Expérimentation de RAMSES

Nous avons défini le framework RAMSES qui implémente le processus de génération de code que nous avons proposé aux chapitres précédents. Ce framework s'appuie sur la formalisation d'un *workflow* qui orchestre le processus de génération en définissant différentes stratégies de génération aux différentes étapes de raffinement. Cette démarche a été intégrée au sein de l'environnement OSATE.

Dans ce chapitre, nous expérimentons ce framework sur le raffinement des communications entre tâches. Premièrement, la section 7.1 illustre les patrons de transformation dans ce contexte particulier. Nous démontrons ainsi l'applicabilité de ces patrons sur des cas concrets d'utilisation. Ensuite, la section 7.2 donne un exemple de processus mis en œuvre par RAMSES. Nous formalisons ce processus à l'aide du modèle de *workflow* que nous avons défini aux chapitres précédents et nous démontrons l'intérêt d'un tel processus capable d'adapter la génération de code selon les propriétés du système modélisé.

7.1 Cas d'applications des patrons de transformation

L'utilisation des patrons de transformation a permis de faciliter la mise en place de différents processus de génération de code. Nous illustrons chacun de ces patrons sur le raffinement des communications :

– *Stratégie* est appliqué pour faire varier le nombre d'appels à un composant selon l'implémentation choisie. Ainsi, *Stratégie* considère un même ensemble d'éléments d'entrée pour lequel le raffinement va être altéré selon le contexte.
– *Adaptateur* est illustré sur le raffinement des données partagées de types hétérogènes *data* et *port*. L'interface de ces types est virtuellement modifiée pour les rendre semblables. *Adaptateur* considère des éléments d'entrée de natures différentes pour appliquer le même traitement.
– *Substitution partielle* est appliquée sur le raffinement des connexions de ports entre threads. Ce patron consiste à altérer l'ensemble des éléments d'entrée d'un raffinement afin d'exclure un sous-ensemble et à appliquer un raffinement différent pour ce dernier.
– *Memento* est appliqué sur le raffinement des ports pour apporter des preuves que la génération ne viole pas les exigences. A la règle initiale est ajoutée la production d'éléments supplémentaires pour conserver trace du raffinement.

7.1.1 Stratégie

Le patron *Stratégie* s'applique pour modifier une partie du comportement d'une règle de transformation. Son principe est de déléguer certains traitements ou certaines expressions à une fonction annexe que l'on remplace selon le besoin. Nous l'avons exploité pour faire varier le nombre d'appels au composant *Send_Output* selon l'implémentation au sein d'une règle principale intégrant l'ensemble des composants de communication. Cette règle est illustrée par le listing 7.1 et concerne les implémentations *PDC_Indices* et *PDC_Indices_Array*. Pour ces deux versions, le composant *Send_Output* prend en paramètre le port de réception du destinataire. Pour cette raison, la règle parcourt 1) l'ensemble des ports d'émission de l'émetteur (ligne 16) puis 2) l'ensemble des ports de réception connectés à ce dernier (ligne 18). Enfin, 3) *Send_Output* est appelé et prend en paramètre le port de réception courant (lignes 19 et 20).

```
1  rule Insert_Communication_Services_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do
4    {
5      -- 1) Append calls to Receive_Input
6      for(inputPort in thread.features->select(f| f.isInput()
7              and f.isPeriodicDelayedEventDataPort())) {
8        seq <- seq.append(thisModule.resolveTemp(
9             Sequence{inputPort, behavior} ,'callReceiveInputs'));
10     }
11     -- 2) Append calls to Next_Value
12     ...
13     -- 3) Append calls to Put_Value
14     ...
15     -- 4) Append calls to Send_Output
16     for(outputPort in thread.features->select(f| f.isOutput()
17             and f.isPeriodicDelayedEventDataPort())) {
18       for(cnxInst in outputPort.srcConnectionInstance) {
19         seq <- seq.append(thisModule.resolveTemp(
20               Sequence{outputPort, behavior, cnxInst} ,'callSendOutput'));
21       }
```

```
22      }
23    }
24  }
```

Listing 7.1 – Règle de transformation introduisant les appels aux services de communication dans la séquence d'appel de chaque thread

Si nous souhaitons maintenant mettre en œuvre *PDC_InsertionSort*, il faut noter que le composant *Send_Output* ne prend aucune file de message en paramètre (voir section 5.3.4) : par conséquent il ne doit être appelé qu'une seule fois par période. La règle du listing 7.1 n'est donc pas réutilisable telle quelle. On peut cependant vouloir la réutiliser étant donné que les lignes 1 à 15 sont identiques (le nombre d'appels aux autres composants de communication étant le même entre les trois implémentations). Pour l'implémentation *PDC_InsertionSort*, les lignes 16 à 22 doivent être modifiées. Nous utilisons le patron *Stratégie* pour remplacer cette portion de code tout en conservant la logique globale de la règle existante. Cette portion de code est déléguée à une règle annexe (listing 7.2) et remplacée par un appel à cette dernière (listing 7.3).

```
1  rule Insert_Send_Output_Service_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do  {
4      for(outputPort in thread.features->select(f| f.isOutput()
5              and f.isPeriodicDelayedEventDataPort())) {
6        for(cnxInst in outputPort.srcConnectionInstance) {
7          seq <- seq.append(thisModule.resolveTemp(
8                Sequence{outputPort, behavior, cnxInst} ,'callSendOutput'));
9        }
10     }
11   }
12 }
```

Listing 7.2 – Règle de transformation annexe introduisant *Send_Output* pour chaque onnexion

```
1  -- 4) Append calls to Send_Output
2  thisModule.Insert_Send_Output_Service_In_CallSequence (thread, behavior)
```

