MGL7160 Méthodes formelles et semi-formelles

Les contrats

Roger Villemaire

Département d'informatique
UQAM

premier avril 2014

Plan

Plan

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Types de données en KeY

- Pour permettre la vérification d'un programme Java, KeY permet l'usage des types suivants.
  - Types prédéfinis Java comme `int`, `char`, `boolean` et `String`.
  - Les types associés aux classes Java que vous définissez.
- KeY implémente en fait un sur-ensemble de Java Card. Par rapport à Java, il manque entre autres les choses suivantes :
  - les types flottants (float et double),
  - la création et le chargement dynamique des classes,
  - le parallélisme.

Mise-à-jour

- Pour traiter les affectations que réalisent les programmes Java, KeY utilise des *mises-à-jours* (updates).
- Une mise-à-jour est une suite d'affectations séparées par des $||$ et placée entre accolades.
- La mise-à-jour apparaît toujours directement devant la formule à laquelle elle s'applique.
- Une mise-à-jour a pour effet de remplacer les variables d'une formule par des expressions.

Mise-à-jour

- Pour traiter les affectations que réalisent les programmes Java, KeY utilise des *mises-à-jours* (updates).
- Une mise-à-jour est une suite d'affectations séparées par des $||$ et placée entre accolades.
- La mise-à-jour apparaît toujours directement devant la formule à laquelle elle s'applique.
- Une mise-à-jour a pour effet de remplacer les variables d'une formule par des expressions.

Mise-à-jour

- Pour traiter les affectations que réalisent les programmes Java, KeY utilise des *mises-à-jours* (updates).
- Une mise-à-jour est une suite d'affectations séparées par des $||$ et placée entre accolades.
- La mise-à-jour apparaît toujours directement devant la formule à laquelle elle s'applique.
- Une mise-à-jour a pour effet de remplacer les variables d'une formule par des expressions.

Mise-à-jour

- Pour traiter les affectations que réalisent les programmes Java, KeY utilise des *mises-à-jours* (updates).
- Une mise-à-jour est une suite d'affectations séparées par des $||$ et placée entre accolades.
- La mise-à-jour apparaît toujours directement devant la formule à laquelle elle s'applique.
- Une mise-à-jour a pour effet de remplacer les variables d'une formule par des expressions.

Mise-à-jour

- Pour traiter les affectations que réalisent les programmes Java, KeY utilise des *mises-à-jours* (updates).
- Une mise-à-jour est une suite d'affectations séparées par des $||$ et placée entre accolades.
- La mise-à-jour apparaît toujours directement devant la formule à laquelle elle s'applique.
- Une mise-à-jour a pour effet de remplacer les variables d'une formule par des expressions.

mise-à-jour KeY

- ```
  \programVariables { int x,y; }
  \problem {
      {x:=-(2*y+x) || y:=(2*y+x) }(x + y = 0)
  }
  ```
- Après le remplacement on obtient
  `-(2*y+x) + (2*y+x) = 0`,
- Ce qui est bien une égalité arithmétique, comme peut le montrer KeY.
  ```

mise-à-jour KeY

- ```
  \programVariables { int x,y; }
  \problem {
      {x:=-(2*y+x) || y:=(2*y+x) }(x + y = 0)
  }
  ```
- Après le remplacement on obtient
  `-(2*y+x) + (2*y+x) = 0,`
- Ce qui est bien une égalité arithmétique, comme peut le montrer KeY.
  ```

mise-à-jour KeY

- ```
  \programVariables { int x,y; }
  \problem {
      {x:=-(2*y+x) || y:=(2*y+x) }(x + y = 0)
  }
  ```
- Après le remplacement on obtient
  `-(2*y+x) + (2*y+x) = 0,`
- Ce qui est bien une égalité arithmétique, comme peut le montrer KeY.
  ```

mise-à-jour KeY

- ```
  \programVariables { int x,y; }
  \problem {
      {x:=-(2*y+x) || y:=(2*y+x) }(x + y = 0)
  }
  ```
- Après le remplacement on obtient
  `-(2*y+x) + (2*y+x) = 0`,
- Ce qui est bien une égalité arithmétique, comme peut le montrer KeY.
  ```

Plan

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Logique dynamique

- La *logique dynamique* est une extension du calcul propositionnel qui permet de traiter l'*exécution* de programme.
- Plus précisément, la logique dynamique introduit pour tout *programme p* les deux *modalités* suivantes.
  - $\langle p \rangle P$ qui signifie :
    - il y a une exécution du programme p qui se termine dans un état où P est vérifié.
  - $[p]P$ qui signifie :
    - toutes les exécutions du programme p (qui se terminent) le font dans un état où P est vérifié.

