

République Tunisienne
Ministère de l'Enseignement Supérieur,
et de la Recherche Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax



Ecole Doctorale
Sciences et Technologies

Mémoire de *MASTERE*
Nouvelles Technologies
des
Systèmes Informatiques
Dédiés

N° d'ordre: 116/11

MEMOIRE

Présenté à

L'École Nationale d'Ingénieurs de Sfax

en vue de l'obtention du

MASTÈRE

Nouvelles Technologies des Systèmes Informatiques
Dédiés

Par

Alvine BOAYE BELLE

**Conception et développement d'un support d'exécution
pour systèmes TR²E dynamiquement reconfigurables**

Soutenu le 3 août 2011, devant la commission d'examen :

M. Khalil DRIRA

Président

Mme Ikram AMMOUS

Rapporteur

M. Mohamed JMAIEL

Encadreur

M. Bechir ZALILA

Invité

Mme Fatma KRICHEN

Invitée

Dédicaces

*À chacun des membres de ma famille qui chaque jour me manquent un peu plus :
mon père Guillaume, ma mère Rachel, ma sœur aînée Sandrine, mon petit frère
Stephane, ma petite sœur Marie-Flore, ma dernière petite sœur Paloma, ainsi que
mon oncle tonton René.*

À mon petit ami Arsène qui a toujours été présent à mes côtés.

Au père dominique qui m'a permis de cheminer dans la vie.

*À toutes les personnes que j'ai cotôquées en Tunisie et avec lesquelles j'ai tissé des
liens d'amitié.*

*À toutes les personnes que je n'ai pas citées, mais qui me sont néanmoins très
chères.*

Remerciements

Je tiens à remercier chaleureusement tous ceux qui m'ont permis de mener à bien mes travaux de recherche, et sans lesquels ce mémoire n'aurait pu voir le jour.

Aussi, mon attention se porte particulièrement vers mes encadreurs qui m'ont guidée tout au long de ce travail, m'apportant une aide précieuse grâce à leurs efforts d'encadrements. Je remercie ainsi le Professeur *Mohamed JMAIEL* pour m'avoir accueillie dans son équipe et pour avoir fait montre d'un grand soutien ainsi que d'une grande compréhension à mon égard. Je souhaite aussi remercier Mr *Bechir ZALILA* pour m'avoir guidée vers la rigueur, la ponctualité, et surtout la quête de l'excellence. Mes remerciements sont également dirigés à l'encontre de Mme *Fatma KRICHEN* qui m'a appris l'abnégation, m'a poussée à me dépasser chaque jour davantage et a contribué à apaiser mes inquiétudes sur le plan académique.

Ma gratitude se porte également vers tous mes camarades de classe qui se sont montrés plus que formidables envers moi, autant sur le plan académique que sur le plan amical. Ils m'ont permis de me sentir bien entourée pendant les moments où j'en avais le plus besoin.

Je me tourne enfin vers les membres de l'unité de *Recherche en Développement et Contrôle des Applications Distribuées* que je tiens à remercier vivement pour leur convivialité sans borne.

Table des matières

Introduction générale	1
1 État de l’art et problématiques	3
1.1 Introduction	3
1.2 Définitions	4
1.2.1 Composant	4
1.2.2 Modèle de composants	4
1.2.3 Application répartie	5
1.2.4 Support d’exécution	5
1.2.5 Système embarqué	6
1.2.6 Système temps réel	6
1.2.7 Reconfiguration	7
1.3 Les systèmes critiques	8
1.3.1 Description des systèmes critiques	9
1.3.1.1 LwCCM	9
1.3.1.2 AADL	10
1.3.1.3 MARTE	12
1.3.2 Quelques technologies liées aux systèmes critiques	13
1.3.2.1 RTSJ	13
1.3.2.2 SpaceWire	13
1.4 Les intergiciels adaptables	14
1.4.1 Les catégories d’intergiciels adaptables	14
1.4.1.1 Les intergiciels configurables	14
1.4.1.2 Les intergiciels génériques	15
1.4.1.3 Les intergiciels schizophrènes	15
1.4.2 Intergiciels adaptables pour systèmes TR ² E	17

1.4.2.1	DynamicTAO	17
1.4.2.2	CIAO	17
1.4.2.3	PolyORB_HI	18
1.4.2.4	FLARe	21
1.4.2.5	SwapCIAO	22
1.5	Critique des solutions existantes	23
1.5.1	La complexité des systèmes TR ² E	23
1.5.2	Le manque de fonctionnalités intergicielles	23
1.5.3	La dualité entre l'évolutivité et la disponibilité	23
1.5.4	L'incapacité à préserver des contraintes temps réel	24
1.5.5	La difficulté à préserver le caractère embarqué	24
1.5.6	Le risque d'incohérence du système reconfiguré	24
1.6	Conclusion	25
2	Intergiciel pour systèmes TR²E dynamiquement reconfigurables	27
2.1	Introduction	27
2.2	Concepts de base	27
2.2.1	Réflexivité	28
2.2.2	Cohérence	29
2.3	Description de l'approche	29
2.3.1	Adéquation aux systèmes TR ² E	29
2.3.2	Assurance de la réflexivité	30
2.3.3	Reconfiguration dynamique	30
2.3.4	Assurance de la cohérence	31
2.3.5	Recours aux services canoniques	32
2.3.5.1	Les services faiblement personnalisables	32
2.3.5.2	Les services fortement personnalisables	33
2.4	Conception de l'approche	35
2.4.1	Mécanismes de réflexivité	36
2.4.2	Mécanismes de cohérence	38
2.4.3	Mécanismes de reconfiguration dynamique	39
2.5	Respect des contraintes liées aux systèmes critiques	45
2.5.1	Respect de l'aspect temps réel	46
2.5.2	Respect du caractère embarqué	46

2.6	Conclusion	47
3	Mise en oeuvre de l'approche	49
3.1	Introduction	49
3.2	Cadre du travail	49
3.2.1	Modélisation des systèmes TR ² E dynamiques	49
3.2.2	Vérification formelle du modèle	50
3.2.3	La génération du code	51
3.3	Environnements et langages de programmation	51
3.3.1	Ocarina : description architecturale	51
3.3.2	Les langages de programmation	53
3.4	PolyORB_HI	54
3.4.1	Description de l'architecture de PolyORB_HI	54
3.4.1.1	Mise en oeuvre des services faiblement personnalisables	54
3.4.1.2	Mise en oeuvre des services fortement personnalisables	57
3.4.2	Gestion locale de la communication	59
3.5	RCES4RTES	60
3.5.1	Choix de l'implémentation	60
3.5.2	Description de l'architecture détaillée de RCES4RTES	61
3.5.2.1	Modification des structures de données	61
3.5.2.2	Mise en oeuvre du processus de reconfiguration dy- namique	62
3.6	Conclusion	69
4	Étude de cas	70
4.1	Introduction	70
4.2	Description de l'étude de cas	70
4.2.1	Propriétés communes aux composants d'un GPS	71
4.2.2	Les MétaModes d'un GPS	72
4.3	Architecture du système initial	72
4.3.1	Le terminal GPS	73
4.3.2	Le satellite	75
4.3.3	La base de contrôle	75
4.4	Architecture du système final	76

TABLE DES MATIÈRES

4.4.1	Le terminal GPS	76
4.4.2	Le satellite	79
4.4.3	La base de contrôle	80
4.5	Évaluation pratique	80
4.6	Conclusion	85
	Conclusion générale et perspectives	87
	Bibliographie	89
	Abbréviations	93

Introduction générale

Assurer le fonctionnement d'un système Temps Réel Réparti Embarqué (abrégé TR²E) est une tâche complexe. En effet, cela requiert la prise en charge de l'hétérogénéité des éléments qui composent ce système (machines, réseaux, langages de programmation, formats de données, etc). De plus, un tel système nécessite parfois de s'adapter ou même évoluer selon les besoins de son environnement d'exécution, et ce tout en restant disponible.

L'adaptation et l'évolution d'un tel système est rendue possible par le biais de la reconfiguration dynamique. Cette dernière consiste à introduire des modifications aussi bien structurelles que comportementales au sein d'un système en cours d'exécution. Une action de reconfiguration peut par exemple être effectuée en ajoutant un nouveau composant sur la plateforme d'exécution, ou en modifiant les propriétés d'un composant donné. L'introduction de mécanismes de reconfiguration dynamique dans les systèmes TR²E est possible en utilisant les supports d'exécution. Ceux-ci sont alors à mesure d'effectuer des modifications au sein de ces systèmes, tout en leur assurant de demeurer disponibles.

Cependant, les supports d'exécution qui adressent ces besoins souffrent souvent de plusieurs insuffisances, notamment au niveau du maintien de l'aspect temps réel du système reconfiguré. En effet, si l'exécution d'une opération de reconfiguration dynamique n'est pas déterministe, elle va mettre à mal le respect des échéances de ce système.

Par ailleurs, les intergiciels pour systèmes TR²E dynamiquement reconfigurables éprouvent souvent des difficultés à conserver le caractère embarqué de ceux-ci. En effet, ils ne prennent pas souvent en compte la criticité des ressources disponibles dans ces systèmes.

De plus, le maintien de la cohérence du système reconfiguré fait également défaut à nombreux de ces intergiciels. En effet, ces derniers effectuent des actions de recon-

figuration sans tenir compte du fait qu'elles peuvent causer des interférences avec les interactions se produisant entre les composants. Ces interférences sont pourtant susceptibles de provoquer des incohérences dans le traitement des requêtes en cours et même de rendre le système inutilisable.

Cela étant, notre rapport reposera donc sur quatre chapitres. Ainsi, le premier chapitre portera sur l'état de l'art et les problématiques qui y sont liées. Le deuxième chapitre sera quant à lui centré sur la conception d'un intergiciel pour systèmes TR²E dynamiquement reconfigurables. La mise en œuvre de notre approche fera l'objet du troisième chapitre. Le dernier chapitre se focalisera alors sur la présentation d'une étude de cas qui fera l'objet d'une évaluation pratique. La conclusion générale ainsi que les perspectives viendront clore ce rapport.

Chapitre 1

État de l'art et problématiques

1.1 Introduction

La complexité sans cesse croissante des systèmes TR²E¹ requiert la mise en place de voies et moyens permettant de réduire le temps ainsi que le coût de développement de ces systèmes. De plus, elle nécessite le recours à des mécanismes permettant à ceux-ci d'être modifiés en cours d'exécution. Ceci leur assure alors de s'adapter à leur contexte d'exécution et d'évoluer en fonction des exigences, tout en restant disponibles. Les supports d'exécution ont été introduits afin de répondre à ces attentes.

Néanmoins, les supports d'exécution pour systèmes TR²E présentent de nombreuses lacunes. Celles-ci sont notamment dues à la difficulté de préserver l'aspect temps réel, les contraintes liées au caractère embarqué, ainsi que le maintien de la cohérence de ces systèmes, une fois qu'ils ont été modifiés.

Aussi, puisqu'il est nécessaire de situer notre travail dans son contexte, ce chapitre s'attellera à rappeler quelques définitions, puis, à présenter des concepts et terminologies liés aux systèmes critiques. Ensuite, il passera en revue quelques intergiciels adaptables conçus pour les systèmes TR²E. Il s'achèvera enfin en exposant les limites des solutions existantes.

1. Temps Réel Répartis Embarqués

1.2 Définitions

Nous introduisons ici quelques notions dont les définitions nous serviront tout au long de ce mémoire.

1.2.1 Composant

Un composant est généralement défini en termes d'unités de calcul et de connecteurs [LT09], et peut être considéré comme étant une boîte noire [Lou10]. Plusieurs définitions ont été proposées pour définir le terme composant. Cependant, celle de Szyperski est l'une des plus utilisées [SGM02] :

“Un composant logiciel est une unité de composition qui a, par contrat, spécifié uniquement ses interfaces et ses dépendances explicites de contexte. Un composant logiciel peut être déployé indépendamment et est sujet à la composition par des tierces entités”.

Il est donc à noter qu'un composant peut être atomique ou imbriqué dans un ou plusieurs autres composants.

1.2.2 Modèle de composants

Un modèle de composants régit la façon dont les composants sont construits, assemblés et déployés, ainsi que la façon dont ils interagissent les uns avec les autres [LW07].

Selon les domaines d'application, plusieurs modèles de composants peuvent être recensés. Ainsi, les modèles industriels (EJB, CCM, COM, .NET) mettent l'accent sur les propriétés non fonctionnelles (sécurité, transactions, etc). Cependant, ils sont plutôt limités sur le plan architectural. D'autres modèles tels que UML 2.0 sont utilisés pour la conception d'architectures et ne possèdent pas de support d'exécution. Des modèles tels que SCA et OSGI peuvent quant à eux, être employés pour implémenter des architectures orientées services².

D'autres modèles sont davantage tournés vers la recherche (FRACTAL) ou le milieu scolaire et visent particulièrement l'analyse (structurelle et comportementale) des architectures. Ces modèles sont souvent associés à un langage dédié (Domain

2. Eng. SOA (Service Oriented Architectures)

Specific Language ou DSL) permettant la description architecturale des systèmes. Celui-ci est couramment appelé langage de description d'architecture³. Les concepts d'un tel langage sont les composants, les connecteurs qui permettent la modélisation des interactions entre ces composants, et la configuration.

1.2.3 Application répartie

Aussi appelée application distribuée, une application répartie est un ensemble de processus applicatifs situés sur des nœuds distants interconnectés grâce à un réseau de communication. Dans une telle application, les éléments sont amenés à coopérer afin d'effectuer des tâches communes.

La notion d'application répartie renvoie à celle de paradigme de répartition qui définit la façon dont les éléments d'une application distribuée doivent communiquer entre eux. Ainsi, on distingue les paradigmes de répartition par envoi de messages, appels de procédures distantes, objets distants, transactions, etc.

1.2.4 Support d'exécution

Parfois appelée intergiciel, l'expression support d'exécution devient de plus en plus récurrente de nos jours. Presque aussi vieille que l'informatique elle-même, cette expression remonte à l'époque où les programmeurs ont commencé à implémenter des fonctions. En effet, pour des raisons liées à la réutilisation de code, ils rassemblaient ces fonctions au sein de bibliothèques. Ainsi, au fur et à mesure que ces dernières se sont spécialisées, elles ont été mises dans des kits de développement (eng. Software Development Kits). Certains de ces kits peuvent être très spécialisés et utilisés avec le squelette d'une application : ce sont des supports d'exécution.

Un support d'exécution peut être défini comme étant une couche logicielle située entre le système d'exploitation et l'application afin d'assurer la communication entre plusieurs nœuds distants d'une même application.

La figure 1.1 montre l'emplacement d'un support d'exécution au sein d'un nœud d'une application distribuée.

Utilisé à des fins économiques (réduction du temps et du coût de fabrication des applications distribuées), un support d'exécution peut implanter toutes les fonctionnalités nécessaires à un système d'exploitation. Il offre ainsi l'avantage de masquer à

3. Eng. Architecture Description Language abrégé ADL

l'utilisateur l'hétérogénéité des éléments utilisés par l'application (systèmes, formats de données, langages de programmation, réseaux, etc). Ceci assure la portabilité et l'interopérabilité au sein d'une application distribuée donnée.

1.2.5 Système embarqué

Cette expression désigne un système informatique conçu pour réaliser des fonctionnalités dédiées à un domaine donné.

Un système embarqué peut comporter un système d'exploitation. Mais généralement, ce dernier est assez simple de façon à tenir sur un seul programme. Des périphériques tels que le clavier, l'écran, des dispositifs de stockage et autres peuvent être absents de sa structure. Les systèmes embarqués sont généralement regroupés afin de former de plus larges systèmes. La fiabilité et la capacité à fonctionner avec des ressources limitées en constituent les principales caractéristiques.

1.2.6 Système temps réel

Un système temps réel est un système dont le fonctionnement correct dépend aussi bien de l'exactitude des traitements effectués que des délais impartis à la réalisation de ces traitements [Hug05].

On distingue deux principaux types de temps réel :

- **le temps réel dur** : le respect d'une échéance temporelle est vital pour l'application. C'est le cas d'un avion.

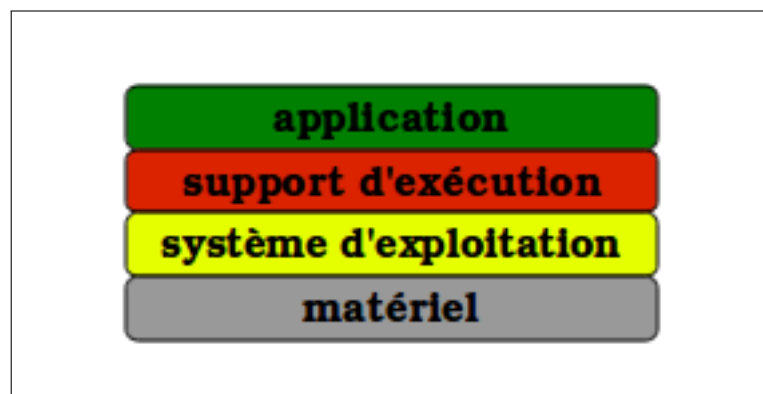


FIGURE 1.1 – Position du support d'exécution

- **le temps réel mou** : le non-respect d'une échéance temporelle n'est pas dramatique pour l'application. C'est le cas d'un distributeur de billets.

1.2.7 Reconfiguration

Le mot *reconfiguration* renvoie à la modification d'une configuration donnée. Le terme configuration désignant lui-même l'environnement courant d'un composant [Sch04]. Pour des raisons d'adaptation et d'évolutivité, un système peut avoir besoin de modifier son architecture et même son implémentation. Pour ce faire, il pourra alors recourir à la reconfiguration.

La reconfiguration peut être structurelle ou comportementale. En effet, ce premier aspect de la reconfiguration consiste à modifier l'architecture d'un système. Ceci peut être fait en changeant par exemple l'emplacement d'une entité. Le second aspect consiste quant à lui à modifier le comportement des entités qui se trouvent sur la plateforme d'exécution. La modification des propriétés d'une entité en est un exemple. D'après ces deux aspects de la reconfiguration, les modifications dans un système peuvent notamment intervenir à travers :

- la migration, l'ajout ou la suppression d'entités,
- l'ajout ou la suppression de connexions entre entités,
- le remplacement d'entités,
- la modification des propriétés des entités,
- la modification du code d'une entité donnée.

La reconfiguration peut être appliquée sur un système en état de marche ou sur un système dont l'exécution a été stoppée. Deux types de reconfiguration sont alors distingués :

- **la reconfiguration statique** : elle consiste à modifier un système lorsque celui-ci est stoppé. Cela peut comporter certains risques tels que l'obtention d'un système incohérent. Ceci est à même de provoquer des catastrophes.
- **la reconfiguration dynamique** : elle renvoie à la modification d'un système alors que celui-ci est en cours d'exécution. Elle s'applique surtout aux systèmes qui ne peuvent être stoppés afin d'être modifiés. Elle permet ainsi de préserver une certaine disponibilité du système grâce à une réduction de l'interruption de service.

La figure 1.2 illustre une application formée par trois nœuds, à savoir : le nœud 1, le nœud 2 et le nœud 3. Le nœud 1 possède deux composants désignés par C1 et C2. Le nœud 2 est constitué par les composants C3, C4 et C5. Le nœud 3 est quant à lui formé par les composants C6, C7 et C8. Ces trois nœuds dont les connexions entre composants sont représentées par des flèches, forment ainsi une configuration donnée. Supposons que cette dernière peut être modifiée à l'aide d'actions de reconfiguration.

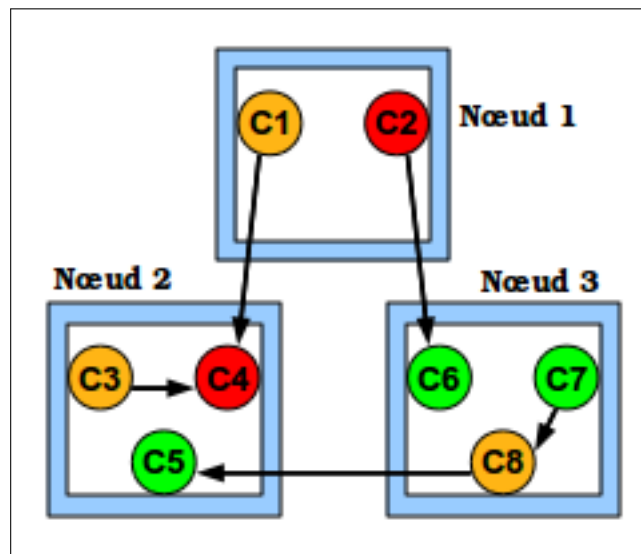


FIGURE 1.2 – Configuration de l'application avant les actions de reconfiguration

Ainsi, nous pouvons par exemple supprimer la connexion entre C7 et C8, ainsi que celle entre C8 et C5. Puis, nous pouvons supprimer le composant C8 et enfin rajouter une connexion entre C3 et C5. Nous obtenons ainsi la nouvelle configuration illustrée par la figure 1.3.

1.3 Les systèmes critiques

Caractérisés par leur sécurité et leur sûreté de fonctionnement, les systèmes critiques nécessitent le recours à des formalismes et technologies permettant de se conformer aux exigences auxquelles ils sont assujettis.

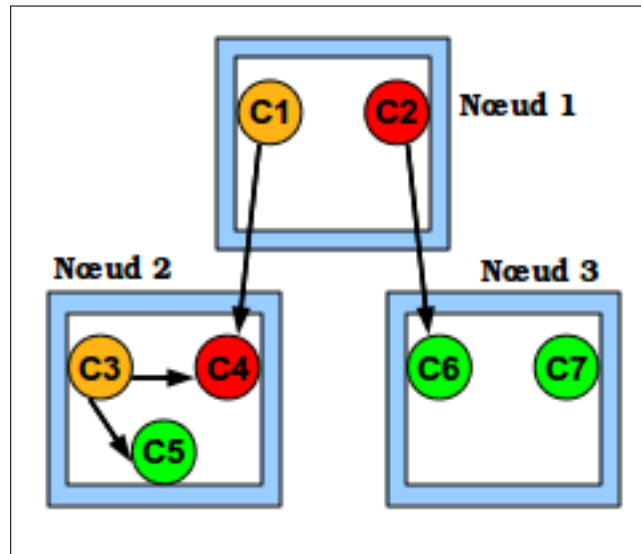


FIGURE 1.3 – Configuration de l'application après les actions de reconfiguration

1.3.1 Description des systèmes critiques

Plusieurs formalismes ont été proposés pour décrire les systèmes critiques. Dans la suite de cette section, nous en présentons quelques-uns.

1.3.1.1 LwCCM

LwCCM (LightWeight CCM) est une spécification de l'OMG⁴ qui standardise le développement, la configuration et le déploiement d'applications développées à base de composants [Bal09]. LwCCM utilise le modèle DOC (Distributed Object Computing) comme architecture sous-jacente, de façon à ce que les applications soient indépendantes des langages de programmation et des plateformes de développement.

Définition 1.1. *Un composant LwCCM est une entité de programmation qui exporte un ensemble d'interfaces utilisables aussi bien par les clients classiques de l'intergiciel que par d'autres composants.*

Les composants peuvent interagir selon deux modes à savoir : requête/réponse et envoi d'évènements asynchrone, avec un mécanisme publier/souscrire⁵.

