Contenu

1. Introduction
2. L'ADL Wright
    2.1 Concepts comportementaux de Wright
    2.2 Concepts structuraux de Wright
    2.3 Propriétés standard de Wright
    2.4 Outils d'analyse de Wright
    2.5 Évaluation
3. Travaux connexes
4. De Wright vers CSP
    4.1Test de l'outil Wr2fdr expérimentation de l'outil Wr2fdr
        4.1.1 Erreurs liées aux propriétés 2 et 3
        4.1.2 Erreurs liées à la propriété 1
        4.1.3 Erreurs liées à la propriété 8
    4.2 Extraction de la description architecturale de l'outil Wr2fdr
        4.2.1Règles de rétro-ingénierie de C++ vers UML
        4.2.2 Diagramme de classe de l'outil Wr2fdr
    4.3 Maintenances correctives proposées
        4.3.1 Corrections liées aux deux propriétés 2 et 3
        4.3.2 Corrections liées à la propriété 1
        4.3.3 Corrections liées à la propriété 8
    4.4 Maintenance évolutive proposée
4.5 Validation de l'outil Wr2fdr
5. De Wright vers Ada
    5.1 Règles de traduction proposées
        5.1.1 Traduction d'une configuration
        5.1.2 Traduction d'un événement observe
        5.2.3 Traduction d'un événement initialisé
        5.2.4 Traduction de l'opérateur de choix interne
        5.2.5 Traduction d'une configuration hiérarchique
    5.2 Réalisation de l'outil Wr2ada
        5.2.1 Méta-modèle partiel pour l'ADL Wright
        5.2.2 Méta-modèle partiel d'Ada
        5.2.3 Programmes de l'outil Wr2ada
    5.3 Validation de l'outil Wr2ada
6. Étude de cas : Gestion de places du parking
    6.1 Description informelle
    6.2 Modélisation formelle en Wright
    6.3 Analyse statique de la configuration Parking Space Management
    6.4 Analyse dynamique de la configuration Parking Space Management
7. Discussion/Évaluation
8. Conclusion
Références

Vérification et validation des descriptions architecturales en Wright

Mohamed Tahar Bhiri[1], Taoufik Sakka Rouis [2*], Mourad Kmimech [3]

[1] MIRACL Laboratory, ISIMS, University of Sfax, Sfax, Tunisia
[2] LIPAH Laboratory, FST, University of Tunis El Manar, Tunis, Tunisia
[3] UR-OASIS Laboratory, ENIT, University of Tunis El Manar, Tunis, Tunisia
[*] srtaoufik@yahoo.fr

Résumé: En génie logiciel, les techniques V&V (Vérification & Validation) sont essentielles pour l'obtention de systèmes corrects et répondant aux attentes du client. Dans ce papier, nous apportons deux outils complémentaires permettant la vérification et la validation des architectures logicielles abstraites décrites par l'ADL formel Wright. Le premier outil appelé Wr2fdr, permet de produire une spécification CSP de Hoare en partant d'une description architecturale Wright. La spécification CSP obtenue est soumise au model-checker FDR afin de vérifier des propriétés architecturales standards. Le second outil proposé est appelé Wr2ada. Ce dernier permet de produire un programme concurrent Ada à partir d'une description architecturale Wright. Le programme concurrent Ada peut être validé moyennant des scénarios définis par le client.

Keywords: Wright Software Architecture; Verification; Validation; Ada Concurrent Program.

1. Introduction

Le problème de vérification et de validation demeure un problème essentiel en génie logiciel. En effet, les activités de vérification et de validation sont deux étapes primordiales tout au long du cycle de développement des systèmes complexes. Elles permettent d'une part de vérifier qu'on a développé le bon système et de l'autre part, de s'assurer que le système est développé correctement. Ainsi, elles augmentent la confiance dans le système étudié. Plus spécifiquement, la vérification et la validation de l'architecture logicielle (AL) du futur système permettent la détection et par conséquent la correction des erreurs très tôt dans le cycle de développement. Ceci réduit les dépenses de développement. En effet, le coût de l'erreur est en général plus faible au niveau architectural que dans le code source. La vérification est souvent assurée par des outils formels comme les model-checkers et prouveurs. Quant à la validation est généralement assurée par des outils d'animation et de simulation. Dans ce papier, nous abordons ce problème de vérification et de validation d'architecture logicielle décrite par l'ADL formel Wright (Allen, 1997). Ce dernier est choisi pour ses aptitudes architecturales. En effet, Wright est l'un des premiers ADL qui permettent de décrire les aspects structuraux et comportementaux d'une architecture logicielle abstraite. En plus, l'ADL Wright est considéré comme un ADL de référence (Taylor et al., 2009 ; Zhang et al., 2010). Il permet de raisonner sur les architectures logicielles. En effet, il définit onze propriétés standard relatives à la vérification de la cohérence d'architectures logicielles. Parmi ces propriétés, quatre sont censées être automatisées par l'outil Wr2fdr. celui-ci accompagne l'ADL Wright. Cependant, la version initiale de l'outil Wr2fdr n'est pas opérationnelle. En plus, cet outil ne couvre pas toutes les tâches à satisfaire par un environnement de travail basique autour de l'ADL Wright. Parmi celles-ci, la validation et la vérification des propriétés spécifiques des AL décrites en Wright sont de première importance. Wright en tant que langage "généraliste" ne peut pas proposer des propriétés architecturales spécifiques, c'est-à-dire liées à un domaine d'application ou à des applications particulières. L'architecte qui désire vérifier (d'une façon statique) des propriétés architecturales spécifiques doit travailler sur des spécifications CSP produites par l'outil Wr2fdr. De plus, il faut que les propriétés à vérifier soient naturellement exprimables sous forme de raffinement CSP. Mais les propriétés spécifiques potentiellement vérifiables dans le cadre CSP ne couvrent pas toutes les classes de propriétés (Schnoebelen, 1999). En effet, CSP ne peut pas traiter naturellement les propriétés d'équité, les propriétés orientées état (invariant du système) et les propriétés qui incluent des événements et en excluent d'autres. Pour faire face à ces problèmes, nous proposons :

- La correction et l'extension de l'outil Wr2fdr développé initialement par et autour de David Garlan à l'université de Carnegie Mellon. Pour ce faire, nous avons dû fournir des efforts louables afin de comprendre un code contenant plusieurs milliers de

lignes C++, le corriger et implémenter des nouvelles fonctionnalités relatives aux propriétés à vérifier. La nouvelle version permet l'automatisation de quatre propriétés standard liées à la cohérence d'un composant, l'absence d'un blocage d'un connecteur, l'absence de blocage d'un rôle et la compatibilité port/rôle reliant les composants aux connecteurs d'une configuration. De telles propriétés sont formalisées par des relations de raffinement entre deux processus CSP, l'un est abstrait et l'autre est concret (ou raffiné). La nouvelle version de l'outil Wr2fdr sera indispensable pour la vérification des propriétés standard d'architectures logicielles Wright.

- La réalisation d'un second outil appelé Wr2ada. Ce dernier permet la traduction systématique d'une configuration abstraite Wright vers un programme concurrent Ada. Ceci permet la validation d'une architecture logicielle abstraite Wright en utilisant les outils d'analyse dynamique associés à Ada tels que le générateur de données de test et le débogage. En effet, l'exécution d'une architecture logicielle abstraite sur des données de test représentatives permet très tôt au client de participer à la validation de l'architecture de son futur système.

Cet article comporte huit sections. La section 2 présente les aspects fondamentaux de l'ADL Wright et de son outil Wr2fdr. Un soin particulier est apporté aux propriétés standards proposées par l'ADL Wright. La section 3 présente les principaux travaux relatifs à la vérification et la validation des architectures Wright. La section 4 présente une approche de maintenance corrective et évolutive de l'outil Wr2fdr utilisé dans notre processus de vérification d'AL Wright. La section 5 présente notre approche de validation des architectures Wright. Rappelant que cette approche est basée sur la réalisation d'un second outil appelé Wr2ada. Ce dernier permet la transformation d'une configuration Wright vers un programme concurrent Ada. La section 6 présente une validation de notre approche de vérification et de validation sur une étude de cas. Une évaluation comparative de ces deux outils est effectuée dans la section 7. Enfin, la section 8 présente une conclusion et des perspectives.

2. L'ADL Wright

L'ADL formel Wright (Allen, 1997) (Garlan, 2003) permet de décrire les aspects structuraux et comportementaux d'une architecture logicielle abstraite. Les aspects structuraux sont décrits grâce à des constructions architecturales offertes par Wright telles que : Component, Connector, Port, Role, Glue, Computation et Configuration. Tandis que les aspects comportementaux sont décrits en CSP de Hoare (Hoare, 1978).

2.1. Concepts structuraux de Wright

Les concepts structuraux de Wright sont assez similaires aux concepts proposés par la plupart des ADL : Component, Connector, Configuration et Style. Dans ce contexte, un composant Wright est une unité de traitement abstraite localisée et indépendante. La description d'un composant contient deux parties importantes : l'interface et la partie calcul (Computation). La partie interface composée d'un ensemble de ports (Port) qui fournissent les points d'interaction entre le composant et son environnement. La partie computation, quant à elle, consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Cependant un connecteur Wright représente une interaction explicite et abstraite entre une collection de composants. Le connecteur contient deux parties importantes : une interface constituée de points d'interactions appelés rôles (Role) et une partie qui représente la spécification d'assemblages (Glue). Le rôle indique comment se comporte un composant qui participe à l'interaction. La glu spécifie les règles d'assemblage entre un ensemble de composants pour former une interaction.

Après la déclaration des instances de composants et de connecteurs, la configuration Wright permet de décrire l'architecture d'un système en attachant les instances de composants avec les instances de connecteurs.

L'ADL Wright permet de décrire les styles architecturaux. Les contraintes associées aux styles sont décrites à l'aide d'un langage de contraintes simple spécifique à Wright. Un tel langage de contraintes englobe des opérateurs logiques du premier ordre et des opérateurs propres à Wright tels que: "Components", "Connectors", "Ports", "Roles", "Type", "Name", etc. Chaque opérateur apporte des informations liées à la configuration architecturale traitée.

La Figure 1 présente une simple description Wright qui illustre l'utilisation des différents concepts Wright. En effet, cette description comporte la définition d'un style appelé PipeFiltre et une configuration qui dérive ce style. Ce dernier propose deux contraintes.

```
Style PipeFilter
 Connector Pipe
  Role Source =
//CSP Expression
  Role Sink   =
//CSP Expression
  Glue        =
//CSP Expression
 Component Filter
  Port Input  =
//CSP Expression
  Port Output =
//CSP Expression
  Computation =
//CSP Expression
 Constraints
  ∀ c : Connectors •
      Type (c) = Pipe  ∧
 ∀ c1 : Components •
      Type (c1) = Filter
End Style
```

```
Configuration System1
 Style PipeFilter
    // components declaration
    // connectors declaration

Instances
  Split,Upper, Merger : Filter
  P1, P2 : Pipe

 Attachments
  Split.Output As P1.Source
  Upper.Input As P1.Sink
  Upper.Output As P2.Source
  Merger.Input As P2.Sink
End Configuration
```

Figure 1: The structure of a Wright description

2.2. Concepts comportementaux de Wright

Wright est l'un des premiers ADL permettant la description de l'aspect comportemental de ces éléments architecturaux. En effet, le comportement d'un composant (respectivement d'un connecteur) est décrit localement à

travers les ports (respectivement les rôles) et globalement à travers le computation (respectivement glue). Les auteurs de Wright proposent de décrire le comportement d'un composant Wright (respectivement d'un connecteur) par des processus CSP. Rappelant que dans le modèle CSP, tout est représenté par des événements. Un événement correspond à un moment ou une action qui présente un intérêt. CSP ne fait pas la distinction entre les événements initialisés et observés. Tandis que CSP de Wright le fait : Un événement initialisé s'écrit sous la forme (ē ou _e); Un événement observé est noté (e). Grâce à cette distinction, il nous est possible de mieux contrôler l'interaction en sachant quel composant a initialisé l'événement de ceux qui l'observent.

Pour définir un comportement, il faut pouvoir combiner ces événements. Un processus correspond à la modélisation du comportement d'un objet par la combinaison d'événements avec d'autres processus simples. Les principaux opérateurs fournis par CSP sont : l'opérateur de préfixage noté (->) , L'opérateur de choix externe ou déterministe noté (□ ou []) l'opérateur de choix interne ou non déterministe noté (⊓ ou |~|), l'opérateur de composition parallèle noté (||), etc. En complément de la notation standard de CSP, Wright introduit une nouvelle notation notée (§). Cette notation désigne le processus de terminaison avec succès, ce qui veut dire que le processus s'est engagé dans un événement succès √ et s'est arrêté. Formellement, § = √ -> STOP (En CSP il est généralement noté «SKIP»).

2.3. Propriétés standard de Wright

Les auteurs de l'ADL Wright proposent onze propriétés standard liées à la cohérence et complétude des architectures logicielles décrites en Wright (Allen, 1977). Ces propriétés sont présentées informellement comme suit :

- **Propriété 1: Cohérence d'un composant**

La spécification de port doit être une projection du calcul (computation), sous la condition que l'environnement obéit à la spécification de tous les autres ports. Intuitivement, la propriété 1 indique que le composant ne se soucie pas des événements non couverts par les ports.

- **Propriété 2: Connecteur sans interblocage**

La glue d'un connecteur interagissant avec les rôles doit être **sans interblocage**. La description du connecteur doit vérifier que la coordination des rôles par la glue est cohérente avec le comportement attendu des composants. Sachant qu'un processus CSP est dit en situation de blocage quand il peut se retirer à tout événement, mais n'a pas encore terminé correctement (en participant à l'événement §). Inversement, un processus est sans blocage s'il ne peut jamais être en situation de blocage.

- **Propriété 3: Rôle sans interblocage**

Chaque rôle d'un connecteur doit être **sans interblocage**. Une autre catégorie d'incohérence est détectable comme une situation de blocage, lorsque la spécification d'un rôle est elle-même incohérente. Dans une spécification d'un rôle complexe, il peut y avoir des erreurs qui conduisent à une situation dans laquelle aucun événement n'est possible pour le participant, même si la glu était prête à prendre n'importe quel événement.

- **Propriété 4: Un seul initialiseur**

Dans une spécification de connecteur, tout événement doit être initialisé par un rôle ou une glu. Tous les autres processus doivent soit l'observer, soit l'oublier (grâce à leur alphabet).

- **Propriété 5: L'engagement d'initialisation**

Si un processus initialise un événement, il doit s'engager à cet événement sans être influencé par l'environnement. La propriété 5 garantit que les concepts d'initialisation et d'observation des événements sont utilisés correctement.

- **Propriété 6: Paramètres de substitution**

Une déclaration d'instance définissant un type doit résulter de ce type de validation après avoir remplacé tous les paramètres formels manquants. Pour le paramètre numérique, nous devons nous assurer que les paramètres donnés entrent dans les limites dans la description du type.

- **Propriété 7: Test sur les valeurs d'intervalle**

Un paramètre numérique ne doit pas être inférieur à la limite inférieure (si elle est déclarée) et ne doit pas être supérieur à la limite supérieure (si elle est déclarée).

- **Propriété 8: Compatibilité port / rôle**

Tout port attaché à un rôle doit toujours poursuivre son protocole dans une direction que le rôle peut avoir. Au niveau des liens, la question qui se pose est: Quels ports peuvent être utilisés pour ce rôle?

