Chapitre 8

Étude des principes de conception d'un modèle de composition spatio-temporel

Sommaire

En plus des modèles de composants logiciels, nous avons présenté dans le chapitre 3 une deuxième famille de modèles de programmation par composition : les modèles de flux de travail. Rappelons que contrairement aux modèles de composants, où la composition d'une application se base sur un raisonnement spatial, les modèles de flux de travail s'appuient sur un raisonnement temporel. Nous avons ainsi discuté les avantages et limitations respectives de ces deux familles de modèles et nous en avons déduit une complémentarité. Les modèles de composants sont mieux adaptés pour réaliser des applications de couplage de code où

le couplage est dit « fort ». Par ailleurs, l'expressivité de l'évolution temporelle de l'exécution d'une application, offerte par les langages de flux de travail, joue un rôle important sur la possibilité d'utiliser au mieux des ressources partagées. Nous avons ainsi considéré dans le chapitre 4 l'importance d'introduire la dimension temporelle dans les modèles de composants logiciels, cela en plus de préserver leurs avantages. L'objectif de cette partie du manuscrit est de présenter nos travaux contribuant à l'achèvement d'un tel objectif.

Ce chapitre est consacré à l'analyse de la possibilité de prendre en compte conjointement les concepts sur lesquels sont fondés les modèles de composants et les modèles de flux de travail, afin d'en construire un modèle dit spatio-temporel. Notre proposition de modèle, nommé STCM, fait l'objet du chapitre suivant.

Dans un premier temps, nous présentons et discutons les tentatives de combinaison des dimensions spatiale et temporelle trouvées dans la littérature. Ensuite, la section 8.2 détermine les propriétés que nous attendons d'un modèle spatio-temporel. Sans perdre de vue ces objectifs, la section 8.3 étudie les concepts fondamentaux sur lesquels peut s'appuyer une composition spatio-temporelle. Enfin, différentes alternatives pour définir une telle composition sont étudiées et discutées dans la section 8.4.

8.1 Analyse de solutions existantes

La question de considérer une composition spatiale et une composition temporelle dans un même modèle de programmation a déjà été abordée dans la littérature. Toutefois, dans les travaux que nous avons pu étudier, la construction d'une application est en soi basée sur une seule dimension. Ainsi, des modèles de composants proposent de prendre en compte le comportement temporel de l'exécution d'une application à travers une composition spatiale. Symétriquement, des langages de flux de travail considèrent la dimension spatiale à travers une composition temporelle. Cette section a pour objectif de présenter et discuter les principales solutions proposées.

8.1.1 Approches basées sur une composition spatiale

Prendre en compte la logique temporelle de l'exécution d'une application composée suivant un raisonnement spatial, consiste généralement en la spécification de méta-informations. ICENI [67] (*the Imperial College e-Science Networked Infrastructure*) est un exemple de modèle de composants adoptant une telle approche. Son objectif est d'être capable d'ordonnancer efficacement les composants d'une application de manière à optimiser leur placement sur des ressources d'exécution. Pour cela, un développeur fournit pour une implémentation d'un composant une description de son comportement interne, ceci sous forme d'un flux de travail. Au déploiement, un comportement global d'une application est construit en reliant les descriptions de ses composants. Le résultat est considéré pour le calcul d'un placement favorisant la rapidité de calcul et/ou un partage efficace de ressources.

Même si la spécification de méta-informations peut aider à optimiser le partage de ressources, les avantages ne sont que partiels. D'une part, la logique algorithmique d'une application composée dans l'espace reste enfouie dans le code. Nous avons vu dans le chapitre 3 que cela va à l'encontre des objectifs de la programmation par composition où un changement de cette logique implique une modification du code. D'une autre part, la spécification de méta-informations demande une connaissance approfondie de toutes les implémenta-

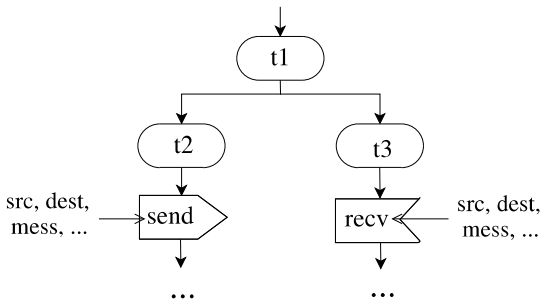


FIG. 8.1 – Exemple de composantes d'un flux de travail dédiées aux communications.

tions des composants. ICENI suppose qu'il revient à la charge du développeur d'une implémentation d'un composant de décrire son comportement. Lorsqu'il s'agit alors de reprendre des codes patrimoniaux, un utilisateur doit fournir un effort pour les étudier avant d'être capable de décrire leur comportement. De plus, reproduire le comportement d'un composant à travers des méta-informations ne garantit pas la conformité avec le comportement réel de son implémentation. Il est toutefois à noter que tous les composants d'une application sont déployés avant son exécution. Ainsi, une erreur dans la description des méta-informations ne devrait affecter que les performances attendues.

8.1.2 Approches basées sur une composition temporelle

Généralement, il est possible de considérer une composition spatiale au sein d'une application construite sous forme d'un flux de travail. Pour cela, deux approches sont principalement utilisées. La première considère une composition spatiale dans l'implémentation d'une tâche. Quant à la deuxième approche, elle consiste à définir des tâches dédiées aux communications entre codes. Ces deux approches sont brièvement analysées dans ce qui suit.

Premièrement, une tâche d'un flux de travail peut encapsuler une composition spatiale. Une telle tâche peut apparaître sous la forme d'un exécutable gérant cette composition. Cette approche est supportée dans des technologies comme TRIANA [106], KEPLER [30] ou ASKALON [59]. Son inconvénient est qu'elle réduit la hiérarchie d'un assemblage à deux niveaux, puisqu'elle s'arrête au niveau d'une tâche contenant une composition spatiale. Il en résulte une limitation de la réutilisation et une complexité de programmation.

Deuxièmement, il est possible de retrouver dans des langages de flux de travail comme GRICOL [52] ou le langage proposé dans [92], des composantes prédéfinies pour la réalisation de communications entre codes. Ces composantes apparaissent sous forme de tâches ou points d'un flux de travail dédiés à l'envoi et la réception de messages (principe illustré à la figure 8.1). En paramétrant ce type de tâches pour déterminer l'émetteur ou le récepteur d'un message, une composition spatiale est implicitement construite. Cependant, adopter une telle solution pour réaliser une composition spatio-temporelle demande le fractionnement de codes métiers afin d'en extraire les communications. Comparée à une approche de programmation à base de composants, n'ayant pas cette contrainte, cette solution semble être complexe à utiliser, en particulier pour programmer des applications de couplage de codes.

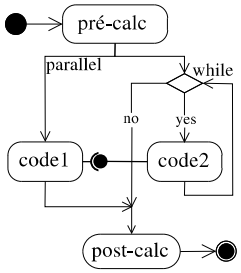


FIG. 8.2 – Exemple d'une composition spatio-temporelle.

8.1.3 Conclusion

Nous venons de présenter les principales approches rencontrées dans la littérature qui tentent de combiner les compositions spatiales et temporelles au sein d'un même modèle de programmation. Même si la composition dans les approches proposées s'appuie en soi sur un seul type, elles ont montré la possibilité de considérer séparément la dimension orthogonale à un autre niveau de conception : à travers des méta-informations ou dans l'implémentation des composantes d'une application. Cependant, cette séparation non seulement limite la réutilisation et complexifie un modèle de programmation, mais ne permet pas de bénéficier pleinement des apports couplés des modèles de composants et des modèles de flux de travail. En d'autres termes, ce qu'un modèle de programmation par composition devrait offrir. Notre objectif dans ce chapitre est de contribuer à l'étude d'un modèle de composition où il serait possible de construire la structure d'une application selon une combinaison de compositions spatiales et temporelles, qui répondrait à la problématique des solutions existantes.

8.2 Objectifs

L'importance d'abstraire le comportement d'une application de l'implémentation de ces composantes a été introduite dans le chapitre 4. Plus précisément, dans la section 4.3.4, nous avons motivé l'intérêt d'abstraire l'évolution dans le temps de l'exécution d'une application. À ce sujet, nous proposons d'offrir à un concepteur la possibilité d'utiliser concurremment une composition spatiale et temporelle à tout niveau de la structure d'une application, en particulier lors de son assemblage. Pour cela, les objectifs suivants sont visés :

Modèle d'assemblage spatio-temporel expressif : il est important que la structure d'une application soit capable d'exprimer un large ensemble de paradigmes de composition tel que des structures de flot contrôle (séquence, branchement conditionnel, boucle, parallélisme, etc.), des constructions de flot de données, le passage de messages, l'appel de procédures/méthodes à distance, le partage de données ou les communications événementielles. D'un point de vue utilisateur, la possibilité d'exprimer de tels paradigmes par assemblage offrirait une façon simple de programmer plusieurs types d'applications. D'une autre part, expliciter au mieux une logique algorithmique à travers une composition permettrait l'automatisation de la gestion d'une composition qui doit aussi être capable d'utiliser efficacement des ressources d'exécution. La structure représentée dans la figure 8.2 est un exemple de

type d'assemblage que nous souhaitons permettre de construire. Cette structure est capable d'exprimer un séquencement de calculs sur un couplage de codes (**`code1-code2`**) rendu explicite dans la composition de la logique algorithmique d'une application par assemblage. Notre objectif est d'être capable de construire simplement un tel type de composition.

Capacité à déduire le cycle de vie des composantes à l'exécution : exprimer simultanément plusieurs paradigmes de composition en combinant dimensions spatiale et temporelle, peut rendre complexe la détermination du cycle de vie des composantes d'un assemblage à l'exécution. La capacité de capturer le comportement global exprimé par une composition permettrait de déduire simplement la suite des actions de création/destruction et de connexion/déconnexion, à effectuer sur ces composantes. Cette suite d'actions détermine l'évolution d'un état rattaché à une composante faisant partie de son cycle de vie. L'objectif est d'avoir un modèle qui spécifie les différents états possibles et leur évolution.

Modèle hiérarchique : cette propriété apparaît actuellement comme une évidence pour maîtriser simplement les structures d'applications complexes et promouvoir la réutilisation.

Adaptation de modèles existants : il est possible de définir un modèle spatio-temporel à partir de rien, ce qui permettrait de se concentrer sur les concepts souhaités. Cependant, il existe déjà des modèles de composition spatiale et de composition temporelle. Un effort important a permis leur réalisation. Au lieu d'avoir à redéfinir complètement les concepts fondamentaux et les principes de composition, il serait plus intéressant de pouvoir réutiliser et fusionner l'existant. Ainsi, nous avons décidé de définir un modèle en extrayant les concepts qui nous intéressent et en étendant des travaux existants.

En s'appuyant sur ces différents objectifs, nous définissons les concepts fondamentaux et nous étudions une approche pour la conception d'un modèle de composition spatio-temporel.

8.3 Concepts fondamentaux

Avant d'analyser différentes façons de concevoir un modèle de composition spatio-temporel il est primordial d'identifier les concepts sur lesquels un tel modèle est fondé. Nous avons identifié trois concepts : la notion de *composant-tâche*, les *ports temporels* et le cycle de vie d'une instance de composant-tâche. Dans cette section, nous présentons ces trois concepts.

8.3.1 Notion de composant-tâche

Le premier concept à définir est l'unité de composition de base. Nous avons rencontré, en particulier dans le chapitre 3, deux types de concepts : les composants et les tâches. Afin de définir une unique unité de base qui pourrait être composé dans le temps, l'espace, ou dans le temps et l'espace, nous proposons de fusionner ces deux concepts au sein d'un seul. Pour cela, il semble être nécessaire d'analyser tout d'abord leurs similitudes et leurs différences.

D'un point de vue architectural, un composant et une tâche sont assez proches : chacun encapsule un code sous forme de boîte noire définissant des ports. Cependant, ils diffèrent principalement sur leur cycle de vie à l'exécution. La durée de vie d'une tâche se résume au

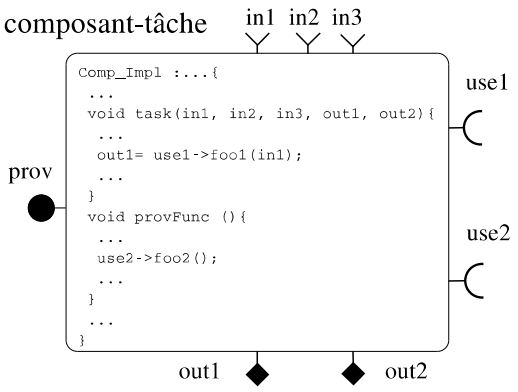


FIG. 8.3 – Une tâche dans un composant. Elle peut utiliser ses ports spatiaux.

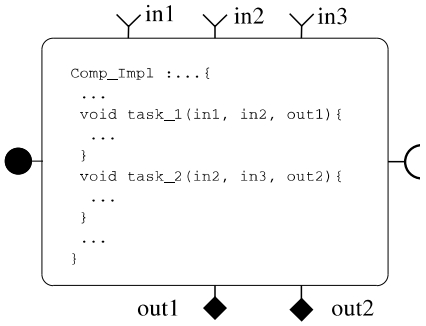


FIG. 8.4 – Exemple d'un composant multi-tâche.

temps de son exécution : son instanciation correspond au lancement de son exécution et la fin de celle-ci à la fin de son existence. Quant à la durée de vie d'une instance de composant, elle peut persister au-delà d'une ou de plusieurs exécutions des fonctionnalités qu'elle fournit.

A partir des similitudes entre composants et tâches, il est possible de penser à reprendre l'un de ces deux concepts et de l'étendre avec les types de ports manquants. Cependant, enrichir le concept de composant semble être le mieux adapté. En effet, le cycle de vie d'une tâche la rend plus proche du concept de fonctionnalité fournie, avec la particularité que la composition temporelle détermine quand elle doit être exécutée. En ce qui concerne un composant, il est déjà conçu pour héberger plusieurs types de fonctionnalités. Nous avons donc imaginé un mécanisme pour introduire le concept de tâche au sein d'un composant. Elle se présente tout simplement comme une fonctionnalité à implémenter par le développeur du composant. La figure 8.3 montre comment cela peut être perçu.

L'alternative de représenter une tâche comme une fonctionnalité d'un composant permet d'avoir des propriétés plus avancées. En particulier, il est possible de définir et d'implémenter une ou plusieurs tâches par composant, pouvant former ainsi des composants multi-tâches. Dans le cas d'une seule tâche, les données en entrée (respectivement en sortie) de la tâche correspondent à tous les ports d'entrée (resp. de sortie) du composant (ces types de ports sont introduits plus loin). Or, pour un composant multi-tâche, différentes tâches peuvent opérer sur différentes entrées. Le support de plusieurs tâches demande alors l'utilisation d'un mécanisme d'association des entrées/sorties d'une tâche à un sous-ensemble de ports d'entrée/sortie du composant. La figure 8.4 montre une association à travers le passage de paramètres des tâches implémentées. Il s'agit seulement d'un exemple illustratif. Une autre approche d'association pourrait être utilisée.

Désormais, le terme *composant-tâche* est utilisé afin de différencier un composant classique d'un composant hébergeant une ou plusieurs tâches. Il est à noter toutefois, qu'il ne s'agit que d'une dénomination, car un composant-tâche est en soi un composant.

8.3.2 Notions de port spatial et port temporel

Un composant étant maintenant enrichi avec le concept de tâche, il en est de même dans cette section avec les types de ports qu'un composant-tâche peut définir. Tout d'abord, nous

classifions ces ports en deux catégories : ports spatiaux et ports temporels. Cette séparation nous permettra de distinguer les ports dédiées à une composition spatiale de ceux dédiés à une composition temporelle.

Les ports spatiaux représentent tout type de ports retrouvés dans les modèles de composants classiques. Dû au fait que nous avons choisi de définir un composant-tâche à partir du concept de composant, les ports spatiaux sont directement hérités. Il est donc possible de composer des composant-tâches suivant des paradigmes de communication basés sur le *RPC/RMI*, le passage de message, le partage de données, etc.

En ce qui concerne les ports temporels, ils sont à spécifier. En s'appuyant sur les modèles de flux de travail étudiés, ils correspondent aux ports d'entrée et sortie d'une tâche. Leur principe se rapproche de celui des ports d'événement, qui d'une part définissent le type des messages (données) transférés et d'une autre part l'arrivée de messages (données) déclencherait l'exécution d'une action (tâche) sur un composant. Il est donc possible de simplement définir un port temporel en lui associant un type de donnée. La définition d'un port d'entrée en pseudo langage de définition d'interface IDL3, serait semblable à ***inPort TypeDonnee nomPort*** et celle d'un port de sortie à ***outPort TypeDonnee nomPort***. En suivant une approche semblable à celle que nous avons proposé pour le support des ports de données présentés dans le chapitre 5, le support des ports d'entrées et sorties serait simplement réalisable.

Par la suite, les ports temporels sont représentés sur les cotés haut (pour les entrées) et bas (pour les sorties) d'une représentation graphique d'un composant. Les ports spatiaux seront sur les autres cotés, comme cela est illustré à la figure 8.3 et la figure 8.4.

