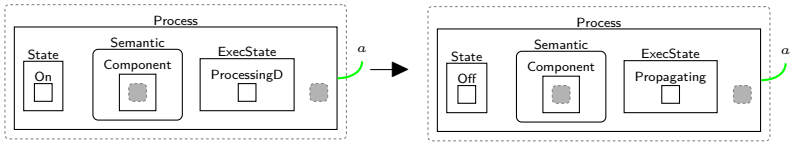
FIGURE 4.7 – Activation d'un `Component`.



(a) Initialisation de la désactivation d'un `Component` et de ses enfants (deact_children_init).



(b) Désactivation des enfants d'un `Component` (deact_children).



(c) Désactivation d'un `Component` (deact_component).

FIGURE 4.8 – Désactivation d'un `Component`.

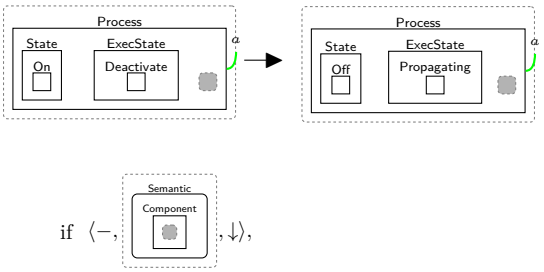


FIGURE 4.9 – Désactivation d'un PROCESSUS (deact_Pr).

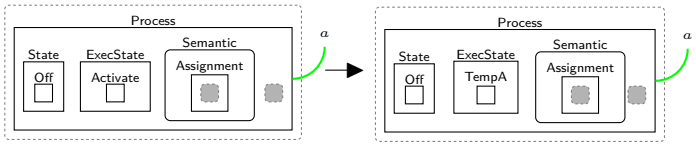


FIGURE 4.10 – Temporisation de l'assignation (temporise_assignment).

s'exécute et propage son changement d'état lorsque toutes les activations/désactivations des PROCESSUS (différent des ASSIGNMENTS) ont été traités. De plus et pour la même raison de cohérence en mémoire, les ASSIGNMENTS doivent s'activer selon leur ordre de dépendance. Une analyse préliminaire sur les ASSIGNMENTS est effectuée afin de déterminer si leurs dépendances permettent de les activer. Pour cela lorsque un ASSIGNMENT doit être activé, la règle décrite dans figure 4.10 passe son état d'exécution à l'état TempA. L'état TempA est un état intermédiaire avant l'état Propagating. Les règles de réaction traitant un ASSIGNMENT dans cet état sont moins prioritaires que toutes les autres règles traitant le changement d'état de tout autre PROCESSUS. Cela permet de temporiser l'exécution/la propagation d'un ASSIGNMENT et de le traiter après tout autre PROCESSUS, ce qui permet, rappelons le, d'avoir un ordre de lecture-écriture en mémoire cohérent.

À l'instant où un ASSIGNMENT peut être activé, aucun autre PROCESSUS, mis à part d'autres ASSIGNMENTS, n'est en attente d'activation. Afin de respecter l'ordre de dépendance parmi les ASSIGNMENTS en attente d'activation, une vérification doit être faite préalablement. Les ASSIGNMENTS en attente forment un sous-graphe du graphe de communications (section 4.2.3) du programme. En conséquence, deux cas de dépendance se présentent : 1) un ASSIGNMENT peut avoir la même cible qu'un autre ASSIGNMENT ; 2) un ASSIGNMENT peut avoir pour cible la source d'un autre ASSIGNMENT. Pour le premier cas, nous disons que les ASSIGNMENT sont en concurrence et pour le second, nous disons qu'un ASSIGNMENT qui a pour cible la source d'un autre ASSIGNMENT, le précède.

Le but de cette vérification est d'activer les ASSIGNMENTS selon leur ordre de dépendance à travers plusieurs étapes ordonnées qui sont définies par des règles de réaction. Ces étapes sont ordonnées grâce à l'ordre de priorité sur les règles de réaction. L'idée est de faire un filtre sur l'ensemble des ASSIGNMENTS en attente à chaque étape pour que seuls ceux dont les dépendances le permettent puissent être activés.

La première étape décrite par la figure 4.11, consiste à encapsuler une entité de contrôle F (pour False) dans l'entité TempA d'un ASSIGNMENT en attente d'activation qui a son entité Source connectée à l'entité Target d'un autre ASSIGNMENT en attente d'activation. Une connexion entre une entité Source et une entité Target survient uniquement si ces entités sont connectées au même PROCESSUS. Cette étape permet de différencier tout les ASSIGNMENT en attente qui ont au moins un prédécesseur.

Après cette première étape, seuls les ASSIGNMENTS sans prédécesseurs peuvent être activés. Cependant, il reste un dernier filtrage à effectuer avant d'activer un quelconque ASSIGNMENT. Il est

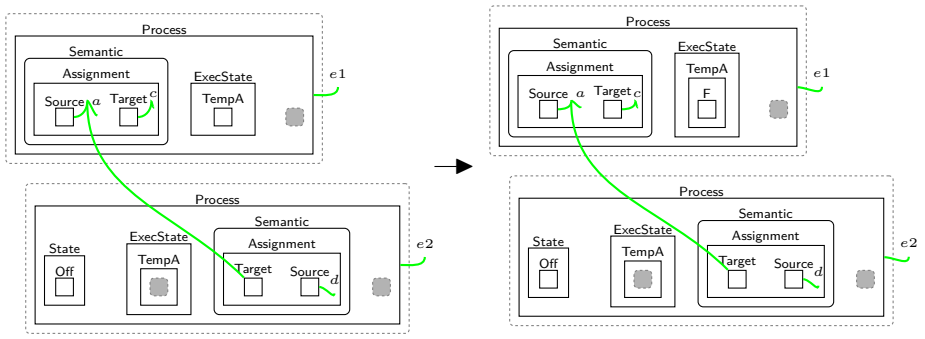


FIGURE 4.11 – Vérification de la présence de prédécesseurs pour un ASSIGNMENT (check_pred).

possible que des ASSIGNMENTS sans prédécesseurs soient en concurrence avec des ASSIGNMENTS ayant un prédécesseur. Il faut donc filtrer ces ASSIGNMENTS pour qu'ils soient activés en même temps que leur ASSIGNMENTS concurrent. La règle de réaction présentée dans la figure 4.12a permet d'effectuer ce filtrage. Dans cette règle, si un ASSIGNMENT non-filtré et en attente d'activation est en concurrence avec un ASSIGNMENT filtré, deux ASSIGNMENTS en concurrence sont dissociable par leurs entités Target connectées, alors l'ASSIGNMENT est filtré en encapsulant une entité F dans son entité TempA.

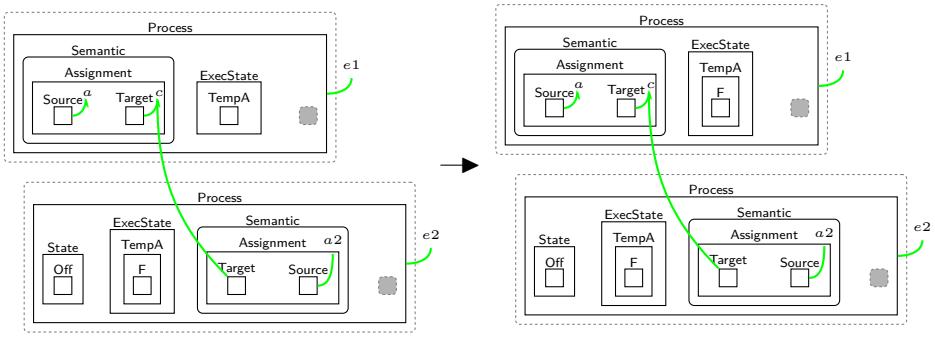
Il est maintenant possible de commencer à activer les ASSIGNMENTS en commençant par ceux en concurrence. Lors de son activation, l'ASSIGNMENT doit aussi activer sa PROPERTY cible. Pour éviter une multiple activation de cette PROPERTY, la règle présentée dans la figure 4.12b va activer l'un des ASSIGNMENTS en concurrence, en passant son état à On, et le faire passer en phase de propagation, en passant son état d'exécution à Propagating. En réitérant cette règle sur l'ensemble d'ASSIGNMENTS en attente, seuls des ASSIGNMENTS indépendants des autres, étant prêts à être activés, sont contenus dans l'ensemble. Pour activer les ASSIGNMENTS restants, la règle dans la figure 4.13 encapsule dans toutes les entités TempA vides une entité T (pour True) indiquant que l'ASSIGNMENT est autorisé à être activé.

L'activation d'un ASSIGNMENT ainsi que sa cible est décrite dans la figure 4.14. Dans cette règle, si un ASSIGNMENT est autorisé à être activé (indiqué par l'entité T contenue dans l'entité TempA de l'ASSIGNMENT) et que le `Spike` de sa PROPERTY cible n'a pas été activé dans cette réaction en chaîne (indiqué par son entité TempA vide), alors l'ASSIGNMENT est activé, son état passe à On et son état d'exécution passe à Propagating, et le `Spike` de sa cible est enclenché pour être activé, ce qui est indiqué par son état d'exécution passant à Activate.

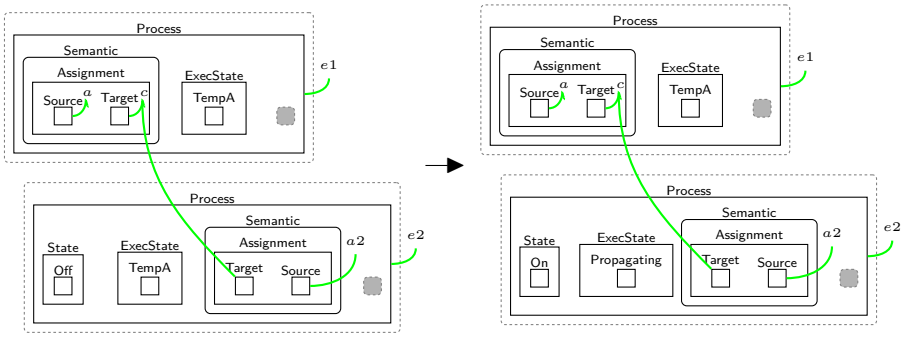
Une fois que tous les ASSIGNMENTS ayant obtenu l'autorisation d'activation ont été activés, il ne reste plus que les ASSIGNMENTS qui ont été filtrés à traiter. Pour cela, la règle décrite dans la figure 4.15 retire l'entité F de leur entité TempA. Ainsi, un nouvel ensemble d'ASSIGNMENTS en attente d'activation est formé et peut passer à travers les étapes décrites précédemment. Cette procédure est réitérée jusqu'à qu'il n'y ait plus aucun ASSIGNMENT à traiter.

Phase d'activation pour un processus quelconque

Enfin, la phase d'activation des autres PROCESSUS est traitée avec la règle présenté dans figure 4.16. Cette règle est conditionnelle et ne s'applique pas sur les `Component` et les ASSIGNMENTS. Cette règle passe l'état de tous les PROCESSUS qui sont dans un état d'exécution Activate à On et les autorise à passer en phase de propagation en passant leur état d'exécution à Propagating. Dans la figure, les conditions sont exprimés comme un triplet avec le symbole − correspondant à la négation et le symbole ↓ correspondant au contexte de la partie gauche de la règle. Une règle conditionnelle se lit



(a) cas 1 (check_pred_multi_targ_1).



(b) cas 2 (check_pred_multi_targ_2).

FIGURE 4.12 – Vérification de concurrence entre ASSIGNMENT.

FIGURE 4.13 – L'ASSIGNMENT est autorisé à être activé (no_pred)).

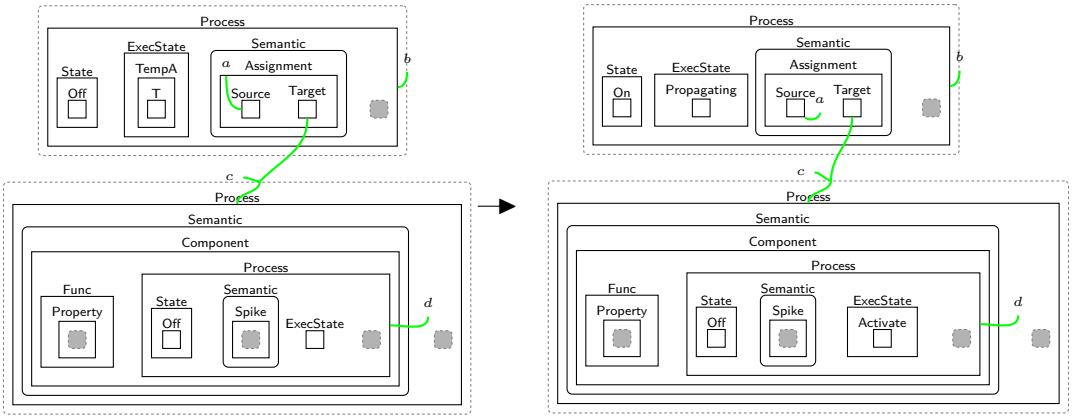


FIGURE 4.14 – Activation de l'ASSIGNMENT (assign_act_targ).

FIGURE 4.15 – L'entité permettant l'analyse de prédécesseurs est réinitialisée (clean_check).

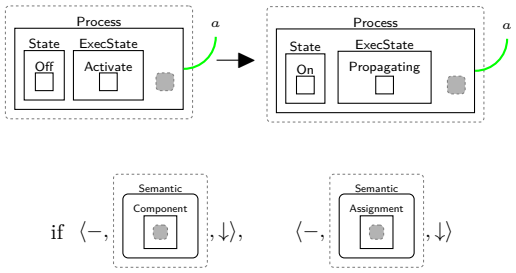


FIGURE 4.16 – Activation d'un PROCESSUS (act_Pr).

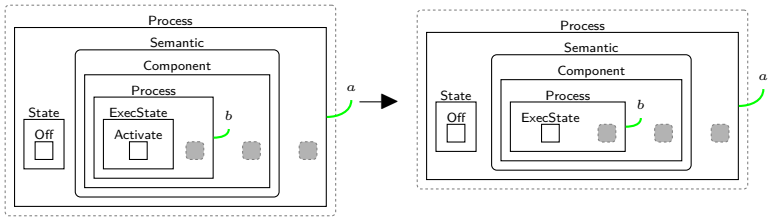


FIGURE 4.17 – Règle empêchant l'activation d'un PROCESSUS si son parent est désactivé (parent_deact).

donc, si le bigraphe, de la deuxième composante du triplet, n'est pas présent en contexte alors la règle est applicable. Si la condition est formée de plusieurs triplets alors ils doivent être tous satisfaits pour que la règle soit appliquée.

Cohérence de la phase d'activation

Les règles précédemment décrites formalisent l'activation de tous les PROCESSUS dans *Biguil*. Cependant, il reste un critère à vérifier avant d'activer un PROCESSUS pour préserver la cohérence de la sémantique. Un PROCESSUS peut être activé seulement si son parent est activé. La figure 4.17 présente une règle qui est appliquée avant chaque activation de PROCESSUS pour vérifier que cette propriété sera vérifiée. Cette règle a pour action de retirer l'entité Activate d'un PROCESSUS si son parent n'est pas activé.

Enfin, les règles définies dans les figure 4.18 et figure 4.19 complètent l'activation/désactivation d'un `GComponent` en activant/désactivant le GProcess représentant le `GComponent` dans la perspective graphique.

Une fois qu'un PROCESSUS est activé, celui-ci entre en phase de propagation, décrite dans la prochaine section, pour propager son changement d'état.

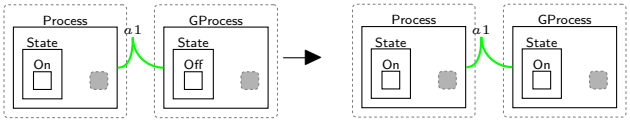


FIGURE 4.18 – Activation d'un PROCESSUS graphique (gProcess_act).

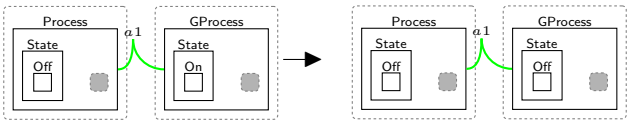


FIGURE 4.19 – Desactivation d'un PROCESSUS graphique (gProcess_deact).

4.3.4 Phase de propagation

La phase de propagation d'un PROCESSUS consiste en la propagation de son changement d'état vers les PROCESSUS qui lui sont connectés en tant cible via un BINDING. Durant sa phase de propagation, un PROCESSUS informe tous les BINDINGS auxquels il est connecté en tant que source de son changement d'état pour permettre aux BINDINGS de changer en conséquence l'état de leur PROCESSUS cible.

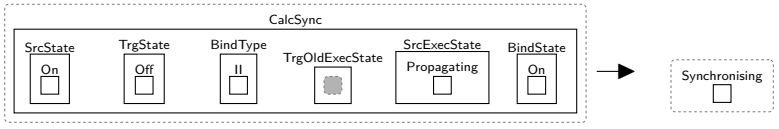
Pour que le programme réagisse à des interactions, le développeur définit des liens de causalité, à travers les BINDINGS, entre des PROCESSUS représentant des interactions et les autres PROCESSUS (graphiques ou non) de l'interface graphique. Ainsi, il est possible que le développeur définisse une interaction tel qu'un clic de souris enclenchant l'apparition d'un composant graphique sur l'interface. Pour cela il reliera via un BINDING un PROCESSUS représentant le clic de souris à un PROCESSUS représentant le composant graphique concerné. Comme dit dans la section 4.2.3, ces liens de causalité sont représentés par le graphe de communication. Lorsqu'un changement d'état survient sur un PROCESSUS, sa propagation est contrainte par le graphe de communication. En effet, de la même manière que l'activation des ASSIGNMENTS, la propagation d'un changement d'état d'un PROCESSUS doit se faire en respectant un certain ordre défini par les dépendances du PROCESSUS. Un PROCESSUS ne doit pas faire de propagation avant qu'un de ses prédécesseurs l'ait fait. Ici, le prédécesseur d'un PROCESSUS est un PROCESSUS qui est connecté en tant que source à celui-ci via un BINDING. De plus, en cas de concurrence (PROCESSUS cible ayant de multiples prédécesseurs), le PROCESSUS cible doit être activé ou désactivé une seule fois et donc un choix, selon le contexte, doit être fait sur la propagation.

Lorsqu'un PROCESSUS a son état d'exécution à Propagating, celui-ci va passer à travers plusieurs étapes, ordonnées dans l'ordre présenté grâce à l'ordonnancement sur les règles de réaction, pour propager son changement d'état à travers les BINDINGS. Ces étapes sont axées sur des règles de réaction qui vont déduire les états d'exécution des PROCESSUS cibles à l'issue de la propagation du changement d'état. Il existe deux types de règles de calcul, le premier couvre les propagations concurrentes sur une cible et le second couvre les propagations simples.

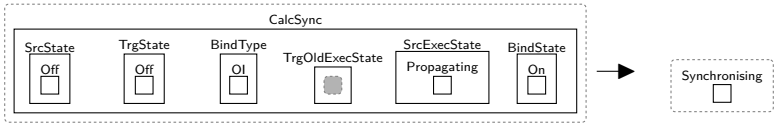
La figure 4.20 rassemble toutes les règles de réaction permettant de calculer l'état d'exécution d'un PROCESSUS cible dans le cas où celui-ci a de multiples prédécesseurs. Pour calculer l'état d'exécution d'un PROCESSUS cible, ces règles ont besoin :

- que l'état d'activation du BINDING soit à On pour assurer que la propagation peut être faite,
- du type du BINDING utilisé, qui va indiquer si le PROCESSUS cible doit être activé (indiqué par l'entité Synchronising) ou désactivé (indiqué par l'entité SynchronisingD) après que les prédécesseurs du PROCESSUS cible se sont synchronisés pour propager leur changement d'état,
- de l'état d'activation du PROCESSUS source ainsi que celui du PROCESSUS pour vérifier si ils sont compatibles avec le type du BINDING et en conséquence vérifier que la propagation peut être faite,
- de l'état d'exécution du PROCESSUS cible pour assurer que le changement d'état a bien eu lieu durant la réaction en chaîne courante, et
- de l'état d'exécution courant du PROCESSUS cible, qui va être restauré si toutes les conditions ne sont pas réunies pour la propagation du changement d'état (figure 4.20e).

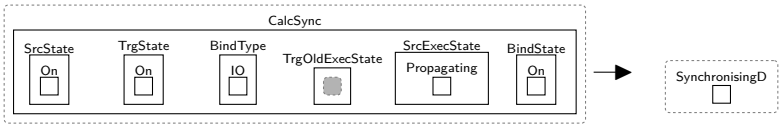
Dans le cas où le PROCESSUS cible n'a qu'un seul prédécesseur, la figure 4.21 rassemble toutes les



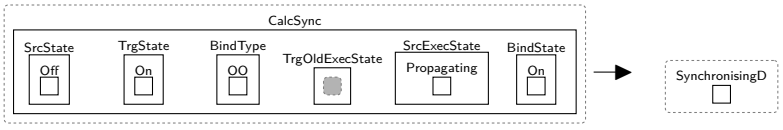
(a) Condition de synchronisation pour un BINDING II (calc_sync_II).



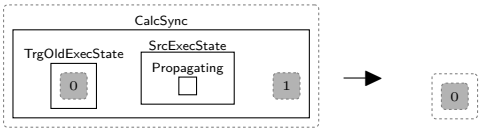
(b) Condition de synchronisation pour un BINDING OI (calc_sync_OI).



(c) Condition de synchronisation pour un BINDING IO (calc_sync_IO).

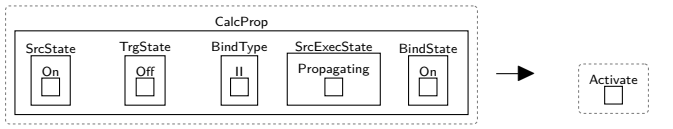


(d) Condition de synchronisation pour un BINDING OO (calc_sync_OO).

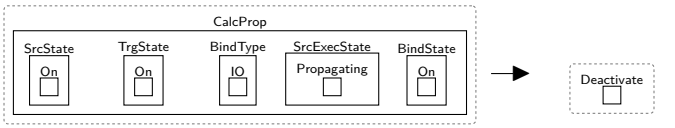


(e) Condition de synchronisation lors de mauvaises conditions réunis (calc_sync_restore_old).

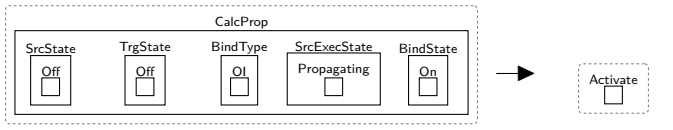
FIGURE 4.20 – Règles de réaction calculant l'état d'exécution de la cible d'un BINDING, dans le cas de propagations concurrentes.



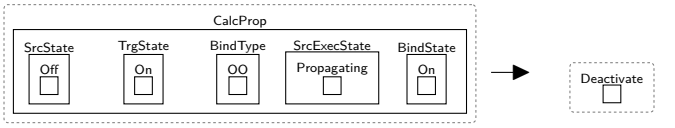
(a) Condition d'activation pour un BINDING II (calc_prop_II).



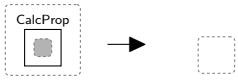
(b) Condition d'activation pour un BINDING IO (calc_prop_IO).



(c) Condition d'activation pour un BINDING OI (calc_prop_OI).



(d) Condition d'activation pour un BINDING OO (calc_prop_OO).



(e) Condition d'activation pour un PROCESSUS non activé (calc_prop_idle).

FIGURE 4.21 – Règles de réaction calculant l'état d'exécution de la cible d'un BINDING, dans le cas d'une propagation simple.

règles de réaction pour calculer son état d'exécution. Pour ce calcul, ces règles ont besoin, comme pour les règles de la figure 4.20 :

— de l'état d'activation ainsi que le type du BINDING,
— de l'état d'activation du PROCESSUS source et du PROCESSUS cible, et
— de l'état d'exécution du PROCESSUS cible.

Contrairement à la figure 4.20, ces règles n'ont pas besoin de l'état d'exécution courant du PROCESSUS cible car si les conditions d'une propagation ne sont pas réunies alors l'entité vide (son état d'exécution par défaut) est restaurée dans l'état d'exécution du PROCESSUS cible (figure 4.21e).

Afin de pouvoir utiliser les règles de calcul présentées dans les figures 4.20 et 4.21, la première étape présentée dans la figure 4.22 consiste à propager les informations nécessaires, venant du PROCESSUS source, pour le calcul. Cette propagation d'information est faite seulement si le PROCESSUS source a pour état d'exécution Propagating et si le BINDING concerné est allumé et n'a jamais été utilisé dans la réaction en chaîne courante, ce qui est indiqué par l'état d'exécution du BINDING vide. Si ces conditions sont respectées alors le transfert d'information se fait à travers l'entité srcPropagating, elle même contenue dans l'entité execState, qui va contenir l'état d'activation et d'exécution du PROCESSUS source.

La propagation des informations d'un PROCESSUS source s'effectue sur tous les BINDINGS auxquels il est connecté. Une fois les informations propagées et selon le type du PROCESSUS source, une des deux règles présentées par les figures 4.23 et 4.24 est appliquée afin de passer son état d'exécution à Idle et indiquer son inactivité pour le reste de la réaction en chaîne.

