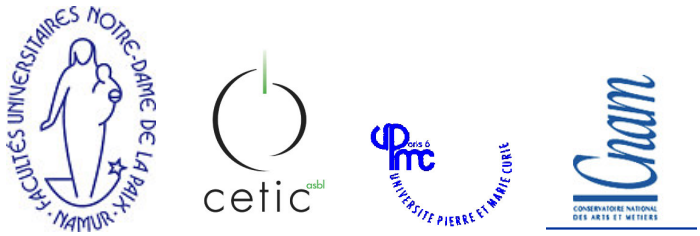


VICS

Vérification d'une Implantation Conformément à sa Spécification
Verification of an Implementation Conforming to its Specification

Rapport de stage déposé en vue de l'obtention du
DESS Développement de Logiciels Sûrs

par Laïka Moussa

Promoteurs : Pierre-Yves Schobbens, Emmanuel Dieul
Tuteur : Mathieu Jaume

le 18 septembre 2003

Je tiens tout d'abord à remercier Emmanuel Dieul, pour sa patience et la qualité de son encadrement. Il a toujours su se montrer disponible, répondre à mes attentes et être à l'écoute de mes questions.

Je remercie également Pierre-Yves Schobbens et les membres de l'équipe LIEL de m'avoir accueillie, ainsi que pour leurs conseils.

Je tiens aussi à remercier Pierre Guisset qui m'a autorisée à effectuer mon stage au CETIC, ainsi qu'Isabelle Dony, qui a accepté de répondre à toutes mes questions.

Je voudrais aussi remercier Mathieu Jaume, mon tuteur.

Je tiens également à remercier Olivier Buffet, Céline Cabré, Nicolas Dohr, Mikaël Riou et les élèves du DESS pour leur soutien, leurs conseils et leurs encouragements.

Enfin, je remercie l'ensemble du CETIC, et tout particulierement l'équipe CRAQ ainsi que Lotfi Guedria, qui m'ont permis de travailler dans une ambiance agréable et enrichissante, et qui m'ont fait partager, pendant ces six mois, la vie du CETIC.

Laïka

Table des matières

1 Présentation de LIEL et du CETIC **9**
- 1.1 L'équipe LIEL . . . 9
  - 1.1.1 Présentation . . . 9
  - 1.1.2 Quelques projets . . . 9
- 1.2 Le Centre d'Excellence en Technologies de l'Information et de la Communication (CETIC) . . . 11
  - 1.2.1 Fondateurs . . . 11
  - 1.2.2 But . . . 11
  - 1.2.3 Domaines de compétences . . . 11
  - 1.2.4 Applications concrètes . . . 12

2 Sujet du stage **13**
- 2.1 Sujet initial . . . 13
- 2.2 Modifications . . . 14

3 Présentation succinte de la méthode B **15**
- 3.1 Introduction à B . . . 15
- 3.2 Les composants B . . . 16
  - 3.2.1 La présentation des composants B . . . 16
  - 3.2.2 La preuve d'une machine . . . 16
  - 3.2.3 La preuve d'un raffinement ou d'une implantation . . . 17
- 3.3 Conclusion . . . 17

4 Le cahier des charges **19**
- 4.1 Rôle du cahier des charges . . . 19
- 4.2 Réalisation du cahier des charges . . . 19
- 4.3 Les difficultés . . . 21
- 4.4 Les décisions . . . 21
  - 4.4.1 Les fonctionnalités de l'outil . . . 21
  - 4.4.2 Le sous-ensemble de B utilisé . . . 22
  - 4.4.3 Le rôle du professeur . . . 23
  - 4.4.4 Autres détails . . . 23

5 Méthode générale **25**
- 5.1 Généralités . . . 25
  - 5.1.1 Notations . . . 25
  - 5.1.2 Objectif . . . 26

5.1.3 Pré-requis : la cohérence de la machine abstraite . . . 27
5.1.4 Restrictions de B pour l'utilisation de Vics . . . 27
5.2 Principe général de fonctionnement . . . 27
5.3 Principe général des vérifications . . . 28
5.3.1 Notations . . . 29
5.3.2 Exemple . . . 29
5.3.3 La correction du code de l'étudiant . . . 30
5.3.4 Le respect des spécifications des machines abstraites . . . 30
5.4 Démarche adoptée . . . 30
5.4.1 Les préconditions des opérations . . . 31
5.4.2 Les post-conditions des opérations . . . 31
5.4.3 Optimisation sur les analyses et vérifications . . . 32

6 Contraintes dues à l'implantation 33
6.1 La génération de nouvelles variables . . . 33
6.2 Notations utilisées . . . 34
6.2.1 Les principaux ensembles . . . 34
6.2.2 Les fonctions . . . 34
6.3 Description des instructions et des expressions . . . 35
6.4 Les contraintes dues aux instructions . . . 36
6.5 Le traitement des expressions . . . 36
6.6 Le traitement des instructions . . . 37
6.6.1 Skip . . . 38
6.6.2 Affectation devient-égal . . . 38
6.6.3 Séquence . . . 39
6.6.4 Les conditionnelles . . . 40
6.7 Exemple . . . 47

7 Contraintes dues à la machine abstraite 49
7.1 La génération de nouvelles variables . . . 49
7.2 Notations utilisées . . . 49
7.3 Description des substitutions . . . 50
7.4 Le traitement des expressions . . . 50
7.5 Les contraintes dues aux substitutions . . . 50
7.5.1 Le traitement des substitutions . . . 50

A Planning 55

B Cahier des charges 57

Introduction

Ce document est un rapport de stage de fin d'études de DESS.
J'ai été, à cette occasion, accueillie par l'équipe LIEL des Facultés Universitaires Notre Dame de la Paix de Namur, du 7 avril au 12 septembre 2003.

En première partie de ce rapport, on trouvera une présentation de l'équipe LIEL, ainsi qu'une présentation du centre où ce stage a été effectué, le CETIC.

Puis, viendront la description du sujet, et une présentation succinte de la méthode B.

On pourra ensuite lire la description du cheminement qui nous a permis d'aboutir au résultat final.

Chapitre 1

Présentation de LIEL et du CETIC

C'est au sein de l'équipe LIEL (Laboratoire d'Ingénierie des Exigences Logicielles) des Facultés Universitaires Notre Dame de la Paix de Namur que ce stage a été effectué. Cependant, pour raisons pratiques, il a eu lieu dans les locaux du CETIC, à Charleroi.
Dans ce chapitre, nous ferons donc une brève présentation de l'équipe LIEL et du CETIC.

1.1 L'équipe LIEL

1.1.1 Présentation

L'équipe LIEL fait partie de l'Institut Informatique des FUNDP de Namur. Elle est dirigéé par Pierre-Yves Schobbens, et est actuellement composée de six chercheurs, dont deux doctorants.

1.1.2 Quelques projets

CREWS

Ce projet a commencé en août 1996, et s'est terminé en octobre 1999.

Le projet CREWS (Cooperative Requirements Engineering With Scenarios), a développé, évalué, et demontré l'application de méthode et outils pour l'élicitation et la validation des exigences basées sur des scenarii coopératifs.

Pour plus d'informations, on peut consulter l'URL suivante :
http ://www.info.fundp.ac.be/∼jjtr/Liel/Liel.html

CAT

Le projet CAT (outils de Conception et d'Analyse de cahiers de charges pour des systèmes informatiques de Télécommunication) a débuté le 1er avril 1995 et devait se terminer le 31 décembre 1999. Après une prolongation, il s'est terminé définitivement le 30 juin 2001.

L'objectif du projet de recherche appliquée CAT était de développer un environnement d'outils CASE supportant la modélisation et l'analyse de cahiers de charges relatifs à des systèmes informatiques distribués et temps réel. Cet outil repose sur le langage de modélisation ALBERT développé aux FUNDP par LIEL et expérimenté dans le contexte de la réalisation

de cahiers des charges pour des applications de télécommunication, de productique et de contrôle de processus répartis.

De par ses objectifs, le projet CAT a contribué à :

- renforcer, dans certaines entreprises, une expertise déjà existante en matière de rédaction de cahiers des charges,
- sensibiliser et à assurer une guidance auprès d'entreprises confrontées au problème de la production d'un cahier des charges.

Pour plus d'informations, on peut consulter l'URL suivante :
http ://www.info.fundp.ac.be/∼jtr/Liel/Liel.html

ARTEMI

Ce projet a commencé en novembre 1999, et se terminera en octobre 2003.

Ce projet consiste en l'extension de méthodes formelles en ingénierie des exigences pour la gestion de risques.

La certification officielle des logiciels est devenue une étape préalable incontournable à la commercialisation d'équipements informatisés. Le point central de cette certification est constituée par l'analyse et le contrôle des risques. Sur base de l'expérience de LIEL en ingénierie des exigences, une approche permettant d'aborder cette gestion de risques a été élaborée. Cette approche a constitué une aide précieuse aux PMEs actives, entre autres, dans le secteur médical.

CFV

Le projet CFV (Centre Fédéré en Vérification) a commencé en octobre 2001, et se poursuivra jusqu'en décembre 2005.

Ce projet consiste en l'études des méthodes formelles pour la vérification assistée par ordinateur de systèmes concurrents :

- traitement du temps réel,
- espaces d'états infinis,
- modularité.

CEDIE

Le projet CEDIE (Cellule d'Expertise et de Diffusion en Ingénierie des Exigences) a commencé en juillet 2001, et s'arrêtera en juillet 2006.

La CEDIE fait partie du CETIC -Centre d'Excellence en Technologies de l'Information et de la Communication-, association à but non lucratif interuniversitaire créée dans le but de transférer des technologies informatiques depuis les universités vers les entreprises. La CEDIE organise ce transfert dans le domaine spécifique de l'ingénierie des exigences (IE) : son objectif est de fournir des services aux entreprises afin d'améliorer leurs pratiques d'IE.

1.2 Le Centre d'Excellence en Technologies de l'Information et de la Communication (CETIC)

1.2.1 Fondateurs

LE CETIC est une association à but non lucratif qui a été crée en juillet 2001 par 3 universités : l'Université Catholique de Louvain, les Facultés Universitaires Notre-Dame de la Paix de Namur, ainsi que la Faculté Polytechnique de Mons. Le CETIC est également soutenu par Commission Européenne et la Région Wallonne.

1.2.2 But

Le CETIC est un centre de transfert de technologie, qui a pour but d'augmenter la qualité logicielle des PME situées en région wallonne. En effet, intégrer des résultats de recherche avancée dans une organisation est souvent une tâche plus ardue qu'il n'y parait. En matière de technologies de l'information et de la communication, le CETIC établit la connexion entre le monde de la recherche et les entreprises, les organisations et les institutions. C'est un acteur déterminant d'intégration de la recherche universitaire de pointe.

Le CETIC se positionne comme un centre de recherche reconnu, et poursuit une mission régionale de soutien et d'impulsion R&D au bénéfice d'entreprises. En tant que centre de recherche, il établit des relations durables de collaboration avec les entreprises et les institutions. Il offre l'accès a un soutien personnalisé par la mise à disposition de ses services et de celles de différents laboratoires universitaires avec lesquels il interagit de façon continue.

De plus, le CETIC met au point de nouveaux produits et procédés en technologies de l'information et de la communication destinés à être valorisés à travers des structures indépendantes.

1.2.3 Domaines de compétences

Le CETIC est forme de 7 équipes dont les compétences se partagent dans les domaines suivants :

- l'ingénierie de base de données,
- les systèmes répartis,
- le traitement du signal en temps réel,
- le génie logiciel.

Dans le domaine de l'ingénierie des bases de donnes, l'équipe CRAQ-REVERSE conçoit et développe des techniques et des outils d'aide a la retro-ingénierie de systèmes de gestion d'information de toute nature, en partie ceux basés sur l'Internet, tels que les sites web, via les technologies XML. Il s'agit de reconstruire la documentation et la spécification d'un système opérationnel, généralement ancien, et parfois mal documenté, en vue d'une conversion de la base de données vers de nouvelles technologies, de son intégration dans une nouvelle application ou d'une amélioration de son évolutivité et de sa maintenabilité. Dans ce domaine, le CETIC apporte une véritable expertise, notamment sous la forme d'aide technique et méthodologique dans les projets de ré-ingénierie de systèmes de gestion d'information.

C'est l'équipe ORAGE qui est dédiée aux systèmes repartis. Cette équipe oeuvre a la mise au point d'outils de programmation visant la haute disponibilité d'applications réparties, et

la tolérance aux pannes de réseau, notamment dans la construction d'applications collaboratives. Une plate-forme de qualité de développement d'applications Internet à disponibilité ininterrompue est en construction. Un prototype appelé GlobalStore a été mis en place. Le CETIC maîtrise également le domaine de l'informatique haute performance, notamment dans ses aspects d'ordonnancement, d'équilibrage de charge et d'évaluation.

L'équipe RETICOM est chargée de l'aspect traitement du signal en temps réel. En effet, le CETIC développe ses compétences dans le domaine de l'implémentation d'algorithmes de traitement du signal sur plate-formes spécifiques (DSP, ASIC, FPGA, architectures reconfigurables, etc.) par exemple à destination de systèmes embarqués, ou temps réel. Les recherches se focalisent sur l'élaboration d'une méthodologie et d'outils logiciels pour le développement de systèmes mixtes hardware/software pour des applications dans le domaine des communications et du traitement de signal.

Dans le cadre du génie logiciel, le CETIC vise les objectifs suivants :
- le développement d'un prototype d'atelier logiciel basé sur KAOS, une approche formelle pour la production de cahier des charges de logiciels évolues (équipe FAUST)
- l'amélioration de la qualité des pratiques d'ingénierie des exigences dans les entreprises par la mise à disposition concrète et active de l'expertise développée, pour permettre une meilleure efficacité dans les relations de sous-traitance de logiciel (équipe CRAQ-CEDIE)
- la production de modèles d'évaluation de la qualité des processus logiciels dans les PME, et la conception de méthodes d'amélioration de ces processus (équipe CRAQ-Qualité), basées sur le modèle de qualité OWPL
- la mise au point d'une méthodologie de certification logicielle basée sur des critères d'évaluation d'objectifs mesurables qui facilitent la distribution et l'échange des produits, composants et services (équipe CRAQ-Certification)

1.2.4 Applications concrètes

Concrètement, le CETIC entretient des relations étroites avec les entreprises, notamment à travers :
- la veille technologique,
- l'animation de groupes de discussion,
- l'accompagnement et la guidance technologique,
- la mise a niveau de compétences, l'impulsion technologique, les transferts de technologies,
- le partenariat dans des projets industriels avancés,
- la recherche sous contrat pour des entreprises,
- l'évaluation de processus de développement logiciel et les recommandations d'amélioration,
- l'encadrement de projets R&D,
- la participation à des réseaux technologiques (Par exemple NOE du sixième Programme Cadre),
- la valorisation de résultats d'actions de recherche.

Chapitre 2

Sujet du stage

Dans ce chapitre, nous présentons le sujet initial du stage, et les modifications qui y ont été apportées dans un premier temps.

2.1 Sujet initial

Voici l'intitulé qui nous a été communiqué au début du stage.

Pré-requis :

- logique du premier ordre,
- preuve de cohérence et de raffinement en B,
- principes de base d'analyse statique

Sujet :

Le but du stage est d'obtenir un programme permettant d'analyser des machines B, de vérifier leur cohérence, et, si l'un des invariants n'est pas respecté, de trouver un contre-exemple. Cet outil sera utilisé en deuxième année de DEUG pour la vérification d'algorithmes.

L'outil actuellement développé par Clearsy permet de prouver qu'une machine est cohérente. Cette preuve est parfois faite automatiquement (cas simples) ou faite manuellement (cas un peu compliqués). Lorsque la spécification n'est pas cohérente, la preuve est manuelle et impossible à faire, mais l'outil ne dit pas que cette preuve est impossible (indécidable). L'idée est donc d'utiliser l'analyse statique pour trouver un contre-exemple et montrer que la spécification n'est pas cohérente.

Le but du stage est donc d'utiliser un outil de résolution de contraintes existant, programmé en Oz, et d'adapter cet outil à un sous-ensemble du langage B. Cet outil pourra servir dans le cadre du cours de programmation de 2e année de DEUG.
Pour ce faire, il faudra apprendre le langage Oz et écrire un parseur pour générer un ensemble de contraintes pour que l'outil existant puisse les analyser et trouver (au-moins) un contre-exemple lorsque la machine est incohérente.

2.2 Modifications

Les travaux effectués ne correspondent pas exactement aux directives de ce sujet. Plusieurs précisions ont été apportées au fur et à mesure, afin de mieux le définir, et de combler certaines imprécisions.

Il a d'abord fallu prendre en compte que le fait que l'utilisateur de notre futur outil ne disposerait probablement pas de l'outil de Clearsy, ou ne serait pas en mesure de s'en servir.