Non déterminisme

- De façon générale la logique dynamique s'applique à des systèmes *non-déterministes*,
  - c.-à-d. qu'il peut y avoir plusieurs façons d'exécuter un programme (ou une action).
- Dans le cas d'un programme Java, la seule part de non-déterminisme est la possibilité que le programme ne se termine pas.

Non déterminisme

- De façon générale la logique dynamique s'applique à des systèmes *non-déterministes*,
  - c.-à-d. qu'il peut y avoir plusieurs façons d'exécuter un programme (ou une action).
- Dans le cas d'un programme Java, la seule part de non-déterminisme est la possibilité que le programme ne se termine pas.

Non déterminisme

- De façon générale la logique dynamique s'applique à des systèmes *non-déterministes*,
  - c.-à-d. qu'il peut y avoir plusieurs façons d'exécuter un programme (ou une action).
- Dans le cas d'un programme Java, la seule part de non-déterminisme est la possibilité que le programme ne se termine pas.

Non déterminisme

- De façon générale la logique dynamique s'applique à des systèmes *non-déterministes*,
  - c.-à-d. qu'il peut y avoir plusieurs façons d'exécuter un programme (ou une action).
- Dans le cas d'un programme Java, la seule part de non-déterminisme est la possibilité que le programme ne se termine pas.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Logique dynamique Java

- KeY utilise une logique dynamique dont les programmes sont en Java.
- KeY se limite aux programmes Java séquentiels. La seule source de non-déterminisme est donc la non-terminaison.
- Nous aurons donc en KeY la sémantique suivante.
  - $\langle p \rangle P$ signifie que le programme *p se termine* dans un état où P est vérifié.
  - $[p]P$ qui signifie que si le programme *p se termine*, le système est alors dans un état où P est vérifié.
- Pour un programme qui se termine toujours, il y a donc aucune différence entre ces deux modalités.

Modalité en KeY

- ```
  \javaSource "src";

  \programVariables{
          Chauffeur c;
  }
  \problem{
  \<{ c = new Chauffeur(); }\> c.age = 0
  }
  ```
- Les variables qui ne sont pas déclarées dans le code Java (comme `c` ci-dessus) doivent apparaître dans `\programVariables`.
  ```

Modalité en KeY

- ```
  \javaSource "src";

  \programVariables{
          Chauffeur c;
  }
  \problem{
  \<{ c = new Chauffeur(); }\> c.age = 0
  }
  ```
- Les variables qui ne sont pas déclarées dans le code Java (comme `c` ci-dessus) doivent apparaître dans `\programVariables`.
  ```

