Preuve automatique dans un environnement de développement UNITY

Michel Charpentier[†] — **Abdellah El Hadri**[‡] — **Gérard Padiou**[†]

[†] *ENSEEIHT-IRIT*
2, rue Charles Camichel
31071 TOULOUSE cedex
e-mail : {charpov, padiou}@enseeiht.fr

[‡] *EMI*
Avenue Ibn Sina BP 765
RABAT (MAROC)
e-mail : elhadri@emi.ma

RÉSUMÉ. *Nous considérons le problème de la vérification automatique de propriétés de systèmes réactifs. En effet, leur caractère parallèle, non-déterministe et éventuellement réparti conduit à des fautes de conception fréquentes et à une mise au point plus difficile. Le développement de programmes prouvés par rapport à une spécification peut apparaître comme une solution à ce double problème. Dans ce cadre, nous décrivons la réalisation d'un environnement fondé sur le formalisme* `UNITY`*. Pour réaliser les preuves nécessaires, celui-ci utilise un démonstrateur de théorèmes et un calculateur de formules* Presburger*. A partir de quelques exemples significatifs, nous exposons les principales difficultés rencontrées.*

ABSTRACT. *We address the problem of automatic verification of reactive systems. The parallelism, the non-determinism and the distribution of these systems lead to frequent design flaws and make debugging difficult. Proving programs with respect to their specification may solve both these problems. In this framework, we describe the implementation of a tool based upon the* `UNITY` *formalism. A theorem prover and a* Presburger *formula calculator are used to perform the underlying proofs. We illustrate the main difficulties encountered with representative examples.*

MOTS-CLÉS : *Preuve de programme, programmes réactifs, modèle* `UNITY`*, démonstration automatique.*

KEY WORDS : *Program proof, reactive programs,* `UNITY` *formalism, theorem proving.*

1. Contexte général de l'étude

Le développement des programmes parallèles et répartis corrects est un processus complexe. En effet, le parallélisme et la répartition du contrôle présents dans les systèmes réactifs engendrent deux difficultés supplémentaires par rapport à la programmation séquentielle. Le parallélisme conduit à une explosion des états possibles du programme lors de son exécution et la répartition entraîne la perte de toute connaissance immédiate d'un état global et d'un référentiel de temps unique. Dans ces conditions, la mise au point de tels programmes par le test s'avère en particulier très délicate [FRO 94].

Pour faire face à ces problèmes, il semble nécessaire de chercher à accroître la rigueur et la formalisation des méthodes et outils utilisés lors d'un développement.

Dans cet esprit, deux approches peuvent exister. En premier lieu, on peut concentrer les efforts sur le suivi et le contrôle du processus de développement. Plusieurs projets d'atelier de développement sont en cours de réalisation [SOU 93]. En deuxième lieu, on peut aussi améliorer les opérations de développement proprement dites. Dans ce cas, on peut espérer converger plus rapidement vers un programme correct par l'utilisation de méthodes ou d'outils bien formalisés [SHA 93a, HER 93]. C'est cette dernière approche que nous explorons.

Le développement de programmes par raffinements successifs est une méthode classique et bien établie. La notion de raffinement peut être formalisée [BAC 88, BAC 91]. Un développement est considéré comme une suite de couples $(Spec, Prog)$ où $Spec$ est une spécification et $Prog$ est un programme. Partant d'une version initiale, les transformations successives engendrent de nouveaux couples $(Spec, Prog)$ qui conduisent à un programme final validé par rapport à sa spécification. De plus, chaque opération de raffinement doit posséder une sémantique précise permettant de contrôler sa validité dans le contexte d'un développement particulier.

La description d'un programme selon deux points de vue différents, une spécification et une implantation, permet de mieux appréhender le problème, chaque description étant complémentaire. Il faut cependant contrôler la cohérence des deux représentations.

Une telle démarche générale nécessite pour être expérimentée un modèle de calcul parallèle possédant quelques propriétés de base :

— offrir un langage de spécification et un langage de programmation ;
— permettre la description de systèmes réactifs incluant parallélisme et répartition ;
— être bien formalisé ;
— permettre une approche par raffinement en conservant le même formalisme durant toute la chaîne de développement.

Compte tenu de ces impératifs, le modèle `UNITY`, proposé par Chandy et Misra [CHA 88], a été choisi comme base expérimentale. Le langage de spécification permet d'exprimer formellement des propriétés de sûreté et de vivacité. Le langage de programmation permet l'expression des deux formes de parallélisme : synchrone ou asynchrone. Malgré son haut niveau d'abstraction, le modèle `UNITY` permet la description d'actions et de données élémentaires jusqu'à la conception d'architectures [DHA 94].

Le langage UNITY peut aussi être considéré comme un langage intermédiaire dont la traduction vers un langage parallèle plus classique tel que *CLinda* ou *Occam* [BRO 94] est possible par compilation. Enfin, le modèle est suffisamment général pour permettre des raffinements successifs tout en conservant le même formalisme.

Dans ce contexte, la formalisation d'un développement comporte deux aspects :

— la recherche et la formalisation d'opérations de raffinements les plus génériques possibles ;
— le contrôle de la cohérence de tout couple $(Spec, Prog)$. Il s'agit de prouver que le programme $Prog$ vérifie bien les spécifications correspondantes $Spec$.

En ce qui concerne le premier point, nous soulignerons que l'avantage d'une approche par raffinements successifs est que seule la preuve de l'énoncé initial $(Spec, Prog)$ est nécessaire dans la mesure où les raffinements sont sains. Bien que cet aspect ne soit pas abordé par la suite, nous nous intéressons sur ce point, à des raffinements permettant de passer d'un algorithme de type centralisé à une solution répartie.

Le deuxième point soulève quant à lui le problème de la preuve d'un programme par rapport à une spécification. Nous étudions donc les possibilités d'automatisation de ce type de preuve dans le contexte d'un environnement de développement UNITY.

Après une présentation rapide du formalisme UNITY, nous décrivons d'abord les fonctionnalités de l'environnement, puis sa réalisation. Nous détaillons plus particulièrement l'utilisation de démonstrateurs de théorèmes.

2. Le formalisme UNITY

Le formalisme UNITY se compose d'un langage de programmation dont la sémantique est fondée sur les systèmes de transitions et d'un langage de spécification fondé sur la logique temporelle linéaire, capable d'exprimer les propriétés des systèmes de transitions décrits.

2.1. *Le langage UNITY*

Un programme UNITY est composé de *quatre* sections :

declare

Il s'agit d'un ensemble de déclarations de variables tel qu'on peut le trouver dans les langages de programmation classiques. Les types de base utilisés sont les *entiers* et les *booléens.* Sont également utilisés les *ensembles* et *suites* (ou *listes*) d'entiers.

initially

Cette partie concerne l'initialisation des variables. Chaque variable est initialisée au plus une fois avec une expression de son type. Les variables non initialisées ont des valeurs arbitraires au commencement de l'exécution du programme.

always

La clause **always** d'un programme `UNITY` est une suite de *macro-définitions* liant les variables du programme.

assign

La partie **assign** constitue le corps proprement dit du programme, c'est-à-dire l'ensemble (non ordonné) des instructions qui le composent. Ces instructions sont séparées par le symbole d'exclusion $[\!]$.

La sémantique de l'exécution d'un programme est la répétition infinie du processus suivant :

— choix d'une instruction et
— exécution *atomique* de cette instruction.

La seule hypothèse sur le choix est celle d'une équité faible : chaque instruction est exécutée infiniment souvent. La terminaison d'un programme correspond à l'atteinte d'un point fixe.

L'unique instruction élémentaire est l'affectation multiple gardée et synchrone de la forme

$$\begin{array}{lcll} v_1, \cdots, v_n & := & e_{11}, \cdots, e_{1n} & if\ g_1 \ \sim \\ & & e_{21}, \cdots, e_{2n} & if\ g_2 \ \sim \\ & & \cdots & \cdots \\ & & e_{m1}, \cdots, e_{mn} & if\ g_m \end{array}$$

A la différence du langage de *commandes gardées* de Dijkstra, les valeurs affectées doivent être *identiques* dans le cas de plusieurs gardes simultanément vraies. Chaque affectation est donc *déterministe*. L'instruction est sans effet (*nop*) dans le cas où aucune garde n'est valide.

A ce format d'instructions s'ajoute une facilité d'expression des affectations synchrones. Le symbole $\|$ permet de séparer plusieurs parties d'une même affectation. Par exemple,

$$x := a \parallel y, z := b, c \parallel t := d$$

équivaut à

$$x, y, z, t := a, b, c, d$$

Les opérateurs peuvent être quantifiés, en particulier *parallèle* ($\|$) et *exclusion* ($[\!]$). Par exemple, $\langle \| \ i : 1 \leq i \leq N :: A[i] := B[i] \rangle$ dénote l'affectation synchrone des éléments du tableau A avec les valeurs du tableau B.

2.2. *La logique UNITY*

Au langage précédemment décrit sont associés un langage de spécification et un système de preuve de ces spécifications. Ce langage permet de spécifier des propriétés de sûreté et de vivacité par des expressions logiques faisant intervenir les variables manipulées par le programme. Ces propriétés peuvent ensuite être prouvées formellement au sein d'une théorie et à l'aide d'un ensemble de théorèmes.