8.3.3 Notion de cycle de vie d'une instance de composant-tâche

Être capable de déterminer le cycle de vie des composant-tâches[1] à partir d'une composition est l'un des objectifs énumérés dans la section précédente. Pour cela, une approche est de définir des règles à partir desquels le cycle de vie de chaque composant-tâche peut être déduit. Par exemple, le cycle de vie d'une instance d'un composant-tâche n'ayant que des ports d'entrée et de sortie serait contrôlé en fonction de sa composition temporelle : elle peut être créée quand toutes les données en entrées sont disponibles et peut être détruite lorsque les sorties sont sauvegardées ou transférées. Cependant, la règle serait différente pour une instance d'un composant-tâche ayant en plus des ports spatiaux : elle doit persister tant qu'un autre composant-tâche aurait besoin d'accéder à ses ports spatiaux. L'objectif de cette section n'est pas de définir ces règles, car elles dépendent d'un modèle de composition, mais de décrire les états propres à un composant-tâche qui sont à contrôler par ces règles.

La figure 8.5 résume les états d'un composant-tâche sous forme d'un graphe de transition d'état. Ils sont globalement repris de modèles de composants existants avec en plus la considération du concept de tâche introduit. Dans ce qui suit, nous décrivons ces états tel qu'ils seront perçus dans un modèle de composition spatio-temporel.

Créée : la création d'une instance de composant-tâche est généralement effectuée par une opération du genre *create_instance(..)*. Elle peut être statique, paresseuse, ou mi-paresseuse.

[1] Par abus de langage, nous employons *cycle de vie d'un composant-tâche* pour désigner le cycle de vie d'une instance du composant-tâche pendant l'exécution d'une application.

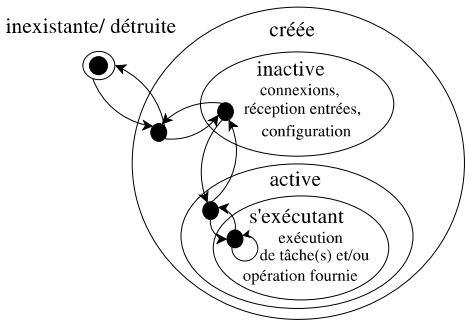


FIG. 8.5 – Graphe d'états d'une instance d'un composant-tâche.

Une instanciation statique signifie la création avant le lancement de l'exécution d'une application. La création paresseuse consiste en l'instanciation par nécessité, par exemple, une fois le flot de contrôle induit par la composition temporelle atteint l'utilisation d'un composant-tâche. Alors que l'instanciation mi-paresseuse permet la création d'une instance avant que le flot de contrôle atteigne son utilisation. Les différentes façons de créer une instance de composant-tâche dépendent des politiques adoptées par un environnement de déploiement/exécution.

Inactive : il s'agit d'un état durant lequel une instance de composant-tâche peut être connectée/déconnectée ou configurée/re-configurée, avant d'être (re)activée. Cet état peut figurer plusieurs fois dans la vie d'une instance, lorsqu'il s'agit de modifier la structure d'une application, impliquant une déconnexion et re-connexion des ports de cette instance, ou bien lorsqu'il s'agit de changer un éventuel état interne, représenté par un ensemble d'attributs par exemple.

Active : la possibilité d'exécuter une fonctionnalité d'un composant-tâche dépend essentiellement de cet état. L'activation d'une instance de composant-tâche implique que celle-ci est dans un état prêt pour recevoir et exécuter une requête et/ou une tâche. Le terme prêt se traduit par le fait qu'une instance a bien été configurée : ports connectés, attributs et données en entrée positionnées.

S'exécutant : Une instance de composant-tâche transite de l'état active à l'état d'exécution au lancement de l'exécution d'une requête reçue sur un ou plusieurs de ses ports (méthode fournie, tâche ou traitement d'un événement). Cet état reste valide jusqu'à la fin de toute exécution, dans quel cas, l'instance retourne à l'état active.

Non-existante ou détruite : habituellement, la destruction d'une instance de composant est effectuée à la fin de l'exécution d'une application, explicitement par l'utilisateur ou bien lorsqu'une restructuration de l'architecture de l'application demande le remplacement d'une instance. Cela est dû au fait qu'une composition spatiale ne détermine par le début ni la fin des communications entre composants. L'objectif dans un modèle de composition spatio-temporel est de pouvoir, grâce à la dimension temporelle, déterminer à quel moment une instance d'un composant-tâche n'est plus utilisée. À partir de là, celle-ci peut être détruite suivi potentiellement de la libération de la ou les ressource(s) qu'elle aurait occupées.

Enfin, la gestion du cycle de vie global d'une application consisterait à gérer simultanément des graphes d'états de tous les composant-tâches composés. Dans ce chapitre, nous nous contenterons des règles simples décrites en début de cette section. La durée de vie d'un composant-tâche, de sa création à sa destruction, dépendra de l'ordre d'exécution des tâches exprimé par un modèle de composition temporelle, plus les dépendances spatiales vis à vis d'autres composants. Désormais, ces dépendances se traduisent par des contraintes d'activation des composants. Autrement dit, pour assurer la disponibilité d'une fonctionnalité fournie par un composant-tâche A et utilisée par un autre composant-tâche B, A doit être activé avant que B ne puisse l'être. Quant à la désactivation de B, elle précédera celle de A. D'autres règles sont définies dans le chapitre suivant, en fonction du modèle de composition qui y sera présenté.

8.3.4 Conclusion

Nous venons de définir trois notions élémentaires sur lesquelles repose notre étude d'un modèle de composition spatio-temporel. Notre approche, basée sur une fusion des concepts de composant et de tâche, a débouché sur la définition de la notion de composant-tâche : une extension d'un composant avec la notion de tâche et ports temporels. Le cycle de vie d'un composant a été modifié légèrement pour prendre en compte ces extensions. Ainsi, l'étude qui suit se base sur différentes façons de composer la notion de composant-tâche.

8.4 Choix d'un modèle de conception

L'objectif de cette section est d'étudier des alternatives de conception d'un modèle de composition spatio-temporel. Dans cette étude, nous nous focalisons sur la description d'un comportement temporel d'une application. Il s'agit d'étudier des formes d'assemblage de composant-tâches suivant différents types de compositions temporelles. Nous nous intéressons à des solutions suivant deux approches. Une composition temporelle peut être construite grâce à un modèle de flot de données ou un modèle de flux de travail, basé sur la conception concurrente d'un flot de contrôle et d'un flot de données.

Nous supposons dans notre étude qu'un modèle de composition est hiérarchique et que la composition dans l'espace d'un composant-tâche est supportée, suivant le principe classique de connexion de ports.

8.4.1 Modèle temporel basé sur la composition d'un flot de données

Une première approche pour concevoir un modèle de composition temporel est de suivre le principe d'un flot de données. Pour cela, il suffit de décrire les dépendances de données entre composant-tâches à travers la connexion de ports d'entrée avec des ports de sortie. Une connexion de deux ports temporels suivrait la même philosophie que celle de deux ports spatiaux : une connexion est à établir entre ports de types compatibles, dans notre cas, le type des données associées. Ainsi, pour décrire une composition, il serait possible d'étendre un langage de description d'architecture existant avec le support des ports temporels. La réalisation de cette extension suivrait une approche similaire aux support des ports de données que nous avons introduits dans une composition spatiale (chapitre 5).

Par ailleurs, il reviendrait à un environnement de déploiement/exécution de déterminer le cycle de vie des composants-tâches, en fonction de l'ordre d'exécution des tâches exprimé

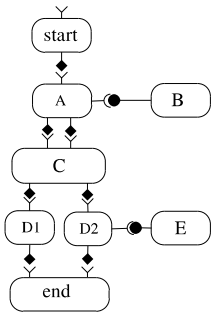


FIG. 8.6 – Composition basée sur un modèle de flot de données.

par un flot de données. Cependant, un ordre implique un début et une fin. Il est donc nécessaire d'être capable de déterminer les tâches à lancer en premier et à partir desquels l'enchaînement des suivantes serait déduit. Pour cela, il suffit de respecter le principe d'un flot de données consistant à ordonner les tâches en fonction de la disponibilité des données. Les tâches à lancer en premier sont celles des composant-tâches dont les ports d'entrée ne sont pas connectés à d'autres composant-tâches. Les données en entrée de ces tâches peuvent être spécifiées de la même manière qu'un attribut de composant, en utilisant par exemple des fichiers de configuration (CCM) ou en positionnant leur valeur dans l'assemblage (FRACTAL).

Un exemple de composition est illustré graphiquement à la figure 8.6. En partant des principes décrits dans cette section, l'exécution de l'application commencerait par le lancement de la tâche du composant-tâche *start* et se terminerait par la fin de celle du composant-tâche *end*. Entre ces deux points, l'ordre d'exécution serait : « *tâche(start) ; tâche(A) ; tâche(C) ; (tâche(D1)//tâche(D2)) ; tâche(end)* » . La durée de vie de chaque composant-tâche est essentiellement déterminée par cette séquence d'exécution avec en plus la considération des dépendances spatiales vis à vis des composants *B* et *E*. Ces dépendances impliquent une contrainte d'activation de *B* (respectivement *E*) avant l'activation de *A* (*D2*) et sa désactivation après celle de *A* (*D2*).

En résumé, l'alternative de définir un modèle de composition spatio-temporel basé sur un modèle de flot de données semble être simple à réaliser. Cependant, il serait plus intéressant d'enrichir la dimension temporelle avec un modèle de composition de flot de contrôle, qui se rapprocherait le plus d'une façon naturelle d'exprimer un ordre d'exécution de tâches. Dans la section suivante nous nous intéressons à cette deuxième possibilité.

8.4.2 Modèle temporel basé sur la composition d'un flux de travail

Contrairement à un langage de description d'architecture qui est basé sur une approche de composition descriptive, un langage de flux de travail permet de rédiger un programme. Concevoir un modèle de composition spatio-temporel qui suivrait une telle approche reviendrait à spécifier un modèle de composition d'un flot de données, d'un flot de contrôle ainsi qu'un modèle de composition spatiale. En gardant les principes de composition d'un flot de données décrits dans la section 8.4.1, cette section se focalise sur l'aspect flot de contrôle.

Nous avons vu dans le chapitre 3, qu'un modèle de flot de contrôle définit des structures de contrôles, tels que la séquence, les tests ou les boucles. L'étude qui suit repose sur la façon de représenter de telles structures, ceci en suivant deux approches : encapsuler les structures

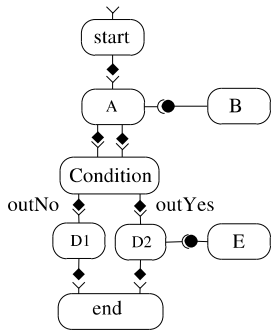


FIG. 8.7 – Composition basée sur un modèle de flot de contrôle implicite.

de contrôles dans des composant-tâches, rendant le flot de contrôle implicite ou adopter un langage de flux de travail explicitant le contrôle. Cette étude s'inspire des modèles de flux de travail vus dans le chapitre 3.

8.4.2.1 Flot de contrôle implicite

Encapsuler une structure de contrôle dans un composant-tâche, reviendrait à enfouir le contrôle dans l'implémentation d'une tâche. Le modèle de composition temporel serait alors proche de celui d'un flot de données. Le principe d'assemblage reste le même : il suffit d'exprimer les dépendances de données à travers les connexions de ports d'entrée avec des ports de sortie. Cependant, la différence réside dans la sémantique reflétée par un assemblage. Dans un flot de données, il est attendu à ce que tous les composant-tâches impliqués dans une composition temporelle reçoivent des données sur leurs entrées et donc, toutes les tâches du flot sont atteignables par celui-ci. Or, dans le modèle présent, l'envoi d'une donnée sur une sortie peut être conditionné par le contrôle enfoui dans un composant-tâche. La connexion d'un port d'entrée à un port de sortie ne garantit pas la circulation des données entre composant-tâches connectés. Autrement dit, les branches dessinées par le flot de données de l'assemblage peuvent ne pas être entièrement exécutées.

Pour mieux percevoir le principe d'un tel modèle, considérons l'exemple de la figure 8.7 et supposons que le composant-tâche ***`Condition`*** simule une structure de contrôle ***`if`*** et conditionne la production des données en sortie (***`outYes`*** et ***`outNo`***). Nous remarquons que l'assemblage est similaire à celui illustré à la figure 8.6, basé sur un modèle de flot de données. Toutefois, dans le cas présent, les données en sortie de ***`Condition`*** peuvent être disponibles uniquement sur l'un de ses port en sortie. Une seule branche (***`D1`*** ou ***`D2`***) peut être exécutée et une seule donnée en entrée du composant-tâche ***`end`*** peut être reçue. Un mécanisme est alors nécessaire pour éviter une attente infinie sur une sortie ou une entrée d'un composant-tâche. Il peut s'agir d'un système de détection de la fin de l'exécution d'une tâche, implémenté par une plate-forme d'exécution. Il reviendrait à celle-ci de déterminer les données disponibles en sortie à la fin de l'exécution d'une tâche et de signaler l'absence d'une sortie à un composant-tâche qui en dépend, par exemple en positionnant une valeur particulière « *vide* » sur le port d'entrée concerné. L'expressivité d'un assemblage est ainsi différenciée de celle d'un assemblage suivant un modèle de flot de donnée.

Même si la composition dans le modèle de flot de contrôle repose sur le principe de base

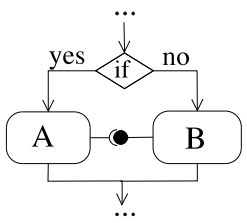


FIG. 8.8 – Exemple de composition spatio-temporelle suscitant différents comportements.

de connexion de ports, la conception et la gestion d'un assemblage peuvent être complexes. Un concepteur peut avoir besoin de connaître le comportement interne des composant-tâches pour maîtriser le contrôle et l'acheminement des données dans l'assemblage. Pour l'exemple de la figure 8.6, si le composant-tâche ***end*** s'attend à avoir des données sur deux entrées et sans savoir que le composant-tâche ***Condition*** exclut l'exécution de ***D1*** ou ***D2***, il serait difficile d'obtenir le comportement attendu. De plus, cette connaissance peut dépendre de différentes implémentations du composant-tâche. La nécessité de comprendre le comportement interne des composants peut rendre difficile la conception. Par ailleurs, le rôle d'une plate-forme d'exécution est aussi de calculer un ordonnancement des composant-tâches en fonction du comportement exprimé par leur assemblage. Ce calcul tient compte des chemins d'exécution possibles d'un assemblage dont le nombre dépend des ports temporels et leurs connexions et peut croître rapidement et inutilement comparé à un calcul qui reposerait sur une visibilité du flot de contrôle. Pour résumer, il est important de rendre un assemblage plus expressif afin de résoudre la complexité du modèle de flot de contrôle implicite.

8.4.2.2 Flot de contrôle explicite

Une dernière approche consisterait à extérioriser les structures de contrôle des composant-tâches, explicitant le contrôle exprimé par une composition. De telles structures feraient en conséquence partie intégrante d'un langage d'assemblage. Il ne s'agit plus de s'appuyer sur un langage de description d'architecture, mais plutôt sur un langage de composition basée sur un principe de programmation, comme dans les modèles de flux de travail tel que ASKALON ou GRICOL. Une telle approche mettrait l'accent sur l'expressivité d'un comportement déterminé à travers le procédé de composition.

Un langage de composition spatio-temporel offrant des structures de contrôle peut être vu comme un langage de flux de travail dans lequel les concepts de composition fondamentaux seraient ceux définis dans la section 8.3. L'unité de base de composition ne serait pas une tâche mais un composant-tâche. Les dépendances de données seraient classiquement exprimées à travers un flot de données. Les structures de contrôle seraient définies par le langage avec aussi des structures ou instructions dédiées à la connexion de ports spatiaux. Quant au cycle de vie des composant-tâches, il serait à déduire depuis la suite des instructions/actions construite dans l'assemblage.

Cependant, envisager cette approche n'a pas pour unique objectif d'expliciter le comportement temporel. Ce dernier ne suffit pas pour exprimer le comportement global que nous souhaitons pouvoir déterminer à partir de l'assemblage. Les dépendances spatiales

entre composant-tâches jouent aussi un rôle important dans cette détermination. Prenons la composition de la figure 8.8. Dans cette composition, la structure de contrôle *if* conditionne l'exécution de la tâche des composant-tâches *A* et *B*. A priori, ce conditionnement remettrait en cause la possibilité de réaliser la composition spatiale décrite. En effet, cela peut être le cas en valorisant la dimension temporelle dans la déduction du comportement. Cependant, il est important de rappeler qu'une tâche est une fonctionnalité d'un composant-tâche comme une autre. Par exemple, le fait que la branche *yes* soit prise, ne devrait pas écarter la possibilité de lui fournir la fonctionnalité offerte par *B*. L'extériorisation de la structure de contrôle n'a donc pas suffi pour déterminer un comportement. Pour cela, il est nécessaire que le langage d'assemblage fournisse des mécanismes permettant à la fois d'exprimer différents comportements possibles mais aussi de déterminer le comportement attendu. Sans pour autant s'étendre sur une solution, le chapitre suivant montre comment l'ordre dans lesquels les structures de contrôles et les instructions de connexions paraissent dans un assemblage, peut être simplement exploité pour définir de tels mécanismes.