Modalité en KeY

- ```
  \javaSource "src";

  \programVariables{
          Chauffeur c;
  }
  \problem{
  \<{ c = new Chauffeur(); }\> c.age = 0
  }
  ```
- Les variables qui ne sont pas déclarées dans le code Java (comme `c` ci-dessus) doivent apparaître dans `\programVariables`.
  ```

Plan

Notion de contrat

- Un contrat est une entente entre un *fournisseur* et un *client* par lequel
  - Le fournisseur s'engage à remplir ses *obligations* dans la mesure où ses *garanties* sont respectées.
  - Le client est prêt à remplir ses *obligations* pour pouvoir profiter des *garanties* prévues pour lui.

Notion de contrat

- Un contrat est une entente entre un *fournisseur* et un *client* par lequel
  - Le fournisseur s'engage à remplir ses *obligations* dans la mesure où ses *garanties* sont respectées.
  - Le client est prêt à remplir ses *obligations* pour pouvoir profiter des *garanties* prévues pour lui.

Notion de contrat

- Un contrat est une entente entre un *fournisseur* et un *client* par lequel
  - Le fournisseur s'engage à remplir ses *obligations* dans la mesure où ses *garanties* sont respectées.
  - Le client est prêt à remplir ses *obligations* pour pouvoir profiter des *garanties* prévues pour lui.

Notion de contrat

- Un contrat est une entente entre un *fournisseur* et un *client* par lequel
  - Le fournisseur s'engage à remplir ses *obligations* dans la mesure où ses *garanties* sont respectées.
  - Le client est prêt à remplir ses *obligations* pour pouvoir profiter des *garanties* prévues pour lui.

Pré et postconditions

- En génie logiciel la notion de contrat peut s'appliquer au niveau des fonctions (ou méthodes).
  - La réalisation d'une fonction doit remplir des obligations, exprimées par les *postconditions*, mais seulement dans la mesure où les garanties, exprimées par les *préconditions* sont remplies.
  - Tout usage de la fonction doit s'assurer que certaines obligations, les *préconditions*, sont respectées pour pouvoir profiter des garanties que sont les *postconditions*.

Pré et postconditions

- En génie logiciel la notion de contrat peut s'appliquer au niveau des fonctions (ou méthodes).
  - La réalisation d'une fonction doit remplir des obligations, exprimées par les *postconditions*, mais seulement dans la mesure où les garanties, exprimées par les *préconditions* sont remplies.
  - Tout usage de la fonction doit s'assurer que certaines obligations, les *préconditions*, sont respectées pour pouvoir profiter des garanties que sont les *postconditions*.

Pré et postconditions

- En génie logiciel la notion de contrat peut s'appliquer au niveau des fonctions (ou méthodes).
  - La réalisation d'une fonction doit remplir des obligations, exprimées par les *postconditions*, mais seulement dans la mesure où les garanties, exprimées par les *préconditions* sont remplies.
  - Tout usage de la fonction doit s'assurer que certaines obligations, les *préconditions*, sont respectées pour pouvoir profiter des garanties que sont les *postconditions*.

Pré et postconditions

- En génie logiciel la notion de contrat peut s'appliquer au niveau des fonctions (ou méthodes).
  - La réalisation d'une fonction doit remplir des obligations, exprimées par les *postconditions*, mais seulement dans la mesure où les garanties, exprimées par les *préconditions* sont remplies.
  - Tout usage de la fonction doit s'assurer que certaines obligations, les *préconditions*, sont respectées pour pouvoir profiter des garanties que sont les *postconditions*.

Vérification formelle

- Le contrat, formé de la pré et de la postcondition est une contrainte que doit satisfaire la fonction.
- On peut donc formellement valider que si la précondition est satisfaite avant l'appel, la postcondition le sera après l'appel.
- C'est exactement l'approche prise par l'outil KeY.

Vérification formelle

- Le contrat, formé de la pré et de la postcondition est une contrainte que doit satisfaire la fonction.
- On peut donc formellement valider que si la précondition est satisfaite avant l'appel, la postcondition le sera après l'appel.
- C'est exactement l'approche prise par l'outil KeY.

Vérification formelle

- Le contrat, formé de la pré et de la postcondition est une contrainte que doit satisfaire la fonction.
- On peut donc formellement valider que si la précondition est satisfaite avant l'appel, la postcondition le sera après l'appel.
- C'est exactement l'approche prise par l'outil KeY.

Vérification formelle

- Le contrat, formé de la pré et de la postcondition est une contrainte que doit satisfaire la fonction.
- On peut donc formellement valider que si la précondition est satisfaite avant l'appel, la postcondition le sera après l'appel.
- C'est exactement l'approche prise par l'outil KeY.

Plan

Pré et postconditions en KeY

- KeY permet la vérification formelle exhaustive de contrats exprimés sous la forme de pré et postconditions.