4. Object Management Group

5. Publish/subscribe

Quelques concepts de LwCCM

- **la facette** est une interface qui fournit un point de vue sur un composant. Elle accepte des invocations issues des autres composants,
- **le réceptacle** indique une dépendance vis-à-vis de l'interface d'une méthode fournie par un autre composant,
- **l'évènement (sources/puits)** indique l'aptitude à échanger des messages avec un ou plusieurs composants,
- **les attributs** sont des propriétés associées aux composants et généralement utilisées lors de la configuration,
- **les fabriques**⁶ permettent de rendre transparents aux clients les détails des stratégies de création des composants, ainsi que les requêtes permettant de localiser les composants.

Discussion

Pour définir les interfaces, LwCCM utilise une extension de IDL appelée IDL3+. Cette dernière présente notamment l'avantage de fournir un support adéquat pour la construction de systèmes TR²E (d'entreprises) composés à base d'architectures hétérogènes. Ceci est effectué tout en minimisant le temps et les efforts consacrés au développement [Hil11].

LwCCM [Loi08] vise à réduire l'empreinte mémoire des applications CCM grâce au retrait des fonctionnalités qui ne sont pas requises dans les systèmes embarqués. Cependant, LwCCM ne permet pas de séparer clairement le code implantant les composants, du code utilisé pour décrire les interactions entre ces composants. En outre, il permet uniquement de reconfigurer statiquement les systèmes.

1.3.1.2 AADL

L'Architecture Analysis and Design Language abrégé AADL est un ADL dérivé de MetaH [BEJV93]. Standard international, AADL est publié par la SAE (Society of Automotive Engineers). Sa première version AADL 1.0 [SAE04] a été publiée en octobre 2004. Une version plus récente nommée AADLV2 [SAE09] a été publiée en janvier 2009 et complétée en janvier 2011 [SAE11].

6. Eng. homes

Utilisé à des fins de conception et d'analyse des systèmes temps réel critiques, AADL repose sur la notion de composant. Ainsi, tout système modélisé à l'aide d'AADL est décrit par un ensemble de composants interconnectés.

Définition 1.2. *Un composant AADL est une entité matérielle ou logicielle qui possède un type décrivant une interface lui permettant d'interagir avec les autres composants. Il peut avoir zéro ou plusieurs implantations.*

AADL peut donc être utilisé pour représenter aussi bien la partie matérielle que la partie logicielle d'un système. En plus de la représentation architecturale du système, il permet aussi de spécifier les propriétés non fonctionnelles (échéances temporelles, empreinte mémoire, latence d'un bus, etc), et de décrire le comportement du système.

Les catégories de composants AADL

Dans AADL, il existe trois familles de composants, à savoir :

- **les composants matériels** permettant de spécifier l'architecture matérielle du système. Ils sont classés en quatre catégories : les processeurs modélisés à l'aide du composant **processor**, les mémoires décrites à l'aide du composant **memory**, les bus modélisés par le composant **bus**, et enfin les périphériques qui sont décrits grâce au composant **device**,
- **les composants logiciels** utilisés afin de décrire les entités logicielles du système. Ils sont au nombre de cinq, à savoir : les processus lourds décrits par le composant **process**, les processus légers modélisés par le composant **thread**, les groupes de processus légers décrits à l'aide du composant **thread group**, les sous-programmes modélisés par le composant **subprogram** et les données qui sont décrites par le composant **data**,
- **les composants hybrides** utilisés quant à eux pour hiérarchiser un modèle AADL. Ne correspondant pas à une entité concrète, ils sont modélisés grâce au composant **system**.

Discussion

AADL est adapté aux systèmes TR²E critiques, dans la mesure où il propose différentes familles de composants permettant de spécifier de façon précise et détaillée les entités utilisées pour modéliser les systèmes. Par ailleurs, AADL présente également l'avantage d'être extensible, et ce grâce à l'utilisation d'annexes qui offrent

la possibilité d'encapsuler les spécifications étendues et de les joindre à celles qui sont standardisées. En outre, AADL rend possible l'incorporation d'informations permettant de déployer et de configurer les systèmes. Celles-ci peuvent permettre de produire automatiquement des applications TR²E grâce aux outils appropriés.

Cependant, AADL est un langage que l'on pourrait qualifier de « bas-niveau ». En effet, les composants qu'il permet de modéliser sont proches des abstractions fournies par le système d'exploitation : threads, processus, etc. En outre, les systèmes modélisés à l'aide d'AADL sont reconfigurés dynamiquement à l'aide de configurations prédéfinies de façon statiques. Cela restreint par conséquent le nombre de configurations possibles [KHZC10].

1.3.1.3 MARTE

Le profil MARTE (Modeling and Analysis of Real-Time and Embedded systems [OMG07]) est une extension d'UML utilisée dans le cadre du développement et de l'analyse des applications temps réel embarquées. Ayant pour objectif d'apporter un support aux techniques existantes, MARTE simplifie les annotations de modèles grâce à des informations requises pour réaliser des analyses avec des outils appropriés. Par ailleurs, MARTE permet de spécifier les entités matérielles et logicielles des systèmes temps réel critiques. Il assure également l'interopérabilité entre les outils utilisés lors des étapes d'analyse et de développement des systèmes.

Notion de composant MARTE

Un composant MARTE est une entité autonome pouvant posséder des données et un comportement. Afin de spécifier ses interactions avec l'extérieur, un composant utilise des propriétés et des interfaces.

Discussion

MARTE permet de spécifier les propriétés non-fonctionnelles des systèmes modélisés (fréquence du processeur, échéances temporelles, etc). Malheureusement, il ne supporte pas les systèmes distribués. Par ailleurs, à l'image d'AADL, MARTE décrit les actions de reconfiguration dynamique à l'aide d'un ensemble de configurations et de transitions entre elles. Ces configurations étant prédéfinies, elles limitent

les possibilités de reconfiguration. En outre, il existe peu d'outils implantant le profil MARTE, ce qui réduit son usage à des fins d'analyse.

1.3.2 Quelques technologies liées aux systèmes critiques

Le développement des systèmes critiques nécessite l'emploi de technologies qui vont leur permettre de respecter les exigences auxquelles ils sont liés. Dans cette section, nous présentons quelques-unes de ces technologies.

1.3.2.1 RTSJ

Le ramasse-miettes⁷ s'exécute de façon imprévisible, ce qui engendre un certain indéterminisme dans la machine virtuelle Java. Le RTSJ ou Real Time Specification for Java a été introduit pour y remédier. Ce dernier est une technologie qui a été formalisée en juin 2000 [BB09]. Il définit la façon dont le Java doit se comporter en présence d'un environnement temps réel. Il propose ainsi des fonctionnalités adaptées aux systèmes temps réel.

Discussion

Offrant de nettes améliorations vis-à-vis du Java classique, le RTSJ permet notamment l'ordonnancement temps réel des threads, ainsi que la gestion déterministe de la mémoire, et cela grâce à l'introduction de nouvelles zones mémoires. Il permet également d'obtenir un code portable.

1.3.2.2 SpaceWire

Utilisé entre autres par la NASA et l'ESA, SpaceWire est un standard qui adresse la gestion des données utiles et des informations de contrôle au sein des engins spatiaux. Pour ce faire, il assure l'envoi fiable de données à grande vitesse (entre 2 Mbps et 400 Mbps). Il est défini par le standard ECSS-E50-12A de l'ECSS (European Cooperation for Space Standardization) et est adapté aux contraintes spatiales [ECS03]

Les équipements utilisant SpaceWire sont interconnectés via des liens série, full-duplex, à haute vitesse ou encore bidirectionnels.

7. Eng. garbage collector

SpaceWire offre des garanties de QoS et de respect des échéances temporelles. À cet effet, il ajoute une couche supplémentaire entre le réseau et les applications. Appelée *SpaceWire-RT*, cette couche rend le réseau fiable en utilisant un mécanisme de redondance proposant des voies alternatives par lesquelles les paquets seront acheminés. Par ailleurs, SpaceWire-RT assure l'exactitude temporelle en recourant à un multiplexage temporel de la bande passante, afin de veiller à ce que les paquets soient délivrés dans les délais [FP09].

Discussion

À la différence de plusieurs autres couches de transport, SpaceWire est appropriée pour envoyer des informations de contrôle ainsi que des informations utiles, et ce de façon fiable et déterministe.

1.4 Les intergiciels adaptables

Afin d'apporter des solutions aux problèmes d'adaptabilité dans les systèmes temps réel embarqués, les intergiciels adaptables ont été proposés.

1.4.1 Les catégories d'intergiciels adaptables

Les intergiciels adaptables se répartissent en trois grandes catégories qui régissent plus ou moins les mécanismes de répartition qu'ils implantent.

1.4.1.1 Les intergiciels configurables

Définition 1.3. *Un intergiciel configurable permet à une application de sélectionner dans une bibliothèque prédéfinie, les composants intergiciels ainsi que les politiques qui adressent le mieux ses besoins.*

Les intergiciels configurables permettent souvent de réduire l'espace mémoire occupé par l'application, en ne choisissant que les composants qui lui sont nécessaires. En outre, ils favorisent le respect des contraintes applicatives grâce à la configuration des composants choisis. Cependant, ils sont dédiés à un mode de répartition et à une spécification donnée, ce qui limite la personnalisation aux normes supportées par les standards utilisés.

Des intergiciels tels que TAO, CIAO, FLARe et SwapCIAO font partie des intergiciels configurables.

L’intergiciel TAO : développé à partir de la spécification Real-Time CORBA, TAO (The ACE ORB) est caractérisé par l’emploi des patrons de conception et s’appuie sur les concepts de l’orienté objet. Il offre des garanties de QoS grâce aux fonctionnalités d’ordonnancement qu’il emploie, ainsi qu’à la limitation de l’inversion de priorités. Il permet également le contrôle des ressources disponibles et empêche qu’un client donné ne monopolise les services offerts par le serveur.

Flexible, performant et portable grâce à l’usage du canevas de communication ACE, TAO est néanmoins difficilement reconfigurable en cours d’exécution. Par ailleurs, le fait qu’il repose sur le Real-Time CORBA le rend incompatible avec certaines règles de codage utilisées par les systèmes temps réel embarqués.

1.4.1.2 Les intergiciels génériques

Définition 1.4. *Les intergiciels génériques étendent le concept d’intergiciels configurables. Ils sont formés par un ensemble d’entités indépendantes de tout modèle de répartition et qui offrent des mécanismes pour la répartition (interfaces et services à étendre).*

Les intergiciels génériques offrent l’avantage de pouvoir être utilisés avec différents modèles de répartition. Cependant, ils sont peu utilisés car ils nécessitent notamment une adaptation qui est à la charge du développeur, et de plus, leur taux de réutilisation est très faible. À titre d’exemple, nous pouvons citer : Jonathan et QuarterWare[Hug05].

1.4.1.3 Les intergiciels schizophrènes

Définition 1.5. *Les intergiciels schizophrènes apportent une réponse aux besoins d’interopérabilité entre intergiciels hétérogènes. Étendant l’architecture des intergiciels génériques, ils permettent de faire collaborer plusieurs paradigmes de répartition au sein d’une même instance.*

Les intergiciels schizophrènes proposent une nouvelle architecture intergicielle reposant sur neuf services appelés “services canoniques”. Ces services sont pour la plupart personnalisables en fonction des besoins de l’application, ce qui leur permet d’être dédiés à cette dernière. Ces services sont :

- le service **adressage** chargé de gérer les références des entités externes à l’intergiciel,

- le service **liaison** permettant la construction et la sélection des ressources de communication,
- le service **typage** chargé de gérer les types de données transmis,
- le service **représentation** fournissant les mécanismes d’emballage et de déballage de données échangées,
- le service **interaction** permettant la gestion du mode d’interaction entre les nœuds,
- le service **protocole** prenant en charge des étapes nécessaires à la transmission d’une requête,
- le service **transport** permettant l’ouverture, la fermeture et l’utilisation des canaux de communication,
- le service **activation** chargé de l’activation des entités nécessaires au traitement d’une requête,
- le service **exécution** permettant de réaliser les actions requises pour au traitement d’une requête reçue.

Les services canoniques sont implantés indépendamment de tout standard de répartition au sein d’une couche neutre. Celle-ci est chargée d’assurer le dialogue entre les personnalités applicatives et protocolaires. Les personnalités applicatives désignent l’endroit où sont implantés les aspects applicatifs d’un paradigme de répartition. Les personnalités protocolaires renvoient quant à elles à l’emplacement où sont implantés les aspects du paradigme de répartition qui se rapportent à la communication.

Les intergiciels schizophrènes présentent l’avantage d’être configurables, génériques et interopérables. Par ailleurs, ils permettent de séparer les préoccupations au sein d’un intergiciel.

L’intergiciel PolyORB

PolyORB est une implantation d’un intergiciel schizophrène. Proposant trois personnalités protocolaires (GIOP, SOAP et SRP) et cinq personnalités applicatives (CORBA, DSA, AWS, MOMA, et DDS), il permet de paramétrer différents aspects d’une application distribuée, à l’image du choix de la politique de QoS.

PolyORB est capable de supporter plusieurs paradigmes de répartition du fait de son caractère schizophrène qui hérite des propriétés des intergiciels génériques. Cela dit, le fait que sa configuration soit dynamique, le rend inadapté aux systèmes TR²E

critiques. Par ailleurs, l'absence de mécanismes de reconfiguration dans PolyORB constitue un inconvénient majeur.

1.4.2 Intergiciels adaptables pour systèmes TR²E

Plusieurs intergiciels adaptables ont été proposés pour adresser les besoins des systèmes TR²E, notamment en termes d'interopérabilité et de portabilité. Nous en présenterons quelques-uns dans cette section.

1.4.2.1 DynamicTAO

Développé à l'université de l'Illinois entre 1998 et 2000 [Kon02], DynamicTAO étend les mécanismes de l'intergiciel TAO et repose sur une ORB conforme à CORBA. Flexible, DynamicTAO implante des mécanismes pour gérer la concurrence, la sécurité et superviser l'exécution du système [KRL⁺00].

DynamicTAO supporte les mécanismes de reconfiguration dynamique. Cette dernière peut correspondre à un déplacement des composants sur la plateforme d'exécution, ou au chargement/déchargement des composants du support d'exécution. Afin de faciliter le processus de reconfiguration dynamique, les composants du système sont constitués en des bibliothèques qui sont chargées en cours d'exécution et qui sont utilisées si nécessaire.

Discussion

DynamicTAO est réflexif et reconfigurable, ce qui lui permet de s'adapter à l'environnement d'exécution du système. Cependant, cette dernière caractéristique engendre quelques difficultés. En effet, DynamicTAO arrive difficilement à gérer la durée de la reconfiguration ainsi que l'état du système en cours de reconfiguration. De plus, comme il est peu adapté à l'embarqué, il est rarement utilisé de nos jours.

1.4.2.2 CIAO

Abrégé CIAO, le Component Integrated ACE ORB est une implémentation libre du modèle de composants LwCCM et des spécifications du Real-Time CORBA. Il fournit des mécanismes pour la spécification, l'implémentation, l'assemblage et le déploiement des composants [SDG⁺07].

CIAO offre également une robuste QoS grâce au découplage des activités de déploiement et de configuration, de l'implémentation des composants. Par ailleurs, il a recours aux techniques de l'orienté aspects afin de supporter la séparation et la composition des aspects temps réel d'avec les préoccupations propres à la configuration [TdOM05].

Discussion

Adapté aux systèmes temps réel embarqués, CIAO est flexible et permet de simplifier et d'automatiser le (re)déploiement et la (re)configuration. Le support de mécanismes de reconfiguration y est évoqué. Cela dit, ni la manière d'implanter ces mécanismes, encore moins la nature des reconfigurations supportées ne sont spécifiées. Par ailleurs, CIAO ne prend pas en charge l'ordonnancement des tâches a périodiques ainsi que l'équilibrage de charges. De plus, le recours à XML à des fins de configuration lui est coûteux en ressources.

1.4.2.3 PolyORB_HI

PolyORB_HI est un intergiciel inspiré de l'architecture de PolyORB. Sa structure repose ainsi sur les services intergiciels canoniques. Il est conforme au Profil Ravenscar. Ce dernier restreint l'usage de certaines constructions de langages de programmation (ADA, C, etc). Cette restriction permet de faire en sorte que l'inversion des priorités des tâches soit effectuée dans un délai donné, et que les risques d'interblocage ainsi que de famines soient écartés.

En outre, PolyORB_HI est un intergiciel dédié à l'application avec laquelle il interagit, dans la mesure où il intègre essentiellement les composantes dont elle a besoin. À cet effet, les composants intergiciels qui sont communs à la majorité des applications sont écrits à la main. Ceux qui sont propres à une application donnée sont générés automatiquement, grâce à une description de l'architecture ainsi que des besoins de l'application. L'adéquation de PolyORB_HI aux systèmes TR²E est rendue possible grâce au support des constructions AADL. Ceci permet un calcul et une allocation automatiques des ressources de l'application.

Personnalisation des services canoniques

Afin d'être dédié à une application donnée, PolyORB_HI divise certains services canoniques en deux autres services. Il répartit ensuite l'ensemble obtenu en deux familles : les services faiblement personnalisables et les services fortement personnalisables [Zal08]. La première famille comporte les services qui sont communs à chaque application et qui constituent l'**intergiciel minimal**. La seconde famille englobe les services qui sont paramétrables en fonction des besoins de l'application cible. Le tableau 1.1 illustre la répartition des services canoniques selon leur degré de personnalisation.

Services canoniques	Nouveaux services	Classement
Adressage	Adressage	Fortement personnalisable
Liaison	Liaison	Fortement personnalisable
Activation	Activation	Fortement personnalisable
Typage	Typage	Fortement personnalisable
Exécution	Parallélisme Exécution	Faiblement personnalisable Fortement personnalisable
Représentation	Représentation élémentaire Représentation avancée	Faiblement personnalisable Fortement personnalisable
Interaction	Interrogation Interaction	Faiblement personnalisable Fortement personnalisable
Protocole	Protocole	Faiblement personnalisable
Transport	Couche basse de transport Couche haute de transport	Faiblement personnalisable Fortement personnalisable

TABLE 1.1 – Nouvelle répartition des services canoniques [Zal08]

Les nouveaux services obtenus lors de cette répartition seront davantage explicités à la section 2.3.5.

Support des constructions AADL

Afin d'être à mesure d'assurer la production de systèmes TR²E, PolyORB_HI prend en charge les constructions AADL suivantes :

- **Les processus légers** : à ce niveau, PolyORB_HI fournit les mécanismes permettant de créer les diverses tâches supportées par les systèmes critiques. Ainsi, il gère les types de tâches suivants :

- **Les tâches périodiques** : elles se répètent à des intervalles réguliers, suite à un évènement temporel qui survient périodiquement,
 - **Les tâches sporadiques** : elles sont cycliques et sont déclenchées par la réception d'un évènement externe, après une durée minimale fixée,
 - **Les tâches hybrides** : elles combinent les caractéristiques des deux tâches précédentes. Leur implantation nécessite le recours à une tâche chargée de leur notifier que leur période est arrivée.
- **Les éléments d'interface** : ce sont des mécanismes permettant aux éléments de l'application distribuée et au code de l'utilisateur d'effectuer de façon déterministe les opérations de lecture et d'écriture au niveau des interfaces des processus légers,
- **Les données partagées** : à ce niveau, PolyORB_HI assure la protection des données partagées entre les éléments applicatifs, si celle-ci n'est pas déjà prise en charge par le langage de programmation. À effet, cet intergiciel propose des fonctionnalités permettant de verrouiller ou de déverrouiller l'accès aux données partagées, de façon à ce qu'elles ne puissent être accédées que par des accesseurs.

Faible empreinte mémoire

Au niveau de chaque noeud, PolyORB_HI inclut uniquement les composants intergiciels nécessaires, ce qui réduit considérablement sa taille. Cette réduction est accentuée par le fait que l'intergiciel inclut uniquement les composants faiblement personnalisables. Les composants produits automatiquement étant quant à eux fortement personnalisés selon les caractéristiques des nœuds qui les abritent.

Configuration statique et automatique

Les composants intergiciels sont configurés statiquement : ils sont paramétrés au moment où l'application est compilée, ou lorsqu'elle est initialisée. Par contre, les éléments requis pour déployer et configurer l'application, sont produits automatiquement, à partir d'une analyse du modèle d'une application donnée.

Discussion

PolyORB_HI présente l'avantage d'être portable, sûr et déterministe. De plus, sa très faible empreinte mémoire fait de lui un candidat idéal pour l'embarqué. Cependant, dans cet intergiciel, les composants configurés statiquement doivent être compilés avec ceux qui sont produits automatiquement. Ceci le contraint à être distribué sous forme de code source. Par ailleurs, il n'implante pas les mécanismes permettant la reconfiguration dynamique des systèmes TR²E.

1.4.2.4 FLARe

Le Fault-tolerant Load-aware and Adaptive middlewaRe abrégé FLARe est un intergiciel qui étend TAO. Adapté aux applications réparties temps réel molles, il permet d'ajuster dynamiquement les cibles tolérantes aux fautes en réponse aux variations de la charge du système et des ressources disponibles.

Afin de fournir des mécanismes de tolérance aux fautes, FLARe utilise la *replication passive* qui consiste à faire traiter toutes les requêtes par un seul serveur appelé "primaire". Lorsque ce dernier tombe en panne, il est remplacé par un autre serveur [Bal07]. L'architecture de FLARe est décrite à l'aide des quatre composantes suivantes :

- **Le MRM (Middleware Replication Manager)** chargé de fournir au serveur des renseignements sur les processeurs ainsi que sur les processus,
- **Les moniteurs** permettant de sonder la vivacité des processus hébergeant les serveurs, ainsi que le taux utilisation du processeur,
- **L'agent de transfert d'état** permettant aux serveurs de fournir des informations sur les changements d'état de l'application,
- **Le gestionnaire du client résistant aux pannes** qui, au lieu de propager l'exception soulevée en cas de panne, la capture et redirige le client vers une cible appropriée.

FLARe offre des mécanismes de reconfiguration qui consistent à rediriger des clients dans les cas de pannes dues par exemple à la surcharge d'un processeur [BSTG⁺09].

Discussion

FLARe implante des mécanismes permettant de gérer les pannes et les surcharges de façon transparente aux clients. Cependant, il n'est pas adapté au temps réel dur. De plus, il ne gère que les tâches périodiques, et sa maintenance est une opération complexe.

1.4.2.5 SwapCIAO

Extension de CIAO, SwapCIAO est un intergiciel qui supporte la mise à jour dynamique des implantations de composants, grâce aux extensions fournies par le modèle de composants LwCCM [Bal09]. SwapCIAO est ainsi caractérisé par :

- **des mises à jour continues et consistantes des clients**, en s'assurant au préalable que :
 - les interactions entre le composant et le client soient achevées,
 - les nouvelles invocations du client soient bloquées jusqu'à la fin de la mise à jour du composant,
- **la transparence des mises à jour** garantie, à l'issue de la mise à jour, par la redirection des invocations du client vers la nouvelle implémentation du composant, et ce, sans que le client ne s'en aperçoive,
- **la reconnexion des composants** effectuée en restaurant, après la mise à jour du composant, les connexions qui ont été sauvegardées dans la mémoire cache, avant le début de la mise à jour. Ces connexions sont au préalable stockées dans des descripteurs XML lors du déploiement de l'application.

Au sein de SwapCIAO, le processus global de reconfiguration dynamique qui consiste en une mise à jour, peut être décrit grâce aux étapes suivantes :

1. désactivation de l'implémentation du composant,
2. retrait de l'implémentation du composant de la plateforme d'exécution,
3. modification de l'implémentation du composant.