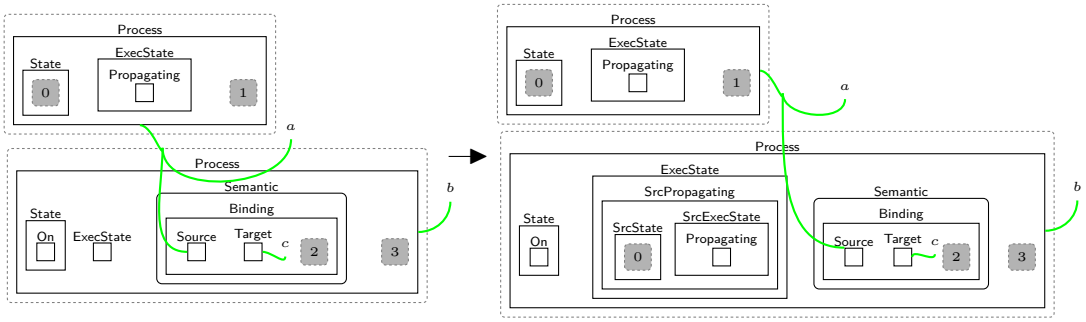


FIGURE 4.22 – Propagation du changement d'état d'une source à un BINDING (src_propagating).

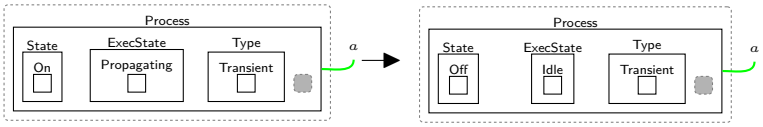


FIGURE 4.23 – Changement d'état d'exécution d'un PROCESSUS Transitoire : de l'état Propagating à un état inactif représenté par l'entité Idle (propagatingT_to_idle).

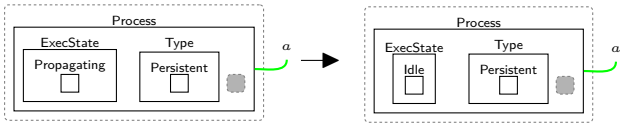


FIGURE 4.24 – Changement d'état d'exécution d'un PROCESSUS Persistant : de l'état Propagating à un état inactif représenté par l'entité Idle (propagatingP_to_idle).

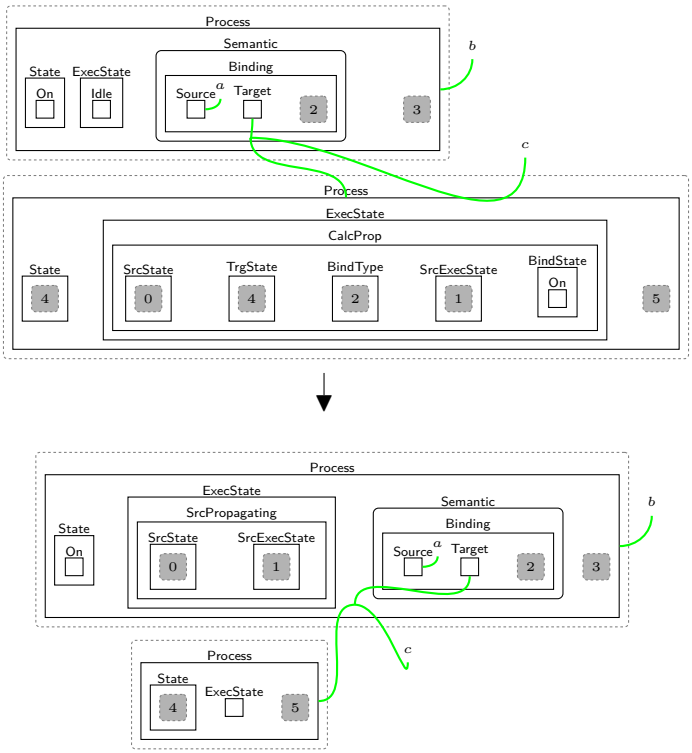


FIGURE 4.25 – Propagation du changement d'état d'une source à la cible du BINDING (trg_propagating).

Une fois que le PROCESSUS source a propagé les informations nécessaires pour la propagation deux cas de figure se présentent.

Le premier cas, présenté dans la figure 4.25, consiste en la propagation des informations d'un PROCESSUS source dans le cas où il n'y a pas de concurrence. La règle de réaction décrite s'applique sur un PROCESSUS ayant un état d'exécution vierge et connecté en tant que cible à un BINDING allumé contenant les informations d'un PROCESSUS source. Dans ce contexte, la règle modifie l'état d'exécution du BINDING en Idle, il ne sera alors plus utilisable durant cette réaction en chaîne, et va encapsuler, dans l'entité CalcProp, toutes les informations nécessaires aux règles décrites dans la figure 4.21 pour que l'état d'exécution du PROCESSUS cible soit calculé.

Le second cas, présenté dans la figure 4.26, consiste en la propagation concurrente des informations des PROCESSUS sources. La règle qui y est décrite s'applique sur un PROCESSUS qui est cible de deux BINDINGS différents allumés. Le principe de cette règle est d'éliminer à chacune de ses applications un BINDING contenant les informations de propagation d'un PROCESSUS source (contenus dans l'entité SrcPropagating). Les informations nécessaires au calcul de l'état d'exécution de la cible sont alors encapsulées dans une entité CalcSync fraîchement créée et contenue dans son état d'exécution, et l'état d'exécution du BINDING est passé à Idle pour que celui-ci ne soit plus utilisable pour le restant de la réaction en chaîne. Après l'application de cette règle, une des règles de la figure 4.20 est utilisée pour calculer l'état d'exécution du PROCESSUS cible. Une fois tout les BINDINGS concurrents traités, trois cas sont possible.

1. Il est possible que l'état d'exécution de la cible soit vide, dans ce cas le PROCESSUS reste inchangé et cela signifie qu'aucune condition de propagation n'a été satisfaite pour l'ensemble

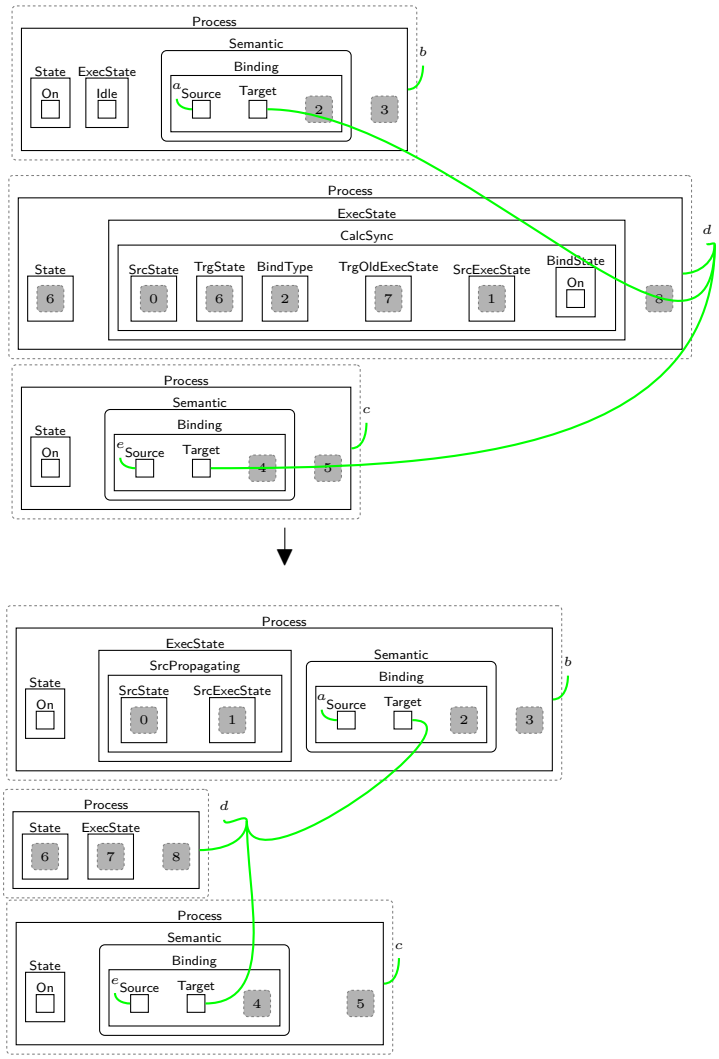


FIGURE 4.26 – Propagation du changement d'état d'une source : cas de cibles multiples (multi_trg_propagating).

des BINDINGS en concurrence.

2. Si l'état d'exécution de la cible contient une entité Synchronising, alors la règle de réaction décrite dans la figure 4.27 s'applique. Celle-ci remplace l'entité Synchronising par une entité Activate de la cible pour que le PROCESSUS commence sa phase d'activation. Les éléments non expliqués de la règle permettent de vérifier qu'un PROCESSUS ayant de multiples prédécesseurs est activé une seule fois, cela sera expliqué plus en détail dans le chapitre 6.
3. Si l'état d'exécution de la cible contient une entité SynchronisingD, le traitement est similaire au cas précédent à l'exception que le PROCESSUS cible rentre dans sa phase de désactivation (figure 4.28).

Lorsque toutes les activations/désactivations et propagations ont été traitées, la règle présentée dans figure 4.29 se charge de réinitialiser l'état d'exécution de tous les PROCESSUS ayant participé à la réaction en chaîne. Cela pourra permettre de traiter une nouvelle interaction utilisateur.

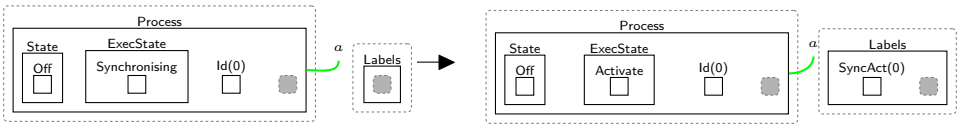


FIGURE 4.27 – Activation d'un PROCESSUS en synchronisation (synch_to_activate).

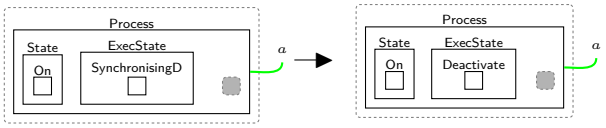


FIGURE 4.28 – Désactivation d'un PROCESSUS en synchronisation (synch_to_deactivate).

Nous terminons cette section en présentant les figures 4.31 et 4.32 qui correspondent à l'exécution du modèle *Biguil*, en figure 4.30. Le modèle est minimaliste, il s'agit de la définition de deux Spikes mis en relation à travers un BINDING. Ainsi, l'activation du Spike `p1` enclenche l'activation du Spike `p2`. Les figures 4.31 et 4.32 illustrent l'exécution du modèle lorsque l'activation du Spike `p1` est enclenchée. La figure 4.31a représente le moment où le Spike `p1` est enclenché. Puis, la figure 4.31b illustre l'activation du Spike `p1` par application de la règle `act_Proc`. Ensuite vient l'application de la règle `src_propagating`, en figure 4.31c, qui permet d'engendrer la propagation de l'état et l'état d'exécution du Spike source à son BINDING. Dans la figure 4.32a, ces informations ainsi que l'état et le type du BINDING sont encapsulés dans une entité CalcProp qui est transmise au Spike cible. Ces informations vont permettre l'application de la règle `calc_prop_II` qui passe l'état d'exécution du Spike cible à Activate, comme l'illustre la figure 4.32b. La figure 4.32c illustre l'activation du Spike cible après l'application de la règle `act_Proc`.

4.4 Conclusion

Dans ce chapitre nous avons présenté *Biguil*, un UIDL pour modéliser les interfaces graphiques, ainsi que sa sémantique formalisée avec les bigraphes. Une interface graphique décrite avec *Biguil* est toujours accompagnée d'un BRS équivalent le représentant. Ce BRS permet de simuler le comportement de l'interface graphique lorsqu'une interaction utilisateur a lieu. Chaque composant graphique et interactif de l'interface est abstrait par un PROCESSUS. Lors d'une interaction utilisateur, le PROCESSUS représentant l'interaction entre en phase d'activation pour s'activer, puis en phase de propagation pour faire savoir à toutes ses dépendances qu'il est activé en provoquant ainsi une réaction en chaîne.

Les règles de réaction présentés dans la section 4.3.2 définissent la réaction de l'interface graphique suite à une telle interaction. Il est possible que pendant la réaction, différentes règles soient applicable sur le bigraphe définissant l'état de l'interface graphique à l'instant. Cependant, la sémantique donnée couvre seulement l'aspect structurel et événementiel des UIDLs. Donc, les machines à état générées par le BRS, représentant les étapes de réaction de l'interface graphique suite à une interaction, confluent

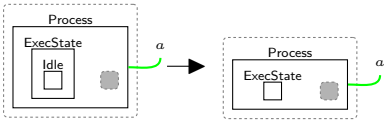


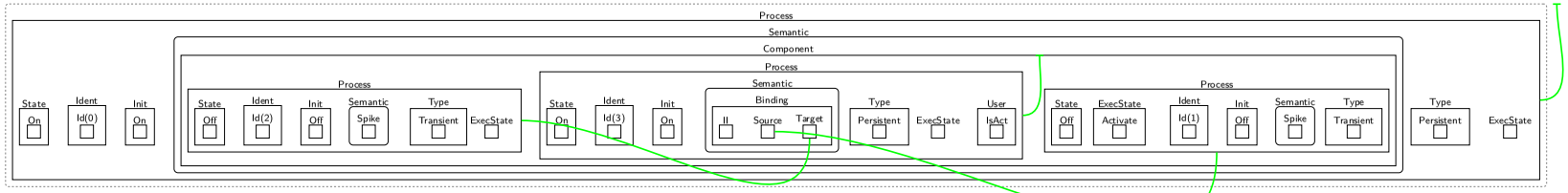
FIGURE 4.29 – Changement d'état d'exécution : de Idle à 1 (idle_to_1).

```
1     root : Component<a> {
2          p1 : Spike <d>
3          p2 : Spike <d>
4          b1 : p1 -><a> p2
5     }
```

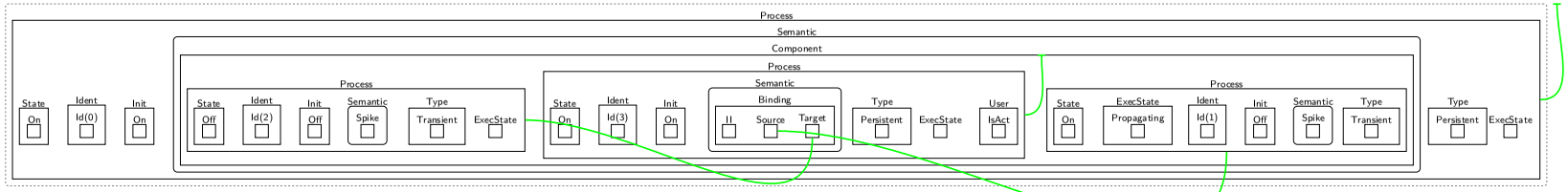
FIGURE 4.30 – Code représentant l'exécution des figures 4.31 et 4.32.

toujours, ce résultat reste cependant à démontrer.

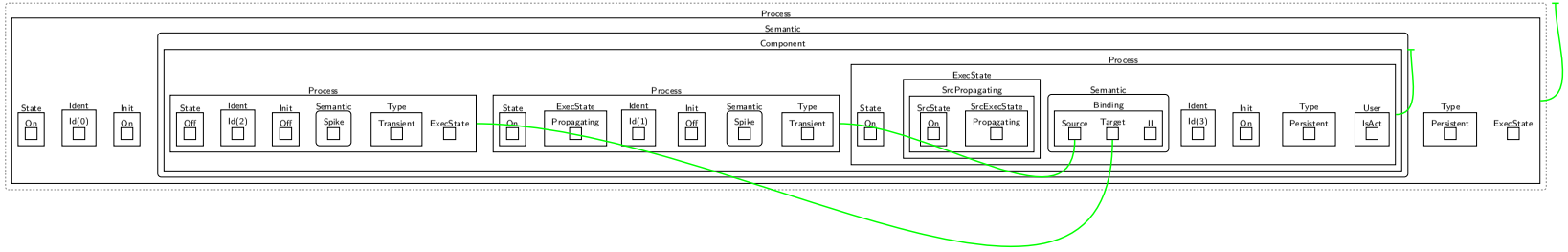
La sémantique donnée dans ce chapitre, couvre un ensemble de propriété garantissant le bon fonctionnement d'une interface graphique. Dans le prochain chapitre, nous compilons le langage SMALA vers *Biguil* afin de faire profiter aux interfaces SMALA, cet aspect. De plus, nous y illustrons le pouvoir expressif, au sens spatio-interactif, de notre UIDL au travers des différentes passes de compilation qui montrent l'implémentation de concepts SMALA en *Biguil*.



(a) Le PROCESSUS source `p1` a été enclenché.

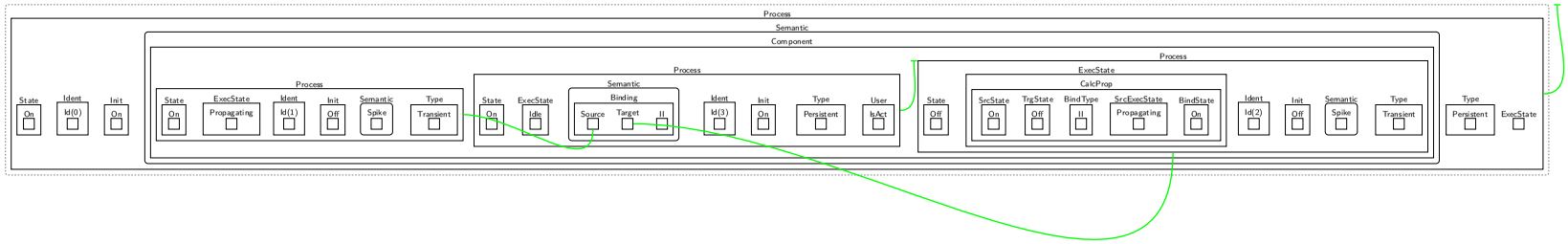


(b) La règle act_Proc a été appliquée. Le PROCESSUS source a été activé, il est maintenant prêt à propager son changement d'état.

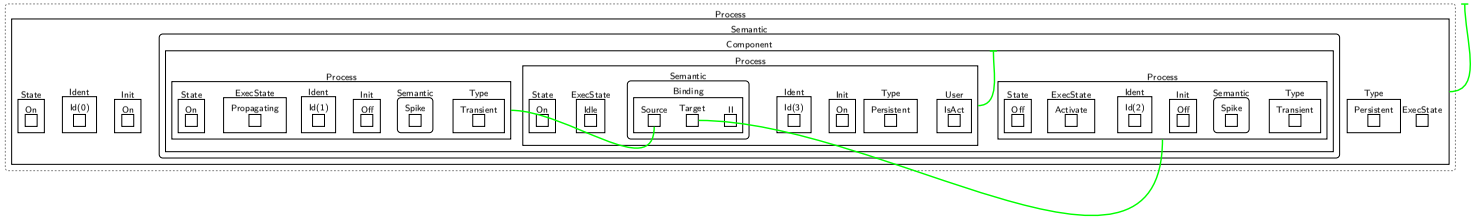


(c) La règle src_propagating a été appliquée. L'état et l'état d'exécution de la source ont été transmis au BINDING.

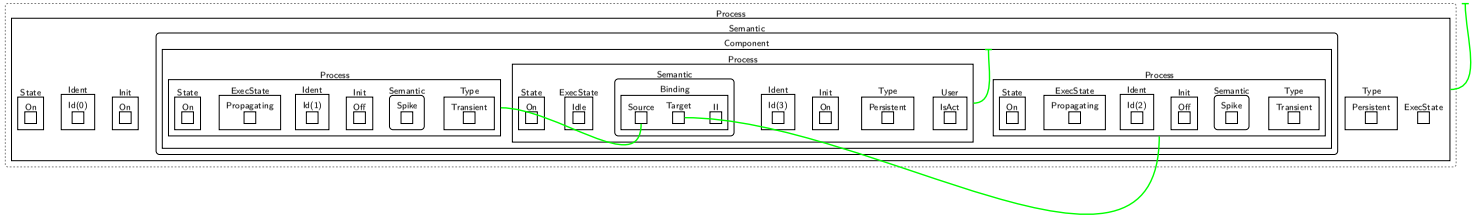
FIGURE 4.31 – Exécution programme – partie 1.



(a) La règle trg_propagating a été appliquée. L'état et l'état d'exécution de la source, l'état du BINDING ainsi que son type et l'état de la cible ont été encapsulés dans l'entité CalcProp pour calculer l'état d'exécution du Spike cible.



(b) La règle calc_prop_II a été appliquée. L'état d'exécution du Spike cible est passé à Activate.



(c) La règlé act_Proc a été appliquée. Le Spike cible est maintenant activé.

FIGURE 4.32 – Exécution programme – partie 2.

Chapitre 5

De Smala vers *Biguil*

Les langages intermédiaires sont utilisés à des fins d'analyse ou d'optimisation des programmes compilés offrant ainsi un cadre adapté pour des opérations à effectuer ciblées. Dans notre contexte, nous avons défini *Biguil* afin d'avoir un cadre théorique permettant de raisonner sur les interfaces utilisateur que nous souhaitons vérifier. Afin de tester l'expressivité de notre UIDL, nous décidons d'utiliser *Biguil* en tant que langage intermédiaire et de compiler le langage Smala vers celui-ci. Nous testons l'expressivité de *Biguil*, en redéfinissant différents concepts venant du langage Smala. Cette démarche a aussi pour avantage de faire profiter du coté formel de *Biguil* aux interfaces descriptibles par le sous-ensemble de Smala que nous implémentons. Nous compilons Smala vers *Biguil* en sept passes et chacune d'elle traite un concept différent de Smala. Ces traitements consistent en la définitions du concept ciblé de Smala avec des concepts tirés de *Biguil*. Nous décrivons chacune de ces transformations dans une section différente. La section 5.1 décrit la première transformation permettant de s'affranchir des chemins absolus, un style de notation proposé par Smala qui engendre la création d'alias pour les noms de processus. La deuxième transformation, décrite dans la section 5.2, permet d'annoter tous les processus d'un programme dont en particulier les processus anonymes tels que les bindings, les connectors ou bien les assignments. Dans Smala l'activation d'un processus par son parent est implicite. La passe suivante, décrite dans la section 5.3, rend cela explicite. La section 5.4 décrit comment les machines à états de Smala sont encodées en utilisant des Components et des bindings. La cinquième passe, décrite dans la section 5.5, consiste à retirer le sucre syntaxique destiné à simplifier l'utilisation des opérateurs arithmétiques dans Smala. Dans Smala, de nombreux processus possèdent des processus enfants implicites. La section 5.6 décrit la sixième passe qui rend explicite les enfants de chaque processus Smala. La dernière passe, présentée en section 5.7, consiste à encoder les connectors en utilisant des bindings et des assignments.

5.1 Supression des chemins absolus

Smala permet d'appeler ses processus par leur chemin relatif ou absolu. Par exemple, un processus `p2` qui a pour parent un processus `p1`, peut être appelé soit `p2` soit `p1.p2` selon le scope de l'appel. *Biguil* a pour but de rester minimal et ne partage donc pas cette caractéristique afin de s'affranchir de toute complexité liée à la gestion des noms de ses processus. Ainsi dans *Biguil*, chaque processus défini un identifiant unique et cela n'empêche pas la compilation de code Smala. L'objectif de cette passe est de renommer chaque processus Smala par un identifiant unique. Pour cela, comme représenté dans figure 5.1, nous remplaçons chaque identifiant de processus Smala par son chemin absolu.

```
_main_
Component root {
  Frame f ("Frame",0,0,400,600)
  Circle c (20,20,10)
}
```

(a) Programme SMALA.

```
_main_
Component root {
  Frame root_f ("Frame"
      ,0,0,400,600)
  Circle root_c (20,20,10)
}
```

(b) Représentation du programme après la première passe.

FIGURE 5.1 – Passe 1 : Renommage des PROCESSUS par leur chemin absolu.

```
_main_
Component root {
 Frame f ("Frame",0,0,400,600)
 Exit e (0,1)
 e -> f.close
}
```

(a) Programme SMALA.

```
_main_
root:Component root {
  root_f:Frame ("Frame"
      ,0,0,400,600)
  root_e:Exit (0,1)
  bind_0:root_e  -> root_f_close
}
```

(b) Représentation du programme après la deuxième passe.

FIGURE 5.2 – Passe 2 : Annotation des PROCESSUS.

5.2 Ajout d'annotation

Dans *Biguil* tous les PROCESSUS doivent posséder un identifiant, ce qui permet de manipuler leur état d'activation. Ceci n'est pas le cas dans SMALA. Afin de compiler un programme SMALA vers le langage *Biguil*, un identifiant doit être attribué à tous les PROCESSUS anonymes de SMALA i.e. les CONNECTORS, les BINDINGS et les ASSIGNMENTS. Pour cela dans cette deuxième passe, comme illustré dans figure 5.2, un identifiant unique est affecté à tous les PROCESSUS du programme.