De plus, notre outil étant destiné à l'apprentissage, il nous a fallu réfléchir à une méthode d'enseignement réaliste pour des élèves de deuxième candidature (2ème année de faculté), n'ayant pas forcément une grande expérience de la programmation, ni de la logique du premier ordre.

Nous avons alors pensé qu'il n'était peut-être pas très instructif pour des élèves d'écrire des spécifications en B, avant même de savoir quelles difficultés en impliqueraient leur implantation, ou sans être en mesure de comprendre le contre-exemple fourni, en cas d'incohérence de la machine.

Il nous est alors apparu que l'exercice le plus intéressant dans un premier temps serait de donner à l'élève une spécification sous la forme d'une machine abstraite cohérente, et de lui demander de l'implanter en B0 en s'y conformant. Il ne faudrait plus alors que vérifier que son implantation est correcte, et correspond bien à la spécification qui lui a été donnée, et trouver des contre-exemples si ce n'est pas le cas, afin de bien lui faire prendre conscience son erreur.

Outre le fait d'avoir le nom et la valeur des variables fautives en cas d'erreur, il nous a aussi paru intéressant de pouvoir localiser le point de programme où se trouverait l'erreur, afin de pouvoir l'exploiter dans une éventuelle version ultérieure de l'outil.

L'outil crée au cours du stage s'utilisera donc en définitive avec une machine abstraite cohérente, et une implantation. Son but sera de vérifier que l'implantation raffine bien la machine abstraite fournie, et de fournir des contre-exemples si ce n'est pas le cas.

Chapitre 3

Présentation succinte de la méthode B

Il nous a paru nécessaire de faire une présentaion sommaire de la méthode B, avant d'exposer nos travaux.

Ce qui suit est un extrait modifié d'un document rédigé par Emmanuel Dieul, dans le cadre d'un groupe de discussion organisé par le CETIC, à l'usage d'industriels.

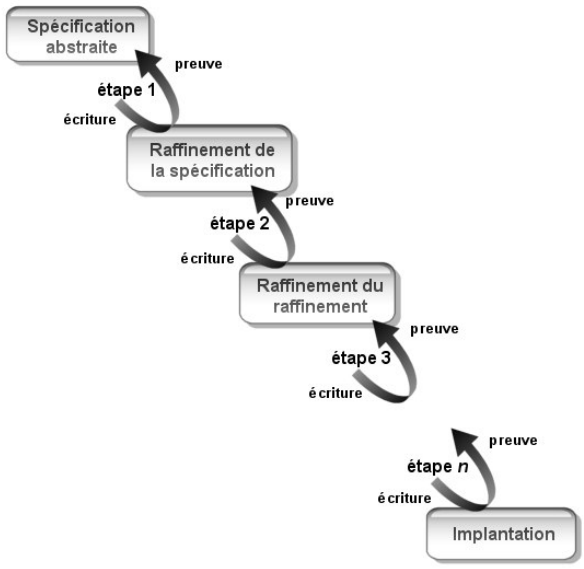
3.1 Introduction à B

La méthode B est une méthode formelle, conçue par Jean-Raymond Abrial, aussi concepteur de la notation Z. Son but premier est la réalisation de systèmes critiques, de systèmes sûrs de fonctionnement.

Le principe de cette méthode est de partir du plus haut niveau de conception (la spécification) pour aller jusqu'à un code implémentable. Ada ou C. Et pour assurer la cohérence des transitions entre les différentes étapes, une preuve mathématique est à faire.

La méthode B fournit le *langage* B, qui permet d'exprimer la spécification dans un langage de haut niveau, et le code dans un langage de bas niveau.

La méthode est donc de spécifier ses besoins dans une spécification de haut niveau, avec la partie du langage la plus abstraite ; de vérifier sa cohérence ; puis de *raffiner* la spécification abstraite par une spécification plus concrète, avec la partie concrète du langage, et de vérifier la cohérence du raffinement. Ce processus peut être représenté par le schéma suivant :



Ce langage est basé au-dessus de la logique du premier ordre, permettant ainsi d'effectuer des vérifications diverses, notamment des vérifications d'invariants.

En effet, la méthode consiste à vérifier que la manipulation des données vérifie toujours les invariants écrits.

3.2 Les composants B

La programmation en B est modulaire. En effet, on peut raisonnablement supposer que l'on développe en B uniquement les modules critiques d'un système ; on part donc d'au-moins un module.

3.2.1 La présentation des composants B

Ce composant va avoir différents niveaux d'abstraction : ces niveaux sont, du plus abstrait au plus concret :

– la *machine*, pour la spécification de haut niveau,
– le *raffinement*, pour la spécification détaillée,
– l'*implantation*, pour le code.

Une machine sera raffinée par un raffinement ; un raffinement sera raffiné par un autre raffinement...et le dernier raffinement sera raffiné par l'implantation.

Dans la pratique, on passera généralement de la machine à l'implantation.

Un module, au sens classique, correspondra à plusieurs composants : au-moins une machine et une implantation.

3.2.2 La preuve d'une machine

Prouver une machine, c'est prouver sa cohérence. En effet, la machine étant la spécification la plus abstraite, il est normal de penser à prouver sa cohérence avant de vouloir la raffiner.

La preuve de cohérence consiste en plusieurs points...

Tout d'abord, il faut prouver que l'initialisation de la machine respecte bien les invariants. Ainsi, la machine est cohérente avant tout traitement.

Ensuite, il faut prouver que chaque opération respecte les invariants : la machine reste cohérente à l'application de traitements.

3.2.3 La preuve d'un raffinement ou d'une implantation

La preuve d'un raffinement ou d'une implantation est un peu plus conséquente. Il n'est plus nécessaire, ici, de prouver la cohérence. Il est nécessaire ici de prouver le raffinement.

Ceci signifie prouver que chaque opération du raffinement satisfait les mêmes exigences que l'opération de la machine abstraite correspondante. Si Op_{MA} est l'opération de la machine abstraite et Op_{MA_i} l'opération du raffinement, il faut que les préconditions de Op_{MA} impliquent celles de Op_{MA_i} (l'opération est appelée dans des conditions prévues par la spécification) ; réciproquement, il faut que les postconditions de Op_{MA_i} impliquent celles de Op_{MA} (l'opération produit bien les effets prévus par la spécification).

3.3 Conclusion

La méthode B se présente donc à travers le langage B, un langage assez simple, proche d'un langage de programmation impératif. Cette méthode de conception en raffinements s'accompagne d'un certain nombre d'obligations de preuves (preuve de cohérence et preuve de raffinement). Ces obligations de preuve sont, pour la plupart, déchargées automatiquement par un prouveur automatique.

Chapitre 4

Le cahier des charges

Après avoir compris le sens global du sujet, il nous a semblé nécessaire de réaliser un cahier des charges permettant de mieux définir le travail à effectuer, et de délimiter les différentes fonctionnalités à implémenter sur l'outil.

Dans ce chapitre nous allons faire le point sur ce que nous avons pu apprendre sur la nature et le rôle d'un cahier des charges, ainsi que les principales décisions que nous avons adoptées.

4.1 Rôle du cahier des charges

Un cahier des charges a les fonctions suivantes :

- établir les bases afin d'arriver à un accord entre le client et son fournisseur sur ce que le produit à réaliser doit faire. Comme l'établissement du cahier des charges scelle l'accord entre les deux partenaires (le cahier des charges est un document contractuel), ceci implique généralement un échange sur la faisabilité et la pertinence des idées soulevées, au vu des contraintes et des impératifs à respecter par les deux partis.
- minimiser le nombre d'heures de codage. En effet, la réalisation d'un cahier des charges permet de révéler au client des omissions, incompréhensions et incohérences tôt dans le cycle de développement, lorsqu'il est encore temps d'éviter certains problèmes.
- servir de base pour l'estimation du coût du produit, et la planification des tâches pour la réalisation du produit. En effet, la description du produit donné dans le cahier des charges est suffisamment réaliste pour permettre l'établissement du devis.
- servir de base pour la validation et la vérification. Effectivement, les plans de tests pourront être développés à partir du document produit.

Dans notre cas, le cahier des charges a servi de base de dialogue entre le client (le maître de stage et le stagiaire) et le fournisseur(le stagiaire).

4.2 Réalisation du cahier des charges

Afin que le document produit soit complet et non ambigu, nous avons essayé de nous conformer aux directives de la norme IEEE Std 830-1998. Le modèle général en est le suivant

```
I Introduction
```

```
II Description générale
III Exigences spécifiques
IV Informations supplémentaires
```

Plusieurs organisations sont cependant possibles pour la partie intitulée “Exigences spécifiques” (partie III) :

- Section 3 organisée par les différents *modes d’utilisation* du produit
- Section 3 organisée autour des différentes *classes d’utilisateur* du produit
- Section 3 organisée autour des différentes *fonctionnalités* du produit
- Section 3 organisée par les différents *stimuli* à donner au produit
- Section 3 organisée autour des différents *objets à utiliser* lors du développement
- Section 3 organisée par *hiérarchie* fonctionnelle

Le canevas qui nous a paru le plus à propos a été celui organisé autour des différentes fonctionnalités du logiciel produit. Ce sont en effet les aspects sur lesquels il nous fallait insister.

Cependant, ce cahier des charges nous semblait incomplet. En effet, nous désirions raffiner la section 1.5 nommée “Vue d’ensemble” . Nous avons donc choisi d’y insérer une section “scenarii d’exécution”.

Voici le plan définitif de notre cahier des charges.

```
1 Introduction
   1.1 But
   1.2 Portée
   1.3 Définitions, acronymes et abréviations
   1.4 Références
   1.5 Vue d’ensemble

2 Description générale
   2.1 Environnement
   2.2 Fonctionnalités de l’outil
       2.2.1 Correction syntaxique
       2.2.2 Correction algorithmique
       2.2.3 Génération de contre-exemples
       2.2.4 Génération de code
       2.2.5 Aide à l’utilisateur
       2.2.6 Configuration de l’outil
   2.3 Caractéristiques des utilisateurs
   2.4 Contraintes
       2.4.1 Contraintes relatives aux entrées
       2.4.2 Contraintes relatives aux types des données
       2.4.3 Contraintes relatives à l’installation de l’outil
       2.4.4 Contraintes relatives aux performances
   2.5 Scenarii d’exécution
   2.6 Hypothèses et dépendances
   2.7 Améliorations futures

3 Exigences spécifiques
```

```
3.1 Exigences spécifiques sur les interfaces externes
     3.1.1 Interfaces utilisateurs
     3.1.2 Interfaces logicielles
     3.1.3 Interfaces matérielles
     3.1.4 Communication entre interfaces
3.2 Exigences spécifiques aux fonctionnalités de l'outil
     3.2.1 Exigences spécifiques pour l'interface de l'outil
     3.2.2 Exigences spécifiques pour la correction syntaxique
     3.2.3 Exigences spécifiques pour la correction algorithmique
     3.2.4 Exigences spécifiques pour la génération de code.
     3.2.5 Exigences spécifiques pour l'aide à l'utilisateur

4 Annexes
```

4.3 Les difficultés

Cette phase du stage a été longue. En effet, plusieurs relectures ainsi que de nombreuses modifications ont été nécessaires avant d'aboutir à la dernière version du document.

Les deux principales difficultés rencontrées ont été, d'une part de délimiter clairement les différentes fonctionnalités de l'outil, et d'autre part, de rédiger le document de telle sorte qu'il soit compréhensible par les intéressés.

En effet, ayant pris tour à tour le rôle de concepteur, de client et de l'utilisateur, il devenait assez difficile de ne pas se laisser submerger par l'un de ces rôles, et de les jouer tous impartialement. Les commentaires constructifs des membres de l'équipe nous ont permis de prendre le recul nécessaire à cette tache.

4.4 Les décisions

Le plus difficile n'a pas été l'écriture du cahier des charges en lui-même, mais les différents choix à faire lors de l'élaboration du document. En effet, comme le sujet n'était pas définitivement établi, il nous a fallu prendre plusieurs décisions. Les plus importantes ont été les suivantes :

- choisir clairement des fonctionnalités de l'outil idéal,
- sélectionner le sous-ensemble du langage B qui devait être utilisé,
- définir exactement le rôle du professeur,
- décider de la forme des opérations.

4.4.1 Les fonctionnalités de l'outil

Plusieurs fonctionnalités nous ont semblé importantes dans cet outil destiné à des étudiants de deuxième candidature.

Afin de rendre son utilisation moins fastidieuse, nous avons choisi d'ajouter aux fonctions prévues de l'outil (à savoir vérifier la correction algorithmique, le raffinement et générer des contre-exemples en cas d'erreurs), il nous a paru essentiel que les erreurs de syntaxe soient repérées.

De plus, afin de que l'élève puisse avoir une trace réutilisable de son travail, nous avons décidé d'adjoindre à l'outil un générateur de code C. Cette fonctionnalité n'a pu être réalisée

dans le cadre de ce stage, faute de temps.

Dans de futures versions de l'outil, il est prévu qu'une aide à l'utilisateur soit crée.

4.4.2 Le sous-ensemble de B utilisé

Voici les clauses de machines abstraites qui seront acceptées par notre outil :

CONSTRAINTS
SETS
CONCRETE_CONSTANTS
ABSTRACT_CONSTANTS
PROPERTIES
CONCRETE_VARIABLES
ABSTRACT_VARIABLES
INVARIANT
ASSERTIONS
INITIALISATIONS
OPERATIONS

Voici les clauses d'implantations qui seront acceptées par notre outil :

IMPLEMENTATION
REFINES
CONSTRAINTS
VALUES
SETS
CONCRETE_CONSTANTS
PROPERTIES
CONCRETE_VARIABLES
INVARIANT
INITIALISATIONS
OPERATIONS

Il faut bien comprendre que notre outil n'analysera qu'une machine abstraite et l'implantation lui correspondant. Notre outil ne prendra donc pas en compte les éventuels raffinements intermédiaires.

Les machines abstraites acceptées par notre outil seront écrites avec les substitutions suivantes :

Skip
ident : = expr
ident : (expr)

De plus, plusieurs substitutions pourront être placées en parallèle (Inst1 || Inst2)

Les implantations acceptées par notre outil seront quant à elles écrites avec les instructions suivantes.

Skip
ident : = expr
VAR ident IN instr END
instr ; instr
ASSERT prop THEN instr END
WHILE prop DO instr INVARIANT prop VARIANT expr END
IF $prop_1$ THEN $instr_1$ ELSEIF $prop_2$ THEN $instr_2$ ... ELSE $instr_n$

4.4.3 Le rôle du professeur

Pour faciliter une première approche de l'outil, nous avons décidé que le professeur fournirait à ses étudiants une machine abstraite, et que ceux-ci seraient chargés de l'implanter, sans pour autant pouvoir changer la machine abstraite initiale. Cette machine abstraite devra être cohérente, et syntaxiquement correcte.

4.4.4 Autres détails

Pour plus d'informations concernant l'outil à réaliser, le cahier des charges placé en annexe peut-être consulté.
Il faut cependant bien noter que nous n'avons pas implanté toutes les fonctionnalités proposées par ce cahier des charges, et que le sous-ensemble de B auquel nous nous sommes restreints est plus petit que le sous-ensemble proposé par le cahier des charges.

Notons de plus qu'aucune vérification concernant le typage des constantes et des variables ne sera fait.

Nous avons décidé d'appelet cet outil "Vics", abbréviation de "Vérification d'une Implémentation Conformément à sa Spécification".

Chapitre 5

Méthode générale

L'intérêt de cette phase du stage a été de mettre en place une méthodologie à appliquer, afin d'éviter l'exploration des solutions possibles au moment de l'implantation, et atteindre plus facilement les objectifs fixés.
Nous avons utilisé une modélisation formelle du système afin d'éviter le plus d'erreurs possible lors de la création de l'outil.
Nous exposerons dans ce chapitre la méthodologie utilisée pour arriver à nos fins.
Nous présenterons dans un premier temps les notations utilisées ainsi que les pré-requis nécessaires à une bonne utilisation de notre outil.
Puis, nous décrirons notre objectif global, et nous signalerons les principales différences entre B et Vics.
Nous donnerons plus de détails sur le fonctionnement de Vics.
Nous préciserons ensuite quelle démarche a été adoptée pour réaliser les vérifications souhaitées.
Enfin, nous détaillerons cette démarche.
Dans ce chapitre, nous avons pu mettre en pratique une partie des connaissances acquises en logique du premier ordre.