La logique `UNITY` est fondée sur des assertions de la forme $\{P\}\ s\ \{Q\}$ (*clauses de Hoare* [HOA 69]) où P et Q sont des prédicats logiques et s une instruction du programme. Une telle écriture signifie que *si l'instruction s débute dans un état du programme satisfaisant P et si elle se termine, l'état résultant vérifiera alors la propriété Q.*

Un programme `UNITY` ne possédant aucun point de contrôle, les propriétés envisagées ne peuvent être attachées à un point précis du programme. Elle sont donc *globales* et s'expriment *indépendamment du texte du programme*. Pour un programme $Prog$ donné et des prédicats P et Q, elles sont généralement de la forme :

$$\begin{array}{lllll} & \forall s,\ s\ \mathbf{in}\ Prog & \{P\}\ s\ \{Q\} & \text{(sûreté)} \\ \text{ou} & \exists s,\ s\ \mathbf{in}\ Prog & \{P\}\ s\ \{Q\} & \text{(vivacité)} \end{array}$$

où l'opérateur **in** dénote l'appartenance de s à l'ensemble des affectations du programme $Prog$

Le système de preuve sous-jacent fait l'hypothèse que toute instruction est assurée de se terminer. Ceci permet, au niveau des propriétés de sûreté, de se dispenser de l'utilisation d'une logique particulière (logique des *at, in, after*) et de se limiter à la logique classique [LAM 80].

Syntaxe des propriétés de sûreté

Les propriétés de sûreté sont exprimées à partir de quatre opérateurs logiques fondamentaux : **Unless**, **Stable**, **Constant** et **Invariant**. Pour un programme $Prog$ et des prédicats P et Q, elles sont définies comme suit :

$$P\ \mathbf{Unless}\ Q \;\equiv\; \langle \forall s : s\ in\ Prog :: \{P \wedge \neg Q\} s \{P \vee Q\}\rangle$$

(si P est vérifié à un instant donné, P est maintenu jusqu'à ce que Q devienne valide).

$$\mathbf{Stable}\ \ P \;\equiv\; P\ \ \mathbf{Unless}\ \ false$$

(si P est vérifié à un instant donné, P sera indéfiniment maintenu par la suite).

$$\mathbf{Constant}\ \ P \;\equiv\; (\mathbf{Stable}\ \ P) \wedge (\mathbf{Stable}\ \ \neg P)$$

(la valeur de vérité de P ne change pas au cours de l'exécution du programme).

$$\mathbf{Invariant}\ \ P \;\equiv\; (conditions\ \ initiales\ \Rightarrow\ P) \wedge (\mathbf{Stable}\ \ P)$$

(P est vérifié au début de l'exécution du programme et à tout instant par la suite).

Syntaxe des propriétés de vivacité

La propriété élémentaire de vivacité est le **Ensures** défini comme suit pour un programme $Prog$ et des prédicats P et Q :

$$P \ \textbf{Ensures} \ Q \ \equiv \ (P \ \textbf{Unless} \ Q) \wedge \langle \exists s : s \ \textbf{in} \ Prog :: \{P \wedge \neg Q\} s \{Q\} \rangle$$

(si P est vérifié à un instant donné, P est maintenu au moins jusqu'à ce que Q devienne valide et il existe une instruction qui établira finalement Q).

Cette propriété permet de définir la propriété de vivacité essentielle : le **leads-to**, noté $\mapsto$, défini par trois axiomes de fermeture transitive et de disjonction infinie du **Ensures** ($P \mapsto Q$ signifie que si P est vérifié à un instant donné, Q le sera à un instant ultérieur).

2.3. *Exemple*

On considère le programme suivant : il utilise deux compteurs croissants i et j dont l'évolution est liée. Les valeurs initiales de ces compteurs sont déterminées par la clause **initially**.

$$\begin{array}{lll}
\textbf{initially} & i, j \ = \ 0, 10; & \\
\textbf{assign} & & \\
 & i := i + 1 & \\
[\!] & j := i & \quad \textbf{if} \ \ i \neq 5
\end{array}$$

Ce programme vérifie les spécifications de sûreté et de vivacité suivantes :

$\forall\, k \ \ \textbf{Stable} \ \ i \geq k$	– i ne peut que croître
$\forall\, k \ \ i = k \ \ \textbf{Unless} \ \ i = k + 1$	– i ne peut que croître par pas de 1
$\forall\, k \ \ i = k \ \ \textbf{Ensures} \ \ i = k + 1$	– i croît effectivement par pas de 1
$\textbf{Invariant} \ \ i \geq 0$	– i est toujours positif ou nul
$\textbf{Invariant} \ \ i \geq 0 \wedge j \geq 0$	– i et j sont toujours positifs ou nuls
$\forall\, k \ \ j = k \ \ \mapsto \ \ j > k$	– j prend des valeurs arbitrairement grandes

Remarque sur la notion d'invariance

Tel que nous venons de le définir, un invariant I est une propriété valide dans l'état initial du programme et maintenue par toute transition de ce programme (stable).

On impose donc que de tout état vérifiant I, une transition conduise à un état vérifiant encore I. La seule hypothèse sur l'état à partir duquel est effectuée la transition est qu'il satisfait I. En particulier, il n'est fait aucune hypothèse sur l'accessibilité de cet état. S'il existe un état à partir duquel I n'est pas maintenu, I n'est pas un invariant, même si cet état ne peut jamais être atteint.

Une propriété peut donc être vraie pour toutes les exécutions d'un programme et ne pas être un invariant, c'est-à-dire ne pas être maintenue à partir des états inaccessibles du programme. Par exemple, dans le programme précédent, on n'a pas $\textbf{Invariant} \ \ j \geq 0$ bien que j soit positif dans toute exécution du programme.

En effet, à partir de l'état (inaccessible) $(i, j) = (-1, 2)$, la seconde instruction ne maintient pas $j \geq 0$.

Une forme plus faible d'invariance a été introduite, qui correspond bien à la notion de propriété *toujours vraie* lors de l'exécution d'un programme [SAN 91]. Elle est fondée sur la notion de *plus fort invariant*. Le *plus fort invariant* d'un programme correspond à la propriété invariante la plus forte (au sens de l'implication) d'un programme. Il est défini comme la conjonction de tous les invariants de ce programme. En d'autres termes, le *plus fort invariant* est valide dans tous les états accessibles d'un programme et seulement dans ceux-là. Il permet donc une caractérisation des états accessibles d'un programme qui évite la difficulté précédente. Une propriété *toujours vraie* (c'est-à-dire un invariant faible) est alors une propriété impliquée par le *plus fort invariant*.

En général, la difficulté pour prouver une propriété *toujours vraie* d'un programme est de trouver un invariant suffisamment fort qui l'implique, c'est-à-dire de déterminer de façon suffisamment précise les états accessibles du programme. Certains outils automatiques de preuve de programme permettent "l'enrichissement" d'une propriété *toujours vraie* pour déterminer un invariant suffisant (cf **6.2**), voire de calculer le *plus fort invariant* dans certaines conditions déterminées (cf **6.1**).

3. L'environnement de développement

3.1. *Description*

L'environnement réalisé se présente sous la forme d'un éditeur syntaxique (du langage `UNITY` et de sa logique) sous `X Window System`. Cet éditeur a été engendré par le `Grammatech Synthesizer Generator` et possède toutes les fonctionnalités des éditeurs engendrés par cet outil dont le détail peut être trouvé dans [REP 89].

Cet environnement gère un texte composé de deux parties :

— l'expression d'un algorithme dans le langage `UNITY` ;
— une spécification des propriétés de sûreté et de vivacité de cet algorithme exprimée dans la logique `UNITY`.

La figure 1 montre un exemple de l'interface utilisateur offerte par l'environnement. Chaque instruction du programme est suivie par la liste des références aux propriétés à vérifier. Chaque référence disparaît si la propriété correspondante parvient à être démontrée.

L'objectif est de garantir automatiquement la cohérence entre le programme et sa spécification (c'est-à-dire de vérifier que le programme respecte bien toutes les contraintes spécifiées). Pour cela, nous utilisons deux outils automatiques : le démonstrateur de théorèmes `Otter` et le calculateur de formules *Presburger* `Omega`.

Actuellement, seul un sous-ensemble des fonctionnalités d'`UNITY` est pris en compte par l'environnement. En ce qui concerne le langage, seules les variables scalaires entières et booléennes sont acceptées. L'absence de tableaux rend inutile la quantification des instructions. Cette limitation est une option temporaire facilitant une première réalisation. En ce qui concerne la logique, l'opérateur $\mapsto$ n'est pas traité. En effet, la vérification de telles propriétés sur un programme présente des difficultés tant théoriques que pratiques (calcul de fermeture transitive).