La vérification du fait que les protocoles du port et rôle sont identiques n'est pas suffisante. En effet, nous voulons avoir la capacité d'attacher un port qui n'a pas un protocole identique au rôle. Considérons le rôle suivant:

```
 Role Source = _write!X ->
Source[]_close -> §
```

Le rôle Source peut être attaché au port suivant :

```
 Port Output = _write!1 -> _write!2 ->
_write!3 -> _close -> §
```

Le rôle Source et le port Output ne sont pas identiques. En effet, le rôle Source qui émet la séquence d'un caractère (X) à une description plus générale du port Output émettant la séquence du caractère suivant (1 2 3). D'autre part, on peut vérifiez qu'il n'y a pas d'incompatibilité entre le rôle et le port attachés. Par exemple, nous n'acceptons pas le fait qu'un port BadOutput (sans l'événement close) peut être attaché au rôle Source.

```
Port BadOutput = _write!X -> BadOutput []
§
```

- **Propriété 9: Contraintes de style**

Les prédicats de style doivent être vrais pour une configuration déclarée dans ce style.

Les contraintes de style doivent être cohérentes.

Exemple:

```
      ∀c : Component; p : Ports(c) •
Type(p) = DataOutput
```

```
        ∃ c : Component; p : Ports(c) •
Type(p) = DataInput
```

Ces deux contraintes sont incohérentes, donc le style contenant ces deux contraintes est incohérent. En effet, le premier prédicat décrit que tous les ports des composants sont de type DataOutput. Cependant, le second prédicat dit qu'il existe un port d'un composant dont le type est DataInput.

- **Propriété 10: Cohérence de style**

Au moins une configuration doit satisfaire les contraintes de style.

La Figure 2 montre une configuration qui ne satisfait pas les contraintes spécifiées au niveau style. En effet, la deuxième contrainte spécifiée dans le style PipeFilter n'est pas satisfaite par la configuration System.

```
Style PipeFilter
Component Filter
  Port Input  = //CSP Expression
  Port Output  = //CSP Expression
  Computation = //CSP Expression
Connector Pipe
 Role Source = //CSP Expression
 Role Sink = //CSP Expression
 Glue = //CSP Expression
Constraints
 ∃      c : Connectors; r : Roles(c) •
Type(r) = Sink
 ∀ c : Components • Type (c) = Filter
End Style

Configuration System Style PipeFilter
 Component Filter2
  Port Input  = //CSP Expression
  Port left =//CSP Expression
  Port right =//CSP Expression
  Computation =//CSP Expression
 Instances
  Split, Merger: Filter2
  Upper: Filter
  P1, P2, P3, P4: Pipe
 Attachments
…
End Configuration
```

Figure 2. Exemple d'une configuration Wright qui dérive d'un style

- **Propriété 11: Exhaustivité des liens**

Chaque port et chaque rôle non attaché dans la configuration doit être compatible avec le processus de terminaison avec succès (noté §).

2.4. Outils d'analyse de Wright

Les auteurs de Wright proposent un outil logiciel indispensable pour l'utilisation de l'ADL Wright. Cet outil, appelé Wr2fdr (ABLE, 2005), permet d'automatiser les quatre propriétés (1, 2, 3 et 8) décrites précédemment. Pour y parvenir, l'outil Wr2fdr traduit une spécification Wright en une spécification CSP dotée des relations de raffinement à vérifier. La spécification CSP engendrée pour l'outil Wr2fdr est soumise à l'outil de model-checking FDR (Failure-Divergence Refinement) (FDR2, 2012). Ce dernier permet de vérifier de nombreuses propriétés sur des systèmes d'états finis. En effet, FDR s'appuie sur la technique de model-checking (Schnoebelen, 1999). Celle-ci effectue la vérification d'un modèle d'un système par rapport aux propriétés qui sont attendues sur ce modèle. Cette vérification est entièrement automatisée et consiste à explorer tout l'espace d'états.

2.5. Évaluation

Hormis les concepts architecturaux, Wright permet de décrire les aspects comportementaux d'une architecture logicielle abstraite. Pour y parvenir, il réutilise avec adaptation un sous-ensemble de CSP de Hoare. Celui-ci est très élégant et bien défini mathématiquement. En outre, Wright définit des propriétés standard permettant de raisonner sur la cohérence et complétude d'une architecture logicielle. Les propriétés 1, 2, 3 et 8 précédemment présentées sont formalisées à l'aide de la technique de raffinement CSP. L'outil Wr2fdr qui accompagne l'ADL Wright est censé automatiser ces quatre propriétés en vue de les vérifier grâce au model-checker FDR2. Cependant, suite à des expérimentations avec l'outil Wr2fdr, nous avons constaté des défaillances liées à l'implémentation de ces quatre propriétés standard. Vu l'importance de cet outil, nous avons contacté les auteurs de Wright, expliqué les problèmes rencontrés et récupérer le code source decet outil (de l'ordre de 16000 lignes C ++). Depuis 2007, un ensemble de maintenances liées à la mise en œuvre de ces propriétés standard est proposé dans la section 4.3. De plus, afin d'améliorer cet outil, la section 4.4 présente une augmentation de son analyseur par un analyseur sémantique de Wright.

3. Travaux Connexes

Les travaux permettant d'établir des transformations automatiques entre des ADL formels et les outils de vérification et de validation ne sont pas nombreux. Dans cette partie on se limite aux travaux liés à l'ADL formel Wright.

Jeffrey et Kuang, (1999) proposent une approche permettant de traduire les constructions Wright en réseaux de Petri. Ensuite, les réseaux de Petri sont traduits dans les langages d'entrée des outils comme SPIN et SMV. Le travail décrit dans (Naumovich et al., 1997) explore l'utilisation des outils d'analyse statique liés au langage concurrent Ada afin de vérifier des propriétés spécifiques à des architectures abstraites Wright. En effet, les auteurs de ce travail proposent une traduction manuelle sans règles explicites de Wright vers Ada. Ces règles spécifiques sont validées sur un système de Station de Gaz (Gas Station system) (Helmbold, Luckham, 1985). Cette traduction manuelle favorise la vérification de l'architecture source en se servant des outils d'analyse statique Ada. En effet, plusieurs propriétés spécifiques au système de Station de Gaz sont vérifiées en utilisant les outils d'analyse statique : FLAVERS (Cobliegh et al., 2002) et INCA (Corbette, Avrunin, 1995). Parmi ces propriétés, nous citons : "Customers get gas in the order they pay, no free gas and customers get the right amount of gas".

Dans des travaux antérieurs (Bhiri et al., 2008 ; Bhiri et al., 2012), nous avons établi des règles permettant de traduire d'une façon systématique une architecture logicielle Wright vers Ada. Les règles proposées couvrent la plupart des constructions de Wright. Ces constructions sont : composant atomique, connecteur atomique, processus CSP et configuration. De telles règles sont décrites sous formes de patrons illustrés par des exemples. Notre traduction systématique de Wright vers Ada ignore les données portées par les événements. En effet, dans CSP de Wright, les données ne sont pas typées explicitement. De plus, le code Ada produit par notre traduction est destiné entre autres à être exécuté ; d'où le problème de choix des valeurs des paramètres in. Les règles de traduction de Wright vers Ada établies dans nos travaux antérieurs ont été appliquées avec succès sur le système de Station de Gaz traité dans (Naumovich et al., 1997).

4. De Wright vers CSP

Rappelant que l'outil Wr2fdr est un outil logiciel indispensable pour l'utilisation de l'ADL Wright. Il permet d'automatiser les quatre propriétés 1, 2, 3 et 8 (voir Section 2.3). Les deux propriétés 2 et 3 concernent la construction connecteur de Wright. Tandis que la propriété 1 touche à la construction composant de Wright. Enfin, la propriété 8 est relative à la construction configuration de Wright. De telles propriétés sont formalisées par des relations de raffinement entre deux processus CSP, l'un est abstrait et l'autre est concret (ou raffiné). La version initiale de l'outil Wr2fdr n'est pas opérationnelle. Depuis 2007, nous avons récupéré le code source de l'outil Wr2fdr : de l'ordre de 16000 lignes C++. Nous avons apporté plusieurs améliorations importantes à cet outil aussi bien correctives qu'évolutives. Pour y parvenir, nous avons dû, en premier temps, combiner les deux activités de test fonctionnel et structurel afin de valider l'outil Wr2fdr. Ceci permet de couvrir ces quatre propriétés et par conséquent de détecter les erreurs. Ensuite, une démarche de retro-ingénierie a été proposée. Ceci permet l'extraction d'une architecture logicielle pour l'outil Wr2fdr. Finalement, en se basant sur cette architecture extraite, nous avons apporté des maintenances correctives et évolutives à l'outil Wr2fdr. La nouvelle version a été bien validée sur différents études de cas. Dans la suite, nous allons décrire la démarche suivie afin de faire corriger et évaluer l'outil Wr2fdr.

4.1. Expérimentations de l'outil Wr2fdr

4.1.1. Erreurs liées aux propriétés 2 et 3

Pour vérifier le comportement de l'outil Wr2fdr vis-à-vis de ces deux propriétés, nous avons testé cet outil par des données de test (DT) sous forme des connecteurs Wright. Et, nous avons détecté la présence de plusieurs erreurs syntaxiques dans les fichiers CSP générés par l'outil Wr2fdr.

Pour illustrer ces erreurs, nous proposons de prendre le connecteur «Pipe» présenté par la Figure 3 comme DT.

```
Style PipeConn
 Connector Pipe
  Role Writer = write -> Writer |~|
close -> TICK
  Role Reader = DoRead |~| ExitOnly
    where {
       DoRead = read -> Reader []
readEOF -> ExitOnly
       ExitOnly = close -> TICK }
  Glue = Writer.write -> Glue []
Reader.read -> Glue
  [] Writer.close -> ReadOnly []
Reader.close -> WriteOnly

    where {
      ReadOnly = Reader.read -> ReadOnly
      [] Reader.readEOF -> Reader.close
-> TICK
      [] Reader.close -> TICK
      WriteOnly = Writer.write ->
WriteOnly [] Writer.close
      -> TICK
      }
  Constraints
      // no constraints
  End Style
```

Figure 3. La donnée de test PipeConn.wrt

L'exécution de l'outil Wr2fdr avec la donnée de test présentée par la Figure 3 a donné naissance au fichier CSP présenté par la Figure 4. Cependant, lors de la vérification du ce fichier CSP généré, nous remarquons que le model-checker FDR2 rencontre des problèmes (voir Figure 5) visiblement d'ordre syntaxique. En effet, un examen du code CSP généré montre les deux erreurs suivantes :

- L'équation relative au rôle «Writer» -coloré en gris clair dans la Figure 4- est syntaxiquement incorrecte car une équation récursive doit avoir la forme suivante : P= x -> P. De même pour la seconde équation relative au rôle «Reader» (**erreur 1**).
- L'identificateur ALPHA_Glue -coloré en gris foncé dans la Figure 4- qui devrait matérialiser un ensemble d'événements est non défini (**erreur 2**).

```
-- FDR compression functions
transparent diamond
transparent normalise
-- Wright defined processes
channel abstractEvent
DFA = abstractEvent -> DFA |~| SKIP
quant_semi({},_) = SKIP
quant_semi(S,PARAM) = |~| i:S @ PARAM(i)
; quant_semi(diff(S,{i}),PARAM)
power_set({}) = {{}}
power_set(S) = { union(y,{x}) | x <- S,
y <- power_set(diff(S,{x}))}
-- Style PipeConn
-- events for abstract specification
channel readEOF, read, close, write
-- Connector Pipe
-- generated definitions (to split long
sets)
ALPHA_Pipe = {|Reader.readEOF,
Reader.read, Reader.close, Writer.write,
Writer.close|}
```

```
ReadOnly = ((Reader.read -> ReadOnly) []
((Reader.readEOF -> (Reader.close ->
SKIP)) [] (Reader.close -> SKIP)))
WriteOnly = ((Writer.write -> WriteOnly)
[] (Writer.close -> SKIP))
Glue = ((Writer.write -> Glue) []
((Reader.read -> Glue) [] ((Writer.close
-> ReadOnly) [] (Reader.close ->
WriteOnly))))
-- Rôle Writer
ALPHA_Writer = { close, write}
ROLEWriter = ((write -> Writer) |~|
(close -> SKIP))
WriterA = ROLEWriter [[ x <-
abstractEvent | x <- ALPHA_Writer ]]
assert DFA [FD= WriterA
-- Rôle Reader
ALPHA_Reader = {readEOF, read, close}
DoRead = ((read -> Reader) [] (readEOF -
> ExitOnly))
ExitOnly = (close -> SKIP)
ROLEReader = (DoRead |~| ExitOnly)
ReaderA = ROLEReader [[ x <-
abstractEvent | x <- ALPHA_Reader ]]
assert DFA [FD= ReaderA
channel Writer: {close, write}
channel Reader: {readEOF, read, close}
Pipe = ( (ROLEWriter[[ x <- Writer.x | x
<- {close, write } ]]
[| diff({|Writer|}, {}) |]
(ROLEReader[[ x <- Reader.x | x <-
{readEOF, read, close } ]]
[| diff({|Reader|}, {}) |]
Glue)) )
PipeA = Pipe [[ x <- abstractEvent | x
<- ALPHA_Glue ]]
assert DFA [FD= PipeA
-- End Style
```

Figure 4. Le fichier PipeConn.fdr2 généré

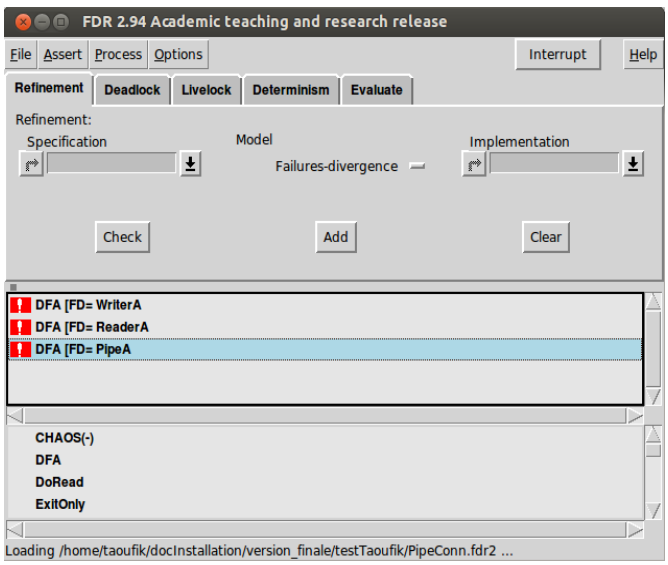


Figure 5. Vérification du fichier PipeConn.fdr2 par l'outil FDR2

4.1.2. *Erreurs liées à la propriété 1*

Pour vérifier l'implémentation de la propriété 1 liée à la cohérence de composant, nous avons exécuté l'outil Wr2fdr avec la donnée de test fournie par la Figure 6. Cette donnée de test correspond à un seul composant appelé Double qui propose deux ports. L'exécution de l'outil Wr2fdr sur cette donnée de test provoque l'arrêt brutal de l'outil Wr2fdr avec l'erreur d'exécution "erreur de segmentation". Ce type d'erreur est lié à l'utilisation d'un pointeur qui pointe nulle part.

```
Style Double
 Component Double
  Port In = read -> In [] close -> TICK
  Port Out = _write -> Out |~| _close -> TICK
  Computation = In.read -> _Out.write ->
Computation []
   In.close -> _Out.close -> TICK
 constraints
  //no constraints
End Style
```

Figure 6. La donnée de test Double.wrt

4.1.3. *Erreurs liées à la propriété 8*

Pour pouvoir vérifier le comportement de l'outil Wr2fdr vis-à-vis l'implémentation de la propriété 8, les données de test doivent être des fichiers Wright contenant des configurations avec notamment les clauses Instances et Attachments. Par exemple, la Figure 7 présente une Configuration Wright fictive. Cette configuration est utilisée comme donnée de test pour couvrir la propriété 8 (cohérence port/rôle).

