OSATE

OSATE (*Open Source AADL Tool Environment*) est un environnement de développement open-source autour du langage de modélisation AADL développé par le SEI (*Software Engineering Institute*). Il s'adresse aux concepteurs de SETRC et fournit des outils de conception et d'analyse. Son architecture est illustrée par la figure 6.1. OSATE est construit autour de l'environnement Eclipse et de son framework de méta-modélisation EMF (*Eclipse Modeling Framework*). Eclipse fournit un ensemble d'abstractions pour spécialiser l'environnement à une application particulière. EMF est une surcouche pour la définition de langages spécifiques et pour la transformation de modèles : EMF inclut notamment le méta-métamodèle Ecore et le support du format d'échange XMI. Pour supporter l'intégration de plug-ins, OSATE définit également des abstractions pour mettre en place des outils d'analyse de modèles AADL ainsi que des générateurs de code pour des plates-formes d'exécution spécifiques.

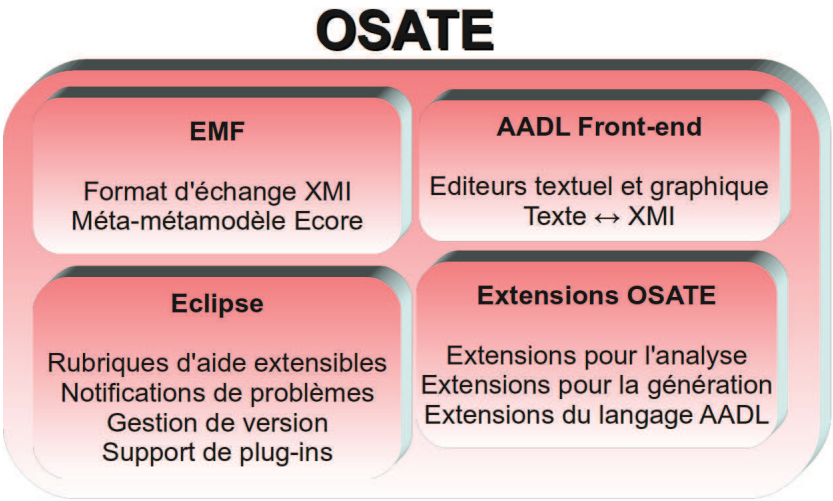

FIGURE 6.1 – Architecture d'OSATE

OSATE est une spécialisation du framework Eclipse [2] pour AADL. Le code source d'OSATE s'organise en un ensemble conséquent de 46 projets Java qui s'insèrent au-dessus du framework Eclipse/EMF. Parmi ces 46 projets, 12 sont dédiés à l'analyse (*e.g.* analyse architecturale, latence, consommation d'énergie...). Les projets restants définissent l'architecture d'OSATE. Notamment :

- ***org.osate.aadl2*** définit le méta-modèle AADL au format Ecore [17]. Ce projet contient également l'ensemble des classes Java qui constituent les éléments du méta-modèle. Elles sont automatiquement générées à partir du méta-modèle Ecore.
- ***org.osate.aadl2.instanciation*** implémente l'instanciation de modèles AADL, c'est-à-dire la sélection du composant AADL principal et la simplification du modèle (suppression des composants non utilisés, fusion des types hérités...).
- ***org.osate.aadl2.modelsupport*** fournit des mécanismes pour faciliter la mise en place de plug-ins d'analyse et de génération : méthodes de parcours des éléments d'un modèle AADL, notifications d'erreurs, utilisation des ressources internes à l'environnement (éditeur textuel, explorateur de projets).
- ***org.osate.annexsupport*** permet la définition et l'intégration d'annexes au langage AADL (*i.e.* extensions du langage). Il fournit les types génériques qu'il faut spécialiser pour assurer un

support complet d'une annexe.
La figure 6.2 donne un aperçu de l'environnement Eclipse dans lequel les développeurs créent leurs plug-ins OSATE (*i.e. back-end*).

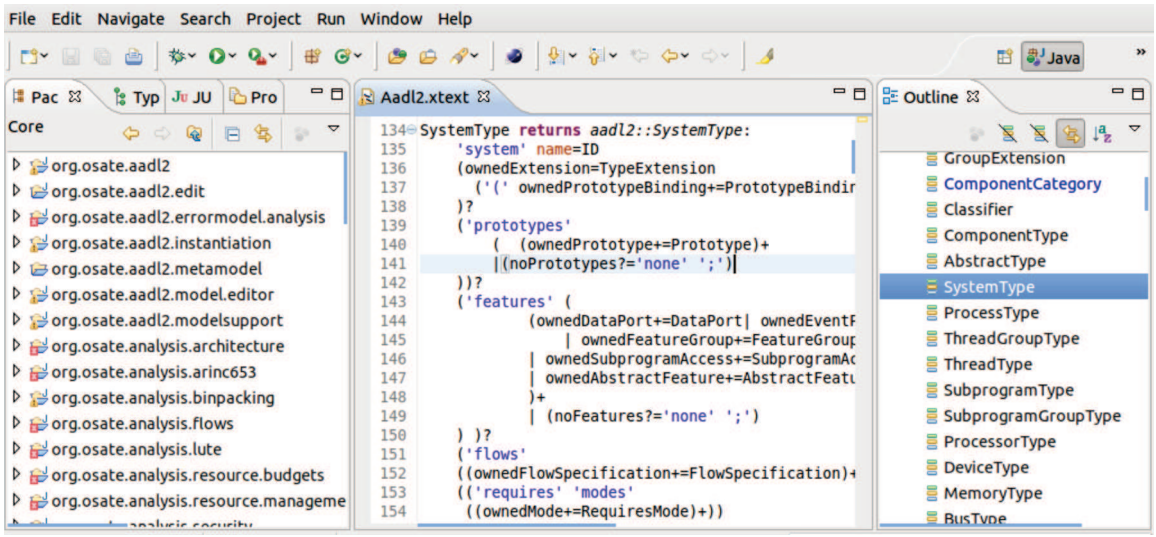

FIGURE 6.2 – Back-end d'OSATE

Cet ensemble de plug-ins construit l'interface utilisateur (*i.e. front-end*) illustrée par la figure 6.3. Cette interface utilisateur est constituée d'un explorateur de projets AADL, un éditeur textuel, un éditeur graphique, un arbre d'objets AADL ainsi qu'une console de notifications. Les plugins d'analyse et de génération de la *back-end* sont disponibles à l'utilisateur via des menus contextuels ou des boutons.

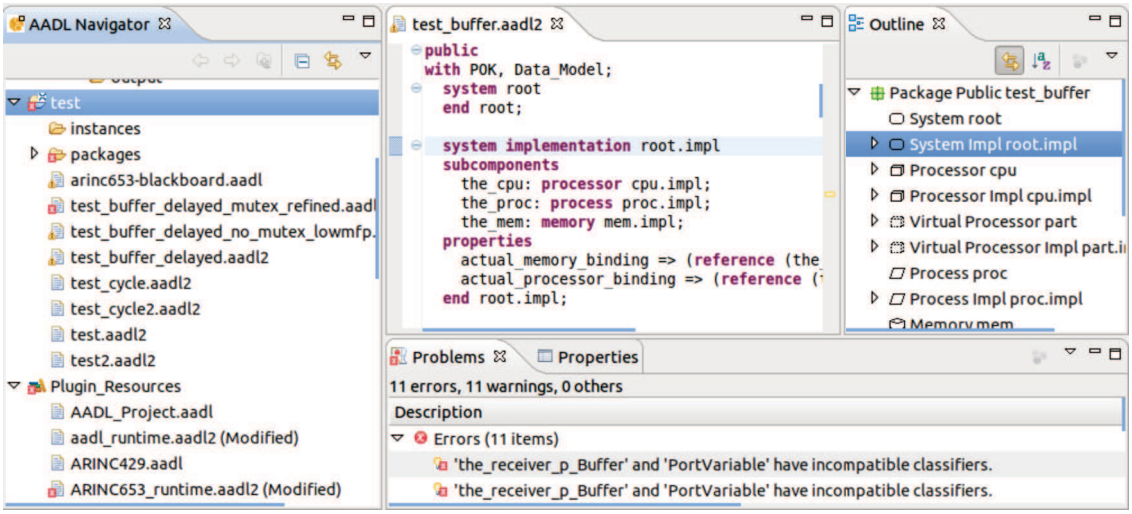

FIGURE 6.3 – Front-end d'OSATE

Cet environnement se constitue d'une surcouche à Eclipse développé essentiellement en langage Java. Nous avons contribué au développement d'OSATE via notre framework RAMSES

qui représente 26 projets Eclipse supplémentaires qui s'intègrent à l'architecture existante. Nous présentons RAMSES dans la section suivante.

