Université de Ngaoundéré
Faculté des Sciences
Departement of Maths-infos

University of Ngaoundere
Faculty of Science
Department of Mathematics
and Computer Science

filière : Systèmes et logiciels en Environnements Distribués (SLED)

TP SURETÉ DES SYSTÈMES

Dirigé par Dr.TCHAKOUNTE FRANKLIN

14 december 2017

Table de Matière I

Table de Matière II

Java PathFinder for Model Checking

Le **Model checking** est une approche automatique de la vérification formelle et exhaustive de tous les comportements du modèle. Il est base sous deux grandes approches de vérification:

- ensembliste (en utilisant les arbres symbiques)
- automate (explicite pour des entrées fournies)

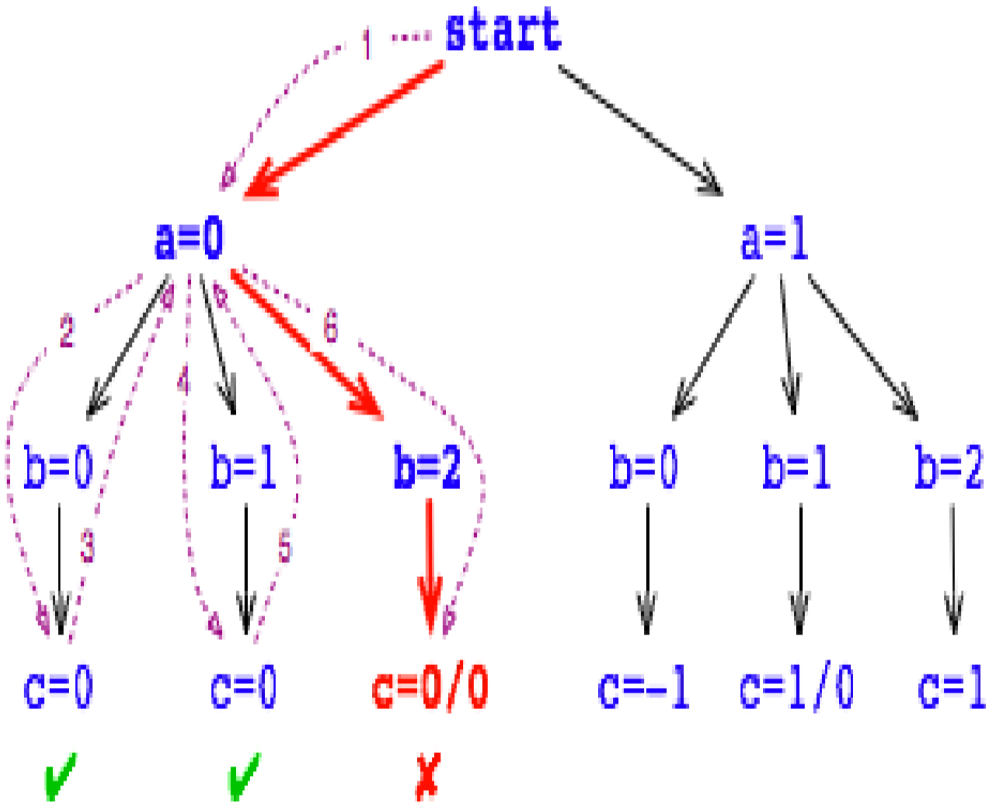
la suite nous aborderons un exemple illustrant l'approche ensembliste.

this link for more information about installation of jpf : https://babelfish.arc.nasa.gov/trac/jpf/wiki/install/start

figure 1

```
import java.util.Random;

public class Rand {
  public static void main (String[] args) {
    System.out.println("computing c = a/(b+a - 2)..");
    Random random = new Random(42);      // (1)

    int a = random.nextInt(2);           // (2)
    System.out.printf("a=%d\n", a);

    //... lots of code here

    int b = random.nextInt(300);           // (3)
    System.out.printf("  b=%d       ,a=%d\n", b, a);

    int c = a/(b+a -200);                  // (4)
    System.out.printf("=>  c=%d     , b=%d, a=%d\n", c, b, a);
  }
}
```