L'exécution de l'outil Wr2fdr avec cette donnée de test provoque le même arrêt brutal rencontré dans la section précédente. En outre, le fichier CSP généré ne comporte que les deux assertions liées à la propriété 2 et l'assertion liée à la propriété 3 (voir Figure 8). Ainsi, la génération des relations des assertions liées à la propriété 8 n'est pas implémentée par la version actuelle de l'outil Wr2fdr.

```
Configuration ABC
 Component Atype
  Port Output = _a -> Output |~| TICK
  Computation = _Output.a -> Computation
|~| TICK
 Component Btype
  Port Input = c -> Input [] TICK
  Computation = Input.c -> _b ->
Computation [] TICK
 Connector Ctype
  Role Origin = _a -> Origin |~| TICK
  Role Target = c -> Target [] TICK
  Glue = Origin.a -> _Target.c -> Glue
[] TICK
 Instances
  A : Atype
  B : Btype
  C : Ctype
 Attachments
  A.Output as C.Origin
  B.Input as C.Target
End Configuration
```

Figure 7. La donnée de test ABC.wrt

```
-- FDR compression functions
transparent diamond
transparent normalise
-- Wright defined processes
channel abstractEvent
DFA = abstractEvent -> DFA |~| SKIP
quant_semi({},_) = SKIP
quant_semi(S,PARAM) = |~| i:S @
PARAM(i) ; quant_semi(diff(S,{i}),PARAM)
power_set({}) = {{}}
power_set(S) = { union(y,{x}) | x <- S,
y <- power_set(diff(S,{x}))}
-- Configuration ABC
-- Types declarations
-- events for abstract specification
channel b, c, a
-- Connector Ctype
-- generated definitions (to split long
sets)
ALPHA_Ctype = {|Target.c, Origin.a|}
GlueCtype = ((Origin.a -> (Target.c ->
GlueCtype)) [] SKIP)
ALPHA_Origin = {a}
ROLEOrigin = ((a -> ROLEOrigin) |~| SKIP)
OriginA = ROLEOrigin [[ x <-
abstractEvent | x <- ALPHA_Origin ]]
assert DFA [FD= OriginA
ALPHA_Target = {c}
ROLETarget = ((c -> ROLETarget) [] SKIP)
TargetA = ROLETarget [[ x <-
abstractEvent | x <- ALPHA_Target ]]
assert DFA [FD= TargetA
channel Origin: {a}
channel Target: {c}
Ctype = (( ROLEOrigin[[ x <- Origin.x |
x <- {a } ]]
[| diff({|Origin|}, {}) |]
( ROLETarget[[ x <- Target.x | x <-
{c } ]]
[| diff({|Target|}, {}) |]
GlueCtype)))
CtypeA = Ctype [[ x <- abstractEvent | x
<- ALPHA_Ctype ]]
assert DFA [FD= CtypeA
-- Currently, the rest of the
configuration is not shown in FDR.
-- End Configuration
```

Figure 8. Partie du fichier ABC.fdr2 généré

4.2. *Extraction de la description architecturale de l'outil Wr2fdr*

Pour pouvoir corriger les erreurs liées à l'implémentation de deux propriétés 2 et 3 et implémenter les deux propriétés 1 et 8, nous devons travailler sur le code C++ de l'outil Wr2fdr. Pour y parvenir, nous avons dû adopter la technique de retro-ingénierie afin d'extraire l'architecture statique de l'outil Wr2fdr.

Le diagramme de classes UML est choisi pour représenter cette architecture extraite. Cette tâche d'extraction est assez difficile. En effet, l'outil Wr2fdr est écrit en C++. Son code source est réparti physiquement sur plusieurs fichiers : trois fichiers «.hpp» et huit fichiers «.cpp». La complexité textuelle de l'outil Wr2fdr est de l'ordre de 16000 lignes C++. L'outil Wr2fdr englobe un analyseur lexico-syntaxique de Wright développé en utilisant les deux générateurs d'analyseurs lexicaux et syntaxiques célèbres Lex et Yacc (Hahne et Sato, 1994) (John et al., 1992).

Pour réaliser cette extraction, nous proposons des règles plus ou moins systématiques permettant la transformation d'un programme C ++ en diagramme de classes UML.

4.2.1. *Règles de rétro-ingénierie de C++ vers UML*

Dans la suite, nous allons proposer des règles de transformation de C++ vers UML illustrées par des exemples issus du code source de l'outil Wr2fdr.

Règle 1:

Un fichier d'entête C++ (fichier d'extension .hpp) contenant la déclaration de plusieurs classes est traduit par un package UML. Par exemple, Le fichier Wr2fdr.hpp contient plusieurs classes telles que AstNode, AstList, Name, Style, Configuration et Component est traduit par un package UML.

Règle 2:

Une classe C ++ est traduite en classe UML. Cette classe UML doit fournir les membres de classe (attributs et méthodes) proposées par la classe source C ++. Les paramètres et les restrictions d'accès (publics, protégés, etc.) des membres de la classe doivent être conservés dans la traduction.

Règle 3:

Une relation client C++ matérialisée par un attribut public à base d'une autre classe est traduite par une association unidirectionnelle UML. La cardinalité de cette association est proportionnelle à l'attribut concerné (unique, tableau, vecteur, etc.). Pour être considéré comme une agrégation ou une composition, nous devons analyser davantage le code C++.

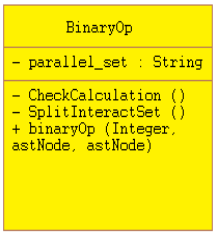
Règle 4:

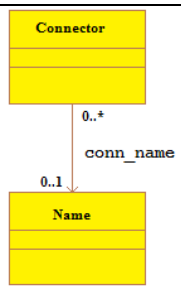
Une relation d'héritage C ++ est traduite par une généralisation UML. De façon systématique, une interface C ++ (ou classe abstraite) peut être traduite en classe abstraite UML.

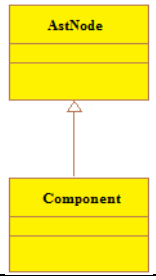
Le tableau 1 montre un ensemble d'exemples qui illustrent comment utiliser ces règles de traduction pour extraire une modélisation UML à partir d'un code C ++.

Tableau 1. Modélisation du code C++ en UML

Code C++	**Modélisation en UML**

`class binaryOp : public astNode {`
`private:`
`char *parallel_set;`
`void CheckCalculations(void);`
`void SplitInteractSet(Set *long_set);`
`public:`
`binaryOp(int, astNode*,astNode*);`
	

…
};
class name {
public:
…
};
 	
class connector {
public:
name *conn_name;
…
};
	

class component : public astNode {
public:
…
};
	

4.2.2. Diagramme de classe de l'outil Wr2fdr

En utilisant les règles de traduction présentées dans la section précédente, nous avons établi un diagramme de classe UML pour l'outil Wr2fdr (voir Figure 9). Ce diagramme regroupe les principales constructions structurelles (Component, Connector, Configuration et Style) et comportementales offertes par l'ADL Wright. Mise à part, les concepts métier offerts par l'ADl Wright, l'outil Wr2fdr offre plusieurs classes non métier : classes d'implémentation, classes de conception, etc. Parmi ces classes d'implémentation nous citons les classes : SList, Set, SymEntry, Relation et LookupTable qui correspondent à des classes C++ permettant l'implémentation des structures de données non génériques. Parmi les classes de conception nous citions :

- La classe AstNode : peut être considérée comme la classe mère de toute construction Wright. En effet, toutes les constructions syntaxiques offertes par l'ADL Wright dérivent de cette classe. En outre, cette classe offre plusieurs attributs et méthodes qui permettent de matérialiser une spécification Wright sous forme d'un arbre abstrait.
- Classe AstList : descend de la classe abstraite AstNode. Elle permet de regrouper sous forme d'une liste linéaire des objets appartenant à des classes qui dérivent directement ou indirectement de la classe AstNode. La classe AstList incarne une liste linéaire polymorphe et offre des opérations standards sur les listes linéaires telles que ajouter, inverser, suivant, copier, et des opérations spécifiques liées à la manipulation des constructions syntaxiques de l'ADL Wright.
- La classe Name : permet de mémoriser l'identification d'un concept Wright et ses paramètres éventuels. En outre, la classe Name offre des services standards et spécifiques permettant de gérer les aspects liés à l'identification d'un concept Wright.

Concernant les concepts structuraux issus de l'ADL Wright, l'outil Wr2fdr propose les quatre classes suivantes : Component, Connector, Configuration et Style. Ces concepts sont implémentés par des classes C++ qui dérivent directement de la classe AstNode. Chaque classe structurelle définit ses propres membres (attributs et méthodes) et surcharge les méthodes héritées de la classe mère AstNode. Par exemple :

- La classe Component permet l'implémentation d'un composant Wright. Cette classe propose les attributs suivants : comp_name, params, ports et computation. Ces attributs mémorisent respectivement des références sur des objets de type Name, AstList et AstNode. De plus, la classe Component fournit des implémentations aux méthodes abstraites copy, eq, wrprint, fdrprint, calculationPass, postCalculationPass et SplitLongSets venant de la classe ascendante AstNode.
- La classe Connector permet l'implémentation d'un connecteur Wright. Cette classe propose les attributs suivants : name, params, roles et glue mémorisant respectivement des références sur des objets de type Name, AstList, AstList et Declaration. En outre, cette classe propose des implémentations aux méthodes copy, eq, wrprint, fdrprint, CalculationPass, PostCalculationPass venant de la classe AstNode.
- La classe Style apporte trois nouveaux attributs : style_name, types et constraint mémorisant des références sur des objets de type Name, AstList et AstNode.
- La classe Configuration propose de nouveaux attributs conf_name, style_name, types, instances, attachments mémorisant respectivement des références sur des objets de type Name, Name, AstList et AsList. En outre, elle implémente les méthodes abstraites suivantes : copy, eq, wrprint, fdrprint, CalculationPass et PostCalculationPass.

Concernant les concepts comportementaux issus d'une spécification CSP, l'outil Wr2fdr propose les six classes C++ suivantes : Declaration, Event, BinaryOp, UnitaryOp, AstTerminal et Where. Ces classes dérivent directement de la classe AstNode. La signification de chaque classe comportementale est la suivante :

- La classe Declaration permet la définition d'un processus CSP. Cette classe propose deux attributs principaux n et defn mémorisant des objets de type AstNode. Ces deux objets sont liés respectivement à l'identité et à la définition du processus concerné.
- La classe Event permet l'implémentation d'un événement CSP. Cette classe propose un attribut principal mémorisant un objet de type Name. En outre, elle apporte un service AddPrefixToEvent permettant d'associer l'événement au processus concerné.
- La classe BinaryOp permet l'implémentation des opérateurs binaires CSP tels que : Choix déterministes ([ ]), choix non déterministe (| ~ |) et préfixage (−>). Cette classe propose deux attributs first et second mémorisant des objets de type AstNode. Ces deux attributs modélisent respectivement les deux

opérandes gauche et droite de l'opérateur binaire concerné

- La classe UnitaryOp permet l'implémentation d'un événement initialisé CSP. Elle apporte un attribut first de type AstNode mémorisant un événement perçu comme initialisé.
- La classe AstTerminal permet l'implémentation des terminaux des processus CSP dans une spécification Wright. Comme terminaux nous citons TICK, √ et STOP indiquant une bonne terminaison d'un processus.
- La classe Where correspondant à l'implémentation de la construction Where de CSP de Hoare. Celle-ci permet de nommer un sous-processus CSP. Elle apporte deux attributs mémorisant deux objets respectivement de type AstNode et AstList.

Le fonctionnement général de l'outil Wr2fdr est décrit par une séquence d'opérations. Dans un premier temps, l'opération parse_result est exécutée afin d'analyser syntaxiquement le fichier d'entrée contenant une spécification Wright. En cas de succès, cette opération produit un arbre syntaxique abstrait (Structure de données astNode). En cas d'échec, des erreurs lexico-syntaxiques sont signalées. Dans un deuxième temps, l'opération fdrprint applicable sur un objet de type astNode est exécutée afin de produire la traduction CSP correspondante.

La Figure 9 regroupe sous forme d'un diagramme de classes UML les principaux classes d'implémentation et les classes métier constituant le code source de Wr2fdr écrit en C++. Un attribut d'une classe A qui matérialise une référence sur un objet d'une autre classe B est modélisé en UML par une association qui relie ces deux classes. Cette association porte le même nom proposé par l'attribut.

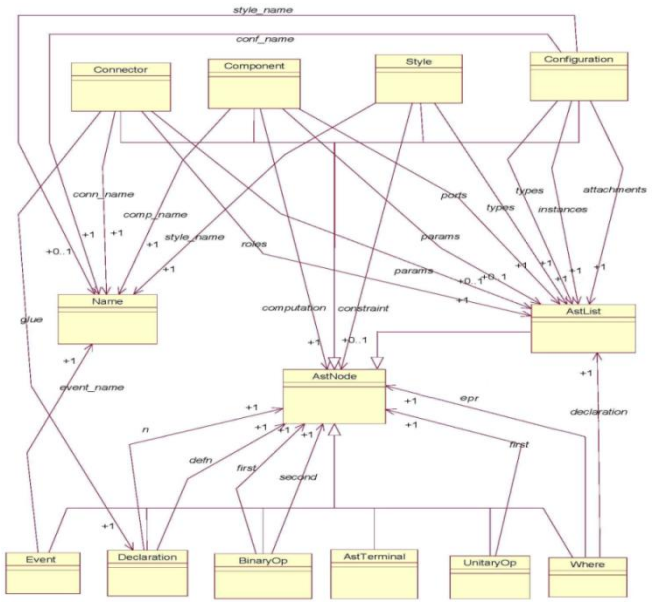


Figure 9. Diagramme de classes de l'outil Wr2fdr

4.3. Maintenances correctives proposées

Dans cette section, nous allons apporter des améliorations significatives à la version initiale de l'outil Wr2fdr.

4.3.1. Corrections liées aux deux propriétés 2 et 3

En se basant sur l'activité de retro-ingénierie (reverse-engineering) effectuée sur le code de l'outil Wr2fdr, nous avons pu localiser la classe et les méthodes entraînant les deux erreurs détectées dans la Section 4.1.1. Concernant la première erreur, lors de la génération du fichier FDR2; la méthode fdrprint implémentée dans la classe Name traite de la même manière les noms associés aux concepts Composant, Connecteur, Configuration, Style, Événement, Port et Rôle; ce qui est complètement faux. Pour corriger cette première erreur, nous avons proposé d'ajouter une structure de contrôle (switch) pour distinguer ces différents concepts architecturaux. Par exemple, dans le cas d'un Rôle, l'injection de son nom est précédée du mot «ROLE». De même pour le concept Port.