Finalement, la présente alternative serait intéressante pour atteindre un niveau d'expressivité souhaité. Certes, elle nécessite de définir plus de concepts dans un langage d'assemblage. Toutefois, elle favoriserait la construction de la logique algorithmique d'une application par assemblage au lieu qu'elle soit enfouie dans le code. Quant à un environnement de déploiement/exécution correspondant, il peut à priori sembler complexe à implémenter. Cependant, il ne serait pas beaucoup plus complexe qu'un moteur de flux de travail. Le comportement d'une application est composé en premier par une logique temporelle. Les connexions spatiales ne font que rajouter des contraintes d'ordre de création/destruction, activation/désactivation, etc., de composant-tâches dépendants. Il serait alors plus intéressant de partir des implémentations existantes pour en hériter la gestion de la logique temporelle et l'adapter au modèle souhaité.

8.4.3 Discussion

Comme nous venons de le voir, différentes approches peuvent être considérées pour concevoir un modèle de composition spatio-temporel. Il en ressort des solutions plus ou moins simples avec une capacité d'exprimer la logique algorithmique d'une application de manière plus ou moins explicite à partir d'un assemblage.

Un modèle où la composition temporelle serait assimilée à un flot de données ou un flux de travail avec flot de contrôle implicite, ne nécessiterait pas un changement dans le principe d'assembler avec un langage de description d'architecture. Il suffirait d'y introduire le support des ports temporels et les connexions de ces ports. Ainsi, la philosophie de composition suivrait toujours une approche descriptive, avec néanmoins un comportement temporel induit dans la composition. Cependant, nous avons vu que ces deux solutions demeurent loin d'être suffisamment expressives et/ou déterministes du comportement d'une application. Ainsi, nous avons étudié comment un langage de composition qui soit analogue à un langage de flux de travail explicitant le contrôle permettrait de mieux atteindre nos objectifs. Nous en avons déduit une approche qui pourrait tout à fait offrir une solution de composition par programmation en modifiant et étendant simplement quelques principes de composition d'un langage de flux de travail classique.

Aux cours de l'étude nous avons brièvement discuté quelques propriétés qu'un environnement de déploiement/exécution implémentant un modèle pourrait avoir. Bien entendu, toute approche proposée demanderait un effort pour être capable de gérer un ordonnance-

ment dynamique des composant-tâches et leur cycle de vie. Nous avons vu que la capacité de déterminer simplement la logique algorithmique d'une application peut jouer un rôle important sur le niveau de complexité de cette gestion. Autrement, un environnement ne serait pas loin de suivre le principe de gestion d'un moteur de flux de travail, avec en plus la considération de contraintes de dépendances spatiales entre composant-tâches.

Au final, la solution d'un modèle de composition qui s'appuierait sur un modèle de flux de travail avec flot de contrôle explicite se rapproche le plus des objectifs fixés dans la section 8.2.

8.5 Conclusion

Dans ce chapitre, nous avons étudié différentes approches qui permettrait de définir un modèle de composition spatio-temporel. L'étude s'appuie essentiellement sur l'objectif d'être capable de composer et d'exprimer la logique algorithmique d'une application à travers le procédé d'assemblage. Cette logique algorithmique considère à la fois une logique temporelle, à travers un séquencement d'actions à exécuter et leurs dépendances de données, ainsi que les dépendances spatiales pouvant exister entre entités de composition.

Tout d'abord, nous avons étudié les principales alternatives de combinaison d'une composition temporelle et spatiale dans l'existant. Nous en avons conclu que la considération des deux dimensions à des niveaux séparés de la conception d'une application – méta-informations associées à un assemblage ou masquage dans l'implémentation des composantes – est à l'origine des limitations de modèles considérés. Nous avons ainsi décidé de focaliser notre étude sur l'alternative de fusionner les dimensions temporelle et spatiale pour être prises en compte au même niveau, celui de l'assemblage d'une application.

Ensuite, les caractéristiques souhaitées et les concepts fondamentaux d'un modèle spatio-temporel ont été définis. Ce modèle se doit d'être expressif, hiérarchique et déterministe du comportement attendu d'une exécution pour une gestion cohérente et simple du cycle de vie des composantes assemblées. Nous avons précisé que l'objectif n'est pas de redéfinir complètement un modèle, mais de partir sur des notions existantes, c'est à dire, de composants et tâches, pour lesquels de nombreux concepts rattachés aux compositions spatiales et temporelles sont déjà bien définis dans les modèles de composants et les modèles de flux de travail existants. Ainsi, nous avons défini le concept de composant-tâche, une intégration du concept de tâche dans un composant, avec des types de ports dits spatiaux et temporels et à qui est rattaché un cycle de vie bien défini.

Enfin, nous avons analysé des solutions de conception d'un modèle de composition spatio-temporel selon trois modèles temporels : modèle de flot de données et modèles de flux de travail avec flot de contrôle masqué dans l'implémentation des composant-tâches ou bien, avec flot de contrôle explicite grâce à des structures de contrôle fournies par un langage d'assemblage. L'ensemble des solutions permettent de composer des composant-tâches suivant les dimensions spatiale et temporelle au même niveau de composition. Toutefois, nous sommes arrivés à la conclusion qu'un modèle avec flot de contrôle explicite répondrait mieux à nos objectifs de départ. Le chapitre suivant présente comment un modèle de composition spatio-temporel suivant une telle approche pourrait être spécifié.

Chapitre 9

Le modèle STCM

Sommaire

Suite à l'analyse de différentes alternatives pour la conception d'un modèle de composition spatio-temporel, présentée dans le chapitre 8, ce chapitre présente le modèle STCM, *Spatio-Temporal Component Model*. Il s'agit d'un modèle de composants où la composition repose sur des principes d'assemblage de composants logiciels avec une approche de composition d'un flux de travail. L'objectif de STCM est de réunir les apports des deux familles de modèles de programmation (chapitre 3 et chapitre 4), longuement séparés par l'utilisation exclusive de la dimension temporelle ou spatiale dans leur modèle d'assemblage respectif.

L'objectif de ce chapitre est de présenter nos travaux autour de la spécification de STCM. Tout d'abord, notre stratégie de conception de cette spécification est présentée. Ensuite, une spécification en terme d'interfaces de programmation illustre comment définir et utiliser le concept de *composant-tâche* et ses ports, introduit dans le chapitre 8 et qui représente notre unité de base de composition. Par la suite, un langage de composition est présenté. Puis,

Concepts requis	Concepts existants	Stratégie choisie
Composant-tâche	Opérations fournies/requises et tâche	Étendre GCM avec le concept de tâche
Ports	**Spatiaux :** ports de GCM **Temporels :** ports de données d'entrée/sortie d'AGWL	Étendre GCM avec le concept de port temporel
Cycle de vie	États et transitions	Étendre les états de GCM Définir les règles de transitions
Composition	**Spatiale :** connexions de GCM **Temporelle :** flot de données et de contrôle d'AGWL	Adapter AGWL pour le support de composant-tâches et connexions de GCM

FIG. 9.1 – Stratégie de réutilisation des concepts de GCM et AGWL pour définir STCM.

un exemple d'utilisation de STCM illustre l'intérêt et l'aboutissement de nos travaux. Enfin, une conclusion permet de résumer les spécificités de STCM.

9.1 Stratégie de conception de STCM

Au lieu de spécifier STCM à partir de rien, nous avons choisi une approche de réutilisation, de fusion et d'extension de spécifications existantes. Il s'agit du même principe adopté lors de l'étude des concepts d'un modèle spatio-temporel de manière générale dans le chapitre 8. Pour rappel, un composant-tâche y a été défini comme étant un composant, dans lequel le concept de tâche a été introduit. Il peut avoir des ports spatiaux, mais aussi des ports temporels pour les données en entrée/sortie de tâches. Nous avons aussi défini ce que pourrait être le cycle de vie d'un composant tâche et enfin, nous avons choisi une approche de composition analogue à l'utilisation d'un langage de flux de travail. Nous poursuivons sur cette même approche de réutilisation, mais cette fois-ci à partir de spécifications existantes de composants et de langage de flux de travail.

Pour concevoir STCM, deux modèles de référence ont été choisis : GCM [87] et ASKALON -AGWL [59]. GCM est un modèle de composants pour les applications de grilles proposé par le réseau d'excellence *CoreGrid*. Il a comme modèle de référence FRACTAL que nous avons présenté dans le chapitre 3. Quant au choix d'ASKALON -AGWL, il est motivé non seulement par la présence d'une spécification d'un langage d'assemblage facilement extensible, mais aussi par sa propriété d'abstraire le modèle de programmation de la nature des grilles de calcul, par sa spécificité hiérarchique et son expressivité à travers sa richesse en terme de structures de contrôle. Ces propriétés sont très proches de ce que nous attendons d'un modèle d'assemblage.

La figure 9.1 résume notre stratégie de réutilisation de GCM et AGWL. Premièrement, l'unité de base de composition, représentée par un composant-tâche, est avant tout un composant. Il est plus direct de le définir à partir d'une extension d'un composant GCM. De plus contrairement à AGWL, il est possible d'acquérir directement la facilité d'étendre une définition d'un composant par un mécanisme d'héritage, ce qui peut promouvoir la réutilisabilité. Deuxièmement, puisqu'un composant-tâche est un composant GCM étendu, il en sera de même pour le support des ports temporels dans GCM. Par rapport à AGWL, le principe de rattacher une donnée à un port d'entrée ou de sortie, semble être satisfaisant pour définir les ports temporels. Troisièmement, le cycle de vie d'un composant-tâche décrit dans

le chapitre précédent à travers un graphe de transition d'états, est à définir dans la spécification étendue de GCM. Il se traduirait par le support des actions de création/destruction, d'activation/désactivation, exécution, etc., et de sauvegarde de l'état correspondant. La gestion du cycle de vie de composant-tâches assemblés dépendra de règles à déterminer en fonction du modèle d'assemblage adopté. Enfin, notre stratégie pour spécifier ce dernier est de partir du langage AGWL et d'y introduire les concepts absents. Cela consiste à remplacer le concept d'*activité* de AGWL par le concept de composant-tâche, d'étendre ce langage avec le support des ports spatiaux et leurs connexions. A partir du langage étendu, une sémantique serait à spécifier pour être capable de déduire une logique algorithmique d'une application. Cette logique prend en compte le comportement construit à travers un flot de contrôle ainsi que les dépendances spatiales entre composant-tâches. Comme nous l'avons introduit dans le chapitre précédent, ces dépendances sont déterminantes des contraintes à respecter pour gérer le cycle de vie des composant-tâches assemblées. Dans la section 9.2 et la section 9.3, notre stratégie de réutilisation de GCM et AGWL est mise en application et les extensions/modifications apportées sont spécifiées.

9.2 Description et utilisation des concepts fondamentaux

Il existe différentes façons d'étendre un composant GCM pour supporter le concept de tâche et de port temporel. L'objectif de cette section est de présenter les pré-requis pour ce support ainsi qu'une solution permettant d'y répondre. Cette solution se présente sous forme d'un ensemble d'opérations étendant l'*API* de la spécification IDL de GCM, décrite dans [87]. Les interfaces de définition de types, de création et de connexion/déconnexion de composants, de définition de contrôleurs, ainsi que celles dédiées à la gestion du cycle de vie des composants GCM, sont les interfaces sur lesquelles nous nous focalisons pour spécifier les concepts fondamentaux de STCM.

9.2.1 Spécification d'un composant-tâche

Un composant-tâche est défini par l'ensemble de ses ports, y compris les ports temporels. Il est nécessaire alors de rajouter le concept de port d'entrée et de sortie dans la définition d'un type de composant-tâche. Ensuite, le concept de tâche est à introduire dans la spécification de GCM. Après avoir vu une solution pour ces deux points, nous présentons une extension de la spécification dédiées à l'utilisation des concepts définis.

9.2.1.1 Définition d'un port temporel

Contrairement à un port client/serveur avec une sémantique d'appel de procédure ou de méthode, un port temporel transporte une donnée typée. Dans les modèles de flux de travail existants, le type peut être un type primitif (***int***, ***string***, etc.), un fichier, un paquetage de données, etc. Pour choisir parmi les différentes solutions nous avons décidé de préserver le principe de typage de ports dans GCM. La figure 9.2 illustre une première partie de notre proposition d'extension de l'interface ***TypeFactory*** de GCM qui est dédiée à la définition des types. L'opération rajoutée ***createFcTmpType*** permet de définir le type d'un port d'entrée (***isInput = true***) ou d'un port de sortie (***isInput = false***) dont le nom est ***name***. Le type de la donnée associée est déterminé par sa signature ***dataType***. Enfin, nous avons

```
interface ExtendedTypeFactory: TypeFactory{

 // Première partie de l'extension
 // Hérite de InterfaceType createFcItfType(...)
 // pour la définition de ports clients/serveurs

 TemporalPortType createFcTmpType (in string name,
                                   in string dataType,
                                   in boolean isInput,
                                   boolean isOptional,
                                   ...)
                  raises(InstantiationException);

 ... // Deuxième partie (figure 9.3)
};

interface TemporalPortType : Type {

 string getFcTmpName  (); string getFcTmpSignature ();
 boolean isFcInputTmp (); boolean isFcOptionalTmp ();
 ...
};
```

FIG. 9.2 – Spécification d'un type de ports d'entrée et de sortie.

pensé à garder quelques propriétés pouvant être rattachées à un port temporel, comme cela est le cas pour les ports classiques. Le paramètre ***isOptional*** par exemple, permet de spécifier si la disponibilité d'une donnée en entrée ou en sortie est indispensable pour le bon déroulement d'une exécution. Enfin, le type en retour de l'opération ***createFcTmpType*** n'est qu'une interface permettant de récupérer des information sur le type défini (voir le bas de la figure 9.2).

9.2.1.2 Définition d'un composant-tâche

Maintenant qu'un type de port spatial et temporel peut être défini, un type de composant-tâche peut être spécifié. Vu qu'un port temporel est différencié des ports clients/-serveurs par la nature du type et la sémantique qui lui est associée, il en est de même pour définir l'ensemble des ports d'un composant-tâche. Pour cela, nous avons rajouté l'opération ***createFcType*** avec un prototype différent (voir figure 9.3), permettant à un composant-tâche de définir en une seule fois l'ensemble de ses ports. Enfin, pour des objectifs d'introspection sur l'ensemble des ports du composant-tâche, l'interface ***ComponentType*** dédiée aux ports spatiaux est étendue à travers l'interface ***ComponentTaskType*** pour prendre en compte les ports temporels. Il est à noter que l'opération d'origine (héritée) n'a plus d'utilité : un composant-tâche peut ne pas définir de ports temporels ni implémenter de tâche. Toutefois, nous restons sur le principe d'extension et non de redéfinition d'une spécification.

En ce qui concerne la création d'une instance de composant-tâche, il n'est pas nécessaire de modifier la spécification GCM. En effet, l'interface dédiée, appelée ***GenericFactory***, et en particulier, l'opération d'instanciation ***Component newFcInstance (in Type type, in Object controllerDesc, in Object contentDesc) raises(InstantiationException)***[1] ne manipule que des types gé-

[1] L'opération ***newFcInstance(..)*** crée un composant de type générique ***Type*** avec une membrane décrite

```
interface ExtendedTypeFactory: TypeFactory{

 ... // Première partie de l'extension (figure 9.2)

// Deuxième partie
// Hérite de ComponentType createFcType (InterfaceType[] itfTypes)
// pour la définition d'un composant GCM

 ComponentTaskType createFcType (in InterfaceType   [] interfaceTypes,
                                 in TemporalPortType[] tmpPortTypes)
                  raises(InstantiationException);
};

interface ComponentTaskType : ComponentType {

 TemporalPortTypeArray getFcTmpTypes ();
 TemporalPortType getFcTmpType (in string name)
                  raises(NoSuchTmpPortException);
};
```

FIG. 9.3 – Spécification d'un type de composant-tâche.

nériques n'ayant pas besoin d'être modifiés ni étendus. Sans oublier qu'un composant-tâche est lui même un composant et que nous nous sommes reposé sur le mécanisme d'héritage des types génériques dans le processus d'extension.

9.2.1.3 Spécification d'une tâche

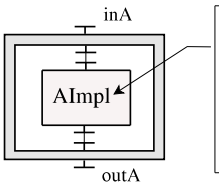
Dans le chapitre précédent, le principe d'intégration du concept de tâche dans un composant-tâche a été décrit de manière générale. Nous y avons défini une tâche comme étant une fonctionnalité à implémenter par l'utilisateur. La façon dont celle-ci peut être définie peut dépendre de certaines hypothèses. En particulier, la définition peut varier selon qu'on soit dans un contexte mono ou multi-tâche. Les deux cas semblent intéressants à approfondir. La définition peut aussi varier selon qu'une tâche est visible ou pas depuis l'extérieur du composant-tâche (à travers un port serveur). Cependant, la visibilité impliquerait une dépendance avec l'implémentation du composant. Le fait qu'un composant soit mono ou multi-tâche ou que le nombre de tâches diffère d'une implémentation à l'autre est une spécificité de l'implémentation. Il est préférable que cette spécificité soit masquée de l'extérieur d'un composant-tâche, ce qui permettrait de changer une implémentation sans avoir à changer le type d'un composant. Pour la suite, nous supposons qu'une tâche est rattachée à un composant primitif, non explicitement accessible depuis l'extérieur (au delà de la membrane) du composant.

Dans ce qui suit, nous présentons en premier une solution mono-tâche, reprise ensuite pour adaptation à un contexte multi-tâche.