5.1 Généralités

5.1.1 Notations

notation	**signification**
$Modules$	ensemble des modules acceptés par Vics
$Machines\_Abstraites$	ensemble des machines abstraites des modules acceptés par Vics
$Implantations$	ensemble des implantations des modules acceptés par Vics
(MA, MA_i)	un couple comprenant une machine abstraite acceptée par Vics et une de ses implantations
$Operations$	ensemble des operations du couple (MA, MA_i
Op_{MA}	ensemble des opérations de MA
Op_{MA_i}	ensemble des opérations de MA_i
$Contraintes$	ensemble des contraintes

On a :

$$Operations = Op_{MA} \cup Op_{MA_i}$$

Définissons les fonctions *pre* et *post*, telles que :

$$pre : Operations \rightarrow Bool$$
$$post : Operations \rightarrow Bool$$

pre et *post* donneront respectivement la précondition et la post-condition d'une opération.

Définissons une fonction *op_imp* telle que :

$$op\_imp : Op_{MA} \rightarrow Op_{MA_i}$$

op_imp est une fonction qui, pour un couple (MA, MA_i) à chaque opération de la machine abstraite fait correspondre une opération de son implantation.
op_imp est une bijection.

En effet, dans notre cas précis, les opérations présentes dans MA_i seront des implantations des opérations de la machine abstraite car toutes les opérations de MA doivent être implantées dans MA_i, et, comme nous n'autorisons pas l'écriture de la clause "LOCAL_OPERATIONS", qui permet de déclarer des opérations locales dans les implantations, toutes les opérations de MA_i sont des implantations des opérations de MA.

Nous appellerons *op_abs* la fonction réciproque de *op_imp*.

5.1.2 Objectif

Nous avons crée un outil d'aide à l'enseignement permettant d'enseigner le langage B à des étudiants de premier cycle. Le but premier de cet outil est de vérifier le raffinement, en donnant des contre-exemples en cas d'erreur, sans que l'étudiant ait beaucoup d'efforts à faire. Nous avons décidé de procéder de la façon suivante : le professeur fournira à l'étudiant une machine abstraite cohérente, et l'étudiant devra l'implanter. Puis, à l'aide de l'outil, l'étudiant devra vérifier une à une toutes les opérations qu'il a écrites. Pour cela, il lui suffira de communiquer à l'outil la machine abstraite, l'implantation qu'il en a écrite, et de lancer les vérifications.
En cas d'erreur, l'outil lui rendra un contre-exemple, et l'emplacement de l'erreur dans le code qu'il a écrit.

5.1.3 Pré-requis : la cohérence de la machine abstraite

En B, nous devons d'abord prouver la cohérence de la machine abstraite. Dans le cadre de l'utilisation de notre outil, nous supposons que nous disposerons d'une machine abstraite cohérente. Vics ne vérifiera donc pas la cohérence de la machine abstraite fournie.

Comme cela a déjà été précisé précédemment, nos modules ne comporteront qu'une machine abstraite, et qu'une implantation.

5.1.4 Restrictions de B pour l'utilisation de Vics

De nombreuses clauses et expressions acceptées en B seront refusées par Vics.

Il nous faut aussi préciser que certaines vérifications syntaxiques ne seront pas effectuées par Vics. Nous avons donc pris pour hypothèse que les fichiers fournis ne comporteront pas d'erreurs syntaxiques susceptibles de remettre en cause le bon fonctionnement de notre outil.

A l'instar de B qui permet de prouver le raffinement de toutes les opérations d'une machine abstraite, notre outil permettra à l'utilisateur de lancer des vérifications de raffinement sur chaque opération.

Notons bien que nous ne parlons pas de *preuves*, mais de *vérifications*. En effet, nous n'effectuerons aucune preuve logique, mais juste des vérifications sur un domaine de définition restreint.

Cependant, la principale différence entre B est Vics est la suivante : en B, les preuves ne sont pas toujours faites automatiquement. Avec Vics, l'utilisateur n'a pas de preuves à faire, et n'a pas besoin de beaucoup de connaissances en logique pour pouvoir vérifier son implantation.

Avec Vics, pour raisons de performances, nous ne pourrons travailler que sur des domaines très restreints.

5.2 Principe général de fonctionnement

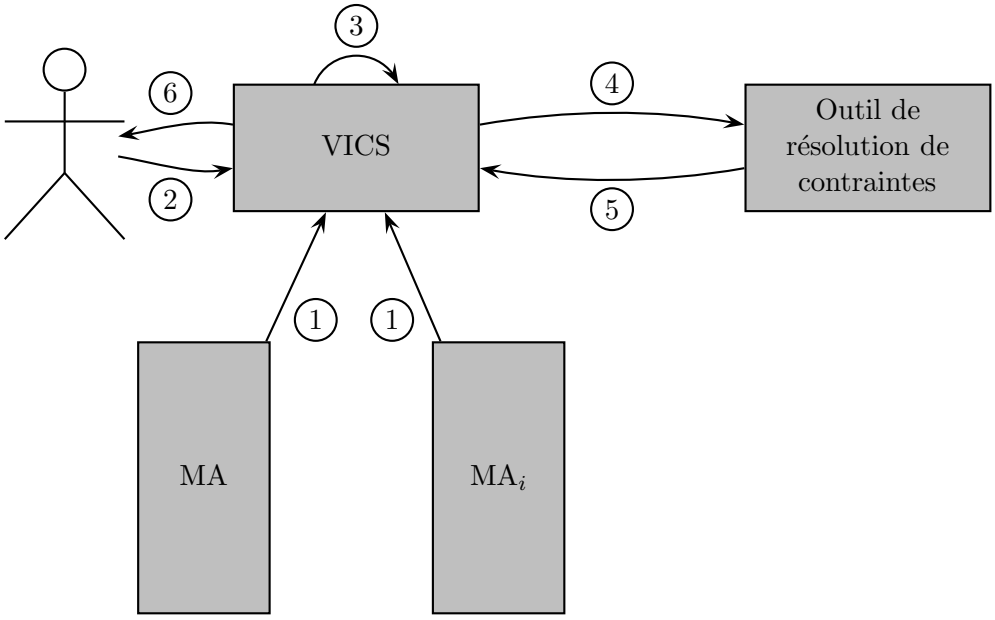
Nous disposons d'une machine abstraite MA, et d'une implantation MA_i.

Nous souhaitons que notre outil vérifie si le code de MA_i est correct, et si MA_i est bien une implantation de MA. Si ce n'est pas le cas, nous souhaiterions savoir où sont situés les problèmes (dans quelle opération, pour quelle instruction).
Pour cela, nous souhaitons avoir des contres-exemples, et pouvoir retrouver la source de l'erreur.

Pour répondre à ces exigences, nous avons choisi de travailler avec un outil de résolution de contraintes, afin de pouvoir obtenir aisément les contre-exemples.
Afin d'effectuer ces vérifications et pour pouvoir rendre un contre exemple, l'outil que nous avons créé analysera le code des opérations dans la machine abstraite et dans l'implantation, et génèrera des contraintes, qu'il fournira à l'outil de résolution de contraintes. Ce dernier communiquera à Vics ses résultats, qui seront interprétés, et communiqués à l'utilisateur.

Notons que l'outil de résolution de contraintes fonctionne par exhaustivité. C'est de la que viennent les faibles performances lors de l'utilisation de grands domaines de variables.

Voici un diagramme permettant de mieux comprendre les différentes activités ayant lieu.



(1) :Données de l'utilisateur

(2) :Choix de l'opération

(3) :traduction en contraintes

(4) :Transmission des contraintes

(5) :Envoi des contre-exemples

(6) :Affichage des informations

5.3 Principe général des vérifications

Pour pouvoir conclure que le code fourni par l'étudiant implante bien la machine abstraite donnée, des vérifications devront être faites.

Pour chaque opération de la machine abstraite, il va falloir effectuer des vérifications similaires aux obligations de preuve de raffinement en B : vérifications sur les post-conditions et sur les préconditions.

Nous montrerons en 5.4.1 qu'aucune vérification sur les préconditions n'est nécessaire.

Pour les post-conditions, nous avons décidé de scinder la vérification en deux parties : la correction du code et le respect des spécifications.

En effet, il peut arriver que le code de l'étudiant soit écrit de façon incorrecte. Il arrive

souvent à des débutants d'écrire des boucles infinies, ou de ne pas respecter des invariants. Le rôle de notre outil est aussi de faire prendre conscience à l'étudiant de ce genre de problèmes, et de lui montrer ses erreurs, s'il en commet.

Cependant, même si le code de l'étudiant est correctement écrit, ce code peut cependant ne pas respecter les spécifications de la machine abstraite donnée par l'enseignant.

Nous voyons donc bien que deux vérifications doivent être faites.

La démarche adoptée restera proche de celle de B : nous vérifierons une à une nos opérations.

5.3.1 Notations

Afin de clarifier les explications qui suivront, nous introduisons quelques notations.

De chaque opération, nous tirerons des contraintes, que nous nommerons "contrainte de correction" et "contrainte de spécification". Elles nous serviront à vérifier respectivement la correction du code, et le respect des spécifications données par la machine abstraite.

Définissons les fonctions $c_{Correction}$ et $c_{Specification}$, telles que :

$$\begin{aligned} c_{Correction} &: Operations \rightarrow Contraintes \\ c_{Specification} &: Operations \rightarrow Contraintes \end{aligned}$$

$c_{Correction}$ et $c_{Specification}$ donneront respectivement la contrainte de correction et la contrainte de spécification d'une post-condition.

5.3.2 Exemple

Notons d'abord que la contrainte de correction et la post-condition d'une opération sont des formules différentes, bien qu'elles soient toutes deux déduites des instructions de l'opération. Il faut bien voir que l'ensemble des contraintes de correction n'est pas un sous-ensemble de l'ensemble des post-conditions.

Voici un exemple permettant de visualiser les notions décrites auparavant. Considérons l'opération suivante :

```
operation_exemple(x):
   WHILE x < 20 DO
      x := x+1
   INVARIANT
      x <= 30
   VARIANT
      20 - x
   END
END
```

La contrainte de spécification de cette opération signifiera qu'à la fin de la boucle, "x < 20" ne sera plus vrai, et que l'invariant sera toujours respecté.

La contrainte de correction de cette opération sera la conjonction des contraintes suivantes :
- l'invariant est vrai avant la boucle,
- l'invariant est vrai à chaque tour de boucle,
- le variant est un nombre naturel,
- le variant décroît.

Nous verrons plus clairement au chapitre suivant comment ces contraintes sont obtenues de façon plus formelle.

5.3.3 La correction du code de l'étudiant

Pour pouvoir conclure que le code fourni par l'étudiant est correct, il va falloir déterminer pour chaque opération ce que nous avons appelé la "contrainte de correction".

Dans la machine abstraite, le code sera fourni par le professeur, nous supposerons donc qu'il est correct, et que la contrainte de correction de chaque opération de la machine abstraite sera *true*.

Pour les opérations de l'implantation, cette contrainte sera liée aux instructions composant cette opération ; nous détaillerons cela au prochain chapitre.

Si l'outil de résolution de contraintes n'arrive pas à vérifier que nos contraintes sont vraies sur le domaine de définition choisi, nous pourrons déduire que le code de l'étudiant est écrit de façon incorrecte.

5.3.4 Le respect des spécifications des machines abstraites

Pour chaque opération du module, il va falloir effectuer des vérifications sur le respect des contraintes de spécification.

Nous générerons donc les contraintes de spécifications des opérations des machines abstraites, et des implantations.

La contrainte de spécification d'une opération dépend des instructions composant cette opération. Nous analyserons donc une à une les instructions d'une opération afin de générer les contraintes nécessaires aux les vérifications voulues.

Si la contrainte due à la contrainte de spécification de l'opération de la machine abstraite est fausse, la spécification donnée par le professeur n'aura pas été respectée, et un contre-exemple sera trouvé.

5.4 Démarche adoptée

Voici les vérifications qui seront effectuées avec l'outil de résolution de contraintes.

5.4.1 Les préconditions des opérations

En B, une partie de la preuve de raffinement consiste à montrer que les préconditions de chaque opération de la machine abstraite impliquent les préconditions des raffinements de ces opérations.

Il nous faudrait donc vérifier :

$$\forall Op.\big(Op \in Op_{MA} \Rightarrow (pre(Op) \Rightarrow pre(op\_imp(Op)))\big)$$

Ici, notre module étant composé d'une seule machine abstraite et de son implantation, cette vérification n'a pas lieu d'être. Comme les opérations de l'implantation n'ont pas de préconditions, ces préconditions valent *true*.

Or, nous savons que pour chaque opération Op, $pre(\text{Op}) \Rightarrow true$ est une tautologie. Il est donc inutile d'effectuer cette vérification.

5.4.2 Les post-conditions des opérations

En B, une autre partie de la preuve de raffinement consiste à montrer, pour chaque opération, que la post-condition de cette opération dans l'implantation implique la post-condition de cette opération dans la machine abstraite.

Il faut vérifier en B :

$$\forall Op.(Op \in Op_{MA_i} \Rightarrow (post(Op) \Rightarrow post(op\_abs(Op))))$$

Comme nous l'avons précisé ci dessus, nous voulons non seulement vérifier que les spécifications sont respectées, mais aussi que le code est correctement écrit.

Nous voulons donc vérifier la formule suivante :

$$\begin{array}{l}\forall Op.(Op \in Op_{MA_i} \Rightarrow \\ \quad (c_{Specification}(Op) \Rightarrow \\ \quad\quad (c_{Correction}(op\_abs(Op)) \wedge c_{Correction}(Op) \wedge c_{Specification}(op\_abs(Op)))))\end{array}$$

Or, nous savons que $c_{Correction}(op\_abs(Op)) = true$.

La formule précedente est donc équivalente à :

$$\begin{array}{l}\forall Op.(Op \in Op_{MA_i} \Rightarrow \\ \quad (c_{Specification}(Op) \Rightarrow (c_{Correction}(Op) \wedge c_{Specification}(op\_abs(Op)))))\end{array}$$

Cette dernière formule équivaut à :

$$\begin{array}{l}\forall Op.(Op \in Op_{MA_i} \Rightarrow \\ \quad ((c_{specification}(Op) \Rightarrow c_{Correction}(Op)) \wedge (c_{Specification}(Op) \Rightarrow (c_{specification}(op\_abs(Op))))))\end{array}$$

Pour chaque opération, nous devrions donc vérifier les deux formules suivantes :

$$\begin{array}{lcl}(c_{Specification}(Op) & \Rightarrow & c_{Correction}(Op)) \\ & \text{et} & \\ (c_{Specification}(Op) & \Rightarrow & (c_{Specification}(op\_abs(Op))))\end{array}$$

Cependant, nous voulons des contre-exemples. Nous n'allons donc pas vérifier ces dernières formules, mais leur négation.

La vérification due à la correction du code

A partir des informations dont nous disposons sur l'état des variables après l'exécution de toutes les instructions de l'opération, il va falloir trouver l'ensemble de solutions de la contrainte suivante, afin de disposer d'un ensemble de contre-exemples.

Pour chaque Op de Op_{MA_i} :

$$(c_{Specification}(Op) \wedge \neg c_{Correction}(Op))$$

La vérification de respect de spécification des opérations

A partir des informations dont nous disposons sur l'état des variables après l'exécution de toutes les instructions de l'opération, il va falloir vérifier que les contraintes de correction sont vraies. Nous allons pour chaque opération essayer de trouver l'ensemble de solutions de la contrainte suivante.

Pour chaque Op de Op_{MA_i} :

$$c_{Specification}(Op) \wedge \neg c_{Specification}(op\_abs(Op))$$

Ainsi, après résolution, si l'ensemble de solutions trouvé par l'outil de résolution de contraintes est vide, la contrainte est vérifiée ; et si l'ensemble de solutions est non vide, il contiendra les valeurs possibles des variables au moment où une erreur est détectée, constituant ainsi des contre-exemples.

5.4.3 Optimisation sur les analyses et vérifications

Plutôt que de faire deux analyses du code de l'implantation afin d'obtenir pour chaque opération d'une part les contraintes correspondant aux contraintes de spécification, et d'autre part, les contraintes correspondant aux contraintes de correction du code, et de lancer deux vérifications, nous avons préféré n'en faire qu'une, et pour chaque opération de Op_{MA_i}, vérifier la contrainte suivante :

$$c_{Specification}(Op) \wedge \neg c_{Specification}(op\_abs(Op)) \wedge c_{Specification}(Op) \wedge \neg c_{Correction}(Op)$$

c'est à dire :

$$c_{Specification}(Op) \wedge \neg(c_{Specification}(op\_abs(Op)) \vee c_{Correction}(Op))$$

Chapitre 6

Contraintes dues à l'implantation

Nous voulons ici exposer comment seront obtenues les contraintes dues à chaque opération de l'implantation. Ces contraintes nous permettront d'effectuer les vérifications liées à la contrainte de spécification et à la contrainte de correction de cette même opération.
Nous présenterons d'abord le principe de base de cette tâche, à savoir la génération de nouvelles variables, ainsi que son utilité. Puis, nous présenterons l'ensemble des notations que nous avons utilisées.
Nous donnerons ensuite la forme des instructions et des expressions.