Discussion

Flexible, performant et adapté aux systèmes TR²E dynamiques, SwapCIAO offre des garanties de QoS. Il est toutefois très difficile à maintenir. À cet effet, il n'est pas pour l'instant sujet à des développements futurs de la part de ses concepteurs.

1.5 Critique des solutions existantes

Les éléments étudiés dans ce chapitre prouvent qu'il est possible de concevoir des supports d'exécution pour systèmes TR²E dynamiquement reconfigurables. Néanmoins, les solutions rencontrées sont plus ou moins confrontées aux problèmes suivants :

1.5.1 La complexité des systèmes TR²E

Les systèmes TR²E sont de plus en plus présents dans notre quotidien, et tendent de ce fait à se diversifier. Couvrant actuellement une large gamme de produits, ils peuvent varier du simple micro-contrôleur à de vastes systèmes distribués.

Constituant de véritables enjeux économiques et suscitant de nombreux intérêts au sein de la communauté scientifique, ces systèmes incluent plusieurs composantes. Celles-ci peuvent être : les langages de programmation, les routines, les formats de données, les protocoles, les réseaux, etc. Aussi, la prise en charge de tous ces éléments hétérogènes constitue un challenge de taille.

1.5.2 Le manque de fonctionnalités intergicielles

Bien souvent, les intergiciels n'implémentent que partiellement les mécanismes permettant de garantir la sûreté de fonctionnement, ainsi que le déterminisme lors de l'exécution. Ceci peut notamment s'expliquer par :

- la mise en place de mécanismes d'échange de données non déterministes,
- le recours à des mesures de sécurité qui ne reflètent pas les risques d'accidents qu'un tel système est susceptible d'engendrer.
- l'assurance de la QoS qui laisse parfois à désirer, ou qui est même laissée à la charge de l'utilisateur.

1.5.3 La dualité entre l'évolutivité et la disponibilité

L'évolution d'un système peut passer par l'introduction de modifications permettant sa mise à jour. Cependant, ces modifications peuvent engendrer des interruptions de service, mettant ainsi à mal la disponibilité de ce système. En effet, ces interruptions de services vont empêcher les clients d'interagir avec le système pendant le laps de temps au cours duquel il subit des modifications.

Il apparaît donc nécessaire de trouver un bon compromis entre la disponibilité et l'évolutivité du système, afin d'optimiser son usage.

1.5.4 L'incapacité à préserver des contraintes temps réel

Dans les systèmes TR²E, le respect des contraintes temporelles est un facteur clé pour la fiabilité. Ces systèmes requièrent donc que la reconfiguration (dynamique) soit également assujettie au respect des contraintes temporelles.

Cependant, la reconfiguration peut nécessiter de transférer l'état d'un composant, de déplacer un composant vers un autre emplacement de la plateforme d'exécution, ou même de bloquer d'autres composants en attente du composant en cours de reconfiguration. Cela requiert alors que les temps nécessaires pour effectuer ces opérations soient aussi déterministes. Malheureusement, cette exigence est souvent difficile à respecter car elle fait intervenir plusieurs paramètres (vitesse d'exécution sur le processeur, temps de transport des données sur le réseau, etc).

1.5.5 La difficulté à préserver le caractère embarqué

La mise en place de mécanismes permettant d'introduire des modifications au sein d'un système TR²E en cours d'exécution doit tenir compte des ressources disponibles dans le système. Celles-ci peuvent être quantifiées en termes d'espace mémoire, de vitesse de processeur, de largeur de la bande passante, etc. Malheureusement, les mécanismes permettant de supporter ces contraintes ne sont pas toujours implantés dans les intergiciels intégrés à ces systèmes.

1.5.6 Le risque d'incohérence du système reconfiguré

L'adaptation et/ou l'évolution (dynamiques) d'un système ne doivent pas être effectuées au détriment de sa fiabilité. Cette dernière indique l'aptitude de ce système à exécuter correctement les services attendus. En effet, les actions de reconfiguration d'un système en cours d'exécution peuvent interférer avec les interactions qui se produisent entre les composants. Cela comporte notamment le risque de mener le système vers un état incohérent : ceci peut le rendre inutilisable. Néanmoins, plusieurs intergiciels pour systèmes TR²E ne tiennent pas compte de cet aspect lors de la reconfiguration dynamique.

1.6 Conclusion

Comme nous l'avons vu précédemment, la conception et le développement de supports d'exécution pour systèmes TR²E dynamiquement reconfigurables requiert la prise en compte de nombreux paramètres. Ceux-ci peuvent aussi bien être la conservation de l'aspect temps réel et du caractère embarqué du système reconfiguré, que la préservation de la cohérence du système après reconfiguration.

Le tableau 1.2 résume les caractéristiques des intergiciels pour systèmes TR²E, présentés à la section 1.4. En effet, il précise la granularité des éléments qui constituent chaque intergiciel et indique si un intergiciel donné est adapté au temps réel dur. Il précise également la taille occupée par l'intergiciel sur un exécutable, et spécifie si celui est réflexif ou non. La prise en charge de la reconfiguration dynamique, la portabilité, ainsi que la gestion de la cohérence du système reconfiguré, sont les autres caractéristiques suivants lesquelles les intergiciels sont classés dans ce tableau.

Caractéristiques	Approches				
	DynamicTAO	CIAO	PolyORB_HI	FLARe	SwapCIAO
Granularité	Objet (RT-CORBA)	Composant (LwCCM)	Composant	Objet (RT-CORBA)	Composant (LwCCM)
Temps réel dur	non	oui	oui	non	N/A
Empreinte mémoire	1,5 Mo	5 Mo	70 Ko	10 Mo	N/A
Réflexivité	oui	N/A	non	oui	oui
Reconfiguration dynamique	oui	oui	non	oui	oui
Portabilité	oui (C++)	oui (C++)	oui (Java)	oui (C++)	oui (C++)
Cohérence	oui (difficile)	N/A	non	non	oui

TABLE 1.2 – Tableau récapitulatif des intergiciels pour systèmes TR²E

Ainsi, nous pouvons par exemple remarquer que certains de ces intergiciels ne sont pas adaptés au temps réel dur. En outre, de tous les intergiciels repertoriés ci-dessus, PolyORB_HI est sans doute le mieux adapté à l'embarqué, au vu de son empreinte mémoire. Cependant, à la différence des autres intergiciels, il ne supporte pas les mécanismes de reconfiguration dynamique. Nous pouvons également remar-

quer que les intergiciels ci-dessus n'implantent pas tous les mécanismes de réflexivité ou de cohérence. Cela ne les empêche pas pour autant d'être tous portables.

Dans le prochain chapitre, nous nous attellerons à adresser les limites rencontrées dans les intergiciels pour systèmes critiques supportant la reconfiguration dynamique. Pour ce faire, nous proposerons une approche permettant de développer et de concevoir un support d'exécution qui leur est adapté.

Chapitre 2

Intergiciel pour systèmes TR²E dynamiquement reconfigurables

2.1 Introduction

Dans le chapitre précédent, un état de l'art sur les intergiciels pour systèmes TR²E modifiables dynamiquement a permis de mettre en exergue les limites de ces intergiciels. En effet, ces derniers sont confrontés à la complexité des systèmes critiques, manquent de fonctionnalités suffisantes et sont confrontés au choix entre l'évolutivité et la disponibilité du système. Par ailleurs, ils peuvent se révéler incapables de préserver la cohérence ainsi que l'aspect critique du système reconfiguré.

Dans ce chapitre, nous décrirons la conception de RCES4RTES, un intergiciel adapté aux systèmes TR²E dynamiquement reconfigurables et qui adresse les limites précédemment citées. Ainsi, après avoir rappelé quelques concepts liés à la reconfiguration dynamique, nous détaillerons notre approche, et proposerons ensuite quelques mécanismes permettant de modifier dynamiquement un système. Le respect des contraintes liées aux systèmes critiques constituera le dernier volet de ce chapitre.

2.2 Concepts de base

La réflexivité et la cohérence sont deux notions très importantes pour les intergiciels qui implantent des mécanismes de reconfiguration dynamique. En effet, la première peut permettre de se faire une idée sur le moment à partir duquel la reconfiguration doit être déclenchée. La seconde offre quant à elle un moyen de

s'assurer que le système fonctionne toujours correctement, après qu'une action de reconfiguration ait été effectuée.

2.2.1 Réflexivité

Définition 2.1. *Un système réflexif offre à l'exécution une représentation de lui-même qui est accessible et modifiable [Leg09].*

Les mécanismes de réflexivité peuvent être mis en place non seulement pour connaître l'état du système en cours d'exécution, mais aussi pour introduire des modifications dans un système. Constituant ainsi un atout très important pour les systèmes dynamiques, la réflexivité englobe deux concepts qui sont : l'**introspection** et l'**intercession**.

L'**introspection** est l'aptitude à pouvoir s'observer soi-même. Cette observation consiste à répertorier des éléments concernant la structure et le comportement du système en cours d'exécution. L'introspection peut ainsi permettre de déterminer à quel moment déclencher la reconfiguration du système, et ce en s'appuyant sur son état. Elle permet également de détecter les erreurs dans le système.

L'**intercession** est quant à elle la capacité pour un système de pouvoir se modifier lui-même. L'intercession va donc permettre à un système de s'adapter à son environnement d'exécution, et même d'évoluer selon les besoins. L'implantation de mécanismes de reconfiguration dynamique constitue un moyen de doter un système de capacités d'introspection.

La réflexivité peut s'opérer aussi bien au niveau de la structure que du comportement d'un système. Deux types de réflexion sont alors distingués :

- **la réflexivité structurelle** qui renvoie à l'observation et à la modification de la structure du système. À ce niveau, l'introspection permet de se renseigner sur les éléments constitutifs d'un composant ainsi que sur ses fonctionnalités. L'intercession correspondra quant à elle à la modification structurelle du composant (par exemple : ajout ou suppression d'une connexion),
- **la réflexivité comportementale** concerne l'exécution du programme. Ainsi, la partie introspection va par exemple permettre de connaître les méthodes en cours d'exécution. L'intercession consistera alors à changer la sémantique des constructions du langage utilisé (modification du code d'une méthode, interception d'un appel à une méthode, etc).

2.2.2 Cohérence

Définition 2.2. *Dans notre contexte de travail, la cohérence renvoie à un état correct de l'architecture et du comportement du système.*

L'introduction de modifications dans un système peut nécessiter l'ajout ou le retranchement de certains éléments de la plateforme d'exécution. Cela peut causer des interférences avec les composants en cours d'exécution et engendrer des erreurs dans le système. Le système peut alors provoquer des catastrophes ou même devenir tout simplement inutilisable. Aussi, afin de rendre fiables les reconfigurations dynamiques, il est donc nécessaire de maintenir en permanence la cohérence de ce système au fil des modifications.

Conserver l'état correct d'un système reconfiguré consiste à s'assurer entre autres que [Bes10] :

- la structure du système obtenu soit conforme aux contraintes liées aux interfaces des composants et respecte la façon dont celles-ci sont connectées,
- le résultat de l'interaction entre deux composants donnés soit perçu par l'un et l'autre de la même manière : échec ou réussite.

2.3 Description de l'approche

Afin d'adresser les différentes limites répertoriées dans les intergiciels pour systèmes TR²E, nous nous proposons de concevoir et de développer un support d'exécution appelé RCES4RTES (Reconfigurable Computing Execution Support for Real Time Embedded Systems).

2.3.1 Adéquation aux systèmes TR²E

Afin d'être adapté aux systèmes TR²E, RCES4RTES supporte la création et l'exécution de quatre types de tâches à savoir : les tâches périodiques, sporadiques, hybrides et apériodiques. Les tâches périodiques sont des processus cycliques qui se répètent après un laps de temps spécifique. Les tâches sporadiques se caractérisent par une exécution cyclique, déclenchée à chaque fois par la réception d'un évènement. Les tâches hybrides possèdent quant à elles des caractéristiques communes aux deux types de tâches précédents. Les tâches apériodiques sont pour leur part, caractérisées par une exécution unique, ainsi que par des instants de début et de fin.

2.3.2 Assurance de la réflexivité

Nous nous intéressons ici à la partie introspective de la réflexivité. En effet, elle permet d'obtenir des informations sur l'état du système en cours d'exécution. Cela offre ainsi la possibilité de mieux cibler le choix des actions de reconfiguration. L'observation du système en cours d'exécution joue donc un rôle important dans le processus de reconfiguration dynamique. À cet effet, RCES4RTES se propose donc de fournir des mécanismes qui vont permettre de déterminer :

- le nombre de composants en cours d'exécution dans le système,
- le nombre de connexions qui relie un composant donné aux autres composants avec lesquels il interagit,
- les derniers temps d'accès en lecture et/ou en écriture des variables partagées de l'application.

2.3.3 Reconfiguration dynamique

Pour décrire les mécanismes de reconfiguration dynamique dans les systèmes TR²E, RCES4RTES propose des fonctionnalités permettant de modifier aussi bien la structure d'un système (reconfiguration structurelle) que le comportement de celui-ci (reconfiguration comportementale).

Au niveau de la reconfiguration structurelle, RCES4RTES supporte les actions suivantes :

- **l'ajout d'un composant** consistant à créer et à déployer ce dernier sur la plateforme d'exécution,
- **la suppression d'un composant** effectuée en le déconnectant d'avec les composants avec lesquels il interagit, en annulant son déploiement, puis en le supprimant de la plateforme d'exécution,
- **la migration de composants** qui revient à déplacer un composant à un autre emplacement de la plateforme d'exécution,
- **l'ajout d'une connexion entre deux composants** consistant à ajouter une voie de communication entre deux composants,
- **la suppression d'une connexion entre deux composants** revenant à supprimer une voie de communication entre deux composants donnés.

Les actions concernant la reconfiguration comportementale du système consisteront à effectuer les opérations suivantes :

- **le remplacement d'un composant** qui revient tout d'abord à supprimer un composant de la plateforme d'exécution, puis à ajouter un autre composant d'un type donné à un emplacement donné. Cette action de reconfiguration pourra être utilisée pour modifier l'implémentation d'un composant, dans le cas où ce dernier est remplacé par un composant de type différent, et situé au même endroit que le composant initial.
- **la modification des propriétés d'un composant** qui va consister à affecter de nouvelles valeurs aux propriétés de ce composant, ou à ajouter des propriétés supplémentaires à ce composant.

2.3.4 Assurance de la cohérence

Les actions de reconfiguration sont susceptibles d'affecter le système en cours d'exécution. En effet, elles peuvent causer des interférences avec les interactions se produisant entre les composants. Ainsi, si les opérations de reconfiguration ne sont pas contrôlées, elles peuvent mener le système dans un état incohérent et le rendre inutilisable.

Voici quelques actions de reconfiguration dynamique qui peuvent conduire un système dans un état incohérent :

- **la modification des propriétés d'un composant alors qu'il traite des requêtes**, car le changement des propriétés d'un composant peut affecter les paramètres liés au traitement d'une requête donnée. Cela peut compromettre l'exactitude du traitement de cette dernière,
- **la suppression d'un composant alors qu'il a des requêtes en cours**, car les requêtes qui sont en train d'être évaluées ne seront pas totalement traitées. Les composants en attente de leur traitement pourraient alors être perturbés dans leur exécution ou même bloqués,
- **la suppression d'un composant sans supprimer ses connexions avec d'autres composants**, ce qui comporte le risque que les composants qui interagissent avec le composant supprimé continuent à lui envoyer des requêtes et soient gênés et même bloqués par la non réception de leurs réponses.

Aussi, RCES4RTES effectue la reconfiguration en tenant compte de la cohérence du système à reconfigurer. Ainsi, pour tout composant dont la modification peut engendrer des incohérences dans le système, il s'assure que :

- ce composant soit bloqué (et au besoin les composants qui lui envoient des données) avant que l'opération de reconfiguration proprement dite puisse être exécutée. Il faut noter que l'opération de blocage consiste à s'assurer que le composant ait terminé de traiter ses requêtes en cours, et ensuite à stopper son activité pendant le laps de temps que dure l'action de reconfiguration,
- une fois le composant reconfiguré, il est alors débloqué (s'il n'a pas été supprimé lors de la reconfiguration) afin de reprendre une activité normale. Si les composants qui lui envoient des données ont aussi été bloqués à l'étape 1, ils sont également débloqués.

2.3.5 Recours aux services canoniques

RCES4RTES s'inspire de l'architecture des intergiciels schizophrènes dans la mesure où la plupart des neuf services canoniques qu'elle propose peuvent être paramétrés selon les besoins de l'application. Cette caractéristique permet de concevoir un intergiciel dédié à l'application pour laquelle il est créé.

À cet effet, RCES4RTES va reprendre le principe de personnalisation des services canoniques déjà évoqué dans la section 1.4.2.3. Ce principe consiste à scinder les services canoniques en deux familles contenant d'une part des services génériques, et d'autre part, des services paramétrables en fonction des besoins de l'application. Cette technique offre l'avantage de séparer les préoccupations, et permet d'obtenir un intergiciel dédié à une application donnée.

Puisque ces services sont pour la plupart statiques, RCES4RTES se propose de les modifier afin qu'ils puissent permettre à l'application cible d'être reconfigurée en cours d'exécution.

2.3.5.1 Les services faiblement personnalisables

Ils sont invariants quelle que soit l'application pour laquelle ils sont conçus. Ils seront donc générés à la main. Ces services sont les suivants :

Le parallélisme qui se charge de créer les archétypes (fonctionnalités génériques décrivant le comportement des tâches). Il effectue aussi pour chaque tâche se trou-

vant sur un nœud de l'application, l'instanciation de l'archétype correspondant à sa catégorie (périodique, sporadique, etc).

La représentation élémentaire se charge quant à elle de l'emballage et du déballage des types de données élémentaires (entiers, booléens, etc). Ainsi, si les plates-formes sur lesquelles les nœuds s'exécutent sont homogènes, la copie mémoire des données dans un tampon de communication pourra être utilisée comme méthode pour emballer et déballer les données de ce tampon. Dans le cas échéant, des méthodes d'emballage beaucoup plus complexes pourront être utilisées. Celles-ci auront par exemple recours à l'alignement des données en mémoire.

L'interrogation fournit les mécanismes régissant l'échange de données entre les tâches et permet de gérer les files d'attente du côté des tâches qui reçoivent les données. Ce service offre également des fonctionnalités permettant la détermination de l'identité de l'expéditeur d'un message ainsi que la lecture du contenu de ce message.

Le protocole permet la mise en place des étapes nécessaires à la transmission d'un message. En effet, il ajoute une en-tête au message et l'envoie à la destination. Dans le cas où le message est trop long, il le fragmente avant de l'expédier, et se charge ensuite de le reconstituer à son arrivée sur le nœud destinataire. Dans le cas d'une communication synchrone, c'est également ce service qui va s'assurer que l'émetteur demeure bloqué jusqu'à ce que l'accusé de réception soit reçu.

La couche basse de transport propose des mécanismes permettant l'accès au moyen de communication entre les nœuds d'une application donnée. Ce service n'est pas nécessaire dans une application monolithique car il est utilisé dans le cadre d'un dialogue entre des entités se trouvant sur des nœuds distants d'une application. Il permet ainsi l'initialisation des structures de données nécessaires à un tel dialogue, ainsi que l'ouverture des canaux de communication qui vont interconnecter tous les nœuds. Par ailleurs, il se charge également de l'envoi et de la réception des données sur le réseau.

2.3.5.2 Les services fortement personnalisables

Leur structure est grandement influencée par l'application pour laquelle ils sont conçus. Ils pourront être générés automatiquement en recourant à des outils adaptés à ce processus [Zal08]. Ces services sont les suivants :

L'adressage chargé de la création, ainsi que de la gestion des références (adresses). Celles-ci vont permettre non seulement la connexion, mais aussi l'envoi de données à une entité référencée. Ce service est mis en place à l'aide de tables statiques permettant d'associer des adresses aux noms des entités.

La liaison est le service chargé de la gestion du mode de connexion entre les entités (envoi de messages, appels de procédures distantes, objets distants). Il s'occupe également de la construction et de la sélection des ressources des services (protocole, représentation, interaction et transport) impliqués dans le dialogue entre une entité adressée et une entité locale ou distante.

L'activation rend active l'entité nécessaire au traitement d'une requête, et engendre la création ou la destruction des objets utiles pour effectuer un traitement donné.

Le typage comme son nom l'indique, est chargé de gérer les types de données échangées entre les entités de l'application. La complexité de sa mise en œuvre est proportionnelle à celle du système dans lequel il doit être implanté.

L'exécution permet la création de toutes les tâches nécessaires au fonctionnement de l'application, et ce grâce au nombre et aux caractéristiques (taille de pile, période, etc) de celles-ci.

La représentation avancée propose les mécanismes permettant d'emballer et de déballer les types de données spécifiés par l'utilisateur. À cet effet, elle va utiliser les méthodes fournies par la représentation élémentaire. Ce service se charge également du calcul de la taille des tampons de communication ainsi que de l'allocation de ces tampons.

L'interaction déduit pour chaque entité, les éléments requis à la détermination du mode de communication de cette dernière, ainsi que la nature des données qu'elle pourrait échanger. Ce service a aussi pour rôle d'instancier les fonctions d'envoi et de réception fournies par le service interrogation.

La haute couche de transport se charge de l'initialisation des couches basses de transport (Ethernet, SpaceWire , etc) qui seront utilisées par le nœud courant. Elle invoque également les méthodes d'envoi des couches basses de transport ainsi que de la couche protocole, afin d'acheminer et de délivrer les messages à destination.

2.4 Conception de l'approche

L'architecture de RCES4RTES repose sur les services canoniques scindés et répartis selon leur personnalisation. Cependant, ces derniers sont adaptés aux systèmes statiques et ne prennent donc pas en charge les mécanismes de reconfiguration dynamique. En effet, le **parallélisme** n'implante pas les archétypes permettant de modifier dynamiquement la structure ou le comportement d'un composant (modification de propriétés, remplacement, migration, etc). Par ailleurs, les tâches étant cycliques, elles sont donc prévues pour s'exécuter indéfiniment dans le système. Cet aspect ne tient donc pas compte du fait qu'une tâche puisse être interrompue afin d'être supprimée de la plate-forme d'exécution. L'**activation** n'est pas non plus adaptée à la reconfiguration dynamique, car elle ne permet pas de rendre actives les entités créées dynamiquement.

De plus, la **couche basse de transport** prévoit de connecter tous les nœuds les uns aux autres. Elle n'offre pas la possibilité de connecter exclusivement un nœud donné avec les nœuds distants nécessaires à son exécution. Elle ne permet pas non plus de déconnecter un nœud donné d'avec un nœud distant, lorsque leur connexion ne s'avère plus nécessaire.

La gestion des références étant implantée par le service d'**adressage** à l'aide de tables statiques, ceci constitue également un frein à la mise en place de mécanismes de reconfiguration. En effet, ces derniers requièrent l'utilisation de tables dynamiques qui vont permettre de créer ou de supprimer des références en cours d'exécution. De plus, le service **liaison** ne permet pas d'ajouter ou de supprimer des connexions entre composants, encore moins de sélectionner dynamiquement des instances des services impliqués dans les communications distantes. Il n'offre pas non plus la possibilité de choisir dynamiquement la fabrique à utiliser afin de réaliser la liaison entre les entités.