5.3 Explicitation de l'activation initiale

Dans SMALA tous les PROCESSUS ne peuvent pas être activés par leur Component parent : cela dépend du PROCESSUS en question. Par exemple, certains PROCESSUS comme les Spikes ne sont pas activés par leur parent et d'autres, comme les ASSIGNMENTS, le sont. Pour savoir si un PROCESSUS SMALA est activable par son parent il faut se référer à la documentation du langage. Les concepteurs du langage ont délibérément choisi de ne pas laisser l'utilisateur avoir contrôle sur ce paramètre pour que SMALA soit moins complexe à comprendre et ainsi faciliter son apprentissage. Dans section 4.2.2, nous avons nommé, dans *Biguil*, l'action d'activation d'un PROCESSUS par son parent *l'activation initiale*. Dans *Biguil*, il est possible de choisir si un PROCESSUS doit être activé par son parent ou non. Ceci permet d'être plus flexible sur l'expressivité des comportements d'un PROCESSUS. Il est alors possible de différencier deux instances d'un même PROCESSUS en différenciant leur activation initiale. Il pourrait y avoir une instance qui est activable par son parent et la deuxième qui est activable uniquement par BINDING. Grâce au contrôle de l'activation initiale dans *Biguil*, il est possible de compiler une description SMALA vers notre UIDL. Ceci est illustré dans la figure 5.3.

```
1 _main_
2 Component root {
3   Int a (0)
4   Int b (42)
5   Spike s
6   a =: b
7 }
```

(a) Programme SMALA.

```
1 _main_
2 root:Component <a> {
3   root_a:Int <a> (0)
4   root_b:Int <a> (42)
5   root_s:Spike <d>
6   ass_0:root_a  =:<a> root_b
7 }
```

(b) Représentation du programme après la troisième passe.

FIGURE 5.3 – Passe 3 : Explicitation de l'activation initiale des PROCESSUS dans SMALA. La notation `<d>` indique que le PROCESSUS ne doit pas être activé par son parent. La notation `<a>` indique que le PROCESSUS doit être activé par son parent.

5.4 Compilation des machines à états

Dans la section 2.2.3 nous avons vu que SMALA permettait de définir des machines à état. Rappelons-le, une machine à état est un PROCESSUS SMALA permettant de structurer différents aspects d'une interface. Cette structuration du code est faite à travers des états qui consistent en des PROCESSUS conteneurs spécifiques représentant chacun un aspect différent de l'interface modélisée. Pendant l'exécution d'une interface, seulement un de ces état est activé à la fois et les changements d'état sont définis par des règles de transition. Une transition, généralement enclenché par un évènement utilisateur, consiste en la désactivation de l'état activé pour activer un nouvel état. *Biguil* ne permet pas d'utiliser les machines à états mais le langage permet toutefois de les implémenter avec les PROCESSUS mis à disposition. La figure 5.4 présente la manière d'encoder les machines à états avec les BINDINGS et les Components. La machine à état ainsi que ses états sont encodés comme des Components, nommés avec leurs identifiants respectifs, qui ont été attribués après la première passe e.g. l'état `red` correspond à un Component nommé `root_fsm_red` et encapsule tous les PROCESSUS encapsulés par celui-ci.

Concernant les transitions, celles-ci sont encodées avec les BINDINGS. Leur implémentation illustre bien l'importance des différents BINDING. En effet, une transition se doit de désactiver un état et d'en activer un autre et leur implémentation est possible uniquement si on a disposition des opérateurs permettant de gérer l'activation et la désactivation de PROCESSUS tels que les opérateurs $\rightarrow$, ! $\rightarrow$, $\rightarrow$! et ! $\rightarrow$!. Une transition de type `etatSource -> etatCible (ev)` passe de l'état source `etatSource` à l'état cible `etatCible` lorsque le PROCESSUS `ev` est activé. La transition d'un état $e1$ vers un état $e2$ peut être interprétée comme la désactivation de $e1$ qui enclenche l'activation de $e2$. Ainsi nous remplaçons la transition par les deux BINDINGS suivants :

— Le premier `ev ->! etatSource` qui désactive l'état source lorsque l'évènement est activé.

— Le deuxième `etatSource !-> etatCible` qui active l'état cible lorsque l'état source se désactive.

Dans l'exemple de la figure 5.4 la transition `red -> green (f.press)` est encodée par les BINDINGS : `root_fsm_bind1_0:root_f_press ->!<1> root_fsm_red` qui désactive le Component `root_fsm_red` correspondant à l'état `red` lorsque le PROCESSUS `f.press` est activé et `root_fsm_bind2_1:root_fsm_red !-><1> root_fsm_green` qui fait de même avec le Component `root_fsm_green` représentant l'état `green` lorsque le Component `root_fsm_red` se désactive.

Une transition de type `etatSource -> etatCible (ev,ev2)` qui passe de l'état source `etatSource` à l'état cible `etatCible` lorsque le PROCESSUS `ev` est activé tout en activant le PROCESSUS `ev2` au moment de la transition, est encodée presque de la même manière que le premier type de transition. La seule différence est qu'un BINDING en plus est défini, permettant d'activer `ev2` lorsque `etatSource`

se désactive.

5.5 Compilation des opérateurs arithmétiques représentés par du sucre syntaxique

SMALA dispose de PROCESSUS correspondants aux opérateurs arithmétiques d'addition, de soustraction, de multiplication, de division et de modulo. Dans la figure 5.5b nous pouvons voir que l'opérateur d'addition correspond en fait à un PROCESSUS nommé `Adder`. Ce PROCESSUS a trois PROCESSUS enfants implicites qui correspondent à des DOUBLE PROPERTIES. Ces PROPERTIES correspondent aux valeurs des opérandes gauche et droite de l'opérateur ainsi qu'au résultat de l'opération. Les paramètres du PROCESSUS `Adder` permettent d'initialiser les valeurs des PROPERTIES correspondant aux opérandes gauche et droite de l'addition. Les autres opérateurs de SMALA sont définis de la même manière que l'addition. Afin de faciliter leur utilisation, SMALA offre la possibilité d'utiliser des opérateurs infixes en sucre syntaxique illustrée dans la figure 5.5a.

Cette passe de compilation a pour but de remplacer le sucre syntaxique correspondant aux opérateurs arithmétiques de SMALA par du code équivalent supporté par le langage *Biguil*. Ici, seule la transformation concernant l'addition est présentée, les transformations pour les autres opérateurs sont similaires.

Rappelons que *Biguil* est un langage permettant d'étudier les aspects interactifs et spatiaux d'une interface. Ainsi, les opérations arithmétiques sont abstraites dans le langage et seul les comportements spatiaux et interactifs des opérateurs sont représentés. Pour implémenter les opérateurs arithmétiques de SMALA, il est important de comprendre leur comportement interactif et leurs relations spatiales afin d'avoir une idée des PROCESSUS de *Biguil* à utiliser. Concernant leurs relations spatiales, nous l'avons dit précédemment, les opérateurs arithmétiques SMALA possèdent trois PROPERTIES enfants implicites. Au sujet de leurs comportements interactifs, les opérateurs arithmétiques mettent à jour leur résultat à chaque fois qu'une de leur deux composantes est modifiée. À cela s'ajoute une subtilité : si jamais les deux composantes sont modifiées alors une synchronisation est effectuée pour que le résultat soit mis à jour une seule fois. La figure 5.6 présente cette transformation. La transformation remplace le PROCESSUS `Adder` par un Component du même nom qui encapsule trois DOUBLE PROPERTIES, correspondant aux enfants implicites de l'`Adder`, et deux BINDINGS. Ces BINDINGS permettent d'activer la PROPERTY `result` lorsque les PROPERTIES `right` ou `left` ont été modifiées. Ces deux BINDINGS sont définis dans le but de simuler l'activation de la PROPERTY `result` lorsque les composantes `left` ou `right` ont été modifiées. Pour terminer, la transformation modifie la définition de l'ASSIGNMENT en prenant compte des précédentes modifications et définit deux CONNECTORS. Les paramètres des opérateurs arithmétiques SMALA impliquent des interactions que nous n'avons pas abordé auparavant et qui sont simulées par les deux CONNECTORS définis. Ces interactions mettent en relation les paramètres de l'opérateur avec ses composantes gauche et droite de manière à modifier ces dernières lorsque les paramètres sont modifiés. Ces CONNECTORS mettent donc en lien respectivement les PROPERTIES `a` et `b` avec les PROPERTIES `left` et `right` permettant ainsi de mettre à jour les composantes gauche et droite de l'addition lorsque les PROPERTIES `a` ou `b` ont été modifiées.

5.6 Abstraction des processus

La sixième passe de compilation a pour but d'implémenter la déclaration des PROCESSUS prédéfinis de SMALA, e.g. les PROCESSUS graphiques et les fonctions mathématiques, dans *Biguil*. Pour ce faire, nous prenons une démarche similaire à celle de la passe précédente, i.e. nous analysons les

```
1  _main_
2  Component root {
3     Frame f ("frame",0,0,400,400)
4     FSM fsm {
5       State red {
6         FillColor r (255,0,0)
7         Circle c (20,20,20)
8       }
9       State green {
10        FillColor g (0,255,0)
11        Circle c (20,20,20)
12      }
13      State blue {
14        FillColor b (0,0,255)
15        Circle c (20,20,20)
16      }
17    red -> green (f.press)
18    green -> blue (f.press)
19    blue -> red (f.press)
20    }
21 }
```

(a) Programme SMALA.

```
1  _main_
2  root:Component <a> {
3    root_f:
4      Frame <a> ("frame"
           ,0,0,400,400)
5    root_fsm:Component <a> {
6      root_fsm_red:Component <a> {
7        root_fsm_red_r:
8          FillColor <a> (255,0,0)
9        root_fsm_red_c:
10         Circle <a> (20,20,20)
11     }
12     root_fsm_green:Component <d> {
13       root_fsm_green_g:
14         FillColor <a> (0,255,0)
15       root_fsm_green_c:
16         Circle <a> (20,20,20)
17     }
18     root_fsm_blue:Component <d> {
19       root_fsm_blue_b:
20         FillColor <a> (0,0,255)
21       root_fsm_blue_c:
22         Circle <a> (20,20,20)
23     }
24     root_fsm_bind1_0:
25       root_f_press ->!<a>
             root_fsm_red
26     root_fsm_bind2_1:
27       root_fsm_red !-><a>
             root_fsm_green
28     root_fsm_bind1_2:
29       root_f_press ->!<a>
             root_fsm_green
30     root_fsm_bind2_3:
31       root_fsm_green !-><a>
             root_fsm_blue
32     root_fsm_bind1_4:
33       root_f_press ->!<a>
             root_fsm_blue
34     root_fsm_bind2_5:
35       root_fsm_blue !-><a>
             root_fsm_red
36   }
37 }
```

(b) Représentation du programme après la quatrième passe.

FIGURE 5.4 – Passe 4 : Encodage des machines à état avec les BINDINGS et Components.

```
_main_
Component root {
  Int a (42)
  Int b (1)
  Int c (0)
  a + b =: c
}
```

(a) Programme SMALA utilisant la syntaxe simplifiée de l'addition.

```
_main_
Component root {
  Int a (42)
  Int b (1)
  Int c (0)
  Adder add (a,b)
  add.result =: c

}
```

(b) Le même programme SMALA utilisant la syntaxe complète de l'addition.

FIGURE 5.5 – Opérateur arithmétique dans SMALA.

```
_main_
Component root {
  Int a (42)
  Int b (1)
  Int c (0)
  a + b =: c

}
```

(a) Programme SMALA.

```
_main_
root:Component <a> {
  root_a:Int <a> (42)
  root_b:Int <a> (1)
  root_c:Int <a> (0)
  bin_op_1:Component Adder<a> {
    bin_op_1_left:
      Double <a> (0.)
    bin_op_1_right:
      Double <a> (0.)
    bin_op_1_result:
      Double <a> (0.)
    bin_op_1_bind1:
      bin_op_1_left  -><a>
         bin_op_1_result
    bin_op_1_bind2:
      bin_op_1_right  -><a>
         bin_op_1_result
  }
  connector_2:
    root_a  =><a> bin_op_1_left
  connector_3:
    root_b  =><a> bin_op_1_right
  ass_0:
    bin_op_1_result  =:<a> root_c
}
```

(b) Représentation du programme après la cinquième passe.

FIGURE 5.6 – Passe 5 : Transformation des opérateurs arithmétiques en PROCESSUS.

```
1 _main_
2 Component root {
3   Int a (42)
4 }
```

(a) Programme SMALA, représentation d'une INT PROPERTY.

```
1 _main_
2 root:Component <a> {
3   root_a:Component <a> Int(42) {
4     root_a_a:Spike <d>
5  }
6 }
```

(b) Représentation d'une INT PROPERTY après la sixième passe.

FIGURE 5.7 – Passe 6 : Transformation d'une PROPERTY.

comportements interactifs et les caractéristiques spatiales du PROCESSUS et nous les implémentons avec les instructions adéquates dans *Biguil*. Généralement, chaque PROCESSUS de SMALA est abstrait par un PROCESSUS Component de *Biguil* qui aura pour identifiant et fonction ceux du PROCESSUS SMALA. De plus, les PROCESSUS enfants des PROCESSUS SMALA seront rendus explicites, définis et seront encapsulés par le PROCESSUS Component de *Biguil*. Chaque PROCESSUS SMALA est différent mais il est toutefois possible de les classifier selon leurs caractéristiques interactives et topologiques afin d'éviter une transformation au cas par cas. Les sections suivantes décrivent chacune de ces classes de PROCESSUS ainsi que la transformation appliquée pour avoir du code *Biguil* les représentant.

5.6.1 Properties

Les PROPERTIES forment une classe de PROCESSUS. Dans SMALA, une PROPERTY est un PROCESSUS qui encapsule une valeur et qui émet une activation transitoire à chaque fois que sa valeur est modifiée. Dans la figure 5.7, nous implémentons une PROPERTY, dans *Biguil*, comme un Component ayant l'identifiant qui lui a été associé après la première passe et un type paramétrique correspondant au type de la PROPERTY et permettant d'encapsuler sa valeur. De plus ce Component encapsule un Spike, nommé `a`, qui simule l'activation de la PROPERTY à chaque fois que celle-ci est modifiée.

5.6.2 Bool property

Les BOOL PROPERTIES, ayant des caractéristiques uniques par rapport aux autres PROPERTIES et plus généralement aux autres PROCESSUS, forment à elles seules une classe de PROCESSUS dans cette sixième passe de compilation. Dans SMALA, une BOOL PROPERTY peut encapsuler deux valeurs, soit 0 (false) soit 1 (true), et possède deux Spikes enfants associés à celles-ci. Lorsque la valeur de la PROPERTY change à 0 alors le Spike qui lui est associé s'active et un comportement symétrique se produit lorsque sa valeur est changé à 1. La figure 5.8 montre qu'une BOOL PROPERTY est compilée en un PROCESSUS Component de type `Bool` encapsulant une INT PROPERTY représentant sa valeur et deux Spikes correspondants au Spike **true** qui est activé lorsque sa valeur est égale à 1 et au Spike **false** qui est activé lorsque sa valeur est égale à 0.

5.6.3 Processus graphique

Dans SMALA, les PROCESSUS graphiques diffèrent pour la plupart par leurs PROCESSUS enfants et pour chacune de ces différences une classe leur est associée. Toutefois, ils partagent tous les mêmes PROCESSUS d'interaction qui sont les PROCESSUS `press`, `release` et `move`, ce qui fait que leurs transformations ont de nombreuses similarités. Ces interactions communes viennent du postulat qu'il est

```
_main_
Component root {
  Bool b (0)
}
```

(a) Programme SMALA, représentation d'une BOOL PROPERTY.

```
_main_
root:Component <a> {
  root_b:Component Bool<a> {
    root_b_value :
     Component <a> Int(0) {
       root_b_value_a :Spike <d>
     }
    root_b_true:Spike <d>
    root_b_false:Spike <d>
  }
}
```

(b) Représentation d'une BOOL PROPERTY après la sixième passe.

FIGURE 5.8 – Passe 6 : Transformation d'une bool PROPERTY.

possible d'interagir, e.g. par des clics, des actions d'appui ou des mouvements, avec tout PROCESSUS graphique dans SMALA. La transformation de ces PROCESSUS SMALA consiste à les abstraire par un GComponent *Biguil* et à expliciter chacun de leur PROCESSUS enfant et d'interaction. La figure 5.9 illustre la transformation d'un PROCESSUS graphique en utilisant le PROCESSUS fenêtre. Celui-ci est représenté par un GComponent de type `Frame` encapsulant ses PROCESSUS enfants i.e. les DOUBLE PROPERTIES `root_f_title`, `root_f_x`, `root_f_y`, `root_f_width` et `root_f_height` ainsi que le Spike `root_f_close`. Ensuite vient la définition des PROCESSUS d'interaction communs aux PROCESSUS graphiques de SMALA. Le PROCESSUS **press** (respectivement **move**) est représenté par un Component de type `Press` (respectivement `Move`) encapsulant deux DOUBLE PROPERTIES correspondant aux coordonnées **x** et **y** de la souris. Le PROCESSUS **release** est lui abstrait via un Spike de type `Release`. L'action d'appui sur un bouton de souris ou bien son déplacement sont des actions persistantes, c'est donc pour cela que ces interactions sont abstraites par des Component. Lorsque le bouton de la souris est appuyé, deux actions sont alors possibles pour l'utilisateur. La première est de relâcher le bouton afin d'effectuer un clic, ceci est représenté par le Spike de type `Release` qui désactive le Component de type `Press`. La deuxième est de déplacer la souris en maintenant le bouton appuyé, ce qui est représenté par un premier BINDING qui active le Component de type `Move` et un deuxième qui le désactive lorsque le Spike de type `Release` est activé.

5.6.4 Fonction mathématique (input, output)

Les PROCESSUS définissant une fonction mathématique dans SMALA ont tous le même comportement interactif et forment une classe dans cette passe. Ces PROCESSUS ont une PROPERTY enfant nommé `input` qui est le paramètre de la fonction mathématique et une PROPERTY enfant nommé `output` qui correspond au résultat de la fonction. Le comportement interactif de ces PROCESSUS consiste à mettre à jour la PROPERTY `output` à chaque fois que la PROPERTY `input` est mise à jour. La transformation de ces PROCESSUS, illustrée par la figure 5.10 via la fonction cosinus, consiste à les transformer en un `Component` avec pour type la nature de la fonction (ici `Cosinus`). De plus ce `Component` encapsule les PROPERTIES `root_c_input` et `root_c_output` qui correspondent respectivement aux PROPERTIES `input` et `output` des fonctions mathématiques SMALA ainsi qu'un BINDING. Le BINDING, qui définit cette relation entre le paramètre de la fonction et son résultat, permet de mettre à jour le résultat lorsque le paramètre est mis à jour.

```
_main_
Component root {
 Frame f ("Frame",0,0,400,600)
}
```

(a) Programme SMALA, représentation d'une fenêtre.

```
_main_
root:Component <a> {
  root_f:GComponent Frame<a> {
    root_f_title :
     Component <a> String("Frame")
         {
      root_f_title_a :Spike <d>
     }
    root_f_x :
     Component <a> Double(0.) {
      root_f_x_a :Spike <d>
     }
    root_f_y :
     Component <a> Double(0.) {
      root_f_y_a :Spike <d>
     }
    root_f_width :
     Component <a> Double(0.) {
      root_f_width_a :Spike <d>
     }
    root_f_height :
     Component <a> Double(0.) {
      root_f_height_a :Spike <d>
     }
    root_f_close:Spike Close<d>
```

(b) Passe 6 : Représentation d'une fenêtre (1).

```
    root_f_press:
     Component Press<a> {
      root_f_press_x:
       Component <a> Double(0.) {
        root_f_press_x_a:Spike <d>
       }
      root_f_press_y :
       Component <a> Double(0.) {
        root_f_press_y_a:Spike <d>
       }
     }
    root_f_release: Spike Release<
       d>
    root_f_move:
     Component Move<a> {
      root_f_move_x:
       Component <a> Double(0.) {
        root_f_move_x_a:Spike <d>
       }
      root_f_move_y:
       Component <a> Double(0.) {
        root_f_move_y_a:Spike <d>
       }
     }
    root_f_bind1:
     root_f_press->&lt;a&gt;root_f_move
    root_f_bind2:
     root_f_release->!<a>
        root_f_move
    root_f_bind3:
     root_f_release->!<a>
        root_f_press
  }
}
```

(c) Passe 6 : Représentation d'une fenêtre (2).

FIGURE 5.9 – Passe 6 : Transformation d'un PROCESSUS graphique.

```
_main_
Component root {
  Cosine c (180)
}
```

(a) Programme SMALA, représentation de la fonction Cosinus.

```
_main_
root:Component <a> {
  root_c:Component Cosine<1> {
    root_c_input :
     Component <a> Double(180.) {
       root_c_input_a :Spike <d>
     }
    root_c_output :
     Component <a> Double(0.) {
       root_c_output_a :Spike <d>
     }
    root_c_bind1:
     root_c_input.a -><1>
        root_c_output.a
  }
}
```

(b) Représentation de la fonction Cosinus après la sixième passe.

FIGURE 5.10 – Passe 6 : Transformation d'une fonction mathématique.

5.6.5 Comparateurs, opérateurs booléens et arithmétiques

Les comparateurs, permettant de comparer deux PROPERTIES du même type, les opérateurs arithmétiques et les opérateurs booléens forment une dernière classe. À propos des opérateurs arithmétiques, la passe décrite dans la section 5.5 concerne seulement les opérateurs arithmétiques abstrait par du sucre syntaxique. Cependant dans SMALA, il est possible d'utiliser ces opérateurs sans utiliser le sucre syntaxique. Le choix de distinguer le traitement des opérateurs arithmétiques en deux passes vient du fait que l'encodage sous-jacent au sucre syntaxique est assez ardu à traiter et cela a permis de simplifier la compilation. Les transformations effectuées pour cette classe de PROCESSUS SMALA est similaire aux opérateurs arithmétiques, i.e. ils possèdent deux paramètres et si l'un d'eux est modifié alors le résultat est modifié, et par conséquence leur transformation, qui est décrite dans figure 5.11, est similaire à celle des opérateurs arithmétique décrite dans la section 5.5.

5.7 Encodage des connectors

La dernière passe permet de transformer les CONNECTORS définis dans le programme initial et durant les précédentes passes en la composition d'un ASSIGNMENT et d'un BINDING comme représenté dans figure 5.12. Le traitement des CONNECTORS est effectué en dernier car ils permettent de simplifier certaines transformations définie dans les passes précédentes e.g. section 5.5.

5.8 Conclusion

Dans ce chapitre nous avons pu montrer un aperçu de l'expressivité de *Biguil* en implémentant certains concepts du langage SMALA. Ces implémentations sont faites via sept passes de compilation qui traitent chacune un concept différent et permettent de compiler une partie du langage SMALA vers *Biguil*. En compilant le langage SMALA vers *Biguil*, nous permettons aux interfaces décrites via SMALA de profiter de l'aspect formel de notre UIDL. Cependant nous nous questionnons à propos

```
_main_
Component root {
  TextComparator tc ("k", "")
}
```

(a) Programme SMALA, représentation d'un comparateur de texte.

```
_main_
root:Component <a> {
  root_tc:
    Component TextComparator<a> {
      root_tc_left:
       Component <a> String("k") {
        root_tc_left_a: Spike <d>
       }
      root_tc_right:
       Component <a> String("") {
        root_tc_right_a: Spike <d>
       }
      root_tc_output:
       Component <a> Bool {
        root_tc_output_value:
         Int <a> (0)
        root_tc_output_true:
         Spike <d>
        root_tc_output_false:
         Spike <d>
       }
     root_tc_bind1:
       root_tc_left -><a>
          root_tc_output
     root_tc_bind2:
       root_tc_right -><a>
          root_tc_output
  }
}
```

(b) Représentation d'un PROCESSUS de comparaison de texte après la sixième passe.

FIGURE 5.11 – Passe 6 : Transformation des PROCESSUS de comparaison et des opérateurs booléens.

```
_main_
Component root {
  Int a (2)
  Int b (42)
  a => b
}
```

(a) Programme SMALA, représentation d'un CONNECTOR.

```
_main_
root: Component root<a> {
  root_a: Component <a> Int(2) {
       root_a_a:Spike <d>
  }
  root_b: Component<a> Int(42) {
       root_b_a:Spike <d>
  }
  assign_1: root_a =:<d> root_b
  con_0: root_a_a -><a> assign_1
}
```

(b) Représentation d'un CONNECTOR après la septième passe.

FIGURE 5.12 – Passe 7 : Transformation d'un CONNECTOR.

de la correction de la sémantique de *Biguil* et le respect des propriétés qu'elle couvre. Le prochain chapitre aborde ce sujet et présente aussi une technique de vérification automatique qui vérifie les propriétés présentées dans le chapitre précédent sur des interfaces décrites avec Smala.

Chapitre 6

Correction

Nous avons défini *Biguil* de sorte que sa sémantique, définie au travers de règles de réaction, couvre les propriétés essentielles qu'un UIDL doit respecter (section 4.3.2) mais cela reste à vérifier. Dans ce chapitre, nous allons montrer que ces règles de réaction sont correctes tout en explorant les différents moyens que les bigraphes nous donnent pour raisonner.