6.2 Architecture de RAMSES

Le framework RAMSES est un ensemble de modules pour l'environnement OSATE. La figure 6.4 donne une vue d'ensemble de l'architecture de RAMSES. Celle-ci est constituée de quatre branches : *Lancement*, *Transformation*, *Analyse* et *Generation*. Celles-ci sont décrites dans les paragraphes suivants.

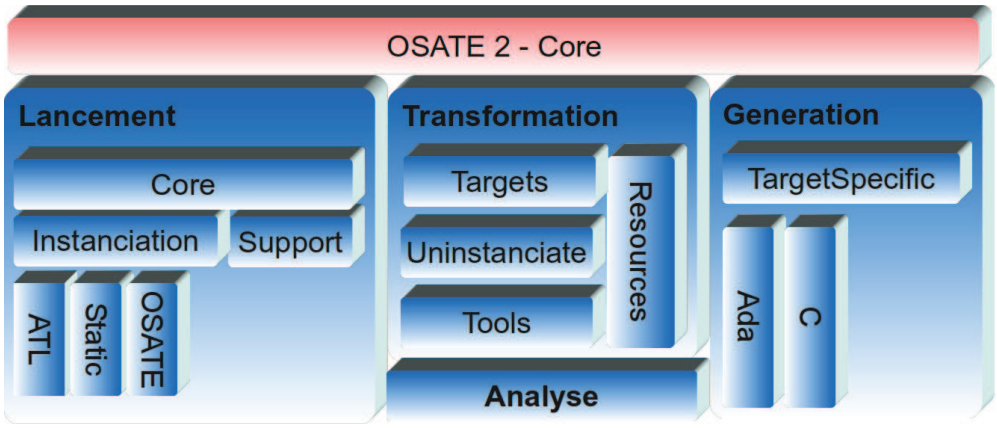

FIGURE 6.4 – Architecture de RAMSES

Lancement Cette branche a pour rôle de piloter le *processus de raffinement incrémental* et de coordonner ses différentes étapes (*e.g.* transformation, analyse, génération). Il s'agit du cœur du framework. Elle implémente la logique d'exécution du processus et contient les ressources nécessaires à son fonctionnement : méta-modèle du processus, moteur de transformation. Les plugins *Core* et *Support* assurent respectivement l'initialisation du processus en ligne de commande et son exécution. L'initialisation est réalisée en ligne de commande par l'utilisateur qui spécifie : une liste de modèles AADL, le code source métier ainsi que le fichier décrivant le processus. Le lancement est détaillé en section 6.3.

Transformation Cette branche réalise les étapes *Transformation* du processus en s'appuyant sur plusieurs bibliothèques :

– *Une bibliothèque de modules de transformation* (*Targets*) : ceux-ci réalisent le raffinement de différents éléments AADL. Pour favoriser la réutilisation et l'adaptation de raffinements, ces modules sont classés par plate-forme d'exécution (*e.g.* common, arinc653, osek) et par objectif de raffinement (*e.g.* communications, architecture). Ils sont référencés par l'utilisateur dans la description du processus pour définir les étapes de transformation : chaque étape référence une liste de modules qui seront *superimposés* lors de l'exécution du processus.
– *Une bibliothèque de fonctions d'interrogation* (*Tools*) : les fonctions d'interrogation sont des expressions OCL qui modélisent des contraintes plus ou moins complexes pour filtrer les éléments AADL sur lesquels les transformations sont appliquées. Ces interrogateurs améliorent ainsi la lisibilité des transformations.

– *Une bibliothèque de transformations identités* (*Uninstanciate*) : cette bibliothèque assure la conservation des éléments AADL lors du séquencement des étapes de transformation du processus (séparation des préoccupations). Ainsi, chaque étape de transformation s'applique potentiellement sur un sous-ensemble d'éléments du modèle. Le sous-ensemble complémentaire est cependant conservé pour être traité lors de la prochaine étape de transformation.
– *Une bibliothèque de modèles d'intergiciels* (*Resources*) : le raffinement ayant pour objectif d'intégrer au modèle les composants intergiciels mis en œuvre par la plate-forme d'exécution, ces derniers sont formalisés dans une bibliothèque de modèles AADL. Ces modèles de composants sont classés par plate-forme d'exécution et par objectif. Par exemple, le modèle *ARINC653.aadl* spécifie l'ensemble des composants représentant les fonctions et les types de données du standard ARINC653. Cette bibliothèque facilite la maintenance de ces composants indépendamment des modules de raffinement.

La branche *Transformation* est détaillée en section 6.4.

Analyse Cette branche réalise les étapes *Analysis* du processus. Elle contient les modules d' analyse de modèles AADL. Chaque objet *Analysis* référence l'identifiant de l'outil d'analyse (*i.e.* attribut *method*). Cet identifiant renvoie à un objet de type *Analyser* qui implémente l'analyse en elle-même. Ainsi, pour être intégré au sein d'un processus, chaque module d'analyse doit hériter du type *Analyzer*, comme le montre la figure 6.5. La méthode *getAnalyzerID()* renvoie l'identifiant de l'outil, correspondant à celui indiqué par l'attribut *method* de l'étape correspondante du processus. La classe doit également fournir les méthodes *performAnalysis()* et *setParameters()* pour respectivement lancer l'analyse et paramétrer l'outil. La méthode *setParameters()* est également utilisée pour stocker les résultats de l'analyse dans un paramètre d'entrée/sortie. Le type *Analyzer* assure donc l'intégration d'outils externes existants (*e.g.* AADL Inspector) et la définition d'analyses internes à RAMSES (*e.g.* calcul de WCET de modèles AADL comportementaux).

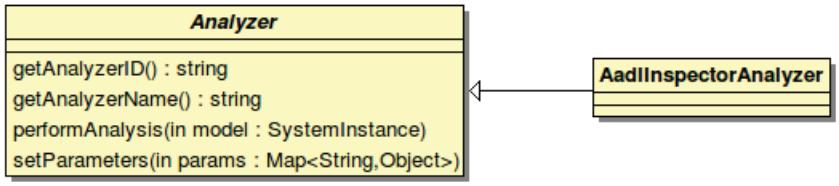

FIGURE 6.5 – Modélisation des outils d'analyse au sein de RAMSES

Nous prenons par exemple l'outil d'analyse *AADL Inspector* [1]. Cet outil réalise diverses analyses de modèles AADL, notamment des analyses d'ordonnancement à l'aide d'autres outils comme Cheddar. Nous souhaitons l'appeler depuis RAMSES : la ligne de commande pour appeler AADL Inspector est donnée par le listing 6.1. L'utilisateur souhaite réaliser une analyse d'ordonnancement du SETRC formalisé par les modèles AADL *mainmodel.aadl* et *runtime.aadl* (ligne 1). Pour cela, il a besoin du module Cheddar (ligne 2). Les résultats seront enregistrés dans un fichier *output.xml* (ligne 3). De plus, l'outil est exécuté en mode non interactif (ligne 4).

```
1 AADLInspector -a mainmodel.aadl, runtime.aadl
2         --plugin /home/myself/AADLInspector/config/plugins.common/cheddar.aip
3         --result /home/myself/output.xml
4         --show no
```