Listing 7.3 – Délégation du bloc de code (lignes 14 à 20) à une règle annexe

Il suffit alors de *superimposer* cette règle annexe pour modifier ces quelques lignes de code et ainsi obtenir un unique appel au composant *Send_Output*. Ainsi, dans le cas de *PDC_InsertionSort*, une nouvelle définition de la règle *Insert_Send_Output_Service_In_CallSequence* est *superimposée* (listing 7.4) afin de modifier le comportement de la règle *Insert_Communication_Services_In_CallSequence*. La ligne 4 du listing 7.4 insère ainsi le composant *Send_Output* une seule fois par tâche.

```
1  rule Insert_Send_Output_Service_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do  {
4      seq <- seq.append (thisModule.resolveTemp(thread, 'callSendOutput'));
5    }
6  }
```

Listing 7.4 – Règle de transformation annexe introduisant *Send_Output* une fois par tâche

En appliquant *Stratégie* sur la règle *Protected_Shared_Data*, nous avons pu réutiliser une grande partie de la règle *Insert_Communication_Services _In_CallSequence* au lieu d'en définir une seconde similaire dans le cas de l'implémentation *PDC_InsertionSort*.

7.1.2 Adaptateur

Le patron *Adaptateur* vise à réutiliser une règle existante pour des éléments dont la logique de raffinement est similaire aux éléments d'entrée de la règle mais dont l'interface les rend incompatibles avec cette dernière. Le principe est de rendre virtuellement compatible l'interface des nouveaux éléments en définissant des fonctions annexes rattachées au type de ces éléments. Nous avons expérimenté ce patron sur le raffinement des ressources partagées entre tâches. Le modèle AADL du listing 7.5 spécifie une donnée *sharedData* (ligne 3) partagée en exclusion mutuelle entre les tâches *reader* et *writer* (ligne 5). La propriété *Concurrency_Control_Protocol* précise le protocole d'exclusion utilisé à la ligne 4.

```
1  process implementation p.impl
2    subcomponents
3      sharedData : data Base_Types::Float_32 {
4        Concurrency_Control_Protocol => PRIORITY_INHERITANCE_PROTOCOL; };
5      writer : thread writer.impl;  reader : thread reader.impl;
6    connections
7      acc1: data access sharedData -> writer.sharedData;
8      acc2: data access sharedData -> reader.sharedData;
9  end p.impl;
10
11 thread writer features
12   sharedData : requires data access Base_Types::Float_32;
13 end writer;
```

Listing 7.5 – Variable partagée et protocole d'exclusion mutuelle

Ce modèle utilise une notation implicite de l'exclusion mutuelle en se limitant à spécifier une propriété d'exclusion. Un raffinement possible d'un tel modèle serait d'expliciter les mécanismes utilisés pour délimiter les sections critiques. Ces mécanismes sont modélisés par les composants *Get_Resource* et *Release_Resource* que nous avions introduits aux chapitres précédents. La règle *Protected_Shared_Data* du listing 7.6 illustre ce raffinement : pour chaque donnée partagée, on la conserve (ligne 10 : copie de la donnée, ligne 11 : copie de son type) et on ajoute les composants *Get_Resource* et *Release_Resource* (lignes 12 et 22). Ces composants sont raffinés à partir de la ligne 12 notamment pour préciser le type de la donnée en entrée (ligne 17 : raffinement du prototype *resource_type* du composant *Get_Resource*).

```
1  rule Protected_Shared_Data {
2    from
3      e : AADL!Data (e.isSharedAndProtected())
4    using
5    {
6      protocolName : String = e.ownedPropertyAssociation->any(
7          pa|pa.property.name='Concurrency_Control_Protocol').ownedValue;
8    }
9    to
10     sharedData: AADL!DataSubcomponent (dataSubcomponentType <- sharedDataType),
11     sharedDataType: AADL!DataClassifier (name <- e.classifier.name, ...)
12     getResource: AADL!SubprogramType (
```

```
13          name <- 'Get_Resource',
14          ownedExtension <- thisModule.GetSubprogram ('Get_Resource', protocolName),
15          ownedPrototypeBinding <- Sequence {getResourceParam}
16      ),
17      getResourceParam: AADL!ComponentPrototypeBinding (
18          formal <- Get_Resource_PrototypeSpg.ownedPrototype
19                      ->any(e|e.name = 'resource_type'),
20          actual <- thisModule.CreateDataComponentPrototypeActual (sharedDataType)
21      ),
22      releaseResource: AADL!SubprogramType (...)
23  }
```

Listing 7.6 – Cas d'utilisation du patron *Adaptateur* pour factoriser le raffinement des ports et données partagées

Dans cet exemple, le patron *Adaptateur* consiste à réutiliser cette règle pour d'autres types de ressources partagées entre les tâches : les *ports*. En effet, le raffinement d'un port consiste à le traduire par exemple en un tableau accompagné de fonctions d'insertion et de suppression, ainsi que des composants *Get_Resource* et *Release_Resource* pour le protéger en exclusion mutuelle. Dans l'état actuel, la règle n'est pas réutilisable pour l'élément *port* : la propriété *Concurrency_Control_Protocol* n'étant pas applicable sur ce type d'élément (notamment par le fait que le protocole est souvent imposé par le support d'exécution dans le cas des ports et n'est pas paramétrable). Si l'on souhaite étendre la règle *Protected_Shared_Data* pour les ports, son exécution va renvoyer une erreur aux lignes 6 et 7 car le nom du protocole ne sera pas spécifié.