Par souci de clarté, avant de nous attacher à la façon d'obtenir les contraintes dues à une opération, nous détaillerons comment seront traitées les expressions booléennes et arithmétiques au sein d'une instruction.
Puis, comme les contraintes relatives à une opération dépendent directement des instructions composant le corps de l'opération, nous allons analyser une à une les instructions du sous-ensemble de B auquel nous nous sommes restreints. Nous détaillerons enfin les contraintes qui seront générées pour chacune de ces instructions.

Nous fournirons au fur et à mesure différents exemples afin de faciliter la lecture de ce document, puis nous fournirons un dernier exemple, plus complet que les précédents.

Dans cette phase du stage, nous avons dû faire appel à des connaissances acquises en sémantique.

6.1 La génération de nouvelles variables

Anticipons quelque peu, et examinons la contrainte générée pour l'instruction suivante :

```
a := 0 ;
b := a + 1 ;
a := b
```

Nous pouvons générer la contrainte liée à cette séquence d'affectations. Nous obtenons alors la formules suivante :

$$a = 0 \wedge b = a + 1 \wedge a = b$$

Nous constatons que cette formule est fausse, avant même de savoir ce que nous devons vérifier.

Nous devons en fait générer la contrainte suivante :

$$a_0 = 0 \wedge b_0 = a_0 + 1 \wedge a_1 = b_0$$

Ainsi, aucune confusion ne sera possible.

Afin d'écrire nos contraintes de correction et nos contraintes de spécification de façon cohérente, nous nous sommes donc vus obligés de générer de nouvelles variables à chaque affectation, afin de renommer les variables situées en membre gauche de ces affectations : si nous ne le faisions pas, plusieurs contraintes contradictoires concernant une même variable du code pourraient être générées.

6.2 Notations utilisées

6.2.1 Les principaux ensembles

Voici comment nous noterons les ensembles utilisés.

notation	signification
$Contraintes$	ensemble des contraintes
$Variables$	ensemble de toutes les variables
$Variables_{generees}$	ensemble des variables générées à chaque affectation
$Identificateurs_B$	ensemble des identificateurs B
$BExpr$	ensemble des expressions du langage B
$Expr_a$	ensemble des expressions arithmétiques de la théorie des ensembles
$Expr_b$	ensemble des expressions booléennes de la théorie des ensembles
$Expr_{generee}$	ensemble des expressions ne contenant que des variables générées, des nombres et des booléens
$Expr$	$Expr_a \cup Expr_b$
$Instr$	ensemble des instructions
Σ	$Identificateurs_B \rightarrow \text{iseq}(Variables_{generees})$

6.2.2 Les fonctions

Fonctions utilisant σ

Soit σ un élément de Σ, x une variable élément de $Identificateurs_B$. On notera :

$$\sigma(x) = S_x, avec\ \ S_x \in seq(Variables_{generees})$$
$$\sigma x = S_x(size(S_x) - 1)$$

De plus, on notera :

$$\sigma[x : v] = \sigma' \mid \sigma'(x) = \sigma(x) \leftarrow v \wedge \forall y.(y \neq x \Rightarrow \sigma'(y) = \sigma(y))$$

On appellera $Expr_{gen}$ l'ensemble des expressions où toute variable x de $Identificateurs_B$ est remplacée par σx.

La fonction $fresh$

Notons $fresh$ la fonction suivante :

$$fresh : Identificateurs_B \times iseq(Variables_{generees}) \rightarrow \\ Variables - (Identificateurs_B \cup Variables_{generees})$$

Pour une variable v de $Identificateurs_B$, cette fonction génère une variable de $Variables$ n'appartenant ni à $Identificateurs_B$ ni à $Variables_{generees}$.

6.3 Description des instructions et des expressions

Voici la grammaire des instructions et des expressions booléennes, qui va nous permettre de mieux analyser nos programmes.

⟨*Instruction*⟩ ::= '`skip`'
| ⟨*Ident*⟩ '`:=`' ⟨*Expr*⟩
| '`VAR`' ⟨*Ident_list*⟩ '`IN`' ⟨*Instruction*⟩ '`END`'
| '`IF`' ⟨*Condition*⟩ '`THEN`' ⟨*Instruction*⟩ { '`ELSIF`' ⟨*Condition*⟩ '`THEN`' ⟨*Instruction*⟩ } ['`ELSE`' ⟨*Instruction*⟩] '`END`'
| '`ASSERT`' ⟨*Expr_b*⟩ '`THEN`' ⟨*Instruction*⟩ '`END`'
| '`WHILE`' ⟨*Condition*⟩ '`DO`' ⟨*Instruction*⟩ '`VARIANT`' ⟨*Expr_a*⟩ '`INVARIANT`' ⟨*Expr_b*⟩'`END`'
| ⟨*Instruction*⟩'`;`'⟨*Instruction*⟩

⟨*Expr*⟩ ::= ⟨*Expr_b*⟩
| ⟨*Expr_a*⟩

⟨*Expr_b*⟩ ::= ⟨*Condition*⟩
| ⟨*Predicat*⟩

⟨*Condition*⟩ ::= ⟨*Expr_a*⟩'`=`'⟨*Expr_a*⟩
| ⟨*Expr_a*⟩ '$\geq$' ⟨*Expr_a*⟩
| ⟨*Expr_a*⟩'`>`'⟨*Expr_a*⟩
| ⟨*Expr_a*⟩ '$\leq$' ⟨*Expr_a*⟩
| ⟨*Expr_a*⟩'`<`'⟨*Expr_a*⟩
| ' $\neg$ ' ⟨*Expr_b*⟩
| ⟨*Expr_b*⟩'`=`'⟨*Expr_b*⟩
| ⟨*Expr_b*⟩ '$\wedge$' ⟨*Expr_b*⟩
| ⟨*Expr_b*⟩ '$\vee$' ⟨*Expr_b*⟩
| ⟨*Expr_b*⟩ '$\Rightarrow$' ⟨*Expr_b*⟩
| ⟨*Var*⟩
| '`true`'
| '`false`'

⟨*Predicat*⟩ ::= '{'⟨*Ident_list*⟩'}' '⊆' ⟨*Ens*⟩

⟨*Expr_a*⟩ ::= ⟨*Expr_a*⟩ '+' ⟨*Expr_a*⟩
| ⟨*Expr_a*⟩ '-' ⟨*Expr_a*⟩
| ⟨*Expr_a*⟩ '*' ⟨*Expr_a*⟩
| ⟨*Expr_a*⟩ '/' ⟨*Expr_a*⟩
| ⟨*Var*⟩
| ⟨*Nombre*⟩

⟨*Ident_list*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩','⟨*Ident_list*⟩

⟨*Ident*⟩ ::= ⟨*Lettre*⟩ (⟨*Lettre*⟩| ⟨*Chiffre*⟩) {{⟨*Lettre*⟩} {⟨*Chiffre*⟩} }

⟨*Var*⟩ ::= ⟨*Ident*⟩

⟨*Ens*⟩ ::= 'BOOL'
| 'NAT'
| ⟨*Nombre*⟩'..'⟨*Nombre*⟩

⟨*Nombre*⟩ ::= ⟨*Chiffre*⟩{⟨*Chiffre*⟩}

⟨*Chiffre*⟩ ::= '1' | '2'| '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

⟨*Lettre*⟩ ::= 'a' | ...| 'z'
| 'A' | ...| 'Z'

6.4 Les contraintes dues aux instructions

6.5 Le traitement des expressions

Afin de faciliter l'écriture de la fonction de génération de contraintes des instructions, nous allons décrire une fonction qui traitera les expressions arithmétiques et booléennes.

Nous remarquons que dans notre langage, une expression dépend des variables qui y figurent.

Plus précisément, elle dépend d'un *état* σ, fonction qui associe à chaque variable La liste des variables générées lui correspondant, et notamment la dernière.

Soit *expr* une expression de *Expr*. Soit σ un élément de Σ. Le traitement effectué sur *expr* sera le suivant : chacune de ses variables sera remplacée par la dernière variable générée qui lui a été rattachée dans σ.

Cette fonction aura donc pour but d'associer à chaque expression l'expression de $Expr_{generee}$ lui correspondant.

$$\mathcal{G}en_{Expr}[\![-]\!] : BExpr \times \Sigma \rightarrow Expr_{generee}$$

Traitement des expressions arithmétiques

Pour les expressions arithmétiques, cette fonction est définie par les équations suivantes :

$$\begin{array}{lcl}
\mathcal{G}en_{Expr}\,[\![\, nombre \,]\!]\,\sigma & = & nombre \\
\mathcal{G}en_{Expr}\,[\![\, x \,]\!]\,\sigma & = & \sigma x \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} + \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma + \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} - \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma - \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} * \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma * \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`}/\text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma / \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, \text{`(`}expr_a\text{`)'} \,]\!]\,\sigma & = & (\mathcal{G}en_{Expr}\,[\![\, expr_a \,]\!]\,\sigma)
\end{array}$$

Exemple : Soit σ un élément de Σ tel que :

$$\sigma = \Big\{ \big(x, (x_0, x_1, x_2)\big), \big(y, (y_0, y_1, y_2, y_3)\big), \big(z, (z_0)\big) \Big\}$$

On a alors :

$$\mathcal{G}en_{Expr}\,[\![\, \text{`('}\ x\ \text{`}-\text{'}\ 2\ \text{`)'}\ \text{`}*\text{'}\ y\ \text{`}+\text{'}\ z \,]\!]\,\sigma = (x_2 - 2) * y_3 + z_0$$

Traitement des expressions booléennes

Pour les expressions booléennes, cette fonction est définie par les équations suivantes :

$$\begin{array}{lcl}
\mathcal{G}en_{Expr}\,[\![\, \text{`}false\text{'} \,]\!]\,\sigma & = & false \\
\mathcal{G}en_{Expr}\,[\![\, \text{`}true\text{'} \,]\!]\,\sigma & = & true \\
\mathcal{G}en_{Expr}\,[\![\, x \,]\!]\,\sigma & = & \sigma x \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} \wedge \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \wedge \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} \vee \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \vee \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} \Rightarrow \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \Rightarrow \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, \text{`}\neg\text{'}a \,]\!]\,\sigma & = & \neg \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} = \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma = \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} < \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma < \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} > \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma > \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} \geq \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \geq \mathcal{G}en_{Expr}\,[\![\, b \,]\!]\,\sigma \\
\mathcal{G}en_{Expr}\,[\![\, a\text{`} \leq \text{'}b \,]\!]\,\sigma & = & \mathcal{G}en_{Expr}\,[\![\, a \,]\!]\,\sigma \leq \mathcal{G}en_{Expr}\,[\![\, b \,]\!] \\
\mathcal{G}en_{Expr}\,[\![\, \text{`(`}expr_a\text{`)'} \,]\!]\,\sigma & = & (\mathcal{G}en_{Expr}\,[\![\, expr_a \,]\!]\,\sigma)
\end{array}$$

Exemple : Soit σ un élément de Σ tel que :

$$\sigma = \Big\{ \big(x, (x_0, x_1)\big), \big(y, (y_0, y_1, y_2)\big), \big(z, (z_0)\big) \Big\}$$

On a alors :

$$\mathcal{G}en_{Expr}\,[\![\, x\ \text{`}\wedge\text{'}\ y \,]\!] = x_1 \wedge y_2$$

6.6 Le traitement des instructions

Nous allons dans cette section décrire la fonction qui génère les contraintes liées à une instruction.

Pour pouvoir générer les contraintes liées à une instruction, quatre informations seront nécessaires, et devront donc être passées en paramètre de la fonction de génération de contraintes :

– l'instruction elle-même,
– les contraintes de correction relatives à cette opération ayant déjà été déterminées,
– l'ensemble des variables qui ont déjà été générées,
– l'environnement d'évaluation de l'instruction

L'ensemble des contraintes de correction déjà déterminées sera nommé $c_{correction}$. Ce paramètre ne nous sera utile que dans certains cas.
L'ensemble des variables générées auparavant sera obtenu grâce à un élément de Σ, nommé $\sigma_{generation}$.
Ce paramètre nous permettra de générer des variables en toute sécurité, sans risque d'attribuer deux fois la même variable à un identificateur B.
L'environnement d'évaluation sera aussi un élément de Σ, nommé $\sigma_{evaluation}$. Ce paramètre nous permettra d'évaluer les expressions de notre instruction, sans avoir à nous préoccuper du contexte dans lequel cette dernière se trouve (nous n'aurons donc pas besoin de savoir si l'instruction courante se trouve dans une boucle, dans une séquence ou une conditionnelle.)

Générer des contraintes ne sera pas le seul rôle de cette fonction. Il va aussi falloir mettre à jour l'ensemble des variables générées et l'environnement d'évaluation.
Notre fonction aura donc pour signature :

$$\mathcal{G}en_{Instr}[\![-]\!] : Inst \times C \times \Sigma \times \Sigma \rightarrow C \times C \times \Sigma \times \Sigma$$

Le premier paramètre de cette fonction est l'instruction à traiter, le deuxième sera la contrainte de correction déjà existante, le troisième l'environnement de génération des variables, le quatrième l'environnement d'évaluation des variables.

Les éléménts du résultat sont respectivement la contrainte de spécification générée par le traitement de l'instruction, la contrainte de correction, l'environnement de génération, et l'environnement d'évaluation aprés le traitement de l'instruction.

Reprenons nos instructions une par une, afin de définir exactement cette fonction.

6.6.1 Skip

Pour cette instruction, aucune nouvelle variable n'est générée. La seule contrainte générée est *true*.

$$\mathcal{G}en_{Instr}[\![\texttt{skip}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}_{c_{correction}} = (true, c_{correction}, \sigma_{generation}, \sigma_{evaluation})$$

6.6.2 Affectation devient-égal

Une affectation s'écrit en B de la façon suivante :

```
id := expr
```

Pour traiter cette instruction, il va falloir :
– générer une nouvelle variable et la rattacher à `id`,
– générer la contrainte signifiant que cette variable générée dépend de celles de `expr`,
– compléter $\sigma_{generation}$,

– compléter $\sigma_{evaluation}$.

Nous avons donc :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{id := expr}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \big(fresh(\texttt{id}, \sigma_{generation}) = \mathcal{G}en_{Expr}\,[\![\,\texttt{expr}\,]\!]\,\sigma,\ c_{correction},\ \sigma_{generation}[\texttt{id} : fresh(\texttt{id}, \sigma_{generation})],\ \sigma_{evaluation}[\texttt{id} : fresh(\texttt{id}, \sigma_{generation})]\big)$$

exemple : Soient $\sigma_{generation}$ et $\sigma_{evaluation}$ tels que :

$$\sigma_{generation} = \Big\{\big(x, (x_0, x_1)\big), \big(y, (y_0, y_1, y_2)\big)\Big\}$$

$$\sigma_{evaluation} = \Big\{\big(x, (x_1)\big), \big(y, (y_0, y_2)\big)\Big\}$$

On a alors :

$$\mathcal{G}en_{Instr}\,[\![\ \texttt{x := y+1}\ ]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \big(x_2 = y_2 + 1,\ c_{correction},\ \sigma_{generation}[x : x_2],\ \sigma_{evaluation}[x : x_2]\big)$$

6.6.3 Séquence

Une séquence est écrite en B de la façon suivante :

```
instruction1 ; instruction2
```

Pour `instruction1`, les résultats seront générés de la façon habituelle. Cependant, pour `instruction2`, il va falloir prendre comme paramètres les résultats obtenus avec `instruction1`.

Notons :

$$c_{correction1} = (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_2$$

$$\sigma_{generation1} = (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_3$$

$$\sigma_{evaluation1} = (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_4$$

Nous obtenons donc en définitive l'équation suivante :

$$\begin{aligned}
&\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1 ; instruction2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \\
&\big((\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_1 \wedge (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation1}}_{\sigma_{generation1}\ c_{correction1}})\downarrow_1,\\
&\quad (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation1}}_{\sigma_{generation1}\ c_{correction1}})\downarrow_2,\\
&\quad (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation1}}_{\sigma_{generation1}\ c_{correction1}})\downarrow_3,\\
&\quad (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation1}}_{\sigma_{generation1}\ c_{correction1}})\downarrow_4\ \big)
\end{aligned}$$

6.6.4 Les conditionnelles

Les conditionnelles peuvent s'écrire en B de différentes façons.