Listing 6.1 – Ligne de commande pour AADL Inspector

Pour intégrer cette ligne de commande au sein de RAMSES, une classe héritant de *Analyzer* doit être définie : la classe *AADLInspectorAnalyzer* (listing 6.2). La méthode *performAnalysis* appelle *AADL Inspector* en ligne de commande comme nous l'avons illustré précédemment. Les résultats d'analyses contenus dans le fichier XML de sortie sont alors extraits et mis en forme. La branche *Analysis* est détaillée en section 6.5.

```
1  public class AADLInspectorAnalyzer extends Analyzer {
2    public String getAnalyzerID(){ return "AADLInspector"; };
3    public String getAnalyzerName(){ return "AADL Inspector"; };
4
5    public void performAnalysis(SystemInstance model){
6       String modelsPaths = serializeAndReturnPaths(model);
7       Process p = Runtime.getRuntime().exec(new String[] {
8            "AADLInspector"                    ,
9            "-a", modelsPaths, "--plugin", PATH+"config/plugins.common/cheddar.aip",
10           "--result", OUTPUT_FILE_PATH, "--show", modeOption});
11      int exitValue = p.waitFor();
12      ...
13      getResults(new File(OUTPUT_FILE_PATH));
14   }...
15 }
```

Listing 6.2 – Intégration de l'outil AADLInspector

Generation Cette branche réalise les étapes *Generation* du processus. Il s'agit des modules produisant le code source exécutable à partir des modèles AADL raffinés. Pour chaque plate-forme d'exécution, une classe spécifique héritant de *AadlTargetUnparser* réalise la génération de code exécutable. Ainsi, les classes *AadlToPokCUnparser* et *AadlToOSEKCUnparser* réalisent la génération de code respectivement pour les plates-formes POK et OSEK. Par exemple, *AadlToPOKCUnparser* s'appuie sur une génération conforme au standard ARINC653 mais également sur une génération de fichiers de compilation (*makefiles*) spécifiques à POK.

Nous avons présenté l'architecture du projet RAMSES qui est une mise en œuvre du *processus de raffinement incrémental* que nous avons proposé au chapitre 4 et détaillé au chapitre 5. Les prochaines sections décrivent plus en détails les branches *Lancement* (6.3), *Transformation* (6.4) et *Generation* (6.6).

6.3 Lancement

Comme précisé précédemment, RAMSES est une surcouche à l'environnement OSATE et intègre le processus de raffinement incrémental décrit dans les chapitres précédents. Il dispose ainsi de l'environnement graphique d'OSATE : entre autres, un éditeur AADL ainsi que des analyseurs syntaxiques et sémantiques. Cependant, nous souhaitions automatiser des tests de non-régression. Par conséquent, RAMSES s'exécute principalement en ligne de commande. Le listing 6.3 donne un exemple de ligne de commande pour lancer un processus de raffinement incrémental avec RAMSES. Les options suivantes doivent être spécifiées :

– la liste des fichiers AADL définis par l'utilisateur (option *-m*),
– le nom du composant AADL principal englobant l'ensemble du système (option *-s*),

– l'identifiant de la plate-forme d'exécution ciblée (option *-g*), le chemin vers le répertoire dans lequel seront produits les modèles raffinés ainsi que le code généré (option *-o*),
– le nom du fichier XMI décrivant le *workflow* (option *–workflow*)

```
1 ToolSuiteLauncher -m ./input/model.aadl2 -s root.impl -g pok -o ./output/
2 --workflow=./input/Workflow.xmi
```

Listing 6.3 – Ligne de commande de lancement de RAMSES

Dans l'exemple donné par le listing 6.3, le modèle dont le chemin est *./input/model.aadl2* sera instancié à partir de son composant *root.impl*. Le modèle instancié est ensuite pris en entrée du processus de raffinement incrémental décrit dans le fichier *./input/Workflow.xmi*. Le code final sera généré pour la plate-forme d'exécution POK dans le répertoire *./output/*. Le fichier XMI décrivant le processus est une instanciation du méta-modèle de *workflow* comme indiqué précédemment. Le listing 6.4 en est un exemple. Dans cet exemple, le workflow est constitué d'une unique étape de transformation (ligne 7). Entre les balises <*list*> et </*list*> (lignes 9 à 13), l'utilisateur spécifie l'ensemble des modules de transformation qui composent cette étape. Elle est compilée juste avant son exécution en *superimposant* les modules dans l'ordre de déclaration. L'utilisateur indique ensuite que le modèle raffiné sera ensuite sauvegardé au format textuel (ligne 15) puis analysé par le logiciel *AADLInspector* (ligne 16). Si l'analyse réussie, le code est généré pour la plate-forme choisie au préalable (ligne 17). Sinon, le processus entre dans un état d'erreur (aucune stratégie alternative de raffinement n'étant formalisée, ligne 18).