On définit alors deux fonctions annexes *getConcurrencyProtocol()* (listing 7.7), l'une obtenant le nom du protocole via la propriété *Concurrency_Control_Protocol* des composants *Data* (lignes 1 à 3), l'autre l'obtenant à partir d'une constante définie dans le module de transformation dans le cas des *Ports* (lignes 4 et 5). Le mot-clé *context* spécifie à quel type d'élément la fonction est rattachée.

```
1  helper context AADL!Data def : getConcurrencyProtocol() : String =
2    self.ownedPropertyAssociation->any(
3      pa|pa.property.name='Concurrency_Control_Protocol').ownedValue;
4  helper context AADL!Port def : getConcurrencyProtocol() : String =
5    'PRIORITY_CEILING_PROTOCOL';
```

Listing 7.7 – Application du patron *Adaptateur*

Les lignes 6 et 7 de la règle *Protected_Shared_Data* sont alors remplacées par un appel à la fonction annexe *getConcurrencyProtocol()*. Ainsi, ces lignes ne sont plus spécifiques aux composants *data* et sont également applicables pour les *ports*. On réécrit alors cette règle afin de la rendre générique et réutilisable pour les *data* et *ports*. Cette règle est renommée *Protected_Shared_ Resource* (listing 7.8) et reprend l'intégralité de la règle précédente. Le type d'entrée est modifié (ligne 2) par un type parent afin qu'il puisse être raffiné en port ou en data, et l'obtention du protocole (ligne 3) est également modifié afin d'utiliser les fonctions annexes précédentes.

```
1  abstract rule Protected_Shared_Resource {
2    from e : AADL!Element
3    using { protocolName: String = e.getConcurrencyProtocol(); }
4    to sharedData: AADL!DataSubcomponent (...),
```

```
5       ...
6   }
```

Listing 7.8 – Application du patron *Adaptateur* : définition d'une règle générique

Enfin, la règle générique est étendue pour préciser les spécificités de chaque type d'élément : data (listing 7.9) et port (listing 7.10).

```
1  rule Protected_Shared_Data extends Protected_Shared_Resource {
2    from e : AADL!Data (e.isSharedAndProtected())
3  }
```

Listing 7.9 – Application du patron *Adaptateur* : réécriture de la règle *Protected_Shared_Data*

Dans le cas des ports (listing 7.10), la nouvelle règle *Protected_Shared_EventDataPort* bénéficie ainsi des raffinements déjà définis (création des composants d'exclusion mutuelle) grâce à la fonction annexe *getConcurrencyProtocol()* créée précédemment. De plus, elle spécialise la règle précédente pour préciser la structure de la donnée *sharedData* qui représente le port (lignes 8 à 10 : propriétés modélisant un tableau).

```
1  rule Protected_Shared_EventDataPort extends Protected_Shared_Resource {
2    from e : AADL!EventDataPort (e.isInput())
3    to
4      sharedData: AADL!DataSubcomponent,
5      sharedDataType: AADL!DataType (
6        name <- e.name + '_Array',
7        ownedPropertyAssociation <- Sequence {
8            thisModule.CreatePA('Data_Model::Data_Representation', 'Array'),
9            thisModule.CreatePA('Data_Model::Base_Type', e.feature.dataClassifier),
10           thisModule.CreatePA('Data_Model::Dimension', e.getQueueSize())
11       }
12     ),
13 }
```

Listing 7.10 – Application du patron *Adaptateur* : extension de la règle générique

La règle générique peut être étendue pour tous les types de ressources partagées prises en compte (*e.g. Data Port* et *Event Port*) afin de factoriser la logique commune entre ces raffinements. Sans l'application de ce patron, le code de transformation aurait été dupliqué pour chaque règle.

7.1.3 Substitution partielle

Le patron *Substitution partielle* vise à assurer la compatibilité de plusieurs générateurs de code dont les plates-formes visées présentent des traductions différentes pour les mêmes éléments. Dans certains cas, des sous-ensembles d'éléments conduisent à des implémentations distinctes alors qu'ils seront assimilés à une unique implémentation dans d'autres cas. Nous avons expérimenté ce patron sur le partitionnement de l'espace d'adressage. Certaines plates-formes renforcent la protection des espaces d'adressage. C'est le cas des plates-formes ARINC653 qui correspondent à des architectures partitionnées. La notion de partition implique une isolation spatiale et temporelle des tâches de différentes partitions : protection contre une corruption de la mémoire causée par une autre partition et protection contre les retards éventuels de tâches de cette autre partition.

La figure 7.1 donne un modèle AADL constitué de deux espaces d'adressage *P1* et *P2* et trois tâches *P1_T1*, *P1_T2* et *P2_T1* réparties sur ces deux espaces d'adressage. La tâche *P1_T1* communique à la fois vers *P1_T2* et *P2_T1*. Dans le premier cas, *P1_T1* écrira directement dans la file de *P1_T2*. Dans le second cas, il passera par une file intermédiaire car il n'a pas le droit d'écrire dans l'espace d'adressage *P2*. Le module ARINC653 gère alors le transfert des messages de la file intermédiaire vers *P2_T1*. Pour une plate-forme d'exécution qui n'assure pas cette protection, *P1_T1* écrira par exemple directement dans la file de *P2_T1*.

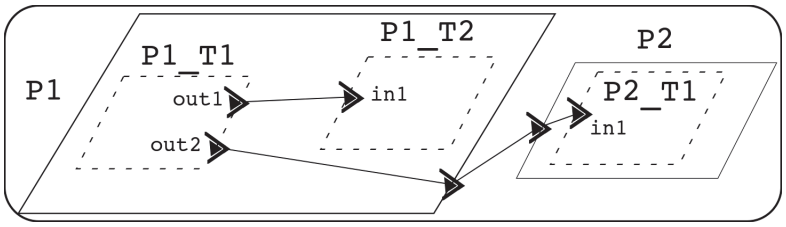
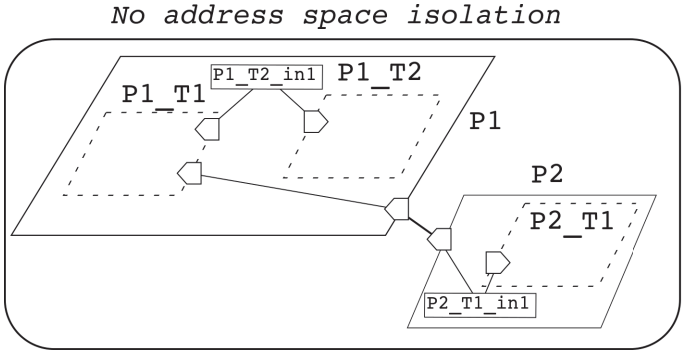