Nous commençons par présenter, dans la section 6.1, comment il est possible de vérifier automatiquement des propriétés sur une interface graphique. Cela est réalisable grâce à l'outil BigraphER qui nous a permis de coder notre formalisation (code disponible en annexe A.1) et ainsi vérifier automatiquement les propriétés de la section 4.3.2 sur des modèles *Biguil*. Par la suite, nous présentons dans la section 6.2, un pipeline de vérification pour les interfaces Smala s'appuyant sur ce procédé de vérification automatique.

La méthode introduit précédemment, permet de vérifier que les propriétés, de la section 4.3.2, sont couvertes par des descriptions *Biguil*. Cependant, cela ne prouve pas que ces propriétés sont vérifiées par l'ensemble des interfaces descriptibles par *Biguil*. Pour cela, il faudrait prouver que notre formalisation vérifie chacune de ces propriété. Dans la section 6.3, nous montrons que notre formalisme respecte bien la propriété 4.3.2 (Cohérence d'activation parent-enfant). Cette preuve que nous présentons, donne une idée des raisonnements qu'il faut effectuer sur un BRS pour prouver qu'une propriété est respectée.

6.1 Vérification des propriétés

L'une des approches de raisonnement envisageable sur les bigraphes, est de raisonner sur les systèmes de transitions générés par un BRS. Les outils BigraphER et PRISM nous aident à mécaniser ce raisonnement en permettant de vérifier des propriétés CTL (Computation tree logic) sur les systèmes de transition générés par les BRS (sections 2.3.3 et 2.3.4). Dans notre cas, cela est essentiel pour vérifier que les interfaces graphiques vérifient les propriétés définissant *Biguil*. Pour faire ces vérifications, nous définissons des prédicats BigraphER (section 2.3.3) qui permettent de générer des systèmes de transition étiquetées. Nous pouvons alors définir des formules CTL, représentant les propriétés de la sémantique de notre UIDL, que nous vérifions sur ces systèmes à transition à l'aide de PRISM. La section 6.1.1 présente ces prédicats et les sections qui suivent présentent les formules CTL utilisées pour vérifier les propriétés de *Biguil* sur une interface graphique.

6.1.1 Prédicats

Afin de vérifier les propriétés du langage sur un système de transition généré depuis un BRS et abstrayant l'exécution d'une interface graphique, nous avons besoin d'étiqueter les états qui nous intéressent. BigraphER permet de définir des prédicats p_i sous forme de bigraphe $p_i = b_i$ (b_i un bigraphe) permettant d'étiqueter, avec p_i, chaque état du système contenant une occurrence de b_i. Dans ce but nous spécifions les prédicats suivants :

Le prédicat `pAct(n)` permet d'étiqueter un état contenant le PROCESSUS `n` activé.

```
fun big pAct(n) = Process{a}.(State.On | Ident.Id(n) | id);
```

Le prédicat `pDeact(n)` permet d'étiqueter un état contenant le PROCESSUS `n` désactivé.

```
fun big pDeact(n) = Process{a}.(State.Off | Ident.Id(n) | id);
```

Le prédicat `transitional(n)` permet d'étiqueter un état contenant le PROCESSUS Transitoire `n`.

```
fun big transitional(n) = Process{a}.(Type.Transient | Ident.Id(n) | id);
```

Le prédicat `multisync(n)` aide à vérifier si un PROCESSUS a été activé plusieurs fois à cause de BINDINGS concurrents. En effet, la règle décrite par la figure 4.27 introduit une nouvelle entité de contrôle Labels dans laquelle l'identifiant d'un PROCESSUS ayant anciennement eu un état d'exécution Synchronising est rajouté. Si jamais cette règle est appliquée plusieurs fois sur un PROCESSUS ayant un état d'exécution Synchronising alors l'entité Labels va contenir de multiples occurrences de son identifiant. Le prédicat `multisync(n)` permet d'étiqueter les états dans ce cas.

```
fun big multisync(n) = Labels.(SyncAct(n) | SyncAct(n) | id);
```

Le prédicat `parentChildrenConsistency` permet d'étiqueter un état ne respectant pas la propriété 4.3.2 de cohérence d'activation parent enfant.

```
big parentChildrenConsistency =
  Process{a}.(State.Off | Semantic.Component.(Processb.(State.On | id) | id) | id);
```

Les prédicats `logicalGraphicalConsistency1` et `logicalGraphicalConsistency2` permettent d'étiqueter les états contenant des incohérences d'état d'activation chez un PROCESSUS entre les perspectives logique et graphique. Ce prédicat va nous aider à vérifier que la propriété 4.3.5 de cohérence d'activation entre perspectives logique et graphique est bien respectée.

```
big logicalGraphicalConsistency1 =
    Process{a}.(State.Off | id) || GProcess{a}.(State.On | id);
big logicalGraphicalConsistency2 =
    Process{a}.(State.On | id) || GProcess{a}.(State.Off | id);
```

Les prédicats suivants vont permettre de vérifier qu'une réaction n'a pas été stoppée prématurément. En effet, toutes les réactions terminent en réinitialisant les états d'exécution de chaque PROCESSUS. Si une réaction termine avec un PROCESSUS ayant un état d'exécution non vide alors la réaction s'est terminée prématurément.

```
big syncp = Synchronising;
big syncDp = SynchronisingD;
big actp = Activate;
big deactp = Deactivate;
big processingp = Processing;
big processingDp = ProcessingD;
big actAp = TempA;
big propagatingp = Propagating;
big idleP = Idle;
big calcPropp = CalcProp.id;
big calcSyncp = CalcSync.id;
```

6.1.2 Définitions des propriétés

Dans cette section, nous donnons la définitions des propriétés CTL que nous vérifions avec PRISM.

Définition propriété 4.3.2 (Cohérence d'activation parent-enfant)

Il est possible de vérifier qu'un PROCESSUS enfant n'est jamais activé lorsque son parent est désactivé avec la propriété suivante :

$$\neg\mathbf{E}[\,\texttt{parentChildrenConsistency}\,]$$

Cette propriété indique *qu'il n'existe aucun* état satisfaisant le prédicat `parentChildrenConsistency`.

Définition propriété 4.3.3 (Activation persistante)

Les PROCESSUS persistants ne devraient pas changer d'état plus d'une fois par réaction, e.g. `Off`→`On` ou `On`→`Off`. Afin de détecter ces séquences d'états invalides, nous définissons une famille de propriétés les capturant comme suit :

$$\neg\mathbf{E}\,[\ \texttt{pDeact(n)}\ \wedge(\mathbf{F}\ (\ \texttt{pAct(n)}\ \wedge(\mathbf{F}\ \ \texttt{pDeact(n)})))\,]$$

et

$$\neg\mathbf{E}\,[\ \texttt{pAct(n)}\ \wedge(\mathbf{F}\ (\ \texttt{pDeact(n)}\ \wedge(\mathbf{F}\ \ \texttt{pAct(n)})))\,]$$

$\forall n, \neg$ `transitional(n)`.

Dans ce cas nous avons utilisé l'opérateur $\neg\mathbf{E}$ (il n'existe pas) pour vérifier qu'à chaque fois qu'un PROCESSUS n est dans un état, e.g. `pAct(n)`, alors si il change *finalement* ($\mathbf{F}$) d'état et que le prédicat `pDeact(n)` est vérifié, alors il ne devrait jamais avoir un état par la suite où le prédicat `pAct(n)` est vérifié.

Définition propriété 4.3.4 (Activation transitoire)

À l'inverse, les PROCESSUS Transitoire, s'ils sont activés devraient propager leur changement d'état et se désactiver de nouveau. Il est possible de formuler une famille de propriétés capturant cette propriété de la manière suivante :

$$\mathbf{A}[\ \texttt{pAct(n)} \implies \mathbf{F}\ \texttt{pDeact(n)}\ ]$$

$\forall n$, `transitional(n)`.

C'est-à-dire, pout tout état (**A**) où un PROCESSUS est activé (`pAct(n)`) alors il existe un chemin où finalement (**F**) il est de nouveau désactivé.

Définition propriété 4.3.5 (Cohérence d'activation entre perspectives logique et graphique)

La perspective graphique se doit de refléter la perspective logique et doit donc suivre tout changement ayant lieu dans celle-ci. Il est possible qu'il y ait des états du système de transition contenant des incohérences entre les deux perspectives le temps que celles-ci soient mises à jour. Mais lorsque la réaction est terminée il faut que les deux perspectives soient cohérentes en terme d'activation de PROCESSUS. La propriété suivante permet de vérifier cela :

$$\mathbf{A}[\,\mathbf{F}\ \texttt{deadlock} \wedge \neg\texttt{logicalGraphicalConsistency1} \wedge \neg\texttt{logicalGraphicalConsistency2}\,]$$

Cette propriété indique que *Pour tout* (**A**) chemins, *finalement* (**F**) un état bloquant va être atteint et les prédicats `logicalGraphicalConsistency1` et `logicalGraphicalConsistency2` ne seront pas satisfaits. Un état bloquant correspond à un état finale du système à transition. Le prédicat `deadlock` de PRISM fait référence à ce type d'état.

Définition propriété 4.3.8 (Unique activation d'un processus à multiples prédécesseurs (bindings))

Nous avons vu qu'il était possible qu'un PROCESSUS ait plusieurs prédécesseurs pouvant l'activer en même temps. Dans une réaction, un PROCESSUS peut être activé ou désactivé seulement une fois et cela est expliqué au travers des règles de réécriture définies dans la section 4.3.4. La propriété suivante nous permet de vérifier cette caractéristique de la sémantique en utilisant la propriété `multisync(n)` définie dans la section 6.1.1 :

$$\neg\mathbf{E}[\,\texttt{multisync(n)}\,]$$

Pour tout n, identifiant un PROCESSUS.

Cette propriété indique *qu'il n'existe aucun* état satisfaisant le prédicat `multisync(n)`.

Définitions propriétés 4.3.10 (Terminaison) et 4.3.11 (Confluence)

Bien que les programmes interactifs s'exécutent continuellement pour attendre les évènement venant de leur environnement, nous voulons que la réaction enclenchée par chaque interaction termine, pour éviter que le système ne soit plus réactif aux évènements. Il faut également que l'état résultant après la réaction, i.e. l'ensemble des états d'activation des PROCESSUS constituant le programme, soit confluent, i.e. pour un même état initial, nous devrions toujours arriver au même état final, quel que soit l'ordre d'exécution des PROCESSUS dans la réaction.

En général, pour la terminaison de la réaction, nous vérifions que *finalement* un état bloquant (deadlock) est atteint avec la propriété :

$$\mathbf{A}[\,\mathbf{F}\ \texttt{deadlock}\,]$$

Cela indique que *Pour tout* chemin (**A**), nous atteignons *finalement* (**F**) un état bloquant. Afin d'assurer que la réaction conflue, nous utilisons la propriété $filter$ de PRISM, i.e. `filter(count, deadlock)`, qui renvoie un compte du nombre d'états bloquants, pour garantir que ce nombre est égal à 1.

Toutefois, cette propriété de terminaison n'est pas assez forte : bien qu'elle détecte la fin de la réaction, elle ne vérifie pas que celle-ci s'est terminée correctement, e.g. une réaction peut être bloquée au lieu d'être finie. Afin d'améliorer cette propriété, nous lui ajoutons 11 prédicats (présentés en section 6.1.1) permettant de vérifier que l'état d'exécution de tous les PROCESSUS à la fin de la réaction est vide. Aves ces prédicats, la propriété de terminaison est définie comme suit :

$$\begin{aligned}\mathbf{A}[\,\mathbf{F}\ (&\texttt{deadlock} \wedge \neg\texttt{syncp} \wedge \neg\texttt{syncDp} \wedge \neg\texttt{actp}\\ \wedge&\neg\texttt{deactp} \wedge \neg\texttt{processingp} \wedge \neg\texttt{processingDp} \wedge \neg\texttt{actAp}\\ \wedge&\neg\texttt{idlep} \wedge \neg\texttt{calcPropp} \wedge \neg\texttt{calcSyncp})\,]\end{aligned}$$

Cette propriété vérifie que lorsqu'un état bloquant est atteint alors au même moment tout PROCESSUS constituant l'interface ont leur état d'exécution vide.

Dans la prochaine section, nous présentons un pipeline de vérification, qui repose sur cette méthode de vérification de prédicats, pour les interfaces graphique SMALA.

6.2 Pipeline de vérification

Les transformations présentées dans les sections précédentes permettent de transformer un programme SMALA en un programme *Biguil*. L'avantage en faisant cela, est la possibilité de vérifier que les propriétés présentées dans le chapitre précédent sont respectées par un programme SMALA. Dans cette section, nous présentons le pipeline de vérification permettant de vérifier ces propriétés sur un programme SMALA.

Le pipeline de vérification pour les programmes SMALA, illustré dans la figure 6.1, regroupe toutes les notions que nous avons vues jusqu'à présent. On y retrouve, la transformation d'un programme *Biguil* en bigraphe ainsi que les règles de réaction formalisant la sémantique de l'UIDL, présentées dans le chapitre 4. À cela s'ajoutent, les sept transformations définies dans le chapitre précédent permettant de transformer une description SMALA en une description *Biguil*. Enfin nous retrouvons aussi, les propriétés CTL sur les systèmes à transition générés par BigraphER présentées dans la section précédente. Ce pipeline de vérification, prend en entrée une description SMALA d'une interface graphique qui est ensuite transformée en une description *Biguil* par les sept transformations présentées dans le chapitre 5. La description *Biguil* obtenue modélise l'interface au travers d'un bigraphe qui, associé à des règles de réaction définissant la sémantique de *Biguil*, peut être simulé par l'outil BigraphER. BigraphER peut ensuite générer un système de transition ([**sect.etatsdeslieux.bigrapher**]) correspondant à une éventuelle exécution de l'interface graphique. Ce système à transition peut être converti en une matrice de transition qui peut être utilisée par l'outil de vérification PRISM. PRISM, présenté en section 2.3.4, pourra ensuite se charger de vérifier l'ensemble des propriétés CTL spécifiées.

De manière à vérifier les propriétés CTL du chapitre 6 sur un certain état de l'interface graphique, il est possible d'annoter une description SMALA afin de spécifier l'état d'exécution ou d'activation des

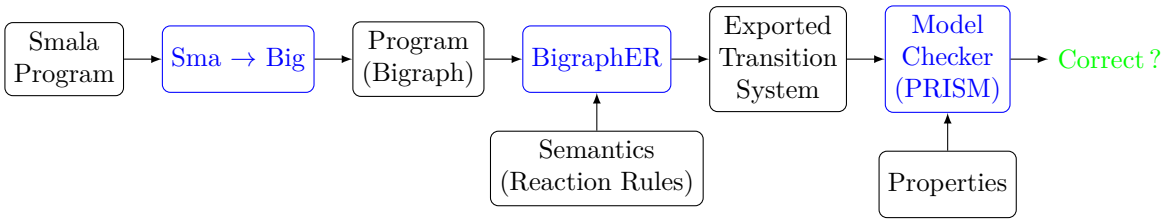


FIGURE 6.1 – Vue d'ensemble sur le pipeline de vérification.

```
1  _main_
2  Component root [@on] {
3   Spike a [@activate]
4   Spike b
5   a ->[@on] b
6  }
```

FIGURE 6.2 – Programme SMALA annoté.

PROCESSUS le définissant. En d'autres termes, ces annotations remplacent les interactions utilisateur en indiquant quel PROCESSUS activer pendant la simulation de l'interface graphique.

1. L'annotation `[@activate]` permet d'initialiser l'état d'exécution d'un PROCESSUS à Activate,
2. l'annotation `[@on]` permet d'initialiser l'état d'activation d'un PROCESSUS à On et
3. l'annotation `[@idle]` permet d'initialiser l'état d'exécution d'un PROCESSUS à Idle.

Par défaut, les état d'exécution des PROCESSUS non spécifiés sont initialisés avec le bigraphe vide et les état d'activation sont initialisés à Off.

Ces annotations se placent à droite des identifiants des PROCESSUS lors de leur définition ou bien de leur symbole lorsqu'il s'agit de PROCESSUS représentés par du sucre syntaxique e.g. BINDING, CONNECTOR ou ASSIGNMENT.

La figure 6.2 présente un programme SMALA annoté. Dans ce programme, on a choisi d'initialiser l'état d'activation du PROCESSUS `root` et du BINDING à On et d'initialiser l'état d'exécution du Spike `a` à Activate. Ainsi l'exécution de ce programme produit l'activation du Spike `a` qui va se propager via le BINDING qui est activé et activer le Spike `b`.

Les annotations permettant d'initialiser les états d'activation et d'exécution des PROCESSUS sont traitées par des règles de réaction préalablement définies et qui sont prioritaires par rapport à toutes les autres règles de réaction présentées jusqu'à présent. Une règle de réaction est associée à chaque annotation :

1. La règle de réaction traitant l'annotation `[@activate]` est définie dans la figure 6.3. Dans la redex de cette règle, le PROCESSUS contient une entité User qui a pour rôle d'encapsuler des entités correspondant aux annotations. Ici, l'entité encapsulée est l'entité Activate correspondant à

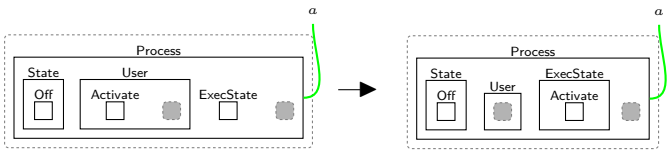


FIGURE 6.3 – Règle de réaction traitant l'annotation `[@activate]`.

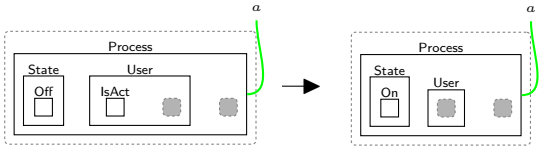


FIGURE 6.4 – Règle de réaction traitant l'annotation `[@On]`.

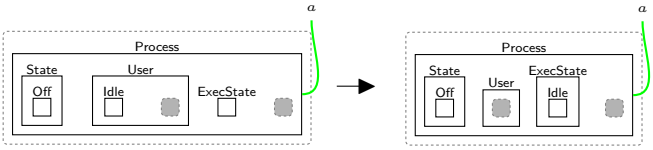


FIGURE 6.5 – Règle de réaction traitant l'annotation `[@idle]`.

l'annotation `[@activate]`. Dans le reactum de la règle nous pouvons voir que l'entité Activate a été retirée de l'entité User du PROCESSUS et que l'état d'exécution de celui-ci a été changé en conséquence.

2. La figure 6.4 décrit le traitement effectué à un PROCESSUS lorsque il est annoté par `[@On]`. Dans la partie gauche de la règle nous pouvons voir que l'entité User contient une entité On. Dans la partie droite de la règle, nous pouvons voir que cette entité a été retirée et que l'état d'activation du PROCESSUS est passé à On.
3. Enfin la figure 6.5 décrit le traitement effectué à un PROCESSUS lorsque celui-ci est annoté par `[@idle]`. Dans la partie gauche de la règle nous pouvons voir que l'entité User contient une entité Idle. Dans la partie droite de la règle, cette entité est retirée et l'état d'exécution du PROCESSUS est passé à Idle.

Nous venons de présenter la constitution du pipeline de vérification ainsi que certaines annotations permettant de modifier l'état d'une interface graphique pour pouvoir l'exécuter a notre convenance. Les exemples sur lesquels nous avons utilisé ce pipeline sont présentés dans l'annexe A.2. Une partie de ces exemples n'est pas graphique et a uniquement pour but de tester la robustesse de l'aspect interactif de la sémantique. Par exemple, il peut s'agir de descriptions focalisées sur les propagations concurrentes de BINDINGS et d'ASSIGNMENTS. Une autre partie de ces exemples est graphique et permet de se focaliser sur l'aspect structurel de l'interface tel que les cohérences entre les perspectives logique et graphique et l'activation parent-enfant. Tous les prédicats présentés en section ont été vérifiés sur tout ces exemples. Le tableau 6.1 énumère ces exemples ainsi que les aspects qu'ils couvrent.

Nous abordons dans la section suivante les preuves sur les BRS en présentant une preuve de la propriété 4.3.2 sur notre formalisation.

6.3 Preuve de propriété de cohérence d'activation parent-enfant

Dans cette section nous allons donner une preuve pour propriété 4.3.2 que nous rappelons ci-dessous.

Propriété (Cohérence d'activation parent-enfant) *Un PROCESSUS, différent du PROCESSUS `root`, est activé seulement si son PROCESSUS parent est activé.*

Afin de prouver cette propriété, nous allons prouver le lemme suivant dont elle découle.

	Propagating simple	Propagation concurrente	sémantique des ASSIGNMENTS	sémantique des CONNECTORS	sémantique des Components	Cohérence perspectives logique et graphique	Transitivités BINDINGS, CONNECTORS, ASSIGNMENTS	Sémantique FSM
BindingII	✓							
BindingIO	✓							
BindingOI	✓							
BindingOO	✓							
Deactivation diamond		✓						
Simple data flow							✓	
Diamond broken link		✓						
Adder small		✓	✓					
Diamond		✓						
Arith			✓					
Double activation assignment			✓					
Assignment 2			✓				✓	
Connector				✓				
Deactivation Component					✓			
Adder big		✓	✓			✓		
Double choice		✓				✓	✓	
Assignment			✓					
Assignment 3		✓	✓				✓	
SceneG						✓		
Elastic Window	✓			✓	✓	✓		
ATM	✓				✓	✓		✓
TCAS	✓				✓	✓		✓

TABLE 6.1 – Aspects testés par les exemples d'annexe A.

Lemme 6.3.1 *L'activation d'un* PROCESSUS *implique que soit le parent du* PROCESSUS *est activé, soit l'activation du* PROCESSUS *a pour origine l'activation de son parent.*

Formellement,

$\forall$ C_0, p_0, C_1,p_1.
($C_0 \circ$ ChildOff $\circ$ p_0 $\rightarrow^*$ $C_1 \circ$ ChildOn $\circ$ p_1
$\Rightarrow$
$\exists$ C_2,p_2.
$C_2 \circ$ ParentOn $\circ$ (childOn | ParentOtherChildren) $\circ$ p_2
avec
ParentOtherChildren correspondant aux autres enfants de ParentOn,
$C_1 \equiv C_2 \circ$ parentOn $\circ$ ((id_0|ParentOtherChildren) || ParentOtherEntities),
ParentOtherEntities le substitut du site id_1 de parentOn et
$p_2 \equiv$ ParentOtherEntities || p_1

$\vee$

$\exists$ C_2,p_2,C_3,p_3.
$C_2 \circ$ ParentOffActivate $\circ$ (ChildOff | ParentOtherChildren) $\circ$ p_2
$\rightarrow^*$
$C_3 \circ$ ParentOnIdle $\circ$ (ChildOn | ParentOtherChildren') $\circ$ p_3
avec
ParentOtherChildren correspondant aux autres enfants de ParentOffActivate,
ParentOtherChildren' correspondant aux autres enfants de ParentOnIdle,
$C_0 \equiv C_2 \circ$ ParentOffActivate $\circ$ ((id_0|ParentOtherChildren) || ParentOtherEntities),
ParentOtherEntities le substitut du site id_1 de parentOffActivate,
$p_2 \equiv$ ParentOtherEntities || p_0,
$C_1 \equiv C_3 \circ$ ParentOnIdle $\circ$ ((id_0|ParentOtherChildren') || ParentOtherEntities'),
ParentOtherEntities' le substitut du site id_1 de parentOnIdle et
$p_3 \equiv$ ParentOtherEntities' || p_1
)
avec
ChildOff $\equiv$ Process{x}.(State.Off | id_0),
ChildOn $\equiv$ Process{x}.(State.On | id_0),
ParentOn $\equiv$ Process{y}.(State.Off | Semantic.Component.id_0 | id_1),
ParentOff $\equiv$ Process{y}.(State.On | Semantic.Component.id_0| id_1),
ParentOffActivate $\equiv$
Process{y}.(State.On | Semantic.Component.id_0| StateExec.Activate | id_1),
ParentOnIdle $\equiv$ Process{y}.(State.On | Semantic.Component.id_0| StateExec.Idle | id_1),
l'opérateur $b_0 \rightarrow^* b_1$ représente les réactions appliquées au bigraphe b_0 pour
obtenir le bigraphe b_1, $\equiv$ l'opérateur d'équivalence et $\circ$ l'opérateur
de composition sur les bigraphes définis dans [31].

6.3.1 Preuve lemme 6.3.1

Pour prouver lemme 6.3.1 nous allons raisonner par contraposée et prouver l'assertion suivante.