Conditionnelle "IF THEN"

Cette conditionnelle s'écrit de la façon suivante :

```
IF condition THEN
    instruction
END
```

Cette instruction sera traitée comme si elle était écrite de la façon suivante :

```
IF condition THEN
    instruction
ELSE
    skip
END
```

Conditionnelle "IF THEN ELSE"

Cette instruction s'écrit :

```
IF condition THEN
    instruction1
ELSE
    instruction2
END
```

L'équivalent logique de cette formule est :

$$(\texttt{condition} \Rightarrow \texttt{instruction1}) \wedge (\neg(\texttt{condition}) \Rightarrow \texttt{instruction2})$$

C'est le traitement de cette instruction qui justifie certains de nos choix à propos de la signature de la fonction $\mathcal{G}en_{Instr}\,[\![-]\!]$. En effet, considérons une conditionnelle, où `instruction1` serait une affectation quelconque ayant pour membre gauche une variable `x`, et où `instruction2` serait une affectation qui aurait pour membre gauche une variable `y`, et où la variable `x` apparaîtrait dans le membre droit, par exemple :

```
IF cond THEN
    x := x+1
ELSE
    y := y+x
END
```

Lors de la génération de contraintes pour la branche "`ELSE`" de notre conditionnelle, il faudra faire attention à ne pas générer de contraintes liant $fresh(\texttt{y})$ à la dernière variable générée pour `x`, mais à la dernière variable générée pour `x` *avant* d'entrer dans la conditionnelle.

Il nous faut pour cela connaître deux choses :

– l'ensemble des variables déjà générées, pour générer $fresh(\texttt{y})$,

– l'environnement avant la conditionnelle, pour pouvoir générer des contraintes cohérentes.

Nous avons donc choisi d'avoir à notre disposition l'ensemble des variables générées, que nous avons appelé $\sigma_{generation}$, et l'environnement antérieur à l'entrée de la conditionnelle, que nous avons appelé $\sigma_{evaluation}$.

Pour traiter cette instruction, il va falloir :
– générer les contraintes liées à (`condition`⇒ `instruction1`)
– générer les contraints liées à (¬`condition`⇒ `instruction2`)
En effet, comme il n'y aura eu aucune évaluation préalable, nous ne pourrons savoir à l'avance la valeur de `condition`, et en conséquence choisir les contraintes à générer.

De plus, pour traiter cette instruction, nous avons décidé de générer des contraintes que nous avons appelées contraintes d'*homogénéisation*. Il va falloir faire en sorte d'*homogénéiser* les environnements résultants des deux branches, afin de simplifier le traitement de la séquence : il faut qu'au sortir de la conditionnelle, chaque variable ayant été touchée par une affectation dans une des branches ou dans les deux, soit associée à la même variable générée au sortir de la branche droite qu'au sortir de la branche gauche.

Considérons par exemple :

```
IF x=0 ∧ z=3 THEN
    x := x+1
ELSE
    x := x+2
END
```

$$\sigma_{generation} = \Big\{\big(x,(x_0)\big),\big(z,(z_0)\big)\Big\}$$
$$\sigma_{evaluation} = \Big\{\big(x,(x_0)\big),\big(z,(z_0)\big)\Big\}$$

Nous voyons bien que deux variables différentes devront être générées pour x, l'une dans la branche "`THEN`", l'autre dans la branche "`ELSE`".
Si une autre instruction utilise x ultérieurement, il va falloir savoir quelle est la dernière variable rattachée à x pour générer les contraintes. Sans contrainte d'homogénéisation, il est impossible de savoir, de façon simple, quelle variable rattachée à x utiliser. Il nous faut donc un identifiant unique pour x à la sortie de la conditionnelle.

Nous avons donc adopté la méthode suivante. Apres avoir généré toutes nos variables, et toutes nos contraintes liées aux instructions, nous générons un nouvel identifiant pour chaque variable se trouvant dans le membre gauche d'une affectation de la conditionnelle, et nous générons ici dans la branche "`THEN`". Avec l'exemple ci dessus, nous obtenons à la fin les environnements suivants :

$$\sigma_{generation-res} = \Big\{\big(x,(x_0,x_1,x_2,x_3\big),\big(z,(z_0)\big)\Big\}$$

$$\sigma_{evaluation-res} = \Big\{\big(x,(x_0,x_1,x_2,x_3\big),\big(z,(z_0)\big)\Big\}$$

Nous aurons la contrainte de spécification suivante :

$$(x_0 = 0 \wedge z_0 = 3) \Rightarrow (x_1 = x_0 + 1 \wedge \mathbf{x_3} = \mathbf{x_1})) \wedge \neg(x_0 = 0 \wedge z_0 = 3) \Rightarrow (x_2 = x_0 + 2 \wedge \mathbf{x_3} = \mathbf{x_2})$$

Notons que ce qui est dû à l'homogénéisation a été placé en gras.

De façon plus formelle, notons :

$$\begin{aligned}
\sigma_{generation1} &= (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_3\\
\sigma_{generation2} &= (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation1}\ c_{correction}})\downarrow_3\\
\sigma_{evaluation1} &= (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_4\\
\sigma_{evaluation2} &= (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation1}\ c_{correction}})\downarrow_4
\end{aligned}$$

On obtient donc en définitive l'équation suivante :

$$\begin{aligned}
&\mathcal{G}en_{Instr}\,[\![\,\texttt{IF condition THEN instruction1 ELSE instruction2 END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} =\\
&\Big((condition \Rightarrow\\
&\quad \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_1 \wedge\\
&\quad \forall x.(x \in dom(\sigma_{evaluation}) \wedge \sigma_{generation}x \neq \sigma_{generation2}x) \Rightarrow\\
&\qquad fresh(x, \sigma_{generation2}) = \sigma_{evaluation1}x)\\
&\wedge\\
&(\neg condition \Rightarrow\\
&\quad \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation1}\ c_{correction}})\downarrow_1 \wedge\\
&\quad \forall x.(x \in dom(\sigma_{evaluation}) \wedge \sigma_{generation}x \neq \sigma_{generation2}x) \Rightarrow\\
&\qquad fresh(x, \sigma_{generation2}) = \sigma_{evaluation2}x)\mathbf{,}
\end{aligned}$$

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}})\downarrow_2 \wedge\ \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction2}\,]\!]^{\sigma_{evaluation1}}_{\sigma_{generation1}\ c_{correction}})\downarrow_2\ \mathbf{,}$$

$$\begin{aligned}
&\sigma_{generation2}\big[\forall x.\big((x \in dom(\sigma_{evaluation}) \wedge \sigma_{generation}x \neq \sigma_{generation2}x) \Rightarrow\\
&\qquad x : fresh(x, \sigma_{generation2})\big)\big]\mathbf{,}
\end{aligned}$$

$$\begin{aligned}
&\sigma_{evaluation}\big[\forall x.\big((x \in dom(\sigma_{evaluation}) \wedge \sigma_{generation}x \neq \sigma_{generation2}x) \Rightarrow\\
&\qquad x : fresh(x, \sigma_{generation2})\big)\big]\Big)
\end{aligned}$$

exemple Soient $\sigma_{generation}$ et $\sigma_{evaluation}$ tels que :

$$\begin{aligned}
\sigma_{generation} &= \Big\{\big(x,(x_0)\big),\big(y,(y_0,y_1)\big),\big(z,(z_0,z_1)\big),big(t,(t_0,t_1,t_2))\Big\}\\
\sigma_{evaluation} &= \Big\{\big(x,(x_0)\big),\big(y,(y_0,y_1)\big),\big(z,(z_0)\big),big(t,(t_0,t_2))\Big\}
\end{aligned}$$

Soit $c_{correction}$ tel que $c_{correction}$= *true*.

Soit l'instruction conditionnelle suivante :

```
IF x=0 ∧ z=3 THEN
    y := y+x
ELSE
    t := t+y+3
END
```

Voici les différentes composantes du résultat :

$$(\mathcal{G}en_{Instr}[\![\texttt{IF x=0 THEN y := y+x ELSE t := t+y+3 END}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}})\downarrow_1 =$$
$$\big(x_0 = 0 \Rightarrow (y_2 = y_1 + 1 \wedge y_3 = y_2 \wedge t_4 = t_2) \wedge$$
$$\neg(x_0 = 0) \Rightarrow (t_3 = t_2 + y_1 + 3 \wedge y_3 = y_1 \wedge t_4 = t_3)\big)$$
$$(\mathcal{G}en_{Instr}[\![\texttt{IF x=0 THEN y := y+x ELSE t := t+y+3 END}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}})\downarrow_2 = \mathit{true}$$
$$(\mathcal{G}en_{Instr}[\![\texttt{IF x=0 THEN y := y+x ELSE t := t+y+3 END}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}})\downarrow_3 =$$
$$\{(x,(x_0),(y,(y_0,y_1,y_2,y_3)),(t,(t_0,t_1,t_2,t_3,t_4))\}$$
$$(\mathcal{G}en_{Instr}[\![\texttt{IF x = 0 THEN y := y+x ELSE t := t+y+3 END}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}})\downarrow_4 =$$
$$\{(x,(x_0),(y,(y_0,y_1,y_3)),(t,(t_0,t_1,t_2,t_4))\}$$

Conditionnelle "IF THEN ELSEIF...ELSE"

Ce genre de conditionnelle s'écrit de la façon suivante : Cette instruction s'écrit :

```
IF condition_1 THEN
    instruction_1
ELSEIF condition_2 THEN
    instruction_2
...
ELSEIF condition_N THEN
    instruction_N
ELSE
    instruction_{N+1}
END
```

Ce genre de conditionnelle se traite de la façon suivante :

```
IF condition_1 THEN
    instruction_1
ELSE
    IF condition_2 THEN
        instruction_2
    ELSE
...
            IF condition_N THEN
                 instruction_N
            ELSE
                instruction_{N+1}
            END
...
    END
END
```

Var... in... end

Ce genre d'instruction s'écrit en B de la façon suivante :

```
VAR
     Id_1, ... Id_n
IN
   instruction
END
```

On a l'équation suivante :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{VAR id IN instruction END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}}$$

En effet, les variables locales devront être affectées avant leur première utilisation. Il y aura donc sur ces variables toutes les contraintes liées à l'affectation.

On pourrait penser que des problèmes de masquage pourraient exister si plusieurs variables locales portent le même nom, dans des instructions différentes. Cependant, grâce au mécanisme de génération de variables, ce ne sera pas le cas.

Assert

Ce genre d'instructions s'écrit de la façon suivante :

```
ASSERT
     condition
THEN
   instruction
END
```

C'est ici notamment que se justifie l'utilité du deuxième argument et du deuxième élément du résultat de notre fonction.

En effet, il nous faut vérifier que lors de l'exécution de l'instruction `ASSERT`, aucun problème n'aura lieu, et que `condition` sera vérifiée.

Si nous nous contentons de placer `condition` dans notre contrainte de spécification, nous voyons bien quels problèmes cela peut poser : pour peu que `condition` ne soit pas respectée, nous aurons un ensemble de solutions vide, qui nous inciterait à penser que le code est correctement écrit, puisque aucun contre-exemple n'a pu être trouvé.

Nous avons donc adopté la solution qui consiste à placer `condition` dans l'ensemble des formules de correction du code.

Nous obtenons donc en définitive l'équation suivante :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{ASSERT condition THEN instruction END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}^{} = \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}^{}$$

While

Cette instruction s'écrit en B de la façon suivante :

```
WHILE condition DO
    instruction
VARIANT var
INVARIANT inv
END
```

Dans une boucle dont la sémantique est correcte, plusieurs choses doivent être vérifiées :
- l'invariant doit être vérifié dans l'environnement d'évaluation initial,
- si l'invariant est vérifié dans l'environnement d'évaluation de départ, il doit être vérifié après l'exécution de `instruction`,
- le variant doit être un nombre naturel,
- le variant dans l'environnement d'évaluation de départ doit être supérieur au variant dans l'environnement d'évaluation après l'exécution de `instruction`.

Nous considérons qu'implicitement, le variant sera toujours un nombre naturel. Il faudra donc compléter $c_{correction}$ par trois contraintes uniquement.

Nous ne savons pas à l'avance combien de passages dans la boucle seront effectués, les seules hypothèses que nous pouvons faire sur l'état de l'environnement après la boucle sont : `cond` ne sera pas vérifiée et l'invariant sera vérifié. C'est la seule contrainte de spécification que nous pouvons poser.

Notons :

$$\sigma_{generation1} = (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}^{}_{c_{correction}}) \downarrow_3$$

$$\sigma_{evaluation1} = (\mathcal{G}en_{Instr}\,[\![\,\texttt{instruction}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}^{}_{c_{correction}}) \downarrow_4$$

$$\sigma_{generation2} = \sigma_{generation1}[\forall x.\sigma_{generation}x \neq \sigma_{generation1}x \Rightarrow x : fresh(x, \sigma_{generation})]$$

$$\sigma_{evaluation2} = \sigma_{evaluation}[\forall x.\sigma_{generation}x \neq \sigma_{generation1}x \Rightarrow x : fresh(x, \sigma_{generation})]$$

Nous traduisons cela par les équations suivantes :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{WHILE cond DO instr VARIANT var INVARIANT inv END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} \downarrow_1 =$$
$$\big(\mathcal{G}en_{Expr}\,[\![\,\texttt{cond}\,]\!]\,\sigma_{evaluation} \Rightarrow$$
$$(\mathcal{G}en_{Expr}\,[\![\,\texttt{inv} \wedge \neg\texttt{cond}\,]\!]\,\sigma_{evaluation2})\big)$$
$$\wedge$$
$$\big((\neg\ \mathcal{G}en_{Expr}\,[\![\,\texttt{cond}\,]\!]\,\sigma_{evaluation}) \Rightarrow$$
$$(\forall x.\sigma_{generation1}x \neq \sigma_{generation1}x \Rightarrow fresh(x, \sigma_{generation}) = \sigma_{evaluation}x)\big)$$

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{WHILE cond DO instr VARIANT var INVARIANT inv END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} \downarrow_2 =$$
$$c_{correction}$$
$$\wedge\, \mathcal{G}en_{Expr}\,[\![\,\texttt{cond}\,]\!]\,\sigma_{evaluation} \Rightarrow$$
$$(\mathcal{G}en_{Expr}\,[\![\,\texttt{inv}\,]\!]\,\sigma_{evaluation}$$
$$\wedge\ (\mathcal{G}en_{Expr}\,[\![\,\texttt{inv}\,]\!]\,\sigma_{evaluation} \Rightarrow \mathcal{G}en_{Expr}\,[\![\,\texttt{inv}\,]\!]\,\sigma_{evaluation1})$$
$$\wedge\ \mathcal{G}en_{Expr}\,[\![\,\texttt{var}\,]\!]\,\sigma_{evaluation1} < \mathcal{G}en_{Expr}\,[\![\,\texttt{var}\,]\!]\,\sigma_{evaluation}$$
$$\wedge \mathcal{G}en_{Instr}\,[\![\,\texttt{instruction}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} \downarrow_2)$$

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{WHILE cond DO instr VARIANT var INVARIANT inv END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} \downarrow_3 =$$
$$\sigma_{generation2}$$

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{WHILE cond DO instr VARIANT var INVARIANT inv END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} \downarrow_4 =$$
$$\sigma_{evaluation2}$$

exemple : Soient $\sigma_{generation}$ et $\sigma_{evaluation}$ tels que :

$$\sigma = \Big\{\big(x, (x_0)\big)\Big\}$$

Soit $c_{correction}$ tel que $c_{correction} = true$.