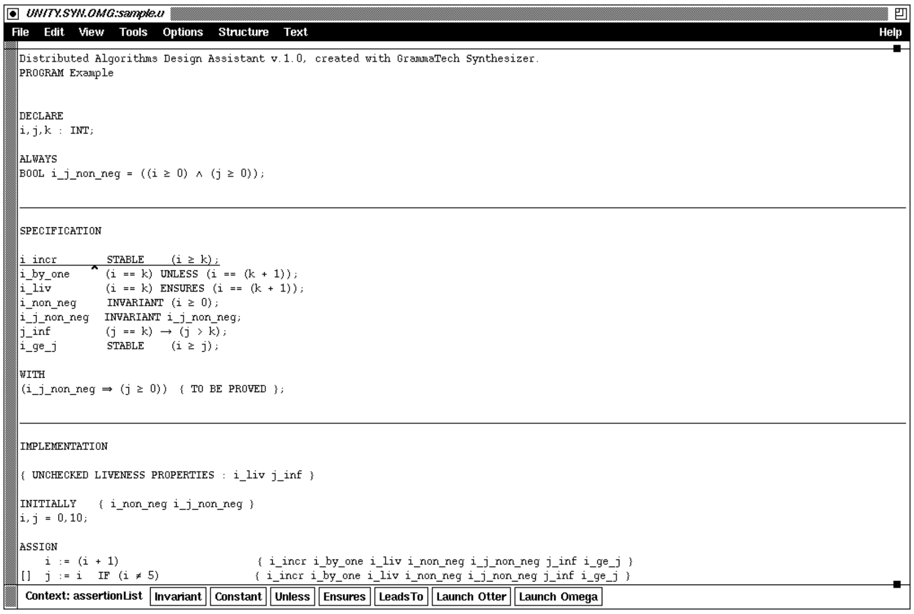

Figure 1. *Exemple de l'interface utilisateur*

3.2. *Utilisation*

Des exemples d'utilisation de l'environnement ont été étudiés dans différents domaines, en particulier ceux des systèmes répartis, des circuits logiques ou encore de la robotique. Nous avons choisi de présenter ici trois exemples consacrés à la synchronisation dans les systèmes et ayant souvent fait l'objet de preuve dans le contexte d'autres projets.

On retrouve dans ces exemples les différentes sections d'un programme (**declare**, **always**, **initially** et **assign**) ainsi qu'une section particulière (**with**) qui fournit une interface directe avec le démonstrateur. Elle permet d'obtenir directement d'un démonstrateur la preuve de prédicats (non temporels) portant sur les variables du programme. Ces preuves sont indépendantes de celles associées au contrôle des spécifications.

Algorithme d'exclusion mutuelle de Peterson

Ce premier exemple est extrait du manuel de référence du *UNITY Verifier* [KAL 94a] (cf **6.1**). Il décrit l'algorithme de Peterson qui assure l'exclusion mutuelle entre deux processus. Le programme et sa spécification sont dus à J. Misra [MIS 90]. Les cinq premières instructions modélisent le comportement du premier processus, les cinq suivantes celui du second. Les deux dernières affectations représentent le non déterminisme des processus susceptibles de ne jamais entrer en zone critique. Les variables m et n sont les compteurs ordinaux des deux processus. En particulier, l'égalité $m = critical$ signifie que le premier processus est en zone critique.

Onze propriétés de sûreté et de vivacité sont exprimées dans la spécification de ce programme. La première (**Invariant** I_UN) permet d'établir la propriété de sûreté fondamentale de ce type d'algorithme : l'exclusion mutuelle, exprimée par le fait que $EXCLUSION$ est une propriété *toujours vraie* du programme. On notera que la preuve de l'implication $I\_UN \Rightarrow EXCLUSION$ est réalisée par l'un des démonstrateurs en utilisant la clause **with**.

Il est important de noter que les invariants $trois$, $cinq$, six et $sept$ ne sont pas validés par l'environnement. Ces propriétés étaient d'ailleurs prouvées fausses par le *UNITY Verifier* et correspondent en fait à une démonstration (manuelle) erronée dans la présentation originale de [MIS 90].

Introduit dans l'environnement, le programme se présente sous la forme suivante (l'expression TEX du programme est produite par l'environnement lui-même) :

```
Program Mutex
declare
    m,n : INT; u,v,p : BOOL; hu,hv : BOOL

always
    INT noncritical = 0,
    INT requesting = 1,
    INT trying = 2,
    INT critical = 3,
    INT exiting = 4,
    BOOL EXCLUSION = ¬((m = critical) ∧ (n = critical)),
    BOOL I_HUIT = ((u ⇔ ((m ≥ requesting) ∧ (m ≤ critical))) ∧ ((m = critical) ⇒ ¬p)),
    BOOL I_NEUF = ((v ⇔ ((n ≥ requesting) ∧ (n ≤ critical))) ∧ ((n = critical) ⇒ p)),
    BOOL I_UN = (EXCLUSION ∧ (I_HUIT ∧ I_NEUF))
```

```
Specifications
    un          invariant  I_UN,
    deux        invariant  (u ⇔ ((m ≥ requesting) ∧ (m ≤ critical))),
    trois       invariant  (((m = critical) ∨ (m = exiting)) ⇒ ¬p),
    quatre      invariant  (v ⇔ ((n ≥ requesting) ∧ (n ≤ critical))),
    cinq        invariant  (((n = critical) ∨ (n = exiting)) ⇒ p),
    six         invariant  ((u ⇔ ((m ≥ requesting) ∧ (m ≤ critical)))
                           ∧ (((m = critical) ∨ (m = exiting)) ⇒ ¬p)),
    sept        invariant  ((v ⇔ ((n ≥ requesting) ∧ (n ≤ critical)))
                           ∧ (((n = critical) ∨ (n = exiting)) ⇒ p)),
    huit        invariant  ((u ⇔ ((m ≥ requesting) ∧ (m ≤ critical))) ∧ ((m = critical) ⇒ ¬p)),
    neuf        invariant  ((v ⇔ ((n ≥ requesting) ∧ (n ≤ critical))) ∧ ((n = critical) ⇒ p)),
    dix         (m = trying) unless (m = critical),
    onze        (m = requesting) ensures ((p ⇔ v) ∧ (m = trying))

with
    (I_UN ⇒ EXCLUSION)
```

```
implementation

initially
    u = false;
    v = false;
    m = noncritical;
    n = noncritical

assign
    u,m := true,requesting          if (hu ∧ (m = noncritical))
[]
    p,m := v,trying                 if (m = requesting)              { cinq sept }
[]
    m := critical                   if (¬p ∧ (m = trying))
```

```
[]
   u,m := false,exiting                 if (m = critical)
[]
   p,m := true,noncritical              if (m = exiting)
[]

   v,n := true,requesting               if (hv ∧ (n = noncritical))
[]
   p,n := ¬u,trying                     if (n = requesting)           { trois six }
[]
   n := critical                        if (p ∧ (n = trying))
[]
   v,n := false,exiting                 if (n = critical)
[]
   p,n := false,noncritical             if (n = exiting)
[]

   hu := ¬hu
[]
   hv := ¬hv

end {Mutex}
```

Exclusion mutuelle avec tolérance aux fautes

Dans cet exemple, nous étudions le problème de l'exclusion mutuelle dans un contexte réparti en mettant l'accent sur l'aspect tolérance aux fautes. Une solution classique de ce problème consiste à faire circuler un message unique entre les sites. Cet algorithme dit du "jeton" assure la propriété d'exclusion mutuelle dans la mesure où seul le site possédant le message-jeton possède le droit (privilège) d'entrer en exclusion. A cette propriété de sûreté s'ajoute une propriété de vivacité devant garantir à tout site demandeur qu'il entrera finalement en exclusion. Cette équité est souvent simplement obtenue en faisant circuler le jeton sur un anneau logique passant par tous les sites.

Dans le formalisme UNITY, il nous faut donc décrire les propriétés de sûreté et de vivacité précédentes. Une première solution est présentée dans le programme ci-dessous. On suppose une architecture comportant trois sites. Chaque site est représenté par une suite d'instructions similaires aux noms de variables près. En effet, tous les sites ont le même comportement générique.

Un site i quelconque peut demander à entrer en exclusion en affectant la valeur vraie à la variable s_i_dem. Ce site n'entrera en exclusion que lorsque la variable s_i_excl deviendra vraie. Ceci est conditionné à la présence du jeton sur le site i, mémorisée par la variable s_i_j. Après quoi, la sortie d'exclusion se traduira par une remise à faux des variables s_i_dem et s_i_excl. Ce comportement peut être répété à l'infini.

La circulation du jeton est décrite, quant à elle, par la mise à jour d'un couple quelconque de variables (s_i_j, s_k_j). L'unicité du jeton repose sur la preuve que dans tout état du programme, une variable s_i_j et une seule est vraie.