La Figure 10 présente un extrait de la version corrigée de la méthode fdrprint.

```
void Name::fdrprint(void) {
…
switch (higherScope_effectif->gtype) {
 case ROLE_T:
 if( this-
>eq(((declaration*)higherScope_effectif)
->n))
   doPrint("ROLE");
  doPrint(n);
  if (((connector
*)((declaration*)higherScope_effectif )
  -> higherScope)->ProduceDetRole==1)
  doPrintWithNoBreak("DET");
 break;
 case PORT_T:
  if( this-
>eq(((declaration*)higherScope_effectif)
->n))
   doPrint("PORT");
```

Figure 10. Extrait de la méthode Name::fdrprint () corrigée

En ce qui concerne la deuxième erreur, les programmeurs de la version initiale de l'outil Wr2fdr ont décidé d'attribuer l'identificateur ALPHA_conn_name à l'alphabet du processus associé au connecteur. Mais la version initiale de la méthode fdrprint de la classe Connector ne respecte pas cette convention. En effet, elle injecte l'identificateur ALPHA_glue plutôt que ALPHA_conn_name. Notre solution proposée consiste à remplacer l'instruction qui injecte «glue» par une instruction qui injecte le nom du connecteur: (conn_name): conn_name-> fdrprint () (Voir Figure 11).

```
void Connector::fdrprint (void) {
…
if (conn_name) {
 newLine();
  conn_name->fdrprint();
 doPrint("A = ");
 conn_name->fdrprint();
 doPrint(" [[ x <- abstractEvent | x <-
ALPHA_");
 // Use ALPHA_conn_name as the connector
```

```
name
 conn_name->fdrprint();
 doPrint(" ]]");
 newLine();
 doPrint("assert DFA [FD= ");
 conn_name->fdrprint();
 doPrint("A");
 } else {
 doPrint("OOPS: can't produce assertion
for null name.");
 }
 subTab();
 newLine();
}
```

Figure 11. Extrait de la méthode Connector::fdrprint () corrigée

4.3.2. Corrections liées à la propriété 1

Rappelant que l'erreur détectée dans le test de la propriété 1 liée à la cohérence port/computation est l'arrêt brutal de l'outil Wr2fdr avec le message d'erreur «erreur de segmentation».

Dans la section 4.1.2 nous avons dit que la présence d'un pointeur qui pointe nulle part est la cause la plus probable de ce type de problèmes dans les programmes C++. Cette hypothèse est confirmée. En effet, en utilisant la structure macro-scopique extraite du code de l'outil Wr2fdr, nous constatons que la méthode calculationPass déclarée dans la classe AstNode et implémentée dans la classe Component est coupable. En effet, le pointeur higherScope utilisé dans la méthode Component::calculationPass est sans initialisation. Sachant que ce pointeur est utilisé directement dans le calcul des alphabets des concepts (port, computation, rôle et glue) et indirectement dans le calcul des alphabets des concepts Component et Connector. Ainsi, nous avons proposé les corrections suivantes:

- Initialiser le pointeur higherScope de chaque port par le pointeur this qui pointe sur le composant lui-même.
- De même pour le calcul des alphabets du concept computation : le pointeur higherScope de la clause computation est initialisé par le pointeur this.
- Calculer la liste des alphabets du composant lui-même en parcourant la liste des alphabets de chaque port et par l'ajout des alphabets de computation.

Cette correction résout l'erreur de segmentation, mais elle ne peut pas générer les assertions CSP liées à la cohérence de port/computation. Pour générer ces assertions, nous devons implémenter la propriété 1 définie dans (Allen, 1997) par "A port specification must be a projection of the Computation, under the assumption that all other port interfaces are obeyed by the environment". Formellement, cette propriété de projection est définie comme suit:

$$\forall \text{Component (C: Computation; P1, P2, ...,Pn :Port)} \bullet \text{Coherent (C, P)} \rightarrow P \sqsubseteq (C \,||\, \forall i : 1..n \,||\, det(Pi \lceil \alpha oPi)) \lceil \alpha P).$$

Où (αoP) est le sous-ensemble des événements correspondant aux événements observés dans l'alphabet (αP).

Ainsi, pour réaliser cette relation de projection, nous devons implémenter les opérations suivantes:

- L'adaptation des processus CSP des composants Wright pour qu'ils soient acceptables par l'outil FDR. L'implémentation de cette adaptation est inspirée du code relatif à l'adaptation des processus CSP des connecteurs à l'outil FDR.
- L'implémentation des versions déterministes des processus CSP associés aux ports et à la clause computation d'un composant. Ces processus déterministes peuvent être réalisés à l'aide de l'opérateur FDR (det).
- L'implémentation de l'opérateur de restriction (⌈) puisque cet opérateur n'est pas implanté dans l'outil FDR. Pour ce faire, nous avons choisi d'utiliser l'opérateur de hiding (\) implémenté dans l'outil FDR2 (FDR2, 2012). En effet, pour un processus P et un ensemble d'événements A; la restriction (P ⌈A) peut être spécifiée en utilisant l'opérateur de hiding (\) et l'opérateur (-) comme suit: $P \lceil A = P \setminus (\alpha P - A)$. La différence entre les alphabets $(\alpha P - A)$ peut être effectuée à l'aide de l'opérateur FDR (diff). Il suffit donc de calculer la différence entre l'alphabet du composant et celle du port considéré.

Après avoir réussi à implémenter les deux opérateurs (det) et (⌈), nous proposons d'utiliser ces opérateurs dans l'implémentation de la relation de raffinement (⊑). Pour réaliser cette relation de raffinement, nous avons suivi les six étapes suivantes:

- Première étape: Donner une représentation de chaque processus en respectant la syntaxe du model-checker FDR (FDR2, 2012).
- Deuxième étape: Pour passer d'une représentation locale à une autre globale, nous devons renommer tous les événements de ports. Ce renommage peut être effectué de deux façons. La première solution consiste à concaténer le nom de chaque événement avec le nom de son port. La seconde consiste à utiliser l'opérateur de préfixage utilisé par le model-checker FDR ([[←]]). Vu que les événements sont déjà présents, la solution de préfixage est retenue.
- Troisième étape: la production de la version déterministe de tout port après sa restriction à ses événements observés $(det(Pi \lceil \alpha OPi))$. Cette restriction d'un processus (Pi) à ses événements observés (αOPi) est réalisée par l'élimination des événements initiés. Nous devons donc commencer par le calcul des événements initialisés de chaque port (voir Figure 12). Ensuite, les deux méthodes Copy et Hiden et l'ensemble des événements initiés sont utilisés avec profit dans la construction de l'arbre représentatif du processus $(Pi \lceil \alpha OPi)$. Enfin, en appliquant l'opérateur (det) sur le processus $(Pi \lceil \alpha 0Pi)$; nous obtenons le processus déterministe requis dans la construction de cette propriété en FDR.
- Quatrième étape: en utilisant l'opérateur FDR (| |), nous construisons la composition parallèle de la computation C avec les ports Pi: $(C||\forall i:1..n||det(Pi \lceil \alpha OPi))$.

- Cinquième étape: la réalisation de l'équation (C||∀i:1..n||det(Pi⌈αOPi)) ⌈αP). Ceci peut être réalisé par une simple restriction du processus (C||∀i:1..n||det(Pi ⌈αOPi) aux alphabets du port (αP) :
- Sixième étape: préparer le test qui vérifie la relation de raffinement globale:
  P ⊑ (C|| ∀i:1..n||det(Pi⌈αOPi))⌈αP).

```
Set * Set::InitiatedEvent(){
Set * s;
sList *c;
s = new Set();
// Browse through all events
for (c=names; c!=NULL; c=c->next) {
 // Check if this is an initialized
event
 if (c->val != NULL && c->val->gtype ==
EVENT_T && ((event*) c->val)->
event_type == INITIATED_T)
  s->add(c->val);
}
return s;
}
```

Figure 12. Calcul des événements initialisés

4.3.3. Corrections liées à la propriété 8

En se basant sur l'activité de rétro-ingénierie effectuée sur le code de l'outil Wr2fdr et sur le résultat de test effectué dans la section 4.1.3; nous pouvons conclure que la méthode fdrprint déclarée dans la classe AstNode et implémentée dans la classe Configuration ne peut pas générer les assertions CSP liées à la propriété 8. Rappelant que cette dernière permet de vérifier que dans une configuration Wright, tout port attaché à un rôle doit toujours poursuivre son protocole dans une direction que le rôle peut avoir. Cette propriété est définie formellement dans (Allen, 1997) comme suit:

Un port P est compatible à un rôle R si et seulement si :

$R_{+(\alpha P - \alpha R)} \sqsubseteq (P_{+(\alpha R - \alpha P)} \| \det(R))$.

Sachant que, pour un processus P et un autre A; P+αA permet d'augmenter la liste des alphabets du processus P par ceux du processus A.

Cependant, on peut conclure que l'implémentation de la propriété 8 exige : l'implémentation de l'opérateur FDR (det) ; l'implémentation de l'operateur de soustraction (-) ; l'implémentation de l'operateur d'augmentation et la génération des relations de raffinement. En ce qui concerne les deux opérateurs (det) et (-), nous avons réutilisé avec profit les implémentations des opérateurs FDR (det) et (diff). Pour l'opérateur d'augmentation, nous avons réutilisé l'opérateur de composition parallèle (| |) offert par FDR pour augmenter chaque processus par les événements manquants. En effet, l'augmentation P_{+A} peut-être implémenté en utilisant l'opérateur de composition parallèle (| |) comme suit: P | | $STOP_A$.

En outre, nous avons conçu et réalisé deux modules C++ nommés respectivement AttachmentPortName et AttachmentRoleName. Ces deux modules permettent respectivement d'extraire le port et le rôle d'un attachement donné. Ces deux modules sont utilisés de manière efficace dans la construction des relations de raffinement CSP liées à la compatibilité des ports/rôles. En effet, l'implémentation de ce raffinement nécessite les traitements suivants:

- Extraction de la version déterministe du processus associé au rôle. Ce processus déterministe peut être extrait à l'aide de l'opérateur FDR (det).
- Augmentation du processus associé au port par des événements spécifiques au processus associé au rôle (voir Figure 13).
- Augmentation du processus associé au rôle par des événements spécifiques au processus associé au port (voir Figure 13).
- Composition parallèle du processus augmenté associé au port et de la version déterministe du processus associé au rôle.
- Génération de la relation de raffinement entre les deux processus déjà construits.

```
void configuration::fdrprint(void){
…
// Increasing of the Port's process by
the Role's events.
      portInstance->first->fdrprint();
/* The name of the Component instance */
      doPrint("_");
      portInstance->second->fdrprint();
/* The name of the Port name */
      doPrint("PLUS = PORT");
      portInstance->second->fdrprint();
      continueLine();
      doPrint("[| diff( ALPHA_");
      roleInstance->second->fdrprint();
      doPrint(" , ALPHA_");
      portInstance->second->fdrprint();
      doPrint(" ) |] STOP");
      newLine();
      …
// Increasing of the Role's process by
the Port's events.
      roleInstance->first->fdrprint();
/* The name of the Connector instance */
      doPrint("_");
      roleInstance->second->fdrprint();
/* The name of the Role name */
      doPrint("PLUS = ROLE");
      roleInstance->second->fdrprint();
      continueLine();
      doPrint("[| diff( ALPHA_");
      portInstance->second->fdrprint();
      doPrint(" , ALPHA_");
      roleInstance->second->fdrprint();
      doPrint(" )|] STOP");
      newLine();
      …
// Generation of the refinement relation
      doPrint("assert ");
```

```
        roleInstance->first->fdrprint();
        doPrint("_");
        roleInstance->second->fdrprint();
        doPrint("PLUS [FD= ");
        portInstance->first->fdrprint();
        doPrint("_");
        portInstance->second->fdrprint();
        doPrint("PLUSDET ");
        newLine();
        newLine();
        current_attachment = (binaryOp *)
attachments->nextNode();
}
```

Figure 13. Implémentation de la propriété 8

4.4. Maintenance évolutive proposée

L'outil Wr2fdr comprend un analyseur lexico-syntaxique de Wright. Cet analyseur est principalement développé à l'aide des deux générateurs: Lex et Yacc (Hahne et Sato, 1994) (John et al., 1992). Afin d'améliorer cet outil, nous avons conçu et réalisé un analyseur de la sémantique statique de Wright. Pour ce faire, nous avons établi et mis en œuvre les six règles suivantes:

- Règle 1: Un identifiant doit désigner un seul élément architectural (composant, connecteur, port, rôle, configuration ou style).
- Règle 2: Le type d'instance (composant, connecteur) doit être préalablement déclaré.
- Règle 3: Toute instance doit être déclarée avant d'être utilisée dans la clause attachments.
- Règle 4: Une interface de composant (interface de connecteur respectivement) doit être de la forme instance.port (respectivement instance.role).
- Règle 5: Chaque attachement doit être de la forme: instance.port as instance.role.
- Règle 6: Chaque port doit être rattaché à un seul et unique rôle et vice versa.

Afin de mettre en œuvre ces règles relatives à la sémantique statique de Wright nous avons enrichi l'analyseur lexical Wright par une table de symboles. Celle-ci permet de grouper les informations utiles à la vérification des règles proposées précédemment. En utilisant la construction C ++ **union** et la technique de hachage d'adressage ouvert (Wenbin et Gregory, 2003), nous avons réalisé un module C ++ qui matérialise cette table de symboles. Basé sur ce module C ++, nous avons pu mettre en œuvre les règles sémantiques Wright précédemment présentées. Par exemple, pour réaliser la règle 5: nous devons rechercher les deux instances dans la table des symboles. En cas de succès, nous devons vérifier que le type de la première instance est un composant et que le type de la deuxième instance est un connecteur (voir Figure 14). La mise en œuvre complète de ces règles sémantiques est disponible sur le page Web[1].

[1] https://sourceforge.net/projects/wr2fdr-tool/

```
Attachment: Interface TOK_ASKW Interface
{
 li1=$1;
 li2=$3;
 // The searching of the first instance
type p=rechercher(ts,li1->sym);
 p=p->s.attributs.casInstance.type;
 //The searching of the second instance
type q=rechercher(ts,li2->sym);
 q=q->s.attributs.casInstance.type;
 // Verification of the types of the two
instances
 if(p->s.nature!=Composant || q-
>s.nature!=Connecteur ){
  yyerror("***Attachement:
Composant.Port as
  Connecteur.Role***");
  YYABORT;
 }
 li1=dernier($1);
 li1->suivant=li2;
 $$=$1;
};
```

Figure 14. Implémentation de la règle 5

4.5. Validation de l'outil Wr2fdr

La nouvelle version de l'outil Wr2fdr a été soigneusement testée. Pour y parvenir, nous avons adopté une approche de test fonctionnel dirigée par deux critères : couverture des symboles terminaux et couverture des règles de production. En outre, nous avons testé les quatre propriétés 1, 2, 3 et 8 en récupérant plusieurs descriptions architecturales en Wright publiées. La plupart des DT (Données de Test) sont disponible sur notre page Web[2]. A titre d'exemple, la Figure 15 donne la modélisation en Wright d'un système médical -issue de (Kmimech, 2010)- comportant trois types d'acteurs : médecins généralistes (GP), médecins spécialistes (SP) et pharmaciens (PH). En utilisant notre outil Wr2fdr sur la configuration Medical de la Figure 15, nous avons obtenu le fichier CSP présenté dans notre page Web[3]. Ce fichier CSP contient 21 propriétés générées en tant qu'assertions CSP. Ces propriétés sont réparties comme suit:

- 7 propriétés liées à la cohérence port /computation (propriété 1)
- 6 propriétés liées au rôle sans interblocage (Propriété 2)
- 3 propriétés liées au connecteur sans interblocage (propriété 3)
- 6 propriétés liées à la compatibilité port / rôle (propriété 8)

La Figure 16 montre que ces propriétés CSP sont vérifiées à l'aide du model-checker FDR2.

