

### 2.1 La vérification des programmes :

Il est établi que les technologies de la vérification pourraient être appliquées à la conception plutôt qu'aux programmes écrits en des langages de programmations modernes. Détecter les erreurs plus tôt au niveau design est crucial et réduira le coût de la maintenance.

L'état de l'art des méthodes formelles est naturellement dirigé vers les designs, ceci est dû simplement au fait que le design est de complexité moindre, ce qui rend l'analyse formelle plus faisable et pratique. Il est évident que, la vérification du design est un axe de recherche très important.

La communauté des méthodes formelles donne une attention particulière à la vérification des programmes, cette nouvelle tendance assez récente est due à certains facteurs, que nous citons ci-après [17] :

Les programmes contiennent souvent des erreurs fatales malgré l'existence de la conception soigneusement élaborée. Plusieurs deadlocks et violations de sections critiques, par exemple, sont introduites à un certain niveau de détails du design. Typiquement, ces erreurs ne sont pas traitées même si le design formel est utilisé. Ceci s'est bien affirmé dans l'analyse du système de contrôle à distance de l'agent spacecraft écrit dans le langage de programmation LISP, est été analysé utilisant le Spin model-checker où plusieurs erreurs classiques de multi-threading ont été détectées et localisées.

D'un autre côté, il existe des erreurs causées par une ambiguïté dans la compréhension des algorithmes complexes, tels que les protocoles de communication pour les systèmes parallèles, ainsi que le garbage collect algorithm. D'autres types d'erreurs sont simples considérant les erreurs de programmation concurrentielle, telles que l'oubli de mettre le code en une section critique ou causer les deadlocks.

Les erreurs de ce genre typiquement ne peuvent pas être détectées au niveau design, et ils sont de réel hasard, en particulier dans les systèmes safety-critical. Les algorithmes complexes peuvent tout à fait être analysés au niveau design, il n'y a aucune raison que de tel design ne peut pas être exprimé en un langage de programmation moderne. Cependant, de profondes erreurs de design peuvent aussi bien apparaître.

Les langages de programmation modernes sont le résultat de décennies de recherche, ils sont conçus sur de bons principes. Et de ce fait, il peut être de bons langages de design et modélisation. Ce-ci est une certaine extension d'une idée déjà appliquée avec l'UML où les conditions de transition entre les états peuvent être annotées avec des fragments de code d'un langage de choix.

Une observation intéressante concernant certaines méthodes de développement de programmes suggère une approche prototypée où la construction du système est incrémentale utilisant un réel langage de programmation, au lieu d'être dérivé d'un design pré-établi. On peut conclure que tout résultat de recherche dans les langages de programmation peuvent être bénéfiques pour la vérification conceptuelle surtout que le design est typiquement moins complexe.

Il serait avantageux pour les méthodes formelles d'être combinées à d'autres domaines de recherches qui traditionnellement se focalisent sur les programmes, telles que l'analyse statique et le test des programmes. Ces techniques sont moins complètes, mais souvent elles donnent de meilleure performance. L'objectif des méthodes formelles n'est pas seulement de prouver la correction du programme mais aussi, le déboguer et localiser les erreurs.

Etudier les méthodes formelles pour les langages de programmation peut générer certains avantages pour la communauté des méthodes formelles surtout, qu'il existe une tendance vers la standardisation des langages de programmation. Ceci pourrait rendre possible de comparer et d'intégrer de différents outils travaillant sur le même langage ou bien sur une «clean-susbsets» de ces langages.

### 2.2. Le principe de vérification des programmes

La vérification des programmes écrits en langages de programmations modernes est assurée en général par le Spin model-checker. Cependant la langue input du Spin est le Promela. Donc si on veut profiter de la puissance de l'outil Spin, on devrait lui présenter nos programmes comme modèles écrits en Promela. Une phase de translation du langage de programmation des programmes sujet de vérification est à prévoir.

La vérification des programmes en vérité se réalise d'une façon indirecte, car elle doit passer par une phase de translation du langage du programme considéré vers le Promela langue input du Spin model-checker. Le principe de la vérification est illustré dans le graphe suivant :