En outre, on suppose possible la perte du jeton par l'un quelconque des sites. Un site privilégié (ici, le site 1) peut alors débuter une procédure de régénération du jeton. Il propage alors une enquête sur l'anneau à l'aide des variables s_i_enq. Si le résultat de cette enquête lui permet de confirmer que le jeton n'est effectivement présent sur aucun site, il procède à la création d'un nouveau jeton.

```
Program jeton

declare
    s1_dem,s2_dem,s3_dem: BOOL;
    s1_excl,s2_excl,s3_excl: BOOL;
    s1_j,s2_j,s3_j: BOOL;
    s1_enq,s2_enq,s3_enq: BOOL

always
    BOOL J_A_S1 = ((s1_j ∧ ¬s2_j) ∧ ¬s3_j),
    BOOL J_A_S2 = ((s2_j ∧ ¬s3_j) ∧ ¬s1_j),
    BOOL J_A_S3 = ((s3_j ∧ ¬s1_j) ∧ ¬s2_j),
    BOOL J_UNIQUE = ((J_A_S1 ∨ J_A_S2) ∨ J_A_S3),
    BOOL J_EXISTE = ((s1_j ∨ s2_j) ∨ s3_j),
    BOOL EXCL_A_S1 = (s1_dem ∧ (s1_excl ∧ (¬s2_excl ∧ ¬s3_excl))),
    BOOL EXCL_A_S2 = (s2_dem ∧ (s2_excl ∧ (¬s1_excl ∧ ¬s3_excl))),
    BOOL EXCL_A_S3 = (s3_dem ∧ (s3_excl ∧ (¬s1_excl ∧ ¬s2_excl))),
    BOOL EXCL_LIBRE = ¬((s1_excl ∨ s2_excl) ∨ s3_excl),
    BOOL EXCLUSION = ((EXCL_A_S1 ∨ EXCL_A_S2) ∨ EXCL_A_S3),
    BOOL ENQ_JETON_1 =   ((s1_enq ⇒ (¬s1_j ∧ ¬s1_excl)) ∧
                         ((s2_enq ⇒ (¬s2_j ∧ ¬s2_excl)) ∧
                         (s3_enq ⇒ (¬s3_j ∧ ¬s3_excl)))),
    BOOL ENQ_JETON_2 = (((s1_j ⇒ ¬s1_enq) ∧ (s2_j ⇒ ¬s2_enq)) ∧ (s3_j ⇒ ¬s3_enq)),
    BOOL UNICITE = ((J_UNIQUE ∨ ¬J_EXISTE) ∧ ENQ_JETON_2),
    BOOL SI_EXCL =       ((s1_excl ⇒ (s1_dem ∧ (¬s2_j ∧ ¬s3_j))) ∧
                         ((s2_excl ⇒ (s2_dem ∧ (¬s1_j ∧ ¬s3_j))) ∧
                         ((s3_excl ⇒ (s3_dem ∧ (¬s1_j ∧ ¬s2_j))) ∧ (UNICITE ∧ ENQ_JETON_1))))
```

```
Specifications
    SI_EXCL       invariant SI_EXCL,
    ENQ_JETON_1   invariant ENQ_JETON_1,
    ENQ_JETON_2   invariant ENQ_JETON_2,
    UNICITE       invariant UNICITE,
    EXCLUSIF      invariant ((EXCLUSION ∨ EXCL_LIBRE) ∧ SI_EXCL)

with
    (J_A_S1 ⇒ ¬(J_A_S2 ∨ J_A_S3)),
    (J_A_S1 ⇔ (J_EXISTE ∧ (¬s2_j ∧ ¬s3_j)))
```

```
implementation

initially
    s1_j,s2_j,s3_j = true,false,false;
    s1_dem,s2_dem,s3_dem = false,false,false;
    s1_excl,s2_excl,s3_excl = false,false,false;
    s1_enq,s2_enq,s3_enq = false,false,false

assign
    s1_dem := true
[]
    s1_excl := true                         if (s1_j ∧ s1_dem)
[]
    s1_dem,s1_excl := false,false           if s1_excl
[]
    s1_j := false

[]
    s2_dem := true
[]
    s2_excl := true                         if (s2_j ∧ s2_dem)
[]
    s2_dem,s2_excl := false,false           if s2_excl
[]
    s2_j := false
```

```
▯
   s3_dem := true
▯
   s3_excl := true                      if (s3_j ∧ s3_dem)
▯
   s3_dem,s3_excl := false,false        if s3_excl
▯
   s3_j := false

▯
   s1_j,s2_j := false,true              if (s1_j ∧ ¬s1_dem)
 ‖ s2_enq := false                      if (s1_j ∧ ¬s1_dem)
▯
   s2_j,s3_j := false,true              if (s2_j ∧ ¬s2_dem)
 ‖ s3_enq := false                      if (s2_j ∧ ¬s2_dem)
▯
   s3_j,s1_j := false,true              if (s3_j ∧ ¬s3_dem)
 ‖1 s1_enq := false                     if (s3_j ∧ ¬s3_dem)

▯
   s1_enq := true                       if (¬s1_j ∧ ¬s1_excl)
▯
   s2_enq := s1_enq                     if (¬s2_j ∧ (¬s2_excl ∧ s1_enq))
▯
   s3_enq := s2_enq                     if (¬s3_j ∧ (¬s3_excl ∧ s2_enq))
▯
   s1_j,s1_enq := true,false            if (s1_enq ∧ (s2_enq ∧ s3_enq))

end {jeton}
```

Le problème du parking

Le problème du parking [AND 83] modélise l'allocation de ressources critiques à des processus concurrents. La capacité C du parking représente le nombre de ressources disponibles. L'arrivée d'une voiture traduit une demande de ressource. L'entrée d'une voiture dans le parking correspond à l'allocation effective d'une ressource et la sortie à la libération de cette ressource. La spécification du problème repose sur des compteurs d'événements représentant respectivement le nombre d'arrivées A, d'entrées E et de sorties S.

La propriété caractéristique invariante consiste à garantir que le nombre de voitures dans le parking reste inférieur ou égal à la capacité du parking, soit en terme d'événements :

$$E - S + D = C$$

où D désigne le nombre de places disponibles et E, S et D sont positifs ou nuls.

Une mise en œuvre répartie peut être modélisée par un parking à plusieurs points d'accès. Chaque point d'accès possède alors ses propres compteurs d'événements et de places disponibles a_i, e_i, s_i et d_i avec les relations de représentation :

$$A = \sum_i a_i, \quad E = \sum_i e_i, \quad S = \sum_i s_i, \quad D = \sum_i d_i$$

1. L'opérateur ‖ est utilisé pour faciliter la lecture de l'algorithme. Il permet de distinguer deux aspects de l'instruction indépendants au sein de l'algorithme, mais dont l'exécution doit rester synchrone.

Dans le programme suivant, on suppose décrit un parking à trois points d'accès. Les instructions `UNITY` traduisent l'évolution des compteurs d'événements et l'échange de places disponibles (d_i) entre les points d'accès selon une structure d'anneau.

```
Program Parking

declare
    C : INT;
    a1,a2,a3,e1,e2,e3,s1,s2,s3,d1,d2,d3 : INT;
    ka,ke,ks : INT

always
    INT S = ((s1 + s2) + s3),
    INT E = ((e1 + e2) + e3),
    INT A = ((a1 + a2) + a3),
    INT D = ((d1 + d2) + d3)
```

```
Specifications
    Inv_gl_Impl  invariant (((E − S) + D) = C),
    d123_ge_0    invariant (((d1 ≥ 0) ∧ (d2 ≥ 0)) ∧ (d3 ≥ 0)),
    a123_ge_0    invariant (((a1 ≥ 0) ∧ (a2 ≥ 0)) ∧ (a3 ≥ 0)),
    e123_ge_0    invariant (((e1 ≥ 0) ∧ (e2 ≥ 0)) ∧ (e3 ≥ 0)),
    s123_ge_0    invariant (((s1 ≥ 0) ∧ (s2 ≥ 0)) ∧ (s3 ≥ 0)),
    a1_Croit     (a1 = ka) unless (a1 > ka),
    e1_Croit     (e1 = ke) unless (e1 > ke),
    s1_Croit     (s1 = ks) unless (s1 > ks)

with
    (((((E − S) + D) = C) ∧ (D ≥ 0)) ⇒ ((E − S) ≤ C))
```

```
implementation

initially
    a1,a2,a3,e1,e2,e3,s1,s2,s3 = 0,0,0,0,0,0,0,0,0;
    d1,d2,d3,C = 2,2,1,5

assign
    a1 := (a1 + 1)
[]
    e1,d1 := (e1 + 1),(d1 − 1)          if ((e1 < a1) ∧ (d1 > 0))
[]
    s1,d1 := (s1 + 1),(d1 + 1)          if (s1 < e1)

[]
    a2 := (a2 + 1)
[]
    e2,d2 := (e2 + 1),(d2 − 1)          if ((e2 < a2) ∧ (d2 > 0))
[]
    s2,d2 := (s2 + 1),(d2 + 1)          if (s2 < e2)

[]
    a3 := (a3 + 1)
[]
    s3,d3 := (s3 + 1),(d3 + 1)          if (s3 < e3)
[]
    e3,d3 := (e3 + 1),(d3 − 1)          if ((e3 < a3) ∧ (d3 > 0))

[]
    d1,d2 := (d1 − 1),(d2 + 1)          if (d1 > 0)
[]
    d2,d3 := (d2 − 1),(d3 + 1)          if (d2 > 0)
[]
    d3,d1 := (d3 − 1),(d1 + 1)          if (d3 > 0)
end {Parking}
```

3.3. *Performances*

Le tableau suivant présente quelques statistiques recueillies lors de l'utilisation de l'environnement de preuve sur les exemples précédents. Pour chaque exemple sont indiqués le démonstrateur utilisé, le nombre d'instructions dans le programme, le nombre de propriétés dans la spécification, le nombre de contraintes logiques engendrées et soumises au démonstrateur et la durée totale de la preuve de l'ensemble de ces contraintes élémentaires.