[2] https://sourceforge.net/projects/wr2fdr-tool/

[3] https://sourceforge.net/projects/Medical

```
Configuration Medical
Component GP   //Description of the
component GP
Port authenticate = startSession ->
_endSession-> authenticate
Port giveData = _writeData -> giveData
Port getPrescription =
acceptPrescription -> getPrescription
Computation = authenticate.startSession
-> _giveData .writeData ->
getPrescription .acceptPrescription ->
_authenticate.endSession -> Computation

Component SP //Description of the
component SP
Port getData = acceptData -> getData
Port giveDiagnostic = _writeDiagnostic ->
giveDiagnostic
Computation = getData.acceptData ->
_giveDiagnostic.writeDiagnostic ->
Computation

Component PH //Description of the
component PH
Port getDiagnostic = acceptDiagnostic ->
getDiagnostic
Port givePrescription =
_writePrescription -> givePrescription
Computation = getDiagnostic.
acceptDiagnostic -> _givePrescription.
writePrescription -> Computation

Connector C1
Role receivedData = _writeData ->
receivedData
Role sendData =  acceptData -> sendData
Glue = receivedData.writeData -
>_sendData.acceptData -> Glue

Connector C2
Role receivedDiagnostic =
_writeDiagnostic -> receivedDiagnostic
Role sendDiagnostic =  acceptDiagnostic
-> sendDiagnostic
Glue =
receivedDiagnostic.writeDiagnostic -
>_sendDiagnostic.acceptDiagnostic ->
Glue

Connector C3
Role receivedPrescription =
_writePrescription ->
receivedPrescription
Role sendPrescription =
acceptPrescription -> sendPrescription
Glue =
receivedPrescription.writePrescription -
>_sendPrescription.acceptPrescription ->
Glue

Instances //Instances of components and
connectors creation
Generalist: GP
Pharmacist: PH
Specialist: SP
GPSP: C1
SPPH: C2
PHGP: C3
Attachments //System compositions
Generalist. giveData As GPSP.
receivedData
Specialist. getData As GPSP. sendData

Specialist. giveDiagnostic As SPPH.
receivedDiagnostic
Pharmacist.getDiagnostic As  SPPH.
sendDiagnostic

Pharmacist. givePrescription  As  PHGP.
receivedPrescription
Generalist. getPrescription As PHGP.
sendPrescription
End configuration
```

Figure 15. Le système Médical en Wright

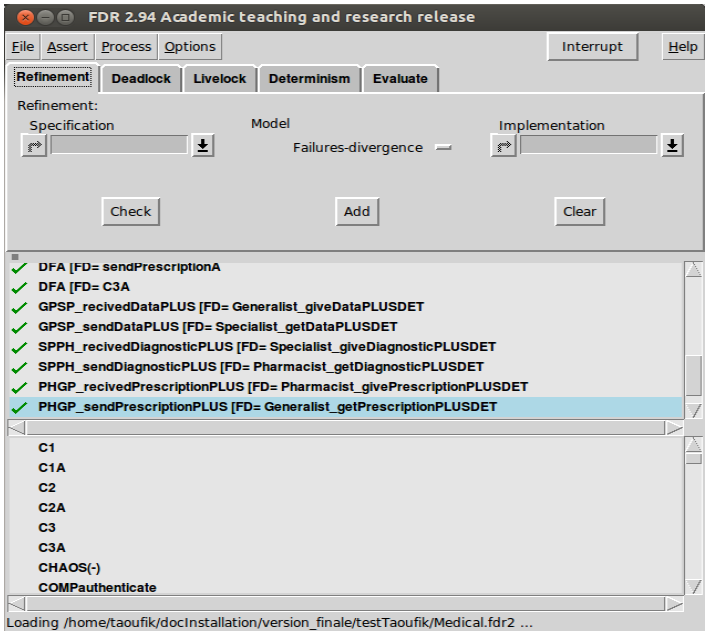


Figure 16. Vérification des propriétés en utilisant FDR2

De même, nous avons testé avec succès notre analyseur sémantique par l'injection des erreurs sémantiques voulues dans les spécifications Wright déjà testées et prouvées correctes. Par exemple, la Figure 17 montre que notre analyseur sémantique détecte une erreur syntaxique dans la configuration ErrorMedical. Sachant que la configuration ErrorMedical est issue de celle de la Figure 15 avec une injection d'erreur dans la clause attachements.

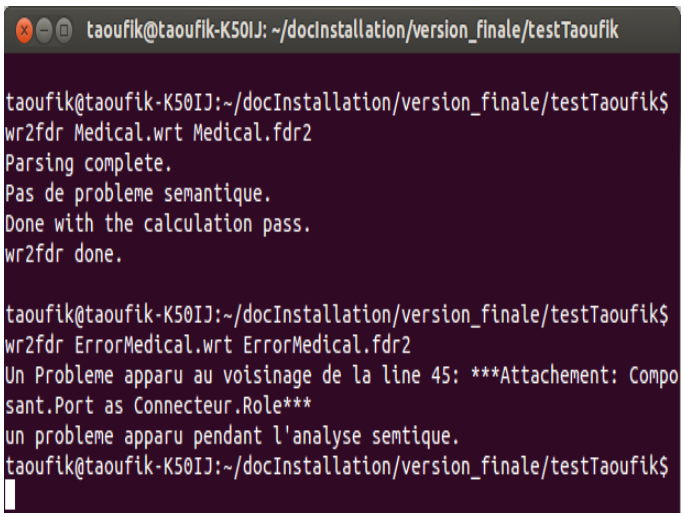


Figure 17. Détection d'erreurs par notre analyseur sémantique

5. De Wright vers Ada

L'ADL Wright permet de décrire les aspects structuraux et comportementaux d'une architecture logicielle abstraite. Les aspects comportementaux sont décrits en CSP et vérifiés en utilisant notre outil Wr2fdr et le model checker FDR. Cependant, cet ADL formel n'offre aucun moyen permettant de concrétiser de telles architectures abstraites. L'objectif de cette section est d'ouvrir l'ADL Wright sur le langage concurrent Ada. Pour ce fait, nous allons proposer un ensemble de règles permettant la traduction systématique de Wright vers Ada. Ces règles sont automatisées en utilisant une approche de type IDM.

5.1. Règles de traduction proposées

En se basant sur la sémantique formelle d'une configuration Wright et sur nos travaux antérieurs (Bhiri 2008) et (Bhiri 2012), nous proposons les correspondances intuitives fournies ci-après permettant de traduire de Wright vers Ada.

5.1.1. *Traduction d'une configuration* :

Une configuration Wright est traduite en Ada par un programme concurrent dans lequel :

- chaque instance de type composant est traduite par une tâche Ada ;
- chaque instance de type connecteur est traduite également par une tâche Ada ;
- les tâches de même type ne communiquent pas entre elles.

La table 2 illustre le principe de la traduction d'une configuration Wright en Ada. Pour des raisons de traçabilité, nous gardons les mêmes identificateurs utilisés dans la spécification Wright. En plus, nous transportons la nature de chaque instance soit Component, soit Connector. Ceci favorise l'identification des parties à modifier dans la configuration Wright dans le cas où sa simulation en Ada produit des défaillances.

Spécification Wright	**Code Ada**
Configuration Client_Server **Component** Client **Component** Server **Connector** RPC **Instances** c : Client s : Server cs : RPC **Attachments** … **End Configuration**	**procedure** Client_Server **is** **task** Component_c **is** **end** Component_c **;** **task** Component_s **is** **end** Component_s**;** **task** Connector_cs **is** **end** Connector_cs**;** **task body** Component_c **is** **end** Component_c**;** **task body** Component_s **is** **end** Component_s**;** **task body** Connector_cs **is** **end** Connector_cs**;** **begin** **null;** **end** Client_Server**;**

Table 2. Traduction d'une configuration Wright

La traduction proposée possède un avantage majeur : elle permet de conserver la sémantique d'une configuration Wright. En effet, celle-ci est définie formellement en CSP comme la composition parallèle des processus modélisant les composants et les connecteurs formant cette configuration. Les types tâches en Ada (task type) permettant de factoriser plusieurs tâches ayant le même comportement (task body) ne sont pas utilisées dans notre approche de traduction de Wright vers Ada. Ceci est expliqué par le fait que dans une configuration Wright deux instances d'un même composant -respectivement d'un même connecteur– n'ont pas le même comportement (task body) bien qu'elles possèdent la même interface (task). Par exemple, deux instances d'un même composant reliées au même connecteur jouent deux rôles différents et entraînent deux demandes de rendez-vous sur deux entrées (entry) différentes offertes pour le connecteur.

5.1.2. *Traduction d'un événement observé:*

Un événement observé de la forme e est traduit par une entrée (entry) et par une acceptation de rendez-vous (instruction accept). La table 3 illustre le principe de la traduction d'un événement observé en Ada.

Spécification Wright	**Code Ada**
Configuration Client_Server **Component** Client **Port** sendRequest= _request -> reply -> sendRequest |~| § **Computation**=_sendRequest.request -> sendRequest.reply -> computation |~| § **Instances** c : Client …	**task** Component_c **is** **entry** sendRequest _reply ; **end** Component_c ; **task body** Component_c **is** ... **accept** sendRequest_reply ; ... **end** Component_c ; …

Table 3. Traduction d'un événement observé

5.1.3. *Traduction d'un événement initialisé:*

Un événement initialisé de la forme (_e) est traduit par une demande de rendez-vous sur l'entrée (e) exportée (ou offerte) par une tâche de type différent (seules les tâches de types différents communiquent en distinguant des

catégories : component et connector) à identifier. Pour y parvenir, il faut analyser la partie attachments de la configuration. La Table 4 illustre le principe de la traduction d'un événement initialisé.

Spécification Wright	Code Ada
Component Client **Port** appelant = _request -> result -> appelant |~|§ …. **Connector** cs **Role** client = _request -> result-> client |~|§ **Role** serveur = request -> _result -> serveur [ ] § **Instances** c : Client cs: RPC **Attachments** Client. appelant as cls.client ...	**task** Component_c **is** **entry** result; **end** Component_c; **task** Connector_cls **is** **entry** result; **entry** request; **end** Connector_cs; **task body** Component_c **is** **begin** Connector_cs.request; **end** Component_c;

Table 4. Traduction d'un événement initialisé

5.1.4. *Traduction de l'opérateur de choix interne:* La notation P Π (ou |~|) Q avec P ≠ Q, dénote en CSP un processus qui se comporte soit comme P soit comme Q, la sélection étant réalisée de façon arbitraire, hors du contrôle ou de la connaissance de l'environnement extérieur. Cet opérateur de choix interne peut être implémenté en Ada en utilisant l'instruction conditionnelle if ou en utilisant l'opérateur à choix multiple case. Cependant l'opérateur de choix externe ou de choix déterministe est traduit en Ada par l'instruction select. La Table 5 illustre la traduction de ces deux opérateurs de choix.

Spécification Wright	Code Ada

P1 |~| P2	if condition_interne then
	Translation P1
	else
	Translation P2
	end if ;
	--condition_interne» est une fonction Ada rendant un booléen

P1 |~| P2 |~| P3 |~| ...	case condition_interne1 is when 1 => Translation P1 when 2 => Translation P2 when 3 => Translation P3 ... end case ; --condition_interne1» est une fonction Ada non déterministe rendant un entier
a ->P1 [] b->P2 [] V->STOP [] c->Q	Select traduction de a puis de P1 or traduction de b puis de P2 or traduction de c or terminate ; end select

Table 5. Traduction d'opérateurs de choix

5.1.5. *Traduction d'une configuration hiérarchique:*

Dans notre traduction de Wright vers Ada, le composant composite et le connecteur composite ont été nouvellement intégrés. Dans Wright, la partie computation du composant (respectivement la partie glue du connecteur) peut être elle-même décrite par une architecture. Le composant (respectivement le connecteur) sert comme une abstraction de liaison pour le sous-système architectural imbriqué (Allen 1997). En effet, le comportement global d'un composant composite (respectivement d'un connecteur composite) décrit par la clause computation (respectivement Glue) est matérialisé par une configuration Wright. Ainsi sur le plan macroscopique, un composant composite peut être traduit en Ada par une tâche mère ayant des tâches filles correspondant aux instances de composants et de connecteurs de la configuration imbriquée (Voir Table 6). Ceci est également vrai pour les connecteurs composites.

Spécification Wright	Code Ada
Configuration HierServer... Component Server... Computation= Configuration ServerDetails Component ConnectionManager... Instances cm : ConnectionManager ...End Bindings Instances server1 : Server	procedure Hierserver is... task Component_server1 is... end Component_server1 ; task body Component_server1 is task Component_cm is ... end Component_cm; task body Component_cm is ...end Component_cm; ... end Component_server1 ;

Table 6. Traduction d'une configuration incluse dans un composant.

Pour un composant composite, la description architecturale imbriquée a un ensemble de liaisons (la clause Bindings) associées qui définissent la manière dont les ports de la configuration interne sont associés aux ports du composant englobant. De même, pour les connecteurs composites l'ensemble de liaisons définissent la manière dont les rôles de la configuration interne sont associés aux rôles du connecteur englobant.

Dans notre traduction vers Ada, le composant composite joue le rôle d'intermédiaire avec le composant interne attaché à son port dans la clause binding (Voir Table 7).

Spécification Wright	Code Ada
Component Server Port receiveRequest= request -> _reply -> receiveRequest [] V ->STOP Computation= Configuration ServerDetails Component ConnectionManager Port externalSocket= request -> _reply -> externalSocket [] V	task Component_server1 is entry receiveRequest_request ; end Component_server1 ; begin loop select accept receiveRequest_request ; Component_cm. externalSocket_request ; or terminate ;

-> STOP ... End Configuration Bindings cm.externalSocket : receiveRequest End Bindings ... End Configuration	end select ; end loop ; end Component_server1 ;

Table 7. Traduction d'un composant composite

Tous les composants et connecteurs de l'architecture interne seront traduits comme présenté dans les sections précédentes à l'exception du composant auquel le composant composite délègue le traitement. Ce dernier sera traduit comme présenté dans la table 8.

Spécification Wright	**Code Ada**
Component Server Configuration HierServer ... Computation= Configuration ServerDetails Component ConnectionManager Computation = ... -> _dbQueryIntf.query -> dbQueryIntf.answer -> _externalSocket.reply -> ... ... Attachments ... cm.dbQueryIntf As sqlq.caller End Configuration Bindings cm.externalSocket : receiveRequest End Bindings ... Attachments ... server1.receiveRequest As rpc.callee End Configuration	task body Component_cm is ... Connector_sqlq .caller_query ; accept dbQueryIntf_answer ; Connector_rpc .callee_reply ; ... end Component_cm ;

Table 8. Traduction du composant interne lié au composant composite

5.2. Realisation de l'outil Wr2ada

Nous avons retenu une approche IDM afin de réaliser notre traducteur de Wright vers Ada. Les deux raisons principales qui militent en faveur de cette approche sont productivité et réutilisabilité. Ainsi, nous considérons la traduction de Wright vers Ada comme une opération de transformation de modèles exogène et verticale : le modèle source et le modèle cible sont issus de deux méta-modèles différents et appartiennent à deux niveaux d'abstraction différents. De plus, compte tenu de notre expérience de maintenance corrective et évolutive de l'outil Wr2fdr écrit en C++ permettant de traduire de Wright vers CSP pour la vérification par model-checking, nous avons lancé une action ayant pour objectif de réaliser notre traducteur de Wright vers Ada en réutilisant les possibilités offertes par Wr2fdr liées notamment à l'analyse lexico-syntaxique de Wright. A terme, ceci permettrait de comparer d'une façon rigoureuse et pragmatique les deux approches traditionnelles et IDM pour cette opération de transformation de Wright vers Ada.

