Rapport Technique CEDRIC 829

Vous avez dit raffinement ?

Frédéric Gervais[1,2], Marc Frappier[2], Régine Laleau[3]

[1] CEDRIC, CNAM-IIE,
18 Allée Jean Rostand, 91025 Évry Cedex, France
frederic.gervais@usherbrooke.ca

[2] GRIL, Université de Sherbrooke,
2500, Boulevard de l'Université
Sherbrooke (Québec) J1K 2R1, Canada
marc.frappier@usherbrooke.ca

[3] LACL, Université Paris 12,
IUT Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
laleau@univ-paris12.fr

15 Mars 2005

Résumé Le raffinement constitue une notion très importante des méthodes de spécification formelles. On peut distinguer principalement deux formes de dérivations : le raffinement algorithmique et le raffinement de détails. D'un côté, le raffinement algorithmique permet de passer d'une formulation abstraite des spécifications à une description de plus en plus proche du code. Le raffinement de détails sert d'un autre côté à rajouter des détails comme de nouveaux événements ou de nouvelles variables d'état. Dans les deux cas, les transformations apportées par le raffinement préservent la correction des nouveaux programmes vis-à-vis de la spécification initiale. L'objet de cet article est de présenter un état de l'art sur le raffinement et sur ses modalités de vérification dans les méthodes formelles. Dans la plupart des cas, le raffinement d'un programme ou d'une spécification est prouvé à l'aide d'une relation de simulation. Ces moyens ne sont pas tous équivalents. Notre motivation est que l'activité de raffinement doit constituer un critère de sélection d'une méthode au même titre que l'expressivité du langage ou la disponibilité des outils.

Mots-clé Raffinement, relation de simulation, correction, preuve

Table des matières

1 Introduction

Nos intérêts portent sur la spécification formelle de systèmes d'information. Le langage EB^3 [FSD03] est un langage de spécification formel inspiré des langages à algèbres de processus comme CSP [Hoa85] et LOTOS [BB87], mais adapté au cadre des systèmes d'information. Afin d'améliorer la méthode de conception, il nous est apparu nécessaire de doter le langage EB^3 d'une relation de raffinement pour spécifier des systèmes complexes tout en restant formel. En analysant les méthodes de raffinement actuelles, il ne nous semblait pas clair quel était le lien entre ces différentes notions. L'état de l'art présenté dans la suite est l'aboutissement de ce constat.

Le raffinement est une approche de plus en plus répandue pour construire progressivement des programmes corrects : il consiste à dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du programme préserve bien sa correction vis-à-vis de la spécification précédente.

Le raffinement constitue donc une activité importante des méthodes de spécification formelles. Il existe selon les langages utilisés de nombreuses notions de raffinement qui ne sont pas toujours équivalentes. Si le raffinement

est invariablement une préservation de la correction, celui-ci s'exprime et se vérifie de différentes manières.

La méthode la plus courante est la recherche d'une relation de simulation de la spécification concrète par sa spécification abstraite. Les approches étudiées proposent la vérification de conditions suffisantes sur la relation afin d'assurer la correction du raffinement. Selon les sémantiques considérées, les conditions ne s'expriment pas de la même façon.

Notre souhait est de présenter les principales formes de raffinement et de les comparer selon leurs effets sur les spécifications à raffiner. Dans un premier temps (Sect. 2), nous introduirons le raffinement en présentant la notion de contrat définie par Back [BvW98] et en la comparant avec la définition du raffinement dans la sémantique relationnelle. Puis nous présenterons le raffinement dans plusieurs sémantiques (Sect. 3) et ses applications dans quelques méthodes formelles (Sect. 4). Trois formes principales de raffinement sont considérées dans cet état de l'art : le raffinement de séquences d'opérations (comme dans les machines B [Abr96] ou les types de données abstraits Z [Spi92]), le raffinement au niveau des transitions d'états (comme en B événementiel [Abr00] ou en CSP [Jos88]) et le raffinement des traces, échecs et divergences (comme en CSP [Ros97]). Nous conclurons dans la section 5 par une analyse et comparaison des approches étudiées.

2 Raffinement : préservation de la correction

La notion de raffinement a été introduite dans les années 1970 par Dijkstra [Dij76], puis formalisée par Back [Bac78, Bac88] dans les années 1980. Plusieurs travaux ont ensuite développé cette notion, en particulier Abadi et Lamport [AL88], Back [BvW98], Morgan [Mor90], Morris [Mor87] et Abrial [Abr96]. Le raffinement est un moyen de construire de manière progressive des programmes corrects.

2.1 Correction des programmes

L'intérêt des méthodes formelles est la possibilité de vérifier de manière rigoureuse des propriétés sur un modèle formel. Pour assurer qu'un programme respecte certaines propriétés, on lui associe une spécification formelle sur laquelle il est possible de raisonner mathématiquement.

Définition et notation. Dans la logique de Hoare, un programme S est associé à une précondition P et une postcondition Q. La précondition P permet de caractériser les états possibles avant l'exécution de S, tandis que la postcondition Q est vérifiée par les états possibles après l'exécution de S.

Un programme S est *totalement correct* vis-à-vis de sa spécification P, Q si, à partir de tout état initial vérifiant la précondition P, le programme termine et fait passer le système à un état satisfaisant la postcondition Q. Si la

terminaison n'est pas assurée, la correction est dite *partielle*. Un programme S totalement correct par rapport à sa spécification P, Q est dénoté dans la suite par $P < S > Q$.

Notion de contrat. Back [BvW98] introduit la notion de *contrat* pour interpréter la correction des programmes. Un contrat est une généralisation des programmes et des spécifications. Il est défini par un langage sur les instructions *inst*. L'instruction la plus simple est une affectaction de valeur. Les instructions sont composées entre elles par des séquences et des choix. Toute instruction peut être associée à une assertion *cond* et à une hypothèse H. Un cas typique de contrat C est de la forme :

$$[H_1]inst_1^j\{cond_1\}\ ;\ ...\ ;\ [H_i]inst_i^j\{cond_i\}\ ;\ ...\ ;\ [H_n]inst_n^j\{cond_n\}$$

Les indices j représentent les entités concernées par l'instruction. Les entités agissent sur le système en exécutant les instructions qui leur sont associées dans le contrat. Une entité peut être un utilisateur, le système ou bien l'environnement : ce sont les *acteurs* du contrat.

Le contrat peut être vu comme une règle du jeu pour les acteurs. Comme le choix est un opérateur sur les instructions, la règle du jeu encadre et limite les options de chaque acteur. Une instruction *inst* n'est exécutable que si l'assertion *cond* qui lui est associée est vérifiée. Les choix d'un acteur peuvent donc conduire un autre acteur à se retrouver bloqué si toutes les assertions deviennent fausses. Les assertions correspondent à la notion de garde. Les hypothèses H représentent les assomptions du contrat. On ne s'intéresse pas dans le contrat à ce qui se passe si les hypothèses ne sont pas vérifiées : cela correspond à la notion de précondition. La correction et le raffinement sont interprétés dans cette sémantique des jeux.