- En fait KeY vérifiera que si les préconditions sont satisfaites alors, après l'exécution de la fonction, toutes les postconditions sont vérifiées.

Pré et postconditions en KeY

- KeY permet la vérification formelle exhaustive de contrats exprimés sous la forme de pré et postconditions.
- En fait KeY vérifiera que si les préconditions sont satisfaites alors, après l'exécution de la fonction, toutes les postconditions sont vérifiées.

Pré et postconditions en KeY

- KeY permet la vérification formelle exhaustive de contrats exprimés sous la forme de pré et postconditions.
- En fait KeY vérifiera que si les préconditions sont satisfaites alors, après l'exécution de la fonction, toutes les postconditions sont vérifiées.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Traduction en logique dynamique

- KeY permet donc de vérifier l'une ou l'autre formalisations suivantes.
  - `préconditions -> <fonction> postconditions`
  - `préconditions -> [fonction] postconditions`
- La différence entre ces deux méthodes est que la première exige que l'exécution de la fonction se termine.

Plan

Historique

- Le *JAVA Modeling Language* (JML) est un langage de spécification spécifique à Java.
  - Contrairement à OCL, il ne s'agit pas d'une norme proposée par une organisation,
  - mais d'un projet de la communauté dirigé par Gary T. Leavens de l'Iowa State University www.jmlspecs.org.

Historique

- Le *JAVA Modeling Language* (JML) est un langage de spécification spécifique à Java.
  - Contrairement à OCL, il ne s'agit pas d'une norme proposée par une organisation,
  - mais d'un projet de la communauté dirigé par Gary T. Leavens de l'Iowa State University www.jmlspecs.org.

Historique

- Le *JAVA Modeling Language* (JML) est un langage de spécification spécifique à Java.
  - Contrairement à OCL, il ne s'agit pas d'une norme proposée par une organisation,
  - mais d'un projet de la communauté dirigé par Gary T. Leavens de l'Iowa State University www.jmlspecs.org.

Historique

- Le *JAVA Modeling Language* (JML) est un langage de spécification spécifique à Java.
  - Contrairement à OCL, il ne s'agit pas d'une norme proposée par une organisation,
  - mais d'un projet de la communauté dirigé par Gary T. Leavens de l'Iowa State University www.jmlspecs.org.

Évolution

- JML évolue donc au gré des besoins et il n'existe pas de référence ultime ou de norme.
- Certaines caractéristiques du langage peuvent donc évoluer rapidement.
- L'usage qu'en fait KeY est donc principalement concentré sur les aspects qui sont stabilisés.

Évolution

- JML évolue donc au gré des besoins et il n'existe pas de référence ultime ou de norme.
- Certaines caractéristiques du langage peuvent donc évoluer rapidement.
- L'usage qu'en fait KeY est donc principalement concentré sur les aspects qui sont stabilisés.

Évolution

- JML évolue donc au gré des besoins et il n'existe pas de référence ultime ou de norme.
- Certaines caractéristiques du langage peuvent donc évoluer rapidement.
- L'usage qu'en fait KeY est donc principalement concentré sur les aspects qui sont stabilisés.

Évolution

- JML évolue donc au gré des besoins et il n'existe pas de référence ultime ou de norme.
- Certaines caractéristiques du langage peuvent donc évoluer rapidement.
- L'usage qu'en fait KeY est donc principalement concentré sur les aspects qui sont stabilisés.

Développement plutôt que conception

- JML se concentre sur la phase de développement en Java.
- C'est une méthode pour ajouter des invariants, pré et postconditions à du code Java.
- Les contraintes JML apparaissent directement dans les commentaires Java (similairement à Javadoc).

Développement plutôt que conception

- JML se concentre sur la phase de développement en Java.
- C'est une méthode pour ajouter des invariants, pré et postconditions à du code Java.
- Les contraintes JML apparaissent directement dans les commentaires Java (similairement à Javadoc).

Développement plutôt que conception

- JML se concentre sur la phase de développement en Java.
- C'est une méthode pour ajouter des invariants, pré et postconditions à du code Java.
- Les contraintes JML apparaissent directement dans les commentaires Java (similairement à Javadoc).

Développement plutôt que conception

- JML se concentre sur la phase de développement en Java.
- C'est une méthode pour ajouter des invariants, pré et postconditions à du code Java.
- Les contraintes JML apparaissent directement dans les commentaires Java (similairement à Javadoc).

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

Expressions Java

- JML utilise directement la syntaxe Java. Par exemple :
  - `!`, `&&`, `||` pour les opérations logiques,
  - `==`, `!=`, `>=` pour les comparaisons.
- De plus, il est possible d'utiliser n'importe quelle fonction Java *qui se termine et ne change pas l'état du système*.
  - Ces fonctions sont indiquées par `/*@ pure @*/`.

fonction pure