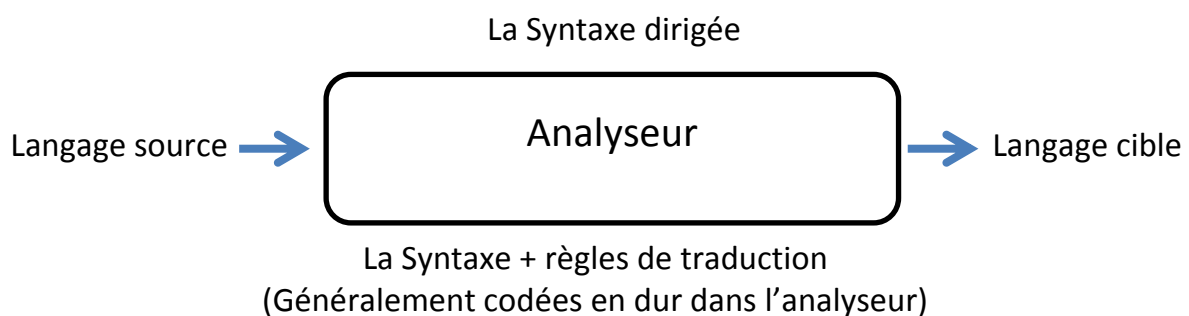


Figure 2.1 : principe de la vérification des programmes

### 2.3. La Traduction dirigée par la syntaxe (syntax directed translation)

Une technique où la structure d'un processeur de langage (par exemple un compilateur) est basée sur la structure de syntaxe abstraite du langage. Il pourrait y avoir une procédure dans le traducteur correspondant à chaque catégorie dans la syntaxe abstraite. Cette procédure est responsable du traitement des constructions de cette catégorie. Chaque procédure pourrait appeler d'autres correspondants à la notion de sous-constituants, puis de combiner leurs résultats pour donner le résultat global de cette construction [18].

Traduction dirigée par la syntaxe se réfère à une méthode de mise en œuvre de compilateur où la traduction en langue source est entièrement pilotée par l'analyseur. En d'autres termes, les processus et les arbres d'analyse sont utilisés pour diriger l'analyse sémantique et la traduction du programme source. Cela peut être une phase séparée d'un compilateur.

Pour traduire une construction d'un langage de programmation, un analyseur peut avoir besoin de garder trace de nombreuses informations en plus du code engendré pour cette construction. Nous pouvons augmenter notre grammaire classique d'information supplémentaire pour contrôler l'analyse sémantique et la traduction. Ces grammaires sont appelées grammaires d'attributs.

Nous augmentons une grammaire en associant des attributs à chaque symbole de grammaire qui décrit ses propriétés. Un attribut a un nom et une valeur associée: une chaîne, un nombre, un type, un emplacement, un registre, toutes les informations dont nous avons besoin. Par exemple, les variables peuvent avoir un attribut "type" (qui enregistre le type déclaré d'une variable, utile plus tard dans la vérification du type) ou une constante entière peut avoir un attribut "value" (dont nous aurons besoin par la suite pour générer du code).

Pour toute règle de la grammaire, on fournit les règles sémantiques dites actions, qui décrivent comment computer la valeur d'attribut associée à chaque symbole grammatical dans la production. La valeur de l'attribut du nœud d'analyse dépend des informations provenant des nœuds fils (successeurs) ou bien des nœuds parents (prédécesseurs).

### 2.3.1 Le principe de traduction dirigé par la syntaxe.

Dans ce qui suit nous présentons les notions de base de cette méthode de traduction dirigée par la syntaxe des langages de programmation.

#### 2.3.1.1 Définition de la syntaxe d'un langage

Pour définir la syntaxe d'un langage nous utilisons les notions suivant :

- Utiliser une notation spéciale.
- La grammaire à libre contexte.
- Un ensemble de règles.

Exemple :

If (expression) statement else statement

Correspond à la règles:

stmt -> if(expr) stmt else stmt

#### 2.3.1.2 Les composants d'une grammaire à libre contexte

- Ensemble de symboles terminaux.
- Ensemble de non-terminaux.
- Ensemble de productions.
  - La tête est un non-terminal.
  - Le corps est une séquence de terminaux et/ou non-terminaux.
- Désignation d'un non-terminal comme symbole Start.

**String de terminaux** : séquence de zero ou plus de terminaux.

**Dérivation** :

- Etant donné une grammaire (productions)
- Commencer avec le symbole Start.
- D'une façon répétitive, remplacer le non-terminal par son corps.
- On obtient le langage défini par la grammaire (groupe de strings terminaux)

**Parsing**

- Etant donné un ensemble de terminaux.
- Montre comment le dérivé à partir du symbole Start de la grammaire.

### 2.3.1.3 L'arbre d'analyse

Une structure de donnée qui montre comment à partir du symbole Start de la grammaire, les strings sont dérivés.

**L'Attribut :** Un attribut est une valeur qui est associée à un nœud de l'arbre syntaxique.

- Un attribut peut représenter une information (ou avoir un type) quelconque une chaîne de caractères, un emplacement...

- Il est possible d'associer autant d'attributs que l'on désire à chaque non Terminal.