Un ange contre un démon. Les acteurs considérés sont un ange et un démon. L'état initial est σ. Le but est d'arriver à l'état q à partir de σ en respectant le contrat C. La correction est associée au point de vue de l'ange.

Les choix du démon risquent de bloquer l'ange et le contrat ne sera alors pas respecté. Si l'ange parvient à l'état q tout en respectant le contrat, alors le démon aura perdu. La difficulté consiste donc à trouver une stratégie gagnante pour l'ange quels que soient les choix du démon.

Pour respecter son contrat, l'ange doit faire ses choix de manière à :

- faire échouer le démon quelles que soient ses options : dans ce cas, le démon se retrouvera bloqué et l'ange aura gagné,
- ou bien sortir des termes du contrat : si les hypothèses ne sont plus vraies, le contrat n'a plus besoin d'être respecté et dans ce cas, l'ange aura gagné.

La correction d'un programme correspond donc à l'existence d'une stratégie gagnante de l'ange pour faire passer le système de l'état σ à l'état q.

Pour préserver la correction, le raffinement correspond à une augmentation des chances de réussite de l'ange et une diminution de celles du démon. Le raffinement se traduit donc par une augmentation des stratégies gagnantes de l'ange et par une diminution des stratégies gagnantes du démon.

L'idée d'un jeu entre un ange et un démon a initialement été proposé par Hintikka [Hin72] puis développé notamment par Moschovakis [Mos72], Aczel [Acz75] et Back [BvW89].

2.2 Sémantique relationnelle des programmes

La notion de contrat est plus générale qu'un programme défini comme une séquence d'opérations. Une sémantique courante pour représenter les programmes consiste à utiliser le *calcul relationnel* [Tar41]. Elle fut notamment décrite par de Bakker en 1980 [dB80] et Mili en 1983 [Mil83]. Les langages VDM [Jon90] et Z [Spi92], par exemple, utilisent cette vue relationnelle.

Représentation des programmes. Un programme est alors défini par trois sortes de relations :

- une initialisation,
- une séquence d'opérations,
- une finalisation.

Les relations gg représentant les programmes sont définies sur un monde global G :

$$gg : G \leftrightarrow G$$

Un programme S défini sur le monde T est une composition relationnelle de la forme :

$$S = ti \; ; \; top_1 \; ; \; ... \; ; \; top_n \; ; \; tf$$

avec une relation d'initialisation $ti : G \leftrightarrow T$, des opérations $top_j : T \leftrightarrow T$ avec $j \in \{1, .., n\}$ et une relation de finalisation $tf : T \leftrightarrow G$.

Choix angélique et démoniaque. Dans le cadre de cette sémantique, il est possible d'interpréter le choix de deux manières distinctes : soit on considère que les choix dans le programme dépendent de l'ange et dans ce cas le choix est dit *angélique* ; soit le choix considéré est *démoniaque*, dans le cas contraire.

Dans le premier cas, un programme S arrive à la postcondition Q à partir d'un état initial σ (noté $\sigma\{|S|\}Q$) s'il existe un état σ' tel que :

- l'exécution de S permet de passer de l'état σ à l'état σ' (ce qui est noté : $\sigma S \sigma'$),
- et l'état σ' satisfait la postcondition Q (noté $Q.\sigma'$).

Il existe donc une stratégie gagnante pour l'ange, et on suppose qu'il choisira toujours cette exécution afin de satisfaire la spécification.

Dans le cas du choix démoniaque, $\sigma\{|S|\}Q$ est vraie si pour tout état σ' satisfaisant $\sigma S\sigma'$, on a : $Q.\sigma'$. Le démon gagne donc quels que soient ses choix.

La sémantique de la correction et du raffinement dépend donc de l'interprétation du choix. S est correct vis-à-vis de la précondition P et de la postcondition Q (noté $P < S > Q$) si pour tout état σ satisfaisant P,

$$\sigma\{|S|\}Q$$

Si le choix est interprété de manière angélique, alors :

$$\forall\sigma : P.\sigma, (\exists\sigma' : \sigma S\sigma' \wedge Q.\sigma')$$

Cette définition correspond à la correction totale : l'existence de σ' assure la terminaison du programme S. Si le choix est démoniaque, alors :

$$\forall\sigma : P.\sigma, (\forall\sigma' : \sigma S\sigma' \Rightarrow Q.\sigma')$$

La correction n'est alors que partielle, car l'existence d'un état final n'est pas assuré.

Le raffinement dépend également de l'interprétation du choix. Si elle est angélique, alors l'ange augmente ses chances de gagner. En termes de relation, cela se traduit par le fait que $S \subseteq S'$ si S est raffiné par S'. Si le choix est démoniaque, alors $S' \subseteq S$ quand S est raffiné par S'.

3 Raffinement : un problème de sémantique

L'expression $P < S > Q$ contient trois paramètres : S, P et Q. Le problème est différent selon que S est connu ou pas. La correction de programmes suppose que le programme S existe déjà. Si S n'est pas connu, le problème consiste à dériver progressivement un programme correct simple. On fixe $P < S > Q$ et on cherche S' tel que :

$$\forall P, Q, (P < S > Q \Rightarrow P < S' > Q)$$

Autrement dit, toute spécification satisfaite par S est aussi satisfaite par S'. On peut donc remplacer S par S', tout en continuant à satisfaire la spécification initiale. S' est un raffinement de S, ce qui est dénoté par :

$$S \sqsubseteq S'$$

La relation $\sqsubseteq$ doit vérifier trois propriétés importantes : elle est réflexive, transitive et monotone. Un programme est ainsi le raffinement de lui-même par réflexivité. La propriété de transitivité permet de raffiner un programme

par étapes successives. Enfin, la monotonie de la relation sert à raffiner les parties d'un programme séparément. La dérivation est ainsi réalisée de manière progressive.

Il est toutefois difficile d'analyser et de comparer des expressions de la forme $P < S > Q$. Plusieurs sémantiques ont permis de fixer quelques paramètres afin de simplifier les vérifications. La correction et le raffinement dépendent en effet de la sémantique utilisée pour interpréter $P < S > Q$.

3.1 Sémantique des transformateurs de prédicats

Pour vérifier la correction des programmes, Dijkstra [Dij76] a introduit la notion de "plus faible précondition", notée wp, qui s'appuie sur la logique de Hoare. L'opérateur wp permet de calculer la plus faible précondition qui garantisse la terminaison d'un programme S et qui laisse le système dans un état qui satisfait la postcondition Q. Pour un programme S et une postcondition Q, $wp(S, Q)$ est la plus faible précondition P telle que $P < S > Q$.