```
public class Chauffeur {
...
    public /*@ pure @*/ int getAge() {
        return this.age;
    }
}
```

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* (`\forall T e;e0;e1`),
  - *existentiel* (`\exists T e;e0;e1`).
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* (`\forall T e;e0;e1`),
  - *existentiel* (`\exists T e;e0;e1`).
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* (`\forall T e;e0;e1`),
  - *existentiel* (`\exists T e;e0;e1`).
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication ==> et la double implication <==>,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Expressions JML

- JML ajoute à Java : l'implication `==>` et la double implication `<==>`,
- ainsi que les quantificateurs :
  - *universel* `(\forall T e;e0;e1)`,
  - *existentiel* `(\exists T e;e0;e1)`.
  - numériques :
    - `\num_of` retourne le nombre de valeurs qui satisfont une expression.
    - `\sum`, `\product`, `\min` et `\max` qui réalisent les opérations arithmétiques correspondantes.

Spécification JML

- JML permet de définir des invariants *de classe* et *d'instance*,
- ainsi que des contrats avec pré et postconditions.
  - En JML un contrat s'appelle plutôt un *cas de spécification* (specification case)
  - alors qu'un *contrat* est l'ensemble des cas de spécification.

Spécification JML

- JML permet de définir des invariants *de classe* et *d'instance*,
- ainsi que des contrats avec pré et postconditions.
  - En JML un contrat s'appelle plutôt un *cas de spécification* (specification case)
  - alors qu'un *contrat* est l'ensemble des cas de spécification.

Spécification JML

- JML permet de définir des invariants *de classe* et *d'instance*,
- ainsi que des contrats avec pré et postconditions.
  - En JML un contrat s'appelle plutôt un *cas de spécification* (specification case)
  - alors qu'un *contrat* est l'ensemble des cas de spécification.

Spécification JML

- JML permet de définir des invariants *de classe* et *d'instance*,
- ainsi que des contrats avec pré et postconditions.
  - En JML un contrat s'appelle plutôt un *cas de spécification* (specification case)
  - alors qu'un *contrat* est l'ensemble des cas de spécification.

Spécification JML

- JML permet de définir des invariants *de classe* et *d'instance*,
- ainsi que des contrats avec pré et postconditions.
  - En JML un contrat s'appelle plutôt un *cas de spécification* (specification case)
  - alors qu'un *contrat* est l'ensemble des cas de spécification.

Modificateur privé/public

- JML emprunte à Java les modificateurs *private/public*.
- Similairement au fait qu'un attribut peut être public ou privé, une spécification JML peut elle-même être publique ou privée.
- Le fait d'être public ou privé ne change pas le sens de la spécification, mais JML interdit d'avoir une spécification publique d'un attribut privé.
- `/*@ spec_public @*/` permet de rendre un attribut privé accessible à une spécification publique.

Modificateur privé/public

- JML emprunte à Java les modificateurs *private/public*.
- Similairement au fait qu'un attribut peut être public ou privé, une spécification JML peut elle-même être publique ou privée.
- Le fait d'être public ou privé ne change pas le sens de la spécification, mais JML interdit d'avoir une spécification publique d'un attribut privé.
- `/*@ spec_public @*/` permet de rendre un attribut privé accessible à une spécification publique.

Modificateur privé/public

- JML emprunte à Java les modificateurs *private/public*.
- Similairement au fait qu'un attribut peut être public ou privé, une spécification JML peut elle-même être publique ou privée.
- Le fait d'être public ou privé ne change pas le sens de la spécification, mais JML interdit d'avoir une spécification publique d'un attribut privé.
- `/*@ spec_public @*/` permet de rendre un attribut privé accessible à une spécification publique.

Modificateur privé/public

- JML emprunte à Java les modificateurs *private/public*.
- Similairement au fait qu'un attribut peut être public ou privé, une spécification JML peut elle-même être publique ou privée.
- Le fait d'être public ou privé ne change pas le sens de la spécification, mais JML interdit d'avoir une spécification publique d'un attribut privé.
- `/*@ spec_public @*/` permet de rendre un attribut privé accessible à une spécification publique.

Modificateur privé/public

- JML emprunte à Java les modificateurs *private/public*.
- Similairement au fait qu'un attribut peut être public ou privé, une spécification JML peut elle-même être publique ou privée.
- Le fait d'être public ou privé ne change pas le sens de la spécification, mais JML interdit d'avoir une spécification publique d'un attribut privé.
- `/*@ spec_public @*/` permet de rendre un attribut privé accessible à une spécification publique.

Attribut `spec_public`