Soit la boucle suivante :

```
WHILE x < 20 DO
    x := x + 1
VARIANT 20 - x
INVARIANT x ≤ 20
END
```

On a alors :

$$\mathcal{G}en_{Instr}\left[\!\left[\begin{array}{l}\texttt{WHILE x < 20 DO x := x + 1 VARIANT 20}\\ \texttt{- x INVARIANT x < 30 END}\end{array}\right]\!\right]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_1 =$$
$$x_0 < 20 \Rightarrow (x_1 = x_0 + 1 \wedge (x_2 \leq 20 \wedge \neg(x_2 < 20)))$$
$$\wedge\neg(x_0 < 20) \Rightarrow x_2 = x_0$$

$$\mathcal{G}en_{Instr}\left[\!\left[\begin{array}{l}\texttt{WHILE x < 20 DO x := x + 1 VARIANT 20}\\ \texttt{- x INVARIANT x < 30 END}\end{array}\right]\!\right]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_2 =$$
$$x_0 < 20 \Rightarrow (x_0 < 30 \wedge (x_0 < 30 \Rightarrow x_1 < 30) \wedge 20 - x_1 < 20 - x_0)$$

$$\mathcal{G}en_{Instr}\left[\!\left[\begin{array}{l}\texttt{WHILE x < 20 DO x := x + 1 VARIANT 20}\\ \texttt{- x INVARIANT x < 30 END}\end{array}\right]\!\right]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_3 = \{(x, (x_0, x_1, x_2))\}$$

$$\mathcal{G}en_{Instr}\left[\!\left[\begin{array}{l}\texttt{WHILE x < 20 DO x := x + 1 VARIANT 20}\\ \texttt{- x INVARIANT x < 30 END}\end{array}\right]\!\right]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_4 = \{(x, (x_0, x_2))\}$$

6.7 Exemple

Soit l'instruction `instr` suivante :

```
x := 0 ; y := 2 ; z := 4 ; t := 8 ;
ASSERT x ≥ 0 THEN
    IF y ≥ 2 THEN
        x := x+3 ; z := z+2
    ELSE
        x := x+2 ; y := y+1 ;
        WHILE z < 10 DO
            z := z + 1
        VARIANT 10 - z
        INVARIANT z ≤ 10
        END
    END
END ;
z :=z+1
```

avec $\sigma_{generation}$ et $\sigma_{evaluation}$ tels que :

$$\sigma_{generation} = \Big\{\big(x, (x_0,)\big), \big(y, (y_0)\big), \big(z, (z_0)\big), \big(t, (t_0)\big)\Big\}$$
$$\sigma_{generation} = \Big\{\big(x, (x_0,)\big), \big(y, (y_0)\big), \big(z, (z_0)\big), \big(t, (t_0)\big)\Big\}$$

Remplaçons dans `instr` les identificateurs B par les variables générées qui leur sont at-

tribuées. Nous obtenons :

```
x1 := 0 ; y1 := 2 ; z1 := 4 ; t1 := 8 ;
ASSERT x1 ≥ 0 THEN
    IF y1 ≥ 2 THEN
        x2 := x1+3 ; z2 := z1+2
    ELSE
        x3 := x1+2 ; y2 := y1+1 ;
        WHILE z1 < 10 DO
            z3 := z1 + 1
        VARIANT 10 - z
        INVARIANT z ≤ 10
        END
    END
END ;
z6 :=z5+1
```

$\mathcal{G}en_{Instr}[\![\ ]\!]$ nous donne le résultat suivant :

$$\mathcal{G}en_{Instr}[\![\texttt{instr}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_1 =$$
$$x_1 = 0 \wedge y_1 = 2 \wedge z_1 = 4 \wedge t_1 = 8$$
$$\wedge y_1 \leq 2 \Rightarrow (x_2 = x_1 + 3 \wedge z_2 = z_1 + 2$$
$$\wedge x_4 = x_2 \wedge y_3 = y_1 \wedge z_5 = z_2)$$
$$\wedge\neg\ (y_1 \geq 2) \Rightarrow (x_3 = x_1 + 2 \wedge y_2 = y_1 + 1$$
$$\wedge z_1 < 10 \Rightarrow z_3 = z_1 + 1 \wedge z_4 \leq 10 \wedge \neg(z_4 < 10)$$
$$\wedge\neg(z_1 < 10) \Rightarrow z_4 = z_1$$
$$\wedge x_4 = x_3 \wedge y_3 = y_2 \wedge z_5 = z_4)$$
$$\wedge z_6 = z_5 + 1$$

$$\mathcal{G}en_{Instr}[\![\texttt{I}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_2 =$$
$$(z_1 < 10 \Rightarrow (z_1 \leq 10 \wedge z_3 \leq 10 \wedge (10 - z_1 > 10 - z_3)))$$

$$\mathcal{G}en_{Instr}[\![\texttt{I}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_3 =$$
$$\{(x,(x_0,x_1,x_2,x_3,x_4)),(y,(y_0,y_1,y_2,y_3,y_4),(z,(z_0,z_1,z_2,z_3,z_4,z_5,z_6),(t,(t_0,t_1))\}$$

$$\mathcal{G}en_{Instr}[\![\texttt{I}]\!]^{\sigma_{evaluation}}_{\sigma_{generation}}{}_{c_{correction}} \downarrow_4 =$$
$$\{(x,(x_0,x_1,x_4)),(y,(y_0,y_1,y_3),(z,(z_0,z_1,z_5,z_6),(t,(t_0,t_1))\}$$

Chapitre 7

Contraintes dues à la machine abstraite

Après avoir exposé au chapitre précédent comment sont obtenues les contraintes dues à chaque opération de l'implantation, nous voulons ici exposer comment seront obtenues les contraintes dues à chaque opération de la machine abstraite.

Ces contraintes représenteront les post-conditions de l'opération analysée dans la machine abstraite.

Nous adopterons le même plan qu'au chapitre précédent.
Nous donnerons d'abord la forme des substitutions dans une machine abstraite.
Puis, comme les contraintes relatives à une opération dépendent directement des substitutions composant le corps de l'opération, nous allons analyser une à une les substitutions de B auxquelles nous nous sommes restreints, afin de détailler les contraintes produites pour chacune de ces substitutions.

7.1 La génération de nouvelles variables

Ce principe est à nouveau utilisé pour le traitement des opérations des machines abstraites, de la même façon que pour les implantations.
Avec les substitutions auxquelles nous nous sommes restreints [1], la génération de nouvelles variables est inutile.
Cependant, il faut bien comprendre que cet outil est destiné à évoluer, et que cette décision se justifiera aisément dès que Vics acceptera d'autres substitutions dans la machine abstraite (la substitution IF, par exemple)

7.2 Notations utilisées

Les notations utilisées sont les mêmes que dans le chapitre précédent.

[1] Il n'y a pas de séquence dans les machines abstraite, et Vics n'accepte pas encore les conditionnelles.

7.3 Description des substitutions

Voici les formes que peuvent prendre les substitutions d'une machine abstraite, telles qu'elles sont acceptées par Vics.

⟨*Substitution*⟩ ::= 'PRE' ⟨*condition*⟩'THEN' ⟨*Autre_substitution*⟩ 'END'
| ⟨*Autre_substitution*⟩

⟨*Autre_substitution*⟩ ::= ⟨*Ident*⟩ ':=' ⟨*Expr*⟩
| ⟨*Ident*⟩ ':(' ⟨*Predicat*⟩')'
| ⟨*Autre_substitution*⟩ '||'⟨*Autre_substitution*⟩

$< Condition >$, $< Predicat >$ et $< Ident >$ ayant les mêmes formes que dans le chapitre précédent, nous n'avons pas jugé utile de détailler à nouveau leurs règles de production.

7.4 Le traitement des expressions

Nous nous servirons de la fonction définie dans le chapitre précédent.

En effet, la forme des expressions n'a pas changé, il n'y a donc pas besoin de redéfinir la fonction $\mathcal{G}en_{Expr}[\![\ ]\!]$.

7.5 Les contraintes dues aux substitutions

7.5.1 Le traitement des substitutions

Là encore, nous nous servirons de la fonction décrite dans le chapitre précédent.

Plutôt que de définir une nouvelle fonction de génération de contraintes, nous avons décidé pour raisons pratiques de réutiliser la fonction $\mathcal{G}en_{Instr}[\![-]\!]$.

Pour cela, nous devons donc respecter la signature de cette fonction, même si certains paramètres et résultats peuvent paraître inutiles.
En effet, au vu des substitutions acceptées par Vics, l'ensemble des contraintes de correction sera toujours vide, et l'environnement d'évaluation sera toujours identique à l'environnement de génération.

Cependant, cet outil devra évoluer et accepter à terme d'autres substitutions dans la machine abstraite, et le besoin de ces paramètres se fera alors sentir (notamment pour la substitution IF).

Il nous faut donc compléter la définition de notre fonction $\mathcal{G}en_{Instr}[\![-]\!]$ par les équations relatives aux substitutions acceptées par Vics. Analysons une à une ces substitutions.

Substitution "PRE ... THEN ... END"

Cette substitution s'écrit en B :

```
PRE condition THEN
    substitution
END
```

Nous avons vu précédemment que nous ne nous préoccupons pas des préconditions. Nous avons donc l'équation suivante :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{PRE condition THEN substitution END}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}}$$

Substitution "devient_tel_que"

Cette substitution s'écrit en B de la façon suivante :

```
Ident :(Predicat)
```

Pour traiter cette instruction, il va falloir :
- générer une nouvelle variable et la rattacher à `Ident`,
- générer la contrainte signifiant que cette variable générée doit respecter `Predicat`
- compléter $\sigma_{generation}$
- compléter $\sigma_{evaluation}$

Nous avons donc :

$$\begin{aligned}\mathcal{G}en_{Instr}\,[\![\,\texttt{id :(Predicat)}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{correction}} = \\ \big(fresh(\texttt{id}, \sigma_{generation}) = \mathcal{G}en_{Expr}\,[\![\,\texttt{Predicat}\,]\!]\,\sigma_{evaluation}[\texttt{id} : fresh(\mathbf{id}, \sigma_{evaluation})],\\ c_{correction},\\ \sigma_{generation}[\texttt{id} : fresh(\texttt{id}, \sigma_{generation})],\\ \sigma_{evaluation}[\texttt{id} : fresh(\texttt{id}, \sigma_{generation})]\big)\end{aligned}$$

Substitutions en parallèle

Une parallèle de substitutions est écrite en B de la façon suivante :

```
substitution1 || substitution2
```

En B, `substitution1` et `substitution2` ne peuvent modifier les mêmes variables. Notons :

$$\sigma_{generation1} = \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{Correction}}\ \downarrow_3$$

$$\sigma_{evaluation1} = \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{Correction}}\ \downarrow_4$$

$$\sigma_{generation2} = \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{Correction}}\ \downarrow_3$$

$$\sigma_{evaluation2} = \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\ c_{Correction}}\ \downarrow_4$$

Nous obtenons donc en définitive l'équation suivante :

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{substitution1 ; substitution2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\,c_{Correction}} =$$

$$\big(\mathcal{G}en_{Instr}\,[\![\,\texttt{substitution1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\,c_{Correction}}\ \downarrow_1 \wedge\ \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\,c_{Correction}}\ \downarrow_1,$$

$$\mathcal{G}en_{Instr}\,[\![\,\texttt{substitution1}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\,c_{Correction}}\ \downarrow_2 \wedge\ \mathcal{G}en_{Instr}\,[\![\,\texttt{substitution2}\,]\!]^{\sigma_{evaluation}}_{\sigma_{generation}\,c_{Correction}}\ \downarrow_2,$$

$$\sigma_{generation}[\forall x.x \in dom(\sigma_{generation1}) \wedge x \in dom(\sigma_{generation1}) \wedge \sigma_{generation1}x \neq \sigma_{generation2}x \Rightarrow$$
$$((Size(\sigma_{generation1})x > Size(\sigma_{generation2})x \Rightarrow x : \sigma_{generation1}x)$$
$$\wedge(Size(\sigma_{generation2})x > Size(\sigma_{generation1}x \Rightarrow x : \sigma_{generation2}x)))],$$

$$\sigma_{generation}[\forall x.x \in dom(\sigma_{evaluation1}) \wedge x \in dom(\sigma_{evaluation1}) \wedge \sigma_{evaluation1}x \neq \sigma_{evaluation2}x \Rightarrow$$
$$((Size(\sigma_{evaluation1})x > Size(\sigma_{evaluation2})x \Rightarrow x : \sigma_{evaluation1}x)$$
$$\wedge(Size(\sigma_{evaluation2})x > Size(\sigma_{evaluation1}x \Rightarrow x : \sigma_{evaluation2}x)))]\big)$$

Conclusion

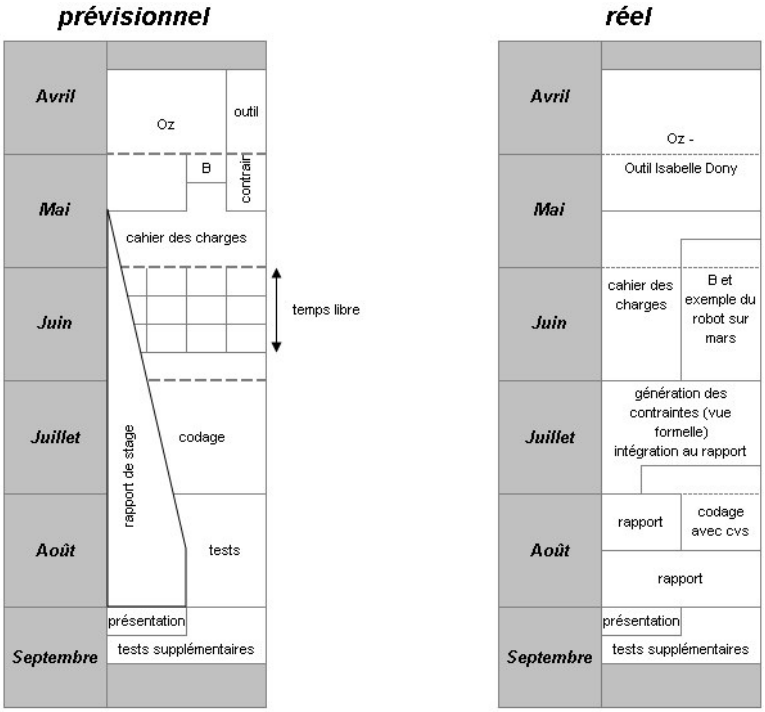
A l'heure actuelle, l'outil réalisé n'est pas encore assez développé pour servir d'aide à un enseignement. Le langage B a été restreint de façon trop stricte pour que des machines abstraites intéressantes puissent être écrites. En effet, ni les quantificateurs, ni les tableaux n'ont été pris en compte dans cette première version de Vics, à cause de la durée limitée du stage.
Dans les prochaines versions de l'outil, ces lacunes seront comblées.

Tout au long de ce rapport, nous nous sommes attachés à montrer une grande partie du cheminement qui nous a permis d'aboutir au résultat final. Cependant, nous n'avons pas jugé bon de détailler la phase d'implantation. En effet, celle-ci ne fait que reprendre les étapes détaillées au long de ce rapport. Faute de temps, l'aspect "programmation par contraintes", qui aurait pu faire l'objet d'un chapitre particulier, n'a pu être approfondi .
Cependant ce stage nous a permis de mieux appréhender les connaissances acquises tout au long de cette année, notamment en logique du premier ordre et en sémantique, et de nous sensibiliser à certains aspects propres à l'ingénierie des exigences.
En définitive, ce stage aura été une expérience très enrichissante, tant sur le plan technique que sur le plan humain.

Annexe A

Planning

Planning des tâches



Annexe B

Cahier des charges

- cahier des charges -

TEP

version	**date**	**auteur**	**parties modifiees**	**commentaires**
1.0	18/06/2003	L. Moussa	tout	création du document

Table des matières

1 Introduction **4**
- 1.1 But . . . 4
- 1.2 Portée . . . 4
- 1.3 Définitions, acronymes et abréviations . . . 5
- 1.4 Références . . . 6
- 1.5 Vue d'ensemble . . . 6

2 Description générale **8**
- 2.1 Environnement . . . 8
- 2.2 Fonctionnalités de l'outil . . . 8
  - 2.2.1 Correction syntaxique . . . 8
  - 2.2.2 Correction algorithmique . . . 8
  - 2.2.3 Génération de Contre-exemples . . . 9
  - 2.2.4 Génération de code . . . 9
  - 2.2.5 Aide à l'utilisateur . . . 9
  - 2.2.6 Configuration de l'outil . . . 9
- 2.3 Caractéristiques des utilisateurs . . . 9
- 2.4 Contraintes . . . 9
  - 2.4.1 Contraintes relatives aux entrées . . . 9
  - 2.4.2 Contraintes relatives aux types des données . . . 11
  - 2.4.3 Contraintes relatives à l'installation de l'outil . . . 12
  - 2.4.4 Contraintes relatives aux performances . . . 12
- 2.5 Scenarii d'exécution . . . 12
- 2.6 Hypothèses et dépendances . . . 16
- 2.7 Améliorations à apporter . . . 16

3 Exigences spécifiques **17**
- 3.1 Exigences spécifiques sur les interfaces externes . . . 17
  - 3.1.1 Interfaces utilisateurs . . . 17
  - 3.1.2 Interfaces logicielles . . . 17
  - 3.1.3 Interfaces materielles . . . 17
  - 3.1.4 Communication entre interfaces . . . 17
- 3.2 Exigences spécifiques aux fonctionnalités de l'outil . . . 17
  - 3.2.1 Exigences spécifiques pour l'interface de TEP . . . 17
  - 3.2.2 Exigences spécifiques pour la correction syntaxique . . . 18
  - 3.2.3 Exigences spécifiques pour la correction algorithmique . . . 18
  - 3.2.4 Exigences spécifiques pour la génération de code . . . 19
  - 3.2.5 Exigences spécifiques pour l'aide à l'utilisateur . . . 19

4 Annexes **20**
4.1 Précisions sur la méthode B 20

Chapitre 1

Introduction

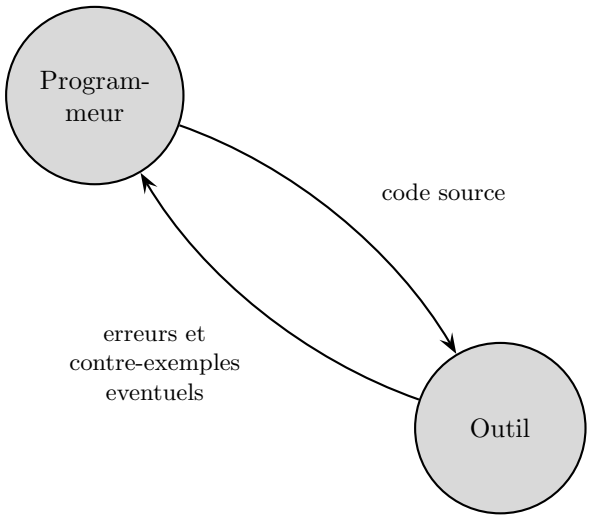
1.1 But

Notre but est d'ici de concevoir et créer un outil dont le rôle sera d'analyser un programme écrit en langage B, et d'y découvrir d'éventuelles erreurs de programmation.