D'une manière générale, les performances en temps du calculateur `Omega` sont meilleures que celles de `Otter`. Malheureusement, les calculs effectués par `Omega` sont extrêmement gourmands en mémoire lorsque la taille des formules devient trop grande. C'est pourquoi certaines propriétés de l'algorithme du jeton (cf **3.2**) n'ont pu être prouvées avec ce démonstrateur et `Otter` a donc été utilisé.

Ces résultats ont été obtenus sur une station de travail ***Sun*** **SPARCstation 5** équipée de 64 MB de mémoire.

Algorithme	Prouveur	Nombre d' instructions	Nombre de propriétés	Nombre de preuves	Temps de preuve
Peterson	Omega	13	12	153	1mn 10s
Jeton	Otter	20	7	726	13mn
Parking	Omega	13	9	102	20s

4. Exploitation d'un démonstrateur automatique de théorèmes

Pour vérifier la cohérence d'un programme par rapport à sa spécification, nous sommes amenés à exploiter un démonstrateur automatique de théorèmes, en l'occurrence `Otter` (cf **4.1**). La composante éditeur de l'environnement engendre en effet l'ensemble des prédicats à valider pour s'assurer que le programme vérifie sa spécification. Cette traduction, effectuée par le biais de grammaires attribuées, est fondée sur le calcul de *weakest precondition* (wp) [DIJ 76].

4.1. *Présentation du démonstrateur Otter*

Les techniques utilisées en démonstration automatique sont diverses et ont donné naissance à des types variés d'outils. Les systèmes de preuve par réfutation fondées sur la résolution et les systèmes de réécriture constituent deux branches parmi les plus importantes. Différents outils ont été étudiés en vue de leur utilisation dans le cadre du projet [KAP 87, GAR 91, AST 92]. Le démonstrateur actuellement utilisé est `Otter` (*Organized Techniques for Theorem-proving and Effective Research*) [MCC 90, MCC 91] qui est un système de preuve de prédicats du premier ordre par réfutation / résolution. Ses atouts sont sa simplicité d'utilisation (ce qui facilite l'interfaçage avec l'éditeur), sa rapidité d'exécution et son aptitude à travailler de manière *entièrement* automatique. En effet, les formules que nous avons à traiter sont relativement simples mais très longues et très nombreuses. Il était donc nécessaire de disposer d'un outil automatique et rapide.

L'exemple suivant, volontairement choisi hors du domaine pour des raisons de clarté et de concision, se traduit directement dans la syntaxe de Otter par trois axiomes correspondant aux hypothèses, une règle de réécriture [2] et la négation de la conclusion qui devient le but à réfuter.

Exemple : *Mon père est le grand-père de mon fils*

Hypothèses : $\forall x\ \forall y\ \forall z \quad pere(x,y)\ \wedge\ pere(y,z)\ \Rightarrow\ grand\_pere(x,z)$

$$fils(Paul, Jacques)$$

$$pere(Paul, Pierre)$$

Règle orientée : $fils(x,y)\ \rightarrow\ pere(y,x)$

Conclusion : $grand\_pere(Jacques, Pierre)$?

A partir des hypothèses et de la conclusion précédentes, Otter produit la preuve suivante :

```
----> UNIT CONFLICT at   0.06 sec ----> 9 [binary,8,4] .

Level of proof is 2, length is 3.

---------------- PROOF ----------------

1 [] (fils(x,y) = pere(y,x)).
2 [] -pere(x,y) | -pere(y,z) | grand_pere(x,z).
3 [demod,1] pere(Jacques,Paul).
4 [] pere(Paul,Pierre).
6 [] -grand_pere(Jacques,Pierre).
7 [binary,6,2] -pere(Jacques,x) | -pere(x,Pierre).
8 [binary,7,3] -pere(Paul,Pierre).
9 [binary,8,4] .

------------ end of proof -------------
```

4.2. *Principe de validation des spécifications*

Le langage de spécification associé à la logique UNITY utilise le principe d'assertions logiques (*clauses de Hoare*) confrontées aux instructions du programme.

Cette approche est très utilisée et diverses techniques ont été introduites pour la mettre en œuvre. En particulier, la méthode la plus classique consiste à considérer les instructions d'un programme comme des manipulateurs de prédicats (*backward predicate transformers*) [DIJ 76]. Elle utilise la notion de wp définie de la manière suivante : pour un prédicat Q et une instruction s, $wp\,(s,Q)$ représente la contrainte *la plus faible* à vérifier *avant* l'exécution de s pour être assuré de la validité de Q *après* l'exécution de s. Un prédicat P ne sera donc une précondition valable à l'établissement de Q que s'il constitue une contrainte *plus forte* que la wp, c'est-à-dire s'il l'implique. On a donc :

$$\{P\}\ s\ \{Q\} \quad ssi \quad P \Rightarrow wp\,(s,Q)$$

2. Nous utilisons ici la possibilité offerte par Otter d'appliquer des règles de réécriture aussi bien à des atomes qu'à des termes.

L'avantage considérable de cette approche est lié à la calculabilité de la wp. En effet, la fonction précédente est parfaitement définie et facilement programmable pour toutes les affectations gardées du langage UNITY. Il est ainsi possible de calculer automatiquement $wp\,(s, Q)$ pour toute instruction s, et donc de ramener un problème de spécification $\{P\}\ s\ \{Q\}$ à celui de la validité de la formule $P \Rightarrow wp\,(s, Q)$. Le choix de cette approche déductive permet en outre la preuve de programmes dont l'espace d'états est infini, ce qui n'est pas le cas dans une approche par test exhaustif sur l'espace d'états (*model checking*). A titre d'exemple, le programme du parking exposé précédemment (cf **3.2**) utilise des compteurs croissants non bornés.

Cas des instructions UNITY

Considéré comme langage de programmation, UNITY n'utilise qu'un unique type d'instruction : l'affectation multiple. La seule wp à calculer est donc celle qui définit la sémantique de l'affectation. Une formule Q sera valide après l'exécution d'une affectation si la formule dont elle est issue par l'affectation était valide avant l'exécution de l'instruction. La wp de l'affectation se calcule donc facilement par substitution :

$$wp\,(v_1, \cdots, v_n \ := e_1, \cdots, e_n \ , \ Q) \quad \equiv \quad Q|^{v_1, \cdots, v_n}_{e_1, \cdots, e_n}$$

Ce principe de calcul s'étend au cas où les affectations sont gardées. Une affectation n fois gardée peut être vue comme n affectations différentes selon la valeur des gardes. La wp doit donc être suffisante quelle que soit l'affectation réellement exécutée. Par contre, la validité d'une garde peut être admise dans la considération d'une affectation particulière et venir renforcer la connaissance liée à la précondition. De plus, à ces n instructions vient s'ajouter l'instruction nop (de wp identité) pour le cas où aucune garde n'est valide (aucune des n affectations n'est exécutée).

Pour une instruction s de la forme :

$$\begin{array}{lcll} v_1, v_2, \cdots, v_n & := & e^1_1, e^1_2, \cdots, e^1_n & if \ \ g_1 \\ & \sim & e^2_1, e^2_2, \cdots, e^2_n & if \ \ g_2 \\ & \cdots & \cdots & if \ \ \cdots \\ & \sim & e^p_1, e^p_2, \cdots, e^p_n & if \ \ g_p \end{array}$$

on a donc :

$$\{P\}\ s\ \{Q\} \ \ ssi \ \ \bigwedge_{i=1}^{p} (P \wedge g_i \Rightarrow wp\,(\alpha_i, Q)) \ \bigwedge \ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

où α_i dénote l'affectation $v_1, v_2, \cdots, v_n \ := \ e^i_1, e^i_2, \cdots, e^i_n$ et où chaque $wp\,(\alpha_i, Q)$ est calculée par substitution sur Q.

Cas de la logique UNITY

Dans le cas de la formule temporelle X $\mathbf{Unless}$ Y, les prédicats P et Q précédents valent respectivement $X \wedge \neg Y$ et $X \vee Y$. Comme $X \wedge \neg Y \Rightarrow X \vee Y$ pour tout X et tout Y, le cas d'une affectation pour laquelle aucune garde n'est vraie n'est pas à considérer. En effet, la contrainte est nécessairement vérifiée par l'instruction nop.

Le résultat reste valable pour les contraintes **Stable**, **Constant** et **Invariant** qui ne sont que des cas particuliers de **Unless**.

4.3. *Optimisation des preuves*

Un certain nombre de preuves peuvent être optimisées à partir de la constatation suivante : lorsqu'un prédicat P ne référence pas les variables d'une instruction s, on a $wp\,(s, P) \;=\; P$. Ce cas de figure, relativement fréquent en pratique, peut être effectivement exploité.