$\forall$ C_0, p_0, C_1,p_1.
($\forall$ C_2,p_2.
¬(C_2 ∘ ParentOn ∘ (childOn | ParentOtherChildren) ∘ p_2)
$\wedge$

$\forall$ C_2,p_2,C_3,p_3.
C_2 ∘ ParentOffActivate ∘ (ChildOff | ParentOtherChildren) ∘ p_2
$\rightarrow\!\!\!\!\!\!\rightarrow^*$
C_3 ∘ ParentOnIdle ∘ (ChildOn | ParentOtherChildren') ∘ p_3
$\Rightarrow$
C_0 ∘ ChildOff ∘ p_0 $\rightarrow\!\!\!\!\!\!\rightarrow^*$ C_1 ∘ ChildOn ∘ p_1
)
avec
l'opérateur b_0 $\rightarrow\!\!\!\!\!\!\rightarrow^*$ b_1 la négation de l'opérateur b_0 $\rightarrow^*$ b_1.

Dans cette preuve, nous allons montrer que pour tout contexte C et paramètre p il n'existe aucune chaîne de réactions menant à l'activation du PROCESSUS ayant pour lien x i.e. le PROCESSUS enfant que nous nommons **Child** dans la suite. Pour cela, nous allons raisonner seulement sur tous les cas couverts par la loi de composition de notre formalisation (section 4.3.1).

Par hypothèse, p_0 est un bigraphe correspondant aux entités manquantes de **Child**. Dans la suite du raisonnement, nous nous préoccupons seulement des entités Semantic, Type et ExecState de **Child** constituant ce bigraphe p_0 ; les autres parties de ce bigraphe ne sont pas essentielles et peuvent être abstraites. La preuve se déroule donc de la manière suivante : nous fixons un bigraphe p_0, ce qui va permettre de raisonner sur les différents états d'exécution et sémantiques que **Child** peut avoir. Concernant les états d'exécution, nous traitons seulement ceux qui peuvent permettre à **Child** de s'activer, c'est-à-dire les états Activate et vide. Nous raisonnons ensuite sur les tous les contextes C_0 possible d'activer **Child**, tels que les contextes qui ont des BINDINGS ou des ASSIGNMENTS qui veulent activer **Child**, et nous montrons que cela ne se produit pas.

Pour cela, une fois les hypothèses émises sur **Child**, nous continuons le raisonnement en appliquant les règles de réaction selon leur priorité, présentées dans la figure 4.5, et montrons qu'aucune d'elles ne permet d'activer **Child**.

Cas 1 : État d'exécution de Child à Activate

Dans ce premier cas, comme résumé dans le tableau 6.2, nous supposons que **Child** est un PROCESSUS quelconque, de type quelconque et qui a pour état d'exécution Activate tel que,

C_0 ∘
Process{x}.(State.Off| ExecState.Activate | id_0)
∘ p_0

avec id_0 le site correspondant aux paramètres manquants de **Child** et p_0 le bigraphe correspondant à ces paramètres manquants.

D'après les règles de réaction définissant le BRS et leur priorité, les premières règles applicables sur notre cas sont les règles gProcess_act et gProcess_deact. En effet, il est possible, pour tout contexte C_0, que certains PROCESSUS aient subi des changements d'état lors d'une précédente réaction et qu'ils répercutent ces changements dans la perspective graphique du bigraphe. L'application de ces règles n'a cependant aucun impact sur **Child**.

ExecState	Semantics	Type	Contexte	Paramètre
Activate	Quelconque	Quelconque	Quelconque	Quelconque

TABLE 6.2 – Cas1 : Hypothèse sur **Child**, son contexte et ses paramètres.

ExecState	Semantics	Type	Contexte	Paramètre
vide	Quelconque	Quelconque	au moins un BINDINGS propageant un changement d'état à **Child**	Quelconque

TABLE 6.3 – Cas 2 : Hypothèse sur **Child**, son contexte et ses paramètres.

La prochaine règle de réaction applicable sur notre cas est la règle `parent_deact`. Cette règle va mettre fin à la tentative d'activation de **Child** car son parent est désactivé par hypothèse. Cette application de règle ainsi que la seconde hypothèse de la contraposée assurant que **Child** ne peut être activé via activation de son PROCESSUS parent, montrent que **Child** ne peut pas être activé dans ce cas.

Cas 2 : activation par bindings

Les hypothèses émises sur **Child** pour ce deuxième cas sont illustré dans le tableau 6.3. Nous supposons maintenant que **Child** est un PROCESSUS de sémantique et de type quelconque, que son état d'exécution est vide et que le contexte $\mathtt{C}_0$ contienne un ou plusieurs BINDINGS voulant propager, à **Child**, un changement d'état. Ce cas va être traité par les règles `multi_trg_propagating`, `trg_propagating`, `synch_to_activate` et `synch_to_deactivate`. Néanmoins, il est possible que le contexte $\mathtt{C}_0$ ou la partie abstraite du paramètre $\mathtt{p}_0$ soient constitués de manière que les règles de réaction précédant ces dernières puissent être applicable.

- Le premier ensemble de règles de réaction contient les règles `gProcess_Act` et `gProcess_deact` et, comme expliqué précédemment, l'application de ces règles n'a aucun impact sur l'activation de **Child** ou bien sur des BINDINGS susceptible de changer son état.
- Le second ensemble de règles contient la règle `parent_deact`, cette règle va empêcher l'activation des PROCESSUS qui ont un parent désactivé dans le contexte $\mathtt{C}_0$ ou bien le bigraphe paramétrique $\mathtt{p}_0$. **Child** ne sera pas impacté par cette règle car initialement son état d'exécution est vide. Concernant les BINDINGS acteurs de changements d'état sur **Child**, leur états d'exécution sont supposés contenir les informations de propagation de sa source et non une entité Activate. Il est donc impossible d'appliquer la règle `parent_deact` sur ces BINDINGS. L'application de la règle `parent_deact` ne va donc aucunement influer sur l'activation de **Child**.
- L'ensemble de règles suivant contient des règles gérant l'activation des Components. L'activation d'un Component pourrait avoir un impact sur **Child** si celui-ci le contient. Cependant, par hypothèse, le parent de **Child** n'est pas activé. Cela justifie que ces règles vont être appliquées seulement aux Components se trouvant en contexte ou en paramètre et ne contenant pas **Child**. L'activation ou la désactivation de ces Components peuvent néanmoins enclencher l'activation de **Child** à travers un BINDING, mais ce cas relève de l'utilisation des règles `src_propagating` et de calcul d'état d'exécution. Ce cas sera donc traité lorsque nous aborderons les impacts de ces règles de réaction sur **Child**.
- Le prochain ensemble de règles se charge de l'activation/désactivation des PROCESSUS et de la

temporisation des ASSIGNMENTS. Ces règles n'ont donc aucun impact sur **Child** car celui-ci est supposé avoir un état d'exécution vide. Celles-ci vont donc activer/désactiver d'autre PROCESSUS en contexte ou en paramètre. De la même manière que l'activation ou la désactivation des enfants d'un Component, ces PROCESSUS peuvent aussi enclencher l'activation de **Child** via l'intermédiaire d'un BINDING. Ce cas découle de l'utilisation des prochains ensembles de règle.

— Les deux ensembles de règles suivant contiennent les règles de calcul des états d'exécution. L'état d'exécution de **Child** étant vide, ces règles ne vont pas s'appliquer à celui-ci. Cependant, elles peuvent s'appliquer à des PROCESSUS en contexte ou en paramètre pouvant provoquer son activation au travers de BINDINGS. Si cela se produit, ce cas sera géré par les ensembles de règles contenant les règles `src_propagating`, `multi_trg_propagating` et `trg_propagating`.

— Le prochain ensemble contient la règle `src_propagating` qui se charge de propager les informations d'un PROCESSUS source au BINDING auquel il est connecté pour que celui-ci propage ces informations, à son tour, à son PROCESSUS cible. Cette règle s'applique sur un PROCESSUS source qui a un état d'exécution Propagating et un BINDING qui est activé et donc n'a aucun impact sur **Child**. En s'appliquant sur des BINDINGS et PROCESSUS en contexte ou en paramètre, cette règle peut toutefois augmenter le nombre de BINDINGS voulant activer **Child**.

On arrive maintenant à notre cas principal qui consiste en un ou plusieurs BINDINGS qui ont pour cible **Child** et voulant lui propager un changement d'état.

— Dans le cas où il y a plusieurs BINDINGS connectés à **Child**, l'application successive des règles `multi_trg_propagating` et de calcul d'état d'exécution vont permettre de traiter tous les BINDINGS et de se retrouver avec deux cas de figure selon les propagations qui ont été faites.

1. Il est possible que tout les BINDINGS connectés ainsi que **Child** ne valident aucune condition d'activation et que **Child** se retrouve avec un état d'exécution vide. Dans ce cas, la propriété est respectée car **Child** reste désactivé
2. Il est aussi possible qu'un BINDING et **Child** aient respecté toutes les conditions pour une activation et que **Child** se retrouve avec un état d'exécution Synchronise. Dans cette situation, la règle `synch_to_activate` est appliquée pour passer l'état d'exécution de **Child** à Activate. Le **cas 1** justifie que **Child** ne s'active pas du fait que son parent est désactivé.

— Dans le cas où un seul BINDING est connecté à **Child**, l'application successive de la règle `trg_propagating` ainsi que les règles de calcul d'état d'exécution vont permettre de se retrouver dans deux cas de figure selon le traitement des propagations par les règles de calcul.

1. Il est possible qu'aucune propagation n'ait été faite car aucune condition d'activation n'a été respectée. Dans ce cas **Child** se retrouve désactivé avec un état d'exécution vide et la contraposée est donc validée.
2. Dans le deuxième cas de figure, **Child** a un état d'exécution à Activate et le **cas 1** permet de terminer le raisonnement.

Cas 3 : activation par assignment

Ce cas traite de la tentative d'activation de **Child** par des ASSIGNMENT en temporisation se trouvant dans le contexte. Pour ce cas, un ASSIGNMENT activant seulement le Spike d'une PROPERTY, nous supposons que **Child** est le Spike en question et que son parent est le Component le contenant, comme illustre en tableau 6.4. De la même manière que les précédents cas, nous allons raisonner sur tous les contextes $\mathtt{C}_0$ et paramètres $\mathtt{p}_0$ possibles et montrer qu'il est impossible d'aboutir à un état où le Spike (**Child**) s'active.

ExecState	Semantics	Type	Contexte	Paramètre
vide	Spike	Transient	Inclus dans un Component afin de constituer une PROPERTY	Quelconque

TABLE 6.4 – Cas 3 : Hypothèse sur **Child**, son contexte et ses paramètres.

Les règles de réaction gérant la temporisation d'un ASSIGNMENT ont une priorité inférieure aux règles gérant l'activation des BINDINGS. De ce fait, le contexte et les paramètres du bigraphe cible peuvent contenir tous les cas de figures énumérés dans le **cas 2** avec en plus, la potentielle présence de BINDINGS pouvant activer **Child**. Le **cas 2** est suffisant pour justifier que ces différents cas ne vont pas activer **Child**.

Comme expliqué dans la section 4.3.3, l'activation d'un ASSIGNMENT en temporisation se fait après l'activation de tous ses prédécesseurs. L'activation des prédécesseurs de l'ASSIGNMENT temporisé n'a aucun impact sur **Child** qui est rattaché à celui-ci. En effet, les règles de réaction se chargeant d'activer les ASSIGNMENTS prédécesseurs, vont modifier seulement ces ASSIGNMENTS et rien d'autre. Il est toutefois possible que les ASSIGNMENTS prédécesseurs enclenchent des PROCESSUS pouvant activer **Child** via des BINDINGS mais le **cas 2** justifie que le PROCESSUS enfant ne va pas s'activer.

Une fois que tous les prédécesseurs des ASSIGNMENTS rattachés à **Child** ont été activés, la règle `check_pred_multi_targ_2` va se charger d'activer tout les ASSIGNMENTS en concurrences jusqu'à qu'il n'en reste qu'un à activer. L'activation des ASSIGNMENTS concurrents peut engendrer l'activation de PROCESSUS connectés à **Child** au travers de BINDINGS. Cependant, nous avons montré dans le **cas 2** qu'ils ne peuvent pas activer **Child**. Le dernier ASSIGNMENT temporisé restant est traité avec la règle `assign_act_targ` qui va changer l'état d'exécution de **Child** en Activate. Cependant, nous avons montré dans le **cas 1** que l'activation de **Child** ne se fera pas.

Nous avons ainsi prouvé, par contraposée, le lemme 6.3.1.

6.3.2 Application du lemme 6.3.1

Prouvons maintenant la propriété 4.3.2 avec le lemme que nous venons de prouver. Soit p un PROCESSUS, différent du PROCESSUS root, qui est activé. Avant d'être activé, l'activation de p a dû être enclenchée par un autre PROCESSUS. Or, d'après le lemme 6.3.1, l'activation de p implique que son parent soit activé ou bien que l'activation de son parent soit à l'origine de son activation. Ce qui justifie que lorsque p est activé alors son parent est aussi activé.

6.4 Conclusion

Dans ce chapitre, nous avons vu comment il était possible de vérifier automatiquement, via PRISM, des propriétés, en énonçant des prédicats via BigraphER et en formulant des propriétés en CTL reposant sur ces prédicats. De plus, nous avons vu comment il était possible de prouver des propriétés sur un BRS, à travers la preuve de propriété 4.3.2 donnée dans section 6.3.

Les formules CTL permettent de couvrir une partie des propriétés de *Biguil* mais ne garantissent pas à coup sûr leur respect. En effet, la vérification de ces règles s'effectue sur un BRS en particulier et ne garantit pas que tous les BRS pouvant être construits à partir de notre formalisation les respectent. Pour garantir de manière infaillible le respect de ces règles, il faudrait faire les preuves des

autres propriétés, mise à part propriété 4.3.1, comme nous l'avons fait dans section 6.3. Concernant propriété 4.3.1, elle requiert la définition d'un sorting sur le modèle donné ainsi que la preuve que les règles de réaction définies le préservent. Cela permettrait de justifier formellement qu'un bigraphe définit via un programme *Biguil* est bien construit et de plus que les règles de réaction agissant sur ce bigraphe vont préserver la propriété de bonne construction.

Enfin, la méthode de vérification présentée dans section 6.1 pourrait être utilisée pour analyser une interface graphique pendant son exécution, par exemple, pour vérifier qu'un évènement enclenche l'affichage d'un certain composant graphique.

Dans le prochain chapitre, nous synthétisons tous les résultats présentés dans cette thèse et discutons de leur limitations.

Chapitre 7

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à la vérification des UIDLs, langages de modélisation d'interface utilisateur, dans le but de produire des interfaces utilisateur fiables pour les systèmes critiques. À long terme, notre but est de concevoir un UIDL formel accompagné d'un compilateur générant du code vérifié pour les interfaces modélisées. Nous aimerions vérifier que le code généré conserve toutes les propriétés de l'interface utilisateur décrites via l'UIDL. Cette approche a deux avantages. Le premier est d'utiliser le langage de manière conventionnelle et de modéliser les interfaces désirées pour obtenir du code exécutable sémantiquement équivalent. Le deuxième est d'utiliser le langage défini comme langage intermédiaire, cela permettra de compiler d'autres UIDLs vers celui-ci pour qu'ils bénéficient de l'aspect formel recherché dans le développement de logiciels critiques. Nous avons commencé ce projet en définissant un UIDL formel que nous avons appelé *Biguil* (Bigraphical user interface language). Pour définir un tel UIDL, nous nous sommes confrontés aux trois questions suivantes :

Quelles sont les caractéristiques et propriétés communes des sémantiques des UIDLs spécialisés dans la description des interfaces graphiques ?

En réponse à cette première question, nous avons observé que la plupart des UIDLs utilisés dans la modélisation d'interfaces graphiques avaient deux caractéristiques communes qui sont la représentation de la topologie de l'interface et la représentation de ses interactions. Ces caractéristiques ainsi que les propriétés qui en découlent sont développées dans la section 7.1. Nous apporterons aussi dans cette section, quelques remarques au sujet de ces caractéristiques notamment sur leur suffisance à définir la sémantique d'un UIDL.

Comment pouvons-nous représenter les aspects spatiaux et interactifs de la sémantique des UIDLs, de façon à pouvoir raisonner efficacement sur celle-ci ?

En réponse à cette seconde question, nous nous sommes orientés vers la théorie des bigraphes que nous utilisons pour présenter les aspects spatiaux et interactifs de la sémantique des UIDLs. La section 7.2 développe nos résultats à ce sujet et présente les limitations de notre approche.

Est-il possible de définir un UIDL aux instructions atomiques qui permettent de former des constructions interactives et spatiales plus complexes ?

En réponse à cette dernière question nous avons présenté notre UIDL *Biguil* permettant de modéliser et vérifier des interfaces graphiques. *Biguil*, basé sur la théorie des bigraphes, possède un ensemble

restreint d'instructions permettant de définir des concepts interactifs et structurels complexes. Nous présentons ces instructions et abordons les limites de notre langage en section 7.3.

Nous terminerons cette thèse avec la section 7.4 qui abordera les perspectives envisagées pour continuer ce projet.

7.1 Caractéristiques et propriétés des UIDLs

D'après les comparaisons effectuées sur différents UIDLs, les documentations des UIDLs étudiés et l'état de l'art, nous concluons dans le chapitre 2 que deux caractéristiques permettent de définir la sémantique des UIDLs. Ces caractéristiques sont la capacité du langage à décrire la topologie des interfaces graphiques et à décrire les interactions qui les constituent. Ces deux caractéristiques sont nécessaires afin de décrire une interface graphique. En effet, la première permet de définir la disposition de chaque élément graphique composant l'interface. La deuxième permet de décrire les interactions entre les différentes entités graphiques. De ces caractéristiques découlent des propriétés et des traits que la sémantique des UIDLs respecte et possède. Parmi ces traits et propriétés nous trouvons :

- une loi de composition régissant l'assemblage des composants graphiques formant l'interface ;
- la cohérence d'activation entre une entité et ses enfants ;
- les types des entités, tels que les entités persistantes ayant une exécution longue et les entités transitoires dont l'exécution est éphémère ;
- le respect des liens de causalité dans les réactions d'une interface ; et enfin
- la terminaison et la confluence des réactions.

Ces deux caractéristiques ainsi que ces propriétés et traits semblent couvrir l'ensemble de la sémantique des UIDLs mais en est-on sûrs ? Comment pouvons-nous évaluer le pouvoir expressif d'un UIDL ? Ici, par expressivité, nous parlons d'expressivité spatio-interactifs, i.e. la capacité du langage à exprimer des idées spatiales et interactives. En effet, un UIDL permet de décrire des interfaces utilisateur et donc chaque instruction de ce langage exprime une idée à caractère spatial ou interactif. Une comparaison des pouvoirs expressifs d'UIDLs est faite dans [4], mais est-il possible d'avoir un moyen formel, permettant de déterminer les limitations d'un UIDL en termes d'expressivité spatiale et interactive, tel que la notion d'expressivité dans la théorie de la calculabilité ?

7.2 Formalisation de la sémantique des UIDLs

Nous répondons à la deuxième problématique avec la réflexion qui a eu lieu dans le chapitre 3 et qui nous a permis d'appuyer le choix des bigraphes pour formaliser la sémantique des UIDLs. Ce chapitre, présente un sous-ensemble du langage Smala ainsi que deux versions de sa sémantique que nous avons exprimée avec deux formalismes différents qui sont les bigraphes et la logique de premier ordre accompagnée de règles d'inférence. Une comparaison de ces sémantiques est ensuite faite afin de déterminer lequel des deux formalismes a une meilleur représentation des connaissances dans notre contexte.

La sémantique formalisée par la théorie des bigraphes a été jugée comme plus adéquate pour représenter la sémantique des UIDLs. En effet, celle-ci est plus facile à comprendre grâce à sa notation graphique, exécute un programme du sous-ensemble avec un nombre inférieur de pas et est constituée de moins de règles. Ces critères mettent en évidence les approches différentes des formalismes utilisés. L'un est plus à même de décrire la topologie d'un système ainsi que ses réactions alors que l'autre est plus favorable à la description de l'état d'un système, dans notre contexte en termes d'activation et de mémoire.

La définition de l'UIDL *Biguil*, présenté dans le chapitre 4 et ayant pour fondation la théorie des bigraphes, confirme notre choix. *Biguil* est accompagné d'une sémantique entièrement basée sur la théorie des bigraphes, qui par nature, permet de formaliser simplement les caractéristiques définissant les UIDLs et les propriétés qui en découlent. Cela nous permet d'avoir un système réactif bigraphique pour chaque interface utilisateur décrite qui correspond à un modèle de l'interface qui permet de l'étudier.

Au sujet de la fiabilité de cette formalisation, le chapitre 6 présente une preuve qui montre que la sémantique active chaque PROCESSUS en cohérence avec l'état d'activation de son parent. Cette preuve montre seulement une propriété parmi les cinq présentées et les autres propriétés restent à prouver. Nous envisageons de faire la preuve de ces cinq propriétés via l'assistant de preuve Coq mais avant cela, il faut définir en Coq un cadre permettant de manipuler et raisonner sur les bigraphes.

Nous terminons cette section sur les limitations de notre formalisme. Cette thèse s'ancre dans un projet visant à générer du code vérifié pour des interfaces graphiques. Cependant la formalisation définie couvre uniquement l'aspect spatial et interactif de la sémantique et ne donne aucune information sur les traces laissées en mémoire par l'interface modélisée ou bien les calculs effectués par celle-ci. Ici, dans le but de générer du code vérifié depuis notre formalisme, il faudrait que nous nous questionnions sur l'intégration de ces aspects manquants. Dans le chapitre 3 nous avons vu que le formalisme utilisé a un impact sur la représentation des informations et que cela est dû à l'approche adoptée par celui-ci. Les bigraphes ont pour but de décrire des traits spatiaux et interactifs d'un système et leur utilisation pour représenter une chose contre nature, tel que l'arithmétique, risquerait d'affecter la représentation des connaissances de notre formalisation.

Ainsi nous nous posons la question suivante :

> Comment exprimer convenablement la sémantique d'un langage qui donne la possiblité à son utilisateur d'exprimer des idées de types différents (e.g. spatio-interactif et informatique) ?

Pour répondre à cette question, il est envisageable d'étendre le formalisme mais une autre solution, généralisable aux langages de programmation rencontrant le même problème, pourrait être envisagée. La communauté de modélisation des systèmes cyber-physiques a déjà fait face à ce problème. En effet, les systèmes cyber-physiques peuvent être décomposés en différentes sous-parties assez hétérogènes. Afin de modéliser ces systèmes, des modèles dit à multi-paradigmes ont émergé [20] permettant de formaliser les différentes parties de natures différentes du système. Nous nous demandons donc s'il est possible d'envisager de faire de même avec la sémantique des langages de programmation proposant des instructions hétérogènes tel que les UIDLs ?

7.3 Un langage de description aux instructions atomiques

Mise à part la formalisation en bigraphes de la sémantique de *Biguil*, nous présentons dans le chapitre 4 l'ensemble d'instructions définissant notre UIDL. Cet ensemble est formé des instructions suivantes :

- les Components, qui sont des conteneurs de PROCESSUS auxquels nous pouvons associer une fonction (e.g. FSM, Adder) ;
- les GComponents, qui sont des conteneurs de PROCESSUS auxquels nous pouvons associer une fonction graphique (e.g. Frame, Circle), ces deux PROCESSUS introduisent les concepts spatiaux du langage ;
- les Spikes qui permettent de propager une activation Transitoire et pouvant être associé d'une fonction (e.g. Release, Press) ;
- les BINDINGS, qui sont des flux de communications pour les PROCESSUS ;

— les ASSIGNMENTS, qui permettent aux PROPERTIES de communiquer leur données, ces trois derniers PROCESSUS ainsi que les Components et les GComponents introduisent les concepts interactifs de *Biguil*.

Dans le chapitre 5, nous avons illustré l'expressivité, spatio-interactive, de *Biguil* en implémentant différents concepts de SMALA avec *Biguil*. Nous avons montré que *Biguil* permettait de définir les machines à état, les CONNECTORS, des composants graphiques (e.g. fenêtre, PROCESSUS de couleur, PROCESSUS de forme géométrique) et différents opérateurs (e.g. arithmétiques et booléens) de SMALA. Les transformations que nous avons définies dans le chapitre 5 permettent d'implémenter 70% du langage SMALA [1]. Les PROCESSUS SMALA que nous n'avons pas pu implémenter sont des PROCESSUS utilisant des concepts impératifs qui ne sont pas couverts par *Biguil*, tel que l'utilisation de la mémoire ou de structures de données et les PROCESSUS impliqués dans le concept de création dynamique de SMALA (section 2.2.3). Ce dernier point nous prouve que *Biguil* n'est pas assez expressif, en terme spatio-interactif, pour définir des opérateurs dynamiques qui pourraient, par exemple, permettre de déplacer ou créer des PROCESSUS pendant l'exécution d'une interface. Cela nous fait revenir au problème soulevé dans la section 7.2 au sujet du calcul de l'expressivité spatio-interactive d'un UIDL. En effet, une telle notion permettrait de donner une idée claire sur les notions spatio-interactives exprimable par un ensemble d'instructions.

7.4 Perspectives envisagées pour la suite du compilateur

Dans les chapitres 5 et 6, nous avons présenté un pipeline de vérification pour les BRS représentant les interfaces utilisateur décrites avec *Biguil* et nous avons montré comment il était possible de compiler le langage SMALA vers *Biguil*. Ces approches, permettent d'effectuer des vérifications préliminaires, telles que vérifier la validité des propriétés présentées dans le chapitre 4, sur des interfaces graphiques modélisées avec *Biguil* ou SMALA.