En outre, la **représentation avancée** n'offre pas les fonctionnalités permettant de calculer dynamiquement la taille des tampons de communication, ou encore d'allouer ces tampons en cours d'exécution.

Par ailleurs, les services canoniques ne fournissent pas les fonctionnalités permettant de connaître l'état du système en cours d'exécution afin de bien cibler l'action de reconfiguration. Et comme ils n'offrent pas des possibilités de reconfiguration

dynamique, ils ne permettent donc pas de s'assurer que le système demeure correct une fois qu'il a été reconfiguré.

Pour y remédier, RCES4RTES se propose d'étendre les services canoniques afin de les adapter aux systèmes TR²E dynamiques. Les fonctionnalités qui leur seront rajoutées leur permettront alors d'effectuer la reconfiguration dynamique, en procédant comme suit :

- observation du système en cours d'exécution afin de connaître son état et de mieux calibrer l'opération de reconfiguration à effectuer,
- exécution de l'opération de reconfiguration souhaitée,
- prise en charge de la continuité de la cohérence du système reconfiguré.

Dans la suite de cette section, nous proposons des algorithmes illustrant les fonctionnalités ajoutées aux services canoniques afin de les rendre dynamiques.

2.4.1 Mécanismes de réflexivité

Afin de permettre à un système d'observer sa propre exécution, nous proposons les algorithmes suivants :

Détermination du nombre d'instances d'un type de composants

Afin de pouvoir dénombrer les instances d'un type de composants t donné sur un noeud n , nous proposons l'algorithme 1 dont le principe est le suivant : si un composant de type t est créé sur un noeud n de la plateforme d'exécution, le nombre de composants de ce type sera incrémenté d'une unité. Par contre, si un composant de type t est supprimé du noeud n , le nombre de composants correspondant à ce type sera décrémenté d'une unité.

Algorithme 1 Mise à jour du nombre d'instances d'un composant

MajInstances (action : Chaîne ; t : TypeComposant ; n : Nœud)

```
1  si (existe(t,n)) alors
2      si (action="creerComposant") alors
3          t.nb_comp ← t.nb_comp +1
4  sinon
5      si(action="supprimerComposant") alors
6          t.nb_comp ← t.nb_comp -1
```

Détermination du nombre de connexions d'un composant

Afin de déterminer le nombre de connexions qui permettent à un composant donné d'interagir avec d'autres composants, l'algorithme 2 est proposé. Cet algorithme consiste à parcourir tous les ports P d'un composant C donné en paramètre. Si P est un port d'entrée, le nombre de connexions est incrémenté par le nombre de ports qui envoient des données au port P . Sinon (P est un port de sortie), le nombre de connexions est incrémenté par le nombre de ports à qui P envoie des données.

Algorithme 2 Calcul du nombre de connexions d'un composant

```
nombreConnexions (C : Composant)
1  pour chaque port P appartenant à C faire
2      si (C.P.type()=PortDentree) alors
3          nb_conn ← nb_conn+C.P.nombreDePortSources()
4  sinon
5      nb_conn ← nb_conn+C.P.nombreDePortDestinations()
6  retourne nb_conn
```

Observation des variables partagées de l'application

L'algorithme 3 permet quant à lui de stocker les derniers accès en lecture et/ou écriture d'une variable partagée de l'application. Pour ce faire, il utilise une table de hachage appelée *Acces* dont chaque clé d'accès correspond à une variable partagée de l'application. Dans *Acces*, chaque clé est associée à une structure de données nommée *TempsAcces*. Cette dernière possède deux attributs : R et W qui permettent respectivement de stocker les plus récents temps d'accès en lecture et en écriture d'une variable partagée donnée.

L'algorithme prend en paramètre une variable dénommée $nomV$ dont les plus récents temps d'accès doivent être stockés. Il prend également en paramètre une variable appelée *action* indiquant si une méthode est en train d'accéder à $nomV$ en lecture, écriture ou même les deux. Ainsi, il s'agira dans un premier temps de vérifier si $nomV$ est effectivement stockée dans *Acces* (et donc reconnue comme étant une variable partagée de l'application). Si cette condition est vérifiée, alors l'instant actuel est récupéré et stocké dans l'attribut adéquat de *TempsAcces*.

Algorithme 3 Observation des variables partagées

```
ObservationVariables (action, nomV : Chaîne de caractères)
1  si (Acces.contientClé(nomV)) alors
2      TempsAcces ← Acces.getValeur(nomV)
3      si (action="écriture") alors
4          tempsAcces.W ← getCurrentTime()
5      sinon si (action="lecture") alors
6          tempsAcces.R ← InstantPrsent()
7      sinon si (action="lecture/écriture") alors
8          tempsAcces.W ← getCurrentTime()
9          tempsAcces.R ← getCurrentTime()
10     sinon
11         afficher("l'accès est en lecture/écriture")
12     Acces.modifier(nomV, tempsAcces)
13 sinon
14     afficher(nomV+" n'est pas une variable partagée")
```

2.4.2 Mécanismes de cohérence

Si une action de reconfiguration dynamique est susceptible de provoquer des interférences avec des composants en cours d'exécution, il s'avère alors nécessaire de faire en sorte que le système demeure cohérent, une fois cette action exécutée. Ainsi, juste avant d'effectuer cette action de reconfiguration, nous invoquons une méthode permettant de bloquer l'activité du composant à reconfigurer, et éventuellement, celle des composants qui lui envoient des données. Une fois que la reconfiguration a été effectuée, nous invoquons une méthode permettant de débloquent le composant reconfiguré ainsi que les autres éventuels composants qui ont été bloqués.

Blocage du composant à reconfigurer

Le blocage d'un composant nécessite qu'il ait terminé de traiter toutes ses requêtes en cours. Et pour que ce composant n'ait plus de requêtes supplémentaires à traiter, il est parfois nécessaire de bloquer les composants qui lui envoient des requêtes (composants sources). Lorsque c'est fait, le composant doit attendre d'avoir terminé de traiter ses requêtes, avant de stopper son activité à l'aide d'un verrou. L'algorithme 4 illustre cette démarche.

Algorithme 4 Blocage d'un composant

```
bloquerComposant (C : Composant ; bloquerSources : Booléen)
1  si (bloquerSources==Vrai) alors
2      pour chaque composant S source de C faire
3          bloquerComposant(S, Faux)
4  C.attendreFinRequetes()
5  C.verrou.obtenir()
```

Déblocage du composant à reconfigurer

Afin d'être débloqué, un composant relâche le verrou qu'il a précédemment acquis, ce qui lui permet de reprendre son exécution. Dans le cas où ses composants sources (composants qui lui envoient des données) doivent aussi être débloqués, le même principe de déblocage leur est aussi appliqué tour à tour. L'algorithme 5 reflète ce comportement.

Algorithme 5 Déblocage d'un composant

```
debloquerComposant (C : Composant ; debloquerSources : Booléen)
1  C.verrou.relâcher()
2  si (debloquerSources==Vrai) alors
3      pour chaque composant S source de C faire
4          debloquerComposant(S, Faux)
```

2.4.3 Mécanismes de reconfiguration dynamique

Notre intergiciel effectue des actions de reconfiguration dynamique en se basant sur les algorithmes décrits ci-dessus.

Ajout d'une connexion entre deux ports

L'ajout d'une connexion entre deux nœuds est une action de reconfiguration qui consiste à connecter deux ports. Ainsi, s'il faut établir une connexion entre un port $P1$ et un port $P2$, il est nécessaire de vérifier si les composants associés à ces deux ports sont imbriqués et si les ports concernés ont la même orientation. Dans le cas échéant, il est alors nécessaire de vérifier que $P1$ est un port d'entrée et que $P2$ est un port de sortie. Naturellement, une telle connexion n'est possible que si les deux ports en question ne sont pas encore connectés.

De plus, le composant associé à $P2$ peut avoir besoin d'un instant à l'autre des données se trouvant sur le composant associé à $P1$. Il s'avère alors nécessaire de bloquer le composant associé à $P1$, avant d'établir la connexion entre les deux ports. Ainsi, les données se trouvant sur le composant ayant $P1$ pour port pourront être envoyées au composant associé à $P2$, et ce juste après la connexion. L'algorithme 6 illustre cette démarche.

Algorithme 6 Ajout d'une connexion entre deux ports

```
ajouterConnexion ( P1, P2 : Port)
1  si (((imbriques(P1.getComposant()),P2.getComposant() ET memeOrient-
   ation(P1,P2)) OU (!imbriques(P1.getComposant()),P2.getComposant() ET
   (P1.type()=PortDeSortie ET P2.type()=PortDentree ))) ET existeConnexion
   (P1, P2)=Faux ) alors
2    bloquerComposant(P1.getComposant())
3    P1.ajouterPortDestination(P2)
4    debloquerComposant(P1.getComposant())
```

Suppression d'une connexion entre deux ports

L'algorithme 7 qui s'appuie sur des pré-conditions similaires à celles de l'algorithme 6 permet de supprimer une connexion entre deux ports $P1$ et $P2$. Ces deux ports doivent être au préalable connectés. À ce niveau, le composant associé au port $P1$ est bloqué car il peut y avoir des requêtes en cours qui ont besoin d'utiliser la connexion entre $P1$ et $P2$ pour leur envoi. Ainsi, une fois que celles-ci ont été expédiées, l'activité du composant est bloquée afin qu'il n'envoie plus de données via cette connexion. Le composant pourra alors reprendre ses activités, une fois la connexion supprimée.

Algorithme 7 Suppression d'une connexion entre deux ports

```
supprimerConnexion ( P1, P2 : Port)
1  si (((imbriqués(P1.getComposant()),P2.getComposant() ET memeOrient-
   ation(P1,P2)) OU (!imbriqués(P1.getComposant()),P2.getComposant() ET
   (P1.type()=PortDeSortie ET P2.type()=PortDentree ))) ET existeConnexion
   (P1, P2)=Vrai ) alors
2    bloquerComposant(P1.ComposantAssocie())
3    P1.supprimerPortDestination(P2)
4    debloquerComposant(P1.ComposantAssocie())
```

Ajout d'une connexion entre deux nœuds

Le dialogue entre deux nœuds d'une application répartie est rendu possible via l'établissement d'une connexion entre ces deux nœuds. Pour ce faire, il faut s'assurer que ces nœuds ne sont pas connectés et qu'ils sont distincts. Puis, si le second nœud est appellable à distance¹, un point de communication peut être créé. La taille maximale des données qu'il pourra faire transiter est alors fixée. Ce point de communication est ensuite associé à l'adresse réseau du second nœud. Puis, pour permettre au premier nœud d'envoyer des données au second, un flux de sortie est alors récupéré. Sous réserve que le premier nœud soit appellable à distance, un flux d'entrée est également récupéré. Il permettra au premier nœud de recevoir des données en provenance du second. Cette démarche est illustrée par l'algorithme 8.

Algorithme 8 Connexion de deux nœuds distants

```
connexionNœuds (taille : Bits ; nœud1, nœud2 : Nœud)
1  si (existeConnexion(nœud1, nœud2) == FAUX) ET
    nœud2.appelableADistance ET nœud1 ≠ nœud2) alors
2      S ← creerUnPointDeCommunication()
3      S.fixerTailleMessage(taille)
4      connecter (S, nœud2.adresseReseau())
5      nœud2.fluxDeSortie ← récupérerFluxDeSortie(S)
6      si (nœud1.appelableADistance()) alors
7          nœud2.fluxDentree ← recupererFluxDentree(S)
8      sinon
9          afficher("Le " + nœud1 + " n'est pas appellable à distance")
10     nœud2.fluxDeSortie.ecrire(nœud1)
```

Déconnexion de deux nœuds

Il est nécessaire d'établir une connexion entre deux nœuds d'une application répartie, afin que ceux-ci puissent dialoguer. Lorsque cette connexion n'est plus nécessaire, ces deux nœuds peuvent être déconnectés afin de libérer les ressources utilisées par cette connexion. Pour ce faire, nous vérifions au préalable qu'il existe bien une connexion entre les deux nœuds en question. Si c'est effectivement le cas, nous fermons le flux de sortie utilisé par le premier nœud pour envoyer des données au second nœud. Si le premier nœud est appellable à distance, nous fermons également

1. Il possède une adresse réseau qui permet de l'atteindre à distance

le flux d'entrée par lequel il reçoit des données en provenance du second nœud. L'algorithme 9 reflète ce processus.

Algorithme 9 Déconnexion de deux nœuds distants

```
deconnexionNœuds (nœud1, nœud2 : Nœud)
1  si (existeConnexion(nœud1, nœud2)==VRAI ET nœud1 ≠ nœud2) alors
2      nœud2.fluxDeSortie.fermer()
3      si (nœud1.appelableADistance()) alors
4          nœud2.fluxDentree.fermer()
5  sinon
6      afficher(nœud1+" et "+nœud2+" ne sont pas connectés")
```

Modification des propriétés d'un composant

Pour modifier des propriétés d'un composant (priorité, échéance, etc), l'algorithme 8 se propose de diviser les propriétés à modifier en deux groupes. Le premier groupe comprend les propriétés dont la modification ne va pas engendrer des incohérences dans le traitement des requêtes en cours : ce sont les propriétés non critiques. Ce premier groupe sera stocké dans le tableau *P_non_critique*. Le deuxième groupe va quant à lui être constitué de propriétés dont la modification peut mener le système dans un état incohérent : ce sont des propriétés critiques. Le tableau *P_critique* va permettre de stocker ces dernières. Celles-ci requièrent le blocage du composant à modifier.

Algorithme 10 Modification des propriétés du composant

```
modifierPropriétés ( C : Composant ; P[] : Proprietes)
1  P.classifierProprietes(P_critique, P_non_critique)
2  pour i de 1 à taille de P_non_critique faire
3      modifierPropriete(P_non_critique[i],C)
4  si (P_critique.taille>0) alors
5      bloquerComposant(C)
6      pour i de 1 à taille de P_critique faire
7          modifierPropriete(P_critique[i],C)
8      débloquerComposant(C)
```

Ajout d'un nouveau composant

L'algorithme 11 permet de créer un nouveau composant sur la plateforme d'exécution, en prenant en paramètre : son type de composant t , le nœud n sur lequel il sera créé, la liste des ports à qui il enverra des données ($dest$), et la liste des ports qui lui enverront des données ($source$). Le tableau $dest$ comprend ainsi les paires associant chaque port de sortie du composant à créer, à son port destination (port d'entrée). Le tableau $source$ contient quant à lui les couples associant chaque port d'entrée du composant à créer, au port de sortie qui lui envoie des données.

Ainsi, le composant est instancié en fonction de son type t et déployé sur le nœud n . Ensuite, chacune de ses connexions spécifiées grâce aux tableaux $source$ et $dest$ est alors créée.

Algorithme 11 Ajout d'un composant dans un nœud

```
ajouterComposant (t : TypeComposant ; n : Nœud ; dest[], source[] :Table
de hachage)
1  si (existe(t, n)) alors
2      C ← créerInstance(t)
3      deployer(C, n)
4      pour i de 1 à taille de dest faire
5          P1 ← getPortDeSortie(dest[i][1])
6          P2 ← getPortDentree(dest[i][2])
7          ajouterConnexion(P1, P2)
8      pour i de 1 à taille de source faire
9          P1 ← getPortDeSortie(source[i][2])
10         P2 ← getPortDentree(source[i][1])
11         ajouterConnexion(P2, P1)
12     retourner C
```

Suppression d'un composant

L'algorithme 12 permet de supprimer un composant de la plateforme d'exécution. Pour ce faire, les connexions existant entre le composant courant et les composants qui lui envoient des données sont supprimées. Puis, L'activité du composant à supprimer est stoppée, et ce, après qu'il ait exécuté toutes ses requêtes en cours. Ensuite, les connexions permettant à ce composant d'envoyer des données à ses destinataires sont supprimées. Après que le déploiement du composant ait été annulé, il est alors supprimé de la plateforme d'exécution.

Algorithme 12 Suppression d'un composant dans un nœud

```
supprimerComposant ( C : Composant ; n : Nœud)
1  pour chaque port d'entrée P1 de C faire
2      pour chaque port de sortie P2 source de P1 faire
3          supprimerConnexion(P2,P1)
4  bloquerComposant(C)
5  pour chaque port de sortie P1 de C faire
6      pour chaque port d'entrée P2 destination de P1 faire
7          P1.retirerPortDestination(P2);
8  annulerDeploiement(C,N)
9  supprimerInstance(C)
```

Migration d'un composant vers un autre nœud

Afin de migrer un composant d'un nœud 1 vers un nœud 2, nous proposons de bloquer les composants sources du composant à reconfigurer, afin qu'ils ne lui envoient plus de messages supplémentaires. L'activité du composant à reconfigurer est ensuite stoppée après qu'il ait terminé de traiter ses requêtes. La file d'attente du composant est alors vide. L'état du composant est ensuite sauvegardé dans la variable *Propriétés*. Cette dernière est alors sérialisée et le flux obtenu est ensuite envoyé sur le réseau, en direction du nœud 2. Lorsque ce flux arrive à destination, il est désérialisé pour permettre de créer le composant *C2* avec les mêmes propriétés que le composant *C1*. Cette démarche est illustrée par l'algorithme 13.

Algorithme 13 Migration d'un composant vers un autre nœud

```
migrerComposant (C1 : Composant ; noeud1, noeud2 : Nœud)
1  si (noeud1  $\neq$  noeud2) alors
2      bloquerComposantsSources(C1)
3      bloquerComposant(C1)
4      Proprietes  $\leftarrow$  C1.getEtat()
5      flux_out  $\leftarrow$  serialiser(Propriétés, noeud1)
6      envoyerSerialisation(flux_out, noeud2)
7      flux_in  $\leftarrow$  recevoirSérialisation()
8      C2  $\leftarrow$  deserialiser(flux_in, noeud2)
9      lancerComposant(C2, noeud2)
10     supprimerComposant( C1, noeud1)
11     debloquerComposantsSources(C1)
12 sinon
13     afficher("la migration se passe entre deux nœuds distincts")
```

Remplacement d'un composant

L'algorithme 14 peut être utilisé pour remplacer un composant s'exécutant sur un nœud 1, par un autre composant d'un type donné, et dont les propriétés nécessaires à sa création sont stockées dans le tableau *Param*. Si le composant à supprimer, ainsi que son remplaçant doivent se trouver sur le même nœud, alors le composant remplaçant est créé et lancé sur le nœud courant. Le composant initial pourra ensuite être supprimé de la plateforme d'exécution.

Mais dans le cas où le composant remplaçant doit se trouver sur le nœud 2 qui est distinct du nœud 1, alors les propriétés du composant à créer et qui sont stockées dans *Param* sont sérialisées dans un flux et envoyées sur le réseau en direction du nœud 2. Lorsque le flux arrive à destination, il est désérialisé et les propriétés résultantes de ce processus servent à créer le composant remplaçant. Ainsi, l'ancien composant peut être supprimé sur le nœud 1, tandis que son remplaçant est lancé sur le nœud 2.

Algorithme 14 Remplacement d'un composant

```
remplacerComposant(C1 : Composant ; noeud1, noeud2 : Nœud ; Param[] : Property)
1  si (noeud1==noeud2 ) alors
2      C2 ←ajouterComposant(Param[1], noeud1, Param[2], Param[3])
3      bloquerComposant(C1)
4      lancerComposant(C2, noeud1)
5  sinon
6      flux_out ← serialiser(Propriétés, noeud1)
7      envoyerSerialisation(flux_out,noeud2)
8      supprimerComposant(C1, noeud1)
9      flux_in ← recevoirSérialisation()
10     C2 ← deserialiser(flux_in,noeud2)
11     lancerComposant(C2, noeud2)
12     supprimerComposant(C1, noeud1)
13
```

2.5 Respect des contraintes liées aux systèmes critiques

Les systèmes TR²E sont contraints à s'adapter et même à évoluer en fonction des variations de leur environnement d'exécution. Pour ce faire, ils ont généralement

recours aux actions de reconfiguration dynamique. Néanmoins, cette dernière peut remettre en cause l'aspect critique de ces systèmes. En effet, elle peut être indéterministe et même trop coûteuse en ressources. RCES4RTES propose des solutions pour remédier à ces inconvénients.

2.5.1 Respect de l'aspect temps réel

Le temps total nécessaire à la reconfiguration est déterminé en fonction des paramètres suivants :

- T_b : temps nécessaire pour bloquer un composant,
- T_{etat} : temps mis pour transférer l'état d'un composant,
- T_{op} : temps mis pour effectuer l'opération de reconfiguration proprement dite (ajout d'une connexion, suppression d'un composant, etc).

Ainsi, si T_{rd} est le temps qui s'écoule entre l'instant où la reconfiguration est demandée et l'instant où la nouvelle configuration est opérationnelle, la formule suivante peut être déduite :

$$T_{rd} = T_b + T_{etat} + T_{op} \quad (2.1)$$

Les temps situés à la droite de l'équation 2.1 sont proportionnels à la taille des données à traiter (composants impliqués dans la reconfiguration, attributs, connexions, etc) et dépendent donc de la vitesse des processeurs et/ou du moyen de communication utilisé (Ethernet, SpaceWire, etc). Le système reconfiguré étant à la base temps réel, ses processeurs vont donc optimiser l'exécution de chacune des opérations représentées par ces temps. Par ailleurs, si la couche de transport est temps réel, à l'image de SpaceWire, les temps de communication seront alors déterministes. Au vu de toutes ces considérations, nous pouvons donc déduire que le temps de reconfiguration T_{rd} est déterministe.

Notons que dans la formule 2.1, nous n'avons pas considéré la latence due à l'accès à la mémoire, car cette dernière doit être calculée pendant la phase de conception de l'application.

2.5.2 Respect du caractère embarqué

Les systèmes TR²E sont soumis à de fortes contraintes, notamment au niveau de la gestion des ressources disponibles. Une action de reconfiguration dynamique

peut alors s'avérer être conflictuelle avec la garantie de l'aspect embarqué de ces systèmes. En effet, elle peut nécessiter l'ajout d'éléments supplémentaires sur la plateforme d'exécution, ce qui peut être coûteux notamment en terme de mémoire ou d'utilisation du processeur.

Aussi, avant toute action de reconfiguration nécessitant une allocation de la mémoire, nous vérifions qu'il reste suffisamment de mémoire permettant d'effectuer cette action. Par ailleurs, nous nous proposons d'optimiser le code à implanter, de façon à pouvoir réduire sa taille, ainsi que sa complexité.

2.6 Conclusion

La description de la conception de RCES4RTES est le maillon sur lequel ce chapitre s'est articulé. Nous avons ainsi pu y montrer les voies et moyens employés par notre intergiciel pour mener à bien les actions de reconfiguration dynamique, tout en préservant la cohérence ainsi que l'aspect critique du système reconfiguré. La figure 2.1 résume les articulations comportées par notre approche.