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <rwf:Workflow xmi:version="2.0"
3   xmlns:xmi="http://www.omg.org/XMI"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:rwf="http://fr.tpt.aadl.ramses.control.workflow/Ramses/1.0"
6   xsi:schemaLocation="http://fr.tpt.aadl.ramses.control.workflow/Ramses/1.0
7     ../../../../fr.tpt.aadl.ramses.control.support/model/RamsesWorkflow.ecore">
8     <element xsi:type="rwf:Transformation">
9      <list>
10       <file path="ACG/targets/shared/SubprogramCallsCommonRefinementSteps"/>
11       <file path="ACG/targets/shared/PortsCommonRefinementSteps"/>
12       <file path="ACG/targets/shared/BehaviorAnnexCommonRefinementSteps"/>
13       <file path="ACG/targets/arinc653/ExpandThreadsDispatchProtocol"/>
14        ...
15     </list>
16     <element xsi:type="rwf:Serialization">
17       <element xsi:type="rwf:Analysis" method="AADLInspector" mode="automatic">
18         <yesOption><element xsi:type="rwf:Generation"/></yesOption>
19         <noOption><Error/></noOption>
20       </element>
21     </element>
22   </element>
23 </rwf:Workflow>
```

Listing 6.4 – Exemple de fichier de workflow

Les actions du processus sont directement visibles depuis l'explorateur d'OSATE (figure 6.6). L'utilisateur a fourni les ressources d'entrée du processus dans le répertoire *input* (modèles AADL, code source métier et description du processus). A l'exécution du processus, les étapes de type *Serialization* sauvegardent les modèles raffinés dans le répertoire *output/refined-models* tandis que

l'étape *Generation* génère le code exécutable dans le répertoire *output/generated-code*. Les modèles sauvegardés dans *refined-models* peuvent être exploités par un processus tierce pour vérifier la cohérence du *processus de raffinement incrémental*. Le code généré dans *generated-code* est automatiquement compilé pour produire un binaire enregistré dans ce même répertoire.

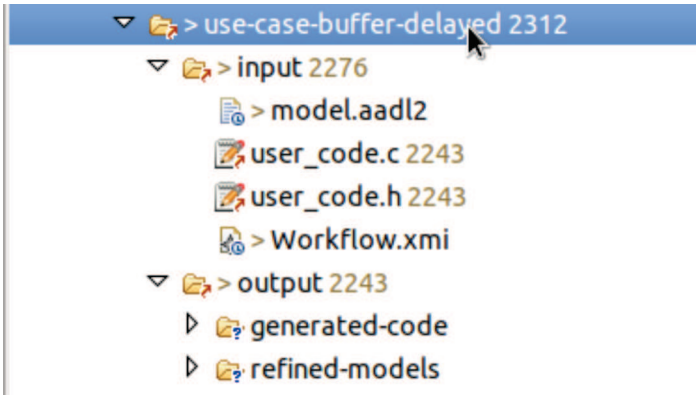

FIGURE 6.6 – Explorateur d'OSATE : organisation des fichiers générés par RAMSES

Nous avons donné une vue générale de RAMSES à travers les modules de lancement. Dans la prochaine section nous abordons la branche *Transformation* de RAMSES qui correspond à la mise en oeuvre des étapes de transformation.

6.4 Transformation

Les transformations de modèles réalisent des raffinements de modèles pour différentes plates-formes d'exécution. Le raffinement ayant pour objectif d'obtenir un modèle proche de l'implémentation réelle, il s'appuie en particulier sur une modélisation :

– *des ressources* fournies par la plate-forme d'exécution/l'intergiciel (*e.g.* composants intergiciels, structures de données) et introduits lors de la génération de code pour mettre en œuvre les abstractions du modèle.
– *du comportement des tâches* selon leur type d'activation (*e.g.* périodique, sporadique). La modélisation comportementale a pour objectif d'assurer que le code généré est cohérent avec la politique d'activation. Elle consiste à générer pour chaque tâche un automate comportemental correspondant à sa politique d'activation.

Modélisation des ressources En AADL, on modélise les ressources précédentes à l'aide de composants *subprogram* pour les fonctions et de composants *data* pour les types de données. La figure 6.7 donne un aperçu de la modélisation en AADL des ressources assurant l'exclusion mutuelle pour différentes plates-formes (ARINC653, POSIX et OSEK). Les variables partagées AADL (*data access*) sont traduites différement selon la cible : en sémaphore pour ARINC653, en mutex pour POSIX ou en ressource pour OSEK. De la même manière, chaque plate-forme fournit ses propres fonctions relatives aux variables partagées (initialisation, prise/relâche de verrou).

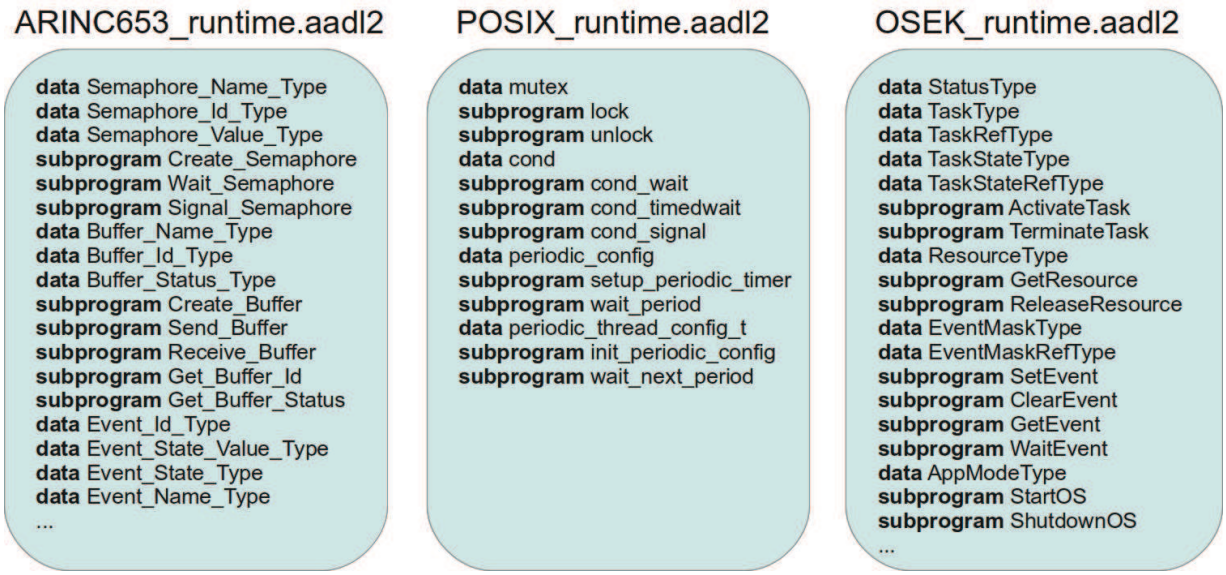

FIGURE 6.7 – Modélisation de différentes plates-formes d'exécution

L'implémentation concrète de chaque composant est spécifiée par l'utilisateur. Par exemple, le listing 6.5 décrit en AADL une partie des composants POSIX. La correspondance entre le composant AADL et les éléments de la plate-forme cible est spécifiée à l'aide des propriétés *Source_Name* et *Source_Text*, comme le montre l'exemple. Le composant AADL *mutex* (ligne 4) correspond au type *pthread_mutex_t* (ligne 6) qui est défini dans le fichier *pthread.h* et qui nécessite l'inclusion de l'archive *libpthread.a* (ligne 7). L'utilisateur peut également utiliser d'autres notations pour détailler l'implémentation du composant. Celles-ci seront abordées à la prochaine section.

```
1  data mutex
2  properties
3    Source_Name => "pthread_mutex_t"; Source_Text => ("pthread.h", "libpthread.a");
4  end mutex;
5  subprogram lock
6  features
7    m: in parameter mutex;
8  properties
9    Source_Name => "pthread_mutex_lock"; Source_Text => ("pthread.h", "libpthread.a");
10 end lock;
```

Listing 6.5 – Modélisation en AADL de composants POSIX

Ces modèles décrivant les composants de la plate-forme ciblée sont ensuite référencés par le processus de raffinement. Le modèle raffiné incorpore alors ces ressources pour se rapprocher de l'implémentation réelle.

Modélisation comportementale Le raffinement introduit également les ressources liées à l'activation des tâches. Ces ressources sont des modèles comportementaux qui correspondent au code généré pour l'activation des tâches. Chaque tâche a une politique d'activation (*e.g.* périodique) qui est spécifiée via la propriété *Dispatch_Protocol*. Dans [60] est proposée une démarche pour

expliciter le comportement des tâches : la valeur de la propriété *Dispatch_Protocol* est traduite en automate comportemental associé à la tâche concernée.

Prenons une tâche périodique initialement modélisée par un composant *thread* pour lequel est spécifié la propriété *Dispatch_Protocol* avec la valeur *Periodic*. La traduction de l'activation périodique en automate comportemental est donnée par le listing 6.6. L'automate est modélisé via l'annexe comportementale à partir de la ligne 1. Un automate est constitué d'un ensembles d'états, de transitions et éventuellement de variables locales. L'état initial est défini par le mot-clé *initial* tandis que les états terminaux sont définis via le mot-clé *final*. Un automate peut également avoir des états *complete* qui représentent une exécution sujette à interruptions/reprises basées sur des conditions de déclenchement externes. Dans notre exemple, une tâche périodique est modélisée via l'automate du listing 6.6. Celui-ci est constitué d'un unique état *stExec* (ligne 3). Il s'agit d'un état *complete*, dont l'unique transition sortante est donc déclenchée par la condition *on dispatch* (ligne 5). Cette condition signifie que la transition est franchie lorsque la tâche démarre une nouvelle activation (périodique). Cette transition est constituée d'un bloc d'actions défini entre accolades. Ces actions correspondent aux opérations réalisées par la tâche lorsque la transition est franchie (*e.g.* appels de *subprograms*, affectations, boucles internes, branches conditionnelles).

```
1 annex behavior_specification {**
2   states
3     stExec: initial complete final state;
4   transitions
5     stExec_Exec: stExec -[on dispatch]-> stExec
6       {  Send!(Data_Source); }
7 **};
```

Listing 6.6 – Automate comportemental d'une tâche périodique

Le listing 6.7 est un second exemple d'automate pour une tâche sporadique. La condition de déclenchement de la transition diffère de la précédente (ligne 5) car la tâche n'est pas activée périodiquement mais lors d'une mise à jour de la donnée partagée *Data_Sink*.