Cas mono-tâche : Le principe d'une tâche est que son exécution soit lancée à la réception des données sur les ports d'entrée d'un composant-tâche et qu'à la fin de son exécution, elle déclenche la récupération des données sur les ports de sorties. Une tâche a en consé-

par un descripteur ***controllerDesc*** et un contenu spécifié par ***contentDesc***

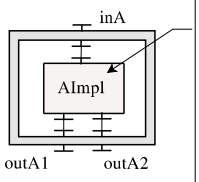
```
interface TaskController{
 void task();
};
interface TskCtrlA: TaskController{
  // complétée plus loin
};
```




```
class AImp implements TskCtrlA,
                      ...{
  void task (){...}
  ...
};
```

FIG. 9.4 – Spécification du concept de tâche dans un cadre mono-tâche.

```
interface MultiTaskController{
 void bindTaskInOut(in long taskId,
                    inout string[] inNames,
                    inout string[] outNames)
      raises(NoTaskDefined);
};
interface MltTskCtrlA: MultiTaskController{
  void task_1();
  void task_2();
  // complétée plus loin
};
```




```
class AImp implements TskCtrlA,..{
 ...
 void task_1(){..} //sortie outA1
 void task_2(){..} //sortie outA2
 void bindTaskInOut(....){
  if (taskId == 1) then {
        inNames[..] = "inA";
        outNames[..] = "ouA1";
  } ...
 }
};
```

FIG. 9.5 – Spécification du concept de tâche dans un cadre multi-tâche.

quence deux point de vue : un point de vue utilisateur qui implémente la tâche et un point de vue canevas qui gère le lancement de son exécution, le positionnement des entrées et la récupération des sorties. Dans GCM, un canevas est représenté par des contrôleurs. Selon le type de contrôleur, celui-ci peut être implémenté au niveau de la membrane, qui forme le conteneur du composant, ou bien, implémenté par l'utilisateur. Dans ce dernier cas, les opérations implémentées sont accessibles par un contrôleur du conteneur à travers des *callbacks*. Tel est le cas pour le contrôleur de liaison (**`BindingController`**) à implémenter du coté client. Suivant ce même principe, nous définissons une tâche comme une opération d'un nouveau type de contrôleur, nommé **`TaskController`**. L'interface correspondante est visible à la figure 9.4. A droite de la figure, un pseudo code représentant l'implémentation du composant-tâche **`A`** qui implémente l'opération **`task`**.

Cas multi-tâche : Définir plusieurs tâches au sein d'un même composant-tâche reviendrait à définir des triplets (tâche, entrées, sorties). Contrairement au cas mono-tâche, ici, les entrées et/ou sorties d'une tâche peuvent être un sous-ensemble des entrées et/ou sorties du composant-tâche. En conséquence, non seulement il est nécessaire d'avoir un mécanisme permettant de définir plusieurs tâches, mais il est aussi nécessaire d'être capable d'associer un sous-ensemble de ports temporels à une tâche. Cela aurait pour objectif de lancer une tâche indépendamment de la réception de données non nécessaires à sa propre exécution. La figure 9.5 illustre une spécification permettant à un utilisateur de définir simplement les triplets de départ. Suivant une approche de contrôleur de manière similaire au cas mono-tâche, une interface **`MultiTaskController`** est à implémenter par l'utilisateur. Premièrement, dans cette interface, les différentes tâches sont à définir. Pour permettre une automatisation de leur gestion, un prototype est imposé : une tâche doit être définie sous la forme **`void task_i()`**, où i, allant de 1 à N, avec N **`= nombre de tâches`**, sert d'identifiant de tâche. Le principe est proche de celui de la spécification de collections d'interfaces dans

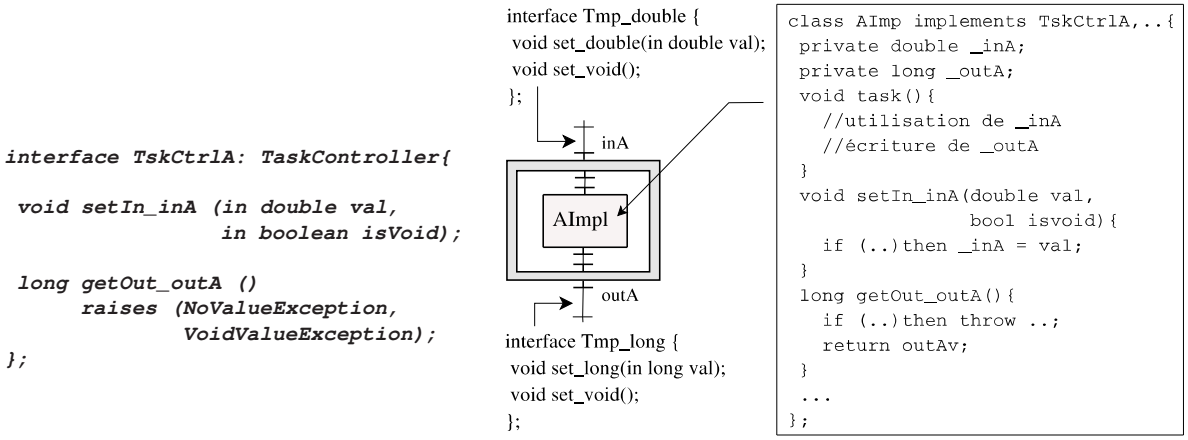


FIG. 9.6 – Interfaces associées aux ports temporels.

FRACTAL . Deuxièmement, pour associer des ports temporels à une tâche, une autre opération, nommée ***bindTaskInOut***, est aussi définie dans la même interface. Cette opération permet à l'utilisateur de spécifier la liste des entrées/sorties d'une tâche à travers les noms des ports correspondants. Cette opération est invoquée automatiquement, à travers un callback, avant l'activation d'un composant-tâche.

Nous venons de définir un composant-tâche GCM, des ports temporels et une mise en œuvre des tâches. Il reste à spécifier leur utilisation. C'est ce à quoi la section 9.2.2 s'intéresse.

9.2.2 Utilisation des ports temporels

Avant de spécifier comment un port temporel peut être utilisé dans STCM, il est important de rappeler ce qui est attendu de pouvoir effectuer à travers. Il est principalement attendu qu'une tâche puisse retrouver ses données en entrée et qu'elle puisse aussi positionner des données en sortie. Cela est à percevoir selon deux points de vue : **1)** un point de vue implémentation d'un composant-tâche, pour l'utilisation interne des données et **2)** un point de vue externe pour le transfert des données entres composant-tâches. Notre approche pour rattacher ces fonctionnalités à un port temporel est d'associer à ce dernier un ensemble d'interfaces de type client/serveur classique pour une utilisation à cet effet. La figure 9.6 montre les interfaces ***TskCtrlA*** et ***Tmp_double*** associées aux ports d'entrée et sortie du composant ***A***. Dans la suite de cette section, la description de ces interfaces est répartie suivant les points de vue interne et externe des ports temporels.

Point de vue interne : Pour être capable de positionner (respectivement d'envoyer) une donnée sur un port d'entrée (de sortie), nous avons choisi le principe d'attribut. Cela signifie que pour toute donnée en entrée, à sa réception, une opération de type *setter* est appelée sur l'implémentation d'un composant-tâche, responsable d'en effectuer une copie. De manière symétrique, une donnée en sortie est récupérable depuis l'implémentation grâce à une opération de type *getter*. L'interface ***TskCtrlA*** de la figure 9.6 définit ces opérations, implémentées dans le code métier du composant-tâche. Étant donné qu'un port d'un composant GCM a un nom unique et est typé, le prototype choisi pour ces opérations est

tout simplement défini sous la forme ***void setIn_PortName(DataType value, ..)*** et ***DataType getOut_PortName()***. Le paramètre ***isVoid*** et les exceptions présentes dans le prototype de ces opérations (voir figure 9.6) permettent de traiter l'absence d'une donnée en entrée (port non connecté par exemple) ou en sortie (donnée non produite) d'une tâche. Enfin, puisque les ports temporels sont fortement reliés au concept de tâche, nous avons choisi d'imposer leur définition dans une interface de type ***TaskController*** ou ***MultiTaskController*** décrites préalablement dans la section 9.2.1.3. Cependant, dans la solution proposée, il est à noter qu'une donnée en entrée ou en sortie n'est pas locale à une tâche. Elle peut être accessible par d'autres fonctionnalités d'un composant-tâche. L'intérêt est de permettre de positionner un état d'un composant à travers une entrée lorsque cet état dépend d'un calcul précédent.

Point de vue externe : afin d'expliquer simplement ce point de vue, considérons en premier un port d'entrée. Une entrée est une donnée à positionner lorsqu'elle est disponible et quand le flot de contrôle atteint le composant-tâche concerné. Il serait donc cohérent d'associer une opération externe de type *setter* à un port d'entrée. C'est pourquoi l'interface ***Tmp_double*** représentée dans la figure 9.6 est spécifiée. Ainsi, nous avons projeté la définition d'un port d'entrée sur un port classique de type serveur. Cependant, STCM est spécifié pour permettre l'établissement concret de connexions entre ports temporels. Cela a pour objectif d'offrir la possibilité de transférer les données directement entre composant-tâches. Un moteur responsable de la gestion de l'ordonnancement et du transfert de données, pourrait alors adopter plusieurs politiques. Par exemple, pour instancier un composant-tâche ***B*** dont la tâche suit celle d'un autre composant-tâche ***A*** et qui consommerait une donnée en sortie de celui-ci, le moteur pourrait décider d'instancier ***B***, connecter ses ports d'entrée pour transférer directement la sortie de ***A***, avant de le supprimer. En prenant en compte cette alternative, il est nécessaire d'assurer la compatibilité des ports connectées. Un port de sortie est donc à son tour projeté sur un port client de la même façon qu'un port d'entrée (***Tmp_double*** ou ***Tmp_long*** dans l'exemple). Une autre politique est toutefois possible. Un moteur peut décider de récupérer les données en sortie pour les positionner par lui même sur des entrées. Dans ce cas, le moteur peut introduire des composant-tâches (de type *proxy*) de gestion et stockage des données intermediaires dans le modèle d'exécution d'une application. Pour l'exemple, un composant-tâche *proxy* aurait un port en entrée pour la récupération de la sortie de ***A*** et un port en sortie à connecter à ***B*** à sa création.

Enfin, pour coordonner l'ensemble, la spécification proposée pour le support des concepts de tâches et de ports temporels a pour objectif d'offrir au code métier une transparence de gestion de la disponibilité des données en entrée/sortie d'un composant-tâche et de l'exécution d'une tâche. Nous proposons que cette gestion soit à la charge d'un contrôleur ***TaskManagerController***. Ce contrôleur a pour objectif de : **1)** implémenter les interfaces externes associées aux ports d'entrée, **2)** positionner les entrées d'une tâche à travers une interface ***TaskController*** implémentée par le composant-tâche, **3)** lancer l'exécution d'une tâche et attendre sa fin, **4)** récupérer les données en sortie, toujours à travers l'interface ***TaskController***, et les envoyer à travers les ports de sortie et **5)** maintenir un état sur l'exécution d'une tâche qui soit accessible depuis l'extérieur (voir la section 9.2.3 sur le cycle de vie). Quant à l'implémentation de ces différentes fonctionnalités, elle serait a priori à fournir par une plate-forme qui implémenterait le modèle STCM. La figure 9.7 illustre un

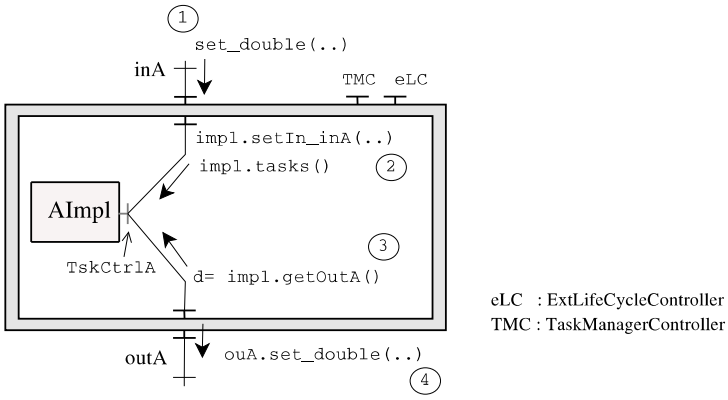


FIG. 9.7 – Aperçu de la gestion automatique de l'exécution d'une tâche.

```
interface LifeCycleController {
 string getFcState ();
 void startFc () raises(IllegalLifeCycleException);
 void stopFc () raises(IllegalLifeCycleException);
};

interface ExtLifeCycleController : LifeCycleController {
 void runFc () raises(IllegalLifeCycleException);
 void completeFc () raises(IllegalLifeCycleException);
 void removeFc () raises(IllegalLifeCycleException);
};
```

FIG. 9.8 – Spécification du cycle de vie d'un composant tâche.

exemple qui résume la gestion des données en entrée/sortie d'une tâche, rendue transparente grâce au contrôleur ***TaskManagerController***.

9.2.3 Cycle de vie d'un composant-tâche

Dans la section 8.3.3, un cycle de vie pour un composant-tâche a été présenté. Il définit les états qu'une instance peut avoir durant l'exécution d'une application. Ces états sont résultats d'actions de création/destruction, activation/désactivation ainsi que de lancement de l'exécution d'une requête ou d'une tâche sur cette instance. Le support de ce cycle de vie dans GCM se traduirait par le support de ses actions. Un contrôleur, nommé ***LifeCycleController***(voir la première partie de la figure 9.8), dédié à la gestion du cycle de vie d'un composant étant déjà défini dans GCM, il suffit d'y intégrer les éventuels actions manquantes. Même si la sémantique des opérations ***startFc*** et ***stopFc*** est assez faible dans GCM et FRACTAL , elle semble se rapprocher de la sémantique d'activation et désactivation d'un composant. Nous avons décidé alors de leur imposer cette dernière et d'étendre la spécification avec des opérations pour la gestion de l'état ***détruite*** et ***s'exécutant*** d'une instance de composant. L'interface ***ExtLifeCycleController*** illustrée dans la deuxième partie de la figure 9.8 montre le résultat de cette extension. Ainsi, les opérations ***startFc*** et ***stopFc*** héritées correspondent respectivement à l'activation et la désactivation d'une instance d'un composant-tâche. Quant aux opérations ***runFc*** et ***completeFc***, elles sont res-

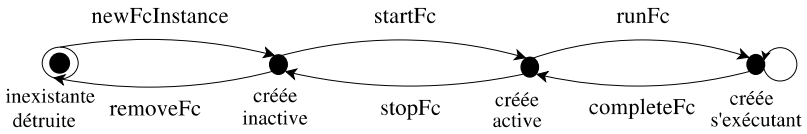


FIG. 9.9 – Transition des états pour la gestion du cycle de vie d'un composant-tâche.

pectivement invoquées au premier appel d'une fonction ou d'une tâche sur un composant et à la détection de la fin de toutes les exécutions d'opérations fournies, de tâches ainsi que celles des processus que ces dernières auraient générées.

Le graphe de transition d'états, aussi introduit dans le chapitre 8, est repris dans la figure 9.9 afin de faire le lien avec les actions qui gèrent, directement ou indirectement, le cycle de vie d'un composant-tâche. Il est à noter que l'opération ***`newFcInstance(..)`*** pour la création d'une instance est une opération externe à l'interface ***`ExtLifeCycleController`***. En ce qui concerne le moment où l'état d'une instance est mis à jour, il est décrit dans la section 8.3.3.

9.2.4 Conclusion

Dans la première partie de ce chapitre nous avons présenté une spécification des concepts fondamentaux de STCM dans une extension de GCM. Cette spécification permet l'introduction du concept de tâche dans un composant GCM, la définition et l'utilisation de ports temporels ainsi que la gestion du cycle de vie d'un composant-tâche.

Mis à part le type d'un port temporel qui introduit une nouvelle sémantique pouvant être rattachée à un port, l'extension s'est reposée sur des concepts existants. Une tâche et le cycle de vie sont associés à des contrôleurs. Quant aux ports temporels ils sont projetés sur des interfaces classiques de type client et serveur, selon un principe d'attributs. La spécification des contrôleurs a tenu compte des objectifs d'automatisation et de transparence de la gestion de la disponibilité des données en entrée/sortie d'une tâche et la gestion de l'exécution d'une ou de plusieurs tâches. Il a donc été possible de définir une spécification en minimisant les nouveaux concepts introduits. Nous ne prétendons pas qu'il s'agit de l'unique ou la meilleure solution. Elle est tout de même une solution qui nous permettrait d'atteindre nos objectifs et de simplifier la réalisation du modèle. Toujours dans le contexte de la spécification de STCM, il reste à spécifier un modèle d'assemblage, ce qui est l'objectif de la section 9.3.

9.3 Un langage pour la composition spatio-temporelle

Dans notre stratégie de conception du modèle STCM (section 9.1), nous avons choisi AGWL [59] comme langage de référence pour spécifier un modèle d'assemblage. Nous proposons une solution d'adaptation des structures *XML* de ce langage pour qu'elles soient appliquées à des composant-tâches et qu'elles intègrent le support des connexions de ports spatiaux. Cette section présente tout d'abord notre approche d'adaptation. Ensuite, elle fournit un aperçu de l'utilisation et de l'intérêt de STCM à travers un exemple d'assemblage dans le langage proposé. Le terme composant utilisé par la suite désigne un composant-tâche.