FIGURE 7.1 – Communications vers deux espaces d'adressage distincts

Cette propriété d'isolation impacte donc le raffinement des connexions. Nous distinguons deux types de connexions : *intra-process* (entre threads d'un même process) et *inter-processes* (entre threads de processes distincts). Si la propriété d'isolation est appliquée, il faut alors protéger les communications *inter-processes*. Par conséquent, chaque connexion *inter-processes* sera traduite par deux files de messages alors que chaque connexion *intra-process* sera traduite par une seule file (exemple précédent). Au contraire, dans le cas où cette propriété d'isolation n'est pas appliquée, alors toute connexion *intra-process* ou *inter-processes* sera traduite par une seule file de message. Finalement, cela revient à disposer de deux règles, l'une (R1) produisant un accès à la file de message du destinataire, l'autre (R2) produisant un accès à une file intermédiaire, et à réduire ou augmenter leur portée en fonction de la propriété d'isolation : s'il y a isolation, alors R1 se limite aux connexions *intra-process* et R2 aux connexions *inter-processes*. S'il n'y a pas isolation, R1 s'applique à l'ensemble des connexions et R2 ne s'applique pas.

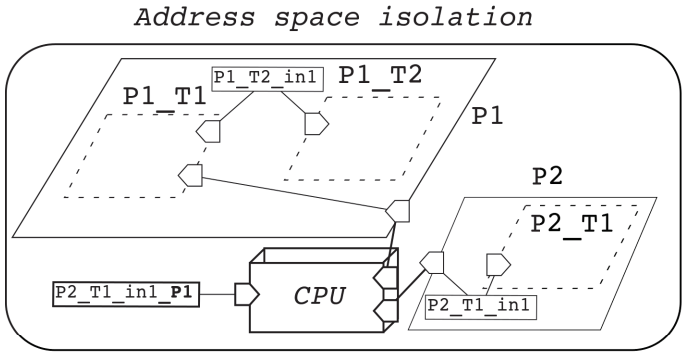
La figure 7.2 illustre la première situation : l'absence de protection (isolation) des espaces d'adressage. *R1* est appliquée à l'ensemble des connexions : la tâche émettrice *P1_T1* a accès aux files de message des destinataires (*P1_T2_in1* et *P2_T1_in1*) indépendamment du fait qu'ils soient situés dans le même espace d'adressage ou non. *R2* n'est donc pas définie dans cette situation.




```
rule R1{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads() )
 to
  a : AADL!DataAccess (...),
  c2: AADL!AccessConnection(...)
}
```

FIGURE 7.2 – Raffinement des connexions : aucune protection des espaces d'adressage

Le modèle de la figure 7.3 illustre le raffinement lorsque la protection est assurée. Cette fois-ci, la connexion vers *P2_T1* doit être protégée. Elle est alors raffinée par la règle *R2* au lieu de la règle *R1* comme précédemment. Cela implique de réécrire la règle *R1* pour restreindre sa portée aux connexions *intra-process*. Une condition *isSameProcess()* est alors ajoutée à la clause *from* de *R1*. Par conséquent, *R2* s'applique sur le sous-ensemble complémentaire *not isSameProcess()*. L'application de *R2* produit ainsi une file intermédiaire *P2_T1_in1_P1* accessible à *P1_T1* et dont les messages seront transférés par le système à la file destinataire *P2_T1_in1*.




```
rule R1{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads()
    and c.isSameProcess() )
 to
  a : AADL!DataAccess
      (...),
  c2: AADL!AccessConnection
      (...)
}
rule R2{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads()
    and not c.isSameProcess())
 to
  -- create intermediate
  -- message queue
  d : AADL!Data (...),
  ...
}
```

FIGURE 7.3 – Raffinement des connexions : protection des espaces d'adressage

L'utilisation du patron *Substitution partielle* vise à réutiliser la règle *R1* sans avoir à la redéfinir pour y ajouter une condition supplémentaire. Les deux définitions de *R1* diffèrent par la présence ou non de la condition *isSameProcess()*. On délègue cette condition à une fonction annexe *requiresProtection()* que l'on redéfinit selon la plate-forme d'exécution. *R2* va alors s'appliquer à l'ensemble des connexions qui valident la condition *requiresProtection()* tandis que *R1* va s'appliquer sur l'ensemble complémentaire. Le listing 7.11 donne la définition de cette fonction lorsque l'isolation n'est pas assurée (ligne 1). Aucune connexion ne nécessite de protection : *R2* n'est donc jamais appliquée tandis que *R1* s'applique à toutes les connexions.

```
1 helper context AADL!PortConnection def : requiresProtection() : Boolean = false;
2 rule R1 {
3   from c : AADL!PortConnection (c.isConnectedThreads()
4                                 AND NOT c.requiresProtection())
5   ... }
```

Listing 7.11 – Délégation de la portée de *R1* à une fonction annexe au sein du module *No Isolation*

Cette première définition de *requiresProtection()* constitue le module *No Isolation* qui contient également la règle *R1* (lignes 2 à 5). Si l'isolation doit être assurée, cette fonction annexe doit être redéfinie en *superimposant* un module *Isolation* (listing 7.12). Cette redéfinition (lignes 1 et 2) augmente la portée de *R2* (préalablement nulle), et diminue celle de *R1*. Ce module contient également la définition de *R2* (lignes 3 à 6). La *superimposition* du module *Isolation* au module *No Isolation* a pour effet d'inclure la règle *R1* tout en réduisant sa portée.

```
1 helper context AADL!PortConnection def : requiresProtection() : Boolean =
2         not self.isSameProcess();
3 rule R2 {
4   from c : AADL!PortConnection (c.isConnectedThreads()
5                                 AND c.requiresProtection())
6   ... }
```

Listing 7.12 – Redéfinition de la portée de *R1* au sein du module *Isolation*

Nous avons montré l'utilisation du patron *Substitution partielle* dans un cas concret. Le processus de génération est alors configuré explicitement par l'utilisateur pour une plate-forme particulière. Cela détermine les modules de transformation à sélectionner et ainsi de redéfinir les fonctions annexes adaptant la portée des règles de transformation.