Figure 2



- JPF explores multiple possible executions GIVEN THE SAME CONCRETE INPUT

Practical test ....

références

Slides partially compiled from the NASA JavaPathFinder project and E. Clarke's course material

Java PathFinder for Symbolic Execution

description

L'exécution symbolique est une technique qui permet d'explorer les chemins d'exécution possibles d'un programme informatique à partir des symboles contenus dans son code source(byte code == .class file in java).
Contrairement à l'exécution concrète qui ne suit qu'un seul des chemins possibles et qui met directement à jour les variables en mémoire, l'exécution symbolique enregistre les formules logiques liant les variables entre elles.

but

Son but est d'analyser statiquement un programme pour trouver des bugs ou prouver certaines propriétés du programme. Il s'agit d'une interprétation abstraite d'un programme.

L'analyse staique est le fait de regrouper une variété de méthodes utilisées pour obtenir des informations sur le comportement d'un programme lors de son exécution sans réellement l'exécuter. De ce fait, l'execution symbolique s'effectue au travers des expressions symboliques. Au cours d'une execution symbolique, il y'a generation d'un arbre symbolique; Cet arbre est composé de :

constraint PC

Program counter

Cet arbre permet la generation des tests de differentes entrées possibles. A chaque branche de l'arbre, il y'a mise à jour du PC: Si la condition est verifée alors on continue l'execution ou l'exploration Sinon on arête le parcours dans l'aborescence.

Symbolic PathFinder

Performs symbolic execution of Java bytecodes

Handles complex math constraints, data structures and arrays, multi-threading, pre-conditions, strings (on-going work)

Applies to (executable) models and code

Generates test vectors and test sequences that are guaranteed to achieve user-specified coverage (e.g. path, statement, branch, MC/DC coverage)

Measures coverage.

Generates JUnit tests, Antares simulation scripts, etc. (output can be easily customizable)

During test generation process, checks for errors

Is flexible, as it allows for easy encoding of different coverage criteria

Is integrated with simulation environment (on-going work)