Les principaux outils de la plate-forme Eclipse utilisés dans ce travail sont : Xtext (Haase et al., 2007), Check (Haase et al., 2007), ATL (Jouault et al., 2008) et Xpand (Klatt, 2007). L'outil Xtext permet de créer un environnement de développement intégré spécifique au langage traité à partir d'une grammaire décrite en Xtext du langage considéré. L'outil Check permet de vérifier des contraintes exprimées dans le langage Check liées à la sémantique statique du langage traité. Sachant que le langage Check est basé sur OCL. Le langage ATL est un langage dédié à la transformation de modèles. Il fournit aux développeurs des moyens permettant de produire un certain nombre de modèles cibles à partir de modèles sources. Enfin, le langage Xpand permet d'écrire des templates afin d'extraire du code à partir d'un modèle XMI.

En utilisant les outils IDM ATL, Xtext, Xpand et Check décrits précédemment, nous avons conçu, réalisé et testé l'outil Wr2ada acceptant en entrée une spécification Wright sous forme de texte et produisant en sortie le code Ada équivalent sous forme textuelle. L'outil Wr2ada est basé sur la réalisation d'un méta-model pour l'ADL Wright, un méta-model pour le langage Ada et trois types d'opérations de transformation de modèles : texte Wright vers modèle Wright (T2M), modèle Wright vers modèle Ada (M2M) et modèle Ada vers texte Ada (M2T).

5.2.1. *Méta-modèle partiel pour l'ADL Wright :*

Dans cette section, nous proposons un méta-modèle partiel de Wright représentant la plupart des concepts issus de ce langage : composant (atomique ou composite), connecteur (atomique ou composite), configuration et processus CSP. Notre méta-modèle Wright comporte 24 méta-classes (Voir Figure 18) et 23 propriétés décrivant des contraintes sémantiques liées à la bonne utilisation de Wright. Ces propriétés sont formellement décrites dans le langage OCL dans notre SourceForget repository (Bhiri et al., 2018).

Le nouveau méta-modèle partiel Wright comporte deux fragments représentant respectivement la partie structurelle et comportementale de Wright. La partie structurelle de notre méta-modèle partiel de Wright est centrée autour de la méta-classe Configuration. Tandis que la partie comportementale est centrée autour de la méta-classe BehaviorDescription. La connexion entre les deux fragments du méta-modèle Wright est assurée par les méta-associations computation (entre Component et BehaviorDescription), glue (entre Connector et BehaviorDescription) et behavior (entre (Port et ProcessExpression) et (Role et ProcessExpression)).

La méta-classe Configuration occupe une position centrale. Elle englobe des composants, des instances de composants, des connecteurs, des instances de connecteurs et des attachements. Ceci est traduit par une métacomposition entre Configuration et respectivement Component, ComponentInstance, Connector, ConnectorInstance et Attachment. Les deux méta-classes CompositeProcess et ProcessExpression héritent de la méta-classe BehaviorDescription. La méta-classe Composite-Process représente le composant composite ou le connecteur

composite. La partie comportementale du composant composite (computation) est représentée par la configuration interne d'où la méta-agrégation entre CompositeProcess et Configuration. Un composant composite lui est associé un ensemble de liaisons (bindings). Une liaison fait interagir deux interfaces de même type, ports pour les composants composites et rôles pour les connecteurs composites. Ceci se présente par les deux méta-classes PortBinding et RoleBinding.

La hiérarchie ayant comme méta-classe fondatrice ProcessExpression modélise le concept de processus en CSP. Les méta-classes descendantes Prefix, ExternalChoice, InternalChoice et ProcessName représentent respectivement les opérateurs préfixage, choix externe (ou déterministe), choix interne (ou non déterministe) et le nommage d'un processus (favorisant la récursion) fournis par CSP. L'autre hiérarchie ayant comme méta-classe fondatrice EventExpression représente le concept d'événement en CSP Wright.

Les méta-classes descendantes EventSignalled, EventObserved, Internal-Traitment et SuccesEvent représentent respectivement : un événement initialisé, un événement observé, un traitement interne et un événement succès fournis par CSP de Wright. Les liens entre ces deux hiérarchies sont traduits par la méta-agrégation entre Prefix et EventExpression.

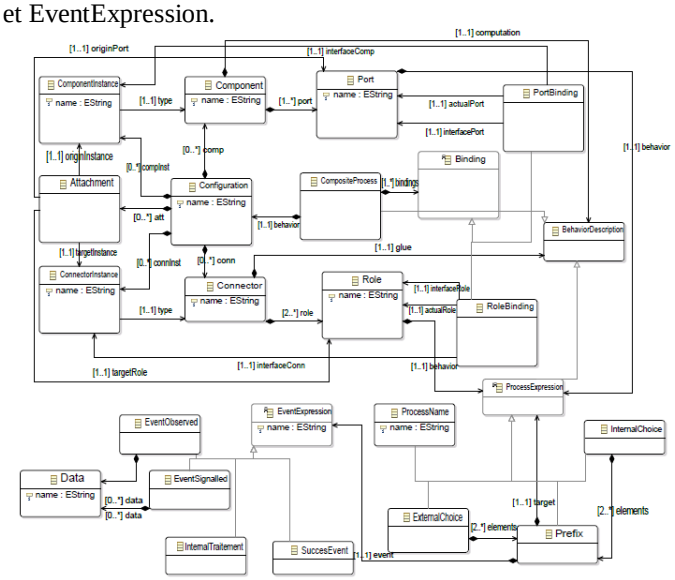


Figure 19. Métamodèle partiel de Wright

Notre méta-modèle partiel de Wright est accompagné des 23 contraintes OCL. Ces dernies décrient les propriétés sémantiques liées à la bonne utilisation de l'ADL Wright. Ces propriétés sont exprimées à la fois d'une manière informelle et de manière formelle en OCL dans notre SourceForget repository (Bhiri et al., 2018). A titre d'exemple, la propriété qui stipule qu'un choix externe doit être basé uniquement sur des événements observés et succès peut être formalisée en OCL par :

```
context ExternalChoice inv event_observed_in_EC: self.elements -> forAll( e : Prefix |e.event.oclIsTypeOf(EventObserved) or e.event.oclIsTypeOf(SuccesEvent) )
```

5.2.2. *Méta-modèle partiel d'Ada:*

Nous avons établi un métamodèle partiel d'Ada issu de la description BNF de ce langage (BNF-Ada, 2009). Notre métamodèle comporte la structure des constructions utilisées dans la transformation de Wright vers Ada. Ces constructions sont : sous-programmes non paramétrés, tâches ayant des entrées non paramétrées, structures de contrôle, demande d'un rendez-vous, acceptation d'un rendez-vous et instruction de non-déterminisme.

Notre métamodèle partiel d'Ada comporte 26 métaclasses (voir Figure 19) et 11 contraintes sémantiques liées à la bonne utilisation d'Ada (Kmimech, 2010). Ces contraintes sont formalisées en OCL. Bien entendu, le traducteur Wr2ada est censé produire du code acceptable par tous les compilateurs Ada certifiés. Donc, il est inutile de vérifier ces contraintes sur le modèle cible produit par Wr2ada. Mais pour des raisons de capitalisation, notre métamodèle est doté des contraintes sémantiques. Ceci favorise sa réutilisation dans des opérations de rétroconception du code vers le modèle.

La structure des instructions composites est définie d'une façon récursive. Par exemple, la métaclasse IfElse descend de Statement et regroupe plusieurs instructions dans les deux parties then et else. Ceci est traduit par les deux méta-agrégations orientées s1 et s2 entre IfElse et Statement.

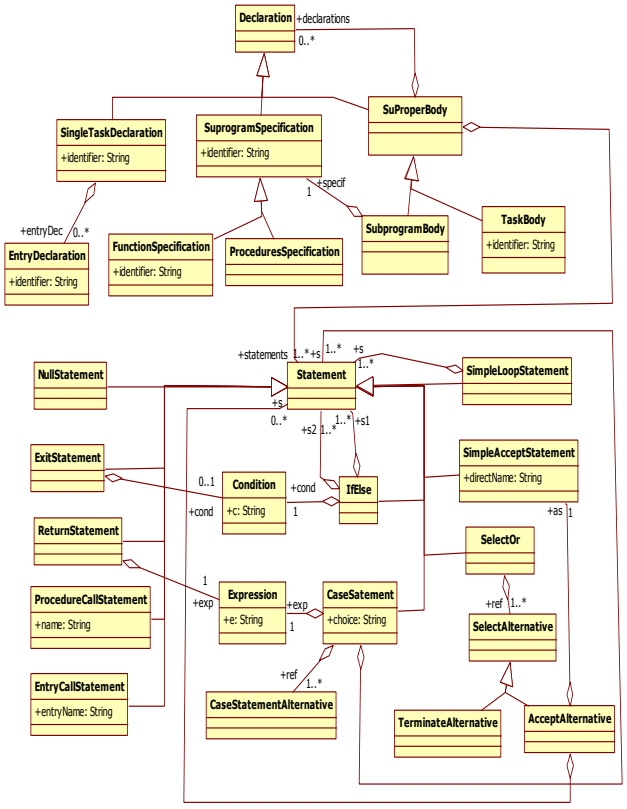


Figure 19. Métamodèle partiel d'Ada

Les propriétés décrivant des contraintes sémantiques liées à la bonne utilisation d'Ada sont exprimées à la fois d'une manière informelle et de manière formelle en OCL (Kmimech, 2010).

A titre d'exemple, la propriété qui stipule qu'une fonction Ada contient au moins une instruction return peut être formalisée en OCL comme suit :

```
context SubprogramBody
inv: specif.oclIsTypeOf(FunctionSpecification) implies statements -> collect(s:Statement|s.oclIsTypeOf(ReturnStatement)) -> size()>=1
```

5.2.3. *Constituants de l'outil Wr2ada :*

Notre approche de transformation de Wright vers Ada est basée sur trois transformations consécutives : (1) texte Wright vers modèle Wright (T2M) ; (2) modèle Wright vers modèle Ada (M2M) et (3) modèle Ada vers texte Ada (M2T).

La première transformation (T2M) est elle-même composée trois étapes consécutives : (1) La première est basée sur la création d'un programme Xtext associé à la grammaire de l'ADL Wright. Ce programme Xtext permet la création d'un méta-modèle Ecore pour l'ADL Wright. La première entité de ce programme Xtext (voir Listing 1) est la configuration. Ce dernier constitue une entité d'accueil du texte Wright accepté par notre outil. Une configuration commence obligatoirement par le mot-clef Configuration suivie d'une chaine de caractères (nom), puis l'ensemble de types qui peuvent être des composants ou des connecteurs, l'ensemble d'instances qui peuvent être des instances de composants ou des instances de connecteurs, et enfin, l'ensemble des attachements. Le symbole " += " signifie que la variable contient un ensemble du type correspondant. Le symbole " * " signifie la cardinalité zéro ou plusieurs. Un composant (respectivement un connecteur) peut être atomique ou composite. Un composant (respectivement un connecteur) est atomique lorsque sa partie computation (respectivement glue) est décrite par un processus CSP de type ProcessExpression. Cependant, un composant est composite lorsque sa partie computation (respectivement glue) est décrite par une configuration de type CompositeProcess. Le symbole de terminaison avec succès SKIP (ou encore §) est désigné dans notre programme Xtext par le processus V -> STOP. Où V est l'événement succès et STOP est un processus prédéfini qui ne fait rien. Sachant que l'événement V est toujours suivi de STOP. (2) Dans la deuxième étape, nous avons réécrit dans le langage Check les 23 propriétés décrivant la bonne utilisation de l'ADL Wright. Le Listing 2 présente quelques contraintes écrites en Check liées au concept hiérarchique. La redéfinition de ces propriétés permet la validation des modèles obtenus par notre programme Xtext. (3) Finalement, nous avons aussi établi un programme ATL appelé GrammaireWright2Wright 4 . Ce dernier permet la transformation des modèles sources générés par notre programme Xtext vers des modèles cibles conformes au méta-modèle Wright.

```
Configuration : "Configuration" name=ID
(TypeList+=Type)*
"Instances"
(InstanceList+=Instance)*
"Attachments"
(att+=Attachment )*
"End Configuration" ;
Instance : ComponentInstance | ConnectorInstance ;
Type : Component| Connector ;
Component : "Component" name=ID (port+=Port )+
"Computation" '=' computation=BehaviorDescription ;
Port : "Port" name=ID '='
behavior=ProcessExpression ;
Connector : "Connector" name=ID (role+=Role )+
"Glue" '=' glue=BehaviorDescription ;
Role : "Role" name=ID '='
behavior=ProcessExpression ;
BehaviorDescription :
ProcessExpression|CompositeProcess ;
CompositeProcess : behavior=Configuration
"Bindings" (bindings+=Bindings )* "End Bindings" ;
Bindings : 'BP' PortBinding|'BR' RoleBinding ;
RoleBinding : interfaceConn=[ConnectorInstance] '-'
interfaceRole=[Role]
' :' actualRole=[Role] ;
PortBinding : interfaceComp=[ComponentInstance] '-'
interfacePort=[Port] ' :' actualPort=[Port] ;
ComponentInstance : name=ID ' :' "Component"
type=[Component] ;
ConnectorInstance : name=ID ' :' "Connector"
type=[Connector] ;
Attachment : originInstance=[ComponentInstance] '-'
originPort=[Port]"
As" targetInstance=[ConnectorInstance]'-'
targetRole=[Role] ;
EventExpression : EventSignalled | EventObserved |
InternalTreatment |
SuccesEvent ;
EventSignalled : '_' name=ID (data+=Data)* ;
EventObserved : name=ID (data+=Data)* ;
InternalTreatment : '-' name=ID ; SuccesEvent :
name='V' ;
Data : (' ?' | ' !') name=ID ;
Prefix : event=EventExpression '->'
target=TargetPrefix ;
TargetPrefix : Parentheses | Prefix | ProcessName ;
ProcessName : name=ID ; Parentheses : '('
p=ProcessExpression ')' ;
ProcessExpression : right=Prefix (('[]'
ECLeft+=Prefix)+ |
('|~|'ICLeft+=Prefix)+) ? ;
terminal ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')* ;
```

Listing 1. Grammaire partielle de Wright hiérarchique en Xtext

```
context PortBinding ERROR " A binding is invalid if the port is not
attached to the instance in question of the component type" :
interfaceComp.type.port.contains(interfacePort ) ;
context Component ERROR " Each binding declared within a
component must use a port declared within the same component " :
```

1. The ATL source code is available at our repository (Bhiri et al., 2018).

```
        (computation.metaType==wright1 : :Composit
eProcess) implies
        (((wright1 : :CompositeProcess)computation).b
indings.typeSelect(
        wright1 : :PortBinding).forAll( b |
port.contains( b.actualPort))) ;
        context Component ERROR " A composite
component must only
        contain binding ports " :
        (computation.metaType==wright1 : :Composit
eProcess) implies
        (((wright1 : :CompositeProcess)computation).b
indings.forAll( b |
        b.metaType==wright1 : :PortBinding)) ;
```