- Un nœud de l'arbre syntaxique est nommé à l'aide d'un symbole non Terminal. Chaque nœud nommé par le même non terminal aura les mêmes attributs.

**Les attributs synthétisés :** la valeur à un nœud de l'arbre d'un attribut Synthétisé est définie à l'aide des valeurs des attributs des fils de ce nœud:

**Les attributs hérités:** la valeur à un nœud de l'arbre d'un attribut hérité est Calculée à l'aide des attributs du père de ce nœud

### 2.3.1.4 L'Action sémantique

Une action sémantique est définie par un algorithme qui permet de calculer les attributs d'un nœud, en fonction des attributs des fils ou du père.

En pratique, les relations entre les attributs nous permettent d'écrire les actions sémantiques.

Il est également possible d'avoir besoin des valeurs des attributs des frères de ce nœud.

#### Les règles sémantiques

Les règles sémantiques impliquent des dépendances entre les attributs. Ces dépendances peuvent être représentées par un graphe:

- les nœuds sont les attributs;
- les arcs représentent les dépendances entre les attributs.

### 2.3.1.5 Table des symboles

Si l'on veut utiliser les deux sortes d'attributs, il peut y avoir des boucles Dans le graphe des dépendances et donc dans le calcul de ces attributs...

Il existe un algorithme qui permet de savoir si une boucle dans le calcul des Attributs apparaît, mais la complexité de cet algorithme est exponentielle en temps.

En pratique:

- l'utilisation de la table des symboles permet de résoudre la plupart des problèmes rencontrés;
- les attributs synthétisés sont très utilisés;
- les attributs hérités sont utiles pour les déclarations de variables.

La construction d'arbres abstraits permet de séparer l'étape de Traduction (génération de code) de l'analyse syntaxique.

L'ordre de construction de l'arbre syntaxique (pendant l'analyse syntaxique) peut être différent de l'ordre d'évaluation de l'arbre abstrait pour générer le code.

### Les expressions

Constructions des arbres abstraits pour les expressions Chaque nœud de l'arbre abstrait contient un champ pour chaque opérande de l'opérateur (appelé étiquette) du nœud considéré.

### 2.3.1.6 Le principe :

- Le processus de translation est dirigé par la syntaxe.
- Les routines de sémantique performe l'interprétation basée sur la structure syntaxique.
- Attacher des attributs aux symboles de la grammaire.
- Les valeurs des attributs sont calculés par les règles sémantiques associées avec les productions de la grammaire.

Exemple d'un arbre d'analyse

- Annoter l'arbre d'analyse par attacher des attributs sémantiques aux nœuds de l'arbre d'analyse.
- Générer le code en visitant les nœuds de l'arbre d'analyse dans un ordre donné.
- Inputs :  $y := 3 * x + z$

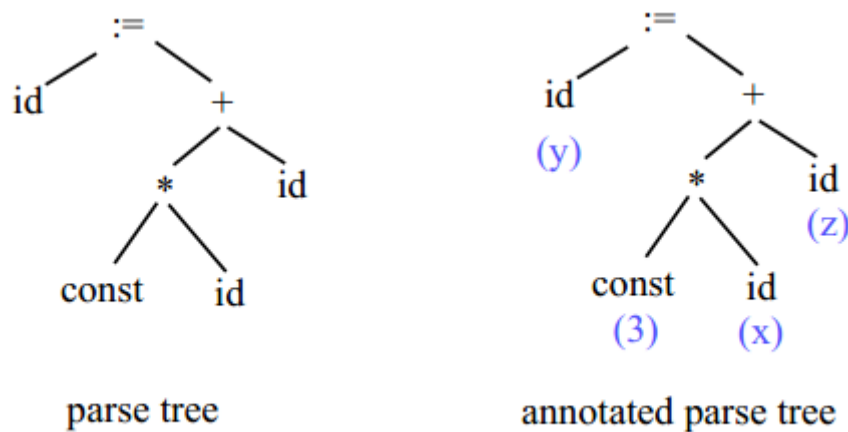


Figure 2.2 : Exemple d'un arbre d'analyse

### 2.4. Conclusion

La traduction des programmes d'un langage de programmation à un autre devient de plus en plus importante dans le développement des logiciels. Le principe de la translation suit l'esprit de la translation dirigée par la syntaxe, qui utilise la grammaire du langage pour capter l'arbre abstrait qui sera annoté. Cette annotation est une augmentation de la grammaire en attribuant des attributs aux symboles de la grammaire pour les doter de la sémantique nécessaire pour mener à bien la traduction.

Dans notre travail, le principe de translation proposé suit la même philosophie, avec l'intégration de certaines techniques qui pourront optimiser la translation. Elle sera présentée dans le chapitre suivant.