Figure 1

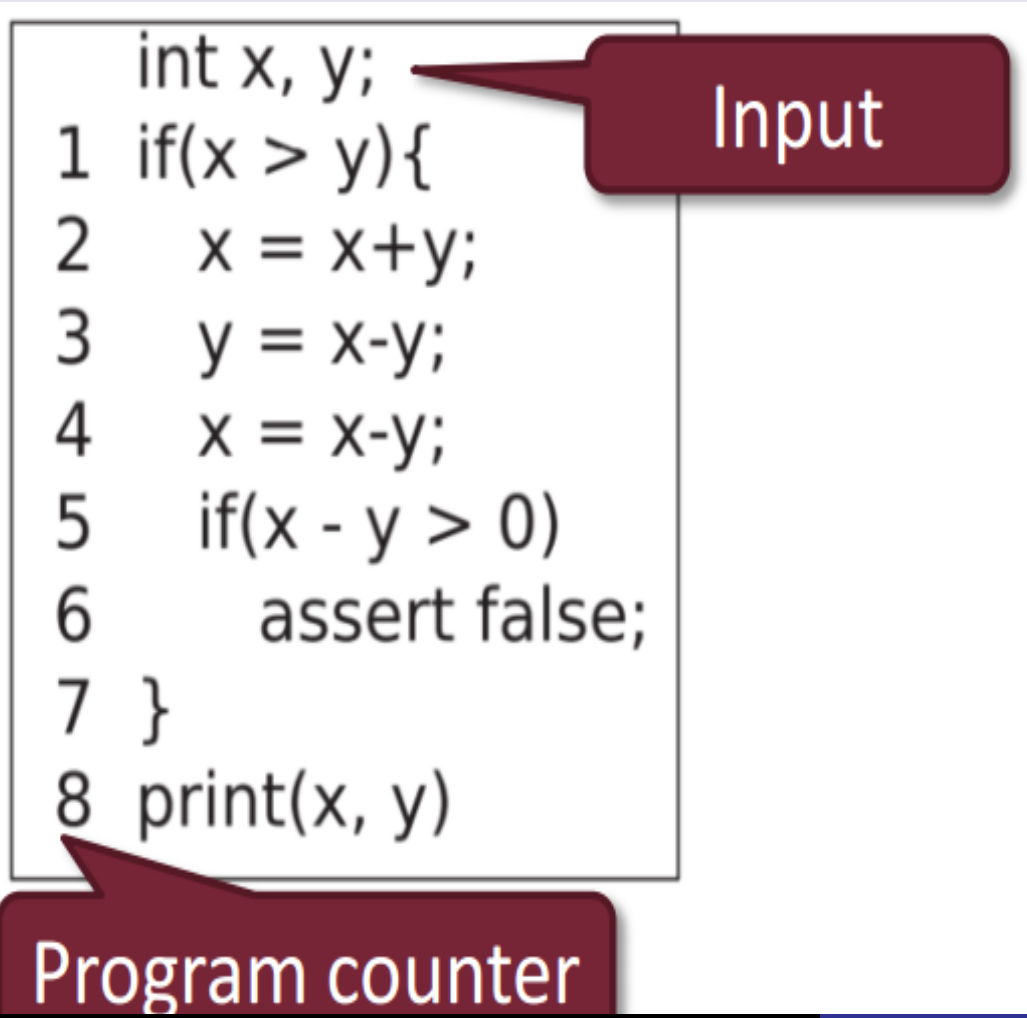


Figure 2