Listing 2. Exemples de contraintes Check du méta-modèle Wright

La deuxième transformation (M2M) est basée sur un programme ATL appelé Wr2ada[5]. Ce dernier permet la transformation d'un modèle source conforme au méta-modèle Wright vers un modèle cible conforme au méta-modèle Ada. Pour exprimer la logique de transformation, le langage ATL offre deux constructions élémentaires : rules et helpers. Les règles ATL peuvent être soit déclaratives (matched rules, lazy rules et unique lazy rules) soit impératives (called rules, action block). Cependant, les helpers ATL permettent aux développeurs de définir leurs propres méthodes dans les différentes sections du programme ATL. Ces méthodes permettent de factoriser le code ATL que l'on peut appeler à différents endroits du programme. Dans ce contexte, les helpers non paramétrés sont appelés des attributs et les helpers paramétrés sont appelés opérationnels helpers. Notre programme ATL comporte : 1 règle standard, 19 règles lazy, 2 attributs helpers, 14 opérationnels helpers et 2 polymorphiques opérationnels helpers. La règle standard est appelée Configuration2subprogram (voir Listing 3). Cette règle est déclenchée automatiquement par l'exécutif d'ATL pour générer un modèle Ada associé à la configuration source Wright. Cette règle comporte des appels aux helpers et aux règles lazy. Ces derniers permettent l'implémentation pas à pas des règles de traduction de Wright vers Ada présentées dans la Section 5.1. Par exemple, la règle lazy ComponentInstance2single_task_declaration (Voir Listing 4) appelée par la règle standard permet de calculer la partie interface d'une tâche Ada correspondant à une instance d'un composant Wright. Cependant, la règle ComponentInstance2single_task_body (Voir Listing 4) permet de générer le corps de cette tache Ada.

```
--- Header of the Wr2ada program
module WrightToAda ;
create exampleAda : Ada from
exampleWright : Wright ;
…
--- Matched rule of the Wr2ada program
rule Configuration2subprogram{ from c :
Wright !Configuration to sb :
Ada !subprogram_body (
specif <- sp , statements <- st ,
declarations <- OrderedSet{cib,cin} ->
union(c.getInternalTraitement() ->
collect(e|thisModule.InternalTraitement2
subprogram(e))) ->
union(c.compInst ->
collect(e|thisModule.ComponentInstance2s
ingle_task_declaration(e)))
-> union(c.connInst ->
collect(e|thisModule.ConnectorInstance2s
ingle_task_declaration(e)))
-> union(c.compInst ->
collect(e|thisModule.ComponentInstance2t
ask_body(e))) ->
union(c.connInst ->
collect(e|thisModule.ConnectorInstance2t
ask_body(e)))),
sp :
Ada !procedure_specification( designator
<- c.name),
st : Ada !null_statement,
cib : Ada !subprogram_body(...),
cin : Ada !subprogram_body(...) }
…
```

Listing 3. – Code ATL de la règle standard Configuration2subprogram

```
lazy rule
ComponentInstance2single_task_declaratio
n{
from ci:Wright!ComponentInstance
to std:Ada!single_task_declaration(
identifier <- 'Component_'+ ci.name,
      entryDec <-
ci.type.computation.getEventObserved()
       -> collect(e|

thisModule.EventObserved2entry_declarati
on(e)))
}
lazy rule ComponentInstance2task_body{
from ci:Wright!ComponentInstance
to   tb:Ada!task_body(
             identifier <- 'Component_'+
ci.name,
             declarations <- if
ci.type.computation.oclIsTypeOf
(Wright !CompositeProcess) then
OrderedSet{} ->
union(ci.type.computation.behavior.getIn
ternalTraitement() -> collect(e|
thisModule.InternalTraitement2subprogram
(e))) ->
union(ci.type.computation.behavior.compI
nst ->
collect(e|thisModule.ComponentInstance2s
ingle_task_declaration(e))) ->
union(ci.type.computation.behavior.connI
nst ->
collect(e|thisModule.ConnectorInstance2s
ingle_task_declaration(e))) ->
```

2. The ATL source code is available at our repository (Bhiri et al., 2018).

```
union(ci.type.computation.behavior.compI
nst ->
collect(e|thisModule.ComponentInstance2t
ask_body(e))) ->
union(ci.type.computation.behavior.connI
nst ->
collect(e|thisModule.ConnectorInstance2t
ask_body(e))) else OrderedSet{}endif, }
```

Listing 4. Code ATL de quelques Lazu rules

La dernière transformation (M2T) est assurée par l'outil Xpand. En effet, nous avons implémenté un programme Xpand qui permet la génération d'un code Ada depuis un modèle Ada conforme au méta-modèle présenté dans la Section 5.2.2. Notre programme Xpand est composé d'une spécification, d'un corps composé d'une partie déclarative et d'une partie exécutive (statement). Ceci peut être traduit par le Listing 5. L'instruction EXPAND appelle un bloc DEFINE et insère sa production "output" à son emplacement. Il s'agit d'un concept similaire à un appel de sous-routine (méthode). Le paramètre d'un statement permet de passer le nom de la tâche au niveau inférieur pour l'affichage lors de la demande d'un rendez-vous.

```
«DEFINE main FOR subprogram_body»
«FILE "adaCode.adb"»
with Ada.Text_IO ;  use Ada.Text_IO  ;
with Ada.Numerics.Discrete_Random  ;
«EXPAND  specification  FOR  this.specif-»
«EXPAND  declaration  FOREACH
this.declarations-»
begin
«EXPAND  statement  (this.specif.designator)
FOREACH this.statements-»
end  «this.specif.designator» ;
«ENDFILE»
«ENDDEFINE»
```

Listing 5. : Xpand main template

Les templates specification, declaration et statement seront redéfinis selon le contexte de leurs appels. Par exemple le Listing 6 présente :

- La spécification d'une fonction Ada;
- La spécification d'une procédure Ada;
- La déclaration de la partie déclarative d'une tache Ada;
- La déclaration de la partie corps d'une tache Ada;
- Le statement correspondant à l'instruction if_else Ada;

```
#{Specification of the Ada procedure}
«DEFINE specification FOR
procedure_specification»
procedure «this.designator» is
«ENDDEFINE»
#{Specification of the Ada function}
«DEFINE specification FOR
function_specification»
function «this.designator» return
«this.returnType» is
«ENDDEFINE»
#{Declaration of the Ada task interface}
«DEFINE declaration FOR
single_task_declaration»
task  «this.identifier»
«IF this.entryDec.isEmpty» ;
«ELSE»  is  «EXPAND Entry FOREACH
this.entryDec-»
end «this.identifier» ;
«ENDIF» «ENDDEFINE»
#{Declaration of the Ada task body }
«DEFINE  declaration  FOR  task_body»
task  body  «this.identifier»  is  «EXPAND
declaration  FOREACH
 this.declarations-»
begin  «EXPAND  statement(this.identifier)
FOREACH
this.statements-»
end «this.identifier»  ;
«ENDDEFINE»
#{Statement of the Ada if_else instruction}
«DEFINE statement FOR if_else»
if « this.cond.c» then
«EXPAND statement FOREACH this.s1»
else
«EXPAND statement FOREACH this.s2»
end if;
«ENDDEFINE»
```

Listing 6 : Part of Xpand program

5.3. *Validation de l'outil Wr2ada*

Dans cette partie, nous présentons une validation de notre outil Wr2ada sur une étude de cas illustrative (Voir Figure 20). L'étude de cas choisie correspond à une architecture hiérarchique Wright nommée "HierServer" (Garlan, 2000). Ce dernier est composé de deux composants : Client et Server. Le composant Client envoie une requête au serveur. Cependant le composant Server la traite afin de répondre au Client. Le traitement de la réponse par le composant Server est assuré par un assemblage des trois composants composites suivant:

- Un composant ConnectionManager qui traite la requête du client et fournit une interface du serveur d'application. Il délègue le traitement de la politique de sécurité à un autre composant.
- Un composant SecurityManager qui permet la vérification de la requête selon la politique de sécurité. Il permet de déterminer si une opération est potentiellement dangereuse ou sensible. Il autorise ou interdit une opération. Il permet aussi de vérifier si une demande d'accès devrait être accordée ou refusée.
- Un composant DataBase qui représente la base de données de l'application.
- Les connecteurs RPC, clearanceRequest, SecurityQuery et SQLQuery permettent de gérer les interactions entre les différents composants.

Le fichier Wright contenant la description architecturale Wright de l'application HierServer est disponible dans notre page Web (Bhiri, et al., 2018). De même, cette page Web supporte le programme concurrent Ada généré par notre outil Wr2ada suite à l'exécution de la description Wright HierServer.

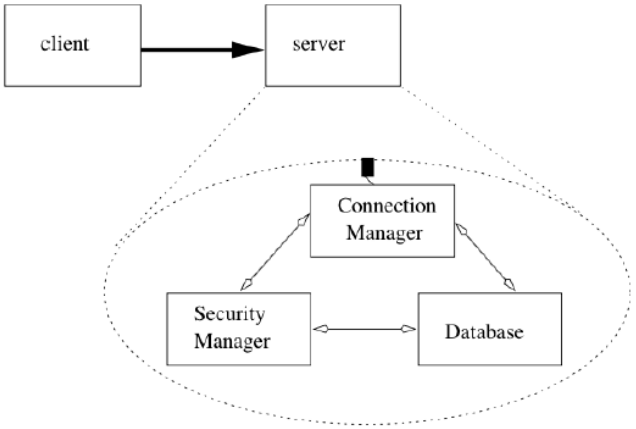


Figure 20. Client-Serveur Hiérarchique

Une simple analyse statique du programme concurrent Ada généré par notre outil Wr2ada, nous permet de constater que ce programme Ada est composé de trois taches basiques appelées respectivement : Component_client1, Component_server1 et Connector_rpc. Ces taches correspondent respectivement au Composant Client, Composant Server et au connecteur d'assemblage RPC qui relie ces deux composants.

De plus, la tâche Component_server1 contient elle-même six sous-tâches : Component_cm, Component_sm, Component_db, Connector_cr, Connector_sq et Connector_sqlq. Ces sous-taches correspondent aux trois composants composites et aux trois connecteurs d'assemblage qui relient ces trois composants composites. En outre, l'exécution de ce programme Ada affiche les différentes interactions entre ces tâches. Ces interactions sont traduites par des rendez-vous Ada (Voir Figure 21). Chaque demande de rendez-vous est affichée par l'identité de la tâche demandée suivie d'une demande de rendez-vous. Et chaque acceptation de rendez-vous est affichée par l'identité de la tâche active suivie d'une acceptation de rendez-vous.

```
Component_client1(Connector_rpc.callerc_request)-
-Connector_rpc(callerc_request)
Connector_rpc(Component_server1.receiveRequest_request)-
-Component_server1(receiveRequest_request)
Component_server1(Component_cm.externalSocket_request)-
-Component_cm(externalSocket_request)
Component_cm(Connector_cr.requestorsc_requestInf)-
-Connector_cr(requestorsc_requestInf)
Connector_cr(Component_sm.securityAuthorization_requestInf)-
-Component_sm(securityAuthorization_requestInf)
Component_sm(Connector_sq.securityManager_query)-
-Connector_sq(securityManager_query)
Connector_sq(Component_db.securityManagerIntf_query)-
-Component_db(securityManagerIntf_query)
Component_db(Connector_sq.requestor_answer)-
-Connector_sq(requestor_answer)
Connector_sq(Component_sm.credentialQuery_answer)-
-Component_sm(credentialQuery_answer)
Component_sm(Connector_cr.grantor_positiveAnswer)-
-Connector_cr(grantor_positiveAnswer)
Connector_cr(Component_cm.securityCheckIntf_positiveAnswer)-
-Component_cm(securityCheckIntf_positiveAnswer)
Component_cm(Connector_sqlq.caller_query)-
-Connector_sqlq(caller_query)
Connector_sqlq(Component_db.queryIntf_query)-
-Component_db(queryIntf_query)
Component_db(Connector_sqlq.callee_answer)-
-Connector_sqlq(callee_answer)
Connector_sqlq(Component_cm.dbQueryIntf_answer)-
-Component_cm(dbQueryIntf_answer)
Component_cm(Connector_rpc.callees_reply)-
-Connector_rpc(callees_reply)
Connector_rpc(Component_client1.sendRequest_reply)-
-Component_client1(sendRequest_reply)
```

Figure 21. Exécution de l'application Client-Serveur hiérarchique

6. Étude de cas : Gestion de places du parking

Tout d'abord, nous allons modéliser en Wright l'application nommée "Parking Space Management" (Bhiri 2008). Ensuite, nous allons analyser statiquement la modélisation Wright obtenue en utilisant l'outil Wr2fdr. En outre, nous allons générer grâce à notre outil Wr2ada le code concurrent Ada correspondant. Enfin, nous allons analyser dynamiquement le code Ada généré.

6.1. Description informelle

Un parking est composé d'un ensemble de places. Chaque place possède un identificateur unique. Une voiture garée dans le parking occupe une place précise. Une voiture peut entrer et sortir via l'un des deux accès du parking.

Lorsqu'une voiture entre dans le parking, le système lui assigne la place qu'elle doit occuper. A l'extérieur du parking, un panneau d'affichage indique le nombre de places disponibles dans le parking.

6.2. Modélisation formelle en Wright

Nous avons retenu les deux types de composants Wright suivants : (1) Le type de composant Access permettant de réserver et libérer des places pour des voitures ; (2) Le type de composant Display jouant le rôle d'observateur car il est informé en permanence du nombre de places disponibles dans le parking. Également, nous avons identifié un type de connecteur Wright appelé Parking permettant de faire coopérer des composants (instances) de type Access et Display. Les interactions entre les composants de type Access et Display passent impérativement via des connecteurs de type Parking.

Le type de composant Access offre une interface appelée r1 (port en Wright) permettant de gérer l'arrivée d'une voiture sur l'un des deux accès du parking. Le comportement du port r1 est exprimé par un processus CSP permettant d'enchaîner les événements introduits

précédemment en utilisant les opérateurs CSP de préfixage (->), choix externe (ou déterministe []) et choix interne (ou non déterministe |~|). Ceci est fourni par la Figure 22.

La modélisation des aspects comportementaux du composant Access nécessite les cinq événements CSP suivants : (1) carArrival modélise l'arrivée d'une voiture sur l'un des deux accès du parking; (2) reserved traduit la demande de réservation d'une place dans le parking; (3) positiveAnswer traduit une réponse positive à la demande de réservation; (4) negativeAnswer traduit une réponse négative à la demande; (5) unreserved modélise la sortie d'une voiture via l'un des deux accès du parking. D'une façon informelle, le comportement du port r1 est : dès qu'il observe l'événement carArrival, il enchaîne (opérateur de préfixage ->) soit par l'initialisation de l'événement _reserved (le symbole _ distingue un événement initialisé d'un événement observé) soit par l'initialisation de l'événement _unreserved (choix interne |~|). L'événement _unreserved est suivi soit par une réponse positive (positiveAnswer) soit par une réponse négative (negativeAnswer) en tenant compte de l'environnement (ici l'état du parking). Ceci est exprimé par un choix externe ([] en CSP). La clause computation exprime le comportement global du composant Access. Un tel comportement fait référence au comportement partiel exprimé par le port r1.

```
Component Access
Port r1 = carArrival -> (_reserved -> (positiveAnswer -> r1 []
negativeAnswer -> r1) |~| _unreserved -> r1)
Computation = r1.carArrival -> (_r1.reserved ->
(r1.positiveAnswer -> computation [] r1.negativeAnswer ->
computation) |~| _r1.unreserved -> computation)
```

Figure 22. Le type de composant Access

Le type de composant Display offre une interface appelée screen permettant de visualiser le nombre de places disponibles dans le parking. Pour y parvenir, il introduit un événement appelé update (Voir Figure 23). Le comportement global du composant Display est exprimé par le processus CSP computation.

```
Component Display
Port screen = update -> screen
Computation = screen.update -> computation
```

Figure 23. Le type de composant Display

Le type de connecteur Parking offre trois interfaces (appelées rôles en Wright) : access1, access2 et display (Voir Figure 24). Ces interfaces expriment les comportements attendus des composants attachés (via les ports) au connecteur de type Parking. Autrement dit, le type de connecteur Parking supporte trois rôles modélisant les deux accès au parking et le panneau d'affichage. Contrairement aux comportements partiels exprimés par les trois rôles access1, access2 et display, la glue du parking exprime son **comportement global.**

```
Connector Parking
Role access1 = carArrival -> (_reserved ->
(positiveAnswer ->
access1 [] negativeAnswer -> access1) |~|
_unreserved -> access1)
Role access2 = carArrival -> (_reserved ->
(positiveAnswer ->
access2 [] negativeAnswer -> access2) |~|
_unreserved -> access2)
Role display = update -> display
Glue = _access1.carArrival -> (access1.reserved
->(_access1.positiveAnswer -> _display.update -> glue
|~|
_access1.negativeAnswer -> glue) []
access1.unreserved ->
_display.update -> glue) |~| _access2.carArrival ->
(access2.reserved -> (_access2.positiveAnswer ->
_display.update
-> glue |~| _access2.negativeAnswer -> glue)
[] access2.unreserved -> _display.update -> glue)
```

Figure 24. Le type de connecteur Parking

La Configuration de l'application "ParkingSpaceManagement" introduite dans la figure 25 comporte quatre instances : deux composants de type Access (AccessA et AccessB) un composant de type Display (display1) et un connecteur de type Parking (parking1). Les attachements entre les composants et le connecteur parking1 sont exprimés dans la clause **Attachments**.

```
Configuration ParkingSpaceManagement
....
Instances
accesA : Acces
accesB : Acces
display1 : Display
parking1 : Parking
Attachments
accessA.r1 As parking1.access1
accessB.r1 As parking1.access2
display1.screen As parking1.display
End Configuration
```

Figure 25 Configuration Parking Space Management

6.3. Analyse statique de la configuration Parking Space Management

Nous avons utilisé notre outil Wr2fdr afin de vérifier par model-cheking les propriétés standard sur l'application ParkingSpaceManagement. Ces propriétés couvrent la cohérence d'un composant, compatibilité port-rôle et l'absence de blocage d'un connecteur (voir Section 2.3).