```
1 annex behavior_specification {**
2   states
3     stExec: initial complete final state;
4   transitions
5     stExec_Exec: stExec -[on dispatch Data_Sink]-> stExec
6       {  Get_Resource!(Data_Sink);
7          Update!(Data_Sink);
8          Release_Resource!(Data_Sink); }
9 **};
```

Listing 6.7 – Automate comportemental d'une tâche sporadique

La transformation consiste ainsi à raffiner certaines abstractions du modèle comme les moyens de communication et la sémantique d'exécution. Cela est réalisé grâce à des bibliothèques de composants AADL ainsi qu'à des automates comportementaux. Il est ensuite nécessaire d'évaluer l'impact de ce raffinement vis à vis des propriétés du modèle initial. Cela est abordé dans la section suivante.

6.5 Analyse

Le processus de raffinement introduisant des appels à des composants spécifiques, comme illustré précédemment, le surcoût engendré doit être évalué (*subprogram* : temps d'exécution, *data* : empreinte mémoire). En particulier, l'impact sur le temps d'exécution nécessite de réévaluer le WCET des tâches et les éventuels accès à des données protégées en exclusion mutuelle pour lesquels il faut identifier les dates de début et de fin des sections critiques. Deux méthodes de calcul de WCET sont proposées : la méthode dynamique [88] et la méthode statique [5, 4]. La méthode dynamique nécessite d'exécuter le code généré et est cependant souvent sujette aux imprécisions. A l'opposé, la méthode statique ne requiert pas l'exécution du code généré et se base sur des modèles mathématiques. Par conséquent, nous orientons notre démarche d'analyse sur cette seconde méthode.

6.5.1 WCET fixe ou variable

Pour tenir compte de l'impact des composants intergiciels sur le WCET des tâches, il est nécessaire d'évaluer le WCET de ces composants. Ces derniers sont modélisés par des *subprograms* AADL comme nous l'avons indiqué en section 6.4. Leur WCET est spécifié via la propriété *Compute_Execution_Time*. Cependant, certains composants ont un WCET variable. Par exemple, la fonction *Put_Value*, réalisant l'insertion d'une donnée dans une file, a un WCET qui dépend de la taille des données (temps de copie) et potentiellement de la taille de la file (temps de parcours). Nous considérons alors deux cas de figure : le WCET fixe et le WCET variable.

6.5.1.1 WCET fixe

Le WCET est fixe et est connu avant raffinement. Le *subprogram* est alors annoté de la propriété *Compute_Execution_Time* comme illustré par le listing 6.8 (ligne 9). Cette propriété indique alors les bornes minimum (*BCET*) et maximum (*WCET*) du temps d'exécution du composant. Dans ce cas de figure, la fonction peut être implémentée directement dans le langage cible. Le modèle indique alors le chemin vers le fichier source ainsi que le nom de la fonction réelle à l'aide des propriétés *Source_Text* et *Source_Name* comme illustré (lignes 7 et 8).

```
1  subprogram Create_Semaphore
2    features
3      SEMAPHORE_NAME      : in parameter Semaphore_Name_Type;
4      ...
5      RETURN_CODE         : out parameter Return_Code_Type;
6    properties
7      Source_Name => "CREATE_SEMAPHORE";
8      Source_Text => ("arinc653/semaphore.h");
9      Compute_Execution_Time => 0 ms .. 1 ms;
10 end Create_Semaphore;
```

Listing 6.8 – Annotation du WCET d'un composant subprogram

6.5.1.2 WCET variable

Le WCET est variable. Dans ce cas de figure, illustré par le listing 6.9, le WCET dépend de paramètres qui ne sont connus qu'après raffinement du composant. Le composant *Put_Value* de l'exemple (ligne 6) réalise l'insertion de la donnée *value* dans le tableau *buffer* de type *BufferType*. La taille du tableau, spécifiable via la propriété *Dimension*, n'est cependant pas indiquée car le composant n'est pas spécifique à un modèle particulier. La propriété *Dimension* sera indiquée sur l'élément du modèle connecté au composant. Cet exemple montre qu'ici le WCET ne peut être spécifié par une propriété.

```
1  data BufferType
2  properties
3    Data_Model::Data_Representation => Array;
4  end BufferType;
5
6  subprogram Put_Value
7  features
8    value:  requires data access ValueType;
9    buffer: requires data access BufferType;
10 ...
11 end Put_Value;
```

Listing 6.9 – Modélisation d'un composant dont le WCET n'est connu qu'après raffinement

Pour évaluer le WCET lors du raffinement, il est nécessaire de détailler l'implémentation du composant. Lors du raffinement, cette description sera traduite en graphe d'exécution sur lequel sera calculé le WCET. L'annexe comportementale étant adaptée à la description fine de l'implémentation, le listing 6.10 reprend l'exemple précédent en y ajoutant cette annexe. Elle détaille une implémentation possible du composant *Put_Value* : celui-ci détermine d'abord à quel indice la donnée doit être insérée dans le tableau *buffer* selon la datation de la donnée (ligne 16). Les données du tableau sont ensuite décalées vers la droite pour insérer la nouvelle donnée (lignes 21 et 22). Pour déterminer le WCET de *Put_Value*, il faut notamment connaître le nombre maximum d'itérations réalisées sur la boucle *while* (ligne 16). Cette borne dépend de la taille du tableau *buffer* qui ne sera connue que lorsque ce paramètre sera précisé lors de l'appel au composant *Put_Value*. La spécification du composant étant incomplète (paramètres définis partiellement), on déclare un ensemble de prototypes (ligne 2) qui devront être précisés pour compléter la définition du composant.

```
1  subprogram Put_Value
2    prototypes
3      valueType : data ValueType;
4      arrayType : data BufferType;
5    features
6      value : in parameter valueType;
7      buffer : requires data access arrayType;
8    annex behavior_specification {**
9      states
10       s : initial final state;
11     variables
12       index: Integer;
13     transitions
14       t : s-[]->s {
15         index := 0;
16         while ((buffer[index].timestamp < value.timestamp)
```

```
17                  and (index < buffer.size)){
18              index := index + 1
19            }
20            if (index < bufferSize) {
21              shiftValues!(buffer,index);
22              buffer[index] := value
23            }
24          }
25    **};
26  end Put_Value;
```

Listing 6.10 – Modélisation d'un composant à WCET variable à l'aide de l'annexe comportementale

Le listing 6.11 donne un second exemple dans lequel ce composant est intégré. Les paramètres sont raffinés à la ligne 3 à l'aide des prototypes définis précédemment. En particulier, on précise le type exact du tableau. Il s'agit du type *TheReceiver1_datain* déclaré à la ligne 7. A ce moment, la taille du tableau et le type des données sont indiqués via les propriétés *Dimension* et *Base_Type*.

```
1  subprogram Put_Value_TheSender1
2  extends RuntimeExample::Put_Value
3  (valueType => data IntegerValue, arrayType => TheReceiver1_datain)
4  end Put_Value_TheSender1;
5
6  data IntegerValue extends ValueType ...
7  data TheReceiver1_datain extends BufferType
8  properties
9    Data_Model::Dimension => (10);
10   Data_Model::Base_Type => (classifier (IntegerValue));
11 end BufferType;
```

Listing 6.11 – Modèle raffiné : intégration du composant *Put_Value*

Une alternative à l'utilisation de l'annexe comportementale est l'utilisation d'une séquence d'appel. Celle-ci décompose la fonction en une séquence de fonctions secondaires et donne la possibilité de séparer les sous-fonctions de WCET fixe de celles de WCET variables. Le listing 6.12 illustre ce second exemple : une séquence d'appel scinde la fonction *Compute* en deux sous-fonctions *Compute_1* et *Compute_2*. La première a un WCET fixe annoté avec la propriété *Compute_Execution_Time* (ligne 13) tandis que la seconde a un WCET variable et son implémentation est détaillée avec l'annexe comportementale (ligne 17). Le WCET du composant global est alors calculé en faisant la somme des WCET des sous-fonctions.