Cet outil sera appelé TEP

1.2 Portée

Tep sera utilisé par des élèves de deuxième candidature, qui ont déjà fait un minimum de programmation, afin de tester leurs programmes.



1.3 Définitions, acronymes et abréviations

mot	**définition**
Tep	TEP signifie Trouver des Erreurs dans un Programme, c'est le nom provisoire de l'outil.
Prédicat	Expression logique dont la valeur peut être vraie ou fausse selon la valeur des arguments. **Exemple :** $x * y = 1$ ou bien $b <= 0$
Assertion	Prédicat destiné à détecter un état anormal causé par une erreur de programmation. Un assertion peut-être placée dans une précondition, dans une postcondition, ou à un point de programme particulier. Attention à ne pas confondre la définition ici présentée avec l'instruction "ASSERT" du langage B.
Précondition	Assertion placée dans le code source de facon à ce qu'elle soit évaluée juste avant une fonction. Si la précondition n'est pas verifiée, la fonction ne peut pas être appellée.
Postcondition	Assertion placée dans le code source de façon à ce qu'elle soit évaluée juste avant un point de sortie d'une fonction. Si la post-condition n'est pas verifiée, la fonction (ou la post-condition) est probablement mal écrite.
Invariant	Propriété qui est conservée sous l'effet d'une instruction, ou d'une suite d'instructions. **Exemple :** *les invariants de boucle*
Contrainte	Formule logique qui contient des variables, et qui définit une relation qui doit être satisfaite par les valeurs de ces variables. **Exemple :** $3 * x + 2 * y = 5 * t$, où x, y et t sont des variables.
B	La méthode B est une méthode formelle permettant le développement de logiciels sûrs. Pour plus de précisions, nous vous invitons à consulter l'annexe, ou/et l'URL suivante : http://www3.inrets.fr/estas/mariano/documents/These/html/These002.html
B0	Sous-ensemble du langage B comportant un ensemble réduit d'instructions possédant chacune une traduction dans un langage impératif (C, ADA, Modula2, etc).

mot	définition
Programmation par contraintes	Paradigme de programmation apparu dans les années 80. L'utilisateur spécifie des contraintes sur les variables, et par réduction du domaine de définition, on obtient l'ensemble solution.
Distribution	Pour résoudre un probleme P, on peut écrire une contraite C, et essayer de résoudre P et C, puis P et non (C). On dit a lors qu'on a distribué P avec C.
Mozart	Mozart est une plate-forme avancée pour le développement d'applications réparties intelligentes. Mozart est basée sur un langage multi-paradigme, Oz.

1.4 Références

Isabelle Dony, de l'Université Catholique de Louvain-La-Neuve, a déjà crée un outil qui analyse un programme impératif, et retourne les éventuelles erreurs de ce programme. Elle a très aimablement accepté de nous faire partager son expérience, et a mis à notre disposition les documents et le code source qu'elle avait déjà écrits.

Afin de mieux maîtriser le langage B, différents documents relatifs à la methode B pourront être consultés (par exemple, le B-Book de Jean-Raymond Abrial)

1.5 Vue d'ensemble

Afin de mieux comprendre comment sera utilisé cet outil, il est nécessaire de savoir comment est constitué un projet écrit en B.

Chaque projet est constitué de plusieurs fichiers. Il y a obligatoirement une machine abstraite, qui décrit les entités utilisées, et donne leurs initialisations. Cette machine contient aussi la signature des opérations ainsi que les spécifications de leur comportement. Cette machine est raffinée une ou plusieurs fois, jusqu'à obtenir une implantation.

Il peut cependant y avoir plusieurs machines abstraites, et de ce fait, plusieurs implantations. Il y a dans chaque projet autant de modules que de machines abstraites : en effet, chaque ensemble machine abstraite - raffinenements - implantation peut-être considérée comme un module.

Chaque module peut être traduit en C ou en Ada, si l'implantation est écrite en code B0.

Pour plus de détails concernant l'atelier B et les raffinements, le manuel de référence de B ainsi que le manuel utilisateur de B pourront être consultés. De plus, quelques documents seront fournis dans l'Annexe.

Afin de simplifier le logiciel, nous avons choisi de n'analyser en entrée que deux fichiers : la machine abstraite et son implantation.

Nous voulons ici réaliser un outil permettant à un programmeur B débutant de *vérifier* la correction de son implantation vis-à-vis de la machine abstraite fournie.

L'idee générale est la suivante : le programmeur fournira à Tep une machine abstraite et son implantation. Il choisira alors une technique de distribution de variables ainsi qu'un niveau de precision pour les résultats qu'il souhaite obtenir.

Tep analysera alors les deux fichiers, et détectera alors les éventuelles violations d'assertion, dans le cas où la machine abstraite est mal implantée. Selon le niveau de précision choisi, Tep pourra donner différents résultats : un message indiquant si des assertions sont violées ou pas, le nom des opérations où ces assertions sont violées, à cause de quelles variables, des contre-exemples s'il y en a, ou encore l'arbre montrant les distributions successives appliquées aux différentes variables.

Chapitre 2

Description générale

2.1 Environnement

Notre logiciel ne vient en aucune façon compléter un système déjà existant. Il constitue un système à lui seul.

Les entités gravitant autour du système seront les élèves et leurs enseignants. Les uns et les autres inter-agiront avec notre système en lui fournissant un projet B à tester, ainsi que leurs différents choix quant aux détails souhaités. Ainsi, le projet B pourra être analysé : en supposant que notre machine abstraite est cohérente, TEP va tenter de vérifier que chaque opération de l'implantation correspond bien aux spécifications de l'opération correspondante dans la machine abstraite.

Notre outil, Tep devra être suffisamment convivial pour paraître attrayant à des élèves de deuxième année de candidature. Ces derniers pourront communiquer avec lui via une interface graphique.

Cet outil sera en forte interaction avec le Browser de Oz.

2.2 Fonctionnalités de l'outil

2.2.1 Correction syntaxique

Plusieurs types d'erreurs de syntaxe pourront être détectés : absence de clause obligatoire dans une machine abstraite ou dans une implantation, mauvaise orthographe du mot introducteur de la clause, nom de variable incorrect, expression mal formée...

TEP affichera alors un message d'erreur explicite.

2.2.2 Correction algorithmique

TEP détectera aussi les erreurs de programmation : en effet, si l'une des opérations viole les invariants de la machine abstraite et de l'implantation, une erreur sera signalée. Le niveau de précision du message d'erreur sera choisi par l'utilisateur. Il pourra obtenir :

– un simple message d'erreur indiquant qu'une erreur a été détectée,

- un message d'erreur précisant quelle est l'assertion qui est potentiellement violée,
- l'arbre de distribution des variables, afin de pouvoir localiser la variable fautive

2.2.3 Génération de Contre-exemples

Si un contre-exemple est trouvé, il sera possible de visualiser sa valeur.

2.2.4 Génération de code

Si l'implantation fournie est écrite en code B0, notre outil pourra, si l'utilisateur le lui demande, générer le code C correspondant. Cependant, si l'implantation n'est pas écrite en B0, aucun résultat ne sera garanti : l'outil ne vérifiera pas que le code fourni est bien du B0.

2.2.5 Aide à l'utilisateur

L'un des premiers souhaits émis par plusieurs utilisateurs a été "disposer d'une aide à l'utilisateur pertinente". malheureusement, faute de temps, cette fonctionnalité ne pourra être développée, faute de temps. Dans cette première version de l'outil, une aide sommaire à l'utilisateur (sous forme d'un fichier pdf) sera proposée, expliquant les différences entre toutes les techniques de distribution.

2.2.6 Configuration de l'outil

Un manuel d'installation devra être fourni à la livraison du produit.

2.3 Caractéristiques des utilisateurs

L'outil sera mis entre les mains de deux populations d'utilisateurs :

- les programmeurs B (débutants ou confirmés) connaissant aussi la programmation par contraintes, aptes à utiliser chaque technique de distribution avec pertinence,
- les programmeurs B (débutants ou confirmés) ne connaissant pas la programmation par contraintes (ou n'en ayant que quelques notions).

2.4 Contraintes

2.4.1 Contraintes relatives aux entrées

A chaque utilisation de TEP, un couple de fichiers devra lui etre fourni : une machine abstraite B, et son implantation.

Pour des raisons d'ergonomie, la totalité du langage B n'est pas analysée.

Cependant, afin que les machines analysées par TEP puissent éventuellement être prouvées avec l'atelier B ultérieurement, le langage supporté par TEP sera un sous ensemble du langage B accepté par la version 3.6 de l'atelier B. Il va donc falloir que les programmeurs se restreignent aux sous-ensembles donnés ci-dessous.

Instructions acceptées par TEP

Les machines abstraites acceptées par TEP seront écrites avec les instructions suivantes :

Skip
ident : = expr
ASSERT prop THEN inst END
IF prop THEN inst ELSEIF prop THEN inst ELSE prop END
CASE expr OF EITHER ident THEN inst OR ident THEN inst ELSE inst END

Les implantations acceptées par TEP seront quant à elles écrites avec les instructions données ci-dessus, mais aussi les instructions suivantes.

VAR ident IN inst END
inst ; inst
WHILE prop DO inst INVARIANT prop VARIANT expr END

Pour mieux connaître B, l'élève débutant consultera ses cours. Le programmeur plus averti pourra se référer à
METTRE REFERENCE

Opérateurs arithmétiques acceptés par TEP

Pour les mêmes raisons que précédemment, nous avons dû restreindre l'ensemble des opérateurs mathématiques disponibles en B. TEP n'acceptera que les opérateurs mathématiques suivants :

- +
- -
- *
- /
- >=
- <=

Opérateurs arithmétiques acceptés par TEP

l'ensembles des opérateurs ensemblistes accepté par TEP

- $\in$ (appartenance à un ensemble)
- $\subset$ (inclusion dans un ensemble)

Opérateurs booléens acceptés par TEP

Là encore, nous avons restreint l'ensemble des opérateurs booléens acceptés par l'atelier B.

$\vee$ (ou logique)	$\wedge$ (et logique)
$\Rightarrow$ (implique)	$\neg$ (not)
$\forall$ (quelque soit)	$\exists$ (il existe)

Clauses acceptées par TEP

Afin de simplifier l'outil, nous avons restreint les clauses acceptées par B, et avons choisi de ne garder que celles qui sont le plus couramment utilisées...

Voici les clauses de machines abstraites qui seront acceptées par TEP

CONSTRAINTS
SETS
CONCRETE_CONSTANTS
ABSTRACT_CONSTANTS
PROPERTIES
CONCRETE_VARIABLES
ABSTRACT_VARIABLES
INVARIANT
ASSERTIONS
INITIALISATIONS
OPERATIONS

Voici les clauses d'implantations qui seront acceptées par TEP :

IMPLEMENTATION
REFINES
CONSTRAINTS
VALUES SETS
CONCRETE_CONSTANTS
PROPERTIES
CONCRETE_VARIABLES
INVARIANT
ASSERTIONS
INITIALISATIONS
OPERATIONS

Il faut bien comprendre que TEP n'analysera qu'une machine abstraite et l'implantation lui correspondant. TEP ne prendra donc pas en compte les éventuels raffinements intermédiaires.

2.4.2 Contraintes relatives aux types des données

Les types acceptés seront les suivants :
- booléens
- entiers appartenant à l'intervalle 0..max (max reste à définir)

Les structures de données utilisées seront les suivantes :

– tableaux,
– enregistrements,
– arbres binaires.

2.4.3 Contraintes relatives à l'installation de l'outil

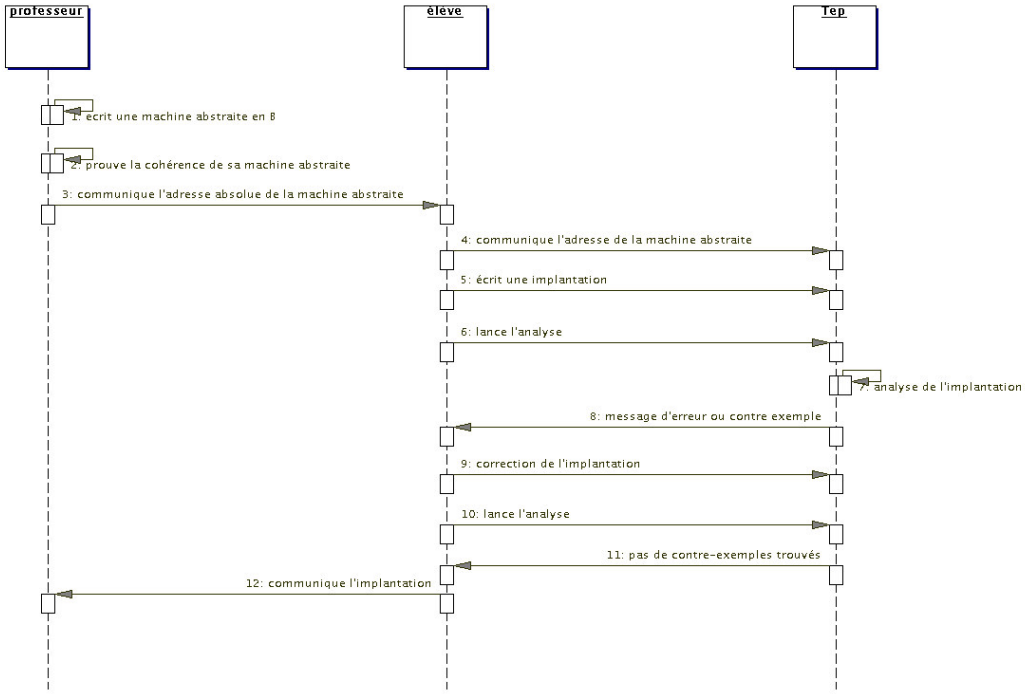
Comme cet outil sera développé en Oz, il faudra impérativement respecter toutes les contraintes d'installation de la plate-forme Mozart (avoir un système d'exploitation apte à supporter Mozart, avoir suffisamment de mémoire...)

2.4.4 Contraintes relatives aux performances

Afin d'améliorer les performances (pour ne pas avoir de temps d'analyse trop grands, nous avons dû restreindre les domaines de définition des entiers).
PRECISER : DONNER le MAX.