7.1.4 Memento

Le patron *Memento* a pour but d'aider à la traçabilité des exigences. Il apporte des preuves que le code généré ne viole pas les exigences. Pour cela, il produit des traces qui mettent en relation les éléments de la spécification initiale (le modèle initial) et les éléments raffinés modélisant le code généré. Ensuite, ces éléments sont comparés pour vérifier que certaines propriétés sont conservées. Nous avons appliqué ce patron sur l'ensemble du processus de raffinement afin de retracer l'intégralité des éléments sources : notamment pour s'assurer que chaque donnée partagée et protégée en exclusion mutuelle et pour vérifier que le dimensionnement des ressources générées correspond à celui spécifié. Nous prenons le cas du raffinement des communications à travers les *ports*. Pour chaque port est générée une file de messages (*e.g.* tableau, liste chaînée, structure quelconque). On va vouloir vérifier que la taille de la file de message générée est cohérente avec celle du modèle initial.

Notamment, dans le cas des modèles de communication que nous avons présentés en section 5.3, on vérifie que la taille est cohérente vis-à-vis des calculs de bornes. L'utilisateur peut également spécifier ces exigences, comme sur le listing 7.13. Il spécifie un composant *thread* avec une file de messages *datain* de taille 10 stockant des données de type *Float_32*.

```
1 thread consumer
2 features
3   datain: in event data port Float_32 { Queue_Size => 10; };
4 end consumer;
```

Listing 7.13 – Modélisation d'un port de taille fixée par l'utilisateur

On s'attend alors à ce que le raffinement de *datain* en tableau conserve ces propriétés comme le montre le listing 7.14. Le composant *consumer_datain_Type* modélise la file de message sous forme de donnée. Les propriétés *Data_Representation*, *Base_Type* et *Dimension* indiquent respectivement la représentation de la donnée (tableau, énumération, ...), le type des éléments contenus dans la donnée ainsi que sa taille. On souhaite obtenir des traces associant les éléments initiaux (listing 7.13) aux éléments raffinés (listing 7.14) afin de vérifier leur équivalence, et ainsi la cohérence du raffinement.

```
1  data consumer_datain_Type
2  properties
3    Data_Model::Data_Representation => Array;
4    Data_Model::Base_Type => classifier (Float_32);
5    Data_Model::Dimension => (10);
6  end consumer_datain_Type;
```

Listing 7.14 – Modélisation d'un port de taille fixée par l'utilisateur

Pour cela, on applique le patron *Memento* sur la règle réalisant le raffinement : en annotant le résultat avec une structure *TraceLink*. Dans notre exemple, il s'agit du raffinement de ports en données. Celui-ci est réalisé par la règle *Protected_Shared_Port* qui a été présentée précédemment pour le patron *Adaptateur*. Cette règle est modifiée pour produire une trace du raffinement (listing 7.15) : Un élément *trace* a été ajouté (lignes 14 à 17) afin de mettre en relation l'élément source *e* et l'élément cible *sharedDataType*. Cette trace permettra de retrouver par la suite quel élément du modèle source est à l'origine de la création de l'élément *sharedDataType*. Cette trace permettra ensuite de vérifier notamment que la taille du tableau *sharedDataType* généré à partir du port *e* est de taille correspondant à celle spécifiée par la propriété *Queue_Size*.

```
1  rule Protected_Shared_Port extends Protected_Shared_Resource {
2    from
3      e : AADL!EventDataPort
4    using { size: Integer = getIntegerProperty('Queue_Size', e); }
5    to
6      sharedDataType: AADL!DataType (
7        name <- e.name + '_Array',
8        ownedPropertyAssociation <- Sequence {
9          thisModule.CreatePA('Data_Model::Data_Representation', 'Array'),
10         thisModule.CreatePA('Data_Model::Base_Type', e.feature.dataClassifier),
11         thisModule.CreatePA('Data_Model::Dimension', size)
12       }
13     ),
14     trace: Trace!TraceLink (
15       ruleName <- 'Protected_Shared_Port',
16       sourceEntities <- Sequence {e},
17       targetEntities <- Sequence {sharedDataType}
18     )
19 }
```

Listing 7.15 – Raffinement des queueing ports : application du patron *Memento*

Pour assurer une vérification des exigences sur l'ensemble du processus de transformation, il faut pour cela ajouter ce type de trace pour chaque règle de transformation. Cependant, cet ajout peut être réalisé automatiquement à l'aide d'une transformation d'ordre supérieur : la définition initiale de la règle *Protected_Shared_Port* (listing 7.10) est alors vu comme un modèle d'entrée d'une seconde transformation afin d'ajouter l'élément *trace* et ainsi obtenir la nouvelle règle du listing 7.15. Le listing 7.16 donne un exemple de transformation d'ordre supérieur réalisant l'ajout de traces d'exécution. La transformation utilise le méta-modèle ATL pour transformer les règles ATL (nous avons utilisé un méta-modèle ATL fictif). La règle *TransformationRule* (ligne 1) prend en entrée chaque règle ATL et en produit une identique à laquelle est rajoutée la production d'une trace dans la clause *to* de cette même règle (ligne 10).

```
1  rule TransformationRule {
2    from
3      r : ATL!TransformationRule
4    to
5      r2 : ATL!TransformationRule (
6          name <- r.name,
7          kind <- r.kind,
8          fromElements <- r.fromElements,
9          oclCondition <- r.oclCondition,
10         toElements   <- r.toElements->union(Sequence{trace}),
11         doElements   <- r.doElements ),
12     trace: ATL!OutputElement (
13         name <- 'trace',
14         type <- thisModule.GetType('Trace!TraceLink'),
15         attributeMapping <- Sequence {attrRuleName,attrFrom,attrTo} ),
16     attrRuleName: ATL!AttributeMapping (
17         name   <- 'ruleName',
18         values <- Sequence{r.name} ),
19     attrFrom: ATL!AttributeMapping (
20         name   <- 'sourceEntities',
21         values <- r.fromElements ),
22     attrTo: ATL!AttributeMapping (
23         name   <- 'targetEntities',
24         values <- r.toElements )
25 }
```