```
1  subprogram implementation Compute.impl
2  calls
3    seq : { call1 : subprogram Compute_1;
4            call2 : subprogram Compute_2; };
5  connections
6    cnx1: data access inputValue -> call1.inputValue;
7    cnx2: data access call1.outputValue -> call2.inputValue;
8  end Compute.impl;
9
10 subprogram Compute_1
11 ...
12 properties
13   Compute_Execution_Time => 0 ms .. 1 ms;
```

```
14 end Compute_1;
15 subprogram Compute_2
16 ...
17 annex behavior_specification { ... };
18 end Compute_2;
```

Listing 6.12 – Description d'un composant à WCET variable à l'aide de la séquence d'appel

Performances de la plate-forme d'exécution L'évaluation du WCET doit également tenir compte des instructions de base de la plate-forme d'exécution. Par exemple, l'utilisation de l'annexe comportementale introduit des instructions spécifiques (*e.g.* affectations, opérations arithmétiques, comparaisons) dont il faut évaluer le temps d'exécution. Pour cela, le langage AADL fournit des propriétés standard pour préciser les performances de la plate-forme d'exécution, et permettant de calculer la durée de ces instructions. Ces propriétés sont réparties sur les différents composants. Le tableau de la figure 6.8 liste les propriétés que nous prenons en compte dans l'analyse de WCET. Celles-ci modélisent à gros grain les performances.

Propriété	**Composant**	**Description**
Source_Data_Size	Data	Taille du type de donnée
Write_Time	Memory	Vitesse d'écriture en mémoire d'une donnée
Read_Time	Memory	Vitesse de lecture en mémoire d'une donnée
Word_Size	Memory	Taille de la plus petite donnée qu'il est possible de stocker en mémoire. Cette constante détermine notamment la taille en mémoire des opérateurs de base.
Assign_Time	Processor	Vitesse de chargement d'un bloc d'octet sur le processeur.

FIGURE 6.8 – Propriétés AADL spécifiant les performances de la plate-forme d'exécution

Ces propriétés sont utilisées lors du processus d'évaluation du WCET, notamment lors de la construction d'un graphe d'exécution pour lequel les sommets sont caractérisés par une durée. Nous avons abordé les différents éléments de modélisation AADL qui seront utilisés pour calculer le WCET des tâches après leur raffinement. Nous présentons le processus d'évaluation dans la section suivante.

6.5.2 Processus d'évaluation

Le processus d'évaluation de l'impact du raffinement sur le temps d'exécution est illustré par la figure 6.9. Le principe de ce processus est d'analyser l'impact du raffinement sur les propriétés telles que le WCET et les dates des sections critiques. Pour cela, il crée un graphe d'exécution qui rassemble de manière uniforme l'ensemble des descriptions comportementales utilisées (*e.g.* automate comportemental, séquence d'appel).

Le processus se déroule en trois étapes. Premièrement, on vérifie que le modèle AADL raffiné est suffisamment déterministe pour être analysé. Pour cela, nous restreignons l'ensemble des automates comportementaux à ceux respectant le profil *Ravenscar*. Deuxièmement, on rassemble au sein d'un unique graphe d'exécution l'ensemble des descriptions comportementales de la tâche

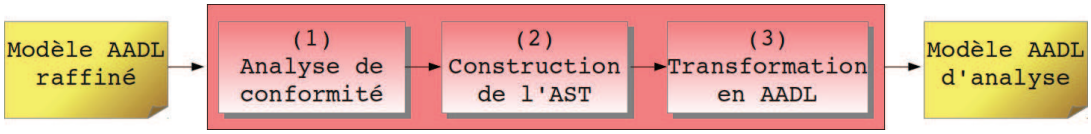

FIGURE 6.9 – Processus d'évaluation des surcoûts temporels

courante des composants appelés. Le graphe est réduit (*e.g.* dépliage de boucles, fusion de blocs de calculs) afin d'obtenir une modélisation simplifiée. Enfin, le graphe simplifié est traduit en automate comportemental simplifié. Le processus produit alors un modèle AADL d'analyse pour lequel les détails de mise en œuvre sont remplacés par des automates comportementaux simplifiés dédiés à l'analyse.

1. Analyse de conformité *Ravenscar* La première étape est d'assurer que l'automate comportemental est déterministe. Pour cela, le profil *Ravenscar* définit un ensemble de restrictions : (R1) aucune allocation dynamique de mémoire, (R2) la tâche réalise une boucle infinie déclenchée par un unique événement, (R3) la tâche ne se termine pas, (R4) les données partagées sont accédées en exclusion mutuelle.

On vérifie alors que l'automate respecte ces restrictions. Concernant (R1), le langage AADL ne permet pas de modéliser de l'allocation dynamique étant donné qu'il s'adresse aux systèmes temps-réel critiques. Ensuite, on analyse l'ensemble des transitions de l'automate pour détecter s'il existe une unique boucle principale (R2). On en déduit l'état qui modélise le comportement périodique (on sépare la phase d'initialisation de la phase de régime permanent). Pour s'assurer que la tâche ne se termine jamais (R3), on vérifie qu'il n'y a aucune transition vers l'état final de l'automate. Enfin, concernant les données partagées (R4), nous traitons exclusivement les files de messages pour lesquelles nous avons proposé plusieurs mécanismes assurant l'exclusion mutuelle (section 5.3) : l'utilisation de verrou ou une attribution des indices qui garantit des intervalles d'exclusion.