2.5 Scenarii d'exécution

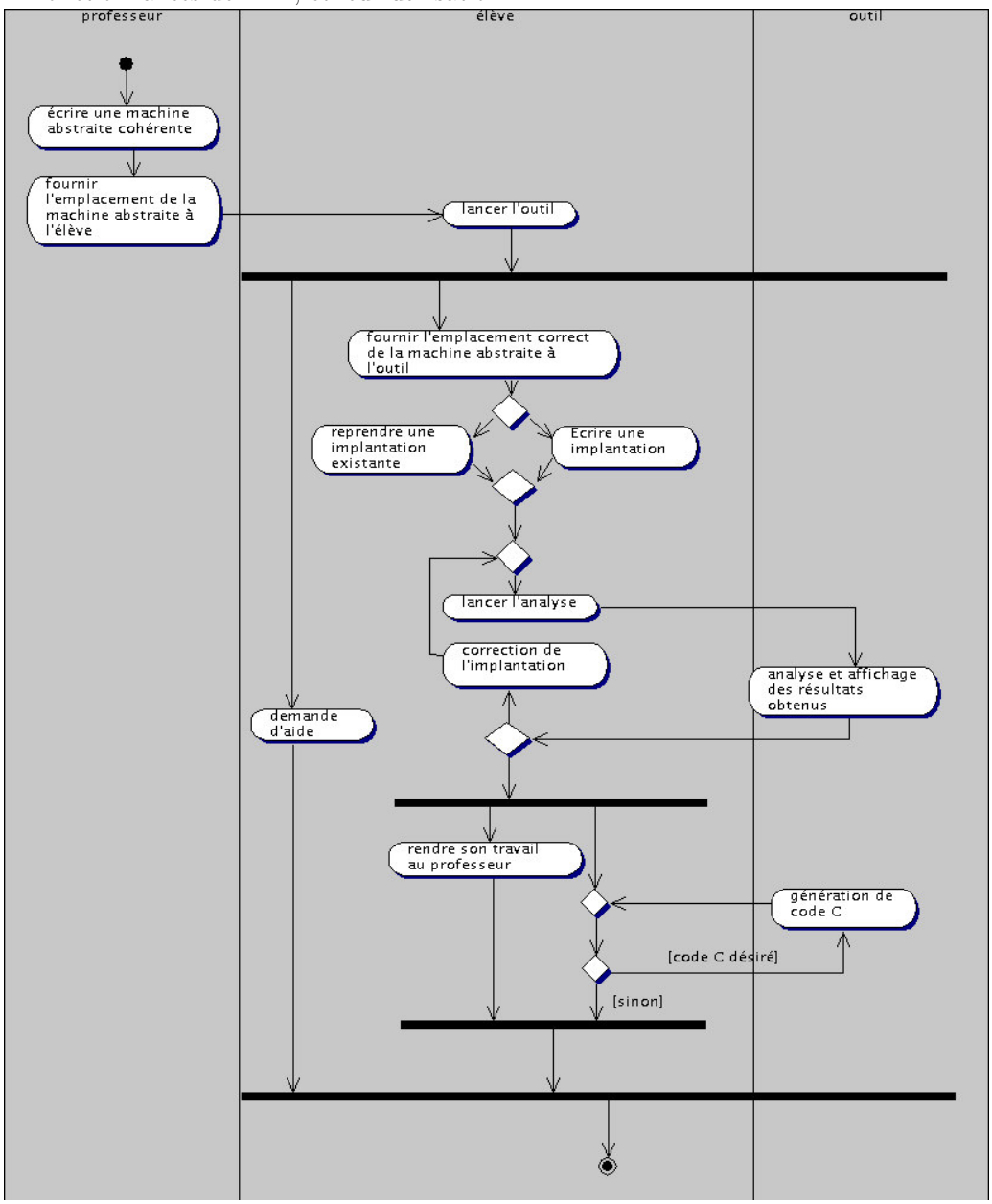
Voici un diagramme de séquence permettant de mieux comprendre le fonctionnement habituel de TEP.



Le professeur écrira une machine abstraite dont il prouvera la cohérence. Il la fournira ensuite à l'élève. Celui-ci la fournira alors à TEP, et s'efforcera d'écrire une implantation qui y correspond. Ces deux fichiers seront ensuite analysés par TEP. Lors de l'analyse, l'outil cherchera d'éventuelles erreurs de syntaxe et contre-exemples. S'il en trouve, il le signalera à l'élève par un message d'erreur. L'élève réécrira alors son implantation qui sera une nouvelle fois analysée. Une

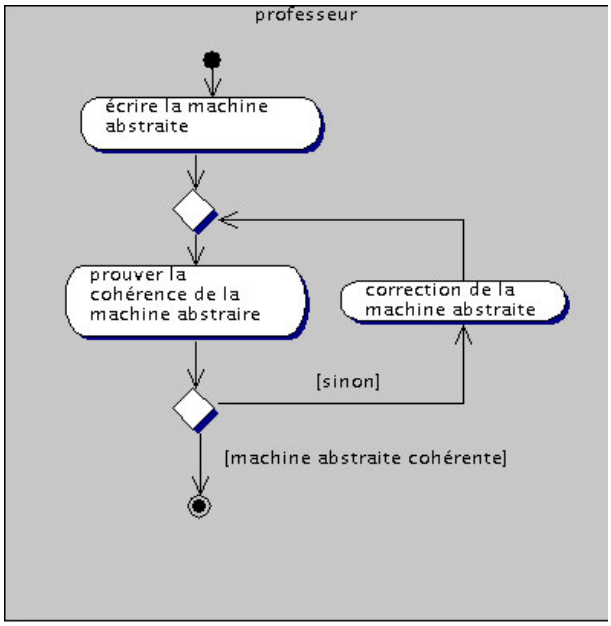
fois l'implantation vérifiée par TEP, l'élève pourra alors rendre son travail à son professeur.

Voici un diagramme d'activité montrant de façon plus complète les différentes fonctionnalités de TEP, et leur utilisation.

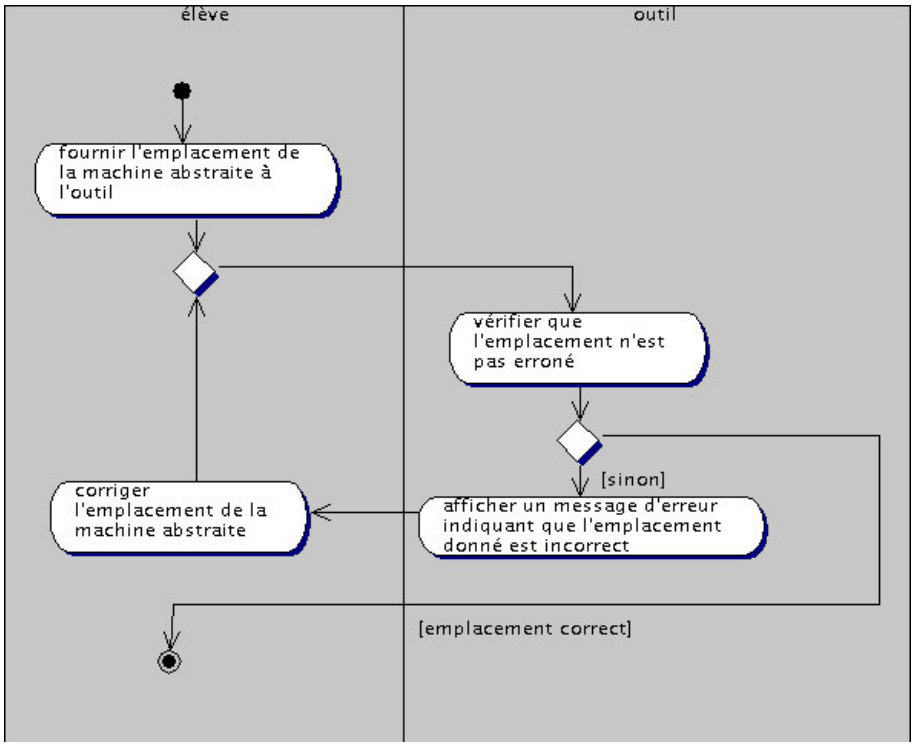


Pour plus de clarté, voici les diagrammes raffinant les différentes activités proposées dans le diagramme ci-avant.

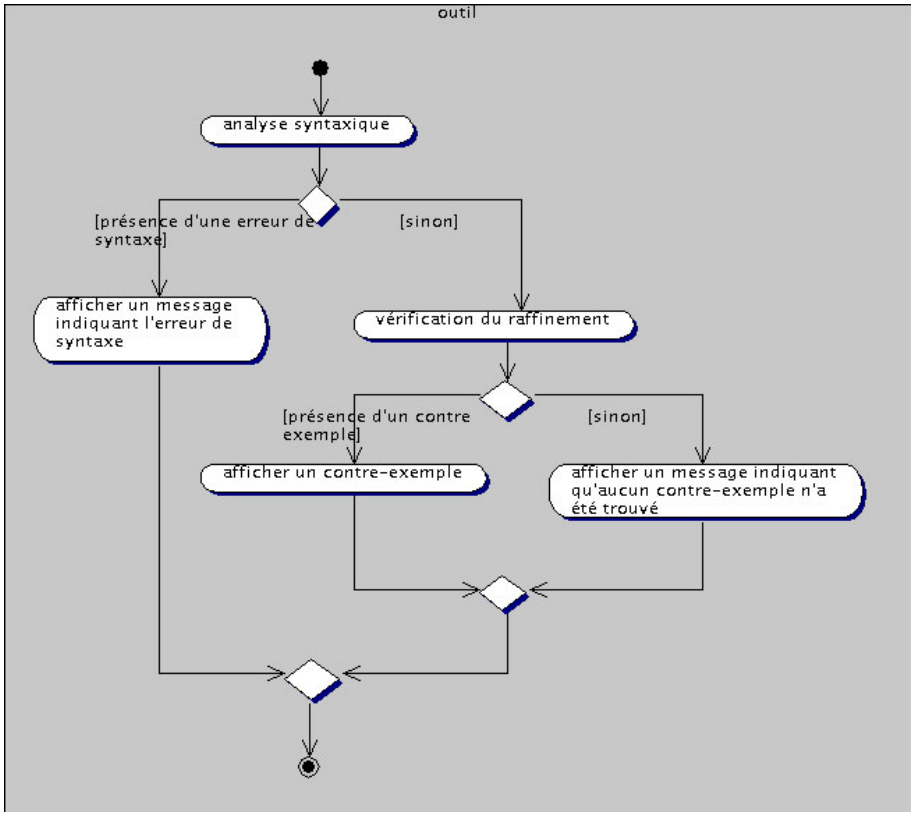
Voici tout d'abord le diagramme raffinant l'écriture de la machine abstraite par le professeur.



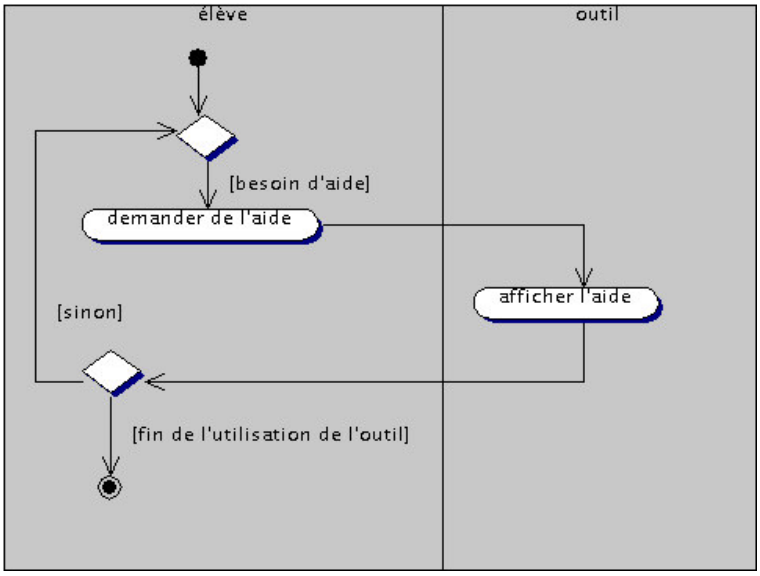
Voici le diagramme permettant de mieux comprendre comment se comporte l'outil lorsqu'on lui communique l'emplacement de la machine abstraite.



Voici le détail de l'analyse par l'outil.



Voici comment se fait l'aide à l'utilisateur.



2.6 Hypothèses et dépendances

Nous supposerons que toutes les machines abstraites fournies à TEP seront cohérentes.

De plus, nous supposerons que chaque opération abstraite sera écrite sous la forme de plusieurs affectations effectuées en parallèle.

2.7 Améliorations à apporter

Dans un premier temps, notre première préoccupation a été de créer un outil permettant de vérifier qu'une machine abstraite donnée était bien raffinée par l'implantation fournie par l'utilisateur. Il sera très certainement possible de rendre meilleures les performances de cet outil, ainsi que la qualité de ses fonctionnalités. Cependant, comme le but du stage était avant tout de créer un noyau stable pouvant servir de base à la suite, nous ne nous sommes pas penchés sur certains problèmes.

Voici quelques points qui pourront être améliorés. Le problème des investigations sur domaines trop restreints devra être résolu. Les performances et temps de réponse de TEP ne seront pas toujours optimales, dans cette première version. Dans une prochaine version, ceci pourrait être amélioré. Le langage supporté par TEP pourrait aussi être augmenté. L'aide à l'utilisateur devrait aussi être grandement améliorée, ainsi que l'analyse syntaxique des fichiers sources, et le signal d'erreur renvoyé qui pourrait être plus pertinent.

Chapitre 3

Exigences spécifiques

3.1 Exigences spécifiques sur les interfaces externes

3.1.1 Interfaces utilisateurs

Pour utiliser cet outil, il faudra disposer d'un clavier et d'une souris, grâce auxquels l'utilisateur pourra interagir avec l'outil. Ce dernier se présentera comme une fenêtre divisée en trois parties : dans l'une s'affichera la machine abstraite, dans la deuxième, on pourra voir l'implantation de l'étudiant, dans la dernière, les vérifications pourront être lancées.

3.1.2 Interfaces logicielles

Notre outil ne s'interfacera avec aucun logiciel externe (base de données ou autre). Il utilisera par contre les fichiers à analyser qui lui seront fournis par l'utilisateur, ainsi que des librairies qui seront fournies soit en standard avec le système d'exploitation, soit fournies avec l'outil.

3.1.3 Interfaces materielles

Notre outil ne s'interfacera avec aucun matériel spécifique, si ce n'est avec le clavier, la souris, et l'écran nécessaire à l'utilisation conviviale de TEP.

3.1.4 Communication entre interfaces

Cette section n'a pas lieu d'être.

3.2 Exigences spécifiques aux fonctionnalités de l'outil

3.2.1 Exigences spécifiques pour l'interface de TEP

Afin que les fichiers de l'utilisateur soient analysés, il faudra que ceux-ci soient fournis à l'outil.

Ces fichiers devront pouvoir être fournis sans faire de copier/coller du code source : les noms des fichiers seront suffisants.

Ces fichiers seront affichés dans des fenêtres spécifiques de TEP.

Aucune modification de ces fichiers ne sera faite automatiquement par cet outil. Cependant, si l'utilisateur souhaite, lui, faire des modifications de son code dans les fenêtres de TEP, il devra lui être possible de sauvegarder ses modifications.

Afin de faciliter le débuggage, il faudra que l'utilisateur ait un moyen de savoir quel est le numéro de la ligne ou il se trouve.

3.2.2 Exigences spécifiques pour la correction syntaxique

Certaines machines, syntaxiquement correctes pour l'atelier B complet ne seront pas acceptées par TEP.

Chaque erreur syntaxique devra être signalée par un message pertinent, et si possible de façon suffisamment explicite pour que le débuggage puisse se faire rapidement, en donnant la ligne où se trouve la première faute, et en expliquant de quel type de faute il s'agit.

3.2.3 Exigences spécifiques pour la correction algorithmique

Le mode d'analyse

Avant chaque analyse, un choix devra être présenté à l'utilisateur quant au mode d'analyse souhaité : SearchOne ou bien SearchAll. SearchOne s'arrêtera des que le premier contre-exemple aura été trouvé, alors que SearchAll cherchera tous les contre-exemples potentiels.

Le choix par défaut sera SearchOne.

Le mode de distribution

L'utilisateur devra choisir le mode de distribution souhaité sur les variables : distribution naïve, ou distribution spécifique. Tous les modes de distribution disponibles avec Oz seront proposés. Un seul mode de distribution pourra être choisi pour une analyse.

La présentation de ces choix devra être claire.

Le choix par défaut sera FirstFail.

Pour plus de détails sur les modes de distribution, on pourra aller à l'adresse suivante : http ://www.mozart-oz.org/documentation/fdt/index.html

Les messages d'erreurs

Enfin, l'utilisateur devra choisir le niveau de précision du message signalant les erreurs éventuelles :

- Choix 1 : message simple
- Choix 2 : message donnant l'assertion violée
- Choix 3 : message donnant la variable fautive
- Choix 4 : message montrant l'arbre de distribution des variables

Une fois de plus, une présentation claire de ces différents choix devra être faite à l'utilisateur.

Ces choix pourront être cumulatifs.

Le choix par défaut sera le choix 1.

3.2.4 Exigences spécifiques pour la génération de code

Si une machine abstraite et son implantation sont vues par l'outil comme correctes, le code C correspondant à l'implantation pourra être généré, si l'utilisateur le désire.

Le code généré devra pouvoir être compilé sans erreur.

Le code généré devra être indenté de façon claire.

Les noms de variables du code généré devront être les mêmes que celles de l'implantation.

3.2.5 Exigences spécifiques pour l'aide à l'utilisateur

La section d'aide à l'utilisateur devra être suffisante pour permettre à un néophyte de comprendre toutes les fonctionnalités de l'outil, même s'il n'a aucune notion de programmation par contraintes.

Chapitre 4

Annexes

4.1 Précisions sur la méthode B

Afin d'avoir plus de précisions sur la méthode B, nous vous recommandons vivement l'URL suivante :

http ://www3.inrets.fr/estas/mariano/documents/These/html/These002.html qui offre une présentation très complète de la méthode B, ainsi que :

http ://www.atelierb.societe.com/ qui offre différentes études de cas.