La prochaine étape de ce projet serait donc de générer du code respectant les spécifications spatiales et interactives des modèles décrits. Pour cela, selon la faisabilité (i.e. la possibilité d'exprimer les concepts de *Biguil*), nous aimerions utiliser soit le compilateur Vélus, soit le compilateur CompCert, mais cela semble être une tâche délicate pour plusieurs raisons.

L'une d'elles est la modélisation des interfaces utilisateur dans les langages Lustre et C. Les interfaces utilisateur et leurs aspects spécifiques seront difficiles à modéliser avec ces langages. Par exemple, nous pensons aux hiérarchies entre les composants graphiques qui seraient difficiles à représenter en Lustre ou bien aux différentes interactions entre composants en C.

Un autre problème est la génération de code correspondant aux composants graphiques. Aucun des deux compilateurs ne supporte de bibliothèque graphique et donc par conséquent générer du code vérifié correspondant à la partie graphique de l'interface graphique. Il est toutefois possible de lier des bibliothèques graphiques C lors de la compilation d'un fichier avec CompCert et ainsi avoir une partie du code vérifié. La partie du code qui sera vérifiée concernera la structure de l'interface graphique ainsi que ses liens de causalité et non l'affichage des composants graphiques qui est gérée par la bibliothèque.

Enfin, les instructions hétérogènes que les UIDLs contiennent nous poussent à nous questionner sur comment les compiler efficacement d'un point de vue vérification formelle avec assistant de preuve. Dans la section 7.2, nous avons soulevé une piste concernant la formalisation de la sémantique des langages de programmation, proposant des instructions hétérogènes. Cette piste correspond à la possibilité de modéliser leur sémantique avec des modèles à multi-paradigmes. Cette précédente question en soulève d'autres concernant leur compilation : comment est-ce que l'utilisation de tels modèles

1. sur 125 PROCESSUS définissant la bibliothèque standard de SMALA nous avons pu en implémenter 88

dans la compilation vérifiée est possible et est-ce que leur utilisation améliore la représentation des connaissances tout en facilitant le processus de vérification d'un compilateur ?

Nous terminons cette thèse avec la figure 7.1 qui résume les parties implémentées de notre projet et celles envisagées. Les parties bleus correspondent aux parties qui ont été implémentées. Parmi celles-ci, nous retrouvons la définition de *Biguil*, l'implémentation de la compilation de Smala vers *Biguil* et la vérification de propriétés, via BigraphER et PRISM, sur des interfaces graphiques. À propos de la vérification de propriétés, nous avons juste présenté la possibilité de vérifier les propriétés de la sémantique de *Biguil* sur des interfaces mais il est tout à fait possible de vérifier d'autres propriétés spécifiques aux interfaces. Les parties noires représentent celles que nous envisageons d'implémenter. Nous aimerions compiler d'autres langages de description d'interface graphique vers *Biguil* tels que FXML ou QML, cela permettrait de continuer notre étude sur l'expressivité de notre UIDL et faire profiter aux utilisateurs de ces langages d'un côté formel pour leurs implémentations. Enfin, comme dit précédemment, nous envisageons de générer du code exécutable (la génération de code Lustre ou C est une potentielle option) et d'implémenter notre formalisme en Coq. Avoir notre formalisme implémenté en Coq, permettrait de prouver des propriétés sur les interfaces graphiques décrites et de prouver que leur sémantique est préservée tout au long de leur compilation.

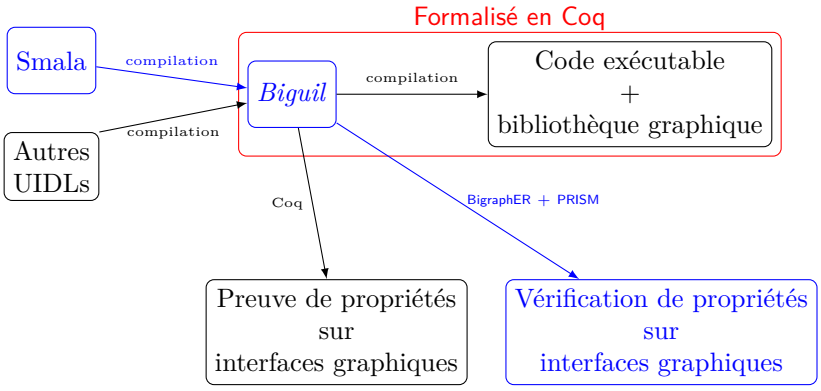


Figure 7.1 – Perspectives envisagées. Les parties en bleues ont été présentées dans la thèse. Les parties noires ainsi que le bloc rouge doivent être implémentés.

Annexe A

Appendix

A.1 Code **BigraphER** correspondant à la sémantique de *Biguil*

```
# Processus
# A process is defined Process.(Ident.id|State.id|Semantic.id|Type.id
    |Exec.1)
ctrl Process = 1;
ctrl GProcess = 1;

#Control for User interaction
ctrl User = 0;

atomic ctrl ActivateD = 0; # Activation from user interaction done
atomic ctrl IsActD = 0; # Initial activation by user interaction done
atomic ctrl IdleD = 0;

############## Semantics of process ******************
ctrl Semantic = 0; # Wrapper Semantic.component, Semantic.Spike etc..
ctrl Init = 0;

##### Component #######
ctrl Component = 0;
ctrl Func = 0;
# Component functions
atomic ctrl Frame = 0;
atomic ctrl Rectangle = 0;
atomic ctrl Circle = 0;
atomic ctrl FillColor = 0;
atomic ctrl Exit = 0;
########################

######## Spike #########
ctrl Spike = 0;
ctrl Event = 0;
```

```
31 ########################
32
33 ######## Property  #########
34 ctrl Property = 0;
35 atomic fun ctrl Double(x)= 0;
36 atomic fun ctrl Int(x)= 0;
37 atomic fun ctrl String(x)= 0;
38
39
40 ###### Binding #########
41 # Binding.(II|Source{x}|Target{y} id)
42 ctrl Binding = 0;
43 ### Binding types I and O like the symbol ON and Off
44 atomic ctrl II = 0;
45 atomic ctrl IO = 0;
46 atomic ctrl OO = 0;
47 atomic ctrl OI = 0;
48 ### Binding components
49 atomic ctrl Source = 1; # Control which will allow to "typed" links
50 atomic ctrl Target = 1; # Control which will allow to "typed" links
51 ########################
52
53 ###### Assignment ######
54 ctrl Assignment = 0;
55 ctrl TempA = 0;
56 ########################
57
58 ********************************************************
59
60 ############ Type of process ###############
61 ctrl Type = 0; # Wrapper Type.Persistent or Type.Transitional
62 ## Type of process
63 # Process once activated stay activated till their deactivation
64 atomic ctrl Persistent = 0;
65 # Process once activated do what they have to do according to their
66 semantics and then return in a deactivate state
67 atomic ctrl Transient = 0;
68 ********************************
69
70 ############ State of process ###############
71 ctrl State = 0; #Wrapper State.IsAct or State.Off
72 ## State of execution
73 atomic ctrl On  = 0;
74 atomic ctrl Off = 0;
75 ********************************************
76
77 ############# Ident of Process #############
```

```
78 ctrl Ident = 0; # Wrapper Ident.Id(x)
79 ## Unique Id for process
80 atomic fun ctrl Id(x)= 0;
81 ##########################################
82
83 ############ Execution state of Process ##############
84 # These controls are mainly used to implement a topological
85 # sort which is going to define the execution order of the processes.
86 ctrl ExecState = 0;
87 ctrl BindType = 0;
88 ## Exec
89 ctrl SrcPropagating = 0;
90 atomic ctrl Propagating = 0;
91 atomic ctrl Idle = 0;
92 atomic ctrl Synchronising = 0;
93 atomic ctrl SynchronisingD = 0;
94 # Flags asking a process to go active and deactive
95 atomic ctrl Activate = 0;
96 atomic ctrl Deactivate = 0;
97 atomic ctrl Processing = 0;
98 atomic ctrl ProcessingD = 0;
99 atomic ctrl IsAct = 0;
100 ####################################################
101
102
103 ####################################################
104 # Lots of rules for synchronisations/propagation that have a very
105 # similar shape.  Suggest we reduce these a bit by adding extra
106 # computation (also makes it clearer what's going on, i.e. what
107 # the II/IO etc means) We can hide these with instantanous rules
108 #so transition systems will be the same (but easier to modify)
109 ctrl CalcSync = 0;
110 ctrl CalcProp = 0;
111 ctrl SrcState = 0;
112 ctrl SrcExecState = 0;
113 ctrl TrgState = 0;
114 ctrl OldState =0;
115 ctrl BindState = 0;
116
117
118 ctrl CheckPred = 2;
119 atomic ctrl T = 0;
120 atomic ctrl F = 0;
121
122 atomic ctrl SP = 1 ;
123 ctrl SetOn = 0;
124 ctrl ToActivate = 0;
```

```
125 ctrl ToDeactivate = 0;
126 ctrl Next = 0;
127
128 ######## Reaction rules for user interactions **********
129
130 react user_setOn =
131 SetOn.(SP{x} | Next.id) || Process{x}.(State.Off | id)
132 -[1]->
133 SetOn.id || Process{x}.(State.On | id);
134
135 react user_ToActivate =
136 ToActivate.(SP{x} | Next.id) || Process{x}.(ExecState.1 | id)
137 -[1]->
138 ToActivate.id || Process{x}.(ExecState.Activate | id);
139
140 react user_ToDeactivate =
141 ToDeactivate.(SP{x} | Next.id) || Process{x}.(ExecState.1 | id)
142 -[1]->
143 ToDeactivate.id || Process{x}.(ExecState.Deactivate | id);
144
145 react user_isAct =
146 Process{a}.(State.Off | User.(IsAct | id) | id )
147 -[1]->
148 Process{a}.(State.On | User.id | id);
149
150 react user_activate =
151 Process{a}.(State.Off | User.(Activate | id) | ExecState.1 | id)
152 -[1]->
153 Process{a}.(State.Off | User.id | ExecState.Activate | id);
154
155 react user_idle =
156 Process{a}.(State.Off | User.(Idle | id) | ExecState.1 | id)
157 -[1]->
158 Process{a}.(State.Off | User.id | ExecState.Idle | id);
159
160 react clean_user = User.1 -[1]-> 1;
161
162 ######### Reaction rules for components activation ****************
163 react init_act_children =
164 Process{a}.(State.Off|Semantic.Component.id|ExecState.Activate|id)
165 -[1]->
166 Process{a}.(State.On|Semantic.Component.id|ExecState.Processing|id);
167
168 react act_children =
169 Process{a}.(State.On
170   | Semantic.Component.(Process{b}.(State.Off
171           | Init.On
```

```
172             | ExecState.1
173             | id)
174               | id)
175   | ExecState.Processing
176   | id
177   )
178 -[1]->
179 Process{a}.(State.On
180   | Semantic.Component.(Process{b}.(State.Off
181             | Init.On
182             | ExecState.Activate
183             | id)
184              | id)
185   | ExecState.Processing
186   | id
187   );
188
189 react act_Pr =
190 Process{a}.(State.Off | ExecState.Activate | id)
191 -[1]->
192 Process{a}.(State.On | ExecState.Propagating | id)
193 if !Semantic.Component in param,
194    !Semantic.Assignment in param;
195
196 react temporise_assignment =
197 Process{a}.(State.Off|ExecState.Activate|Semantic.Assignment.id|id)
198 -[1]->
199 Process{a}.(State.Off|ExecState.TempA.1|Semantic.Assignment.id|id);
200
201 react act_component =
202 Process{a}.(State.On|Semantic.Component.id|ExecState.Processing|id)
203 -[1]->
204 Process{a}.(State.On|Semantic.Component.id|ExecState.Propagating|id);
205
206
207 ########## Reaction rules for component deactivations
      ******************
208 react deact_children_init =
209 Process{a}.(State.On|Semantic.Component.id|ExecState.Deactivate|id)
210 -[1]->
211 Process{a}.(State.On|Semantic.Component.id|ExecState.ProcessingD|id);
212
213 react deact_children =
214 Process{a}.(State.On
215   | Semantic.Component.(Process{b}.(State.On|ExecState.1|id)|id)
216   | ExecState.ProcessingD
217   | id)
```

```
-[1]->
Process{a}.(State.On
  | Semantic.Component.(
         Process{b}.(State.On|ExecState.Deactivate|id)|id)
         | ExecState.ProcessingD
         | id
        ) @ [0,1,2];

react deact_Pr =
Process{a}.(State.On | ExecState.Deactivate | id)
-[1]->
Process{a}.(State.Off | ExecState.Propagating | id)
if !Semantic.Component in param;

react deact_component =
Process{a}.(State.On|Semantic.Component.id|ExecState.ProcessingD|id)
-[1]->
Process{a}.(State.Off|Semantic.Component.id|ExecState.Propagating|id)
    ;

######## Reaction rules for activation through bindings **********
react src_propagating =
Process{a}.(State.id | ExecState.Propagating | id) ||
Process{b}.(State.On
  | ExecState.1
  | Semantic.Binding.(id | Source{a} | Target{c})
  | id)
-[1]->
Process{a}.(State.id | ExecState.Propagating | id) ||
Process{b}.(State.On
  | ExecState.SrcPropagating.(SrcState.id | SrcExecState.Propagating)
  | Semantic.Binding.(id | Source{a} | Target{c})
  | id)
  @[0,1,0,2,3];

react trg_propagating =
Process{b}.(State.On
  | ExecState.SrcPropagating.(SrcState.id | SrcExecState.id)
  | Semantic.Binding.(id | Source{a} | Target{c}) | id) ||
Process{c}.(State.id | ExecState.1 | id)
-[1]->
Process{b}.(State.On
  | ExecState.Idle
  | Semantic.Binding.(id | Source{a} | Target{c})
  | id) ||
Process{c}.(State.id
  | ExecState.CalcProp.(SrcState.id | TrgState.id | BindType.id |
```

```
      SrcExecState.id | BindState.On)
  | id)
  @ [2,3,4,0,4,2,1,5];

react multi_trg_propagating =
Process{b}.(State.On
  | ExecState.SrcPropagating.(SrcState.id | SrcExecState.id)
  | Semantic.Binding.(id | Source{a} | Target{d})
  | id) ||
Process{c}.(State.On|Semantic.Binding.(id|Source{e}|Target{d})|id)||
Process{d}.(State.id | ExecState.id | id)
-[1]->
Process{b}.(State.On
  | ExecState.Idle
  | Semantic.Binding.(id | Source{a} | Target{d})
  | id) ||
Process{c}.(State.On|Semantic.Binding.(id|Source{e}|Target{d})|id) ||
Process{d}.(State.id
  | ExecState.CalcSync.(SrcState.id
            | TrgState.id
            | BindType.id
            | OldState.id
            | SrcExecState.id
            | BindState.On)
  | id)
  @[2,3,4,5,6,0,6,4,7,1,8];

######################## Topological sorting
    ***************************
react calc_sync_II = CalcSync.(SrcState.On
                 | TrgState.Off
            | BindType.II
                      | OldState.id
            | SrcExecState.Propagating
            | BindState.On) -[1]->  Synchronising @[];

react calc_sync_OI = CalcSync.(SrcState.Off
                 | TrgState.Off
            | BindType.OI
            | OldState.id
            | SrcExecState.Propagating
            | BindState.On) -[1]->  Synchronising @[];

react calc_sync_IO = CalcSync.(SrcState.On
            | TrgState.On
            | BindType.IO
            | OldState.id
```

```
309             | SrcExecState.Propagating |
310             BindState.On) -[1]->   SynchronisingD @[];
311
312 react calc_sync_OO = CalcSync.(SrcState.Off
313             | TrgState.On
314             | BindType.OO
315             | OldState.id
316             | SrcExecState.Propagating
317             | BindState.On) -[1]->  SynchronisingD @ [];
318
319 react calc_sync_restore_old =
320 CalcSync.(id | OldState.id | SrcExecState.Propagating) -[1]-> id @
        [1];
321
322 react calc_sync_max_idle =
323 CalcSync.(OldState.id | SrcExecState.Idle | id) -[1]-> Idle @[]
324 if !(SynchronisingD) in param,
325    !(Synchronising) in param ;
326
327 react calc_sync_min_idle =
328 CalcSync.(OldState.id | SrcExecState.Idle | id) -[1]-> id @[0];
329
330 react calc_prop_II = CalcProp.(SrcState.On
331             | TrgState.Off
332             | BindType.II
333             | SrcExecState.Propagating
334             | BindState.On) -[1]-> Activate;
335
336 react calc_prop_IO = CalcProp.(SrcState.On
337             | TrgState.On
338             | BindType.IO
339             | SrcExecState.Propagating
340             | BindState.On) -[1]-> Deactivate;
341
342 react calc_prop_OI = CalcProp.(SrcState.Off
343             | TrgState.Off
344             | BindType.OI
345             | SrcExecState.Propagating |
346             BindState.On) -[1]-> Activate;
347
348 react calc_prop_OO = CalcProp.(SrcState.Off
349             | TrgState.On
350             | BindType.OO
351             | SrcExecState.Propagating
352             | BindState.On) -[1]-> Deactivate;
353
354 react calc_prop_idle = CalcProp.id -[1]-> Idle @ [];
```

```
355
356 # To make it easier to see multiple syncs we will just
357 # tag them in the model (since we match on states not rules)
358 ctrl Labels = 0;
359 atomic fun ctrl SyncAct(n) = 0;
360
361 fun react synch_to_activate(n) =
362 Process{a}.(State.Off | ExecState.Synchronising | Id(n) | id) ||
363 Labels.id
364 -[1]->
365 Process{a}.(State.Off | ExecState.Activate | Id(n) | id ) ||
366 Labels.(id | SyncAct(n));
367
368 react synch_to_deactivate =
369 Process{a}.(State.On | ExecState.SynchronisingD | id)
370 -[1]->
371 Process{a}.(State.On | ExecState.Deactivate | id );
372
373 react propagatingP_to_idle =
374 Process{a}.(ExecState.Propagating | Type.Persistent | id)
375 -[1]->
376 Process{a}.(ExecState.Idle | Type.Persistent | id);
377
378 react propagatingT_to_idle =
379 Process{a}.(State.On | ExecState.Propagating | Type.Transient | id)
380 -[1]->
381 Process{a}.(State.Off | ExecState.Idle | Type.Transient | id);
382
383 react idle_to_1 =
384 Process{a}.(ExecState.Idle | id)
385 -[1]->
386 Process{a}.(ExecState.1 | id);
387
388 ########## Reaction rules for the assignments ###########
389
390 react check_pred =
391 Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
392   | ExecState.TempA.1
393   | id) ||
394 Process{e2}.(State.Off
395   | Semantic.Assignment.(Target{a} | Source{d})
396   | ExecState.TempA.id
397   | id)
398 -[1]->
399 Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
400         | ExecState.TempA.F
401   | id) ||
```

```
Process{e2}.(State.Off
  | Semantic.Assignment.(Target{a} | Source{d})
  | ExecState.TempA.id
  | id);

react check_pred_multi_targ_1 =
Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
  | ExecState.TempA.1
  | id) ||
Process{e2}.(State.Off
  | ExecState.TempA.F
  | Semantic.Assignment.(Target{c} | Source{a2})
  | id)
-[1]->
Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
        | ExecState.TempA.F
  | id) ||
Process{e2}.(State.Off
  | ExecState.TempA.F
  | Semantic.Assignment.(Target{c} | Source{a2})
  | id);

react check_pred_multi_targ_2 =
Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
  | ExecState.TempA.1
  | id) ||
Process{e2}.(State.Off
  | ExecState.TempA.1
  | Semantic.Assignment.(Target{c} | Source{a2})
  | id)
-[1]->
Process{e1}.(Semantic.Assignment.(Source{a} | Target{c})
  | ExecState.TempA.1
  | id) ||
Process{e2}.(State.On
  | ExecState.Propagating
  | Semantic.Assignment.(Target{c} |  Source{a2})
  | id);

react no_pred = TempA.1 -[1]-> TempA.T ;

react assign_act_targ =
Process{b}.(State.Off
  | ExecState.TempA.T
  | Semantic.Assignment.(Source{a} | Target{c}) | id) ||
Process{c}.(Semantic.Component.(Func.Property.id
            | Process{d}.(State.Off
```

```
449             | Semantic.Spike.id
450             | ExecState.1
451             | id)
452               )
453       | id)
454 -[1]->
455 Process{b}.(State.On
456   | ExecState.Propagating
457   | Semantic.Assignment.(Source{a}|Target{c})
458   | id) ||
459 Process{c}.(Semantic.Component.(Func.Property.id
460               | Process{d}.(State.Off
461             | Semantic.Spike.id
462             | ExecState.Activate
463             | id)
464          )
465      | id);
466
467 react assign_act =
468 Process{b}.(State.Off
469   | ExecState.TempA.T
470   | Semantic.Assignment.(Source{a} | Target{c})
471   | id)
472 -[1]->
473 Process{b}.(State.On
474   | ExecState.Propagating
475   | Semantic.Assignment.(Source{a}|Target{c})
476   | id);
477
478
479 react clean_check  = TempA.F -[1]-> TempA.1;
480
481 ############## Condition on parents ############################
482
483 react parent_deact =
484 Process{a}.(State.Off
485   | Semantic.Component.(Process{b}.(ExecState.Activate|id) | id)
486   | id)
487 -[1]->
488 Process{a}.(State.Off
489   | Semantic.Component.(Process{b}.(ExecState.1|id) | id)
490   | id);
491
492 ############## Activation of graphical processes #################
493 react gProcess_act =
494 Process{a1}.(State.On | id) ||
495 GProcess{a1}.(State.Off | id)
```

```
-[1]->
Process{a1}.(State.On | id) ||
GProcess{a1}.(State.On | id);

react gProcess_deact =
Process{a1}.(State.Off | id) ||
GProcess{a1}.(State.On | id)
-[1]->
Process{a1}.(State.Off | id) ||
GProcess{a1}.(State.Off | id);

################## Predicates to check  ************************
fun big pAct(n) = Process{x}.(State.On | Ident.Id(n) | id);
fun big pDeact(n) = Process{x}.(State.Off | Ident.Id(n) | id);
fun big transitional(n) = Process{x}.(Type.Transient|Ident.Id(n)|id);
fun big multisync(n) = Labels.(SyncAct(n) | SyncAct(n) | id);
big parentChildrenConsistency =
Process{a}.(State.Off
  | Semantic.Component.(Process{b}.(State.On | id) | id)
  | id);
big logicalGraphicalConsistency1 =
Process{a}.(State.Off | id) || GProcess{a}.(State.On | id);
big logicalGraphicalConsistency2 =
Process{a}.(State.On | id) || GProcess{a}.(State.Off | id);
big syncp = Synchronising;
big syncDp = SynchronisingD;
big actp = Activate;
big deactp = Deactivate;
big processingp = Processing;
big processingDp = ProcessingD;
big actAp = TempA;
big propagatingp = Propagating;
big idlep = Idle;
big calcPropp = CalcProp.id;
big calcSyncp = CalcSync.id;
```

A.2 Exemples de programmes Smala pour le pipeline de vérification

Cette section présente différent exemples SMALA pour le pipeline de vérification. En début de chaque de programme un commentaire l'expliquant est donné.