Sachant que l'outil Wr2fdr accepte en entrée Wright et produit en sortie du CSP accepté par l'outil de model-cheking FDR. Les propriétés à vérifier formellement sont exprimés par la construction assert de l'outil FDR mettant en correspondance deux processus CSP (l'un abstrait et l'autre concret) par la technique de raffinement du CSP. Bien entendu, les deux processus abstrait et concret sont calculés par l'outil Wr2fdr en se basant sur la spécification Wright et la nature de la propriété à vérifier. La Figure 26 regroupe les assertions relatives à l'application ParkingSpaceManagement vérifiées par le model-checker FDR.

Xxxxx fig xxx

Figure 26. Propriétés standard vérifiées par FDR.

6.4. *Analyse dynamique de la configuration Parking Space Management*

La soumission de la configuration Parking Space Management à notre outil Wr2ada fournit le programme concurrent Ada donné par la figure 27. L'exécution du programme Parking Space Management affiche les différentes interactions entre ses constituants (tâches). Ces interactions sont traduites par des rendez-vous Ada (Voir Figure 28).

```
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Numerics.Discrete_Random;
procedure Psm is
function internal_condition return Boolean is
begin
subtype Intervalle is Integer range 0..1;
package Aleatory is new Ada.Numerics.Discrete_Random(Intervalle);
use Aleatory;
g:Generator; n:Integer;
begin
Reset(g); n:=Random(g); return n<1;
end internal_condition;
task Component_accesA is
entry r1_positiveAnswer ;
entry r1_negativeAnswer ;
entry r1_carArrival ;
end Component_accesA;
task Component_accesB is
entry r1_carArrival ;
entry r1_positiveAnswer ;
entry r1_negativeAnswer ;
end Component_accesB;
task Component_display1 is
entry screen_update ;
end Component_display1;
task Connector_parking1 is
entry acces2_reserved ;
entry acces2_unreserved ;
entry acces1_unreserved ;
entry acces1_reserved ;
end Connector_parking1
;
task body Component_accesA is
begin loop
accept r1_carArrival;
Put_Line("-Component_accesA(r1_carArrival)");
if internal_condition then
Connector_parking1.acces1_unreserved;
Put_Line("Component_accesA(Connector_parking1.acces1_unreserved)-");
else Connector_parking1.acces1_reserved;
Put_Line("Component_accesA(Connector_parking1.acces1_reserved)-");
select
accept r1_negativeAnswer;
Put_Line("-Component_accesA(r1_negativeAnswer)");
or
accept r1_positiveAnswer;
Put_Line("-Component_accesA(r1_positiveAnswer)");
end select; end if; end loop;
end Component_accesA;
task body Component_accesB is
begin loop
accept r1_carArrival;
Put_Line("-Component_accesB(r1_carArrival)");
if internal_condition then
Connector_parking1.acces2_unreserved;
Put_Line("Component_accesB(Connector_parking1.acces2_unreserved)-");
else
Connector_parking1.acces2_reserved;
Put_Line("Component_accesB(Connector_parking1.acces2_reserved)-");
select
accept r1_negativeAnswer;
Put_Line("-Component_accesB(r1_negativeAnswer)");
or
accept r1_positiveAnswer;
Put_Line("-Component_accesB(r1_positiveAnswer)");
end select; end if; end loop;
end Component_accesB;
task body Component_display1 is
begin loop
accept screen_update;
Put_Line("-Component_display1(screen_update)");
end loop;
end Component_display1;
task body Connector_parking1 is
begin loop
if internal_condition then
Component_accesA.r1_carArrival;
Put_Line("Connector_parking1(Component_accesA.r1_carArrival)-");
select
accept acces1_reserved;
Put_Line("-Connector_parking1(acces1_reserved)");
if internal_condition then
Component_accesA.r1_negativeAnswer;
Put_Line("Connector_parking1(Component_accesA.r1_negativeAnswer)-");
else Component_accesA.r1_positiveAnswer;
Put_Line("Connector_parking1(Component_accesA.r1_positiveAnswer)-");
Component_display1.screen_update;
Put_Line("Connector_parking1(Component_display1.screen_update)-");
end if;
or accept acces1_unreserved;
Put_Line("-Connector_parking1(acces1_unreserved)");
Component_display1.screen_update;
Put_Line("Connector_parking1(Component_display1.screen_update)-");
end select;
else Component_accesB.r1_carArrival;
Put_Line("Connector_parking1(Component_accesB.r1_carArrival)-");
select
accept acces2_reserved;
Put_Line("-Connector_parking1(acces2_reserved)");
if internal_condition then
Component_accesB.r1_negativeAnswer;
Put_Line("Connector_parking1(Component_accesB.r1_negativeAnswer)-");
else
Component_accesB.r1_positiveAnswer;
Put_Line("Connector_parking1(Component_accesB.r1_positiveAnswer)-");
Component_display1.screen_update;
Put_Line("Connector_parking1(Component_display1.screen_update)-");
end if;
or
accept acces2_unreserved;
Put_Line("-Connector_parking1(acces2_unreserved)");
Component_display1.screen_update;
Put_Line("Connector_parking1(Component_display1.screen_update)-");
end select; end if; end loop;
end Connector_parking1;
begin null; end Psm;
```

Figure 27. Code Parking Space Management généré par Wright2Ada

```
Connector_parking1(Component_accesB.r1_carArrival)-
-Component_accesB(r1_carArrival)
Component_accesB(Connector_parking1.acces2_unreserved)-
-Connector_parking1(acces2_unreserved)
Connector_parking1(Component_display1.screen_update)-
-Component_display1(screen_update)
Connector_parking1(Component_accesA.r1_carArrival)-
-Component_accesA(r1_carArrival)
Component_accesA(Connector_parking1.acces1_unreserved)-
-Connector_parking1(acces1_unreserved)
Connector_parking1(Component_display1.screen_update)-
-Component_display1(screen_update)
```

Figure 28. Exécution du programme Parking Space Management

D'une façon générale, le programme concurrent Ada généré par notre outil Wr2ada peut être utilisé dans plusieurs activités : simulation, vérification des propriétés spécifiques et raffinement. En effet, le client peut participer à la validation par la simulation de l'architecture abstraite en Ada produite par l'outil Wr2ada en préparant des scénarios d'exécution plausibles et voir s'ils sont satisfaits ou non par la simulation de l'architecture abstraite. En outre, l'architecte peut raffiner l'architecture abstraire en introduisant par exemple les données portées par les évènements (!, ?). Il peut également utiliser avec profit les outils d'analyse statique et dynamique associés à Ada afin de vérifier des propriétés spécifiques à l'application. Par exemple, en utilisant les outils d'analyse statique INCA et FLAVERS (Naumovich, 1997), nous pouvons prouver les propriétés comportementales d'un programme concurrent Ada. Enfin, si l'architecte avait opté pour Ada afin d'implémenter son futur système, il pourrait prendre comme point de départ le squelette généré par notre outil Wr2ada. En effet, un tel programme traduisant l'architecture abstraite

en Ada peut être raffiné step-by-step en prenant des décisions conceptuelles et techniques.

7. Discussion

Dans un premier temps, nous avons apporté quatre contributions à la version initiale de l'outil Wr2fdr développé à l'université de Carnegie Mellon autour de David Garlan. La première contribution concerne la détection et la correction des erreurs liées à l'implémentation de deux propriétés 2 et 3 relatives à l'absence de blocage des rôles et des connecteurs. La deuxième contribution touche à l'implémentation de la propriété 1 relative à la cohérence des composants. La troisième contribution concerne l'implémentation de la propriété 8 relative à la compatibilité port/rôle. Enfin, la quatrième contribution est liée à l'analyse de la sémantique statique de Wright. Pour y parvenir, nous avons dû travailler avec méthode sur un code de plusieurs milliers de ligne C++ développé par autrui. Ainsi, la nouvelle version de l'outil Wr2fdr est bel et bien opérationnelle.

Concernant notre second outil Wr2ada, la littérature montre que les travaux permettant d'établir des connexions automatiques entre des ADL formels et le langage concurrent Ada ne sont pas nombreux. Pour ce fait, nous proposons de comparer notre outil Wr2ada avec l'outil Ocarina [Lasnier 2009]. Ce dernier permet de manipuler les descriptions AADL pour générer du code Ada. La génération du code Ada permet de produire des systèmes exécutables en passant par l'environnement Ada95. Un programme exécutable est produit pour chaque noeud de l'application répartie. L'outil Ocarina accepte en entrée une description architecturale décrite en AADL. Tandis que notre outil Wr2ada accepte en entrée une description architecturale décrite en Wright. Celui-ci permet de décrire des architectures abstraites définies formellement en CSP. Le code Ada produit par l'outil Ocarina correspond à l'implémentation de la description architecturale concrète décrite par AADL. Tandis que le code Ada généré par notre outil Wr2ada correspond à l'architecture abstraite décrite par Wright. L'exécution du code Ada généré par notre outil est assimilée à une simulation du futur système. Par contre, l'exécution du code Ada produit par Ocarina traduit une utilisation réelle du système. En outre, le programme concurrent généré par notre outil Wr2ada peut être utilisé pour :

- **la validation** de l'architecture source en utilisant les outils d'analyse dynamique associés à Ada tels que le générateur de données de test et le débogage. En effet, l'exécution d'une architecture logicielle abstraite sur des données de test représentatives permet très tôt au client de participer à la validation de l'architecture de son futur système ;
- **la vérification** des propriétés spécifiques à l'architecture source en utilisant divers outils de vérification des programmes concurrents supportant Ada. Il existe une chaîne d'outils permettant aux programmes écrits en Ada d'être analysés statiquement par SPIN, SMV, FLAVERS et INCA (Dwyer et al., 1998). Ces quatre outils sont des représentants de quatre approches permettant l'analyse statique des programmes concurrents Ada ; ils sont complémentaires. En effet, vis-à-vis de la spécification des propriétés à vérifier, les outils SPIN et SMV favorisent les propriétés orientées état. Tandis que INCA et FLAVERS favorisent les propriétés orientées chemin ;
- **le raffinement** de l'architecture logicielle abstraite vérifiée (analyse statique en utilisant les outils de model-checking associés à Ada) et validée (analyse dynamique : exécution). Ceci permet l'obtention pas à pas d'une architecture concrète cohérente vis-à-vis de l'architecture abstraite validée et vérifiée. La correction de chaque étape de raffinement peut être assurée par les outils d'analyse statique associés à Ada : le programme concurrent raffiné doit conserver les mêmes propriétés vérifiées sur le programme concurrent abstrait.

8. Conclusion

Le problème de vérification et de validation de spécification formelle abstraite est un problème essentiel en génie logiciel. Dans ce travail, nous avons apporté deux contributions complémentaires pour la vérification et la validation d'architectures logicielles Wright. La première contribution est la réalisation d'une activité de maintenance corrective et évolutive sur l'outil Wr2fdr. Pour ce faire, nous avons effectué des tests syntaxiques orientés activité de test fonctionnel pour vérifier si l'outil Wr2fdr répond à ses spécifications. Cela nous a permis d'identifier les erreurs de l'outil Wr2fdr. Pour localiser et corriger les erreurs détectées par l'activité de test fonctionnel, nous avons suivi un processus de reverse-engineering sur l'outil Wr2fdr. Cela a permis la localisation de la classe et des méthodes responsables de ces erreurs. Ainsi, nous avons effectué un ensemble de corrections liées aux propriétés standard automatisées par l'outil Wr2frd. Ainsi, la nouvelle version de l'outil est bel et bien opérationnelle et peut être utilisée avec profit dans la vérification des architectures logicielles Wright.

Notre seconde contribution concerne la réalisation d'un outil IDM appelé Wr2ada. Ce dernier accepte en entrée une configuration Wright comportant des composants atomiques, composants composites, des connecteurs atomiques et des connecteurs composites. Il génère en sortie un programme concurrent Ada comportant plusieurs tâches (task en Ada). La réalisation de l'outil Wr2ada est basée sur les trois types d'opérations de transformation de modèles : texte vers modèle (T2M), modèle vers modèle (M2M) et modèle vers texte (M2T) en utilisant respectivement les outils adéquats Xtext, ATL et Xpand. Un tel programme concurrent Ada peut être validé par une activité de test : scénarios d'exécution issus du cahier des charges, politiques d'acceptation des rendez-vous inter-tâches. Également, il peut être raffiné en introduisant l'échange de données inter-tâches ou en implémentant le non-déterministe. Également, le programme concurrent Ada généré par notre outil Wr2ada peut être validé davantage en utilisant les outils d'analyse statique associés à Ada tels que FLAVERS et INCA.

A court terme, nous comptons améliorer davantage les outils développés dans le cadre de ce travail. En ce qui concerne l'outil Wr2fdr, deux prolongements pourraient être envisagés. Le premier concernerait l'intégration des composants et connecteurs composites avec des propriétés comportementales adéquates. Le second serait l'exploration de la spécification CSP produite par l'outil Wr2fdr afin d'exprimer des contrats comportementaux plus ou moins spécifiques à l'application traitée. En ce qui concerne l'outil

Wr2ada, nous envisageons l'intégration des facilités syntaxiques offertes par CSP de Wright et l'étude d'autres mécanismes d'implémentation des composants et connecteurs composites basés sur les tâches non imbriquées en Ada. A moyen terme, il serait intéressant de valider empiriquement la boîte à outils autour de l'ADL formel Wright : Wr2fdr, Wr2ada, Model-checker FDR2, environnement d'exécution des programmes Ada, outils d'analyse statique et dynamique associés à Ada. Pour y parvenir une application industrielle devrait être identifiée et développée step-by-step en utilisant judicieusement les outils associés à Wright précédemment cités. Une telle étude empirique devrait confirmer la complémentarité entre ces différents outils.

Références