La correction et le raffinement sont maintenant exprimés dans cette sémantique. Un programme S de précondition P et de postcondition Q est alors correct si :

$$P \Rightarrow wp(S, Q)$$

Dans la sémantique wp, le raffinement se traduit par :

$$S \sqsubseteq S' \Leftrightarrow (\forall Q, wp(S, Q) \Rightarrow wp(S', Q))$$

À partir d'un programme S et d'une postcondition Q, l'opérateur wp permet de revenir sur les préconditions possibles. La sémantique wp est dite *backward*.

De nombreux raffinements dans les méthodes formelles sont basés sur la sémantique wp, en particulier les raffinements de Z [DW96] et B [Abr96]. Les raffinements dans les Actions Systems [BvW98] et dans CSP [Hoa85] ont été reliés au raffinement des wp.

3.2 Sémantique des jeux

La notion de contrat présentée dans la section 2.1 est associée à des jeux, autrement dit les choix de chaque joueur, dans la sémantique du jeu. On note $gm(C)$ l'ensemble des jeux du contrat C.

Les notions de correction et de raffinement de la sémantique wp sont interprétées par la sémantique des jeux de la manière suivante : un programme S est correct s'il existe une stratégie gagnante pour l'ange. On note $ws(J, Q)$ le prédicat qui indique si le jeu J est une stratégie gagnante sous le contrat C pour atteindre un état qui satisfait la postcondition Q.

Back définit dans [BvW89] une sémantique opérationnelle des jeux avec des règles d'inférence.

3.3 Sémantique du choix

La sémantique *wp* permet de passer des postconditions aux préconditions. Elle est dite *backward*. Une sémantique *forward* est la sémantique du choix. Elle permet de considérer les ensembles de postconditions du programme S en fonction d'une précondition P. Ces ensembles sont dénotés par $ch(S, P)$. Un programme S de précondition P et de postcondition Q est correct si :

$$Q \in ch(S, P)$$

Intuitivement, on associe à tout état initial un ensemble de postconditions possibles pour l'état final. Une postcondition peut être vue comme un prédicat sur les états ou bien comme un ensemble d'états. Les ensembles d'ensembles $ch(S, P)$ tels que définis par Back [BvW98] sont fermés par le haut. En particulier, pour tous prédicats Q, Q',

$$(Q \in ch(S, P) \wedge Q \subseteq Q') \Rightarrow Q' \in ch(S, P)$$

Le retrait d'éléments dans un ensemble d'états revient à ajouter de nouveaux ensembles dans l'ensemble d'ensembles. En termes de postcondition, le rajout d'un choix possible de postconditions revient à renforcer les postconditions dans ces choix. Le raffinement se traduit dans cette sémantique par :

$$S \sqsubseteq S' \Leftrightarrow (\forall P, ch(S, P) \subseteq ch(S', P))$$

Ces trois premières sémantiques, ainsi que la notion de contrat, sont détaillées dans le livre de Back [BvW98]. La figure 1 représente les liens entre les différentes sémantiques.

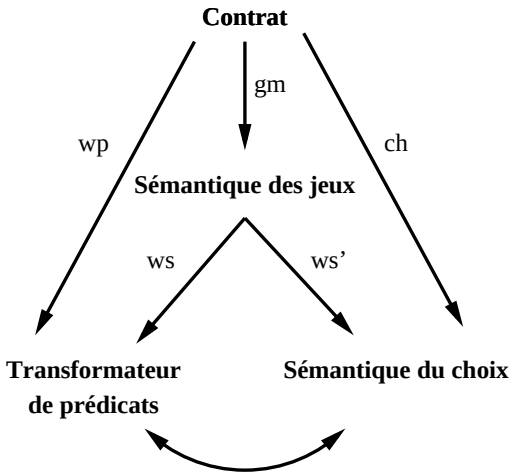

FIG. 1 – Liens entre les sémantiques wp, des jeux et du choix

En particulier, le lien entre la sémantique des jeux et les transformateurs de prédicats wp est l'existence de stratégies gagnantes pour l'ange :

$$wp(S, Q) = ws(gm(C), Q)$$

De même, il est possible de définir un opérateur ws' de manière à relier la sémantique des jeux à la sémantique du choix :

$$ch(S, P) = ws'(gm(C), P)$$

Enfin, pour tous P, Q,

$$(P \Rightarrow wp(S, Q)) \Leftrightarrow (Q \in ch(S, P))$$

3.4 Sémantique relationnelle

Dans la sémantique relationnelle (voir section 2.2), le raffinement s'exprime au niveau des types de données. Un type de donnée $\mathcal{P}$ est défini comme la donnée de $(P, pi, pf, \{pop_j\})$. Un programme sur $\mathcal{P}$ est défini comme une séquence de la forme :

$$pi\ ;\ pop_1\ ;\ ...\ ;\ pop_n\ ;\ pf \subseteq gg_{\mathcal{P}}$$

où $gg_{\mathcal{P}}$ désigne l'ensemble des programmes sur le type de donnée $\mathcal{P}$.

Un type de donnée $\mathcal{A}$ est raffiné par un type de donnée $\mathcal{C}$ si l'ensemble des programmes possibles de $\mathcal{C}$ est inclus dans l'ensemble de tous les programmes possibles de $\mathcal{A}$. Autrement dit :

$$\mathcal{A} \sqsubseteq \mathcal{C} \Leftrightarrow gg_{\mathcal{C}} \subseteq gg_{\mathcal{A}}$$

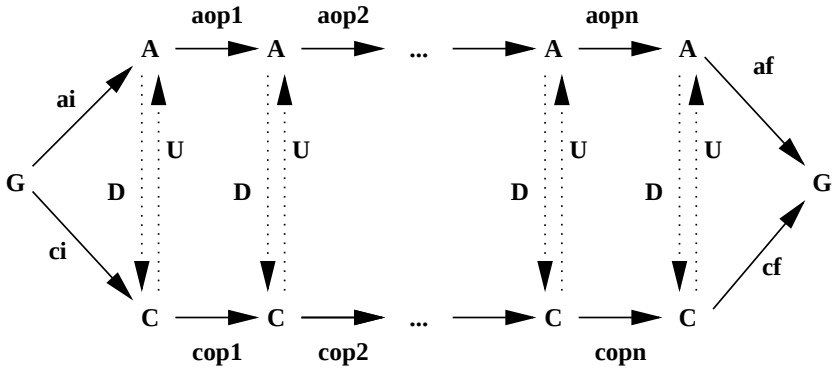

FIG. 2 – Relations de simulation

Afin de prouver ce raffinement sans raisonner sur l'espace de tous les programmes, He et al. montrent que deux ensembles de règles sont suffisants

pour prouver le raffinement [HHS86]. La figure 2 représente les relations d'un programme à travers le monde abstrait $\mathcal{A}$ et à travers le monde concret $\mathcal{C}$. Les règles de He et al. s'appliquent si les séquences d'opérations dans $\mathcal{A}$ et dans $\mathcal{C}$ sont les mêmes, dans le sens où on définit une bijection entre les opérations de $\mathcal{A}$ et les opérations de $\mathcal{C}$. Une relation de simulation permet de relier les types de données $\mathcal{A}$ et $\mathcal{C}$.