```
public class Chauffeur {
    protected /*@ spec_public @*/ int age;
    protected /*@ spec_public @*/ Camion camion;
}
```

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Comportements

- Un contrat JML peut prendre plusieurs formes :
  - `normal_behavior` indique que la méthode ne doit pas lever une exception,
  - `exceptional_behavior` indique que la méthode doit lever une exception,
  - `behavior` permet de combiner les deux.
- Il peut y avoir plusieurs contrats séparés par `also`.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

Clauses d'un contrat

- Pour un contrat `normal_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
    - indiquer `\nothing` s'il n'y rien de modifié.
    - cette clause absente est équivalent à indiquer `\everything`.
  - `ensures` une postcondition de la spécification,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
    - `diverges false`, la méthode doit toujours se terminer,
    - `diverges true`, la non-termination est permise.

normal_behavior

```
public class Camion { ...
    /*@
      public normal_behavior
      requires this != null;
      assignable chauffeur, c;
      ensures chauffeur == c && c.camion == this;
    @*/
    public void estConduitPar(Chauffeur c){
        chauffeur = c;
        chauffeur.setCamion(this);
    }
}
```

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

Clauses d'un contrat

- Pour un contrat `exceptional_behavior` on peut spécifier :
  - `requires` une précondition de la spécification,
  - `assignable` la liste des attributs qui peuvent changer durant l'exécution de la méthode,
  - `diverges` une condition qui doit être vraie avant l'exécution de la méthode, si elle ne se termine pas,
  - `signals` indique pour une Exception la postcondition qui doit être vérifiée si elle est levée,
  - `signals_only` les exceptions permises.

exceptional_behavior

```
public class Camion {...
/*@
 public exceptional_behavior
 requires poids + article.poids <= poidsMax;
 requires prochaineLivraison == noLivraisonsMax;
 signals (ExcCamionPlein) prochaineLivraison
          == \old(prochaineLivraison);
 signals_only ExcCamionPlein;
@*/
 public boolean charger(Article article,
                        Lieu destination)
         throws ExcCamionPlein {
...}
}
```

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Postcondition JML

- Dans une postcondition (`ensures`) on peut utiliser
  - `\result` pour la valeur de retour,
  - `\old(expression)` pour référer à l'ancienne valeur de `expression`.
    - Contrairement au `@pre` de OCL `\old` s'applique à toute l'expression et non seulement à la référence.

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Valeurs par défaut

- Par défaut, une clause absente sera interprétée de la façon suivante.
  - `requires true;`
  - `assignable \everything;`
  - `ensures true;`
  - `diverges false;`
  - `signals (Exception) true;`
  - `signals_only` “toutes les exceptions de la méthode”;

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.

Conclusion

- De façon générale, si les préconditions (`requires`) sont satisfaites avant l'appel à la méthode, alors :
  - pour un `normal_behavior` l'exécution doit se terminer normalement et toutes les postcondtions (`ensures`) doivent être satisfaites.
  - pour un `exceptional_behavior` l'exécution doit lever une des exceptions de `signals_only` (ou un de ses sous-types), et toutes les clauses `signals`, pour tous les *sur-types* de l'exception levée, doivent être satisfaites.
  - Dans tous les cas :
    - Au plus les emplacements indiqués dans les clauses `assignable` ont été modifiés.
    - Si l'appel ne se termine pas, alors la condition indiquée dans la clause `diverges` était vraie *avant* l'appel.