9.3.1 Définition d'un composant

```
<!-- activité AGWL -->
<activity name="name" type="type">
  <dataIn name="name"/>*
  <dataOut name="name"/>*
</activity>

<!-- sub-workflow AGWL -->
<subworkflow name="name">
  <dataIn name="name"/>*
  <body> <activity>+ </body>
  <dataOut name="name"/>*
</subworkflow>
```

```
1  <!-- AGWL modifié: définition d'un composant -->
2  <component name="name" (extends="parentType")?>
3   <clientPort name="name" type="interfaceName"/>*
4   <serverPort name="name" type="interfaceName"/>*
5   <dataIn     name="name" type="dataType"/>*
6   <dataOut    name="name" type="dataType"/>*
7   <attribute  name="name" type="attributeType"/>*
8   <!-- autres types de port -->
9   ( <impl type="exe|dll|.." signature="sign" />
10  | <body> ... </body> )
11  <controllerDesc desc="desc"/>?
12 </component>
```

FIG. 9.10 – D'une activité/tâche et d'un sub-workflow vers un composant STCM.

Pour qu'un composant soit l'unité de composition dans AGWL, l'idée est de remplacer la spécification d'une tâche par celle d'un composant. Nous analysons tout d'abord les propriétés d'un composant que nous souhaitons préserver dans le langage d'assemblage. Nous présentons ensuite les changements effectués dans AGWL pour décrire un composant STCM.

Un composant est décrit par l'ensemble de ses ports. Pour sa structure interne (un binaire pour un composant primitif ou une composition pour un composite), nous la considérons comme une propriété de configuration d'un composant. Un composant peut être configuré pour être primitif ou composite et/ou avec des implémentations différentes. En conséquence, qu'un composant soit primitif ou non, lorsqu'il est défini, il est vu comme un concept unique. Par ailleurs, un type d'activité dans AGWL est clairement différencié selon qu'une activité soit atomique (primitive) ou un *sub-workflow* (composite). La partie gauche de la figure 9.10 illustre ces deux concepts. Cependant, selon notre point de vue, ce principe de distinction n'est pas indispensable, car une composition au sein d'un *sub-workflow* peut tout à fait être vue comme une implémentation particulière. Nous avons alors décidé de préserver une seule unité de composition configurable et donc, de remplacer les deux définitions rattachées à une activité avec l'unique définition d'un composant.

La figure 9.10 illustre comment un composant peut être défini dans un langage AGWL modifié. Les structures *XML* spécifiées sont inspirées du langage AGWL et du langage d'assemblage de GCM [87]. Elles permettent à un composant d'étendre la définition d'un composant préalablement défini (ligne 2) et de définir les différents types de ports : spatiaux, temporels et attributs (lignes de 3 à 8). La structure d'un composant peut être configurée à l'aide des éléments *XML* **`impl`** (ligne 9) ou **`body`** (ligne 10) pour décrire une structure interne d'un composant primitif ou respectivement d'un composite, et de l'élément **`controllerDesc`** (ligne 11) pour décrire sa membrane. Plus de détails sur l'élément **`body`** sont fournis dans la suite de cette section.

9.3.2 Description d'une connexion directe entre ports

La deuxième étape de la conception du modèle d'assemblage consiste à décrire comment les connexions de ports peuvent être spécifiées. La figure 9.11 propose une telle spécification. Une connexion peut être établie entre deux instances de sous composants d'un composite où l'élément **`instance`** définit une instance d'un élément **`component`**. Elle peut aussi être établie entre une instance d'un sous composant et le composant parent, suivant le

```
1  <component name="name" (extends="...")?>
2   <dataIn name="name" (type="...")?  set="outputRefORvalue"/> *
3   <clientPort name="name" (type="...")?  set="serverRef" />*
4   ...
5   <attribute name="name" (type="...")?  set="value" />*
6   <body>
7    <component>*
8    <instance name="name" componentRef="CompName" > ... </instance> *
9    <setPort client="name" server="name" /> *
10   <setPort in="name" out="name" /> *
11    ...
12  </body>
13 </component>
```

FIG. 9.11 – Eléments pour la connexion de ports spatiaux.

même principe que GCM. Ensuite, une connexion est vue comme une configuration de la valeur d'un port de type client classique ou d'un port d'entrée. Bien entendu, cette valeur n'est autre que la référence vers un port serveur ou respectivement un port de sortie. Deux solutions de configuration sont offertes. Un port peut être connecté lorsqu'il est défini (voir lignes 2 et 3) à l'aide de l'élément ***set***. Il peut aussi l'être ultérieurement dans le corps d'un composant composite (***body***, lignes 6 à 12) grâce à l'élément ***setPort*** (lignes 9 et 10). L'élément ***setPort*** est une instruction utilisée pour la connexion de ports spatiaux (lignes 9) et de ports temporels (ligne 10).

Le spécification que nous venons de décrire suit une logique similaire que celle de la connexion de ports dans AGWL. Les structures de connexions de ce dernier ont seulement été syntaxiquement adaptées pour la considération de tout type de ports.

9.3.3 Description d'une composition spatio-temporelle structurée

Nous venons de modifier AGWL pour définir un composant et connecter ses ports spatiaux et temporels. Dans cette section, nous nous intéressons à la structuration d'un assemblage de composants. L'objectif est de spécifier un langage d'assemblage « *à la* AGWL » qui garde la logique d'AGWL en ce qui concerne la construction d'un flot de contrôle et d'un flot de données pour la dimension temporelle et qui intègre la dimension spatiale à différents niveaux d'un assemblage.

Considérons tout d'abord les modifications apportées à la composition d'un flot de contrôle. Le langage AGWL modifié hérite des constructions de contrôle offertes par AGWL puisqu'il est question de préserver le niveau d'expressivité du comportement temporel d'une application. Ces structures sont : ***sequence***, ***if***, ***switch***, ***while***, ***for***, ***forEach***, ***dag***, ***parallel***, ***parallelFor*** et ***parallelForEach***. Cependant, ces constructions, au lieu d'être des activités particulières, sont considérées dans STCM comme des instructions du langage d'assemblage. Une instruction apparaît sous la forme d'une structure interne pré-définie d'un composant qui préserve le modèle de flot de données AGWL rattaché à cette structure pour des tests de validité d'assemblage. Néanmoins, dans STCM, une telle structure peut définir des ports spatiaux.

Sachant que l'adaptation d'une construction AGWL pour STCM suit le même principe pour l'ensemble des constructions de contrôle, nous nous focalisons dans cette section sur la spécification de deux constructions : la séquence et le branchement conditionnel ***if***. Sans

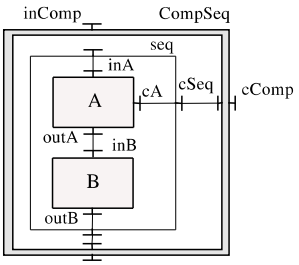


FIG. 9.12 – Exemple graphique d'une composition avec séquence.

```
1  <component name="CompSeq">
2   <dataIn name="inComp" type=../>
3   <clientPort name="cComp" type=../> ..
4   <body>
5    <sequence name="seq">
6     <dataIn name="inSeq" type=.. set="this.inComp"/>
7     <clientPort name="cSeq" type=.. set="this.cComp"/>
8     <declare>                                                  <-----
9      <component name="A" ..>                                        |
10       ... <!-- ports "inB", "outB" et "clientA" -->                |
11     </component>                                                   |
12     <component name="B" ..>                                        |
13       ... <!-- ports "inB" et "outB" -->                           |
14     </component>                                                   |
15                                                                déclarations
16     <instance name="a" componentRef="A".. </instance>              |
17     <instance name="b" componentRef="B".. </instance>              |
18                                                                    |
19     <setPort client="a.clientA" server="this.cSeq">                |
20     <setPort in="a.inA" out="this.inSeq">                          |
21     <setPort in="b.inB" out="a.outA">                              |
22     ...                                                            |
23    </declare>                                                  <-----
24
25    <exectask nameInstance="A">                                 <-----
26    <exectask nameInstance="B">                                 instructions
27    ...                                                         <-----
28   </sequence>
29  </body>
30 </component>
```

FIG. 9.13 – Exemple de composition dans STCM illustrant la spécification d'une séquence. La représentation graphique est illustrée à la figure 9.12.

perte de généralité, nous présentons la spécification respective pour les deux constructions à travers deux exemples, illustrés à la figure 9.13 pour la séquence et la figure 9.14 pour le branchement conditionnel ***if***. Dans une construction de contrôle, il est possible d'avoir des définitions de ports, des déclarations et des instructions. Premièrement, une construction de contrôle peut définir des ports spatiaux et temporels de la même façon qu'un composant (voir lignes 6 et 7 pour la figure 9.13 et ligne 5 pour la figure 9.14). Deuxièmement, il est possible de définir des composants, les instances de composants utilisées à l'intérieur d'une construction de contrôle et des connexions de ces instances (voir partie déclaration dans la figure 9.13). Il s'agit d'être capable de délimiter la portée des définitions de composants et de connexions par la construction de contrôle qui les utilise. Nous allons voir dans la sec-

```
1  <component name="CompIf">
2   ...
3   <body>
4    <if name="ifctrl">
5     ... <!-- ports spatiaux et temporels -->
6     <declare>
7      ...
8      <instance name="compyes" componentRef=.. />
9      <instance name="compno"  componentRef=.. />
10    </declare>
11    <condition> <!-- expression logique --> </condition>
12    <then> <exectask nameInstance="compyes"/>  </then>
13    <else> <exectask nameInstance="compno"/>  </else>
14   </if>
15  </body>
16 </component>
```

FIG. 9.14 – Exemple de composition dans STCM illustrant l'utilisation du ***if***.

tion suivante qu'il est possible aussi d'avoir des déclarations à l'intérieur et l'extérieur des corps d'une construction de contrôle, pour le ***if*** par exemple, à l'intérieur et l'extérieur des corps ***then*** et ***else***. L'emplacement des déclarations a un impact important sur la détermination du comportement exprimé par un assemblage spatio-temporel et sur le cycle de vie des composants d'une application, comme cela est discuté à la section 9.3.5. Troisièmement, la description des tâches à exécuter se fait à l'aide de l'instruction ***executetask*** sur une instance de composant préalablement déclarée. Pour l'exemple de la séquence, l'ordre de l'apparition des instructions ***executetask*** détermine la séquence. Enfin, pour les constructions de contrôle conditionnel comme le branchement ***if***, la description d'une condition de branchement se rajoute par rapport à la séquence (voir ligne 11 de la figure 9.14). Une condition est décrite sous la forme d'une expression logique *XSLT* (*Extensible Stylesheet Language Transformation*) sur des données et ports temporels dont les types doivent pouvoir être interprétés par le langage XML, comme cela est effectué dans d'AGWL.

En ce qui concerne un flot de données, il est construit à partir de l'ensemble des connexions de ports temporels spécifiées dans la composition d'une application. Le principe est le même que dans AGWL.

Nous ne fournissons pas de spécification pour l'implémentation des constructions de contrôle ou la gestion d'un flot de données, laissant la liberté à une plate-forme STCM de choisir une implémentation d'un modèle d'exécution d'une application. Toutefois, il serait intéressant d'avoir un modèle d'exécution à base de composants, ce qui favoriserait la réutilisation et simplifierait les mises à jour ou extensions d'une implémentation.

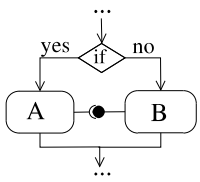
9.3.4 Une grammaire pour le langage d'assemblage STCM

Cette section résume la structure d'une composition STCM à travers une proposition de grammaire pour le langage d'assemblage. La figure 9.15 illustre l'essentiel de cette grammaire. Elle définit la syntaxe pour spécifier des composants, décrire la structure interne d'un composant primitif ou composite, décrire une instance de composant, configurer les ports (connecter/déconnecter) et utiliser les instructions du langage. Cette syntaxe s'inspire du langage d'assemblage de GCM pour la définition des composants, des ports et leur connexion. Elle s'inspire du langage AGWL pour les structures de contrôle. Nous retrou-

```
component    ::= <component name=string (extends= string)?>
                  port* attribute* content? membrane?
                 </component>
port         ::= clientport | serverport | inport | outport
clientport   ::= <clientPort name= string type=string (set= string)?/>
serverport   ::= <serverPort name= string type=string/>
inport       ::= <dataIn   name=string type=string (set= string)?/>
outport      ::= <dataOut   name=string type=string/>
attribute    ::= <attribute name= string type=string (set= string)?/>

membrane     ::= <controllerDesc desc=  string/>

content      ::= primitive | composite
primitive    ::= <impl type= string signature= string/>
composite    ::= <body> stcmassembly </body>
stcmassembly ::= declaration? instruction?

declaration  ::= <declare> component* instance* configport* </declare>
instance     ::= <instance name=string componentRef=string>
                   content? membrane?
                 </instance>
configport   ::= clientserver | inout
clientserver ::= <setPort client=   string server= string/>
                | <unsetPort client=   string (server= string)?/>
inout        ::= <setPort in=  string out=string/>
                | <unsetPort in=  string (out= string)?/>

instruction  ::= instance | executetask | configport | seq | if
                | switch | while | for | forEach | dag | parallel
                | parallelFor |  parallelForEach
executetask  ::= <exectask nameInstance= string/>

seq          ::= <sequence name=string> port* attribute*
                                        declaration instruction+ </sequence>
if           ::= <if name=string> port* attribute* declaration condition then else? </if>
condition    ::= <condition>  expr </condition>
then         ::= <then> stcmassembly </then>
else         ::= <else> stcmassembly </else>

parallel     ::= <parallel name= string> port* attribute* declaration section+ </parallel>
section      ::= <section>  stcmassembly </section>

switch       ::= <switch name=string> port* attribute* declaration case+ default? </switch>
case         ::= <case condition=  string (break= boolean)?>  stcmassembly </case>
default      ::= <default>  stcmassembly </default>
boolean      ::= "true" | "false"

// Même principe pour while, for, forEach, dag, parallelFor et parallelForEach.
...
// expr : expression logique AGWL.
// string : chaîne de caractères.
```

FIG. 9.15 – Aperçu de la grammaire du langage d'assemblage STCM. En gras : mots clés de la grammaire. En italique : chaînes de caractères du langage d'assemblage (les guillemets sont retirés pour simplifier).

vons dans cette syntaxe deux propriétés importantes du langage d'assemblage proposé. Premièrement, le langage est hiérarchique et suppose qu'un assemblage est un composant afin de favoriser la réutilisation (***component***). Deuxièmement, il offre la possibilité de déclarer des composants et instances de composants et de connecter des ports dans le temps ou dans l'espace à différents niveaux d'un assemblage : à l'intérieur ou l'extérieur des constructions




```
<if name="name">
 ...
 <then>
  <setPort client="A.pA"
           server="B.pB"
  />
  <exectask nameInstance="A"/>
 </then>
 <else>
  <exectask nameInstance="B"/>
 </else>
</if>
```

```
<if name="name">
 ...
 <declare>
 ...
  <instance name="B" ...>...</>
  <instance name="A" ...>
  <setPort client="A.pA"
           server="B.pB"/>
 </declare>
 <then>
  <exectask nameInstance="B"/>
 </then>
 <else>
  <exectask nameInstance="A"/>
 </else>
</if>
```

FIG. 9.16 – Compositions illégale (gauche) et légale (droite), en fonction de la dimension temporelle ou spatiale qui domine.

de contrôle (voir ***declaration*** dans la grammaire). Cette possibilité est importante pour déterminer la portée des définitions de composants et des connexions entre composants durant l'exécution d'une application et pour déterminer le comportement d'une application (voir section 9.3.5).

Il est à noter que des améliorations sont à apporter à cette grammaire, en particulier par rapport aux limitations du langage AGWL. En effet, dans la spécification courante de STCM, nous nous sommes contentés de reprendre la logique AGWL. Cette logique ne traite pas le cas multi-tâche et elle limite les expressions logiques (***expr***) pour décrire des conditions dans les constructions de contrôle conditionnelles comme le***if*** à des expressions simples et sur des données (ports temporels ou valeurs) de types triviaux (limitation abordée dans la section 3.3.2.1 du chapitre 3). Il serait intéressant d'étudier par exemple une solution qui permettrait de représenter une expression par un composant capable d'effectuer des tests sur des données complexes et de fournir en sortie le résultat du test de la condition.

9.3.5 Impact de la composition structurée sur le cycle de vie

Dans la section 8.4.2.2, nous avons noté qu'il n'est pas suffisant d'expliciter le contrôle dans un modèle de composition lorsqu'un composant ou plusieurs sont assemblés dans le temps et l'espace, pour être capable de déterminer le comportement d'une application à travers son assemblage. Nous avons aussi souligné l'importance de cette détermination pour gérer automatiquement et efficacement le cycle de vie de l'ensemble des composants de l'application. Pour exemple, nous avons étudié la composition illustrée à gauche de la figure 9.16 qui peut refléter plus d'un comportement. Nous avons vu qu'il est nécessaire de fournir un mécanisme à l'utilisateur pour permettre d'expliciter simplement le comportement attendu, ceci de manière déterministe. L'objectif de cette section est de fournir un aperçu de la solution que nous proposons dans le cadre de STCM pour spécifier le comportement exprimé par un assemblage.