Une relation de simulation *forward* ou *downward* (voir sens de la relation sur la figure 2) est une relation $D : A \leftrightarrow C$ telle que :

$$\begin{aligned} ci &\subseteq ai \,;\, D \\ D \,;\, cf &\subseteq af \\ D \,;\, copi &\subseteq aopi \,;\, D \end{aligned}$$

Une relation de simulation *backward* ou *upward* est une relation $U : C \leftrightarrow A$ telle que :

$$\begin{aligned} ci \,;\, U &\subseteq ai \\ cf &\subseteq U \,;\, af \\ copi \,;\, U &\subseteq U \,;\, aopi \end{aligned}$$

Les conditions sur les relations de simulation sont appelées respectivement les règles forward et backward. Les règles de He et al. sont toutefois limitées, car elles imposent la totalité des relations. Plusieurs travaux, comme [BDW99], ont permis ensuite de considérer des cas plus généraux, avec des relations non totales. Le livre de de Roever [dRE98] fait le point sur les différentes techniques de simulation d'un point de vue théorique.

3.5 Sémantique opérationnelle : LTS

Un système de transition étiqueté (ou *labelled transition system* en anglais : LTS) est défini de la manière suivante. Un LTS est la donnée de :

- un ensemble d'états E,
- un ensemble d'états initiaux I, avec $I \subseteq E$,
- un ensemble d'étiquettes de transitions (ou actions) L,
- et une relation de transition $T \subseteq E \times L \times E$.

Dans le cadre de programmes utilisant des variables, le LTS est augmenté d'une fonction l qui associe à chaque état s de E des affectations de valeurs aux variables V du système, sous la forme d'une conjonction de propositions d'états correspondant aux différentes variables. Un chemin entre deux états s et s' de E est une séquence finie de transitions $a_1, ..., a_n$ de T telles qu'il existe des états intermédiaires $s_1, ..., s_{n-1}$ de E vérifiant :

$$s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} ... \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s'$$

Une définition possible du raffinement au niveau des LTS a été donnée dans [BJK00, Dar02]. Soient les deux LTS $ST_1 = (E_1, I_1, L_1, T_1, l_1)$ et $ST_2 =$

$(E_2, I_2, L_2, T_2, l_2)$ définis comme ci-dessus. Pour faire le lien entre les états abstraits de E_1 et les états concrets de E_2, on utilise un invariant de collage $I_{1,2}$ et une relation de collage $\rho : E_2 \times E_1$ telle que :

$$s_2 \ \rho \ s_1 \Leftrightarrow ((l_2(s_2) \wedge I_{1,2}) \Rightarrow l_1(s_1))$$

Pour vérifier que le LTS ST_1 est raffiné par ST_2, il suffit de montrer que les conditions suivantes sont satisfaites : elles permettent de prouver que les nouvelles transitions introduites par T_2 ne contredisent pas le comportement de ST_1.

1. Le raffinement strict des transitions consiste à vérifier que pour chaque transition de T_2, il existe une transition correspondante dans le LTS abstrait :

$$(s_2 \ \rho \ s_1 \wedge s_2 \xrightarrow{a} s_2' \in T_2) \Rightarrow \exists s_1' (s_1 \xrightarrow{a} s_1' \in T_1 \wedge s_2' \ \rho \ s_1')$$

2. Le bégaiement de transition est l'introduction d'une transition muette τ entre deux états concrets correspondant au même état abstrait :

$$(s_2 \ \rho \ s_1 \wedge s_2 \xrightarrow{\tau} s_2' \in T_2) \Rightarrow s_2' \ \rho \ s_1$$

3. Un état qui n'a pas de transition avec un autre état est un état qui bloque dans le LTS. Il s'agit d'un blocage du système de transitions. Pour ne pas introduire de nouveaux blocages pendant le raffinement, on vérifie que chaque état qui bloque dans ST_2 (noté $\nrightarrow_2$) correspond à un état qui bloque dans ST_1 :

$$(s_2 \ \rho \ s_1 \wedge s_2 \nrightarrow_2) \Rightarrow s_1 \nrightarrow_1$$

4. Pour éviter que les nouvelles transitions τ ne prennent indéfiniment le contrôle, on vérifie la non τ-divergence. Si un état σ_1 de E_1 est collé à plusieurs états concrets, les k états sont distingués par la notation (σ_2, i), $i \in \{1, ..., k\}$.

$$\forall \sigma_2, k \ (\sigma_2 \in \Sigma(T_2) \wedge k \geqslant 0 \Rightarrow \exists a, k' (a \in L_1 \wedge k' \geqslant k \wedge (\sigma_2, k'-1) \xrightarrow{a} (\sigma_2, k') \in T_2))$$

5. Le LTS concret doit préserver le non-déterminisme externe de T_1 :

$$(s_1 \xrightarrow{a} s_1' \wedge s_2 \ \rho \ s_1) \Rightarrow \exists s_2', s_2'', s_1'' (s_2' \ \rho \ s_1 \wedge s_2' \xrightarrow{a} s_2'' \in T_2 \wedge s_1 \xrightarrow{a} s_1'' \in T_1 \wedge s_2'' \ \rho \ s_1'')$$

6. Enfin, pour tout état initial concret, il existe un état initial abstrait qui lui est collé :

$$\forall s_2 \in I_2 \ \exists s_1 \in I_1 : \ s_2 \ \rho \ s_1$$

Il existe toutefois d'autres notions de raffinement des LTS (voir relations de Josephs [Jos88] en section 4.3).

3.6 Sémantique dénotationnelle : traces-divergences

Pour finir cette revue des sémantiques, nous présentons la sémantique dénotationnelle des algèbres de processus comme CSP [Hoa85]. Elle repose sur l'observation du comportement des processus. Les trois principaux modèles sémantiques [Ros97] sont les traces, les échecs stables et les traces-divergences.

Le modèle des traces associe à chaque processus les séquences finies d'événements admises par ce processus. Les traces du processus P sont dénotés par $traces(P)$. Ce modèle permet donc de représenter les comportements possibles des processus sous forme de traces.