A.2.1 Exemple BindingII

```
/*
Test bindingII propagation
```

```
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {

  Spike a [@activate]
  Spike b

  a ->[@isactive] b
}
```

A.2.2 Exemple BindingIO

```
/*
Test binding IO propagation
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {
  Spike a
  Int b [@isactive] (23)
  a[@activate] ->![@isactive] b

}
```

A.2.3 Exemple BindingOI

```
/*
Test BindingOI propagation
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {
  Int a[@isactive] (42)
  Spike s1
  Spike s2
```

```
  s1[@activate] ->![@isactive] a
  a !->[@isactive] s2
}
```

A.2.4 Exemple BindingOO

```
/*
Test bindingOO propagation
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {
  Int a[@isactive] (32)
  Int b[@isactive] (31)
  Spike s
  s[@activate] ->![@isactive] a
  a !->![@isactive] b

}
```

A.2.5 Exemple Deactivation diamond

```
/*
  Concurrent deactivation
        In this test, several processes try to deactivate e. e is
           only deactivate once because all the deactivation are
           synchronised.
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {

  Int a (43)
  Int b [@isactive] (45)
  Int c [@isactive] (47)
  Int d [@isactive] (48)
  Int e [@isactive] (49)

  a[@activate] ->![@isactive] b
  a ->![@isactive] c
```

```
21   c !->![@isactive] d
22   d !->![@isactive] e
23   b !->![@isactive] e
24
25 }
```

A.2.6 Exemple Simple data flow

```
1 /*
2 Test for connectors transitivity
3 */
4 use core
5 use base
6 use display
7 use gui
8
9 _main_
10 Component root [@isactive] {
11   Frame f [@isactive] ("Exemple", 0, 0, 500, 500)
12
13   Double a  [@isactive] (0)
14   Double b  [@isactive] (0)
15
16   f.press.x [@activate] => [@isactive] a
17   a => [@isactive] b
18
19
20
21 }
```

A.2.7 Exemple Diamond broken link

```
1 /*
2  In this test the binding in line 18 is not activated. The
       deactivation is only propagated to processes c, d and e.
3 */
4 use core
5 use base
6 use display
7 use gui
8
9 _main_
10 Component root [@isactive] {
11
12   Spike a
13   Spike b
14   Int c [@isactive] (47)
15   Int d [@isactive] (48)
```

```
16   Int e [@isactive] (49)
17
18   a [@activate] ->  b
19   a ->![@isactive] c
20   c !->![@isactive] d
21   d !->![@isactive] e
22   b ->[@isactive] e
23
24 }
```

A.2.8 Exemple Adder small

```
1  /*
2    Adder small
3
4  This test shows that the property result is only updated once even if
       its both operand are updated.
5  */
6  use core
7  use base
8  use display
9  use gui
10
11 _main_
12 Component root [@isactive] {
13   Frame f  [@isactive] ("Example Adder", 0, 0, 500, 500)
14
15   Double a [@isactive] (0)
16   Double b [@isactive] (0)
17   Double c [@isactive] (0)
18   Double d [@isactive] (2)
19   Adder add1 [@isactive](0,2)
20   Adder add2 [@isactive](0,0)
21
22
23   f.press.x [@activate] => [@isactive] a
24   f.press.x => [@isactive] b
25
26   a => [@isactive] add1.right
27   add1.result => [@isactive] add2.left
28   b => [@isactive] add2.right
29   add2.result => [@isactive] c
30
31 }
```

A.2.9 Exemple Diamond

```
/*
This test shows that a4 is activated once.
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {

Spike a1
Spike a2
Spike a3
Spike a4

a1[@activate] ->[@isactive] a2
a1 ->[@isactive] a3
a3 ->[@isactive] a4
a2 ->[@isactive] a4

}
```

A.2.10 Exemple Arith

```
/*
This test check the semantics of an assignment when it has an
    arithmetic operation as source.
*/
use core
use base
use display
use gui

_main_
Component root [@activate] {

  Double a (22)
  Double b (0)

  a + 2 =:b

}
```

A.2.11 Exemple Double activation assignment

```
  /*
```

```
2   This test check the semantics of an assignment. It shows that only
       the component of each properties are activated and not their
       spikes
3 */
4 use core
5 use base
6 use display
7 use gui
8
9 _main_
10 Component root [@activate]  {
11
12   Int a (42)
13   Int b (0)
14
15   a =: b
16
17
18 }
```

A.2.12 Exemple Assignment2

```
1 /*
2 This test checks the activation order of these assignments which is
     determined by their causal links.
3 */
4 use core
5 use base
6 use display
7 use gui
8
9 _main_
10 Component root [@activate] {
11
12   Double a (22)
13   Double b (0)
14   Double c (0)
15
16   a =:b
17   b =:c
18 }
```

A.2.13 Exemple Connector

```
1 /*
2 This test checks the semantics of a connector
3 */
4 use core
```

```
use base
use display
use gui

_main_
Component root [@isactive]{

Int a (32)
Int b[@isactive] (31)

a[@activate] =>[@isactive] b

}
```

A.2.14 Exemple Deactivation Component

```
/*
This test check the  deactivation of a component
*/
use core
use base
use display
use gui

_main_
Component root [@isactive] {
  Spike a

  Component comp [@isactive] {
    Int b[@isactive] (42)
  }

  a[@activate] ->![@isactive] comp

}
```

A.2.15 Exemple Adder Big

```
/*
Test checking concurrency on adders
*/
use core
use base
use display
use gui

_main_
Component root [@isactive]  {
```

```
11   Frame f [@isactive] ("Example Adder", 0, 0, 500, 500)
12
13   Double a [@isactive] (0)
14   Double b [@isactive] (0)
15   Double c [@isactive] (0)
16
17   Adder add1 [@isactive] (1,0)
18   Adder add2 [@isactive] (0,0)
19   Adder add3 [@isactive] (0,2)
20   Adder add4 [@isactive] (0,0)
21   Adder add5 [@isactive] (0,4)
22
23
24   f.press.x [@activate]  =>[@isactive]  a
25   f.press.x =>[@isactive]  b
26
27   a =>[@isactive]  add1.right
28   add1.result =>[@isactive]  add2.left
29   b =>[@isactive]  add2.right
30
31   add1.result =>[@isactive] add3.left
32
33   add2.result =>[@isactive] add4.left
34   add3.result =>[@isactive] add4.right
35
36   add4.result =>[@isactive] add5.left
37
38   add5.result =>[@isactive] c
39 }
```

A.2.16 Exemple Double choice

```
1  /*
2
3  In this test there is a conflict of propagation. r4 tries to activate
       r5 and r3 tries to deactivate it.
4  Since r5 is initially activated, r3 will win the clash and deactivate
       r5.
5  */
6  use core
7  use base
8  use display
9  use gui
10
11 _main_
12 Component root [@isactive]  {
13   Frame f[@isactive] ("Example Adder", 0, 0, 500, 500)
```

```
14
15   Rectangle r1 [@isactive] (0, 0, 50, 50, 5, 5)
16   Rectangle r2 [@isactive] (53, 0, 50, 50, 5, 5)
17   Rectangle r3 [@isactive] (106, 0, 50, 50, 5, 5)
18   Rectangle r4 [@isactive] (166, 0, 50, 50, 5, 5)
19   Rectangle r5 [@isactive] (220, 0, 50, 50, 5, 5)
20
21
22   f.press[@activate] ->[@isactive] r1
23   f.press ->![@isactive] r2
24   r2 !->![@isactive] r3
25   r1 ->[@isactive] r4
26   r3 !->![@isactive] r5
27   r4 ->[@isactive] r5
28 }
```

A.2.17 Exemple Assignment

```
1 /*
2
3 Test which checks assigment semantics, using complex arithmatics
      operations as source.
4 */
5 use core
6 use base
7 use display
8 use gui
9
10 _main_
11 Component root [@activate] {
12
13   Double a (22)
14   Double b (0)
15   Double c (3)
16
17   a/b+a*b+a-b =: c
18
19 }
```

A.2.18 Exemple Assignment 3

```
1 /*
2 This test shows a concurrency case with assigments. It checks if d is
       activated once.
3 */
4 use core
5 use base
6 use display
```

```
use gui

_main_
Component root [@activate] {

  Double a (22)
  Double b (0)
  Double c (0)
  Double d  (0)

  a =:b
  a =:c
  c =:d
  b =:d
}
```

A.2.19 Exemple SceneG

```
/*
Test showing the logical and graphical perspectives consistency

*/
use core
use base
use display
use gui

_main_
Component root[@isactive] {

    Frame f[@isactive] ("Frame",0,0,400,400)
    FillColor r (255,0,0)
    Rectangle rect (0, 0, 100, 50, 0, 0)

    FillColor g (0,255,0)
    Rectangle rect2 (0, 300, 100, 50, 0, 0)
}
```

A.2.20 Exemple Elastic window

```
/*
  Elastic window

Program dÃ©crivant une fenÃªtre elastique. A chaque fois que la
    hauteur de la fenÃªtre est redimensionnÃ©e alors elle revient Ã
sa taille d'origine qui est Ã©gale Ã  la largeur de la fenÃªtre.
Test qui illustre la transitivitÃ© entre bindings et les connector

```

```
Program describing an elastic window.
Each time the window height is resized then the window height is
    reinitialised to its origin value (equal to the width).
This test checks the transitivive aspect of bindings and connectors
*/

use core
use base
use display
use gui

_main_
Component root [@isactive]  {
 Frame f [@isactive]  ("elastic window",0,0,500,400)
 Int n [@isactive] (42)
 f.release [@activate] -> [@isactive] f.width
 n =: f.width
 f.width =:> [@isactive] f.height
}
```

A.2.21 Exemple ATM

```
/*
  ATM
Simplistic ATM proposing 6 amounts. Once the user chose an amount the
    ATM thanks her/him.
This test checks the finite state machine semantics.
*/

use core
use base
use display
use gui



_main_
Component root [@isactive] {
  Frame f[@isactive] ("ATM", 0, 0, 600, 400)
  Exit ex (0,1)
  f.close ->[@isactive] ex
  Spike click

  FSM fsm[@isactive] {
    State amount[@isactive] {
      FillColor white (0, 0, 255)
      Rectangle r (0, 0, 600, 400, 0, 0)
```

```
25
26        FontSize fsize (6,0.1)
27        FillColor white (255,255,102)
28        Text t1 (50, 120, "$20")
29        Text t2 (50, 210, "$40")
30        Text t3 (50, 310, "$60")
31
32        Text t4 (490, 120, "$80")
33        Text t5 (490, 210, "$100")
34        Text t6 (490, 310, "$200")
35
36        Text t7 (18, 40, "PLEASE SELECT REQUIRED AMOUNT")
37
38
39
40      }
41      State withdraw {
42
43       FillColor white (0, 0, 255)
44       Rectangle r (0, 0, 600, 400, 0, 0)
45       FontSize fsize (6,0.1)
46       FillColor white (255,255,102)
47       Text t7 (18, 40, "Don't forget to collect your money ;-)")
48
49
50      }
51      amount->[@isactive]withdraw (amount.t1.press[@activate])
52      amount->[@isactive]withdraw (amount.t2.press)
53      amount->[@isactive]withdraw (amount.t3.press)
54      amount->[@isactive]withdraw (amount.t4.press)
55      amount->[@isactive]withdraw (amount.t5.press)
56      amount->[@isactive]withdraw (amount.t6.press)
57
58    }
59
60
61
62
63  }
```

A.2.22 Exemple TCAS

```
1  /*
2    TCAS
3
4  Simplistic simulation of TCAS through FSM.
5  Check if FSM work well and logical and graphical perspectives
```

```
    consistensy in a big size example.
*/
use core
use base
use display
use gui

_main_
Component root {
  Frame f ("my frame", 0, 0, 600, 600)
  Exit ex (0, 1)
  f.close -> ex

  Component flight {
    NoFill noFillflight
    OutlineWidth oWflight (5)
    OutlineColor oCflight (255,255,0)
    Line p1 (300, 590, 300, 540)
    Line p2 (290, 585, 310, 585)
    Line p3 (275, 552.5, 325, 552.5)
  }
  Component circle1 {
    NoFill noFillCircle1
    OutlineWidth outWidthCircle1 (3)
    OutlineColor outColorCircle1 (255,255,255)
    Line l0(150.0,552.5,150.63987355574483,538.6597460805046)
    Line l2(152.55403504741474,524.9375723275144
           ,155.72615352407715,511.45055148918755)
    Line l4(160.12916558934663,498.3137500719271
           ,165.72550629674066,485.6392466335193)
    Line l6(172.46742964055787,473.53517556839665
           ,180.29741590796408,462.1048045431115)
    Line l8(189.14866241690112,451.4456534530164
           ,198.9456534530164,441.6486624169011)
    Line l10(209.60480454311153,432.7974159079641
           ,221.0351755683966,424.9674296405579)
    Line l12(233.13924663351924,418.22550629674066
           ,245.81375007192707,412.6291655893466)
    Line l14(258.95055148918755,408.2261535240772
           ,272.43757232751443,405.05403504741474)
    Line l16(286.1597460805047,403.1398735557448,300.0,402.5)
    Line l18(313.8402539194953,403.13987355574477,
           327.5624276724855,405.05403504741474)
    Line l20(341.04944851081245,408.2261535240772
           ,354.1862499280729,412.6291655893466)
    Line l22(366.86075336648065,418.2255062967406
           ,378.96482443160335,424.96742964055784)
```

```
52      Line l24(390.3951954568885,432.7974159079641
53              ,401.0543465469836,441.6486624169011)
54      Line l26(410.8513375830989,451.44565345301646
55              ,419.7025840920359,462.1048045431115)
56      Line l28(427.5325703594421,473.5351755683966
57              ,434.27449370325934,485.63924663351924)
58      Line l30(439.87083441065334,498.313750071927
59              ,444.2738464759228,511.45055148918755)
60      Line l32(447.44596495258526,524.9375723275144
61              ,449.36012644425523,538.6597460805048)
62  }
63
64
65
66      Component circle2 {
67       NoFill noFillCircle2
68       OutlineWidth outWidthCircle2 (3)
69       OutlineColor outColorCircle2 (255,255,255)
70       Line l0(0.0,552.5,0.5919814715185225,533.6628441412059)
71       Line l2(2.3655896056566235,514.9000299307087
72               ,5.313824781393407,496.2856056242826)
73       Line l4(9.425051661410691,477.8930338505436
74               ,14.683045111453964,459.7949016875158)
75       Line l6(21.067054233524573,442.0626341945966,
76              28.55188426019413,424.7662125304782)
77       Line l8(37.10799598684093,407.9738977694854
78                ,46.70162234939548,391.751961506301)
79       Line l10(57.29490168751576,376.16442431225806
80               ,68.84602716726323,361.2728030753931)
81       Line l12(81.30941177357653,347.1358682213934
82                ,94.63586822139342,333.8094117735765)
83       Line l14(108.77280307539309,321.34602716726323
84                ,123.66442431225803,309.7949016875158)
85       Line l16(139.25196150630103,299.2016223493955
86               ,155.47389776948546,289.6079959868409)
87       Line l18(172.26621253047819,281.05188426019413
88                ,189.56263419459657,273.5670542335246)
89       Line l20(207.29490168751576,267.18304511145396
90               ,225.3930338505436,261.9250516614107)
91       Line l22(243.7856056242826,257.8138247813934
92               ,262.4000299307087,254.86558960565668)
93       Line l24(281.16284414120594,253.09198147151852,300.0,252.5)
94       Line l26(318.837155858794,253.09198147151852
95               ,337.59997006929126,254.86558960565662)
96       Line l28(356.2143943757174,257.8138247813934
97               ,374.60696614945647,261.9250516614107)
98       Line l30(392.70509831248415,267.1830451114539
```

```
99                ,410.4373658054033,273.5670542335246)
100      Line l32(427.7337874695218,281.0518842601942
101               ,444.52610223051465,289.60799598684093)
102      Line l34(460.748038493699,299.2016223493955
103               ,476.3355756877419,309.7949016875158)
104      Line l36(491.2271969246069,321.34602716726323
105               ,505.36413177860663,333.8094117735766)
106      Line l38(518.6905882264234,347.13586822139337
107               ,531.1539728327367,361.2728030753931)
108      Line l40(542.7050983124842,376.164424312258
109               ,553.2983776506045,391.7519615063009)
110      Line l42(562.8920040131591,407.97389776948546
111               ,571.4481157398059,424.76621253047813)
112      Line l44(578.9329457664754,442.06263419459657
113               ,585.316954888546,459.79490168751573)
114      Line l46(590.5749483385894,477.8930338505434
115               ,594.6861752186066,496.28560562428265)
116      Line l48(597.6344103943434,514.9000299307087
117               ,599.4080185284815,533.6628441412059)
118    }
119
120    Component circle3 {
121      NoFill noFillCircle3
122      OutlineWidth outWidthCircle3 (3)
123      OutlineColor outColorCircle3 (255,255,255)
124      Line l0(-150.0,552.5,-149.3832906395582,528.9488196906752)
125      Line l2(-147.53485291572298,505.4621915295559
126               ,-144.459753267812,482.10449073189614)
127      Line l4(-140.16642033021253,458.9397391320083
128               ,-134.66662183008071,436.0314297038657)
129      Line l6(-127.97543233281908,413.4423525312737
130              ,-120.11119192374076,391.2344227046149)
131      Line l8(-111.09545593917039,369.46851061588995
132              ,-100.95293588476557,348.20427511720396)
133      Line l10(-89.7114317029974,327.5
134                ,-77.4017555754408,307.41243424323784)
135      Line l12(-64.05764746872637,287.99663646838707
136               ,-49.7156826556369,269.3058240275731)
137      Line l14(-34.41517146482744,251.39122713851378
138               ,-18.198051533946398,234.30194846605366)
139      Line l16(-1.1087728614862158,218.08482853517262
140                ,16.805824027573124,202.7843173443631)
141      Line l18(35.49663646838707,188.44235253127363
142                ,54.912434243237755,175.09824442455925)
143      Line l20(74.99999999999994,162.78856829700266
144               ,95.70427511720393,151.5470641152345)
145      Line l22(116.96851061588984,141.4045440608296
```

```
146             ,138.73442270461484,132.38880807625924)
147       Line l24(160.94235253127366,124.52456766718092
148             ,183.53142970386568,117.83337816991929)
149       Line l26(206.43973913200836,112.33357966978747
150             ,229.60449073189608,108.040246732188)
151       Line l28(252.96219152955595,104.96514708427702
152             ,276.4488196906753,103.1167093604418)
153       Line l30(299.9999999999999,102.5,
154              323.55118030932465,103.1167093604418)
155       Line l32(347.037808470444,104.96514708427696
156              ,370.39550926810386,108.040246732188)
157       Line l34(393.5602608679917,112.33357966978747
158             ,416.4685702961343,117.83337816991929)
159       Line l36(439.0576474687263,124.52456766718086
160             ,461.2655772953851,132.38880807625924)
161       Line l38(483.03148938411005,141.40454406082955
162             ,504.295724882796,151.54706411523443)
163       Line l40(524.9999999999999,162.7885682970026
164             ,545.087565756762,175.09824442455908)
165       Line l42(564.5033635316129,188.44235253127363
166             ,583.1941759724268,202.78431734436305)
167       Line l44(601.108772861486,218.0848285351725
168             ,618.1980515339463,234.3019484660536)
169       Line l46(634.4151714648274,251.39122713851373
170             ,649.7156826556369,269.3058240275732)
171       Line l48(664.0576474687264,287.99663646838707
172             ,677.4017555754408,307.4124342432377)
173       Line l50(689.7114317029974,327.5,700.9529358847656
174             ,348.2042751172039)
175       Line l52(711.0954559391705,369.46851061588995
176             ,720.1111919237408,391.2344227046149)
177       Line l54(727.9754323328191,413.4423525312736
178             ,734.6666218300807,436.03142970386574)
179       Line l56(740.1664203302125,458.9397391320083
180             ,744.459753267812,482.1044907318961)
181       Line l58(747.534852915723,505.46219152955604
182             ,749.3832906395583,528.9488196906752)
183     }
184
185     Clock c (200)
186     Spike change
187
188     FSM fligth1 {
189
190       State pos1 {
191         FillColor col (255,255,255)
192         Rectangle r (100, 10, 20, 20, 5, 5)
```

```
193       }
194
195
196       State pos2 {
197         FillColor col (255,255,255)
198         Rectangle r (100, 20, 20, 20, 5, 5)
199       }
200
201       State pos3 {
202         FillColor col (255,255,255)
203         Rectangle r (100, 30, 20, 20, 5, 5)
204       }
205
206       State pos4 {
207         FillColor col (255,255,255)
208         Rectangle r (100, 40, 20, 20, 5, 5)
209       }
210
211       State pos5 {
212         FillColor col (255,255,255)
213         Rectangle r (100, 50, 20, 20, 5, 5)
214       }
215
216       State pos6 {
217         FillColor col (255,255,255)
218         Rectangle r (100, 60, 20, 20, 5, 5)
219       }
220
221       State pos7 {
222         FillColor col (255,255,255)
223         Rectangle r (100, 70, 20, 20, 5, 5)
224       }
225
226       State pos8 {
227         FillColor col (255,255,255)
228         Rectangle r (100, 80, 20, 20, 5, 5)
229       }
230
231       State pos9 {
232         FillColor col (255,255,255)
233         Rectangle r (100, 90, 20, 20, 5, 5)
234       }
235
236       State pos10 {
237         FillColor col (255,255,255)
238         Rectangle r (100, 100, 20, 20, 5, 5)
239       }
```

```
240
241         State pos11 {
242           FillColor col (255,255,255)
243           Rectangle r (100, 110, 20, 20, 5, 5)
244         }
245
246         State pos12 {
247           FillColor col (255,255,255)
248           Rectangle r (100, 120, 20, 20, 5, 5)
249         }
250
251         State pos13 {
252           FillColor col (255,255,255)
253           Rectangle r (100, 130, 20, 20, 5, 5)
254         }
255
256
257         State pos14 {
258           FillColor col (255,255,255)
259           Rectangle r (100, 140, 20, 20, 5, 5)
260         }
261
262
263         State pos15 {
264           FillColor col (255,255,255)
265           Rectangle r (100, 150, 20, 20, 5, 5)
266         }
267
268         State pos16 {
269           FillColor col (255,255,255)
270           Rectangle r (100, 160, 20, 20, 5, 5)
271         }
272
273         State pos17 {
274           FillColor col (255,255,255)
275           Rectangle r (100, 170, 20, 20, 5, 5)
276         }
277
278         State pos18 {
279           FillColor col (255,255,255)
280           Rectangle r (100, 180, 20, 20, 5, 5)
281         }
282
283         State pos19 {
284           FillColor col (255,255,255)
285           Rectangle r (100, 190, 20, 20, 5, 5)
286         }
```

```
287
288         State pos20 {
289           FillColor col (0,255,0)
290           Rectangle r (100, 200, 20, 20, 5, 5)
291         }
292
293         State pos21 {
294           FillColor col (0,255,0)
295           Rectangle r (100, 210, 20, 20, 5, 5)
296         }
297
298         State pos22 {
299           FillColor col (0,255,0)
300           Rectangle r (100, 220, 20, 20, 5, 5)
301         }
302
303         State pos23 {
304           FillColor col (0,255,0)
305          Rectangle r (100, 230, 20, 20, 5, 5)
306         }
307
308         State pos24 {
309           FillColor col (0,255,0)
310           Rectangle r (100, 240, 20, 20, 5, 5)
311         }
312
313         State pos25 {
314           FillColor col (0,255,0)
315           Rectangle r (100, 250, 20, 20, 5, 5)
316         }
317
318         State pos26 {
319           FillColor col (0,255,0)
320           Rectangle r (100, 260, 20, 20, 5, 5)
321         }
322
323         State pos27 {
324           FillColor col (0,255,0)
325           Rectangle r (100, 270, 20, 20, 5, 5)
326         }
327
328         State pos28 {
329           FillColor col (0,255,0)
330           Rectangle r (100, 280, 20, 20, 5, 5)
331         }
332
333         State pos29 {
```

```
334         FillColor col (0,255,0)
335         Rectangle r (100, 290, 20, 20, 5, 5)
336       }
337
338       State pos30 {
339         FillColor col (255,165,0)
340         Rectangle r (100, 300, 20, 20, 5, 5)
341       }
342
343       State pos31 {
344         FillColor col (255,165,0)
345         Rectangle r (100, 310, 20, 20, 5, 5)
346       }
347
348       State pos32 {
349         FillColor col (255,165,0)
350         Rectangle r (100, 320, 20, 20, 5, 5)
351       }
352
353       State pos33 {
354         FillColor col (255,165,0)
355         Rectangle r (100, 330, 20, 20, 5, 5)
356       }
357
358       State pos34 {
359         FillColor col (255,165,0)
360         Rectangle r (100, 340, 20, 20, 5, 5)
361       }
362
363       State pos35 {
364         FillColor col (255,165,0)
365         Rectangle r (100, 350, 20, 20, 5, 5)
366       }
367
368       State pos36 {
369         FillColor col (255,165,0)
370         Rectangle r (100, 360, 20, 20, 5, 5)
371       }
372
373       State pos37 {
374         FillColor col (255,165,0)
375         Rectangle r (100, 370, 20, 20, 5, 5)
376       }
377
378       State pos38 {
379         FillColor col (255,165,0)
380         Rectangle r (100, 380, 20, 20, 5, 5)
```

```
381       }
382
383       State pos39 {
384         FillColor col (255,165,0)
385         Rectangle r (100, 390, 20, 20, 5, 5)
386       }
387
388       State pos40 {
389         FillColor col (255,165,0)
390         Rectangle r (100, 400, 20, 20, 5, 5)
391       }
392
393       State pos41 {
394         FillColor col (255,165,0)
395         Rectangle r (100, 410, 20, 20, 5, 5)
396       }
397
398       State pos42 {
399         FillColor col (255,165,0)
400         Rectangle r (100, 420, 20, 20, 5, 5)
401       }
402
403       State pos43 {
404         FillColor col (255,165,0)
405         Rectangle r (100, 430, 20, 20, 5, 5)
406       }
407
408       State pos44 {
409         FillColor col (255,165,0)
410         Rectangle r (100, 440, 20, 20, 5, 5)
411       }
412
413       State pos45 {
414         FillColor col (255,165,0)
415         Rectangle r (100, 450, 20, 20, 5, 5)
416       }
417
418       State pos46 {
419         FillColor col (255,165,0)
420         Rectangle r (100, 460, 20, 20, 5, 5)
421       }
422
423       State pos47 {
424         FillColor col (255,165,0)
425         Rectangle r (100, 470, 20, 20, 5, 5)
426       }
427
```

```
428       State pos48 {
429         FillColor col (255,165,0)
430         Rectangle r (100, 480, 20, 20, 5, 5)
431       }
432
433       State pos49 {
434         FillColor col (255,165,0)
435         Rectangle r (100, 490, 20, 20, 5, 5)
436       }
437
438       State pos50 {
439         FillColor col (255,165,0)
440         Rectangle r (100, 500, 20, 20, 5, 5)
441       }
442
443       State pos51 {
444         FillColor col (255,165,0)
445         Rectangle r (100, 510, 20, 20, 5, 5)
446       }
447
448       State pos52 {
449         FillColor col (255,165,0)
450         Rectangle r (100, 520, 20, 20, 5, 5)
451       }
452
453       pos1 -> pos2 (c.tick)
454       pos2 -> pos3 (c.tick)
455       pos3 -> pos4 (c.tick)
456       pos4 -> pos5 (c.tick)
457       pos5 -> pos6 (c.tick)
458       pos6 -> pos7 (c.tick)
459       pos7 -> pos8 (c.tick)
460       pos8 -> pos9 (c.tick)
461       pos9 -> pos10 (c.tick)
462       pos10 -> pos11 (c.tick)
463       pos11 -> pos12 (c.tick)
464       pos12 -> pos13 (c.tick)
465       pos13 -> pos14 (c.tick)
466       pos14 -> pos15 (c.tick)
467       pos15 -> pos16 (c.tick)
468       pos16 -> pos17 (c.tick)
469       pos17 -> pos18 (c.tick)
470       pos18 -> pos19 (c.tick)
471       pos19-> pos20 (c.tick)
472       pos20 -> pos21 (c.tick)
473       pos21 -> pos22 (c.tick)
474       pos22 -> pos23 (c.tick)
```

```
475       pos23 -> pos24 (c.tick)
476       pos24 -> pos25 (c.tick)
477       pos25 -> pos26 (c.tick)
478       pos26 -> pos27 (c.tick)
479       pos27 -> pos28 (c.tick)
480       pos28 -> pos29 (c.tick)
481       pos29 -> pos30 (c.tick)
482       pos30 -> pos31 (c.tick)
483       pos31 -> pos32 (c.tick)
484       pos32 -> pos33 (c.tick)
485       pos33 -> pos34 (c.tick)
486       pos34 -> pos35 (c.tick)
487       pos35 -> pos36 (c.tick)
488       pos36 -> pos37 (c.tick)
489       pos37 -> pos38 (c.tick)
490       pos38 -> pos39 (c.tick)
491       pos39 -> pos40 (c.tick)
492       pos40 -> pos41 (c.tick)
493       pos41 -> pos42 (c.tick)
494       pos42 -> pos43 (c.tick)
495       pos43 -> pos44 (c.tick)
496       pos44 -> pos45 (c.tick)
497       pos45 -> pos46 (c.tick)
498       pos46 -> pos47 (c.tick)
499       pos47 -> pos48 (c.tick)
500       pos48 -> pos49 (c.tick)
501       pos49 -> pos50 (c.tick)
502       pos50 -> pos51 (c.tick)
503       pos51 -> pos52 (c.tick)
504     }
505
506
507
508     FSM fligth2 {
509
510       State pos1 {
511         FillColor col (255,255,255)
512         Rectangle r (300, 10, 20, 20, 5, 5)
513       }
514
515       State pos2 {
516         FillColor col (255,255,255)
517         Rectangle r (300, 20, 20, 20, 5, 5)
518       }
519
520       State pos3 {
521         FillColor col (255,255,255)
```

```
522         Rectangle r (300, 30, 20, 20, 5, 5)
523       }
524
525       State pos4 {
526         FillColor col (255,255,255)
527         Rectangle r (300, 40, 20, 20, 5, 5)
528       }
529
530       State pos5 {
531         FillColor col (255,255,255)
532         Rectangle r (300, 50, 20, 20, 5, 5)
533       }
534
535       State pos6 {
536         FillColor col (255,255,255)
537         Rectangle r (300, 60, 20, 20, 5, 5)
538       }
539
540       State pos7 {
541         FillColor col (255,255,255)
542         Rectangle r (300, 70, 20, 20, 5, 5)
543       }
544
545       State pos8 {
546         FillColor col (255,255,255)
547         Rectangle r (300, 80, 20, 20, 5, 5)
548       }
549
550       State pos9 {
551         FillColor col (255,255,255)
552         Rectangle r (300, 90, 20, 20, 5, 5)
553       }
554
555       State pos10 {
556         FillColor col (255,255,255)
557         Rectangle r (300, 100, 20, 20, 5, 5)
558       }
559
560       State pos11 {
561         FillColor col (255,255,255)
562         Rectangle r (300, 110, 20, 20, 5, 5)
563       }
564
565       State pos12 {
566         FillColor col (255,255,255)
567         Rectangle r (300, 120, 20, 20, 5, 5)
568       }
```

```
569
570       State pos13 {
571         FillColor col (255,255,255)
572         Rectangle r (300, 130, 20, 20, 5, 5)
573       }
574
575       State pos14 {
576         FillColor col (255,255,255)
577         Rectangle r (300, 140, 20, 20, 5, 5)
578       }
579
580       State pos15 {
581         FillColor col (255,255,255)
582         Rectangle r (300, 150, 20, 20, 5, 5)
583       }
584
585       State pos16 {
586         FillColor col (255,255,255)
587         Rectangle r (300, 160, 20, 20, 5, 5)
588       }
589
590     State pos17 {
591       FillColor col (255,255,255)
592       Rectangle r (300, 170, 20, 20, 5, 5)
593     }
594
595     State pos18 {
596       FillColor col (255,255,255)
597       Rectangle r (300, 180, 20, 20, 5, 5)
598     }
599
600     State pos19 {
601       FillColor col (255,255,255)
602       Rectangle r (300, 190, 20, 20, 5, 5)
603     }
604
605     State pos20 {
606        FillColor col (0,255,0)
607        Rectangle r (300, 200, 20, 20, 5, 5)
608     }
609
610     State pos21 {
611       FillColor col (0,255,0)
612       Rectangle r (300, 210, 20, 20, 5, 5)
613     }
614
615     State pos22 {
```

```
616       FillColor col (0,255,0)
617       Rectangle r (300, 220, 20, 20, 5, 5)
618     }
619
620     State pos23 {
621       FillColor col (0,255,0)
622       Rectangle r (300, 230, 20, 20, 5, 5)
623     }
624
625     State pos24 {
626       FillColor col (0,255,0)
627       Rectangle r (300, 240, 20, 20, 5, 5)
628     }
629
630     State pos25 {
631       FillColor col (0,255,0)
632       Rectangle r (300, 250, 20, 20, 5, 5)
633     }
634
635     State pos26 {
636       FillColor col (0,255,0)
637       Rectangle r (300, 260, 20, 20, 5, 5)
638     }
639
640     State pos27 {
641       FillColor col (0,255,0)
642       Rectangle r (300, 270, 20, 20, 5, 5)
643     }
644
645     State pos28 {
646       FillColor col (0,255,0)
647       Rectangle r (300, 280, 20, 20, 5, 5)
648     }
649
650     State pos29 {
651       FillColor col (0,255,0)
652       Rectangle r (300, 290, 20, 20, 5, 5)
653     }
654
655     State pos30 {
656       FillColor col (255,165,0)
657       Rectangle r (300, 300, 20, 20, 5, 5)
658     }
659
660     State pos31 {
661       FillColor col (255,165,0)
662       Rectangle r (300, 310, 20, 20, 5, 5)
```

```
663     }
664
665     State pos32 {
666       FillColor col (255,165,0)
667       Rectangle r (300, 320, 20, 20, 5, 5)
668     }
669
670     State pos33 {
671       FillColor col (255,165,0)
672       Rectangle r (300, 330, 20, 20, 5, 5)
673     }
674
675     State pos34 {
676       FillColor col (255,165,0)
677       Rectangle r (300, 340, 20, 20, 5, 5)
678     }
679
680     State pos35 {
681       FillColor col (255,165,0)
682       Rectangle r (300, 350, 20, 20, 5, 5)
683     }
684
685     State pos36 {
686       FillColor col (255,165,0)
687       Rectangle r (300, 360, 20, 20, 5, 5)
688     }
689
690     State pos37 {
691       FillColor col (255,165,0)
692       Rectangle r (300, 370, 20, 20, 5, 5)
693     }
694
695     State pos38 {
696       FillColor col (255,165,0)
697       Rectangle r (300, 380, 20, 20, 5, 5)
698     }
699
700     State pos39 {
701       FillColor col (255,165,0)
702       Rectangle r (300, 390, 20, 20, 5, 5)
703     }
704
705     State pos40 {
706       FillColor col (255,165,0)
707       Rectangle r (300, 400, 20, 20, 5, 5)
708     }
709
```

```
710     State pos41 {
711       FillColor col (255,165,0)
712       Rectangle r (300, 410, 20, 20, 5, 5)
713     }
714
715     State pos42 {
716       FillColor col (255,165,0)
717       Rectangle r (300, 420, 20, 20, 5, 5)
718     }
719
720     State pos43 {
721       FillColor col (255,165,0)
722       Rectangle r (300, 430, 20, 20, 5, 5)
723     }
724
725     State pos44 {
726       FillColor col (255,165,0)
727       Rectangle r (300, 440, 20, 20, 5, 5)
728     }
729
730     State pos45 {
731       FillColor col (255,0,0)
732       Rectangle r (300, 450, 20, 20, 5, 5)
733     }
734
735     State pos46 {
736       FillColor col (255,0,0)
737       Rectangle r (300, 460, 20, 20, 5, 5)
738     }
739
740     State pos47 {
741       FillColor col (255,0,0)
742       Rectangle r (300, 470, 20, 20, 5, 5)
743     }
744
745     State pos48 {
746       FillColor col (255,0,0)
747       Rectangle r (300, 480, 20, 20, 5, 5)
748     }
749
750     State pos49 {
751       FillColor col (255,0,0)
752       Rectangle r (300, 490, 20, 20, 5, 5)
753     }
754
755     State pos50 {
756       FillColor col (255,0,0)
```

```
757       Rectangle r (300, 500, 20, 20, 5, 5)
758     }
759
760     State pos51 {
761       FillColor col (255,0,0)
762       Rectangle r (300, 510, 20, 20, 5, 5)
763     }
764
765     State pos52 {
766       FillColor col (255,0,0)
767       Rectangle r (300, 520, 20, 20, 5, 5)
768     }
769
770     pos1 -> pos2 (c.tick)
771     pos2 -> pos3 (c.tick)
772     pos3 -> pos4 (c.tick)
773     pos4 -> pos5 (c.tick)
774     pos5 -> pos6 (c.tick)
775     pos6 -> pos7 (c.tick)
776     pos7 -> pos8 (c.tick)
777     pos8 -> pos9 (c.tick)
778     pos9 -> pos10 (c.tick)
779     pos10 -> pos11 (c.tick)
780     pos11 -> pos12 (c.tick)
781     pos12 -> pos13 (c.tick)
782     pos13 -> pos14 (c.tick)
783     pos14 -> pos15 (c.tick)
784     pos15 -> pos16 (c.tick)
785     pos16 -> pos17 (c.tick)
786     pos17 -> pos18 (c.tick)
787     pos18 -> pos19 (c.tick)
788     pos19-> pos20 (c.tick)
789     pos20 -> pos21 (c.tick)
790     pos21 -> pos22 (c.tick)
791     pos22 -> pos23 (c.tick)
792     pos23 -> pos24 (c.tick)
793     pos24 -> pos25 (c.tick)
794     pos25 -> pos26 (c.tick)
795     pos26 -> pos27 (c.tick)
796     pos27 -> pos28 (c.tick)
797     pos28 -> pos29 (c.tick)
798     pos29 -> pos30 (c.tick)
799     pos30 -> pos31 (c.tick)
800     pos31 -> pos32 (c.tick)
801     pos32 -> pos33 (c.tick)
802     pos33 -> pos34 (c.tick)
803     pos34 -> pos35 (c.tick)
```

```
804     pos35 -> pos36 (c.tick)
805     pos36 -> pos37 (c.tick)
806     pos37 -> pos38 (c.tick)
807     pos38 -> pos39 (c.tick)
808     pos39 -> pos40 (c.tick)
809     pos40 -> pos41 (c.tick)
810     pos41 -> pos42 (c.tick)
811     pos42 -> pos43 (c.tick)
812     pos43 -> pos44 (c.tick)
813     pos44 -> pos45 (c.tick)
814     pos45 -> pos46 (c.tick)
815     pos46 -> pos47 (c.tick)
816     pos47 -> pos48 (c.tick)
817     pos48 -> pos49 (c.tick)
818     pos49 -> pos50 (c.tick)
819     pos50 -> pos51 (c.tick)
820     pos51 -> pos52 (c.tick)
821
822     }
823 }
```