Cas des invariants conjonctifs : pour montrer qu'une propriété P_0 est *toujours vraie* lors de l'exécution d'un programme, on peut être amené à montrer que $\bigwedge_{i \geq 0} P_i$ est un invariant de ce programme. Vis-à-vis d'une instruction s, la formule à démontrer est alors :

$$\begin{array}{ll} & \bigwedge P_i \Rightarrow wp\,(s, \bigwedge P_i) \\ \equiv & \bigwedge P_i \Rightarrow \bigwedge wp\,(s, P_i) \\ \equiv & \forall k \;:\; \bigwedge P_i \Rightarrow wp\,(s, P_k) \end{array}$$

Or, dans ce cas d'invariants conjonctifs de taille souvent très importante, l'instruction s ne modifie qu'un sous-ensemble limité des variables des prédicats P_k. Par conséquent, il apparaît un nombre non négligeable de tautologies du type $\bigwedge P_i \Rightarrow P_k$.

Cas des invariants implicatifs : une autre manière d'établir qu'une propriété Q est *toujours vraie* sur un programme est de montrer que $P \Rightarrow Q$ et P sont des invariants de ce programme. Pour une instruction s donnée, il faut alors prouver

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

c'est-à-dire

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

Si l'affectation s considérée ne concerne que des variables à droite de l'implication, la formule se simplifie en :

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

De la même manière, si l'affectation s ne modifie que des variables apparaissant à gauche de l'implication, la formule à vérifier est de la forme :

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

De telles simplifications se généralisent au cas où l'implication considérée fait partie d'une conjonction. On utilise alors les équivalences suivantes :

$$((P \Rightarrow Q) \wedge R) \Rightarrow (P \Rightarrow wp\,(s, Q)) \quad \equiv \quad (P \wedge R) \Rightarrow (Q \Rightarrow wp\,(s, Q))$$

et

$$((P \Rightarrow Q) \wedge R) \Rightarrow (wp\,(s, P) \Rightarrow Q) \quad \equiv \quad (\neg Q \wedge R) \Rightarrow (wp\,(s, P) \Rightarrow P)$$

Exemple : Dans l'exemple d'exclusion mutuelle avec tolérance aux fautes (Cf **3.2**), considérons l'invariant :

$$ENQ\_JETON\_2 \;=\; \bigwedge_{i=1}^{3} (s_i\_j \Rightarrow \neg s_i\_enq)$$

et la seizième instruction :

$$s_{16} \;=\; s_1\_enq \;:=\; true\;\; if\; (\neg s_1\_j \wedge \neg s_1\_excl)$$

Le prédicat à valider est alors

$$ENQ\_JETON\_2 \wedge (\neg s_1\_j \wedge \neg s_1\_excl) \Rightarrow wp\,(s_{16}, ENQ\_JETON\_2)$$

Comme les deux dernières conjonctions de ENQ_JETON_2 ne référencent pas la variable s_1_enq affectée par l'instruction s_{16}, la formule à vérifier devient

$$ENQ\_JETON\_2 \wedge (\neg s_1\_j \wedge \neg s_1\_excl) \Rightarrow wp\,(s_{16}, s_1\_j \Rightarrow \neg s_1\_enq)$$

Comme de plus s_1_enq n'apparaît qu'à droite de l'implication, une seconde simplification est possible et la contrainte devient

$$s_1\_j \wedge \bigwedge_{i=2}^{3} (s_i\_j \Rightarrow \neg s_i\_enq) \wedge (\neg s_1\_j \wedge \neg s_1\_excl) \Rightarrow (\neg s_1\_enq \Rightarrow wp\,(s_{16}, \neg s_1\_enq))$$

Cette dernière formule est bien valide puisque la conjonction $s_1\_j \wedge \neg s_1\_j$ apparaît en partie gauche de l'implication.

4.4. *Implantation*

Nous précisons quelques points d'implantation liés à l'exploitation du démonstrateur `Otter`. Nous abordons successivement le problème de l'arithmétique, du prétraitement et de la généralisation des formules engendrées.

Problème de l'arithmétique : la description d'algorithmes en `UNITY` fait appel fréquemment à la notion de compteurs, représentés par des variables entières. Ces compteurs, mis à jour par les affectations, sont à la base de la plupart des gardes. Typiquement, une instruction est exécutable si la valeur d'un compteur se situe dans un intervalle donné. L'affectation

$$a := a + 1 \quad if \quad a < b$$

vérifie la propriété

$$\textbf{Stable} \quad a \leq b$$

La vérification de cette contrainte se fait par la preuve de la formule

$$a < b \quad (\wedge \quad a \leq b) \quad \Rightarrow \quad a + 1 \leq b$$

Ceci suppose donc un minimum de connaissances arithmétiques de la part du démonstrateur utilisé. Malheureusement, `Otter` ne possède aucune notion intrinsèque d'addition ou de relations d'ordre. Il ne manipule ces fonctions qu'au sein de constantes numériques et ne dispose d'aucune arithmétique *symbolique* interne.

Il a donc été nécessaire de lui fournir une base d'axiomes lui permettant d'aborder les problèmes liés à la gestion des compteurs. Celle-ci se limite aux opérations d'addition et de soustraction, et aux relations d'ordre larges et strictes sur les nombres entiers, ce qui constitue une partie décidable de l'arithmétique. Ceci permet donc de ne pas nuire à la complétude théorique d'`Otter`.

Malheureusement, l'introduction de ces axiomes multiplie considérablement la complexité des démonstrations réalisées par `Otter`. On assiste en effet à une explosion combinatoire du nombre de clauses engendrées par le démonstrateur dont la complétude n'est assurée que par un parcours en largeur de l'arbre de recherche. La multiplication des clauses rend ce parcours très coûteux et limite la profondeur des démonstrations accessibles en un temps raisonnable.

Prise en compte de particularités de programmation : certaines simplifications de formules sont étroitement liées à des choix ou à des nécessités de programmation, en particulier au niveau du calcul des wp. Par exemple, la réduction d'expressions numériques (plus efficace au niveau du `Grammatech Synthesizer Generator` qu'au niveau d'`Otter`), la simplification de constantes booléennes (introduites notamment comme valeurs de fin de récursivité) ou encore la suppression de termes redondants (introduits à différents niveaux du calcul). Ces simplifications, très sommaires, ne sont pas liées à la sémantique des spécifications ou des programmes `UNITY`.

Généralisation des formules : pour des raisons pratiques, une durée maximum est imposée à toute tentative de démonstration. Or le démonstrateur peut s'avérer capable de valider rapidement une formule à partir de certaines hypothèses alors qu'un temps beaucoup plus long lui est nécessaire avec des hypothèses plus fortes (donc une formule plus longue). Ce phénomène peut donc conduire à un échec apparent d'une validation à cause de l'expiration du délai.

Pour cette raison, si (et seulement si) la démonstration de $P \wedge Q \Rightarrow R$ a échoué, on peut essayer d'établir $P \Rightarrow R$ ou $Q \Rightarrow R$. En particulier, si une formule Q doit être prouvée stable pour une instruction s gardée par G, on est amené à prouver

$$Q \wedge G \Rightarrow wp(s, Q)$$

Si la garde n'apporte aucun renseignement utile vis-à-vis de la propriété étudiée, il est très intéressant de remplacer le problème par $Q \Rightarrow wp(Q)$ ce qui permet de tenir compte ensuite de la structure particulière de Q (disjonction, conjonction, implication).

Par exemple, dans le problème du parking du paragraphe **3.2**, la vérification de la propriété **Invariant** $d1 \geq 0 \wedge d2 \geq 0 \wedge d3 \geq 0$ au niveau de l'instruction $s1, d1 := s1+1, d1+1 \;\; if \;\; s1 < e1$ ne nécessite pas de considérer la garde $s1 < e1$.

De la même manière, on peut être amené (toujours en cas d'échec de la démonstration directe) à remplacer $P \vee Q$ par P (ou Q), ou encore $P \Rightarrow Q \vee R$ par $P \Rightarrow Q$ (ou $P \Rightarrow R$).

L'inconvénient majeur de cette généralisation des formules est qu'elle amène, dans le cas où une formule *n'est pas* valide, à considérer une multitude de problèmes dérivés qui nécessairement ne sont pas valides.

5. Exploitation d'un calculateur de formules *Presburger*

5.1. *Présentation du calculateur Omega*

Devant la difficulté de mettre en œuvre, au sein d'un démonstrateur automatique de théorèmes classique tel qu'`Otter`, les simplifications arithmétiques nécessaires à la vérification des formules, un autre type d'outil a été utilisé. Il s'agit du calculateur `Omega` [PUG 92, KEL 94]. `Omega` n'est pas à proprement parler un démonstrateur automatique de théorèmes, mais se limite à la manipulation d'ensembles et de relations entre n-uplets d'entiers. Ces ensembles et relations sont définis par des formules en arithmétique *Presburger*, une classe de formules construites à partir de contraintes linéaires sur des variables entières.