Le modèle des échecs stables associe à chaque processus P les couples de la forme (t, E), où t est une trace finie admise par P et E est l'ensemble des événements que le processus ne peut pas exécuter après avoir exécuté les événements de t. L'ensemble de ces couples est noté *failures(P)*. Ce modèle permet de caractériser les blocages de P. En effet, si E est égal à l'ensemble des événements exécutables par P, alors P se retrouve bloqué.

Enfin, le modèle des échecs-divergences associe à chaque processus P l'ensemble de ses échecs stables et l'ensemble de ses divergences. Un processus P n'est divergent que s'il se retrouve dans un état dans lequel les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences de P, noté *divergences(P)*, est l'ensemble des traces t telles que le processus se retrouve dans un état divergent aprés avoir exécuté t. Si le processus est déterministe, alors *divergences(P)* est vide.

Le raffinement consiste alors à calculer et à comparer les modèles sémantiques de deux processus. Le raffinement dépend donc du modèle considéré. Par exemple, dans le cas du modèle des échecs-divergences, si P et Q sont deux processus, alors Q raffine P si :

$$\begin{aligned} failures(Q) &\subseteq failures(P) \\ divergences(Q) &\subseteq divergences(P) \end{aligned}$$

Dans cet exemple, il n'est pas utile de comparer $traces(P)$ et $traces(Q)$, car par définition des échecs stables :

$$failures(Q) \subseteq failures(P) \Rightarrow traces(Q) \subseteq traces(P)$$

3.7 Conclusion

La notion de raffinement dépend donc de la sémantique considérée. Chaque sémantique permet d'interpréter la notion de correction d'une certaine manière et la vérification du raffinement en dépend. Dans l'expression $P < S > Q$, la sémantique fixe un ou plusieurs paramètres pour interpréter les autres.

Relier ces différentes notions de raffinement est difficile, car cela revient à les comparer dans une sémantique commune. Comme chaque sémantique constitue une vue particulière des aspects syntaxiques, cette comparaison n'est pas aisée.

De Roever [dRE98] a comparé les relations de simulations utilisées pour montrer le raffinement dans les langages orientés-modèles comme VDM et Z. Il définit les simulations de manière rigoureuse et compare les méthodes avec des correspondances de Galois. Back [BvW98] introduit son calcul du raffinement à l'aide de la notion générale de contrat et des trois sémantiques décrites ci-dessus : *wp*, *ch* et *gm*. Les fondements mathématiques du calcul du raffinement de Back reposent notamment sur la théorie des catégories.

Concernant les approches basées sur les états et celles basées sur les événements, plusieurs travaux ont permis de relier les sémantiques *wp* et relationnelle avec les modèles sémantiques de CSP : Josephs [Jos88] et Hoare [HHS86] ont défini les premières relations de simulation, tandis que plus récemment Bolton [BD02] et Boiten [DB03] ont affiné les équivalences entre les sémantiques du raffinement.

4 Raffinement : applications dans les méthodes formelles

Les relations de raffinement présentées dans la section précédente sont souvent difficiles à vérifier dans le cas d'exemples précis. Les méthodes formelles utilisent généralement des relations de simulation ou des outils pour assurer la correction du raffinement.

La simulation est une technique qui consiste d'une part à expliciter une relation entre un programme et sa dérivation et d'autre part à vérifier que cette relation satisfait certaines propriétés. Ces conditions sont suffisantes pour assurer la propriété de raffinement. Dans la section précédente, nous avons donné des exemples de simulation dans la sémantique relationnelle et avec les LTS. Nous en présenterons trois autres dans les paragraphes suivants avec les langages B, Z et CSP. Une autre technique consiste à vérifier le raffinement par *model-checking*, comme dans le cas de CSP.

Nous présentons dans cette section les applications du raffinement dans cinq exemples de méthodes formelles : B, B événementiel, CSP, Z et Object-Z.

4.1 Langage B

Le langage B [Abr96] utilise un langage de substitutions généralisées pour modifier les états du système modélisé. Ce langage qui permet de relier les états avant et après l'exécution d'une opération s'appuie sur une notion de transformateur de prédicats qui est proche de celle de Dijkstra (*wp*). Dans

la sémantique de B, l'équivalent de *wp* est dénoté par *str*.

Une substitution S est vue comme une relation. La sémantique de B s'appuie sur le calcul relationnel. Pour assurer la correction des programmes en B, plusieurs relations et prédicats sur les substitutions sont définies. Soit S une substitution quelconque, $pre(S)$ est l'ensemble de précondition, autrement dit l'ensemble des états pour lesquels la précondition de S est satisfaite. La relation $rel(S)$ relie les états avant et les états après l'exécution de S, elle exprime donc la dynamique de la substitution. Les définitions de ces relations sont détaillées dans [Abr96].

Le transformateur *str* vérifie en particulier :

$$str(S)(p) = pre(S) \cap \overline{rel(S)^{-1}[\overline{p}]}$$

$str(S)(p)$ permet de caractériser les états qui assurent la terminaison de S ($pre(S)$) mais qui ne risquent pas d'aboutir aux états de $\overline{p}$ par une des relations dynamiques de $rel(S)$ ($\overline{rel(S)^{-1}[\overline{p}]}$).

Le raffinement B est défini par :

$$S \sqsubseteq S' \Leftrightarrow (\forall a(a \subseteq s \Rightarrow str(S)(a) \subseteq str(S')(a)))$$

En utilisant la propriété exprimant *str* en fonction de *pre* et de *rel*, la relation de raffinement se traduit alors par :

$$S \sqsubseteq S' \Leftrightarrow (pre(S) \subseteq pre(S') \wedge rel(S') \subseteq rel(S))$$

Autrement dit, le raffinement affaiblit les préconditions et retire de l'indéterminisme dans les opérations.

Le raffinement B est vérifié à l'aide d'une relation de simulation *backward*. Par le théorème 11.2.4 du B-Book [Abr96], s'il existe une relation totale $v \in c \leftrightarrow b$ entre l'ensemble concret c de la machine N et l'ensemble abstrait b de la machine M telle que, pour chaque opération,

$$\begin{array}{rcl} v^{-1}[pre(T)] & \subseteq & pre(U) \\ v^{-1} \,;\, rel(U) & \subseteq & rel(T) \,;\, v^{-1} \end{array}$$

où T est la substitution abstraite de l'opération et U sa substitution concrète, alors N est bien un raffinement de M.