Listing 7.16 – Définition d'une transformation d'ordre supérieur

Nous avons présenté des cas d'applications pour les différents patrons de transformation introduits aux chapitres précédents. Ceux-ci simplifient le développement des générateurs de code grâce à la *superimposition* et à la réécriture des règles de transformation. Nous avons illustré ces patrons sur le raffinement des communications. La prochaine section décrit la troisième contribution sur la mise en œuvre de ce processus de raffinement des communications.

7.2 Processus de raffinement des communications

Dans cette section, nous illustrons notre troisième contribution : la *génération de composants de communication adaptés*. Premièrement, nous précisons en section 7.2.1 la structuration des raffinements en modules de transformations. Deuxièmement, nous expliquons en section 7.2.2 comment nous déterminons statiquement un ordre d'application de ces stratégies. Troisièmement, en section 7.2.3 nous formalisons le processus de raffinement selon l'ordre de sélection et la structuration des raffinements. Nous nous appuyons sur le méta-modèle proposé en section 5.1. Enfin, nous fournissons en section 7.2.4 différents scénarios afin d'illustrer l'application de chaque alternative : chacune est validée puis invalidée selon les contraintes du modèle initial.

7.2.1 Structuration des raffinements

Le processus de génération de code se constitue en particulier d'un ensemble d'étapes de raffinements des communications. Nous avons introduit trois approches différentes des communications

différées : *PDC_Indices*, *PDC_Indices_Array* et *PDC_InsertionSort*. Les deux premières correspondent à l'implémentation sans verrou que nous avons proposée aux chapitres précédents. Pour la première, les indices d'écriture/lecture sont calculés à l'exécution tandis qu'ils sont stockés en mémoire pour la seconde. La troisième correspond à une implémentation classique pour laquelle chaque file de message est une liste chaînée protégée par un verrou.

Les trois stratégies de raffinement des communications partagent cependant une logique commune. Par conséquent, celles-ci sont définies par un ensemble de modules de transformation *superimposés* et partagés. La figure 7.4 illustre la composition de ces stratégies.

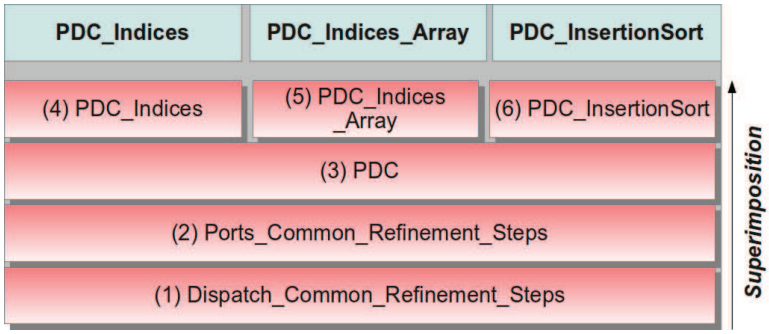


FIGURE 7.4 – Communications : structuration en modules de raffinement

– Le module 1 ***Dispatch_Common_Refinement_Steps*** correspond à l'introduction d'un gabarit d'automate comportemental pour chaque tâche selon le type d'activation de celle-ci (*e.g.* périodique, sporadique). Ce gabarit d'automate inclura par la suite des appels à différents composants de communication.
– Le module 2 ***Ports_Common_Refinement_Steps*** fournit des raffinements génériques d'éléments *port* en variables partagées dont le type est précisé par les modules suivants.
– Ensuite, le module 3 ***PDC*** contient les raffinements communs à l'ensemble des implémentations : composants *data* modélisant différentes constantes (*e.g.* taille de chaque port), le type des variables partagées modélisant les *ports* est raffiné en tampon circulaire.
– Enfin, les modules 4 et 5 (***PDC_Indices***, ***PDC_Indices_Array***) précisent les raffinements spécifiques pour les deux implémentations sans verrou. Le module 4 introduit les constantes et les fonctions nécessaires au calcul d'indice (*e.g.* identifiant de tâche, période, échéance, priorité) alors que le module 5 introduit les tableaux d'indices.
– Pour terminer, le module 6 ***PDC_InsertionSort*** est spécifique à la stratégie *PDC_InsertionSort*. Il spécialise le tampon circulaire en liste chaînée dont chaque message est daté. Il introduit également les composants associés à la liste chaînée (*e.g.* insertion triée) ainsi que les sections critiques (appels aux composants *Get_Resource*/*Release_Resource*).

La section suivante aborde l'ordre dans lequel les stratégies de raffinements sont testées.

7.2.2 Ordre de sélection de l'implémentation

L'objectif du processus de raffinement des communications est d'assurer une maîtrise du coût du code généré. Nous avons proposé trois implémentations alternatives. Chacune remplit des ob-

jectifs différents. D'une part, *PDC_InsertionSort* est une implémentation simple (insertion triée, verrous) qui ne nécessite aucune structure supplémentaire que la file de message. Par conséquent, celle-ci est assez simple à déboguer. Au contraire, les deux implémentations *PDC_Indices* et *PDC_Indices_Array* nécessitent des algorithmes et des structures de données supplémentaires qui complexifient l'analyse et le débogage du code exécuté. Cependant, ces deux alternatives ne nécessite aucun verrou et ont un coût potentiellement moindre que la première. Le tableau 7.1 compare les trois alternatives sur les éléments impactant l'empreinte mémoire et le temps d'exécution.