En réalité, le mécanisme proposé a été implicitement introduit dans la spécification des structures de contrôle STCM. Il s'agit d'exploiter l'emplacement de la définition des composants et de leurs compositions par rapport à une telle structure pour spécifier un système de priorité entre dimensions temporelle et spatiale. Ce système est utilisé pour déterminer un

comportement. En reprenant l'exemple de la construction de contrôle **`if`** (voir les figures 9.14 et 9.15), nous remarquons qu'il est possible de définir et/ou de connecter des composants à différents niveaux : à l'intérieur ou à l'extérieur d'un corps (**`then`** et **`else`**) de la structure. En fonction de l'emplacement de la définition d'une instance de composant, une priorité est accordée à l'une des deux dimensions. Voici les principales règles rattachées à ce principe de priorité :

1. Si un composant est défini et composé dans l'espace directement dans le corps d'une structure de contrôle, alors la dimension temporelle est prioritaire. Dans ce cas, le composant peut être instancié autant de fois qu'il serait atteint par l'exécution du flot de contrôle. Des cas d'utilisation seraient par exemple des exécutions en l'absence de composants avec état, ou bien lorsque des connexions spatiales sont optionnelles.
2. Si une connexion spatiale entre deux composants est spécifiée en dehors des corps de structure de contrôle dans lesquels ces composants apparaissent, alors la dimension spatiale est prioritaire. Dans ce cas, la connexion persiste entre deux uniques instances des deux composants, cela pendant la durée de validité de la connexion. Cette durée est déterminée par la portée de la connexion, délimitée par le corps du composant dans lequel elle est décrite. Ce cas est particulièrement intéressant dans le cadre où un composant aurait un état interne (donnée globale du composant non accessible depuis l'extérieur) qui est à préserver par exemple durant l'exécution d'une boucle.
3. Si dans le cadre d'une priorité temporelle, une connexion spatiale est établie entre deux composants et si seul un composant est atteignable à la fois par le flot de contrôle, alors la composition est considérée illégale. Une tel cas est décrit dans la figure 9.16.

Ainsi, non seulement le système de priorité permet de différencier des comportements pouvant être exprimé, mais il est aussi considéré comme moyen de détecter des incohérences dans un assemblage et de décider de la validité de ce dernier. En même temps, ce système aide à déterminer le cycle de vie des composants et doit être considéré dans l'automatisation de sa gestion.

Bien qu'il soit possible de décrire le système de priorité à travers des phrases, ceci représente une limitation pour STCM. Il serait plus intéressant, même indispensable, d'avoir un formalisme bien défini, c'est à dire, une sémantique opérationnelle qui serait capable de refléter la diversité des comportements qu'un assemblage puisse exprimé. Pour un utilisateur, elle apporterait une meilleure maîtrise de la façon dont il peut décrire différents comportement. Pour une plate-forme implémentant STCM, elle faciliterait l'automatisation de la gestion du cycle de vie des composants au sein d'une application. Une première voie de recherche à laquelle nous pensons est l'adoption d'un calcul formel. Sans s'attarder sur le sujet, plusieurs calculs formels sont proposés comme le *pi-calcul* [84] ou *CCS* pour *Calculus of Communicating Systems* de Robin Milner [83]. Ces exemples définissent à l'origine des sémantiques opérationnelles pour des systèmes interconnectés en sens large. Pour commencer, il serait intéressant d'étudier les travaux effectués autour de l'adaptation de ces calculs par des modèles de flux de travail et modèles de composants logiciels existants.

9.3.6 Conclusion

Nous venons de présenter notre proposition de spécification d'un modèle d'assemblage pour STCM. Nous en avons présenté les principes d'adaptation du langage AGWL afin d'y remplacer le concept de tâche/activité par celui d'un composant, d'y introduire des ports

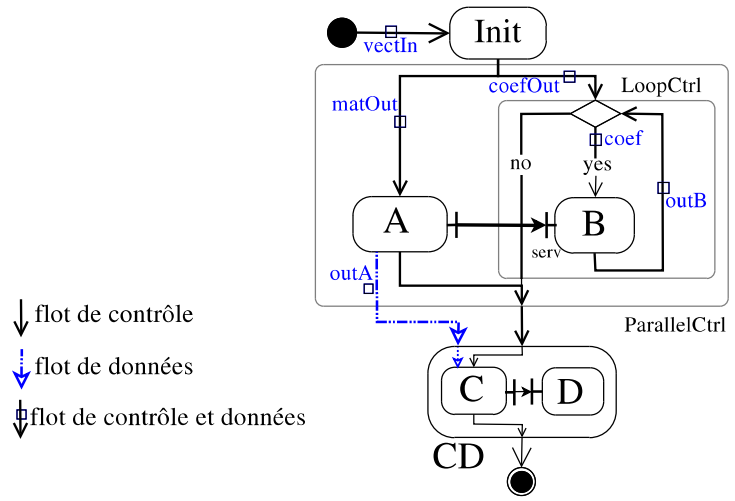


FIG. 9.17 – Exemple d'application.

spatiaux et d'étendre des structures de contrôle pour le support de connexions spatiales. Ainsi, lors de la composition d'une application, il est possible d'assembler des composants dans l'espace, le temps ou l'espace et le temps au même niveau d'assemblage.

Pour ce qui est des objectifs d'expressivité, le langage d'assemblage STCM hérite tout d'abord de ce qui peut être exprimé du comportement temporel dans AGWL. Le support en plus de la composition spatiale fait que les dépendances spatiales sont désormais intégrée dans la logique algorithmique exprimée par un assemblage. Cependant, nous avons souligné que cela pouvait présenter des cas problématiques qui altéreraient la capacité de déterminer l'unique comportement attendu. Toutefois, nous y avons remédié par la mise en place d'un système simple de priorité entre composition spatiale et temporelle. Il est ainsi à exploiter par l'utilisateur pour structurer un assemblage et être capable de distinguer simplement entre différents comportements exprimable par l'assemblage. Ce système est actuellement décrit en langue française. Nous avons mis l'accent sur la perspective de définir une sémantique opérationnelle dans la spécification de STCM. L'intérêt serait non seulement de donner un sens formel à cette spécification mais aussi de valider notre approche.

9.4 Exemple d'utilisation

Cette section présente un exemple d'utilisation de STCM. Elle part de la spécification d'un comportement attendu d'une application pour l'exprimer de manière explicite à travers son assemblage. L'objectif est d'illustrer les avantages offerts par STCM par rapport aux modèles de composants et modèles de flux de travail existants.

La figure 9.17 représente une structure simplifiée d'une application synthétique que cette section a pour objectif de composer dans STCM. Cette application s'inspire du comportement de deux applications réelles destinées à être exécutées sur grilles de calcul. La première est une application de traitement d'images sur l'analyse de mouvements et trajectographies [54][2]. La deuxième est l'application numérique de simulation couplée océan-

[2]Cette application est développée par l'équipe-projet VISTA de l'INRIA avec qui nous travaillons pour la structurer sous forme d'un assemblage de composants.

atmosphère, introduite dans la section 3.2.1 du chapitre 3.

Au niveau le plus élevé de la hiérarchie, l'application contient un couplage de codes, représenté par la connexion des composants **A** et **B**. Le composant **A** effectue des calculs sur une matrice, préalablement initialisée par le composant **init**, en fonction de certaines conditions initiales de l'application. Le résultat du calcul de **A** dépend de certains paramètres fournis par **B**. Ces paramètres varient dans le temps, en fonction des calculs itératifs effectués grâce à la boucle représentée. Il est supposé que **B** ait un état persistant dont dépend entièrement le calcul effectué dans **A**. Une deuxième étape de calcul, cette fois-ci en fonction de la sortie de **A**, est effectuée par le composant **CD**. Pendant cette étape, le sous composant **C** envoie des résultats intermédiaires au composant **D** pour visualisation. Le couplage de codes à l'intérieur de **CD** est volontairement rendu visible pour illustrer différents niveaux de composition et rappeler qu'une composition peut être hiérarchique et que la structure interne d'un composant peut être complexe. Pour simplifier, les membranes et la structure interne des composants qui sont des composants STCM ne sont pas représentées. Quant aux ports temporels, ils sont implicitement présents à travers le flot de données figuré. Par la suite, seuls quelques ports sont spécifiés.

La figure 9.18 illustre l'équivalent de la composition graphique de l'application dans le langage d'assemblage proposé pour STCM. Cette composition correspond aux comportement qui vient d'être décrit. Premièrement, la concurrence d'exécution des tâches des composants **A** et **B** est assurée grâce à la structure de contrôle **parallel** (lignes 18 à 50), où chaque composant est inclus dans une section parallèle (lignes 37 à 49). Cette concurrence est aussi imposée par la connexion spatiale décrite entre **A** et **B** (ligne 33). Conformément au système de priorité défini dans la section 9.3.5, la définition préalable de cette connexion impose une priorité spatiale. Cela répond à la préservation de l'état de **B**. La création d'une unique instance **B** est alors imposée pour l'exécution de la boucle (lignes 41 à 48). De plus cette instance doit rester active tant que l'exécution des deux sections parallèles n'est pas terminée (sortie de **A** et sortie de la boucle). Deuxièmement, le couplage de code au sein du composant **CD** ne requiert qu'une composition spatiale qui impose l'activation du composant **D** durant l'activation de **C**. Dans ce cas, une priorité temporelle suffit. Troisièmement et enfin, le séquencement globale des étapes de calcul est assuré grâce à la structure de contrôle **sequence** (lignes 4 à 54).

Nous avons construit ainsi un assemblage qui rend visible le comportement attendu de l'application. Le cycle de vie des composants de l'assemblage est aussi déterminé, ce qui devrait être suffisant pour automatiser la gestion de la structure de l'application et de décider d'un plan de déploiement efficace.

Si nous considérons la programmation de cet exemple d'application dans GCM ou AGWL, l'expressivité d'un assemblage serait réduite et la programmation serait complexifiée. Pour une programmation avec GCM, les étapes de calcul et les structures de contrôle seront masquées dans l'implémentation des composants. Soit un concepteur se contente de décrire un assemblage avec l'ADL de GCM et de déployer tous les composants avant l'exécution de l'application, ce qui conduirait à occuper des ressources inutilement, soit il décide de gérer le cycle de vie des composants au risque de complexifier la programmation. Par ailleurs, une programmation avec AGWL inciterait le concepteur à masquer une partie de l'assemblage dans l'implémentation de tâches primitives. Par exemple, la priorité spatiale du couplage de codes **A–B** sur la boucle **while**, incite le concepteur à implémenter une tâche primitive qui masque l'assemblage des composants **A** et **B** avec leur structure interne et le contrôle qui les entoure, en l'occurrence la boucle et la structure parallèle. La hiérarchie d'un

```
1  <component name ="example">
2   <dataIn name="vectIn" type="Vect"/>
3   <body>
4    <sequence name="Seq" >
5     <dataIn name="vectIn" type="Vect" set="example.vectIn"/>
6     <declare>
7      <component name="Init">
8       <dataIn name="vectIn" type="Vect"/>
9       <!-- out: Matrix: matOut; Double: coefOut -->
10      ...
11     </component>
12     <component name="CD"> <!-- voir figure 9.19      -->  </component>
13     <instance name="init" componentRef="Init">..</instance>
14     <setPort in="init.vectIn" out="Seq.vectIn"/>
15    </declare>
16
17    <exectask nameInstance="Init"/>
18    <parallel name="ParallelCtrl">
19     <declare>
20      <component name="A">
21       <!-- in: Matrix: inA, out: Matrix: outA -->
22       <clientPort name="pA" type="GetRes"/>
23       ...
24      </component>
25      <component name="B">
26       <!-- in: Double: inB, out: Double: outB -->
27       <serverPort name="pB" type="GetRes"/>
28       ...
29      </component>
30      <instance name="a" componentRef="A">..</instance>
31      <setPort in="a.inA" out="this.matOut"/>
32      <instance name="b" componentRef="B">..</instance>
33      <setPort client="a.pA" server="b.pB"/>                  //priorité spatiale
34      ...
35     </declare>
36
37     <section>
38      <exectask nameInstance="a"/>
39     </section>
40     <section>
41      <while name="LoopCtrl" >
42       <dataIn name="coef" type="Double" set="init.coefOut" loopSet="B.outB"/>
43       <condition> coef < 1000 </condition>
44       <loopBody>
45        <declare> <setPort in="b.inB" out="this.coef"/> </declare>
46        <exectask nameInstance="b"/>
47       </loopBody>
48      </while>
49     </section>
50    </parallel>
51    <instance name="cd" componentRef="CD">..</instance>
52    <setPort in="cd.inCD" out="a.outA"/>
53    <exectask nameInstance="cd"/>
54   </sequence>
55  </body>
56 </component>
```

FIG. 9.18 – Exemple d'assemblage d'une application dans STCM.

```
1  <component name="CD">
2   <dataIn name="inCD" type="Matrix"/>
3   <body>
4    <sequence >
5     <declare>
6      <component name="D">
7       <serverPort name="pD" type="Print"/>
8       ...
9      </component>
10     <component Name="C">
11      <dataIn name="inC" type="Matrix" set="cd.inCC"/>
12      <clientPort name="pCC" type="Print" set="D.pD"/>
13        ...
14     </component>
15    <declare>
16
17    <instance name="c" componentRef="C">..</instance>
18    <instance name="d" componentRef="D">..</instance>
19    <setPort client="c.pC" server="d.pD"/>
20    <exectask nameInstance="C"/>
21   </sequence>
22  </body>
23 </component>
```

FIG. 9.19 – Définition du composant ***CD*** référencée à la ligne 12 de la figure 9.18. En gras : les instructions de l'assemblage.

assemblage est ainsi limitée et le concepteur perd l'avantage de pouvoir changer ce contrôle juste en réorganisant la structure de son application sans avoir à modifier et recompiler des codes. L'ensemble a pour conséquence de réduire la réutilisation et complexifier aussi la programmation. Nous revenons ainsi sur des limitations de l'existant, discutées dans les sections 3.4 et 8.1.3 des chapitres 3 et 8.

En conclusion, STCM a l'avantage de réduire la complexité de conception, de promouvoir la réutilisation et d'améliorer l'expressivité d'un assemblage qui est importante pour encourager l'automatisation de la gestion dynamique de la structure d'une application et l'utilisation efficace des ressources d'exécution.

9.5 Discussion et conclusion

Ce chapitre a présenté notre proposition d'un modèle de composants spatio-temporel, que nous avons nommé STCM. La conception de ce modèle repose sur le principe de réutilisation et d'extension de l'existant. Le modèle se retrouve être une solution croisée entre un modèle de composants logiciel et un modèle de flux de travail. La solution proposée a permis d'atteindre les objectifs que nous nous sommes fixés dans le chapitre 4 et le chapitre 8.

En première partie, le modèle de composants GCM a été le modèle de référence pour spécifier un composant-tâche, l'unité de composition de base de STCM. Cette unité apparaît sous la forme d'un composant GCM étendu qui introduit une spécification pour les ports temporels, le concept de tâche et le cycle de vie. Mis à part l'ajout d'une nouvelle sémantique rattachée aux ports GCM, la spécification étendue s'appuie sur des concepts préalablement supportés : ensemble d'interfaces client/serveur et de contrôleurs.

Dans un deuxième temps, les concepts essentiels de la spécification du modèle d'assemblage STCM ont été décrits, cette fois-ci en considérant le langage de flux de travail

ASKALON -AGWL comme langage de référence. Après raffinement des concepts offerts par AGWL, nous y avons tout simplement remplacé le concept de tâche/activité par un composant GCM étendu et introduit le concept de composition spatiale. Ce travail a abouti sur un langage offrant une approche de programmation pour composer une application. Désormais, une composition est capable d'exprimer une logique algorithmique incluant un comportement temporel et des dépendances spatiales entre composants. De cette manière, le rapprochement entre un modèle de composants et un modèle de flux de travail a été possible.

Même si STCM fournit des réponses aux principales limitations de l'existant en réunissant la dimension spatiale et temporelle au niveau de l'assemblage, sa spécification demeure un travail en cours de recherche. Il est important de considérer en perspective la difficulté d'utilisation du langage X ML pour l'assemblage, le manque d'une sémantique opérationnelle et l'absence d'une implémentation des concepts présentés. Les perspectives à cet égard sont présentées dans le chapitre 10.

Cinquième partie

Pour conclure

Chapitre 10

Conclusions et perspectives

Conclusion générale

Contexte d'étude

Le recours au calcul scientifique ne cesse de se répandre dans le domaine scientifique. Il est devenu ainsi un champ important de l'informatique. À la recherche continue de réponses plus précises pour mieux comprendre les phénomènes complexes qui nous entourent, les applications scientifiques sont de plus en plus complexes. Cette complexité se retrouve dans la taille, la structure et l'aspect multi-disciplinaire de ces applications. Pour y répondre, les chercheurs en informatique ont pour défi d'offrir des modèles de programmation qui facilitent la structuration des programmes et leur réutilisation.

Cependant, ce défi n'est pas unique. Les applications scientifiques exigent de plus en plus d'importants moyens de calculs et de stockage. De son coté, le matériel informatique évolue continuellement à la recherche de puissances accrues. Une variété d'architectures matérielles sont apparues en allant des simples processeurs aux grilles de calcul et en passant par les super-calculateurs, les grappes et les processeurs multi-cœur. La nature distribuée et/ou parallèle des architectures offrant de grandes puissances de calcul, comme les grilles, font du calcul distribué (incluant le calcul parallèle) une préoccupation importante des modèles de programmation actuels.