Bibliographie

[1] Dave Aitken. *Introducing Verified React*. Anglais. Jan. 2019. URL : https://medium.com/imandra/introducing-verified-react-9c2ef03f821b.

[2] Blair Archibald, Muffy Calder et Michele Sevegnani. « Conditional bigraphs ». In : *Graph Transformation : 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25–26, 2020, Proceedings 13*. Springer. 2020, p. 3-19.

[3] Blair Archibald et al. « Modelling and verifying BDI agents with bigraphs ». en. In : *Science of Computer Programming* 215 (mars 2022), p. 102760. ISSN : 01676423. DOI : 10.1016/j.scico.2021.102760. URL : https://linkinghub.elsevier.com/retrieve/pii/S0167642321001532 (visité le 04/12/2022).

[4] Eric Barboni et al. « Bridging the gap between a behavioural formal description technique and a user interface description language : Enhancing ICO with a graphical user interface markup language ». en. In : *Science of Computer Programming* 86 (juin 2014), p. 3-29. ISSN : 01676423. DOI : 10.1016/j.scico.2013.04.001. URL : https://linkinghub.elsevier.com/retrieve/pii/S0167642313000993 (visité le 17/06/2022).

[5] Bruno Barras et al. « The Coq proof assistant reference manual : Version 6.1 ». PhD Thesis. Inria, 1997.

[6] Pascal Béger. *Vérification formelle des propriétés graphiques des systèmes informatiques interactifs*. en, p. 195.

[7] Gil Benkö, Christoph Flamm et Peter F. Stadler. « Generic Properties of Chemical Networks : Artificial Chemistry Based on Graph Rewriting ». In : *Advances in Artificial Life*. Sous la dir. de Wolfgang Banzhaf et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 10-19. ISBN : 978-3-540-39432-7.

[8] Silvia Berti et al. « The TERESA XML language for the description of interactive systems at multiple abstraction levels ». In : *Proceedings workshop on developing user interfaces with XML : advances on user interface description languages*. 2004, p. 103-110.

[9] J. Bertin et M. Barbut. *Sémiologie graphique : les diagrammes, les réseaux, les cartes*. Mouton, 1973. ISBN : 978-3-11-118976-5. URL : https://books.google.fr/books?id=F4weAAAAMAAJ.

[10] Timothy Bourke, Lélio Brun et Marc Pouzet. « Towards a verified Lustre compiler with modular reset ». en. In : *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. Sankt Goar Germany : ACM, mai 2018, p. 14-17. ISBN : 978-1-4503-5780-7. DOI : 10.1145/3207719.3207732. URL : https://dl.acm.org/doi/10.1145/3207719.3207732 (visité le 20/10/2022).

[11] Luca Cardelli et Andrew D. Gordon. « Mobile Ambients ». In : *Foundations of Software Science and Computation Structure*. 1997.

[12] A Carl. « Petri. kommunikation mit automaten ». In : *PhD, University of Bonn, West Germany* (1962).

[13] Gail Chappell et Nancy Hildebrandt. *Using FXML to create a User Interface*. English. Sept. 2013. URL : https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm.

[14] Stéphane Chatty. « Supporting Multidisciplinary Software Composition for Interactive Applications ». en. In : *Software Composition*. Sous la dir. de David Hutchison et al. T. 4954. Series Title : Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 173-189. ISBN : 978-3-540-78788-4 978-3-540-78789-1. DOI : 10.1007/978-3-540-78789-1_14. URL : http://link.springer.com/10.1007/978-3-540-78789-1_14 (visité le 02/11/2022).

[15] Stéphane Chatty, Mathieu Magnaudet et Daniel Prun. « Verification of properties of interactive components from their executable code ». en. In : *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Duisburg Germany : ACM, juin 2015, p. 276-285. ISBN : 978-1-4503-3646-8. DOI : 10.1145/2774225.2774848. URL : https://dl.acm.org/doi/10.1145/2774225.2774848 (visité le 30/06/2022).

[16] Qt Company. *Signals & Slots*. Anglais. 2022. URL : https://doc.qt.io/qt-6/signalsandslots.html.

[17] Vincent Danos et al. « Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models ». en. In : (2012). Artwork Size : 13 pages Medium : application/pdf Publisher : Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 13 pages. DOI : 10.4230/LIPICS.FSTTCS.2012.276. URL : http://drops.dagstuhl.de/opus/volltexte/2012/3866/ (visité le 04/12/2022).

[18] Cédric Fournet et Georges Gonthier. « The reflexive CHAM and the join-calculus ». In : *ACM-SIGACT Symposium on Principles of Programming Languages*. 1996.

[19] Josefina Guerrero-Garcia et al. « A Theoretical Survey of User Interface Description Languages : Preliminary Results ». en. In : *2009 Latin American Web Congress*. Merida, Yucatan, Mexico : IEEE, nov. 2009, p. 36-43. ISBN : 978-0-7695-3856-3. DOI : 10.1109/LA-WEB.2009.40. URL : http://ieeexplore.ieee.org/document/5341626/ (visité le 17/06/2022).

[20] Cécile Hardebolle et Frédéric Boulanger. « Exploring Multi-Paradigm Modeling Techniques ». en. In : *SIMULATION* 85.11-12 (nov. 2009), p. 688-708. ISSN : 0037-5497, 1741-3133. DOI : 10.1177/0037549709105240. URL : http://journals.sagepub.com/doi/10.1177/0037549709105240 (visité le 12/11/2022).

[21] P. Humphreys. *Extending Ourselves : Computational Science, Empiricism, and Scientific Method*. Oxford University Press, 2004. ISBN : 978-0-19-803600-5. URL : https://books.google.fr/books?id=ZIot7QGz7eEC.

[22] Kurt Jensen et Grzegorz Rozenberg. *High-level Petri Nets : Theory and Application*. Springer Berlin, Heidelberg. 1991.

[23] Jean Krivine. *From Molecules to Systems : the problem of knowledge representation in molecular biology*. Nov. 2019. URL : https://www.college-de-france.fr/agenda/seminaire/la-biologie-de-information-un-dialogue-entre-informatique-et-la-biologie/from-molecules-to-systems-the-problem-of-knowledge-representation-in-molecular-biology.

[24] Xavier Leroy. « Formal certification of a compiler back-end or : programming a compiler with a proof assistant ». In : *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2006, p. 42-54.

[25] N.G. Leveson et C.S. Turner. « An investigation of the Therac-25 accidents ». en. In : *Computer* 26.7 (juill. 1993), p. 18-41. issn : 0018-9162. doi : 10.1109/MC.1993.274940. url : http://ieeexplore.ieee.org/document/274940/ (visité le 21/11/2022).

[26] Michael Löwe. « Algebraic approach to single-pushout graph transformation ». In : *Theoretical Computer Science* 109.1 (1993), p. 181-224. issn : 0304-3975. doi : https://doi.org/10.1016/0304-3975(93)90068-5. url : https://www.sciencedirect.com/science/article/pii/0304397593900685.

[27] Mathieu Magnaudet et al. « Djnn/Smala : A Conceptual Framework and a Language for Interaction-Oriented Programming ». en. In : *Proceedings of the ACM on Human-Computer Interaction* 2.EICS (juin 2018), p. 1-27. issn : 2573-0142. doi : 10.1145/3229094. url : https://dl.acm.org/doi/10.1145/3229094 (visité le 29/06/2022).

[28] Ingo Maier, Tiark Rompf et Martin Odersky. *Deprecating the observer pattern*. Rapp. tech. 2010.

[29] Cécile Marcon et al. « Représentation de programmes SMALA grâce à la théorie des bigraphes ». In : *AFADL*. 2021.

[30] R. Milner. *Communicating and Mobile Systems : The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[31] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press. USA, 2009.

[32] Brad A. Myers. « Separating application code from toolkits : eliminating the spaghetti of callbacks ». en. In : *Proceedings of the 4th annual ACM symposium on User interface software and technology - UIST '91*. Hilton Head, South Carolina, United States : ACM Press, 1991, p. 211-220. isbn : 978-0-89791-451-2. doi : 10.1145/120782.120805. url : http://portal.acm.org/citation.cfm?doid=120782.120805 (visité le 29/06/2022).

[33] Manfred Nagl. « Graph Rewriting Systems and their Application in Biology ». In : *Mathematical Models in Medicine*. Sous la dir. de Jürgen Berger et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 1976, p. 135-156. isbn : 978-3-642-93048-5.

[34] Nicolas Nalpon, Cyril Allignol et Célia Picard. « Towards a User Interface Description Language Based on Bigraphs ». In : *Theoretical Aspects of Computing – ICTAC 2022* (2022). Publisher : LNCS. url : https://hal.archives-ouvertes.fr/hal-03790499.

[35] Nicolas Nalpon et Alice Martin. « Why semantics are not only about expressiveness : The reactive programming case ». English. In : *Program and short abstracts, HaPoP 2022* (juin 2022). url : https://www.shift-society.org/hapop5/boa.pdf.

[36] Nicolas Nalpon et al. « Vers la vérification de SMALA, un langage réactif interactif ». In : *AFADL 2020-19èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels*. 2020.

[37] David Navarre et al. « ICOs : A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability ». en. In : *ACM Transactions on Computer-Human Interaction* 16.4 (nov. 2009), p. 1-56. issn : 1073-0516, 1557-7325. doi : 10.1145/1614390.1614393. url : https://dl.acm.org/doi/10.1145/1614390.1614393 (visité le 24/06/2022).

[38] Tobias Nipkow, Markus Wenzel et Lawrence C Paulson. *Isabelle/HOL : a proof assistant for higher-order logic*. Springer, 2002.

[39] Fabio Paterno', Carmen Santoro et Lucio Davide Spano. « MARIA : A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments ». en. In : *ACM Transactions on Computer-Human Interaction* 16.4 (nov. 2009), p. 1-30. ISSN : 1073-0516, 1557-7325. DOI : 10.1145/1614390.1614394. URL : https://dl.acm.org/doi/10.1145/1614390.1614394 (visité le 20/10/2022).

[40] Peter Bambulis et al. *Validating Properties of Component-based Graphical User Interfaces*. English. 1996.

[41] Daniel Prun et Pascal Béger. « Formal Verification of Graphical Properties of Interactive Systems ». en. In : *Proceedings of the ACM on Human-Computer Interaction* 6.EICS (juin 2022), p. 1-30. ISSN : 2573-0142. DOI : 10.1145/3534521. URL : https://dl.acm.org/doi/10.1145/3534521 (visité le 09/12/2022).

[42] *RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A*. English. Déc. 2011.

[43] Sebastian Sardina, Lavindra De Silva et Lin Padgham. « Hierarchical planning in BDI agent programming languages : A formal approach ». In : *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. 2006, p. 1001-1008.

[44] Michele Sevegnani et Muffy Calder. « BigraphER : Rewriting and Analysis Engine for Bigraphs ». en. In : *Computer Aided Verification*. Sous la dir. de Swarat Chaudhuri et Azadeh Farzan. T. 9780. Series Title : Lecture Notes in Computer Science. Cham : Springer International Publishing, 2016, p. 494-501. ISBN : 978-3-319-41539-0 978-3-319-41540-6. DOI : 10.1007/978-3-319-41540-6_27. URL : http://link.springer.com/10.1007/978-3-319-41540-6_27 (visité le 13/10/2022).

[45] Peter Sewell. « Global/local subtyping and capability inference for a distributed π-calculus ». In : *Automata, Languages and Programming*. Sous la dir. de Kim G. Larsen, Sven Skyum et Glynn Winskel. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998, p. 695-706. ISBN : 978-3-540-68681-1.

[46] Carlos Eduardo Silva et José Creissac Campos. « Can GUI Implementation Markup Languages Be Used for Modelling ? » en. In : *Human-Centered Software Engineering*. Sous la dir. de David Hutchison et al. T. 7623. Series Title : Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 112-129. ISBN : 978-3-642-34346-9 978-3-642-34347-6. DOI : 10.1007/978-3-642-34347-6_7. URL : http://link.springer.com/10.1007/978-3-642-34347-6_7 (visité le 30/06/2022).

[47] Yong Kiam Tan et al. « A new verified compiler backend for CakeML ». In : *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, p. 60-73.

[48] Jean Vanderdonckt et al. « UsiXML : a user interface description language for specifying multimodal user interfaces ». In : *Proceedings of W3C Workshop on Multimodal Interaction WMI*. T. 2004. sn. 2004.

[49] David Waszek. « Informational Equivalence but Computational Differences ? Herbert Simon on Representations in Scientific Practice ». In : *Minds and Machines* (2023), p. 1-24.

[50] Benjamin Weyers et al., éd. *The Handbook of Formal Methods in Human-Computer Interaction*. en. Human–Computer Interaction Series. Cham : Springer International Publishing, 2017. ISBN : 978-3-319-51837-4 978-3-319-51838-1. DOI : 10.1007/978-3-319-51838-1. URL : http://link.springer.com/10.1007/978-3-319-51838-1 (visité le 22/11/2022).