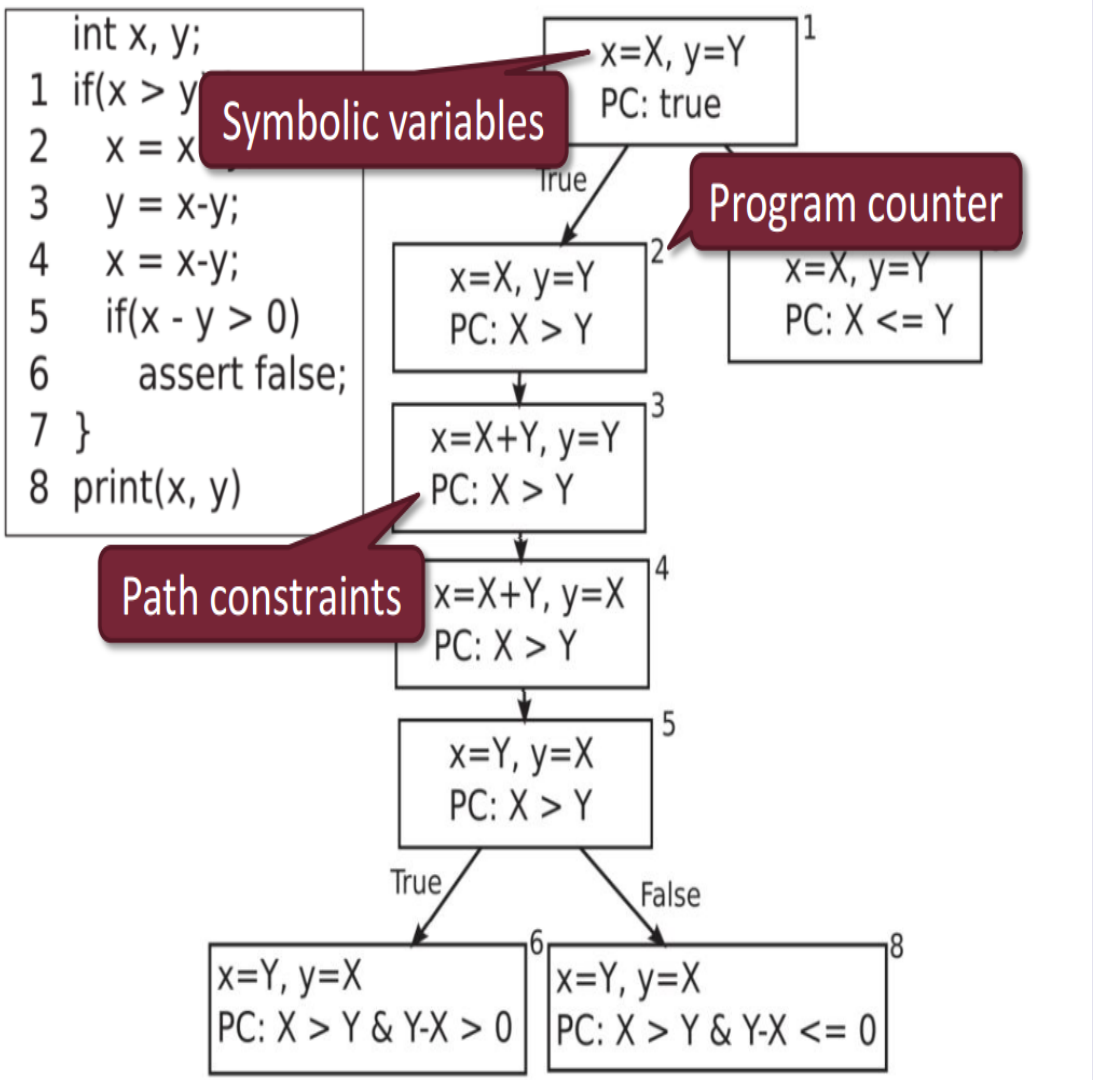
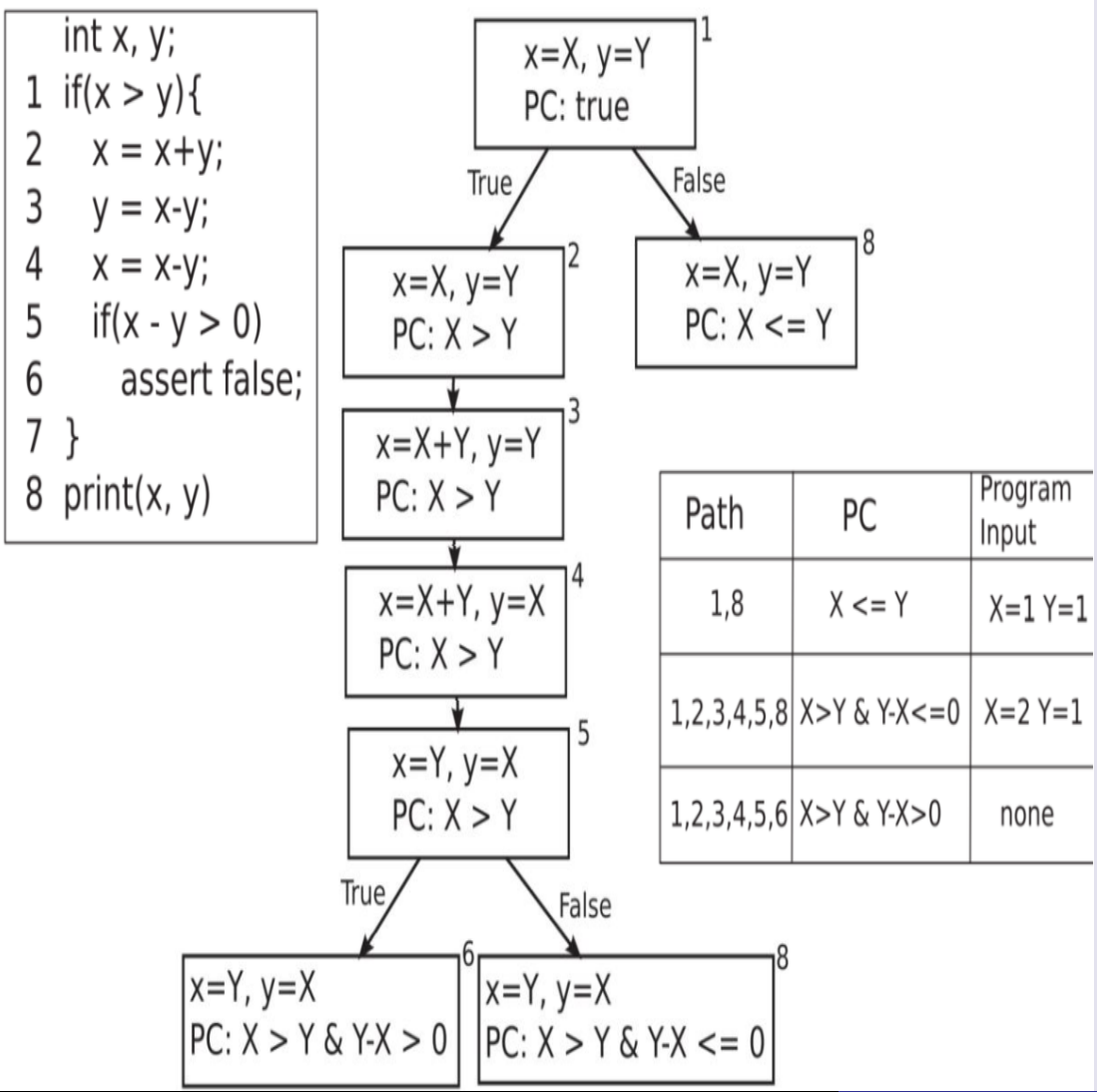