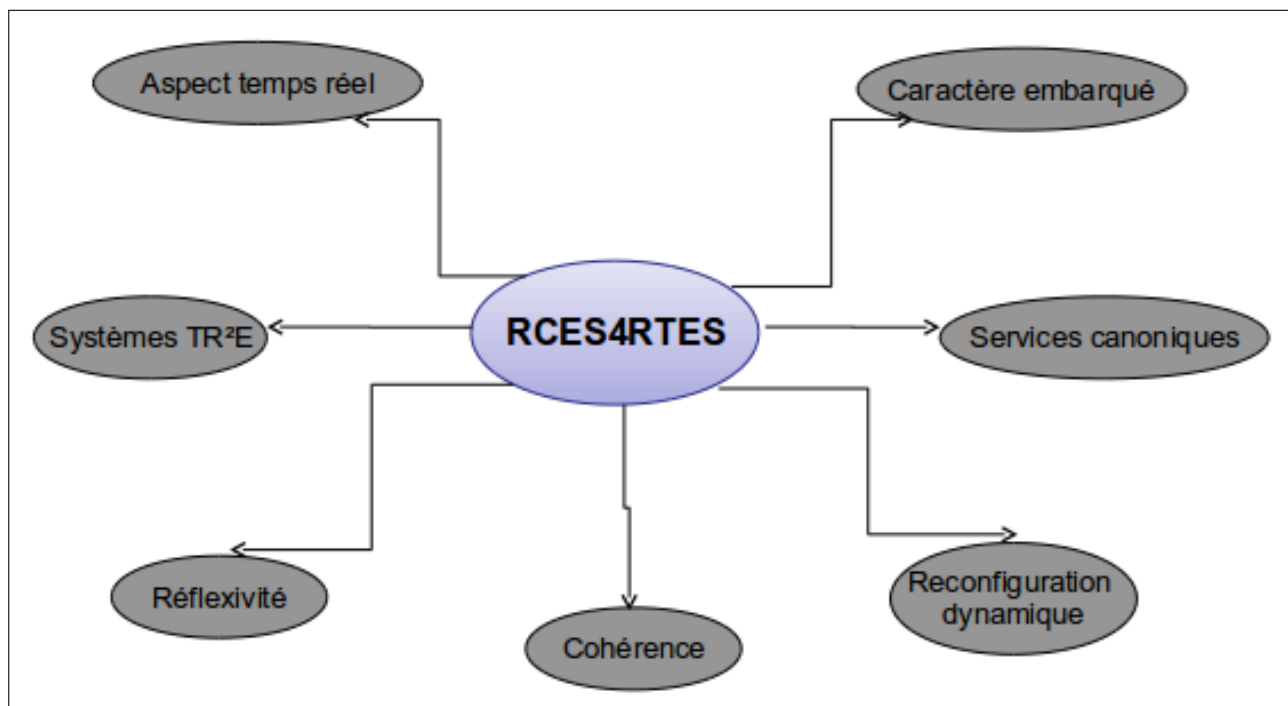


FIGURE 2.1 – Points clés de notre approche

2.6 Conclusion

Dans le chapitre qui suit, nous nous attellerons à détailler les techniques employées pour mettre en œuvre notre approche.

Chapitre 3

Mise en oeuvre de l'approche

3.1 Introduction

Dans le chapitre précédent, nous nous sommes attelés à indiquer les points clés de notre approche. Pour ce faire, nous avons présenté la conception d'un intergiciel appelé RCES4RTES, qui est adapté aux systèmes TR²E dynamiquement reconfigurables. Il s'agit alors à présent de répertorier les voies et moyens permettant de mettre en oeuvre notre approche, tout en respectant ses spécifications.

Aussi, après avoir donné un aperçu de notre cadre de travail, nous nous attellerons dans ce chapitre à spécifier les environnements de travail ainsi que les langages de programmation utilisés. Nous y expliciterons ensuite nos choix d'implémentation.

3.2 Cadre du travail

Notre travail de mastère s'inscrit dans le cadre de travaux de recherche débutés par Mme Fatma Krichen il y a maintenant trois ans. Ces travaux ont pour objectif de proposer une approche dirigée par les modèles. Cette dernière vise l'automatisation du développement des systèmes TR²E dynamiques [KHZC10]. Ces travaux se répartissent en trois étapes illustrées par la figure 3.1. Ces étapes sont développées dans les lignes qui suivent.

3.2.1 Modélisation des systèmes TR²E dynamiques

Cette étape consiste à spécifier les architectures des systèmes TR²E dynamiquement reconfigurables. C'est à cet effet qu'est introduit le méta-modèle RCA4RTES

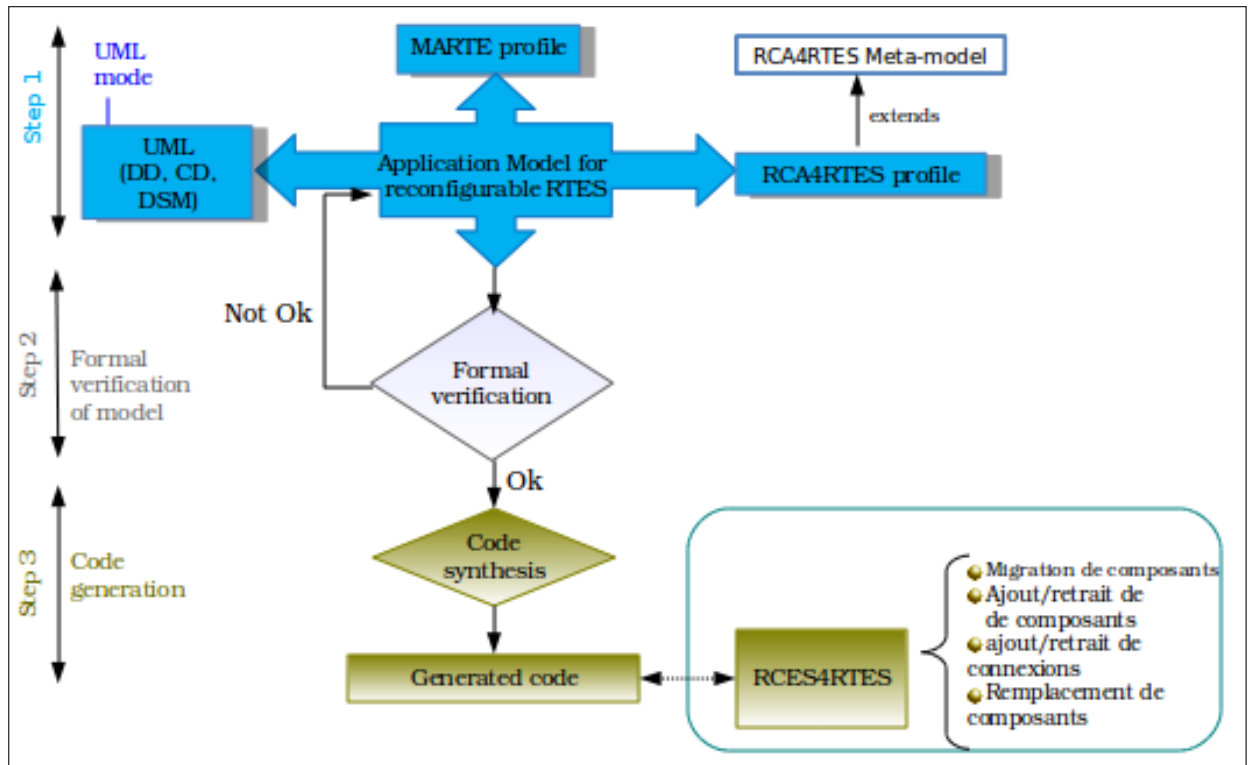


FIGURE 3.1 – Cadre du travail

(Reconfigurable Computing Architecture for Real Time Embedded Systems). Il est obtenu de la combinaison d'AADL, de MARTE, ainsi que de notions empruntées à l'orienté composant [KHZC10]. Ce méta-modèle repose sur le concept de *MétaMode* qui est un ensemble de configurations décrites à l'aide de composants, de connecteurs et de contraintes (structurelles et non-fonctionnelles). À la différence des autres approches, l'utilisation de metaModes permet ici de capturer et de spécifier les configurations. Cela permet ainsi d'éviter d'avoir à définir chacune des configurations.

La reconfiguration dynamique de ces systèmes est alors décrite à l'aide de machines à états constituées d'un ensemble de metaModes et de transitions entre eux [KHZJ11].

3.2.2 Vérification formelle du modèle

Cette étape consiste à vérifier formellement quelques propriétés non fonctionnelles des systèmes TR²E dynamiquement reconfigurables, en prenant en entrée le

résultat de l'étape 1, c'est-à-dire : le modèle d'un système TR²E dynamique. Parmi les propriétés à vérifier formellement se trouvent : la consommation mémoire, l'utilisation de la bande passante, le respect des échéances ainsi que la présence d'interblocage [Gas11].

Ainsi, si la vérification de ces propriétés est réalisée avec succès, la troisième étape est effectuée. Sinon, un nouveau modèle répondant mieux aux spécifications doit être généré. Il s'agira alors de retourner à l'étape 1.

3.2.3 La génération du code

Une fois que le modèle généré a été vérifié formellement avec succès, la troisième étape peut alors être effectuée. Celle-ci comporte deux volets. Le premier volet correspond à notre travail de maîtrise qui vise à concevoir un intergiciel supportant les mécanismes de reconfiguration dynamique dans les systèmes TR²E. RCES4RTES est l'intergiciel que nous avons conçu pour répondre à ces besoins. Dans la suite de ce chapitre, nous donnerons de plus amples détails sur son développement.

Le second volet va consister à synthétiser et à générer automatiquement du code à partir de notre intergiciel. Il fera l'objet de travaux ultérieurs.

3.3 Environnements et langages de programmation

Ocarina[VZH11] est une suite d'outils qui sert à la génération, au déploiement, ainsi qu'à la configuration des applications TR²E et ce, à partir de leur modèle AADL. Elle permet également la production automatique de code à partir d'une modélisation AADL vers différents intergiciels. C'est dans cette optique que nous l'utiliserons pour générer automatiquement la partie fortement personnalisables des services canoniques de notre intergiciel.

Par ailleurs, nous avons choisi d'implanter les autres services de notre intergiciel avec le RTSJ qui offre des garanties temps réel à l'exécution des tâches.

3.3.1 Ocarina : description architecturale

Projet lancé en 2003 par l'équipe S3 de TELECOM ParisTech, Ocarina comprend trois générateurs de code : le premier pour le langage C, le second pour le langage

ADA et le troisième pour le Java temps réel.

Les trois bibliothèques principales sur lesquelles repose l'architecture d'Ocarina sont :

- **une bibliothèque centrale** dédiée à la construction, ainsi qu'à la manipulation des arbres syntaxiques,
- **des composantes frontales** qui utilisent les fonctionnalités de la bibliothèque centrale à des fins d'analyse syntaxique et sémantique de fichiers AADL,
- **un ensemble de composantes dorsales** permettant la vérification du modèle et la génération de code.

La figure 3.2 propose une illustration des trois bibliothèques intégrées dans la suite d'outils Ocarina.

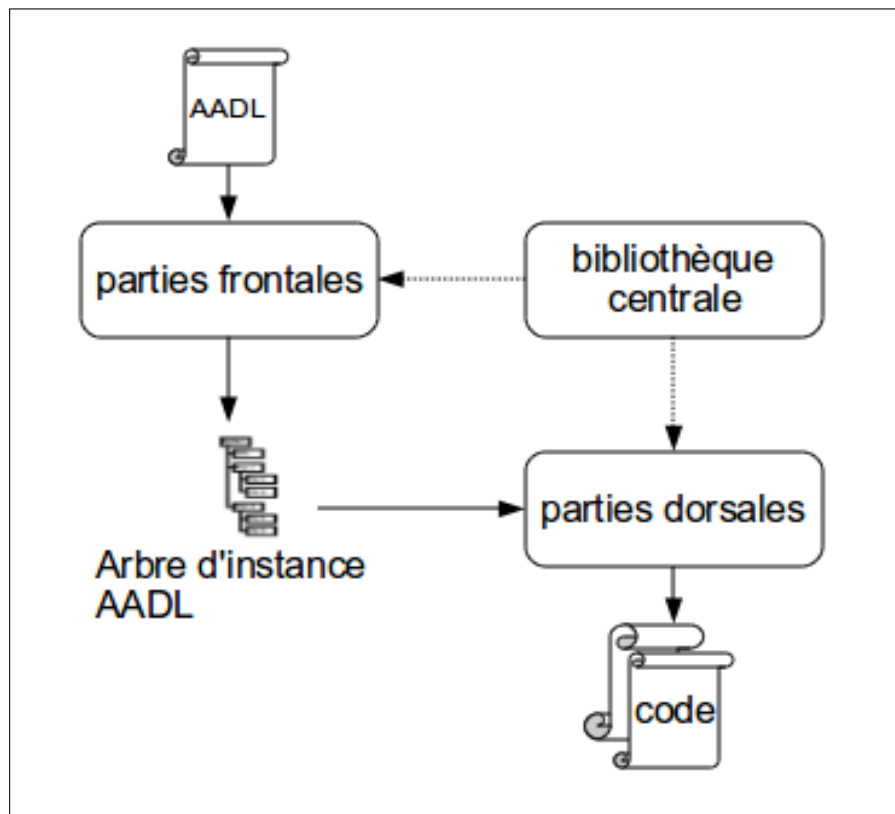


FIGURE 3.2 – Architecture d'Ocarina

Bibliothèque principale

Elle fournit les routines nécessaires pour manipuler un arbre correspondant à un formalisme donné. Elle est également complétée par des routines propres aux

formalismes qu'elle supporte (AADL, C, ADA, etc). Ces routines vont permettre aux composantes frontales et dorsales d'effectuer l'échange et la manipulation des arbres d'un formalisme donné.

Composante frontale

Prenant en entrée des fichiers écrits en AADL, elle effectue une analyse syntaxique et sémantique de ceux-ci, afin de produire un arbre syntaxique abstrait qui reflète la structure de ces fichiers. La composante frontale intègre ainsi :

- **un analyseur lexical** générant des lexèmes (opérateurs, identificateurs, séparateurs, etc) à partir d'un fichier AADL pris en entrée,
- **un analyseur syntaxique** gérant l'analyseur lexical et effectuant la transformation des lexèmes en arbre syntaxique abstrait. Il permet de détecter les erreurs syntaxiques,
- **un analyseur sémantique** chargé de la vérification sémantique du modèle et de la décoration de l'arbre syntaxique à l'aide d'opérations facilitant sa navigation,
- **l'instanciation** permettant la génération d'un arbre d'instance qui prendra en considération les spécificités de chacun des composants applicatifs.

Composante dorsale

Une composante dorsale prend en entrée le résultat de la composante frontale, à savoir : l'arbre syntaxique décoré. Elle va ensuite étendre cet arbre en simplifiant ses constructions. Le nouvel arbre ainsi obtenu subit alors une transformation en un arbre syntaxique correspondant au langage choisi (ADA, RTSJ, C). Cet arbre sera ensuite parcouru pour générer du code.

3.3.2 Les langages de programmation

Afin de développer notre intergiciel, nous utiliserons le langage de programmation RTSJ [BB09] qui dispose de fonctionnalités adaptées aux systèmes temps réel. Notre choix s'est porté sur ce langage car il est capable de gérer tous les types de threads que notre intergiciel souhaite implanter (threads périodiques, sporadiques, hybrides et aperiodiques), et ce en leur garantissant le respect de leurs échéances temporelles. Par ailleurs, le fait que l'allocation de la mémoire limite le recours au ramasse miettes

(*garbage collector*) constitue également un solide atout. En effet, cette limitation va permettre d'éviter l'indéterminisme dû à l'action imprévisible du ramasse miettes.

3.4 PolyORB_HI

Inspiré de l'architecture des intergiciels schizophrènes, PolyORB_HI est un intergiciel qui permet de déployer et de configurer les applications TR²E.

3.4.1 Description de l'architecture de PolyORB_HI

PolyORB_HI est constitué des services canoniques définis par l'architecture des intergiciels schizophrènes. Afin de mieux séparer les préoccupations, il divise certains de ces services en deux et répartit les nouveaux services obtenus en deux familles. La première famille est constituée des services qui sont communs à toute application. La seconde famille comprend les services qui sont personnalisés en fonction des besoins de l'application cible. La mise en œuvre de ces services est décrite dans la suite de cette section.

3.4.1.1 Mise en œuvre des services faiblement personnalisables

Les services faiblement personnalisables sont génériques à toute application. Leur implantation correspond à un code écrit manuellement [Zal08]. La description de leur implantation est proposée dans ce qui suit.

Mise en œuvre du parallélisme

L'implantation du service parallélisme correspond aux méthodes permettant de créer les différentes catégories de tâches supportées par l'intergiciel, à savoir : les tâches périodiques, sporadiques et hybrides. Ces tâches possèdent entre autres les propriétés suivantes :

- l'échéance de la tâche (déduite suite à l'analyse du modèle de l'application),
- la priorité de la tâche (déduite grâce à une analyse du modèle de l'application),
- la période de la tâche (s'il s'agit d'une tâche périodique ou hybride), ou la durée minimale qui s'écoule entre deux déclenchements successifs de la tâche (s'il s'agit d'une tâche sporadique). Cette propriété est également déduite du modèle,

- la taille de la pile de la tâche, qui est déduite à partir du modèle,
- le sous-programme chargé de bloquer la tâche en attente d'un évènement externe (cas d'une tâche hybride ou sporadique). Ce sous-programme retourne le port qui a reçu l'évènement en question,
- le sous-programme qui décrit le travail que la tâche effectue à chaque cycle,
- le sous-programme qui est éventuellement appelé après que la tâche ait été initialisée.

Comme mentionné dans la section 1.4.2.3, l'existence d'au moins une tâche hybride sur un nœud nécessite le recours à une tâche supplémentaire permettant le réveil des tâches hybrides, une fois que leur période est arrivée. Cette tâche est implantée grâce aux paramètres suivants :

- un tableau fournissant des renseignements sur toutes les tâches hybrides du système (identificateurs, périodes, ports sur lesquels elles reçoivent les évènements périodiques),
- sa priorité qui correspond à la valeur maximale des priorités des tâches hybrides existant sur le nœud,
- la taille de la pile de la tâche,
- la routine de la couche haute de transport permettant de délivrer les évènements périodiques aux tâches hybrides.

Comme indiqué plus haut, le parallélisme propose des routines permettant de bloquer ou de débloquent une tâche donnée. Ces routines permettent d'activer simultanément toutes les tâches, une fois que l'application a été initialisée.

Les routines offertes par le parallélisme sont liées au composant **Deployment**. Ce composant fournit une représentation de la structure de l'application et associe chaque tâche à son identifiant. Sa génération est automatique.

Le composant **Activity** contient quant à lui toutes les instances des tâches nécessaires au fonctionnement de l'application. Il est aussi généré automatiquement.

Mise en œuvre de la représentation élémentaire

Celle-ci fournit les mécanismes permettant d'emballer et de débloquent une donnée dans un message. Dans le cas où les nœuds de l'application répartie se trouvent sur des plates-formes homogènes, les données sont emballées par copie mémoire. Dans le cas échéant, des méthodes plus complexes pourront être utilisées. Celles-ci pourront

par exemple faire intervenir l’alignement des données en mémoire ou le boutisme de la plate-forme qui envoie des données.

En plus des routines d’emballage et de déballage des types de données élémentaires, l’implantation de la représentation élémentaire fournit également deux routines communes à toute application distribuée. En effet, la première routine qu’elle propose permet d’emballer les données temporelles correspondant à l’instant où un message est reçu sur un port destination. La seconde routine permet quant à elle d’emballer l’identificateur correspondant à un port qui envoie ou reçoit des données.

Mise en œuvre de l’interrogation

L’interrogation est le service de base autour duquel s’articule l’intergiciel minimal. L’implantation de ce service permet à chaque composant AADL de type **thread** d’effectuer de façon transparente l’instanciation de mécanismes d’envoi et de réception de données au sein du composant **Activity**. Cette implantation propose ainsi des routines utiles à :

- l’envoi d’un message sur un port destination,
- la lecture d’un message reçu,
- la détermination de l’identité de l’expéditeur d’un message,
- le calcul du nombre de messages stockés dans la file d’attente d’un port de type événement,
- le blocage d’une tâche en attente de l’arrivée d’évènements.

Mise en œuvre du protocole

L’implantation de ce service correspond à la mise en place d’une routine permettant l’envoi de messages entre deux entités de l’application. Cette routine prend ainsi en paramètres : l’entité expéditrice, l’entité destinataire, ainsi que le message à acheminer. Elle fait alors appel à la routine d’envoi de la couche haute de transport, à qui elle transmet le message qui doit être acheminé vers l’entité destinataire.

Mise en œuvre de la couche basse de transport

L’implantation de ce service dépend étroitement du système d’exploitation utilisé, ainsi que du matériel employé pour assurer le transport des données (nature de la couche physique, bus de communication, etc). L’intergiciel minimal peut inclure

les implantations de différentes couches de transport. Chacune de ces implantations est indépendante de celles des autres. La couche haute de transport se charge ensuite de sélectionner l'implantation de la couche basse de transport qui répond le mieux aux caractéristiques de l'application.

L'implantation d'une couche basse de transport est réalisée grâce à :

- une routine permettant l'initialisation des processus légers, ainsi que des structures de données requises par la couche basse de transport. Cette routine permet également l'ouverture des canaux de communication nécessaires au dialogue entre les nœuds de l'application distribuée,
- une routine permettant d'envoyer un tampon de communication entre deux nœuds de l'application distribuée,
- une routine chargée de recevoir les messages provenant du réseau. Celle-ci a recours à une tâche qui est bloquée en attente de la réception de messages. Lorsqu'un nouveau message est reçu, la routine Deliver de la couche haute de transport est invoquée. Cette routine permet de délivrer le message à son destinataire.

3.4.1.2 Mise en œuvre des services fortement personnalisables

Les services fortement personnalisables sont générés automatiquement en se basant sur les propriétés de l'application. À cet effet, la modélisation d'applications TR²E a été effectuée avec le langage AADL pour lequel il existe des outils supportant ce processus de production automatique.

Production automatique des applications TR²E

Une application TR²E peut-être générée automatiquement à partir de son modèle AADL. Le processus de production automatique correspondant s'opère selon les quatre étapes suivantes :

- **l'analyse** consistant à vérifier la cohérence du modèle AADL. Il s'agit ici d'établir si le modèle est conforme sémantiquement et syntaxiquement parlant aux normes du modèle AADL,
- **la production automatique de code** permettant la génération automatique des composants intergiciels fortement personnalisables, ainsi que celle des composants applicatifs. La première famille de composants est ainsi géné-

rée à partir de la topologie de l'application. La seconde famille est quant à elle produite grâce au modèle de l'application,

- **le déploiement et la configuration** constituent une étape qui revient à choisir les composants de l'intergiciel minimal qui sont requis pour un nœud cible. Ce processus est rendu automatique grâce aux informations de déploiement récupérées du modèle. La configuration des composants choisis, ainsi que de ceux produits automatiquement est alors effectuée selon les caractéristiques du nœud. Cette étape permet ainsi l'obtention de composants dédiés à l'application,
- **l'intégration** consiste à collecter les composants applicatifs obtenus à la deuxième étape. Elle permet la génération d'exécutables pour un nœud donné de l'application distribuée. Au terme de cette étape, l'application est désormais exécutable.

Génération des services fortement personnalisables

Une fois que le modèle AADL a été analysé (cf. première étape du processus décrit plus haut), les services fortement personnalisables peuvent alors être générés automatiquement.

Le service **typage** est obtenu de la production automatique des types de données requis dans chacun des nœuds.

La génération du service **interaction** est garantie par la production de toutes les méthodes nécessaires pour accéder au composant de type **thread**.

L'**exécution** et l'**activation** sont des services générés statiquement. En effet, c'est à l'initialisation de l'application que sont créées toutes les tâches devant se trouver sur un nœud donné. En outre, le sous-programme décrivant le comportement d'une catégorie de tâches est généré automatiquement. Ce sous-programme sera invoqué automatiquement à chaque déclenchement d'une tâche appartenant à cette catégorie.