	PDC_Indices	**PDC_Indices_Array**	**PDC_InsertionSort**
Débogage	Difficile	Difficile	Facile
Empreinte mémoire (par file)	Taille file donnée en section 5.3.4	Idem PDC_Indices + Tableaux d'indices. Taille du code minimale (affectations).	Propriété *Queue_Size* (AADL).
Surcoût temporel (par file)	Calcul des indices	*Aucun*	Insertion triée (N) et verrous

TABLE 7.1 – Éléments spécifiques à chaque alternative

Ainsi, le premier objectif que nous nous fixons consiste à fournir une implémentation simple à analyser et à déboguer. Cependant, si celle-ci représente un coût trop important, alors on s'oriente vers une implémentation optimisée mais plus difficile à déboguer. La prochaine section formalise un tel processus.

7.2.3 Formalisation du processus de raffinement

Cette section décrit le processus de raffinement des communications. Le processus (figure 7.5) est constitué d'une première étape de raffinement dans laquelle le squelette de l'architecture logicielle est généré. Ensuite, le processus raffine les communications avec l'objet *Loop* qui sélectionne l'une des trois implémentations alternatives, chacune étant constituée de quatre modules de transformations *superimposés* (voir 7.2.1). Chaque implémentation est évaluée par une analyse d'ordonnancement puis par une analyse d'empreinte mémoire. L'implémentation est considérée valide si ces deux analyses réussissent (séquence de type *Conjunction*). Si l'implémentation courante est validée, le processus traduit le modèle raffiné en code exécutable (étape *Generation*). Dans le cas où aucune des trois implémentations n'a été validée, le processus entre dans un état d'erreur (étape *ErrorState*).

La prochaine section illustre l'exécution de ce processus et l'application des différentes stratégies en fonction du modèle d'entrée.

7.2.4 Évaluation

On dispose de plusieurs implémentations alternatives des mécanismes de communication. Le processus modélisé précédemment va alors tester ces alternatives dans différentes situations afin d'illustrer la validation puis l'échec de chacune.

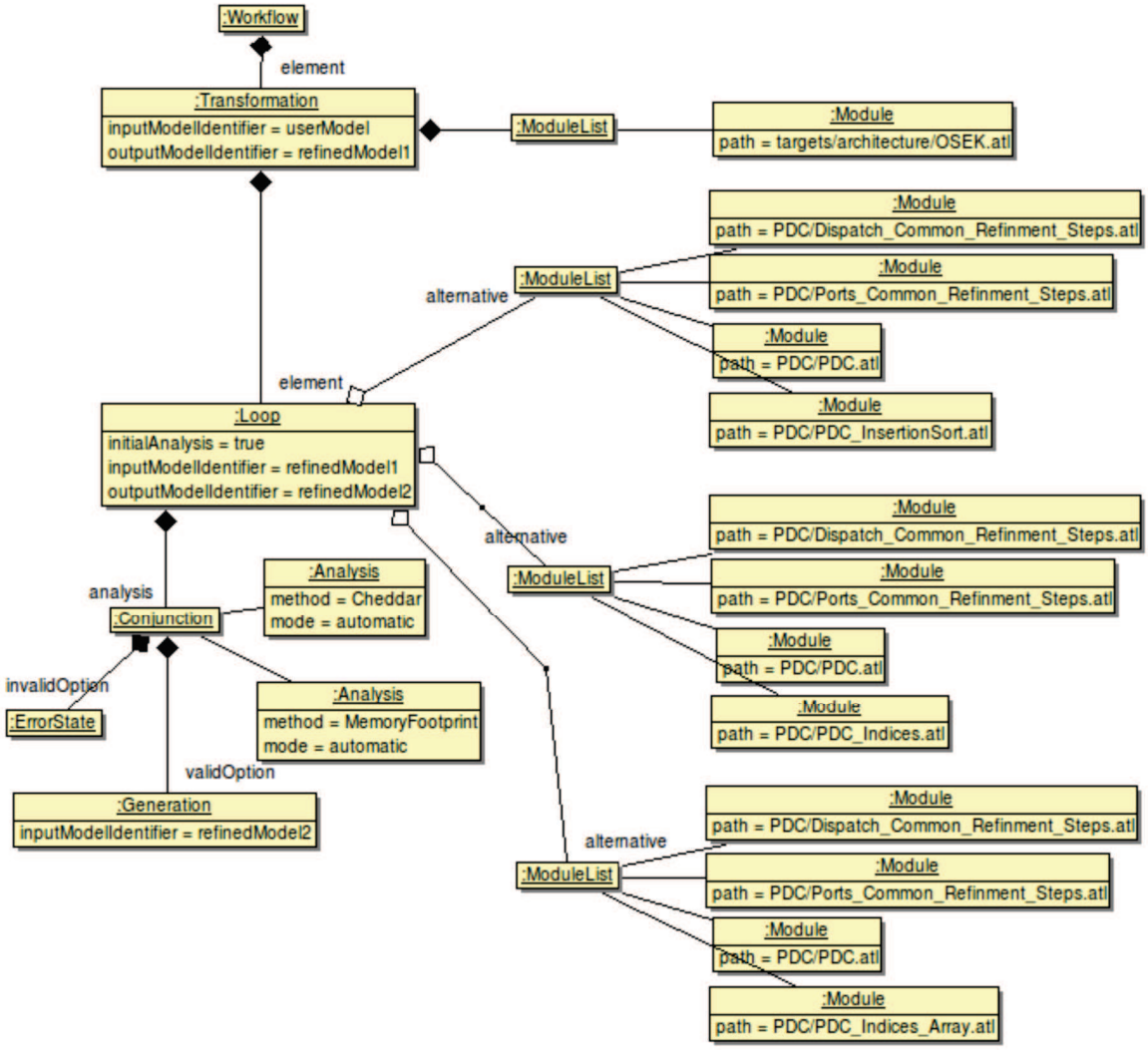


FIGURE 7.5 – Modélisation du processus de raffinement des communications

Pour évaluer notamment le temps d'exécution des modèles AADL, le processus d'évaluation nécessite une estimation des performances du système. Nous renseignons le modèle AADL initial avec une estimation des vitesses de lecture et d'écriture en mémoire. Sur la figure 7.17, nous indiquons ces propriétés pour un processeur de 200 MHz (lignes 3 et 4).

```
1 memory mainmem
2 properties
3  Read_Time  => [Fixed => 300 ns; PerByte => 300 ns;];
4  Write_Time => [Fixed => 300 ns; PerByte => 300 ns;];
5  ...
6 end mainmem;
```