2. Construction du graphe d'exécution Lorsque l'automate est validé, on le convertit en graphe d'exécution. Etant donné les différents types d'éléments de modélisation qui renseignent l'implémentation d'une tâche (*e.g.* annexe comportementale, séquence d'appel, propriétés), le graphe d'exécution a pour objectif d'obtenir un unique modèle comportemental qui rassemble l'ensemble des notations utilisées. La figure 6.10 donne un exemple de modèle AADL décrivant le comportement d'une tâche à l'aide de différentes notations imbriquées. La tâche est décrite à l'aide d'un automate comportemental. Celui-ci référence des *subprograms* qui sont également décrits à l'aide d'un automate comportemental ou d'une séquence d'appel. La figure 6.11 donne le modèle de graphe d'exécution utilisé. Le type *ASTNode* modélise un noeud du graphe. Son attribut *kind* indique s'il s'agit d'une action quelconque (*Compute*) ou bien d'un début/fin de section critique (*CriticalSectionStart/CriticalSectionEnd*). Dans le cas d'une section critique, l'attribut *sharedData* référence la donnée partagée.

Le processus va alors analyser l'ensemble des descriptions comportementales associées à chaque tâche pour produire le graphe d'exécution. Celui-ci est crée avec un ensemble de nœuds connectés

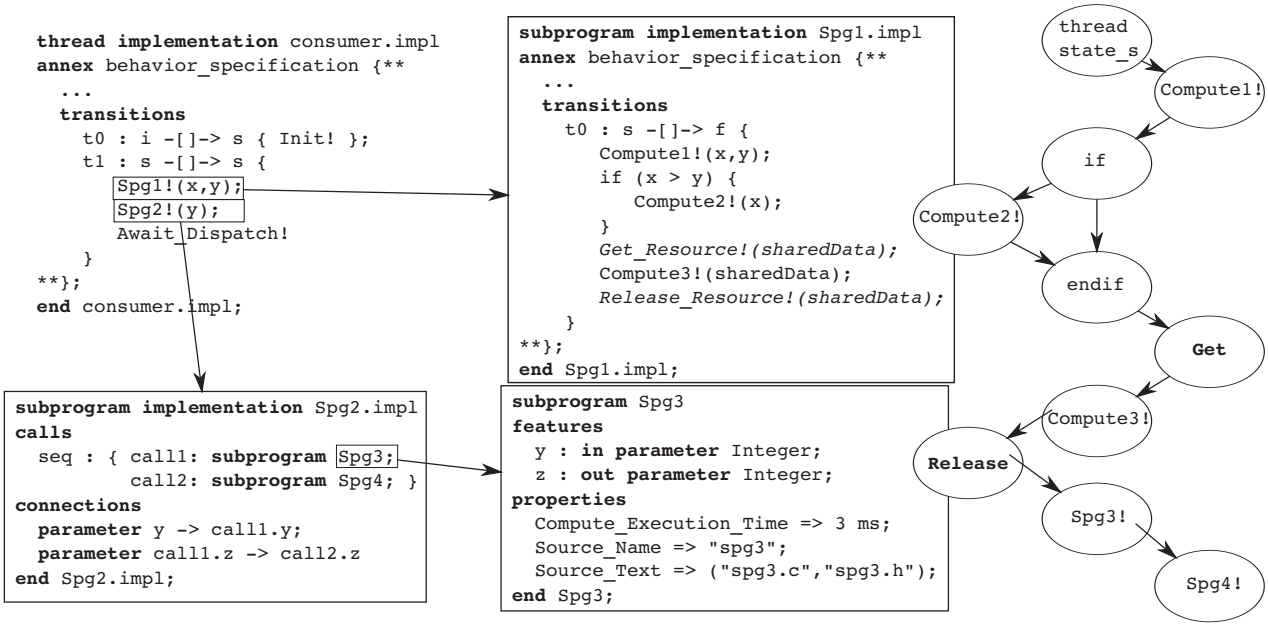

FIGURE 6.10 – Imbrication des descriptions comportementales AADL

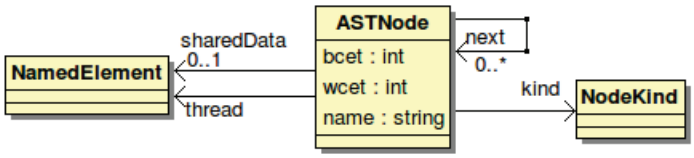

FIGURE 6.11 – Modélisation du graphe d'exécution pour le calcul du WCET

qui correspondent aux états et transitions de l'automate. On traite ensuite les blocs d'instructions de chaque transition. On considère alors différents types d'instructions :

– **Affectation** : un nœud est ajouté au graphe d'exécution et la propriété *wcet* est initialisée avec la durée maximum de l'affectation. La durée de l'affectation tient compte de trois durées :

  1. ***Assign_Time*** : le temps pour charger les instructions sur le processeur,
  2. ***Read_Time*** : le temps pour lire les opérateurs et le contenu des opérandes,
  3. ***Write_Time*** : le temps pour écrire le résultat dans une variable.

  Chaque expression d'affectation est décomposée en sous-expressions de la forme $A := B\ op\ C$ pour lesquelles est calculé le WCET à partir de ces trois durées :

$$
\begin{aligned}
WCET &= Assign\_Time + Read\_Time + Write\_Time \\
Assign\_Time &= Assign\_Time(A) + Assign\_Time(B) + Assign\_Time(op) + Assign\_Time(C) \\
Read\_Time &= Read\_Time(B) + Read\_Time(C) \\
Write\_Time &= Write\_Time(B\_C) + t_{op}
\end{aligned}
$$

  Chaque type d'opération nécessite un temps d'opération variable. Pour chaque opérateur op, on tient compte de la durée t_{op} pour réaliser l'opération. Ces paramètres sont fixés par l'utilisateur selon l'architecture.

– **Conditionnelle** : les blocs conditionnels sont traduits en un sous-graphe. Il se compose d'un nœud *if* relié à deux sous-graphes *then* et *else*. Les sous-graphes se rejoignent au noeud *endif* modélisant la fin du bloc conditionnel. Lorsque les sous-graphes sont construits, on détermine celui dont le WCET est le plus grand par une analyse de graphe. L'ensemble du bloc conditionnel est alors remplacé par un unique nœud dont l'attribut *wcet* correspond au WCET précédent auquel est ajouté la durée d'évaluation de la condition du *if*.
– **Boucle** : chaque boucle est dépliée avant d'être intégrée au graphe d'exécution. On suppose que le nombre maximum d'itérations est fourni par une variable. Il peut s'agir d'une boucle de type *for* ou de type *while*. Dans le second cas, on cherche une sous-condition (*AND*) du while qui utilise un compteur d'itérations sous la forme $iter < expr$. On vérifie dans le corps de la boucle que le compteur est incrémenté avec une constante en cohérence avec l'opérateur de comparaison. Le nombre maximum d'itérations est déterminé par la valeur de l'expression de la sous-condition. Un exemple d'une telle boucle est donné en listing 6.10. L'expression doit correspondre soit :
  – à une constante explicite,
  – à une constante fournie par un paramètre d'entrée de type *data*. On considère que le composant *data* est une constante s'il n'est pas typé et qu'il possède la propriété *Initial_Value* indiquant sa valeur initiale. Ce second cas est illustré par le listing 6.13.