int x, y;
1 if(x > y
2 x = x
3 y = x-y;
4 x = x-y;
5 if(x - y > 0)
6 assert false;
7 }
8 print(x, y)
Symbolic variables
Program counter
Path constraints
x=X, y=Y
PC: true
1
True
x=X, y=Y
PC: X > Y
2
x=X, y=Y
PC: X <= Y
x=X+Y, y=Y
PC: X > Y
3
x=X+Y, y=X
PC: X > Y
4
x=Y, y=X
PC: X > Y
5
True
False
x=Y, y=X
PC: X > Y & Y-X > 0
6
x=Y, y=X
PC: X > Y & Y-X <= 0
8

Figure 3



Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none

In Java we can perform symbolic execution through Symbolic java path finder (SJPF).For the test we must grap source for jpf-symbc package at follow instructions: Download and instal mercurial who has a DVCS like git(sudo apt-get install mercurial in linux system for example) hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc and clone throw mercurial in netbeans or eclipse download and add this plugin in netbeans gov-nasa-jpf-netbeans-runjpf.nbm who help to perform the the verification test to java path finder and then check jpf-symbc configuration site.properties files who are locate in In last perform your test as to create java class and her jpf test file and then right click and verify it.

- Budapest University of Technology and Economics(Csaba Debreceni)
- https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc#no1
- [ASE'10] "Symbolic PathFnder: Symbolic Executionfor Java Bytecode"- toolpaper, C. Pasareanu and N. Rungta
- [ISSTA'08] "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software", C. Pãsãreanu, P. Mehlitz,D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape
- [FSE'08] "Differential Symbolic Execution",

JAVA MODELING LANGUAGE

Java Modeling Language(JML), est un langage de spécification formel des interfaces et du comportement des classes JAVA [Leavensb03]. Conçu au début, à l'université de Iowa State par Gary Leavens, pour des fins recherche, ce langage a pu rapidement susciter l'intérêt des industriels grâce à sa simplicité et sa puissance. Il fut alors l'objet d'un projet ouvert regroupant des industriels (Compac et Gemplus) et des laboratoires de recherches (MIT, INRIA, etc).