L'**adressage** et la **liaison** doivent permettre de retrouver en une durée fixe, les données requises pour qu'un nœud donné atteigne les nœuds avec lesquels il interagit. Pour ce faire, une table de nommage/adressage est créée. Elle va permettre aux nœuds de l'application distribuée de dialoguer. En effet, pour chaque port d'entrée du nœud, cette table fournit les données requises afin d'initialiser l'élément qui recevra

les messages. Cette table fournit aussi les informations de transport permettant à un nœud donné d'atteindre des ports situés sur d'autres nœuds. La création d'une table de nommage/adressage pour chacune des couches basses de transport utilisées par le nœud servira à initialiser ces couches.

La **couche haute de transport** est quant à elle constituée des routines produites automatiquement pour chacun des nœuds :

- la routine **INIT** chargée d'initialiser l'ensemble des couches basses de transports nécessaires à l'exécution d'un nœud donné,
- la routine **SEND** chargée d'expédier un message donné. Elle a pour paramètres le port d'entrée ainsi que le port destinataire entre lesquels elle achemine le message auquel elle ajoute une en-tête. Dans le cas d'une communication distante, elle invoque la routine de la couche basse de transport qui permet l'envoi d'un message,
- la routine **DELIVER** permettant de délivrer un message à son destinataire. Elle a pour arguments l'entité destinataire, ainsi que le message qui doit lui être délivré. Dans le cas d'une communication locale, cette routine est appelée par la couche basse de transport. Sinon, elle est invoquée par la routine de la couche basse de transport qui gère la réception des messages.

La **représentation avancée** est très proche du service typage, puisqu'elle s'assure que les données à envoyer sont cohérentes. Le type des informations échangées est connu grâce à une analyse des composants de type **thread**. Les routines ci-dessous sont automatiquement produites pour chacun des types de données échangées :

- la routine **Marshall** permettant d'emballer les données,
- la routine **Unmarshall** utilisée pour déballer les données.

3.4.2 Gestion locale de la communication

PolyORB_HI étant conforme au profil Ravenscar, le dialogue inter tâches se limite donc aux objets protégés. En outre, l'usage de ce profil requiert que chaque tâche réalise une attente unique en vue de la réception de nouveaux éléments sur ses différents ports. À cet effet, le mécanisme permettant de respecter cette dernière contrainte est mis en place à l'aide d'un tableau formé grâce à la concaténation de tableaux circulaires. Ces derniers correspondent chacun à la file d'attente d'un port

d'entrée d'une tâche, et dont la taille est précisée par l'utilisateur. La concaténation de ces tableaux permet ainsi d'obtenir la représentation d'une file unique contenant les messages destinés à la tâche. Un second tableau est alors utilisé pour renseigner sur l'ordre d'arrivée des messages dans la file. Il est lui aussi circulaire.

3.5 RCES4RTES

Reposant sur une architecture inspirée de celle des intergiciels schizophrènes, RCES4RTES est donc formé par les neuf services canoniques. Par ailleurs, ayant été conçu pour les systèmes TR²E dynamiques, il se propose de gérer l'aspect critique de ces systèmes, ainsi que d'assurer leur cohérence, une fois qu'ils ont été reconfigurés. Dans la suite de cette section, nous décrivons la démarche adoptée afin de mener à bien son développement.

3.5.1 Choix de l'implémentation

PolyORB_HI est doté d'une empreinte mémoire beaucoup plus faible que la plupart de celles des intergiciels adaptés comme lui aux systèmes TR²E. De plus, il permet de respecter les échéances des systèmes temps réel durs, tout en leur assurant d'être portables. Par ailleurs, le fait qu'il soit constitué par les neuf services canoniques définis par l'architecture des intergiciels schizophrènes, permet une séparation des préoccupations, et le rend dédié à l'application cible. Toutes ces caractéristiques font de lui un candidat idéal pour les systèmes TR²E.

Conçu pour les systèmes TR²E statiques, PolyORB_HI comporte par conséquent quelques insuffisances qui l'empêchent d'être adapté à la reconfiguration dynamique. Nous pouvons ainsi relever les limites suivantes au sein de cet intergiciel :

- **absence de mécanismes de reconfiguration dynamique** : PolyORB_HI a été conçu à la base pour des systèmes statiques. C'est la raison pour laquelle il n'offre pas de fonctionnalités permettant à ceux-ci de s'adapter à leur environnement d'exécution, ou même d'évoluer,
- **manque de réflexivité** : utilisée pour observer l'exécution du système en vue de sa modification éventuelle, la réflexivité n'est pas prise en charge dans PolyORB_HI. En effet, ce dernier ne supporte pas la reconfiguration dynamique,

- **absence de mécanismes de gestion de la cohérence** : puisque PolyORB_HI n’implante pas les mécanismes permettant de modifier les systèmes, il ne prend par conséquent pas en charge la gestion de la cohérence nécessitée par leur modification.

Aussi, afin d’éviter de réinventer la roue, nous allons développer RCES4RTES en tant qu’extension de PolyORB_HI. Il s’agira alors de compléter ce dernier afin de lui rajouter des mécanismes de reconfiguration dynamique, de réflexivité, ainsi que des fonctionnalités lui permettant de prendre en charge la cohérence du système reconfiguré.

3.5.2 Description de l’architecture détaillée de RCES4RTES

Notre intergiciel a pour objectif de permettre la reconfiguration dynamique des systèmes TR²E. Son implantation prend en charge trois types de composants, à savoir : les composants de type **process**, ceux de type **thread**, ainsi que ceux de type **data**. Ils correspondent respectivement aux nœuds de l’application, aux processus légers, ainsi qu’aux données échangées entre ces processus. Dans la suite, nous considérerons que seuls les processus légers sont associés à des ports qui leur permettent d’interagir entre eux.

3.5.2.1 Modification des structures de données

Les structures de données qui ont servi au développement de PolyORB_HI sont statiques, puisque l’intergiciel en question est destiné aux applications TR²E statiques. Aussi, pour des raisons liées à la reconfiguration dynamique, ces structures de données statiques doivent être remplacées par d’autres structures de données à caractère dynamique.

Ainsi, les tables de nommage/adressage qui ont permis l’implantation du service adressage étant statiques, elles doivent être remplacées par des tables de hachage de nommage/adressage. Celles-ci vont notamment permettre de gérer les références des entités ajoutées dynamiquement au système.

Par ailleurs, la reconfiguration dynamique requiert parfois la connexion ou la déconnexion de deux nœuds distants de l’application répartie. À cet effet, les structures de données (tables) utilisées par la couche basse de transport afin de permettre

l'interconnexion des nœuds, doivent devenir dynamiques. Dans ce cas aussi, l'usage de tables de hachage nous semble beaucoup plus approprié.

3.5.2.2 Mise en œuvre du processus de reconfiguration dynamique

Notre intergiciel repose sur l'architecture de PolyORB_HI qu'il étend afin de la rendre adaptée aux systèmes TR²E dynamiques. Dans cette section, nous décrivons donc les fonctionnalités que nous lui avons rajoutées afin de pouvoir supporter les mécanismes de reconfiguration dynamique dans le langage RTSJ.

Mise en œuvre de la réflexivité

La réflexivité (introspection) permet à un système donné d'observer sa propre exécution. Dans le cas de RCES4RTES, sa mise en œuvre s'articule autour de trois routines principales. Celles-ci pourront être invoquées par l'utilisateur afin de mieux cibler l'action de reconfiguration dynamique voulue.

En effet, la première routine nommée **number_instances** permet de déterminer le nombre d'instances de tâches d'un type donné et qui s'exécutent sur le nœud courant. Cette routine a pour argument le type de tâches dont on veut déterminer le nombre d'instances correspondant. Elle a recours à une table de hachage dont la clé est l'identificateur d'une tâche à laquelle est associée son type de tâches. Cette table de hachage est créée à l'initialisation de l'application.

Ainsi, à chaque fois qu'un composant est ajouté ou retranché de la plate-forme d'exécution, son entrée dans la table de hachage est mise à jour. La routine initialise au préalable un compteur qui va contenir le nombre d'instances de composants correspondant au type passé en paramètre. La table de hachage est ensuite parcourue. Ainsi, à chaque fois que la valeur associée à l'une de ses clés correspond au type de composants passé en argument, le compteur est alors incrémenté. Au terme de cette routine, la valeur contenue dans le compteur peut alors être retournée à l'utilisateur.

Dénommée **number_connections**, la seconde routine permet quant à elle de déterminer le nombre de connexions reliant une tâche donnée aux autres tâches avec lesquelles elle interagit. Cette routine a pour argument le composant dont le nombre de connexions doit être calculé. Elle initialise un compteur à zéro. Ce der-

nier contiendra le nombre de connexions à déterminer. La routine parcourt ensuite l'ensemble des ports de cette tâche. Ainsi, si un port donné est un port de sortie, elle récupère le nombre de ports destinations qui lui sont associés et incrémente le compteur à hauteur de ce nombre. Sinon, s'il s'agit d'un port d'entrée, elle récupère le nombre de ports qui lui envoient des données et augmente la valeur du compteur par ce nombre. La routine peut, enfin, retourner le contenu du compteur.

observation_variables est la routine créée pour répertorier les plus récents instants d'accès à une variable partagée de l'application. Pour ce faire, elle prend en paramètre une variable partagée, ainsi que la nature de l'accès à cette variable (lecture, écriture, lecture/écriture). Cette routine utilise une table de hachage créée à l'initialisation de l'application et ayant pour clés les différentes variables partagées de l'application. Cette table associe à chacune de ses clés, un couple formé par le plus récent temps d'accès en lecture à une variable partagée, ainsi que par le plus récent temps d'accès en écriture de cette variable.

Ainsi, la routine vérifie tout d'abord si la table de hachage contient effectivement la variable partagée prise en argument. Si tel est bien le cas, elle récupère ensuite l'instant présent. Elle met alors la table de hachage à jour, en affectant cet instant au plus récent temps d'accès en lecture, au plus récent temps d'accès en écriture, ou simultanément à ces deux temps, et ce en fonction de la nature de l'accès à la variable partagée.

Mise en œuvre de la cohérence

Une action de reconfiguration peut mettre à mal la cohérence d'un système, car elle peut provoquer des interférences avec les actions en cours entre les composants. Cela comporte notamment le risque d'obtenir un système inutilisable. Pour y remédier, nous proposons des routines qui pourront garantir la cohérence du système, une fois qu'il a été reconfiguré.

La routine **blockTask** permet ainsi de bloquer la tâche à reconfigurer et au besoin, ses sources. Pour cela, elle prend en argument une structure de données permettant à la tâche à reconfigurer d'interagir avec les autres composants. Cette structure de données est formée d'un triplet contenant la tâche à reconfigurer, la file d'attente de

cette tâche, ainsi que le buffer qui va permettre à chaque fois de stocker le message à envoyer. Le second paramètre de cette routine est une variable booléenne qui indique s'il faut aussi bloquer les sources de ladite tâche. Si cette variable booléenne a la valeur *TRUE*, alors la routine parcourt un à un les composants sources en se réinvokant elle-même, prenant cette fois-ci la valeur *FALSE* comme variable booléenne en paramètre.

Ceci va permettre d'éviter que le blocage des sources n'engendre le blocage de toutes les autres tâches de l'application. La routine attend ensuite que la file d'attente de la tâche à reconfigurer soit vide, afin d'appliquer un verrou à la structure de données. Lorsque ce verrou est placé, l'activité du composant est stoppée. Cependant, si la tâche est périodique, il faut également appliquer un verrou sur la tâche elle-même.

unblockTask est quant à elle une routine qui, comme son nom l'indique, permet de débloquer une tâche à reconfigurer et éventuellement, ses sources. Elle prend en paramètre une structure de données qui permet à la tâche reconfigurée d'interagir avec les autres composants. Elle a comme autre paramètre une variable booléenne indiquant s'il faut débloquer les sources de cette tâche (dans le cas où elles ont été bloquées). Le verrou précédemment appliqué à la structure de données est alors relâché. De même pour celui qui avait été appliqué à la tâche, si cette dernière est périodique.

La tâche reconfigurée peut alors reprendre son activité. Dans le cas où la variable booléenne a la valeur *TRUE*, la routine est alors réinvokée pour chaque source de la tâche, en prenant cette fois-ci la valeur *FALSE* comme variable booléenne en paramètre. Ceci permet aux sources d'être à leur tour débloquées.

Mise en œuvre de la reconfiguration dynamique

La reconfiguration dynamique est un moyen efficace pour s'assurer qu'un système s'adapte ou même évolue en fonction des exigences liées à son environnement d'exécution, tout en restant disponible. Pour des raisons liées au respect des échéances temporelles, nous allons déclencher chaque action de reconfiguration dynamique à l'aide d'une tâche possédant une échéance temporelle à respecter. Cette tâche sera créée sur chaque nœud de l'application distribuée. Elle pourra alors permettre d'in-

roduire des modifications au sein du nœud sur lequel elle est créée, et ce en fonction du choix de l'utilisateur. Cette tâche de reconfiguration pourra ensuite informer les autres tâches de reconfiguration (situées sur les nœuds distants) de l'action de reconfiguration qu'elle a mené. Ces dernières pourront ainsi mettre à jour leurs informations de déploiement. Les différentes routines de reconfiguration dynamique que nous proposons alors sont les suivantes :

La routine **addConnection** permettant de créer une connexion entre deux ports afin de permettre le dialogue entre les deux composants auxquels ils sont respectivement associés. Cette routine ne sera appliquée qu'aux processus légers. En effet, comme les nœuds et les données échangées ne sont pas associés à des ports, ils ne seront donc pas pris en considération. De plus, comme un processus léger ne peut être imbriqué dans un autre processus léger, le cas de composants imbriqués pris en compte dans l'algorithme 6 ne sera donc pas implanté dans cette routine.

addConnection prend en paramètre deux ports associés respectivement aux tâches à connecter. Afin de se conformer au standard AADL, la connexion doit être orientée du premier port vers le second port. Aussi, la routine vérifie si le premier port est un port de sortie et si le second port est un port d'entrée. Si c'est effectivement le cas, elle s'assure que la tâche associée au premier port soit bloquée. Il est en effet possible que la seconde tâche ait besoin des messages que la tâche bloquée pourrait émettre pendant que la connexion est en train d'être établie. L'identificateur du second port est ensuite ajouté à la liste des identificateurs de ports auxquels le premier port envoie des données. Cette liste est stockée dans un tableau de taille variable. Réciproquement, l'identificateur du premier port est aussi ajouté à la liste des identificateurs de ports qui envoient des données au second port. Cette liste est également maintenue dans un tableau de taille variable. Une fois ces ajouts effectués, La tâche peut alors être débloquée.

removeConnection est une routine proposée pour supprimer une connexion entre deux ports. Elle ne s'applique également qu'aux processus légers, et ce sans tenir compte de l'imbrication de composants. La routine s'assure au préalable que le premier port est un port de sortie et que le second est un port d'entrée. Si c'est effectivement le cas, elle bloque ensuite la tâche associée au premier port. Ce blocage

va permettre de s'assurer que la connexion à supprimer n'est pas en cours d'utilisation. L'identificateur du second port est ensuite retiré de la liste des identificateurs de ports auxquels le premier port envoie des données. De même, l'identificateur du premier port est également retiré de la liste des identificateurs de ports qui envoient des données au second port. La tâche associée au premier port est ensuite débloquée.

connect est la routine qui a été implantée dans PolyORB_HI RTSJ afin de permettre l'établissement d'une connexion entre le nœud courant et un nœud distant. Cette routine est utilisée à l'initialisation de la couche basse de transport afin d'interconnecter le nœud courant avec tous les autres nœuds. Ceci s'avère coûteux en ressources. Pour y remédier, nous nous proposons d'interconnecter le nœud courant uniquement avec l'ensemble des nœuds nécessaires à son exécution. À cet effet, nous stockons dans un tableau l'ensemble des nœuds avec lesquels le nœud courant a besoin de communiquer. Nous modifions alors la routine **connect** afin qu'elle puisse mettre à jour ce tableau en y ajoutant le nœud distant avec lequel la connexion est établie. Ainsi, à l'initialisation de la couche basse de transport, la routine **connect** sera invoquée en fonction des nœuds stockés dans ce tableau.

La routine **disconnect** sert à déconnecter le nœud courant d'avec un nœud distant qu'elle prend en argument. Elle requiert que ceux-ci soient distincts et que le nœud distant soit associé à une adresse réseau. Cette dernière condition indique que le nœud distant est appelable à distance. Cette routine nécessite aussi que le nœud courant soit repertorié dans le tableau contenant l'ensemble des nœuds interconnectés au nœud courant. Si toutes ces conditions sont remplies, le flux par lequel le nœud courant envoie des données au nœud distant est alors fermé. La routine vérifie également si le nœud courant est aussi associé à une adresse réseau. Si tel est effectivement le cas, le flux permettant au nœud courant de recevoir des données issues du nœud distant est également fermé. Le nœud distant est alors retiré de l'ensemble des nœuds reliés au nœud courant.

changeProperties est une routine qui propose de modifier certaines propriétés d'une tâche donnée. Elle prend ainsi en paramètres un tableau contenant les paires (nom de propriété, valeur de la propriété), ainsi qu'une structure de données formée du triplet déjà explicité plus haut. La routine divise au préalable le tableau

de propriétés en deux autres tableaux. Le premier tableau contient les propriétés non critiques (pire temps d'exécution, priorité). Le second tableau est formé des propriétés critiques. Il s'agit par exemple du verrou d'une tâche périodique dont la modification peut bloquer la tâche. Cette dernière est alors incapable de traiter ses requêtes et bloque les autres tâches en attente de celles-ci.

Ainsi, **changeProperties** affecte aux propriétés de la tâche, les valeurs des propriétés du premier tableau. Puis, la tâche est bloquée et les propriétés critiques affectées. La tâche est ensuite débloquée. Signalons que si l'une des propriétés contenues dans l'un des tableaux n'existe pas chez la tâche, cette propriété est alors créée au niveau de la tâche.

Pour ajouter un nouveau composant sur la plate-forme d'exécution, la routine **addTask** est proposée. Elle prend en paramètres les composantes nécessaires à la création d'une tâche : type, catégorie, identifiant, période, échéance, liste des ports destinations, liste des ports qui envoient des données, etc. La catégorie indique le genre de tâches qui peuvent exister sur un nœud donné (périodique, sporadique, etc). Le type renvoie quant à lui au rôle joué par le composant au sein une application donnée. À l'initialisation de l'application, une table de hachage associant chaque nœud aux types de tâches pouvant y exister, est créée.

Ainsi, la routine vérifie d'abord si le type de tâches passé en paramètre, existe bel et bien dans la table de hachage. Si c'est effectivement le cas, une instance de tâches correspondant au type spécifié est créée. Les données de déploiement relatives à la tâche sont également enregistrées. Sont alors créées les connexions vers les ports qui reçoivent et vers les ports qui envoient des données à la tâche. Remarquons qu'au moment où la routine est invoquée, les ports qui doivent être associés à la tâche à créer existent déjà.

removeTask est une routine permettant de supprimer une tâche sur un nœud donné. Pour ce faire, elle prend en argument le nœud sur lequel la tâche doit être supprimée, ainsi qu'une structure de données formée du triplet mentionné plus haut. La routine commence par supprimer toutes les connexions qui permettent à la tâche de recevoir des données. La tâche est ensuite bloquée afin d'achever le traitement de ses requêtes en cours et d'interrompre son activité. Les connexions qui permettent à

la tâche d'envoyer des données dont alors supprimées. Une fois que les informations relatives au déploiement de la tâche ont été supprimées, celle-ci est alors détruite.

Nous proposons la routine **migrateTask** afin de migrer une tâche d'un nœud origine vers un nœud destination. À cet effet, cette routine prend en paramètres le nœud sur lequel s'exécute la tâche à migrer, le nœud vers lequel celle-ci doit être migrée, ainsi qu'une structure de données formée du triplet évoqué précédemment. Ainsi, après s'être assurés que les deux nœuds sont distincts, nous bloquons la tâche ainsi que ses sources. Ceci garantit que la file d'attente de la tâche soit vide avant qu'elle ne soit interrompue. Ensuite, nous sauvegardons dans un tableau l'état de la tâche. Il correspond aux valeurs des propriétés de la tâche. Ce tableau est alors sérialisé en un flux d'octets qui est emballé dans un tampon de communication et envoyé au nœud destination.

Une fois que ce dernier a reçu le tampon, un flux d'octets est alors obtenu du déballage de ce tampon sur le nœud destination. Ce flux est ensuite désérialisé en un tableau correspondant à l'état de la tâche initiale. Une nouvelle tâche est alors créée sur le nœud destination, et ce à partir des valeurs contenues dans le tableau. La nouvelle tâche est alors démarrée sur le nœud destination, tandis que la tâche initiale est supprimée du nœud origine.

replaceTask est la routine que nous avons implantée afin de pouvoir remplacer une tâche s'exécutant sur un nœud donné par une autre tâche. Ses arguments sont : le nœud sur lequel s'exécute la tâche à remplacer, le nœud où s'exécutera la tâche remplaçante, un tableau contenant les propriétés devant servir à créer cette dernière, ainsi qu'une structure de données formée du triplet mentionné précédemment. Ainsi, si les deux nœuds coïncident, nous créons une nouvelle tâche sur le même nœud que la tâche à remplacer. Cette dernière est ensuite bloquée afin d'achever le traitement de ses requêtes, puis de stopper son activité. Nous lançons ensuite l'exécution de la nouvelle tâche. Celle-ci correspond aux propriétés du tableau donné en argument. La tâche initiale est ensuite supprimée.

Par contre, si les nœuds sont distincts, nous sérialisons le tableau de propriétés afin d'obtenir un flux d'octets. Ce flux est alors emballé dans un tampon de communication qui est ensuite acheminé vers le second nœud. À la réception du tampon,

un flux d'octets en est extrait. Ce dernier est alors désérialisé afin d'obtenir un tableau correspondant aux propriétés de la tâche remplaçante. Cette dernière est alors créée sur le second nœud, conformément aux propriétés stockées dans ce tableau. La nouvelle tâche est ensuite démarrée sur le second nœud. Puis, La tâche initiale est supprimée du nœud origine.

3.6 Conclusion

Notre intergiciel supporte les mécanismes de reconfiguration dynamique dans les systèmes TR²E. Pour ce faire, il effectue la reconfiguration dynamique en s'appuyant sur la réflexivité et la cohérence. En effet, la réflexivité permet de connaître l'état du système afin de mieux cibler l'action de reconfiguration à effectuer. La gestion de la cohérence permet quant à elle de s'assurer que l'exécution du système demeure correcte une fois qu'il a été reconfiguré.

Le chapitre suivant sera axé sur la présentation d'une étude de cas portant sur notre travail.

Chapitre 4

Étude de cas

4.1 Introduction

Les deux derniers chapitres ont porté sur la conception, ainsi que sur la mise en œuvre d'un support d'exécution adapté à la reconfiguration dynamique dans les systèmes TR²E. Nous allons à présent illustrer l'utilisation de notre intergiciel en recourant à une étude de cas. Cette dernière sera le lieu de l'application d'actions de reconfiguration dynamique au sein d'un système TR²E. Elle nous permettra ainsi de procéder à une évaluation pratique de l'intergiciel que nous avons développé.