```
1 data MyConstant
2 properties
3   Data_Model::Initial_Value => ("10");
4 end MyConstant;
```

Listing 6.13 – Modélisation d'une constante en AADL

  – à une constante implicite *size* pour toute donnée de type tableau. Cette constante renvoie la taille du tableau, spécifiée via la propriété *Dimension*. Un exemple a été donné précédemment en listings 6.10 et 6.11.
– **Appel de *subprogram*** : on distingue plusieurs cas. Si la propriété *Compute_Execution_Time* est annotée au *subprogram*, alors on crée un noeud dont l'attribut *wcet* est initialisé avec la valeur de cette propriété. S'il n'en a pas, on analyse sa description comportementale. S'il n'en possède aucune, le processus notifie un avertissement et considère sa durée nulle. S'il possède un automate comportemental, alors on extrait le graphe d'exécution de celui-ci avec la même méthode de construction. Sinon, on analyse sa séquence d'appel. Pour chaque *subprogram* appelé, on extrait son sous-graphe et on l'incorpore dans le graphe principal. On tient compte également de la durée du passage de chaque paramètre au *subprogram*. Cette durée est calculée de la façon suivante. Pour chaque paramètre : 1) on détermine quelle variable est connectée au *subprogram* 2) on obtient la taille de la variable à partir de sa propriété *Source_Data_Size* 3) on calcule le temps pour charger la variable selon les propriétés *Assign_Time* et *Read_Time* (voir paragraphe affectation).
– **Appel aux *subprograms Get_Resource* et *Release_Resource*** : ces fonctions correspondent respectivement au début et fin de section critique. On effectue la même opération que pour un appel de *subprogram*. Cependant, on insère un nœud en début et un second en fin du sous-graphe pour modéliser la section critique. On initialise d'une part leur attribut *kind* respectivement avec les constantes *CriticalSectionStart* et *CriticalSectionEnd*, et d'autre part leur attribut *sharedData* avec une référence vers la donnée partagée.

3. Traduction du graphe d'exécution en automate comportemental Lorsque ce processus a été réalisé sur l'ensemble des tâches du modèle AADL, on réalise alors la production d'un modèle AADL d'analyse. Celui-ci remplace les descriptions comportementales des tâches par des

automates comportementaux utilisant des notations simplifiées. Ceux-ci sont ensuite interprétés par *AADLInspector* pour évaluer les écarts de performance avec le modèle non raffiné. Le procédé de traduction du graphe d'exécution en automate comportemental est le suivant :

- On produit un automate comportemental constitué de trois états i, s et f modélisant l'initialisation, l'exécution et la terminaison (état jamais atteint, en accord avec le profil Ravenscar). L'état s boucle sur lui-même pour modéliser l'exécution sans fin de la tâche.
- On parcourt les nœuds du graphe d'exécution. Ce dernier étant préalablement réduit (boucles, conditionnelles), il est parcouru à la manière d'une liste. On traduit chaque nœud en instruction selon l'attribut *kind* :
  - *Compute* : le noeud est traduit en instruction *Computation*. Celle-ci prend en paramètre la valeur de l'attribut *wcet*.
  - *CriticalSectionStart* : le noeud est traduit en instruction $*! <$ (notation réduite de *Get_Resource*).
  - *CriticalSectionEnd* : le noeud est traduit en instruction $*! >$ (notation réduite de *Release_Resource*).

Le processus d'analyse est illustré par la figure 6.12. Premièrement, le graphe d'exécution est construit à partir du modèle AADL. Ensuite, celui-ci est transformé en une simple séquence afin de réduire sa complexité. Enfin, le graphe d'exécution réduit est traduit en automate comportemental AADL réduit.

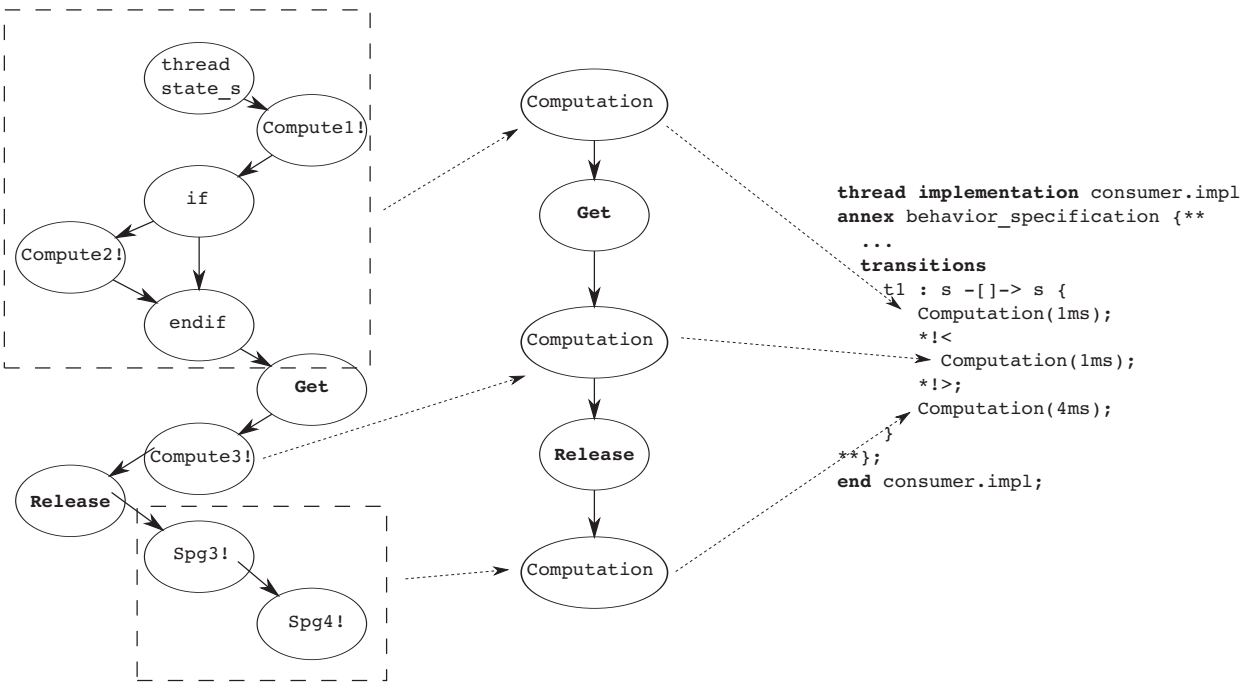

FIGURE 6.12 – Transformation du graphe d'exécution en modèle AADL d'analyse

Nous avons décrit le processus d'évaluation des surcoûts temporels lors du raffinement. Le raffinement du modèle AADL intégrant de nouveaux composants, ceux-ci sont pris en compte en construisant un graphe d'exécution intermédiaire qui est ensuite traduit en automate comportemental AADL. Le modèle AADL résultant de ce processus intègre alors les surcoûts temporels et peut être analysé. Le processus de génération peut alors être maîtrisé grâce à cette technique. La prochaine section aborde la phase finale de génération de code.

6.6 Génération de code

La dernière étape du processus est la génération de code exécutable. RAMSES fournit pour chaque plate-forme d'exécution un générateur de code spécifique. Le générateur respecte une architecture particulière illustrée par la figure 6.13. Le type *AadlTargetUnparser* définit la classe mère pour l'ensemble des générateurs de code développés pour RAMSES. Ce type assure une certaine organisation du code généré.

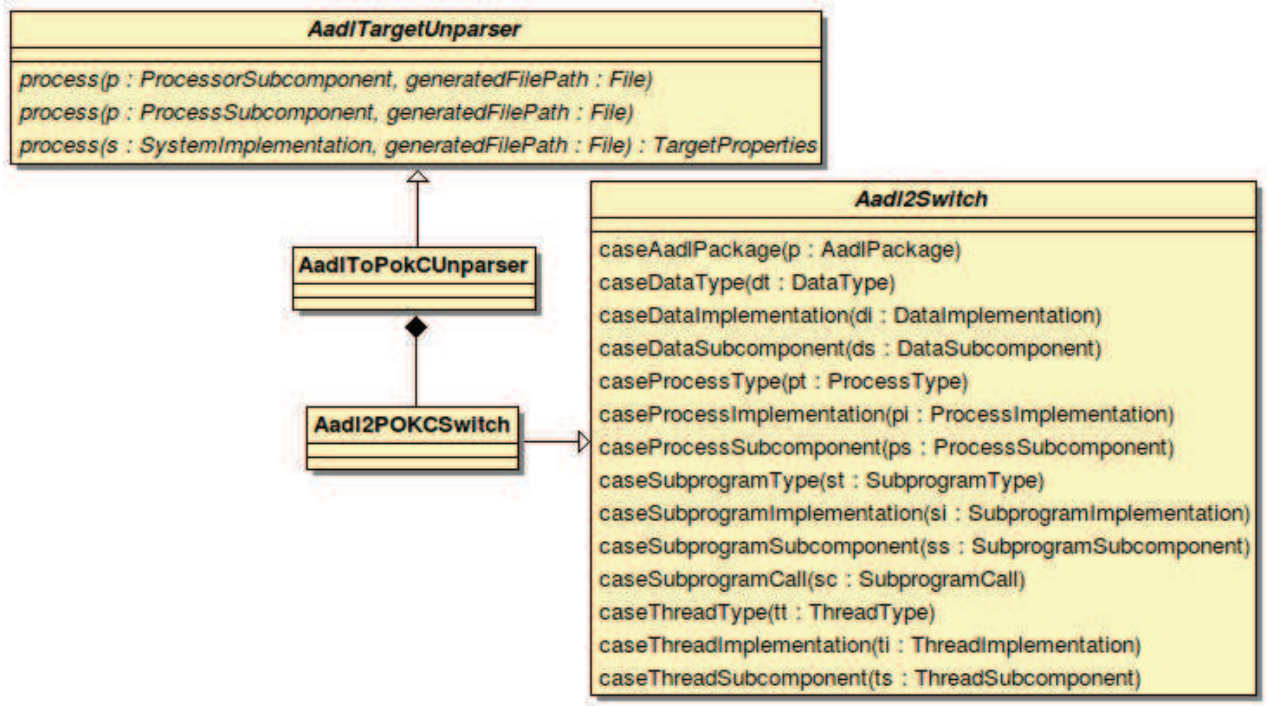

FIGURE 6.13 – Architecture des générateurs de code de RAMSES

Le code généré par RAMSES reprend l'organisation des fichiers générés fournie par Ocarina [49, 26]. Celle-ci est déterminée par les composants AADL. Les composants *system, processor et process* représentent respectivement le système global, un noeud du système (distribué), et une unité de compilation contenant des activités concurrentes. Le code généré va donc être organisé selon le schéma $< system > / < processor > / < process > /$. Le dernier répertoire, associé à un *process*, contient l'implémentation des tâches et des ressources associées à celui-ci. La classe *AadlTargetUnparser* est alors dérivée pour chaque plate-forme, par exemple *POKCUnparser* pour POK. De plus, ces classes s'aident de visiteurs dérivés de la classe *Aadl2Switch* qui spécifient le code à générer pour chaque type d'élément AADL parcouru. Cependant, contrairement à Ocarina, RAMSES factorise une grande partie de la génération de code commune à l'ensemble des plates-formes d'exécution grâce aux transformations de modèle indépendantes de la syntaxe du langage cible. Les classes héritant de *AadlTargetUnparser* ne se contentent ainsi que de traduire les éléments du modèle dans une syntaxe propre à la plate-forme visée.

Nous avons ainsi présenté le framework RAMSES que nous avons développé pour mettre en place le processus introduit dans les chapitres précédents. Nous avons décrit son architecture ainsi que son utilisation. Le prochain chapitre illustre l'application de RAMSES sur le raffinement des communications.