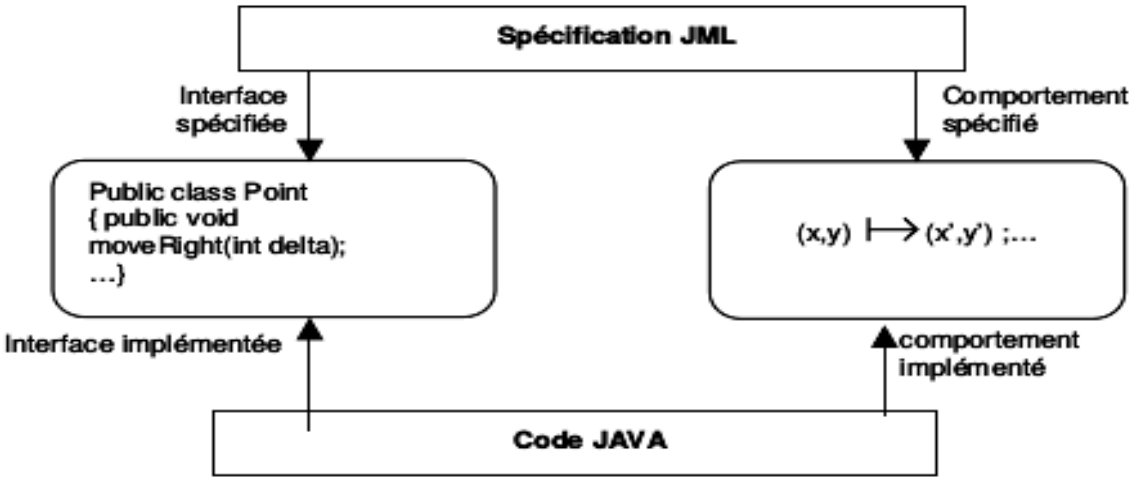
L'idée de base de JML est de permettre l'écriture de spécifications à base de contrats spécifiés en terme de conditions booléennes au niveau des classes (invariants et contraintes historiques), et des méthodes (préconditions , postconditions et assertions) afin de décrire deux aspects des modules JAVA :

L'interface:

l'interface d'une classe correspond à sa signature. Elle est constituée par son nom, les noms et types de ces attributs, et ses méthodes ;

Le comportement

C'est la façon dont un module doit réagir quand il est interpellé. Décrire le comportement d'une méthode revient à décrire les transformations d'états qu'elle peut subir [Leavensb03].



Les spécifications JML sont ajoutées au code Java sous forme d'annotations dans les commentaires. Ces commentaires Java sont interprétés comme annotations JML lorsqu'ils commencent par un @. Exemple de commentaires JML :
//@ <Spécifications JML> ou
/*@ <Spécification JML> @*/
La syntaxe de base de JML est basées sur les mots clés suivants :
requires
Définit une précondition pour la méthode qui suit.
ensures
Définit une postcondition pour la méthode qui suit.
signals Définit une condition déterminant quand une exception donnée peut être lancée par la méthode qui suit.
assignable
Définit quels attributs sont assignables par la méthode qui suit.

pure
Déclare une méthode pure, sans effet de bord ne modifiant rien (raccourci pour assignable).
invariant
Définit une propriété invariante de la classe.
also
Permet la déclaration de sur spécifications pour rajouter aux spécifications héritées de la superclasse.
assert
Définit une assertion JML.
Le JML fournit aussi de base les expressions suivantes:
Un identifiant pour la valeur de retour de la méthode qui suit. (<name>)
Un modificateur pour se reporter à la valeur de la variable <name> au moment de l'appel de la méthode.

le code suivant est un exemple simple d'une classe de compte bancaire annotée de spécifications JML :

exemple 1

```
1. public class Compte{
2. final int MAX_SOLDE;
3. private int solde;
4. private int points_fidélité;
5.
6. //@  public invariant 0<= solde && solde <= MAX_SOLDE ;
7. //@  private invariant points_fidélité>=0;
8. //@ public constraint  points_fidélité>= \old(points_fidélité)
9.
10.private byte[] pin
11.
12./*@ private invariant pin != null & pin.length ==4 &&
13.                (\forall int I; 0<= I && I<4;
14.                  0<= byte[I] && byte[I] <=9);
15.@*/
16.public int retrait(int montant) throws exception{
17...}
```

exemple 2

```
1.  public class Compte{
2.  final int MAX_SOLDE;
3.  private int solde;
4.  private int points_fidelite;

5.  //@  private invariant 0<= solde && solde <= MAX_SOLDE ;
6.  private byte[] pin
7.  /*@ private invariant pin != null & pin.length ==4 &&
            i.  (\forall int I; 0<= I && I<4;
                    1.  0<= byte[I] && byte[I] <=9);
8.  @*/
9.  /*@ requires montant >=0;
10. assignable solde;
11. ensures solde==\old(solde)- montant &&
12.           \result== solde;
13. signals ( RetraitException) solde ==\old(solde);
14. @*/
15. public int  retrait (int montant) throws exception{
16. ..}
```

Il existe des outils variés offrant des fonctionnalités basée sur les annotations JML. L'outil Iowa State JML permet de convertir les annotations JML en exécutable d'assertions via un compilateur de vérification d'assertion jmlc, de générer une Javadoc améliorée incluant des informations tirées des spécifications JML, et de générer des tests unitaires pour JUnit via le générateur jmlunit.