Les conditions suffisantes de la simulation sont ensuites traduites en termes d'obligations de preuve sur les invariants et sur les préconditions des opérations. Soient A une machine abstraite B d'invariant I et d'initialisation $Init$ et C un raffinement de A dont l'initialisation est la substitution $Init'$. L'invariant de collage J de la spécification C se déduit de la relation de simulation v présentée ci-dessus : il relie les variables d'état concrètes de C avec les variables abstraites de la machine A. Les obligations de preuve du raffinement sont :

– L'initialisation du raffinement ne doit pas contredire l'initialisation de la machine raffinée :

$$[Init']\neg[Init]\neg J$$

– La machine abstraite et son raffinement contiennent les mêmes opérations : seules leurs subtitutions et leurs préconditions diffèrent. Pour chaque opération, les invariants et la précondition abstraite P doivent d'une part établir la précondition concrète P' et d'autre part éviter que la substitution concrète S' de l'opération n'empêche la substitution abstraite S d'établir l'invariant de collage J :

$$I \wedge J \wedge P \Rightarrow P' \wedge [S']\neg[S]\neg J$$

Le raffinement utilisé en B repose donc sur un concept analogue à celui exprimé avec le transformateur de prédicats *wp*, mais il se restreint de plus à des formes particulières de spécifications regroupées au sein de machines. Le raffinement de S par S' s'exprime sous la forme d'une inclusion de $str(S)$ dans $str(S')$. Pour simplifier la vérification d'une telle propriété, le raffinement est prouvé en B à l'aide d'obligations de preuve suffisantes.

4.2 B événementiel

Le B événementiel [AM98] est une évolution du langage B pour l'adapter à la spécification de systèmes complexes constitués de plusieurs composants.

Les principales différences avec B sont d'une part la considération d'un système fermé pour représenter l'ensemble des composants dans un seul modèle et d'autre part la définition du comportement sous la forme d'événements et non par des opérations comme en B. L'objectif est de prendre en compte l'ensemble du système.

Un événement est défini en B événementiel par une garde, ie. une condition bloquante qui assure la cohérence du système en cas d'exécution de l'événement, et d'une action exprimée à l'aide du langage de substitutions généralisées comme en B. Un événement est de la forme générale :

any $x, y, ...$ **where**
  $P(x, y, ..., v, w, ...)$
then
  $S(x, y, ..., v, w, ...)$
end

avec $x, y, ...$ des variables locales et $v, w, ...$ des constantes ou des variables d'état du système d'événements. Dans cet exemple, P est la garde et S est l'action. Lorsqu'aucune variable locale n'est définie, l'expression d'un événement se simplifie par :

select $P(v, w, ...)$
then $S(v, w, ...)$
end

Le raffinement en B événementiel permet de raffiner les structures de données comme en B, mais aussi de rajouter des détails avec la définition de nouveaux événements. Le raffinement des états s'exprime comme en B à l'aide d'un invariant de collage. Le raffinement au niveau des événements se traduit par un renforcement des gardes et par la préservation de l'invariant de collage. Soit un événement abstrait de la forme :

any x **where** $P(x, v)$
then $v := E(x, v)$
end

qui est raffiné par l'événement concret :

any y **where** $Q(y, w)$
then $w := F(y, w)$
end

avec comme invariant abstrait $I(v)$ et comme invariant de collage $J(v, w)$. Dans ce cas, l'obligation de preuve est :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow \exists x(P(x, v) \wedge J(E(x, v), F(y, w)))$$

Les nouveaux événements permettent de détailler le comportement du système. L'ajout d'un nouvel événement correspond en fait à un raffinement d'un événement qui ne fait rien au niveau abstrait. Ainsi, pour un nouvel événement de la même forme que l'événement concret décrit ci-dessus, l'obligation de preuve du raffinement est la suivante :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow J(v, F(y, w))$$

De plus, les nouveaux événements ne doivent pas prendre le monopole du contrôle : un variant $V(w)$ est défini dans ce but et l'obligation de preuve correspondante est :

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow V(F(y, w)) < V(w)$$

Enfin, une dernière contrainte exprimée par les obligations de preuve du raffinement en B événementiel permet d'éviter un plus grand nombre de blocages au niveau concret qu'au niveau abstrait. Pour chaque événement abstrait de la forme décrite ci-dessus, il faut prouver que :

$$I(v) \wedge J(v, w) \wedge P(x, v) \Rightarrow Q_1 \vee ... \vee Q_n$$

où les Q_i sont les gardes des événements du système concret.

Le raffinement en B événementiel est donc plus complexe qu'en B classique car il permet de rajouter de nouveaux événements. Cette possibilité impose la vérification d'un plus grand nombre d'obligations de preuve pour éviter des contradictions avec le comportement du système raffiné.

4.3 CSP

Le langage CSP [Hoa85] décrit le comportement d'un système sous la forme de processus communiquant les uns avec les autres. La sémantique de CSP repose sur l'observation des effets des processus (voir section 3.6) : modèles des traces, des échecs stables et des traces-divergences.

Comparer les processus dans ces modèles peut se révéler complexe à mettre en œuvre, car le calcul et la comparaison de tels ensembles sont souvent difficiles. Il existe toutefois des outils comme FDR [For97] qui permettent d'analyser les traces, les échecs et les divergences d'un processus. Ils sont cependant limités par la complexité des modèles des expressions de processus. Comme pour le *model-checking*, l'analyse d'un processus avec FDR peut échouer à cause d'une explosion du nombre d'états.

Une autre approche consiste à se ramener à une sémantique opérationnelle des processus sous la forme de LTS. Josephs a défini deux relations de simulation consistantes par rapport au raffinement CSP [Jos88]. Autrement dit, si une des deux relations présentées ci-après est vérifiée, elle est suffisante pour assurer le raffinement au sens CSP. Pour obtenir une condition nécessaire du raffinement, il faut considérer les deux relations de simulation conjointement.

Pour définir ses relations de simulation, Josephs impose comme contrainte que les deux processus aient le même alphabet, c'est-à-dire les mêmes ensembles d'événements pour chacun des processus. Un LTS est défini par la donnée de l'alphabet, des états possibles, de la relation de transition et des états initiaux. Soient $(A, S_1, \longrightarrow_1, R_1)$ et $(A, S_2, \longrightarrow_2, R_2)$ les LTS des processus P_1 et P_2 respectivement avec le même alphabet A. Par convention, l'ensemble des prochains événements du processus P à partir d'un état σ est défini par :

$$next_P(\sigma) = \{e \in A | \exists \sigma' \in S \bullet \sigma \xrightarrow{e} \sigma'\}$$

Si les deux processus ont les mêmes espaces d'états (ie. $S = S_1 = S_2$), alors P_1 est raffiné par P_2 si :

1. À chaque état, les processus peuvent s'engager sur les mêmes événements :
$$\forall \sigma \in S, next_{P_1}(\sigma) = next_{P_2}(\sigma)$$
2. Chaque transition de P_2 est aussi une transition de P_1 :
$$\longrightarrow_2 \subseteq \longrightarrow_1$$

3. Chaque état initial de P_2 est un état initial de P_1 :

$$R_2 \subseteq R_1$$

Josephs définit deux relations de simulation pour des processus dont les espaces d'état sont différents. P_1 est dit une *simulation vers le bas* de P_2 s'il existe une relation $D \subseteq S_1 \times S_2$ telle que :

1. $\forall \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 D \sigma_2 \Rightarrow next_{P_1}(\sigma_1) = next_{P_2}(\sigma_2)$
2. $\forall \sigma_1 \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_1 D \sigma_2 \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2'$
   $\Rightarrow (\exists \sigma_1' \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1' D \sigma_2')$
3. $\forall \sigma_2 \in R_2 \bullet \exists \sigma_1 \in R_1 \bullet \sigma_1 D \sigma_2$

Avec cette règle *forward*, le processus P_1 peut simuler le comportement de P_2 tant que les deux processus sont dans des états correspondants.