4.2 Description de l'étude de cas

Un GPS ou Global Positioning System est un système de navigation radio qui fournit des signaux de navigation précis à tout endroit donné de la terre [KHZJ11]. Il permet ainsi aux utilisateurs de déterminer, depuis l'emplacement où ils se trouvent, le chemin à suivre afin de se diriger vers des lieux spécifiés en utilisant l'information fournie par des satellites en orbite. Pour ce faire, ces derniers envoient des signaux aux terminaux GPS possédés par les utilisateurs. Ces signaux contiennent des données nécessaires à la localisation, ainsi qu'à la synchronisation. À la réception d'un signal, le terminal GPS détermine le temps qu'a pris celui-ci pour lui parvenir. Ainsi, comme la vitesse d'émission du signal est connue (célérité de la lumière), la distance entre le satellite et le terminal GPS est alors déduite comme le produit du temps déterminé par la vitesse d'émission du signal. La synchronisation des horloges des satellites est quant à elle assurée par la base de contrôle. À cet effet, celle-ci envoie

et reçoit des informations de ces satellites afin de faire en sorte que leurs horloges soient conformes aux normes temporelles usuelles.

Le fonctionnement d'un GPS fait donc intervenir les trois éléments que sont le terminal GPS, le satellite GPS, ainsi que la base de contrôle. Le schéma 4.1 illustre ainsi l'architecture d'un GPS [ZH10].

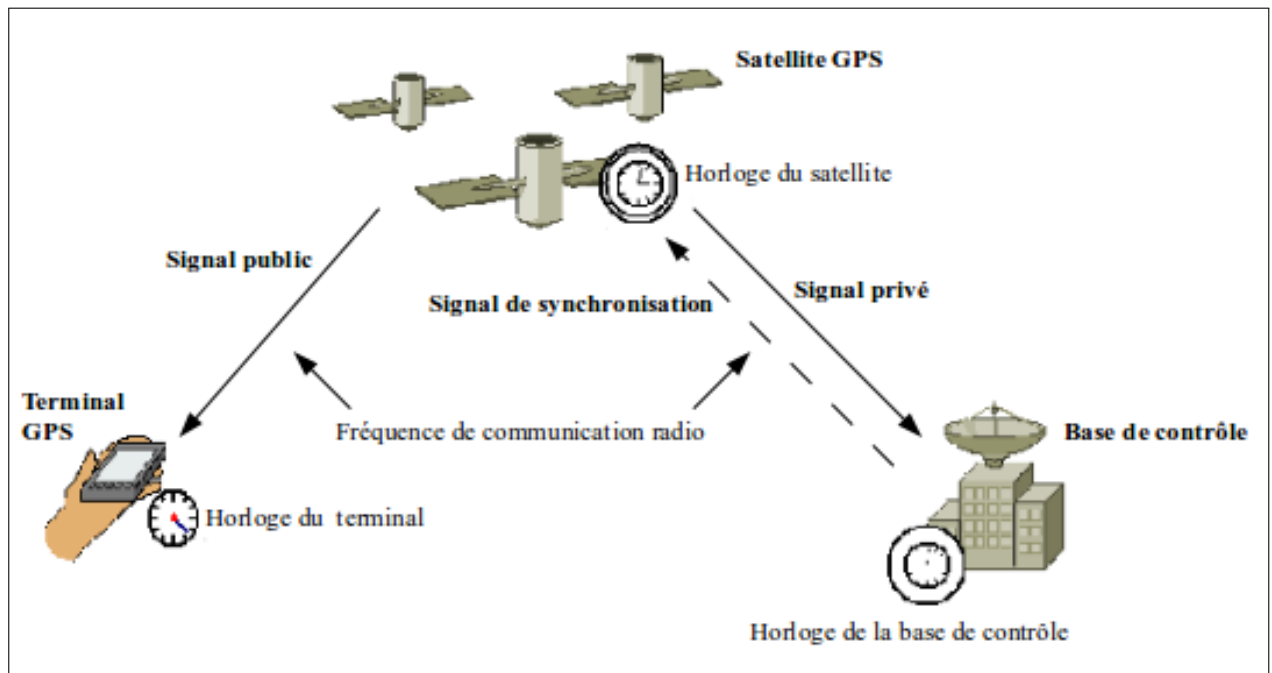


FIGURE 4.1 – Architecture d'un GPS

4.2.1 Propriétés communes aux composants d'un GPS

Les différents composants qui constituent un GPS possèdent les propriétés communes suivantes [KHZJ11] :

- **nature** indiquant s'ils sont périodiques, sporadiques, hybrides ou aperiodique,
- **période** définissant soit la période d'une tâche périodique, ou encore l'intervalle minimal de temps entre deux déclenchements successifs d'une tâche sporadique,
- **échéance** qui indique le délai d'exécution d'une tâche périodique ou sporadique,

- **instant de début** définissant l’instant à partir duquel une tâche aperiodique doit démarrer,
- **instant de fin** indiquant à quel moment une tâche aperiodique doit achever son execution,
- **WCET 1** définissant le pire temps d’execution pour une tâche periodique ou sporadique sur un processeur ayant une fréquence donnée. Il est le rapport du nombre d’instructions sur la fréquence du processeur,
- **WCET 2** représentant la constante du pire temps d’execution des tâches periodiques et sporadiques. Sa valeur est fixée et ne dépend pas du processeur. Il peut par exemple être un temps d’attente,
- **taille mémoire** renvoyant à l’espace de stockage requis pour l’execution d’une tâche.

4.2.2 Les MétaModes d’un GPS

Les différentes configurations d’un GPS peuvent être regroupées en trois MétaModes [KHZJ11], à savoir :

- le MétaMode initial correspondant à l’initialisation du GPS,
- le MétaMode non sécurisé consistant en une utilisation publique du GPS,
- le MétaMode sécurisé correspondant à une utilisation restreinte du GPS et qui s’accompagne d’exigences liées à la sécurité.

Le passage d’un MétaMode à l’autre est alors assuré grâce à des actions de reconfiguration dynamique. Dans la suite de notre travail, nous nous intéresserons aux MétaModes sécurisé et non sécurisé.

4.3 Architecture du système initial

Dans un GPS, le satellite est connecté à la base de contrôle avec laquelle il échange des données qui lui permettent de se synchroniser. Le satellite est également connecté au sous-ensemble de composants du terminal GPS qui sont chargés de la réception des signaux qu’il envoie. Le satellite et la base de contrôle sont des composants basiques, tandis que le terminal GPS est obtenu de l’assemblage de plusieurs composants.

Le système initial sur lequel nous avons choisi de travailler correspond à l'une des configurations du MétaMode non sécurisé du GPS. Il est formé par trois nœuds qui correspondent respectivement au terminal GPS, au satellite, ainsi qu'à la base de contrôle. L'architecture de ce système est détaillée dans la suite de cette section.

4.3.1 Le terminal GPS

Dans le MétaMode non sécurisé, le terminal GPS est constitué des composants suivants : **Position**, **Receiver**, **Decoder**, **TreatmentUnit** et **Encoder**. Aussi, afin d'obtenir la structure du terminal GPS de notre système initial, nous créons une instance de chacun des composants de ce MétaMode. Nous obtenons ainsi les composants **position**, **primaryReceiver**, **decoder**, **treatment** et **encoder**. L'assemblage de ces derniers nous permet d'obtenir le terminal GPS de notre système initial. La figure 4.2 propose une illustration de ce terminal GPS.

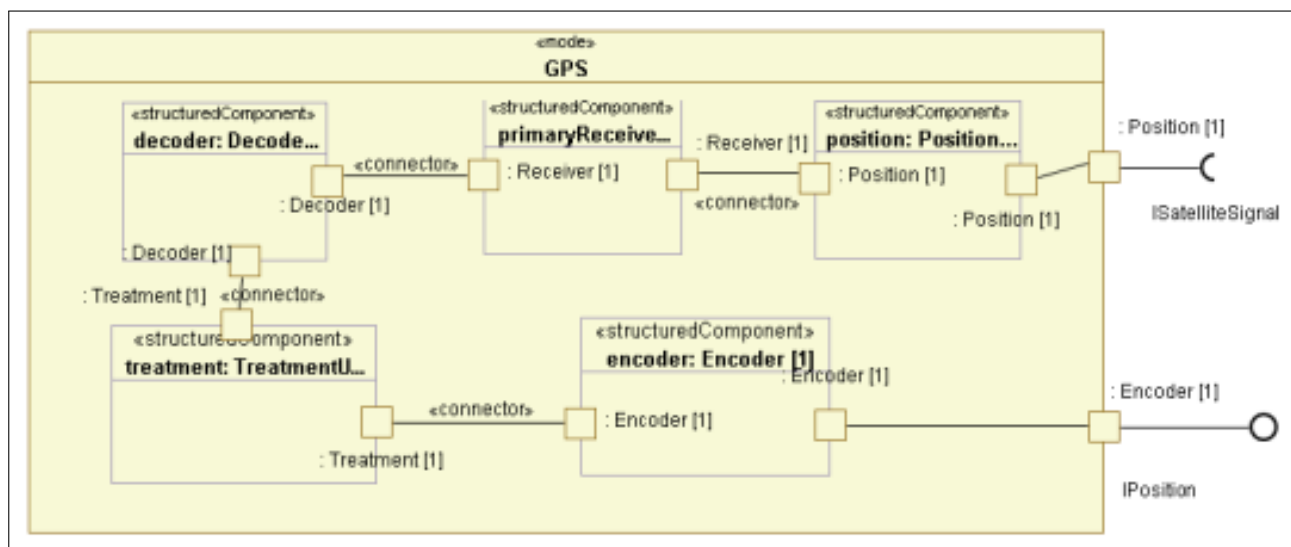


FIGURE 4.2 – Structure du terminal GPS du système initial

Le composant position

Ce composant correspond à une tâche sporadique qui possède un port d'entrée pour la réception des données, ainsi qu'un port de sortie par lequel elle envoie des

données. À chacun de ses cycles, cette tâche se bloque en attente du signal analogique qui lui est envoyé par le satellite. Lorsque ce dernier envoie un signal à la tâche, il lui fait aussi parvenir des informations liées à l'instant d'émission de ce signal. Lorsque la tâche reçoit des données issues du satellite, elle enregistre alors l'instant correspondant à la réception du signal analogique reçu. Ensuite, elle envoie au **primaryReceiver** le signal analogique, ainsi que les informations sur les instants d'envoi et de réception de ce signal. Une fois l'envoi effectué, elle se bloque en attente de la réception d'un autre signal du satellite.

Le composant **primaryReceiver**

Le **primaryReceiver** correspond également à une tâche sporadique qui possède un port d'entrée ainsi qu'un port de sortie. À chacun de ses cycles, la tâche est bloquée en attente des données issues du composant **position**. Une fois ces données reçues, la tâche effectue alors une conversion du signal analogique reçu en signal numérique. Elle transmet ensuite ce signal numérique ainsi que les informations temporelles reçues au composant **decoder**. Puis, la tâche se remet en état bloqué, et ce en attendant de recevoir de nouvelles données de la part du composant **position**.

Le composant **decoder**

Ce composant est une tâche sporadique à laquelle sont associés un port de sortie, ainsi qu'un port d'entrée. Cette tâche est débloquée par la réception des données qui lui sont envoyées par le composant **primaryReceiver**. Elle décode alors l'information contenue dans le signal reçu. Pour ce faire, elle extrait de ce signal, les données nécessaires au calcul de la distance. Elle transmet alors ces données, ainsi que les informations temporelles dont elle dispose au composant **treatment**. Ensuite, elle se bloque en attente de nouveaux événements.

Le composant **treatment**

Le composant **treatment** correspond à une tâche sporadique qui possède également un port d'entrée et un port de sortie. Cette tâche reçoit sur son port d'entrée, des informations pour calculer la distance, ainsi que des informations sur le temps d'émission du signal par le satellite, et sur l'instant de réception de ce signal. Elle effectue alors la différence entre les instants d'émission et de réception du signal.

Ensuite, elle calcule la distance qui sépare le terminal GPS du satellite. À cet effet, elle se base sur la différence de temps calculée, ainsi que sur les informations fournies pour calculer la distance. Elle transmet alors le résultat obtenu au composant **encoder**, puis repasse à l'état bloqué, le temps de recevoir de nouvelles données.

Le composant encoder

Ce composant est une tâche sporadique à laquelle est associé un port d'entrée, ainsi qu'un port de sortie. Cette tâche reçoit des données issues du composant **treatment**. Ces données correspondent à la distance qui sépare le terminal GPS du satellite. La tâche utilise alors cette distance pour déterminer la position où se trouve le terminal GPS. Elle met ensuite ce résultat à disposition de l'utilisateur, puis se remet en état bloqué, attendant de recevoir de nouvelles données de la part du composant **treatment**.

4.3.2 Le satellite

Le satellite correspond à une tâche périodique dénommée **satellite**. Celle-ci possède deux ports de sortie et un port d'entrée. À chacun de ses cycles, cette tâche transmet un message à la base de contrôle via le premier de ses ports de sortie. Ce message est une demande d'informations qui doivent permettre au satellite de se synchroniser avec l'heure terrestre. En effet, comme le satellite gravite dans l'espace, son horloge ne quantifie pas le temps de la même façon que cela est effectué sur terre. Une fois que la tâche a reçu une réponse de la base de contrôle, elle synchronise son horloge avec l'heure terrestre. Puis, via son second port de sortie, elle envoie ensuite un signal analogique au composant **position**. Ce signal est accompagné de son instant d'émission. Une fois l'envoi effectué, la tâche attend que sa prochaine période arrive. Elle reprend alors son cycle.

4.3.3 La base de contrôle

Elle est représentée par une tâche sporadique appelée **controlBase**. Un port de sortie, ainsi qu'un port d'entrée lui sont associés. Cette tâche est à chaque fois bloquée en attente d'un message émis par la tâche **satellite**. À la réception d'un message, le composant **controlBase** réunit les données nécessaires à la synchronisation du satellite et les envoie au composant **satellite**. Il se remet ensuite à l'état

bloqué, en attendant que de nouvelles données lui soient envoyées de la part de ce composant.

4.4 Architecture du système final

Le système final est obtenu en appliquant des actions de reconfiguration sur le système initial. Il correspond à l'une des configurations du MétaMode sécurisé et est formé par trois nœuds qui correspondent respectivement au terminal GPS, au satellite, ainsi qu'à la base de contrôle. L'architecture de ce système est présentée dans la suite de cette section.

4.4.1 Le terminal GPS

Dans le MétaMode sécurisé, les composants qui forment le terminal GPS sont les suivants : le **SecurePosition**, l'**AccessController**, le **Receiver**, le **Decoder**, le **Treatmentunit**, ainsi que l'**Encoder**. Aussi, pour obtenir la structure du terminal GPS de notre système final, nous créons une instance de chacun des composants de ce MétaMode. Seul le composant **Receiver** correspondra à deux instances qui vont permettre de s'assurer que chaque instance d'un même composant possède le même comportement. Nous obtenons alors respectivement les composants suivants : le **securePosition**, le **controller**, le **primaryReceiver**, le **secondaryReceiver**, le **decoder**, le **treatment**, ainsi que l'**encoder**. De l'assemblage de ces derniers, nous obtenons le terminal GPS de notre système final. La structure de ce terminal GPS est illustrée par la figure 4.3.

Le terminal GPS de notre système final est obtenu en appliquant des actions de reconfiguration sur les composants du terminal GPS du système initial. Pour ce faire, nous recourons au préalable aux mécanismes de réflexivité de notre intergiciel. En effet, nous vérifions que :

- le nombre d'instances du composant de type **SecurePosition** est nul,
- le nombre d'instances du composant de type **AccessController** est nul,
- le nombre d'instances du composant de type **Receiver** est égal à un. Cette unique instance du composant **Receiver** correspond au composant **primary-Receiver** du terminal GPS du système initial.

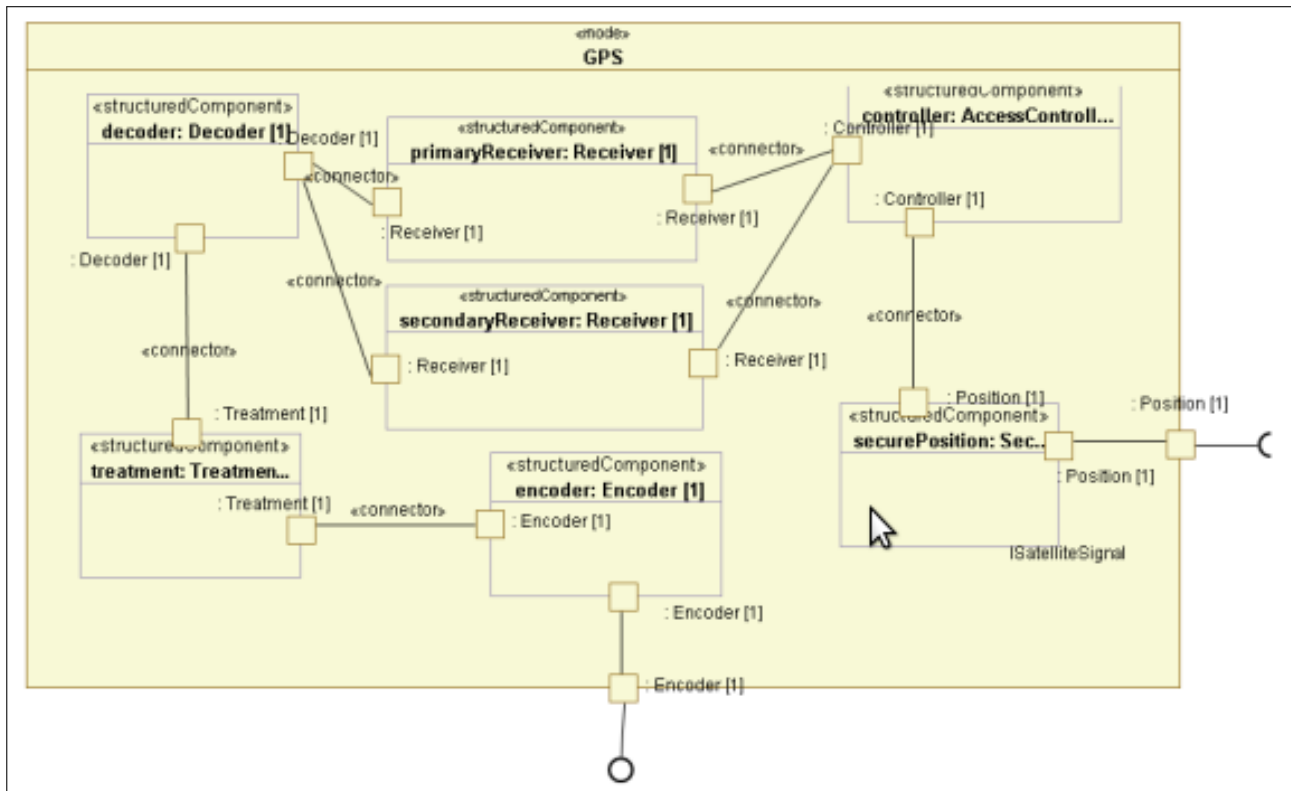


FIGURE 4.3 – Structure du terminal GPS du système final

Après avoir effectué ces vérifications sur l'état du système en cours d'exécution, nous pouvons donc procéder à la reconfiguration du système. Aussi, afin de respecter les échéances temporelles lors du passage du système initial vers le système final, nous avons instancié sur chaque nœud, une tâche sporadique démarrée à l'initialisation du système. Cette tâche est utilisée à chaque fois pour déclencher une action de reconfiguration dynamique sur un nœud donné.

Nous procédons alors à l'exécution des actions de reconfiguration proprement dites. Ainsi, les composants **securePosition**, **controller**, et **secondaryReceiver** sont créés sur la plate-forme d'exécution. La création du composant **securePosition** nécessite :

- l'ajout d'une connexion dirigée du composant **satellite** (du système initial) vers le **securePosition**,
- l'ajout d'une autre connexion issue du **securePosition** vers le **controller**.

Afin de créer le composant **controller**, il s'avère nécessaire d'ajouter des connexions qui vont lui permettre d'interagir avec d'autres composants de la plate-forme. Ainsi, seront créées :

- une connexion dirigée du **controller** vers le **primaryReceiver**,
- une connexion issue du **controller** vers le **secondaryReceiver**.

Une connexion dirigée du **secondaryReceiver** vers le **decoder** sera également ajoutée lors de la création du **secondaryReceiver**.

Le composant **position** ne fait pas partie du système final : il est donc supprimé de la plate-forme d'exécution. Ainsi, la connexion dirigée du **satellite** vers le **position**, ainsi que la connexion dirigée du **position** vers le **primaryReceiver**, sont supprimées. Les composants qui ont été ajoutés à la plate-forme d'exécution sont alors démarrés. Le terminal GPS du système final est alors obtenu.

L'obtention du terminal GPS du système final a donc nécessité la suppression du composant **position** qui existait dans le terminal GPS initial. Elle a également nécessité d'ajouter à ce dernier les composants **SecurePosition**, **secondaryReceiver** et **controller**. Dans la suite de cette section, nous décrivons les composants ajoutés au terminal GPS initial.

Le composant **securePosition**

Ce composant correspond à une tâche sporadique qui possède un port d'entrée pour la réception des données, ainsi qu'un port de sortie par lequel elle envoie des données. À chacun de ses cycles, cette tâche se bloque en attente du signal analogique qui lui est envoyé par le satellite. Pour plus de sécurité, ce signal est crypté par le satellite. Lorsque ce dernier envoie un signal à la tâche, il lui fait aussi parvenir des informations liées à l'instant d'émission du message. Lorsque la tâche reçoit ce signal, elle enregistre alors l'instant correspondant à la réception du signal analogique. Ensuite, elle envoie au **controller** le signal reçu, ainsi que les informations sur les instants d'envoi et de réception de ce signal. Une fois l'envoi effectué, elle se bloque en attente de la réception d'un autre signal du satellite.

Le composant **controller**

Ce composant est représenté par une tâche sporadique à laquelle est associée deux ports : un port d'entrée, ainsi qu'un port de sortie. Cette tâche est à chaque

fois bloquée en attente des données issues du **securePosition**. Lorsque ces données sont reçues, la tâche va effectuer le décryptage du signal analogique reçu. Elle envoie ensuite le signal décrypté ainsi que les informations sur les instants d'envoi et de réception de ce signal à ses destinataires. Ceux-ci sont les composants **primaryReceiver** et **secondaryReceiver**. La tâche repasse ensuite à l'état bloqué et attend de recevoir des données supplémentaires.

Le composant **secondaryReceiver**

Ce composant joue le même rôle et a la même structure ainsi que la même nature que le composant **primaryReceiver**. Il possède ainsi un port d'entrée et un port de sortie. Ce dernier lui permet d'envoyer des données au composant **decoder**.

4.4.2 Le satellite

Dans le MétaMode sécurisé, le satellite correspond à une tâche périodique dénommée **satellite**. Celle-ci possède deux ports de sortie et un port d'entrée. À chacun de ses cycles, cette tâche transmet également à la base de contrôle, un message en vue de sa synchronisation. Ce message est transmis via son premier port de sortie. Une fois que la tâche a reçu une réponse de la base de contrôle, elle synchronise son horloge avec l'heure terrestre. Ensuite, via son second port de sortie, elle envoie cette fois-ci un signal analogique crypté au composant **securePosition**. Ce signal est accompagné de son instant d'émission. Une fois l'envoi effectué, la tâche attend qu'arrive sa prochaine période afin de reprendre son cycle d'exécution.