Devant la complexité des applications scientifiques, la diversité des modèles ou langages de programmation et l'évolution des infrastructures matérielles, il ne semble pas raisonnable d'imposer aux programmeurs l'utilisation d'un modèle, d'un langage ou d'une infrastructure particulière. En revanche, il est souhaitable de leur offrir la possibilité de programmer une application à partir d'un assemblage de codes, probablement réutilisés (patrimoniaux), pouvant être réalisés avec différents modèles de programmation et pouvant avoir différentes implémentations (séquentielles, parallèles, distribuées, suivant différents algorithmes, etc). La programmation se doit aussi d'être indépendante du type d'infrastructures matérielles pour avoir une maîtrise sur leur évolution. Le choix de l'implémentation des programmes et les ressources ne devrait se faire qu'au moment de l'exécution, moment durant lequel une application peut être adaptée aux ressources.

Les modèles de programmation actuels, bien qu'ils résolvent de multiples problèmes, ne

parviennent pas encore à concrétiser complètement cette vision « *idéale* ». Dans le cadre de nos travaux de recherche, nous avons analysé l'existant des modèles de programmation par composition. Une étude comparative des modèles de composants logiciels et des modèles de flux de travail nous a permis de mieux illustrer les limitations de l'existant. Les modèles de composants logiciels restent de bas niveau. Ils n'offrent pas suffisamment de concepts pour permettent aux programmeurs, tel que les scientifiques, d'exprimer simplement le comportement d'une application, de se focaliser sur les aspects métiers ou de découpler la programmation des ressources d'exécution, rendues complexes à considérer. Quant aux modèles de flux de travail, ils adressent plus la simplicité de programmer pour une gestion automatique de l'ordonnancement des tâches que la réutilisation. Ensuite, ils ne sont pas forcément adaptés à plusieurs types d'applications, telles que les applications de couplage de codes ou des programmes parallèles impliquant souvent des communications entre codes parallélisés. Finalement, aucune approche n'apporte de réponses entièrement satisfaisantes.

Contributions

Nous pensons que les modèles de composants logiciels ne sont pas loin d'offrir la vision souhaitée. Les standards proposés pour définir un composant n'ont pas besoin d'être remis en question. D'après ce que nous avons constaté de l'existant, il s'agit essentiellement d'y apporter des extensions et d'abstraire la programmation, ceci afin de :

– Réduire la complexité de programmation.
– Accroître la réutilisation.
– Découpler la programmation d'une infrastructure matérielle.
– Être capable d'adapter efficacement une même application pour une exécution sur différentes infrastructures matérielles.

Les travaux de recherche présentés dans ce manuscrit contribuent à ces objectifs pour les approches de programmation à base de composants.

Nos contributions visent plus particulièrement à rendre explicite un paradigme de communication entre composants, un paradigme de programmation parallèle et un comportement temporel d'un assemblage de composants. Il a été question d'aborder différents niveaux de la conception d'une application et de traiter des cas récurrents dans les applications scientifiques et non supportés simplement dans les modèles de composants.

Modèle abstrait pour le partage de donnée : ce modèle a pour objectif d'introduire le modèle de communication à mémoire partagée dans les modèles de composants. Pour cela, nous avons proposé la notion de *port de donnée*. Il s'agit d'une catégorie additionnelle de ports à travers laquelle un composant exprime le partage d'une donnée avec d'autres composants. Un port de donnée a la capacité de séparer la partie fonctionnelle de la partie non fonctionnelle liée au partage. D'une part, il offre à l'implémentation d'un composant l'illusion de manipuler localement une donnée partagée. D'autre part, il est relié de manière transparente à un système de partage de donnée. Il revient à ce dernier de gérer la localisation et la synchronisation des accès concurrents à une donnée partagée. Cette gestion tient compte de la cohérence, la persistance, le transfert et la volatilité de cette donnée.

Par ailleurs, un port de donnée peut être relié à différents systèmes de partage de données. Il permet de tirer profit des nombreux travaux de recherche sur les systèmes existants,

comme les systèmes d'exploitation pour un partage local, les systèmes *MVP* pour grappes ou les services de partage de données pour grilles. Étant donnée que le partage est rendu explicite, le choix d'un système approprié peut se faire à l'exécution, en fonction des ressources et du placement des composants ou autres critères comme la taille des données partagées.

La faisabilité de cette proposition a été démontrée pour le modèle de composants CORBA par les travaux de stage d'ingénieur de Landry Breuil. Ces travaux ont été effectués en collaboration avec Mathieu Jan[1] pour une utilisation de *JuxMem*, un service de partage de donnée sur grille proposé par ses travaux de doctorat. Une présentation de l'implémentation des ports de données dans un composant CCM se trouve dans [4].

Nous avons aussi étudié le partage de donnée pour le passage de paramètres à l'appel d'opérations entre composants. Cette étude s'appuie sur les ports de donnée pour un partage plus dynamique. Une extension de la spécification du modèle de composant CCA pour introduire cette proposition est présentée dans [1].

Dans cette première contribution, nous avons défendu la possibilité d'introduire simplement un modèle de communication complexe dans les modèles de composants en offrant en même temps un niveau d'abstraction élevé par rapport aux ressources d'exécution.

Modèle abstrait pour le paradigme MAÎTRE-TRAVAILLEUR : ce modèle a pour objectif d'offrir un moyen d'abstraire la programmation d'applications ou partie(s) d'applications qui suivent le paradigme MAÎTRE-TRAVAILLEUR. D'une part, il propose le concept de *collection*. Une collection représente un ensemble de composants qui permet au moment de la programmation de rendre transparent leur nombre et la politique d'ordonnancement des requêtes arrivant sur ces composants. Ainsi, assembler des composants suivant le paradigme MAÎTRE-TRAVAILLEUR peut se résumer en la description d'un composant maître utilisant des fonctionnalités fournies par une collection de travailleurs. Au moment de l'exécution, il revient à la plate-forme d'exécution des composants d'identifier l'utilisation du paradigme MAÎTRE-TRAVAILLEUR à partir de l'assemblage. Une fois une collection identifiée, la plate-forme a pour rôle de déterminer le nombre de travailleurs à instancier et d'introduire un mécanisme de transfert de requêtes entre le maître et ses travailleurs.

D'une autre part, le modèle propose la notion de *patron de transfert de requêtes*. Un patron permet de spécifier le prototype d'une implémentation associée à une politique de transfert de requêtes. La spécification de plusieurs patrons permet d'offrir la possibilité de choisir une politique de transfert qui soit la mieux appropriée à un contexte d'exécution d'une application, essentiellement pour des fins de performances. Ce choix doit tenir compte de l'architecture des ressources (homogène, hétérogène, hiérarchique pour les grilles, etc.), du nombre de travailleurs, de méta-informations comme le nombre et la taille des requêtes. Un patron étant découplé de l'implémentation d'une application, il est possible aussi pour des objectifs d'adaptation, de modifier la structure d'un patron ou le remplacer au cours de l'exécution.

Par ailleurs, le modèle proposé tient compte des nombreux travaux de recherche qui ont déjà développé plusieurs environnements MAÎTRE-TRAVAILLEUR, tel que les systèmes de type *NES* (*Network Enabled Server*) et les plate-formes de type *Desktop Grid* pour le calcul global (*Global Computing*), et qui proposent des politiques de transfert de requêtes avancées. Pour tirer profit de ces nombreux efforts, nous avons proposé qu'un patron puisse représenter l'architecture d'un tel environnement.

[1] Ancien doctorant de l'équipe-projet PARIS

Après avoir étudié ce modèle, nous avons proposé une extension simple du modèle de composants CORBA pour y introduire le concept de collection et obtenir un langage d'assemblage abstrait. Nous avons aussi démontré la possibilité de décrire et utiliser un assemblage hétérogène qui intègre un patron associé à l'architecture de l'environnement DIET, dédié aux grilles de calcul.

Un ensemble d'expérimentations sur une application synthétique a permis d'une part d'illustrer la nécessité d'avoir plusieurs patrons de transfert de requêtes, la complexité de leur gestion et le besoin par la même occasion de masquer cette complexité aux programmeurs. D'une autre part, nous avons démontré qu'en faisant des choix de patrons adéquats, il est possible d'aboutir à des performances d'exécution satisfaisantes. Il en est de même pour les résultats obtenus en utilisant DIET.

Au final, même si nous avons adopté une approche manuelle dans les choix et l'intégration de patrons, nous avons pu confirmer l'essentiel de ce que nous avons défendu dans notre étude. Les résultats préliminaires nous mènent aussi à croire que notre approche d'abstraction est prometteuse, car elle ne va pas à l'encontre de la possibilité d'obtenir de bonnes performances.

Modèle abstrait pour le comportement temporel d'une application : ce modèle a pour objectif de définir un modèle de composition spatio-temporel (STCM). Ce dernier prend la forme d'un modèle croisé entre les modèles de composants logiciels et les modèles de flux de travail. Il définit la notion de *composant-tâche* et de *ports temporels*. Un composant-tâche est un composant qui introduit le concept de tâche, repris depuis les modèles de flux de travail. Il s'agit d'une fonctionnalité dont l'exécution est déclenchée par l'arrivée de données sur des ports temporels en *entrée*, définis par le composant-tâche. De manière symétrique, à la fin de son exécution, une tâche peut produire des données en sortie. Celles-ci sont envoyées à d'autres composant-tâches à travers des ports temporels en *sortie*. En plus des ports temporels, un composant-tâche peut définir des ports classiques, de type clients/serveurs (dits ports spatiaux). Un composant-tâche est ainsi capable d'exprimer des dépendances dans le temps et dans l'espace.

Après avoir analysé des alternatives d'assemblage de composant-tâches pour la dimension temporelle (suivant un flot de données ou un flux de travail), notre contribution a été de proposer un modèle STCM *à la flux de travail*. Dans ce modèle, un composant-tâche est un composant GCM étendu et le langage d'assemblage s'inspire du langage de flux de travail ASKALON -AGWL (*Abstract Grid Workflow Language* [59]). D'une part, nous avons étendu la spécification de GCM pour y introduire le concept de tâche et les ports temporels. L'extension spécifie la définition et l'utilisation d'une tâche et d'un port temporel. Elle spécifie aussi le contrôle de l'exécution d'une tâche, la disponibilité des données en entrée/sortie de celle-ci et le cycle de vie modifié d'un composant, qui tient compte du rajout de la notion de tâche. D'une autre part, nous avons remplacé le concept d'activité dans le langage ASKALON -AGWL par un composant GCM étendu et introduit le concept de composition spatiale dans différentes structures de contrôle. Le résultat obtenu est un modèle de composants hiérarchique avec un langage d'assemblage spatio-temporel.

Étant donné que le modèle proposé n'est qu'au stade de spécification, nous avons présenté un exemple d'assemblage spatio-temporel à travers lequel nous avons discuté les avantages attendus, ceci par rapport à l'utilisation d'un langage de description d'architecture (ADL) ou d'un langage de flux de travail. Ces avantages abordent la simplicité de program-

mation pour différents types d'applications, la réutilisation améliorée et la possibilité d'automatiser la gestion dynamique de la structure d'une application. Cette gestion doit aussi prendre en compte la disponibilité et le partage des ressources d'exécution.

Les modèles proposés ne sont pas inaccordables. Ils peuvent être systématiquement fusionnés au sein d'un même modèle spatio-temporel générique qui traite différents problèmes d'abstraction. De son coté, une application scientifique peut avoir une structure réunissant plus d'un paradigme étudié. Un exemple simple est l'application du lancer de rayon, qui suit le paradigme MAÎTRE-TRAVAILLEUR avec partage de donnée entre les travailleurs (voir section 4.3.2 du chapitre 4).

Perspectives

Les travaux effectués au cours de cette thèse ouvrent des perspectives à plus ou moins long terme.

Implémentations

Dans cette thèse nous nous sommes focalisés sur la spécification des modèles proposés et nous avons démontré la faisabilité d'une partie seulement des concepts définis dans ces modèles. Une perspective évidente est de poursuivre l'implémentation de ces concepts. Il reste en particulier à implémenter le modèle STCM, autrement dit, les concepts de composant-tâche et ports temporels ainsi qu'une plate-forme pour la construction et l'exécution d'un assemblage spatio-temporel. À court terme, au lieu de développer entièrement une plate-forme, une solution qui consisterait à traduire un assemblage STCM en un flux de travail classique afin d'utiliser un moteur existant, nous semble une alternative envisageable pour démontrer la faisabilité du modèle. Cette traduction serait possible dans la mesure où une composition dans STCM peut être traduite en un séquencement d'actions de gestion du cycle de vie de l'ensemble des composants d'une application. Autrement, une plate-forme pour STCM doit aussi intégrer l'implémentation du modèle de partage de données et le support du paradigme MAÎTRE-TRAVAILLEUR. À long terme, il est préférable d'implémenter une plate-forme qui supporterait l'ensemble des concepts proposés.

Évaluations étendues

Il serait intéressant de poursuivre nos travaux d'évaluation des modèles abstraits pour le partage de données et le support du paradigme MAÎTRE-TRAVAILLEUR. Un axe d'évaluation pour le partage de données serait de mesurer la nécessité de changer le système de gestion du partage pour une même application, en variant le type de ressources, la taille d'une donnée partagée et le nombre de composants partageant une donnée. Pour le MAÎTRE-TRAVAILLEUR, une suite logique est de commencer par des expérimentations à plus grande échelle (> 1024 travailleurs) et pour des cas d'applications complexes. Pour une même application, il serait possible de varier plusieurs paramètres tel que l'hétérogénéité des ressources, l'utilisation de patrons de transfert de requêtes avancés ou le nombre de travailleurs de manière dynamique en fonction par exemple de la disponibilité des ressources matérielles. Il serait aussi intéressant de traiter des cas d'applications multi-paradigmes avec plusieurs

collections. L'objectif serait de caractériser les performances de différents patrons et d'estimer les surcoûts d'une adaptation dynamique. Il serait aussi question d'estimer les conséquences de la gestion du paradigme MAÎTRE-TRAVAILLEUR, rendue plus complexe dans le modèle proposé, sur les performances d'exécution d'une application.

Applications

Il serait intéressant de valider notre étude par rapport à des applications réelles. Il serait question d'utiliser le niveau d'abstraction proposé pour leur programmation, démontrer la simplicité de programmer et évaluer les performances de calcul. Une première application serait le lancer de rayon qui implique le paradigme MAÎTRE-TRAVAILLEUR et le partage de donnée. Autrement, nous avons eu l'occasion de travailler en collaboration avec l'équipe-projet VISTA de l'INRIA sur une application de traitement d'images dont l'objectif est l'analyse de mouvements et trajectographies [54]. À l'origine, il s'agissait d'un code séquentiel qui nécessite une décomposition pour une exécution à l'échelle d'une grille. Dans un premier temps nous avons décomposé ce code sous forme de composants CORBA afin de paralléliser plusieurs étapes de calcul. Le degré de parallélisme peut varier d'une étape à l'autre et peut atteindre plus de 10000 processus parallèles. L'étape suivante consisterait à utiliser le modèle d'assemblage spatio-temporel pour la composition et l'exécution sur grille. D'autres cas d'applications se présentent aussi dans le cadre du projet ANR LEGO, tel que l'application numérique de simulation couplée océan-atmosphère, introduite dans le chapitre 3.

Développement d'outils pour l'instanciation de modèles et interaction avec un canevas d'adaptation

Dans notre étude, nous avons déterminé le rôle d'un système pour automatiser l'instanciation d'un assemblage abstrait au moment de l'exécution d'une application et la gestion dynamique de son architecture. À cet égard, deux perspectives à moyen et long terme se dessinent.

Tout d'abord, l'instanciation d'un assemblage a besoin de programmes incluant des fonctions de décision (choix d'un système de partage de donnée ou sélection d'un patron de transfert de requêtes pour le paradigme MAÎTRE-TRAVAILLEUR) et de fonctions pour la transformation d'assemblages. Nous pensons que ces programmes pourraient être une extension de l'outil de déploiement générique ***Adage***, puisqu'ils doivent tenir compte des ressources d'exécution.

Ensuite, pour automatiser la gestion de la structure dynamique d'une application, il nous semble intéressant d'interagir avec les travaux de thèse de Boris Daix [2] sur le déploiement dynamique. Ces travaux visent la prise en compte de structures dynamiques d'applications sur différentes formes, comme un les flux de travail ou le paradigme MAÎTRE-TRAVAILLEUR. Des interactions avec un canevas d'adaptation serait aussi à considérer. Nous pensons particulièrement au canevas d'adaptation ***Dynaco*** proposé par Jérémy Buisson[3]. Le rôle de ce canevas serait de gérer le besoin d'adapter la structure d'une application, comme le contenu d'une collection de travailleurs, aux variations dynamiques des conditions d'exécution. Pour une collection de travailleurs, il s'agirait par exemple de gérer la nécessité de rajouter des

[2] Doctorant au sein de l'équipe-projet PARIS
[3] Ancien doctorant au sein de l'équipe-projet PARIS

travailleurs en raison d'un ralentissement dans l'exécution des requêtes. Une étude préliminaire analysant l'intégration du canevas ***Dynaco*** dans une collection est présentée dans [3].

Améliorations

Nous pensons que le langage XML apporte une complexité et une limitation aux modèles que nous avons proposé. Il nous semble plus intéressant de spécifier un langage d'assemblage de haut niveau qui se rapprocherait d'un langage de programmation avec des structures de contrôles, des instructions pour définir des types, déclarer des composants, connecter des ports, construire des collections, etc.