Un processus P_1 est dit une *simulation vers le haut* de P_2 s'il existe une relation $U \subseteq S_2 \times S_1$ telle que :

1. $\forall \sigma_2 \in S_2 \bullet \exists \sigma_1 \in S_1 \bullet \sigma_2 U \sigma_1 \Rightarrow next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2)$
2. $\forall \sigma_1' \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_2' U \sigma_1' \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2'$
   $\Rightarrow (\exists \sigma_1 \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_2 U \sigma_1)$
3. $\forall \sigma_1 \in S_1, \sigma_2 \in R_2 \bullet \sigma_2 U \sigma_1 \Rightarrow \sigma_1 \in R_1$

Avec cette relation, si P_2 atteint un certain état, alors P_1 est capable de "retracer" la séquence d'événements afin de trouver un état correspondant à partir duquel P_1 peut simuler le comportement de P_2 : il s'agit d'une simulation *backward*.

Josephs montre que si l'une des deux relations de simulation est vérifiée, alors le processus P_2 est un raffinement de P_1.

4.4 Z et Object-Z

Les règles de simulation présentées dans la section 3.4 sont adaptées aux langages Z [Spi92] et Object-Z [Smi00]. Il existe plusieurs règles de simulation possibles selon les interprétations de la totalité ou non des relations : voir travaux de Bolton, Davies et Woodcock [BDW99, Bol02, BD02] et Smith et Derrick [SD01].

Lorsqu'une relation n'est pas totale, cela signifie que certains états ne sont pas considérés par les programmes. Il existe alors deux interprétations possibles. Si la sémantique considérée est *bloquante*, alors les opérations ne peuvent pas être exécutées en dehors du domaine de la relation : les conditions associées aux états du domaine de la relation sont appelées des *gardes*. Si l'interprétation de la relation partielle est *non bloquante*, alors les opérations peuvent être exécutées en dehors du domaine, mais le résultat n'est pas garanti : en particulier, le système peut alors diverger. Ces conditions sont couramment appelées les *préconditions* des opérations.

Pour revenir à des relations de simulation totales comme définies par He et al. [HHS86], les relations partielles définies dans les sémantiques de Z et d'Object-Z sont rendues totales à l'aide du rajout de nouveaux éléments symbolisant l'échec, la réussite ou l'élément indéfini selon la sémantique considérée. Les relations classiques de simulation utilisées pour raffiner des schémas Z et des classes Object-Z sont présentées dans [DB01]. On se restreint dans la suite à une stratégie bloquante pour les relations partielles. Cela correspond à la sémantique d'Object-Z ou bien à des opérations gardées de Z.

Soient $A = (AState, AInit, \{AOp_i\}_{i \in I})$ et $C = (CState, CInit, \{COp_i\}_{i \in I})$ deux types de données Z. Les initialisations et les opérations relient les états avant et les états après exécution. Par convention, $State$ dénote un état avant alors que $State'$ est un état après. La notation **pre** Op désigne le domaine de la relation Op. La relation R définie sur $AState$ et $CState$ est une simulation vers le bas de A vers C si :

1. $\forall CState' \bullet CInit \Rightarrow \exists AState' \bullet AInit \wedge R'$
2. $\forall i \in I, \forall AState, CState \bullet R \Rightarrow (\textbf{pre } AOp_i \Leftrightarrow \textbf{pre } COp_i)$
3. $\forall i \in I, \forall AState, CState, CState' \bullet R \wedge COp_i \Rightarrow R' \wedge AOp_i$

Concernant la simulation *backward*, T est une relation vers le haut de A vers C si :

1. $\forall CState \bullet \exists AState \bullet T$
2. $\forall AState', CState' \bullet CInit \wedge T' \Rightarrow AInit$
3. $\forall i \in I, \forall CState \bullet \exists AState \bullet T \wedge (\textbf{pre } AOp_i \Rightarrow \textbf{pre } COp_i)$
4. $\forall i \in I, \forall AState', CState, CState' \bullet (T' \wedge COp_i) \Rightarrow \exists AState \bullet T \wedge AOp_i$

Les travaux de Bolton [Bol02] ont montré que les relations de simulation définies pour Z et Object-Z par [DB01] n'étaient pas consistantes par rapport au modèle des échecs-divergences de CSP (voir section 3.6) comme c'est le cas pour les règles de Josephs [Jos88] et de He et al. [HHS86]. Les relations de simulation ci-dessus sont valides et complètes par rapport au modèle des échecs singletons défini par Bolton dans [Bol02]. Ce modèle est un intermédiaire entre le modèle des traces-divergences et celui des échecs-divergences. Contrairement aux échecs stables, les échecs singletons considèrent au plus un événement comme échec, et non pas un ensemble d'événements comme dans la définition. La différence vient du fait que la sémantique relationnelle permet de prendre en compte l'exécutabilité des opérations de manière individuelle alors que le modèle des échecs stables caractérisent les ensembles d'événements.

5 Conclusion : analyse, comparaison et commentaires

Les relations de simulation utilisées pour prouver le raffinement expriment des propriétés différentes selon les langages formels. Ces différences dépendent de la sémantique et de l'expressivité des langages.

Les tableaux 1 et 2 résument les principales caractéristiques de chacune des vérifications de raffinement présentées dans cet article. Les abréviations

Tab. 1 – Comparaison des approches - partie 1

Méthode	Langage B	B évén.	Z et Object-Z
Sémantique	wp	wp	relationnel
Ajout op. ou évén.	non	oui	non
Diversité des raffinements	1 relation	1 relation	2 types : 2 relations ind. 2 relations coll.
Oblig. de preuve	sim. back. [Abr96]	sim. back. [Abr00]	simulations [DB01, Bol02]
Outil de vérif.	Atelier B [Cle] B-Toolkit [B-C]	Atelier B [Cle]	

Tab. 2 – Comparaison des approches - partie 2

Méthode	CSP	
Sémantique	traces, échecs et divergences	LTS
Ajout op. ou évén.	non	oui
Diversité des raffinements	3 modèles et 2 relations	2 relations
Oblig. de preuve	m.c. [Ros97] et sim. [HHS86]	simulation [Jos88]
Outil de vérif.	traces-div. : FDR [For97]	

utilisées dans le tableau sont : sim. back. pour relation de simulation *backward*, sim. pour relations de simulation, m.c. pour *model-checking*, relations

ind. pour les relations de simulation consistantes par rapport au modèle des échecs singletons et relations coll. pour les celles qui sont consistantes par rapport au modèle des échecs stables.