Une telle formule est donc de la forme :

$$\begin{array}{ll} & \phi \wedge \psi \\ ou & \phi \vee \psi \\ ou & \neg \phi \\ ou & \exists\, v_1, v_2, \cdots, v_n : \phi \\ ou & \forall\, v_1, v_2, \cdots, v_n : \phi \\ ou & C \end{array}$$

où ϕ, ψ sont des formules Presburger et C est une contrainte de la forme $e_1, e_2, \cdots, e_n \;\; op \;\; e'_1, e'_2, \cdots, e'_n$ où *op* est un opérateur parmi $=, \neq, <, >, \leq, \geq$. Les e'_i sont des expressions linéaires des variables.

Un ensemble est donc défini de la manière suivante :

$$\{[u_1, u_2, \cdots, u_n] : \phi\};$$

où ϕ est une formule Presburger, et une relation est de la forme :

$$\{[u_1, u_2, \cdots, u_n] \rightarrow [v_1, v_2, \cdots, v_n] : \phi\};$$

`Omega` propose alors une série d'opérations ensemblistes et relationnelles via une interface textuelle.

Exemple :

$$\begin{array}{l} pairs := \{[x] : \exists\ k : x = 2k\}; \\ impairs := \{[x]\}\ -\ pairs; \\ double := \{[x] \rightarrow [2x]\}; \\ double(impairs)\ \ subset\ \ pairs; \end{array}$$

Les trois premières requêtes à `Omega` mettent à jour trois variables. La quatrième exprime que le double d'un nombre impair est un nombre pair (`Omega` calcule la réponse *TRUE*).

5.2. *Principe de validation des spécifications*

Une seconde interface a donc été ajoutée à l'éditeur syntaxique pour permettre une vérification des spécifications `UNITY` par le calculateur `Omega`. Le principe est de traduire les préconditions et postconditions par des ensembles d'états et l'affectation gardée par une relation entre n-uplets.

La vérification de $\{P\}\ s\ \{Q\}$ va consister à évaluer la *plus forte postcondition* (sp) de P vis-à-vis de s et à vérifier que $sp\,(s, P) \Rightarrow Q$.

Pour une instruction s de la forme :

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if\ \ g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if\ \ g_2 \\ & \cdots & \cdots & if\ \ \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if\ \ g_p \end{array}$$

on a donc :

$$\{P\}\ s\ \{Q\}\ \ ssi\ \ \bigwedge_{i=1}^{p} (sp\,(\alpha_i, P \wedge g_i) \Rightarrow Q)\ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

où α_i dénote l'affectation $v_1, v_2, \cdots, v_n\ := \ e_1^i, e_2^i, \cdots, e_n^i$.

5.3. *Implantation*

Un état est défini par le n-uplet des valeurs courantes des variables du programme. Au lieu de manipuler directement les prédicats eux-mêmes, on considère des ensembles d'états vérifiant ces prédicats. En ce qui concerne les affectations, elles sont considérées comme des fonctions définies sur l'ensemble des états. L'implication $sp\,(s, P) \Rightarrow Q$ est donc vue comme l'inclusion $\mathcal{S}\,(\mathcal{P}) \subset \mathcal{Q}$ où $\mathcal{S}$ est la fonction associée à l'instruction s et $\mathcal{P}$ et $\mathcal{Q}$ sont les ensembles d'états associés aux prédicats P et Q. Cette approche évite le calcul explicite de la sp et permet de ne calculer ni sp ni wp au niveau de l'éditeur.

La vérification d'une spécification de la forme $\{P\}\, s\, \{Q\}$ où s est l'instruction

$$\begin{array}{lcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if \;\; g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & if \;\; g_2 \\ & \cdots & \cdots & if \;\; \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if \;\; g_p \end{array}$$

engendre donc les requêtes suivantes à `Omega` :

$$\begin{array}{ll} pre := & \{[v_1, v_2, \cdots, v_n] : P\}; \\ post := & \{[v_1, v_2, \cdots, v_n] : Q\}; \\ s := & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^1, e_2^1, \cdots, e_n^1] : g_1\} \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^2, e_2^2, \cdots, e_n^2] : g_2\} \\ \cdots & \cdots \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^p, e_2^p, \cdots, e_n^p] : g_p\} \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [v_1, v_2, \cdots, v_n] : \neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_p\}; \\ s(pre) & subset \;\; post; \end{array}$$

On remarquera que, la notion de variable booléenne n'existant pas dans les formules *Presburger*, le traitement des affectations booléennes est un peu particulier. Une affectation de la forme $p := q \;\; if \;\; r$ sera traduite par :

$$\begin{array}{ll} \{[p] \rightarrow [1] : q \wedge r\} & union \\ \{[p] \rightarrow [0] : \neg q \wedge r\} & union \\ \{[p] \rightarrow [p] : \neg r\} & \end{array}$$

6. Travaux connexes

Nous pouvons distinguer deux approches différentes dans la preuve de programmes. Bien qu'il s'agisse toujours de démontrer des propriétés d'un programme, dans un premier cas on s'intéresse à la conception d'une preuve correcte alors que dans l'autre on cherche à éviter le développement explicite de la preuve.

La première approche a pour principe de vérifier à l'aide d'un démonstrateur une preuve décrite formellement et entièrement par le développeur. Les systèmes **HOL** [GOR 93], **Deva** [CAZ 89] et **PVS** [SHA 93b] sont utilisés pour ce type de validation de preuves. Dans ce type d'approche, il n'y a pas confrontation d'un programme par rapport à une spécification *au niveau de l'outil*. Seule la preuve résultant d'une telle analyse est considérée. La théorie `UNITY` a été décrite notamment en **HOL** [AND 92] et en logique de **Boyer-Moore** [GOL 92].

La deuxième approche a pour principe d'automatiser le processus même de développement de la preuve, les détails de cette preuve restant cachés à l'utilisateur. Le système *TLP* [ENG 94] illustre cette approche en utilisant la logique temporelle d'actions [LAM 94]. Nous examinons plus en détail, deux autres projets nous paraissant parmi les plus proches de nos travaux. L'un, appelé *UNITY Verifier* [KAL 94b], utilise lui-aussi le formalisme `UNITY`, mais procède par *model checking* pour réaliser des preuves. L'autre, appelé STeP [MAN 94], réalise la validation d'un programme par rapport à une spécification en logique temporelle linéaire.

6.1. *La vérification par model checking*

Une voie traditionnelle de la preuve automatique de programmes est celle du *model checking*. Elle consiste à vérifier qu'un modèle (un programme) est valide vis-à-vis d'une formule logique (une spécification), la plupart du temps en examinant exhaustivement toutes les transitions possibles du programme. Parce qu'elle se limite à des programmes dont le nombre d'états est fini, cette méthode permet de vérifier de manière automatique tous les types de propriétés d'un programme, en particulier les propriétés de type **leads-to** et les propriétés *toujours vraies* (cf **2.3**).

Cette approche, appliquée au formalisme `UNITY`, est à la base d'un outil interactif : le *Unity Verifier* (UV) [KAL 94b, KAL 94a]. A travers une interface `X Window System`, cet outil permet de vérifier la cohérence entre une spécification de la logique `UNITY` et un programme décrit dans une version réduite du langage (absence de tableaux, de quantification des instructions, de la clause **always**).

Outre que son utilisation du *plus fort invariant* le conduit à manipuler des programmes et des spécifications non facilement composables [COL 94b], la principale limitation de cet outil provient de l'impossibilité de manipuler des programmes à états infinis et notamment ceux qui font usage de variables numériques entières.

6.2. *L'environnement STeP*

Le projet STeP (*The Standford Temporal Prover*) est un exemple d'environnement d'assistance à la conception et preuve de systèmes réactifs modélisés sous la forme de systèmes de transitions équitables. L'approche adoptée est, comme dans notre cas, le développement simultané d'une spécification en logique temporelle linéaire et d'un programme dans un langage parallèle (SPL) décrivant un système de transitions et ses propriétés. Les bases théoriques sont donc identiques à celles du modèle `UNITY`. Cependant, les différences portent sur plusieurs points :

— L'environnement utilise une approche mixte sous deux aspect. D'une part, au niveau de l'utilisation, la preuve des propriétés peut être entièrement automatique ou guidée par l'usager et d'autre part, au niveau méthode de preuve, une propriété temporelle peut être vérifiée soit par *model checking*, soit par appel à un démonstrateur de théorème.

— Le langage de description "externe" d'un système de transitions est plus évolué et déterministe que le langage `UNITY`. On dispose des structures de contrôle classiques et de primitives de synchronisation (sémaphores par exemple). De ce point de vue, le langage `UNITY` apparaît comme plus "spartiate" mais suffisant pour exprimer les actions parallèles ou exclusives avec un maximum de parallélisme et de non-déterminisme.

— Les opérateurs temporels ne sont pas identiques. En particulier, les propriétés de vivacité s'expriment à l'aide d'un connecteur de base $next$ noté $\bigcirc$. Le prédicat temporel $p \bigcirc q$ est valide si et seulement si toutes les transitions possibles depuis un état vérifiant p conduisent à un état vérifiant q. L'opérateur **ensures** de `UNITY` exprime une vivacité plus faible comme nous l'avons vu.