Il serait aussi intéressant d'adopter une approche générique qui séparerait la spécification d'un langage abstrait d'une technologie particulière de composants ou de flux de travail. En d'autres termes, il s'agirait de suivre une démarche *MDA* (*Model Driven Architecture*) pour spécifier ce langage. Par exemple, les ports possibles d'un composant n'ont pas besoin d'être définis suivant les particularités d'un modèle. La spécification d'un ensemble minimal extensible ou d'un type paramétrable pour des ports spatiaux et temporels semble envisageable. Ensuite, le langage serait à personnaliser et potentiellement à étendre, selon un choix souhaité d'un modèle de composants et d'un langage de flux de travail.

À long terme, une sémantique opérationnelle est aussi nécessaire pour prouver notre proposition de langage d'assemblage. Cette sémantique est importante pour tester la validité d'un assemblage qui doit être déterministe du comportement attendu à l'exécution d'une application. Elle est aussi à considérer pour implémenter des outils de gestion automatique d'un assemblage, plus précisément pour la gestion du cycle de vie des composants durant l'exécution. Dans le chapitre 9 nous avons cité des pistes sur le calcul formel qu'il serait possible d'envisager pour cette perspective de travail.

Évolutions du modèle de programmation abstrait

Dans le prolongement de nos travaux de recherche, il serait intéressant de poursuivre l'étude de solutions d'abstraction pour d'autres paradigmes de programmation. Dans le chapitre 4, nous avons cité des exemples de paradigmes pour la programmation parallèle, comme le *pipeline*, le *diviser pour régner* ou la décomposition géométrique. Exprimer un pipeline à travers une suite de connexions d'une séquence de composants imposerait le respect d'un ordre d'exécution d'opérations et un ordre de données à traiter. La gestion de cet ordre ainsi que l'optimisation des communications entre les stages du pipeline sont à gérer de manière transparente aux programmeurs. Quant au paradigme *diviser pour régner*, il nécessiterait une adaptation du degré de parallélisme maximum, en fonction des ressources et des données en entrée. Il peut demander aussi une gestion dynamique du nombre de composants suivant la variation dans le temps du nombre de calculs concurrents. Même si la programmation parallèle occupe une grande place dans le calcul scientifique, il peut exister d'autres paradigmes de programmation distribuées qui seraient aussi intéressant de considérer dans notre étude à long terme.

L'intérêt de traiter plusieurs paradigmes de programmation est d'élargir les types d'application qui bénéficieraient d'une simplicité de programmation, d'une amélioration de la réutilisation, d'une indépendance vis à vis du matériel et de la possibilité d'adaptation à différents contextes d'exécution.

Bibliographie

[11] CERFACS - European Centre for Research and Advanced Training in Scientific Computation. ***http://www.cerfacs.fr/***.

[12] Chapel language specification 0.702. ***http://chapel.cs.washington.edu/spec-0.702.pdf***.

[13] CRAY The Supercomputer Company.***http://www.cray.com/***.

[14] Enabling Grids for E-sciencE. ***http://www.eu-egee.org/***.

[15] Fujitsu : The possibilities are infinite. ***http://www.fujitsu.com/***.

[16] Julia. ***http:http://fractal.objectweb.org/julia***.

[17] Laboratoire d'Océanographie et du Climat : Expérimentations et Approches Numériques. ***http://www.locean-ipsl.upmc.fr/***.

[18] Le réseau National de t élécommunications pour la Technologie l'Enseignement et la Recherche. ***http://www.renater.fr/***.

[19] Lego. ***http://graal.ens-lyon.fr/LEGO/***.

[20] LoadLeveler for AIX 5L and Linux V3.3.1 using and administering. ***http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%/com.ibm.cluster.loadl.doc/loadl331/am2ug30302.html***.

[21] OpenPBS. ***http://www.openpbs.org***.

[22] Skeletal parallelism. ***http://homepages.inf.ed.ac.uk/mic/Skeletons/***.

[23] TeraGrid. ***http://www.teragrid.org/***.

[24] The Fortress Language Specification, version 1.0beta. ***http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf***.

[25] TOP500 Supercomputer Sites. ***http://www.top500.org/***.

[26] Total runoff integrating pathways. ***http://hydro.iis.u-tokyo.ac.jp/~taikan/TRIPDATA/TRIPDATA.html***.

[27] Uniform Interface to Computing Resources. ***http://www.unicore.eu/***.

[28] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance grid programming in the grid.it project. In *International Workshop on Component Models and Systems for Grid Applications*, Saint-Malo, France, july 2004. Springer-Verlag.

[29] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The grid application toolkit : Toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93(3) :534–550, march 2005.

[30] Ilkay Altintas, Adam Birnbaum, Kim K. Baldridge, Wibke Sudholt, Mark Miller, Celine Amoreira, and Yohann. A framework for the design and reuse of grid workflows. In *First International Workshop on Scientific Applications of Grid Computing (SAG'04))*, pages 120–133, Berlin/Heidelberg, 2005. Springer.

[31] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services version 1.1. Technical report, May 2003.

[32] Gabriel Antoniu. *DSM-PM2 : une plate-forme portable d'implémentation de protocoles de cohérence multithread pour MVP*. PhD thesis, ENS Lyon, 2001.

[33] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. A practical evaluation of a data consistency protocol for efficient visualization in grid applications. In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2006)*, volume 4395 of *Lecture Notes in Computer Science*, pages 692–706, Rio de Janeiro, Brazil, July 2006. Held in conjunction with VECPAR'06, Springer Verlag.

[34] Didier Badouel, Kadi Bouatouch, and Thierry Priol. Ray tracing on distributed memory parallel computers : Strategies for distributing computation and data. *IEEE Computer Graphics and Application*, 14(4) :69–77, July 1994.

[35] L. Baduel, F. Baude, and D. Caromel.Efficient, flexible, and typed group communications in java. *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, 2002.

[36] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Gener. Comput. Syst.*, 13(4-5) :361–372, 1998.

[37] L. Bellissard, S. Ben Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with olan. In J.P. Briot and J.M. Geib et A. Yonezawa, editors, *International Workshop on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, Tokyo, 1996. Springer Berlin/-Heidelberg.

[38] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2) :163–202, 2006.

[39] Julien Bigot and Christian Pérez. Enabling collective communications between components. In *CompFrameÓ7 : Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 121–130, New York, NY, USA, 21-22 October 2007. ACM Press.

[40] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, 1984.

[41] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lantéri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, November 2006.

[42] G. W. Brams. *Réseaux de Petri : théorie et pratique*. 1983.

[43] T. Bures, P. Hnetynka, and F. PlasilSofa 2.0 : Balancing advanced features in a hierarchical component model. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 40–48, Washington, DC, USA, August 2006.

[44] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[45] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems : XtremWeb architecture, programming models, security, tests and convergence with grid. *FGCS*, 21(3) :417–437, 2005.

[46] E. Caron, F. Desprez, F. Lombard, J.M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[47] H. Casanova and J. Dongarra. NetSolve : A Network-Enabled Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3) :212–223, 1997.

[48] Pushpinder Kaur Chouhan. *Automatic Deployment for Application Service Provider Environments*. PhD thesis, École normale supérieure de Lyon, France, September 2006.

[49] Murray Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.

[50] Murray Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3) :389–406, march 2004.

[51] Loïc Cudennec. Modèles et protocoles de cohérence des données en environnement volatil. Technical Report inria-00000979, IRISA-Université de Rennes 1, jun 2005.

[52] N. Currle-Linde, F. Boes, and M. Resch. GriCoL : A Language for Scientific Grids. In *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*, Amsterdam, Netherlands, 2006.

[53] N. Currle-Linde, U. Kuester, M. Resch, and B. Risio. Science experimental grid laboratory (segl) dynamical parameter study in distributed systems. In *ParCo 2005 - Parallel Computing*, Malaga, Spain, September 2005.

[54] Anne Cuzol and Etienne Memin. A stochastic filter for fluid motion tracking. In *10th IEEE International Conference on Computer Vision, ICCV'05*, Beijing, China, october 2005.

[55] Marco Danelutto. QoS in parallel programming through application managers. In *International Euromicro PDP : Parallel Distributed and network-based Processing*, pages 282–289, Lugano, Switzerland, february 2005. IEEE.

[56] A. Denis. *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*. PhD thesis, Université de Rennes 1, IRISA, Rennes, December 2003.

[57] E. Bruneton and T. Coupaye and J.B. Stefani. The Fractal Component Model, version 2.0-3. Technical report, ObjectWeb consortium, February 2004.

[58] EJB 3.0 Expert Group. Jsr 220 : Enterprise javabeans version 3.0. Technical report, December 2005.

[59] Thomas Fahringer, Jun Qin, and Stefan Hainzer. Specification of Grid Workflow Applications with AGWL : An Abstract Grid Workflow Language. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and Grid 2005 (CCGrid 2005)*, volume 2, pages 676–685, Cardiff, UK, May 2005.

[60] P. Felber and R. Guerraoui. Programming with object groups in CORBA. *IEEE Concurrency*, 8(1) :48–58, 2000.

[61] Micheal J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21 :948–960, 1972.

[62] High Performance Fortran Forum. High performance fortran language specification, January 1997. Version 2.0.

[63] I. Foster and C. Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[64] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *Intl Journal of Supercomputer Applications*, 15(3), 2001.

[65] Ian Foster. What is the grid ? a three point checklist. *GRIDtoday*, 1(6), July 2002. **`http://www.gridtoday.com/02/0722/100136.html`**.

[66] Ian T. Foster. Globus toolkit version 4 : Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4) :513–520, 2006.

[67] Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, Tony Field, and John Darlington. ICENI : Optimisation of component applications within a grid environment. *Journal of Parallel Computing*, 28(12) :1753–1772, 2002.

[68] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[69] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. Massachusetts Institute of Technology Press, 1994.

[70] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. Massachusetts Institute of Technology Press, 1999.

[71] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, Jan 1995.

[72] Mathieu Jan. *JuxMem : un service de partage transparent de données pour grilles de calculs fondé sur une approche pair-à-pair*. PhD thesis, Université de Rennes 1, IRISA, Rennes, 2006.

[73] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2 : A Grid Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5) :551 – 563, 2003.

[74] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. In *IEEE Computer Society*, pages 78–83, Los Alamitos, CA, USA, February 2001.

[75] Manojkumar Krishnan, Yuri Alexeev, Theresa L. Windus, and Jarek Nieplocha. Multilevel parallelism in computational chemistry using common component architecture and global arrays. In *SC '05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 23, Washington, DC, USA, 2005. IEEE Computer Society.

[76] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXcluster : a closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2) :130–146, 1986.

[77] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. PhD thesis, Université de Rennes 1, IRISA, Rennes, 2005.

[78] M. Beisiegel and H. Blohm and D. Booz and M. Edwards and O. Hurley and S. Ielceanu and A. Miller and A. Karmarkar and A. Malhotra and J. Marino and M. Nally and E. Newcomer and S. Patil and G. Pavlik and M. Raepple and M. Rowley and K. Tam and S. Vorthmann and P. Walker and L. WatermanSCA Service Component Architecture - Assembly Model Specification, version 1.0. Technical report, Open Service Oriented Architecture collaboration (OSOA), 2007.

[79] Jeff Magee, Naranker Dulay, and Jeff Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburgh, US, March 1994.

[80] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. Efficient mpi collective operations for clusters in long-and-fast networks. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006. IEEE.

[81] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO.

[82] Theo Dirk Meijler and Oscar Nierstrasz. Beyond objects : Components. In M. P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems : Current Trends and Directions*, pages 49–78. Academic Press, 1997.

[83] Robin Milner. *Communication and Concurrency*. Prentice Hall international, 1989.

[84] Robin Milner. *Communicating and Mobile Systems : The Pi Calculus*. Cambridge University Press, New York, NY, USA, 1999.

[85] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. *Future Generation Computer Systems*, 20(2), January 2004.

[86] Anand Natrajan, Anh Nguyen-Tuong, Marty Humphrey, Michael Herrick, Brian P. Clarke, and Andrew S. Grimshaw. The legion grid portal. *Concurrency and Computation : Practice and Experience*, 14(13-15) :1365–1394, 2002.

[87] Partners of the CoreGrid Programming Model institute. Basic features of the grid component model. Technical report, march 2007. D.PM.04.

[88] OMG. CORBA component model, v4.0. Document formal/2006-04-01, April 2006.

[89] OMG. Deployment and Configuration of Component-based Distributed Applications Specification, v4.0. Document formal/2006-04-02, April 2006.

[90] OMG. Unified modeling language : Superstructure, version 2.1.1. Document formal/2007-02-05, February 2007.

[91] David Orchard, David Booth, Francis McCabe, Eric Newcomer, Mike Champion, Hugo Haas, and Christopher Ferris. Web services architecture. Technical report, W3C Web Services Architecture Working Group, 2004.

[92] Sabri Pllana and Thomas Fahringer. Uml based modeling of performance oriented parallel and distributed applications. In E. Yucesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes, editors, *In proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.

[93] J. Protić, M. Tomasević, and V. Milutinović. *Distributed Shared Memory : Concepts and Systems*. IEEE, August 1997.

[94] André Ribes. *Contribution à la conception d'un modèle de programmation parallèle et distribué et sa mise en oeuvre au sein de plates-formes orientées objet et composant*. PhD thesis, IRISA, Université de Rennes 1, IRISA, Rennes, 2004.

[95] Ward Rosenberry, David Kenney, and Gerry Fisher*Understanding DCE*. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 1992.

[96] Felix Schüller, Jun Qin, Farrukh Nadeem, Radu Prodan, Thomas Fahringer, and Georg Mayr. Performance, scalability and quality of the meteorological grid workflow meteoag. In *Second Austrian Grid Symposium*. OCG Verlag, September 2006.

[97] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A component model engineered with components and aspects. In *The 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063, pages 139–153, Vasteras, Sweden, june 2006. Springer.

[98] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C.A. Lee, and H. Casanova.Overview of GridRPC : A Remote Procedure Call API for Grid Computing. In *Grid Computing*, pages 274 – 278, 2002.

[99] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of gridrpc : A remote procedure call api for grid computing. In *Third International Workshop on Grid Computing (GRID'2002)*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278, Baltimore, MD, USA, 2002. Springer Berlin.

[100] R. Srinivasan. RPC : Remote procedure call protocol specification version 2. Ietf request for comment 1831, August 1995.

[101] D. Swade. 'it will not slice a pineapple'-charles babbage and the first computer. In *IEE Review*, volume 37, pages 217–220, 20 June 1991.

[102] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, second edition, 2002.

[103] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *J. Grid Compututing*, 1(1) :41–51, 2003.

[104] A.S. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002.

[105] Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields, editors. *Workflows for e-Science : Scientific Workflows for Grids*. Springer, New York, 2007.

[106] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4) :153–169, September 2005.

[107] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing : Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[108] Fahringer Thomas, Prodan Radu, Duan Rubing, Nerieri Francesco, Podlipnig Stefan, Qin Jun, Siddiqui Mumtaz, Truong Hong-Linh, Villazon Alex, and Wieczorek Marek. ASKALON : A Grid Application Development and Computing Environment. In *Proceedings of the 6th International Workshop on Grid Computing*, pages 122–131, Seattle, USA, November 2005.

[109] B.A. Toole. Ada byron, lady lovelace, an analyst and metaphysician. *Annals of the History of Computing, IEEE*, 18(3) :4–12, Fall 1996.

[110] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl : Yet another workflow language. *Information Systems*, 30(4) :245–275, 2005.

[111] W.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3) :5–51, July 2003.

[112] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4) :171–200, september 2005.

[113] Berkeley Open Infrastructure for Network Computing. ***http://boinc.berkeley.edu/***, 2002.

Hinde Lilia BOUZIANE

De l'abstraction des modèles de composants logiciels pour la programmation d'applications scientifiques distribuées

Mots-clefs : modèle de composants logiciels, applications distribuées, ressources matérielles, abstraction, paradigme de programmation, partage de données, maître-travailleur, flux de travail.

La complexité croissante et la large échelle des applications scientifiques distribuées font de leur programmation un véritable défi. La programmation à base de composants logiciels se révèle prometteuse par son approche d'assemblage et de réutilisation d'entités logicielles. Cependant, il est aussi important de découpler la programmation des ressources d'exécution. Celles-ci sont de nature variée, complexe et ne cessent d'évoluer (super-calculateurs, grappes, grilles, etc.). Le niveau d'abstraction de la programmation se doit d'être suffisamment élevé pour masquer cette nature et assurer la portabilité et l'efficacité d'exécution des applications sur différentes ressources. À cet égard, les modèles de composants actuels sont de bas niveau.

Nos travaux de thèse contribuent à élever le niveau d'abstraction des modèles de composants existants, selon une étude générique. Ils proposent des moyens permettant de rendre visible un paradigme de programmation à travers un assemblage. Le but est qu'un concepteur puisse se concentrer sur des aspects métiers et laisser le soin à une plate-forme d'exécution de gérer les dépendances d'un paradigme vis à vis des ressources. Des modèles sont proposés pour abstraire le partage de donnée entre composants (localisation, réplication, accès concurrents, etc.), le paradigme maître-travailleur (collecte des travailleurs, ordonnancement des requêtes, etc.) et le flux de travail d'une application (dynamicité de l'assemblage, exploitation des ressources, etc.).

Les concepts proposés ont été validés par leur projection sur des technologies existantes (CCM, GCM, CCA) ainsi que par des expérimentations sur la plate-forme expérimentale *Grid'5000*.