Listing 7.17 – Expérimentations : modélisation des performances du système en AADL

Jeu de tâches Nous considérons un jeu de quatre tâches périodiques *T1*, *T2*, *T3* et *T4* ordonnancées selon *Rate Monotonic*. Le listing 7.18 spécifie les périodes et échéances des tâches ainsi que les échanges de messages, chaque message étant codé sur 4 octets. Les durées d'exécution seront fixées lors des différents scénarios proposés aux paragraphes suivants.

```
1  process implementation p.impl
2  subcomponents
3    T1: thread T1 {Period => 7 ms;  Deadline => 7 ms;};
4    T2: thread T2 {Period => 12 ms; Deadline => 12 ms;};
5    T3: thread T3 {Period => 20 ms; Deadline => 20 ms;};
6    T4: thread T4 {Period => 25 ms; Deadline => 25 ms;};
7  connections
8    cnx1: port T1.dataout -> T2.datain;  cnx2: port T1.dataout -> T3.datain;
9    cnx3: port T2.dataout -> T4.datain;
10   cnx4: port T3.dataout -> T4.datain;
11 end p.impl;
```

Listing 7.18 – Expérimentations : modélisation des communications

Nous fournissons au processus des modèles AADL basés sur ce jeu de tâches pour illustrer l'application des différentes stratégies d'implémentation. Nous faisons varier les propriétés telles que le temps d'exécution initial des tâches (*Compute_Execution_Time*) et le dimensionnement mémoire (*Word_Count*) pour aboutir à différentes implémentations.

Scénario 1 : PDC_InsertionSort Le premier scénario considère le WCET des tâches fixé à $C_{T1} = C_{T2} = C_{T3} = 1$ ms et $C_{T4} = 2$ ms. On obtient alors des pires temps de réponse respectivement de 1,2,3 et 5 ms : ce jeu de tâche est ordonnançable. En tenant compte du surcoût temporel lié à l'implémentation *PDC_InsertionSort*, le WCET des tâches est réévalué : on tient compte du temps nécessaire à l'envoi et à la réception d'un message. On obtient alors $C'_{T1} = 1.76$ ms, $C'_{T2} = C'_{T3} = 1.69$ ms et $C'_{T4} = 2.63$ ms. Le surcoût pour *T1* est plus important que pour *T2* et *T3* car le surcoût en émission est plus important qu'en réception. La nouvelle analyse d'ordonnancement réévalue les pires temps de réponse respectivement à 2, 4, 6 et 11 ms en se basant sur la borne supérieure des temps d'exécutions. Le système reste ordonnançable après prise en compte du surcoût temporel lié à l'implémentation *PDC_InsertionSort*. De plus, concernant l'empreinte mémoire, la taille du code généré est de 323 Ko. Cette stratégie est validée si le dimensionnement initial est suffisant.

Scénario 2 : PDC_Indices Ce second scénario considère un jeu de tâches similaire avec des durées d'exécution plus importantes : $C_{T1} = C_{T2} = C_{T3} = 2$ ms et $C_{T4} = 3$ ms. Ce nouveau modèle est ordonnançable. En reprenant la stratégie précédente *PDC_InsertionSort*, on réévalue ces temps d'exécution. Ainsi, on obtient $C'_{T1} = 2.76$ ms, $C'_{T2} = C'_{T3} = 2.69$ ms et $C'_{T4} = 3.63$ ms. Cela conduit à un pire temps de réponse plus pessimiste que précédemment : notamment pour *T4* on obtient 37 ms, signifiant que cette tâche a raté ses échéances. Le système n'est plus ordonnançable.

La stratégie suivante, *PDC_Indices*, est alors testée sur ce jeu de tâches. Cette nouvelle implémentation a un surcoût temporel moindre que la précédente mais a une empreinte mémoire plus importante, avec un surcoût global de 15 Ko de plus que la précédente (fonctions de communication plus volumineuses). Cette stratégie est validée notamment si le dimensionnement mémoire, spécifié initialement via la propriété *Word_Count*, est supérieur à la taille totale du code généré (338 Ko).

Scénario 3 : PDC_Indices_Array Le scénario précédent est donc invalidé si le dimensionnement mémoire initial est insuffisant. Dans ce cas, la troisième stratégie, *PDC_Indices_Array* est testée. Cette dernière a une empreinte mémoire moindre, s'expliquant par les fonctions de communication minimalistes qui nécessitent peu de code. Par conséquent la taille totale du code généré passe de 338 à 247 Ko. De plus, le surcoût temporel est quasiment nul. Cette dernière implémentation est donc la moins coûteuse en terme de temps d'exécution et d'empreinte mémoire. Cependant, le code généré est plus difficile à déboguer qu'avec les stratégies précédentes.

Scénario 4 : Etat d'erreur La dernière implémentation, *PDC_Indices_Array*, peut également être invalidée dans le cas d'un dimensionnement mémoire insuffisant. Le processus, ne disposant pas d'autres stratégies, entre dans un état d'erreur pour signaler l'incapacité à fournir une stratégie d'implémentation qui assure le respect des contraintes de temps et de dimensionnement mémoire.

Ces quatre scénarios permettent ainsi d'explorer les différentes branches du processus afin d'assurer le respect des contraintes du modèle. Nous avons donc montré une application concrète du processus de raffinement incrémental proposé et nous avons illustré sa capacité à adapter sa stratégie selon les contraintes du modèle.