Pour passer du composant **satellite** du système initial, vers celui du système final, il a fallu appliquer deux actions de reconfiguration sur le composant **satellite** initial. En effet, dans le système initial, ce composant est connecté aux composants **position** et **controlBase**. Par contre, dans le système final, le **satellite** est plutôt connecté aux composants **securePosition** et **controlBase**. Aussi, pour obtenir le composant **satellite** correspondant au système final, il a donc été nécessaire de :

- supprimer la connexion dirigée du composant **satellite** vers le composant **position**,
- ajouter une connexion entre le composant **satellite** et le composant **securePosition**.

4.4.3 La base de contrôle

Dans le MétaMode sécurisé, la base de contrôle est toujours représentée par une tâche sporadique. Cette dernière est appelée **controlBase** et elle est associée à un port de sortie, ainsi qu'à un port d'entrée. Cette tâche conserve le même comportement que celui de la tâche sporadique **controlBase** décrite à la section 4.3.3. En effet, le passage du MétaMode non sécurisé vers le MétaMode sécurisé affecte uniquement le satellite et le terminal GPS. Par conséquent, le composant **controlBase** du système initial a été conservé tel quel dans le système final.

4.5 Évaluation pratique

Afin de pouvoir tester notre étude de cas, nous avons travaillé sur un ordinateur fonctionnant sur une fréquence de 1,8 GHz. Ce dernier utilise un système d'exploitation *Ubuntu 9.10 (Karmic)*. Il dispose d'une version du RTSJ correspondant au JDK 1.5, ainsi que d'un compilateur Ocarina 2.

Pour lancer les différents nœuds de notre système initial, nous avons effectué des simulations sur trois terminaux de cet ordinateur. Ainsi, le premier terminal correspond au terminal GPS (nœud 0) et le deuxième terminal au satellite (nœud 1). Le troisième terminal correspond quant à lui à la base de contrôle (nœud 2). Nous avons donc démarré le système initial à partir de ces trois terminaux. Les différentes valeurs associées aux propriétés des composants du système initial sont répertoriées dans le tableau 4.1.

La figure 4.4 présente ainsi une trace de l'exécution du terminal GPS qui correspond au nœud 0 du système initial. Les composants **position**, **primaryReceiver**, **decoder**, **treatment** et **encoder** y sont respectivement associés aux identifiants allant de 0 à 4.

Nous passons ensuite au système final en initiant les actions de reconfiguration dynamiques spécifiées dans la section 4.4. Le tableau 4.2 liste les différentes valeurs que nous avons associées aux propriétés des composants du système final.

Dans le système final, les composants **securePosition**, **controller**, et **secondaryReceiver** sont respectivement associés aux identifiants 5, 6 et 7. La figure 4.5 montre une trace de l'exécution du système final.

Composant	Propriétés							
	nature	période	échéance	ins. déb.	ins. fin	WCET 1	WCET 2	tail. mém.
satellite	périodique	400 ms	400 ms	0	0	720 ms	0,1 ms	100 Ko
position	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
primaryReceiver	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
decoder	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
treatment	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
encoder	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
controlBase	sporadique	200 ms	200 ms	0	0	720 ms	0,1 ms	100 Ko

TABLE 4.1 – Valeurs des propriétés des composants du système initial

Signalons qu’un système fonctionnant avec une fréquence de 1 MhZ a un **WCET 1** égal à 0.4 milliseconde. Ainsi, nous avons donc déduit les différentes valeurs des **WCET 1** des composants, en effectuant une règle de trois. Par ailleurs, nous avons fixé arbitrairement à 0,1 milliseconde la valeur du **WCET 2** des différentes tâches qui constituent notre étude de cas.

Réflexivité

Avant de lancer la reconfiguration du système, nous vérifions notamment que les composants que nous devons ajouter n’existent pas encore sur la plate-forme d’exécution. Pour ce faire, nous calculons le nombre d’instances leur correspondant. Cette observation est illustrée par la figure 4.6. Après la reconfiguration, nous vérifions que les composants ajoutés existent bien sur la plateforme d’exécution. Nous nous assurons également que le composant supprimé (**position**) n’existe plus sur la plateforme d’exécution. Pour ce faire, nous déterminons également le nombre d’instances leur correspondant. Cette observation est illustrée par la figure 4.7.

```

Creation of the position task on node 0
Sporadic task 0 : wait initialization
Sporadic task 0 : new Dispatch
Entity 0 is waiting for incoming events
Creation of the primary receiver task on node 0
Sporadic task 1 : wait initialization
Sporadic task 1 : new Dispatch
Entity 1 is waiting for incoming events
Creation of the decoder on node 0
Sporadic task 2 : wait initialization
Sporadic task 2 : new Dispatch
Entity 2 is waiting for incoming events
Creation of the treatment task on node 0
Sporadic task 3 : wait initialization
Creation of the encoder task on node 0
Sporadic task 4 : wait initialization
Sporadic task 4 : new Dispatch
Sporadic task 3 : new Dispatch
Entity 4 is waiting for incoming events
Entity 3 is waiting for incoming events
Node 0 : receive 18 bytes from node 1
Entity 0 : store received message
Entity 0, storeIn : global storage position = 0
Entity 0, historyIncrementLast : globalHistoryLast = 0
Entity 0, storeIn : enqueued event [data] message for input port 1
Entity 0, waitEvent : oldest unread event [data] input port 1
Entity 0 receives of event [data] message on event [data] input port 1
Sporadic task 0 : received an event
Entity 0, count : FIFO exists for input port 1
Entity 0, getCount of input port 1 is 1
Entity 0, getValue of input port 1
Entity 0, readIn : reading the oldest element in the queue of event [data] input port 1

```

FIGURE 4.4 – Trace de l'exécution du terminal GPS du système initial

La cohérence du système reconfiguré

L'application d'actions de reconfiguration sur le système initial nous a permis d'obtenir un système final qui fonctionne correctement. Un exemple de la mise en

Composant	Propriétés							
	nature	période	échéance	ins. déb.	ins. fin	WCET 1	WCET 2	tail. mém.
satellite	périodique	400 ms	400 ms	0	0	720 ms	0,1 ms	100 Ko
securePosition	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
controller	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
primaryReceiver	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
secondaryReceiver	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
decoder	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
treatment	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
encoder	sporadique	100 ms	100 ms	0	0	720 ms	0,1 ms	100 Ko
controlBase	sporadique	200 ms	200 ms	0	0	720 ms	0,1 ms	100 Ko

TABLE 4.2 – Valeurs des propriétés des composants du système final

œuvre de la cohérence est ainsi illustré par la figure 4.8. Cette dernière montre que le composant **position**, associé à l'identifiant 0, achève d'abord de traiter ses requêtes, avant d'être retiré de la plateforme d'exécution. Pendant ce temps, les composants nouvellement ajoutés ont commencé leur exécution.

Le respect des échéances temps réel

Afin de contribuer à l'exécution d'actions de reconfiguration dynamique dans un délai de temps déterministe, nous avons instancié sur chaque nœud, une tâche sporadique supplémentaire. Celle-ci se charge d'effectuer l'action de reconfiguration dynamique demandée, et ce en respectant son échéance. Cela nous a aidé à faire en sorte que l'exécution du système demeure déterministe, malgré l'introduction de modifications liées à la reconfiguration dynamique.

```

Entity 5, sendOutput done.
Entity 6, waitEvent : oldest unread event [data] input port 13
Entity 6 receives of event [data] message on event [data] input port 13
Sporadic task 6 : received an event
Sporadic task 5 : new Dispatch
Entity 6, count : FIFO exists for input port 13
Entity 6, getCount of input port 13 is 1
Entity 5 is waiting for incoming events
Entity 6, getValue of input port 13
Entity 6, readIn : reading the oldest element in the queue of event [data] input port 13
Entity 6, readIn : value read from input port 13
Entity 6, readIn : reading the oldest element in the queue of event [data] input port 13
Entity 6, readIn : value read from input port 13
Entity 6, getValue done
Entity 6, nextValue for event [data] input port 13
Entity 6, dequeue : dequeuing event [data] input port 13
Entity 6, historyIncrementFirst : globalHistoryFirst = 1
Controller calls another sub program :
  at t=12h46m24s657ms ***** the node 0 controlled the signal to get the value :4
Entity 6, storeOut : storing value for output port 12
Entity 6, storeOut : value stored for output port 12
Entity 6, sendOutput for output port 12
Entity 6, setInvalid : setting invalid for output port 12
Entity 6, sendOutput output port 12
Entity 6, send output to input port 3 of entity 1
Entity 1 : store received message
Entity 1, storeIn : global storage position = 0
Entity 1, historyIncrementLast : globalHistoryLast = 0
Entity 1, storeIn : enqueued event [data] message for input port 3
Entity 1, waitEvent : oldest unread event [data] input port 3
Entity 6, send output to input port 15 of entity 7
Entity 1 receives of event [data] message on event [data] input port 3
Entity 7 : store received message
Entity 7, storeIn : global storage position = 0

```

FIGURE 4.5 – Trace de l'exécution du terminal GPS du système final

La faible empreinte mémoire

RCES4RTES est un intergiciel qui possède une empreinte mémoire qui est de loin inférieure à celle des intergiciels usuels pour les systèmes TR²E. En effet, l'empreinte mémoire de cet intergiciel a pu être mesurée lors du test mené sur l'étude de cas.

```
***** the reconfiguration of the initial system begins now
Reflection: there is 1 instances of the component type Receiver
Reflection: there is 0 instances of the component type SecurePosition
Reflection: there is 0 instances of the component type AccessController
Reflection: there is 1 instances of the component type Position
```

FIGURE 4.6 – Observation du système avant la reconfiguration

```
Reflection: there is 2 instances of the component type Receiver
Reflection: there is 1 instances of the component type SecurePosition
Reflection: there is 1 instances of the component type AccessController
Reflection: there is 0 instances of the component type Position
```

FIGURE 4.7 – Observation du système après la reconfiguration

```
Entity 0, sendOutput for output port 0
Entity 0, setInvalid : setting invalid for output port 0
Entity 0, sendOutput output port 0
Entity 0, sendOutput done.
Entity 5, setInvalid : setting invalid for output port 10
Entity 5, sendOutput output port 10
Dynamic reconfiguration: The task 0 just ended its cycle and is now removed
Entity 5, send output to input port 13 of entity 6
```

FIGURE 4.8 – Un exemple de la cohérence du système final

Cette mesure a révélé que RCES4RTES occupe un espace mémoire de 122 Ko sur l'exécutable. Aussi, les ressources des systèmes initial et final qui y ont recours ne sont que légèrement entamées par la présence de cet intergiciel.

4.6 Conclusion

L'étude de cas portant sur le GPS nous a permis d'évaluer notre intergiciel. Pour ce faire, nous avons travaillé sur un système initial correspondant à une configuration du MétaMode non sécurisé. Nous avons ensuite introduit des mécanismes de reconfiguration sur ce système, après s'être au préalable renseignés sur l'état du sys-

4.6 Conclusion

tème en cours d'exécution. Nous avons alors obtenu un système final qui correspond à une configuration du MétaMode sécurisé et dont les échéances temporelles sont respectées. En outre, ce système reconfiguré est portable, et son fonctionnement est correct. Dans la volée, nous avons également pu vérifier que la taille occupée par l'intergiciel dans le système à reconfigurer est faible.

Conclusion générale et perspectives

Un intergiciel est une couche intermédiaire utilisée afin de garantir l'interopérabilité et la portabilité des systèmes distribués. Cela étant, la plupart des supports d'exécution pour systèmes TR²E souffrent de plusieurs lacunes qui empêchent de garantir à ces systèmes de s'adapter ou même d'évoluer, tout en restant disponibles. En effet, ces lacunes peuvent être situées aussi bien au niveau de la difficulté à gérer les systèmes TR²E, du manque de fonctionnalités intergicelles, ou encore de la dualité entre l'évolutivité et la disponibilité. En outre, l'incapacité à préserver les contraintes temps réel, la difficulté à préserver le caractère embarqué, ainsi que le risque d'incohérence du système reconfiguré sont également des lacunes qui viennent compléter ce tableau.

Afin d'adresser toutes ces limites, nous avons alors proposé une approche permettant de concevoir et de développer un support d'exécution pour les systèmes TR²E dynamiquement reconfigurables. Ce support d'exécution supporte les mécanismes de reconfiguration en faisant intervenir les notions de réflexivité et de cohérence. En effet, la réflexivité permet d'observer l'exécution d'un système afin de mieux cibler l'action de reconfiguration à effectuer. La cohérence consiste quant à elle maintenir le système dans un état correct, une fois qu'il a été reconfiguré.

Notre approche a ainsi été validée par le biais d'une étude de cas portant sur un GPS. Ceci nous a alors permis d'effectuer une évaluation pratique de notre support d'exécution. Il en ressort ainsi que le recours à notre intergiciel présente plusieurs avantages notamment en termes de respect des échéances temporelles ou d'espace mémoire occupé sur l'exécutable.

Cela étant, notre intergiciel supporte des mécanismes permettant l'exécution d'une seule action de reconfiguration dynamique à la fois. Dans de futurs travaux, il

serait donc intéressant d'y introduire des modifications afin qu'il assure l'exécution simultanée de plusieurs actions de reconfiguration dynamique. À ce titre, il devra donc être capable de gérer la concurrence entre ces reconfigurations parallèles.

Enfin, notre intergiciel pourrait être étendu afin de supporter les mécanismes de tolérance aux fautes par réplication. En effet, notre support d'exécution prend déjà en charge les mécanismes permettant de migrer un processus léger d'un nœud à un autre du système réparti. Cet aspect peut être exploité en choisissant une stratégie de réplication (passive ou active) qui va régir la migration des processus dans le système distribué, et ce en cas de surcharge de processeurs.

Bibliographie

- [Bal07] Jaiganesh Balasubramanian. Flare : a fault-tolerant lightweight adaptive real-time middleware for distributed real-time and embedded systems. *Middleware (Doctoral Symposium)*, 2007.
- [Bal09] Jaiganesh Balasubramanian. *Resource-aware deployment, configuration, and adaptation for fault-tolerant distributed real-time embedded systems*. PhD thesis, 2009.
- [BB09] Eric J. Bruno and Greg Bollella. chapter The Real-Time Specification for Java. Prentice Hall, 2009.
- [BEJV93] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 1993.
- [Bes10] Xavier Besseron. *Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle*. PhD thesis, 2010.
- [BSTG⁺09] Jaiganesh Balasubramanian, Chenyang Lu Sumant Tambe, Anirudha S. Gokhale, Christopher D. Gill, and Douglas C. Schmidt. Adaptive failover for real-time middleware with passive replication. *IEEE Real-Time and Embedded Technology and Applications Symposium*), 2009.
- [ECS03] ECSS. *Space engineering. SpaceWire - Links, nodes, routers and networks*, january 2003.
- [FP09] Albert Ferrer Florit and Steve Parkes. Unified communication infrastructure for small satellites. <http://www.iafastro.net/iac/archive/browse/IAC-09/B4./6A./5351/>, 2009.

- [Gas11] Amal Gassara. Vérification formelle des propriétés non fonctionnelles des systèmes embarqués temps réel distribués dynamiquement reconfigurables. Master's thesis, National Engineering School of Sfax, 2011.
- [Hil11] James H. Hill. Modeling interface definition language extensions (idl3+) using domain-specific modeling languages. *ISORC*, 2011.
- [Hug05] Jérôme Hugues. *Architecture et Services des Intergiciels Temps Réel*. PhD thesis, 2005.
- [KCBC02] Fabio Kon, Fabio M. Costa, Gordon S. Blair, and Roy H. Campbell. The case for reflective middleware. In *Communications of the ACM - Adaptive middleware*, volume 45 Issue 6, pages 33–38. ACM, june 2002.
- [KHZC10] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Bernard Coulette. Designing dynamic reconfiguration for distributed real time embedded systems. *NOTERE*, pages 249–254, 2010.
- [KHZJ11] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. Towards a Model-Based Approach for Reconfigurable Distributed Real Time Embedded Systems (regular paper). In *European Conference on Software Architecture (ECSA), Essen, Germany, 13/09/2011-16/09/2011*, <http://www.springerlink.com>, septembre 2011. Springer.
- [Kon02] Fabio Kon. <http://srg.cs.uiuc.edu/2k/dynamicTA0/>, july 2002.
- [Kri10] Fatma Krichen. Position paper : Advances in reconfigurable distributed real time embedded systems. *NOTERE*, pages 273 – 278, 2010.
- [KRL⁺00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2000.
- [LAS08] Gilles LASNIER. Étude et support du standard aadlv2 dans ocarina. Master's thesis, Télécom PARISTech, 2008.
- [Leg09] Marc Leger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants*. PhD thesis, 2009.

- [Loi08] Frédéric Loiret. *Tinap : Modèle et infrastructure d'exécution orienté composant pour applications multi-tâches à contraintes temps réel souples et embarquées*. PhD thesis, 2008.
- [Lou10] Sihem Loukil. Extension d'un langage de description d'architecture pour la programmation orientée aspect. Master's thesis, National Engineering School of Sfax, 2010.
- [LT09] Kung-Kiu Lau and Faris M. Taweel. Domain-specific software component models. *CBSE*, 2009.
- [LW07] Kung-Kiu Lau and Zheng Wang. A survey of software component models. *IEEE Transactions on Software Engineering*, 33, October 2007.
- [OMG07] OMG. Uml profile for marte, beta 1, ptc/07-08-04. <http://www.omg.org/cgi-bin/doc?ptc/2007-08-04>, january 2007.
- [RP05] Andreas Rasche and Andreas Polze. Dynamic reconfiguration of component-based real-time software. In *WORDS*, pages 347 – 354, 2005.
- [SAE04] SAE. Architecture analysis & design language (as5506). <http://www.sae.org>, september 2004.
- [SAE09] SAE. Architecture analysis & design language (as5506). <http://www.sae.org>, january 2009.
- [SAE11] SAE. Architecture analysis and design language (aadl) annex volume 2. <http://www.sae.org>, january 2011.
- [Sch04] Etienne Schneider. *A Middleware Approach for Dynamic Real-Time Software Reconfiguration on Distributed Embedded Systems*. PhD thesis, 2004.
- [SDG⁺07] Venkita Subramonian, Gan Deng, Christopher D. Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha S. Gokhale, and Nanbor Wang. The design and performance of component middleware for qos-enabled deployment and configuration of dre systems. *Journal of Systems and Software* 80(5), pages 668–677, 2007.
- [SGM02] C. Szyperski, FD. Gruntz, and S. Murer. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

- [SSGW04] Venkita Subramonian, Liang-Jui Shen, Christopher Gill, and Nanbor Wang. The design and performance of configurable component middleware for distributed real-time and embedded systems. *Real-Time Systems Symposium*, pages 252 – 261, 2004.
- [TdOM05] Cssia Yuri Tatibana, Romulo Silva de Oliveira, and Carlos Montez. Dynamic guarantee in component-based distributed real-time systems. In *Emerging Technologies and Factory Automation (ETFA)*, pages 8–14, september 2005.
- [TMdO07] Cssia Yuri Tatibana, Carlos Montez, and Rômulo Silva de Oliveira. Real-time dynamic guarantee in component-based middleware. In *ISORC'07*, pages 214–221, 2007.
- [VHPK04] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Polyorb : a schizophrenic middleware to build versatile reliable distributed applications. *Ada-Europe*, 3063, 2004.
- [VZH11] Thomas Vergnaud, Bechir Zalila, and Jérôme Hugues. Ocarina : a compiler for the aadl. Technical report, june 2011.
- [Zal08] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, 2008.
- [ZGL10] Juanfang Zhang, Christopher D. Gill, and Chenyang Lu. Configurable middleware for distributed real-time systems with aperiodic and periodic tasks. *IEEE Real-Time and Embedded Technology and Applications Symposium*, 21 no. 3 :393–404, march 2010.
- [ZH10] Adel Ziani and Brahim Hamid. Clock Synchronization Modeling in DRTES (regular paper). In *Hands-on Platforms and tools for model-based engineering of Embedded Systems (workshop at ECMFA 2010) (HoPES), Paris, 15/06/2010-16/06/2010*, pages 51–56, <http://www-list.cea.fr>, 2010. CEA LIST.

Abréviations

ACE	Adaptive Communication Environment
ADL	Architecture Description Language
AADL	Architecture Analysis and Design Language
CCM	Corba Component Model
CIAO	Component Integrated ACE ORB
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
DOC	Distributed Object Computing
DSA	Distributed System Annex
DSL	Domain Specific Language
ECSS	European Cooperation for Space Standardization
EJB	Enterprise JavaBeans
ESA	European Spatial Agency
FLARe	Fault-tolerant Load-aware and Adaptive middlewaRe
GIOP	General Inter-ORB Protocol
GPS	Global Positioning System
IDL	Interface Definition Language
LwCCM	LightWeight CCM
MARTE	Modeling and Analysis of Real-Time and Embedded systems
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
OSGI	Open Services Gateway Initiative
PolyORB_HI	PolyORB High Integrity
QoS	Quality of Service

-
- RCA4RTES** Reconfigurable Computing Architecture for Real Time Embedded Systems
- RCES4RTES** Reconfigurable Computing Execution Support for Real Time Embedded Systems
- RTSJ** Real Time Specification for Java
- SAE** Society of Automotive Engineers
- SCA** Service Component Architecture
- SOA** Service Oriented Architectures
- SOAP** Simple Object Access Protocol
- TAO** The ACE ORB
- TR²E** Temps Réel Réparti Embarqué
- UML 2.0** Unified Modeling Language 2.0
- WCET** Worst Case Execution Time
- XML** Extensible Markup Language

Conception et développement d'un support d'exécution pour systèmes TR²E dynamiquement reconfigurables

Alvine BOAYE BELLE

الخلاصة: معظم دواعم التنفيذ للأنظمة الآتية الموزعة والمضمنة تعاني من مشاكل عديدة تمنعها من ضمان القدرة على التكيف، أو حتى التطور في حين تبقى متوفرة. لتصحيح هذا، نقترح نهجا لتصميم وتطوير نظام لدعم التنفيذ للأنظمة الآتية الموزعة والمضمنة المتغيرة وهذا من خلال استعمال الفعل المنعكس، والاتساق.

المفاتيح: TR²E، دعم التنفيذ، إعادة تشكيل ديناميكي، الانعكاسية، الاتساق.

Résumé : la plupart des supports d'exécution pour systèmes temps réel répartis embarqués (TR²E) souffrent de plusieurs lacunes qui les empêchent de garantir à ces systèmes de s'adapter ou même d'évoluer, tout en restant disponibles. Afin d'y remédier, nous proposons une approche permettant de concevoir et de développer un support d'exécution qui supporte la reconfiguration dynamique dans les systèmes TR²E, et ce en recourant à la réflexivité, ainsi qu'à la cohérence.

Mots clés: TR²E, support d'exécution, reconfiguration dynamique, réflexivité, cohérence.

Abstract: Many middlewares for distributed real time embedded systems (DRE) suffer from several drawbacks that prevent them to ensure these systems to adapt or even to evolve, while remaining available. To address these shortcomings, we propose an approach to design and develop a middleware which enables dynamic reconfiguration in DRE systems, and this through the use of reflection and consistency.

Key-words: DRE, middleware, dynamic reconfiguration, reflection, coherence.