Notons qu'un point fort de l'environnement est la synthèse automatique d'invariants.

7. Conclusion

Dans cette conclusion, nous évaluons l'état actuel de l'environnement ainsi que ses perspectives de développement.

Au regard de nos expérimentations, l'utilisation de plusieurs outils de démonstration disposant chacun de leurs atouts semble une solution plus réaliste que la recherche et la paramétrisation d'un outil universel. Par exemple, dans sa configuration actuelle, notre environnement permet à l'utilisateur de choisir `Otter` pour la preuve de formules purement logiques de très grande taille et de réserver l'utilisation de `Omega` pour les formules faisant intervenir des problèmes arithmétiques. Cette approche "multi-outils" est d'ailleurs aussi une caractéristique du projet STeP. C'est pourquoi nous envisageons l'utilisation d'autres démonstrateurs automatiques, en particulier le *Nqthm* de Boyer et Moore [BOY 88].

La génération automatique d'invariants, telle qu'elle apparaît dans les projets *UNITY Verifier* et STeP, constitue un apport important dans l'assistance à la preuve. La non calculabilité du *plus fort invariant* dans le cas d'espaces d'états infinis pose néanmoins un problème dans ce type d'approche. En ce qui nous concerne, la technique des raffinements peut permettre une recherche incrémentale d'invariants.

Les programmes validés par notre environnement, bien que de taille relativement modeste, nécessitent chacun plusieurs centaines de preuves élémentaires. La preuve de programmes de grande taille reste donc difficile. Face à cet obstacle, deux axes de recherche sont étudiés :

— la modularité : plusieurs modules peuvent être prouvés indépendamment et intégrés ultérieurement. Il faut pour cela disposer d'opérations de composition sémantiquement validées. C'est le cas du formalisme `UNITY` qui permet donc d'envisager cette voie de recherche [COL 94b, COL 94a].

— la définition d'opérations de raffinements génériques elles aussi sémantiquement validées.

Dans les deux cas, la tâche de l'environnement consistera à vérifier les conditions d'application de ces transformations. Ces deux directions sont actuellement approfondies. En particulier, des raffinements génériques pour la répartition d'un programme ont été spécifiés [FIL 93].

Remerciements

Nous tenons à remercier les lecteurs d'une première version de cet article pour leurs critiques, leurs conseils et l'intérêt qu'ils ont porté à ces travaux, sans oublier les membres de l'équipe, Mamoun Filali, Philippe Mauran, Pierre Maurice et Philippe Quéinnec.

8. Bibliographie

[AND 83] F. ANDRÉ, D. HERMAN, et J.-P. VERJUS. *Synchronisation de programmes parallèles*. Dunod Informatique, 1983.

[AND 92] F. ANDERSEN. « *A Theorem Prover for UNITY in Higher Order Logic* ». PhD thesis, Technical University of Denmark, 1992.

[AST 92] O. ASTRACHAN. « Exploring Model Elimination Theorem Proving ». Technical Report CS-1992-22, Duke University, Durham, ,NC USA, December 1992.

[BAC 88] R.-J.R. BACK. « A Calculus of Refinment for Program Derivation ». *Acta Informatica*, (25):593–624, 1988.

[BAC 91] R.-J.R. BACK and K. SERE. « Stepwise Refinment of Action Systems ». *Structured Programming*, pages 17–30, 1991.

[BOY 88] R. S. BOYER and J. S. MOORE. *A Computational Logic Handbook*. Academic Press, 1988.

[BRO 94] N. BROWN. « Vérification et Mise en œuvre Distribuée des Programmes UNITY ». In L. BOUGÉ, éditeur, *Actes des sixièmes Rencontres francophones du Parallélisme*, pages 259–262. ENS Lyon, France, juin 1994.

[CAZ 89] J. CAZIN, R. JACQUART, M. LEMOINE, P. MICHEL, and P. MAURICE. *Method driven programming*. G.X. Ritter Editor, Information Processing 89. Elsevier Science Publishers B.V. (North-Holland), 1989.

[CHA 88] K.M. CHANDY and J. MISRA. *Parallel Program Design : A Foundation*. Addison-Wesley, 1988.

[CHA 95] M. CHARPENTIER, A. El HADRI, and G. PADIOU. « A UNITY-based Algorithm Design Assistant ». In BRICS Notes Series NS-95-2, editor, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 131–145, May 1995.

[COL 94a] P. COLETTE. « An explanatory presentation of composition rules for assumption-commitment specifications ». *Information Processing Letters*, 50:31–35, 1994.

[COL 94b] P. COLETTE. « *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to UNITY* ». Thèse de docteur en sciences appliquées, Faculté des Sciences Appliquées, Université Catholique de Louvain, juin 1994.

[DHA 94] Ph. DHAUSSY. « *Un modèle de programmation des contrôleurs globaux* ». thèse de doctorat, Université de Rennes 1, novembre 1994.

[DIJ 76] E. W. DIJKSTRA. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[ENG 94] U. ENGBERG. « *TLP Manual* ». Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, DENMARK, release 2.5a édition, May 1994.

[FIL 93] M. FILALI, Ph. MAURAN, et G. PADIOU. « Raffiner pour répartir ». Rapport Technique 93-29/R, Institut de Recherche en Informatique de Toulouse, France, novembre 1993.

[FRO 94] E. FROMENTIN and M. RAYNAL. « Proving Liveness Properties of Concurrent Programs ». *ACM Operating Systems Review*, April 1994.

[GAR 91] S. J. GARLAND and J. V. GUTTAG. « A Guide to LP, The Larch Prover ». Technical Report 82, Systems Research Center, Digital Equipment Corporation, December 1991.

[GOL 92] D. M. GOLDSCHLAG. « *Mechanically Verifying Concurrent Programs* ». PhD thesis, University of Texas at Austin, May 1992.

[GOR 93] M.J.C. GORDON and T.F. MELHAM. *Introduction to HOL : A Theorem Proving Environnement for Higher Order Logic*. Cambridge University Press, 1993.

[HER 93] P. HERRMANN and H. KRUMM. « Report on Analysis and Verification Techniques ». Research Report 485/1993, Universität Dortmund, Computer Networks and Distributed Systems Research Group, D-44221 Dortmund, Germany, 1993.

[HOA 69] C.A.R. HOARE. « An axiomatic basis for computer programming ». *Communications of the ACM*, 12:576–580,583, 1969.

[KAL 94a] M. KALTENBACH. « *Model Checking for UNITY : The UV System* ». Department of Computer Sciences, The University of Texas at Austin, revision 1.10 édition, May 1994.

[KAL 94b] M. KALTENBACH. « *The UV System. User Interface Manual.* ». The University of Texas at Austin, revision 1.13 édition, December 1994.

[KAP 87] D. KAPUR and H. ZHANG. « *RRL : Rewriting Rule Laboratory User's Manual* ». Department of Computer Science, State University of New York at Albany, The University of Iowa, June 1987.

[KEL 94] W. KELLY, V. MASLOV, W. PUGH, E. ROSSER, T. SHPEISMAN, and D. WONNACOTT. « *The Omega Calculator and Library, version 0.90* ». Dept. of Computer Science and Institute for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742, November 1994.

[LAM 80] L. LAMPORT. « The "Hoare logic" of concurrent programs ». *Acta Informatica*, 14, 1980.

[LAM 94] L. LAMPORT. « The Temporal Logic of Actions ». *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[MAN 94] Z. MANNA and AL.. « SteP : The Stanford Temporal Prover ». Technical Report STAN-CS-TR-94-1518, Department of Computer Science, June 1994.

[MCC 90] W. W. MCCUNE. « OTTER 2.0 Users Guide ». Technical Report ANL-90/9, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, March 1990.

[MCC 91] W. W. MCCUNE. « What's New in OTTER 2.2 ». Technical Report ANL/MCS-TM-153, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, July 1991.

[MIS 90] J. MISRA. « A Family of 2-process Mutual Exclusion Algorithms ». Notes on UNITY, 13, 1990.

[PUG 92] W. PUGH. « The Omega Test : a fast and practical integer programming algorithm for dependence analysis ». *Communications of the ACM*, August 1992.

[REP 89] T. W. REPS and T. TEITELBAUM. *The Synthesizer Generator Reference Manual*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[SAN 91] B. A. SANDERS. « Eliminating the Substitution Axiom from UNITY Logic ». *Formal Aspects of Computing*, 3(2):189–205, April-June 1991.

[SHA 93a] A. U. SHANKAR. « An Introduction to Assertional Reasoning for Concurrent Systems ». *ACM Computing Surveys*, 25(3):225–262, September 1993.

[SHA 93b] N. SHANKAR, S. OWRE, and J.M. RUSHBY. « A tutorial on Specification and Verification Using PVS (Beta Release) ». Technical Report, Computer Science Laboratory, SRI International,, Menlo Park, CA, March 1993.

[SOU 93] J. SOUQUIÈRES. « *Aides au développement de spécifications* ». Thèse de doctorat d'état, Centre de recherche en informatique de Nancy, Université de Nancy I, janvier 1993.