Sémantique. Le raffinement et sa vérification dépendent en premier lieu de la sémantique considérée. Les trois sémantiques présentées dans [BvW98], la sémantique relationnelle, la sémantique opérationnelle des LTS et la sémantique dénotationnelle n'ont pas la même abstraction et la même vision des programmes. Les sémantiques *wp* et *ch* (sections 3.1 et 3.3) sont plus abstraites que la sémantique des jeux (section 3.2), dans le sens qu'il est possible d'exprimer des cas particuliers de *wp* dans la sémantique des jeux, tandis que la réciproque est fausse. D'autre part, les sémantiques *wp* et *ch* sont plus abstraites que la sémantique relationnelle, car elles prennent en compte simultanément les choix angéliques et démoniaques, alors qu'une seule interprétation à la fois n'est possible pour les relations.

Concernant les liens entre les approches basées sur les états et les approches événementielles, la sémantique des jeux qui associe à un contrat une séquence d'instructions semble analogue au modèle des traces de CSP (section 3.6) qui associe à un processus une séquence d'événements. Le modèle des échecs-divergences permet en outre de caractériser les blocages et les divergences du processus. Ce modèle est plus expressif que la sémantique *wp* (voir section 4.4) : la sémantique des transformateurs de prédicats correspond en fait au modèle des échecs singletons et des divergences de Bolton [Bol02]. Toutefois, la sémantique *wp* permet de représenter les opérations qui sortent des préconditions, cela n'est pas possible avec les LTS ou le modèle des échecs-divergences. Enfin, Derrick et Boiten montrent qu'il est possible de faire correspondre une sémantique au modèle des échecs-divergences en caractérisant les finalisations des programmes dans la sémantique relationnelle [DB03].

Simulations et outils. Les relations de simulation permettent de prouver le raffinement entre deux systèmes à l'aide de conditions suffisantes. Il existe deux principales relations : *forward* et *backward*. Toutes les méthodes formelles ne proposent pas les deux. Dans les cas de B et B événementiel, le raffinement est prouvé à l'aide d'obligations de preuve correspondant à une simulation *backward*. La disponibilité d'outils a également son importance : une méthode n'utilisant qu'un seul type de relation avec un bon outil peut s'avérer dans la pratique plus efficace et plus utile qu'une méthode sans outil.

Plusieurs notions de raffinement. Nous avons présenté quatre formes principales de raffinement : *wp*, séquences d'opérations, LTS et échecs-divergences. Le raffinement de séquences d'opérations (comme les types de

données ou les machines B) permet de réduire le non déterminisme et d'augmenter les préconditions. Cette notion de raffinement est prouvée à l'aide de conditions suffisantes, grâce à des relations de simulation. Les algèbres de processus proposent des relations de raffinement plus ou moins fines selon les modèles sémantiques. Le raffinement peut alors être prouvé par *model-checking*. Il existe également des relations de simulation exprimées à l'aide de LTS pour prouver le raffinement en CSP. Le raffinement s'adapte donc aux caractéristiques et aux objectifs de chaque méthode et il constitue un critère important dans la sélection d'un langage de spécification formel.

Références

[Abr96] J.R. Abrial. *The B-Book : Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr00] J.R. Abrial. Guidelines to formal systems studies. ClearSy, November 2000.

[Acz75] P. Aczel. Quantifiers, games and inductive definitions. In *Proc. 3rd Scandinavian Logic Symposium*, North Holland, 1975.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. Technical report, Digital Systems Research Center, Palo Alto, California, 1988.

[AM98] J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *Second Conference on the B Method*, volume 1393 of *LNCS*, pages 83–128, 1998.

[B-C] B-Core (UK) Ltd. B-Toolkit. `http ://www.b-core.com/btoolkit.html`.

[Bac78] R.J. Back. *On the correctness of refinement in program development*. PhD thesis, University of Helsinki, 1978.

[Bac88] R.J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25 :593–624, 1988.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[BD02] C. Bolton and J. Davies. A comparison of refinement orderings and their associated simulation rules. In *Proc. of Refine 2002*, volume 70 of *ENTCS*. Elsevier Science Publishers, 2002.

[BDW99] C. Bolton, J. Davies, and J. Woodcock. On the refinement and simulation of data types and processes. In *IFM*, pages 273–292, 1999.

[BJK00] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Proc. FASE 2000*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[Bol02] Christie Bolton. *On the refinement of state-based and event-based models*. PhD thesis, New College, Hilary Term, 2002.

[BvW89] R.J. Back and J. von Wright. Duality in specification languages : a lattice-theoretical approach. Technical Report 77, Abo Akademi, 1989.

[BvW98] R.J. Back and J. von Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Cle] Clearsy. Atelier B. `http ://www.atelierb-societe.com`.

[Dar02] Christophe Darlot. *Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements*. PhD thesis, Université de Franche-Comté, 2002.

[dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[DB03] J. Derrick and E. Boiten. Reconciling event and state-based notions of refinement. ST.EVE Workshop at FM03, September 2003.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[dRE98] W.P. de Roever and K. Engelhardt. *Data Refinement : Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[DW96] J. Davies and J.C.P. Woodcock. *Using Z : Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[For97] Formal Systems (Europe) Ltd. Failures-Divergences Refinement : FDR2 User Manual. `http ://www.formal.demon.co.uk`, 1997.

[FSD03] M. Frappier and R. St-Denis. EB^3 : an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2) :134–149, July 2003.

[HHS86] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *ESOP 86 : European Symposium on Programming*, Saarbrucken, West Germany, 17-19 March 1986. Springer-Verlag.

[Hin72] J. Hintikka. *Language games and information*. Clarendon, 1972.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jon90] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[Jos88] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3 :9–18, 1988.

[Mil83] A. Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20 :315 – 328, 1983.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9 :287–306, 1987.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Mos72] Y.N. Moschovakis. The game quantifier. In *Proc. of the American Mathematical Society*, pages 245 – 250, 1972.

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

[SD01] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18 :249–284, May 2001.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi92] J.M. Spivey. *The Z Notation : a Reference Manual*. Prentice-Hall, 1992.

[Tar41] A. Tarski. On the calculus of relations. *Symbolic Logic*, 6 :73 – 89, 1941.