MÉTHODE B

La méthode B est une méthode formelle qui permet le raisonnement sur des systèmes complexes ainsi que le développement logiciel. La méthode B permet de modéliser de façon abstraite le comportement et les spécifications d'un logiciel dans le langage de B, puis par raffinements successifs d'aboutir à un modèle concret dans un sous-ensemble du langage B transcodable en Ada ou en C exécutables par une machine concrète. La méthode B permet de formaliser le système et son environnement de manière abstraite, puis par raffinements successifs, de rajouter les détails au modèle du système. Une activité de preuve formelle permet de vérifier la cohérence du modèle abstrait et la conformité de chaque raffinement avec le modèle supérieur (prouvant ainsi la conformité de l'ensemble des implémentations concrètes avec le modèle abstrait).

D'un point de vue purement théorique, l'objectif de la méthode B est de prouver qu'il n'y a pas d'écart entre la spécification et le code exécuté. Alors que les méthodes basées sur des tests permettent juste d'affirmer que les testeurs n'en ont pas trouvé ; et ce, quel que soit le niveau d'automatisation de la génération des scénarios de tests et les moyens mis en œuvre.

. Le langage B est un langage de spécification formel, basé sur la notion de machine abstraite. Les fondements théoriques de la méthode sont spécifiés dans le B-Book [Abr96]. Machine abstraite. Une machine abstraite représente un état spécifié par une partie statique (à l'aide de variables d'état et des propriétés d'invariance) et une partie dynamique (à l'aide d'opérations). Le langage pour la description de la statique repose sur la théorie des ensembles et sur la logique du premier ordre. Les variables sont ainsi typées par des ensembles et les invariants sont spécifiés à l'aide de conjonctions de prédicats du premier ordre. L'état de la machine abstraite ne peut être modifié que par des opérations. Le langage permettant d'exprimer la partie dynamique est un langage de substitutions généralisées. Il permet de décrire les opérations qui font évoluer l'état du système modélisé. Lors des phases initiales de spécification, le langage est abstrait : les instructions des opérations utilisent des préconditions et de l'indéterminisme.

Exemple Machine est un exemple de machine abstraite spécifiée avec le lan- gage B :

MACHINE Exemple Machine

VARIABLES x

INVARIANT

$x \in 0..20$

INITIALISATION

$x := 10$

OPERATIONS

change =

pre

$x + 2 \geq 20$

$x - 2 \leq 0$

then
choice
x := x + 2
or
x := x - 2
end
end
On remarque que, dans le corps de l'opération abstraite change, la substi- tution est spécifiée à l'aide de la commande choice qui n'est pas déterministe. Dans ce cas, la variable abstraite x peut être substituée par x +2 ou par x 2. La précondition (pre) permet de faire respecter l'invariant (voir la clause IN- VARIANT) si l'opération change est exécutée.

À la différence des approches précédentes, B possède un outil très puis- sant qui couvre toutes les phases de la méthode de conception. L'Atelier B [Cle], commercialisé par la société Clearsy, est en effet un environnement permettant de gérer des projets en langage B. Il offre différentes fonctionnalités : – automatisation de certaines tâches (vérification syntaxique, génération au- tomatique de théorèmes à démontrer, traduction de B vers C, C++, ...), – aide à la preuve pour démontrer automatiquement des théorèmes, – aide au développement. Plus précisément, le prouveur de l'Atelier B permet de vérifier quatre points importants des spécifications B : – au niveau de la machine : la dynamique doit respecter la statique, – au niveau de l'initialisation : l'initialisation (clause INITIALISATION) établit l'invariant (clause INVARIANT), – au niveau des opérations : chaque opération (clause OPERATIONS) doit préserver les propriétés d'invariance (clause INVARIANT), – l'Atelier B permet enfin de prouver la correction du raffinement par

MERCI POUR VOTRE ATTENTION !!!