THÈSE D'HABILITATION DE L'UNIVERSITÉ PIERRE ET MARIE CURIE
PARIS VI

spécialité:
INFORMATIQUE

présentée par
Fabrice KORDON

pour l'obtention d'une
HABILITATION À DIRIGER DES RECHERCHES

Sujet de la thèse

Prototypage d'Applications Coopératives Réparties et d'Environnements de Génie Logiciel

Soutenue le 2 Décembre 1998 devant le jury composé de

Claude **Timsit**, Professeur à l'Université de Versailles St-Quentin et à L'École Supérieure d'Électricité	Président
Didier **Buchs**, Adjoint Scientifique à l'Ecole Polytechnique Fédérale de Lausanne	Rapporteur
Jean-Claude **Derniame**, Professeur à l'Institut Polytechnique de Lorraine	Rapporteur
Eric **Simon**, Directeur de Recherches à l'INRIA	Rapporteur
Pascal **Estraillier**, Professeur à l'Université Paris VI	Directeur
Jean-Pierre **Briot**, Chargé de Recherches au CNRS	Examinateur
Christine **Choppy**, Professeur à l'Université Paris XIII	Examinateur
Claude **Girault**, Professeur à l'Université Paris VI	Examinateur

A Alix, la femme de mes rêves
et Chéphrèn, le prototype du fils idéal...

... pour leur Amour, leur Attention et leur Soutien

Je tiens à remercier vivement…

Messieurs Claude Girault et Pascal Estraillier, tous deux Professeurs à l'Université Pierre & Marie Curie, pour leur soutien et leur aide constants depuis des années.

Messieurs Didier Buchs, Adjoint Scientifique à lEcole Polytechnique Fédérale de Lausanne, Jean-Claude Derniame, Professeur à l'Institut Polytechnique de Lorraine et Eric Simon, Directeur de Recherches à l'INRIA pour l'intérêt qu'ils ont manifesté pour mes travaux et l'honneur qu'ils me font d'avoir accepté de rapporter le présent document.

Madame Christine Choppy, Professeur à l'Université Paris XIII et Monsieur Jean-Pierre Briot, Chargé de Recherches au CNRS pour l'intérêt qu'ils ont porté à mes travaux et l'honneur qu'ils me font d'avoir accepté d'être membre de mon jury.

Claude Timsit, Professeur à l'Université de Versailles-Satin-Quentin et à l'Ecole Supérieure d'Electricité, pour m'avoir donné au départ une chance à un âge inhabituel et, à l'autre extrémité, de l'honneur qu'il me fait d'avoir accepté d'être membre de mon jury.

Les membres passés ou présents de l'équipe Systèmes Répartis et Coopératifs pour leur aide au quotidien, leurs remarques et bug reports;-). Je pense en particulier à Jean-Luc Mounier, Marie-Pierre Gervais, Alioune Diagne, Isabelle Vernier, Karim Foughali, William El Kaïm, Emmanuel Paviot-Adet et Xavier Bonnaire.

L'ensemble du personnel enseignant et administratif de l'UFR d'Informatique de l'Université P. & M. Curie avec une mention particulière pour Véronique Varenne puis Brigitte Costes qui ont contribué à faciliter la vie quotidienne de notre équipe.

Table des Matières

Chapitre 3
Prototypage d'Environnements de Génie Logiciel

Chapitre 4
Conclusion et Perspectives

Références Bibliographiques

Table des Figures

Introduction

Le développement et la maintenance d'applications industrielles deviennent sans cesse plus délicats. Les systèmes concernés sont de plus en plus complexes **[185]**, les technologies évoluent rapidement, les cahiers des charges intègrent des facteurs «sociaux» **[119]** et la commercialisation s'effectue dans des «fenêtres» de temps réduites, ce qui implique des temps de production (time to market) courts. Ces raisons expliquent l'existence d'une crise chronique du logiciel **[116]**. Les coûts induits par cette crise sont estimés à quatre-vingt un milliards de dollars en 1995 **[265]**.

L'un des problèmes du développement est l'évaluation du système que l'on construit. Le test après implémentation n'est pas fiable, car il est difficile de traiter toutes les exécutions possibles dans le cas de systèmes complexes. L'explosion du vol Ariane-501 **[66]** est un excellent exemple d'erreur triviale résistant à un processus de validation du logiciel élaboré et à des normes de développement contraignantes. La simulation de spécifications exécutables, souvent envisagée pour rechercher des erreurs, n'est pas non plus fiable à 100%.

Pour résoudre ce problème, des méthodes de Génie Logiciel ont été introduites puis mises en œuvre au moyen d'Ateliers de Génie Logiciel. Elles permettent d'identifier les problèmes et les solutions à mettre en œuvre au moyen de modèles décrits dans des formalismes(1) dédiés (comme SA-RT **[134]**, OMT **[245]**, UML **[246]**, etc.). Cependant, une fois les solutions décrites, leur implémentation engendre une dérive importante liée en général aux choix de réalisation effectués par les équipes de développement **[213]**.

Le prototypage défini par l'IEEE **[175]** comme une approche «privilégiant le développement de prototypes dès les premières étapes du cycle de vie du logiciel afin d'obtenir des réactions et des analyses utiles pour la suite du processus de développement»(2) apporte une solution en reliant plus étroitement spécification et produit. Cette définition du prototypage est diversement interprétée **[190, 281, 167]** tout au long du cycle de vie du logiciel. Il devient maquettage (prototype jetable), génération de code ou, plus récemment, approche de conception /

(1) Le terme formalisme est ici pris au sens de «technique de représentation d'architectures logicielles», il est explicitement utilisé dans **[254]** et **[235]** .

(2) La citation originale est: «A type of prototyping in which emphasis is placed on developping prototypes early in the development process to permit early feedback and analysis in support of the development process».

développement / maintenance (introduction du RAD[3]-Cycle **[157]**). Des mesures tendent à prouver qu'une démarche de prototypage intégrée dans un processus de développement (comme c'est le cas dans le domaine du matériel) réduisent considérablement les phases de vérification **[255]**.

Qu'il soit maquettage, génération de code ou approche de développement, le prototypage s'appuie sur la notion de modèle (jetable ou non) qui constitue un «fil conducteur» de la spécification d'une solution à sa réalisation. Ces modèles doivent être exprimé dans un formalisme rigoureux (i.e. non ambigu). Il supporte en général l'exécution (simulation), sert de base à la génération automatique de programmes et peut également subsister jusqu'à la phase de maintenance de l'application (c'est le cas dans l'outil SAO **[52]** développé par l'Aérospatiale).

Les méthodes formelles sont communément reconnues comme pouvant apporter une solution au problème de l'évaluation des modèles et ce, très tôt dans le cycle de vie du logiciel **[194]**. Les fondements mathématiques sur lesquels elles reposent sont exploitables pour vérifier des propriétés attendues à l'aide d'une axiomatique, évaluer des propriétés structurelles ou générer l'espace des états du système (model checking). Leur application pose cependant des problèmes **[193]** comme :

- les notations utilisées sont difficilement exploitables pour des programmeurs souvent peu versés dans les techniques mathématiques,
- les notations mathématiques associées sont applicables dans différents domaines mais nécessitent en réalité un travail extrêmement différent selon les domaines[4],
- les informations nécessaires à leur application sont soit absentes, soit formulées de manière inadéquate dans les formalismes couramment utilisés dans l'industrie,
- l'application de ces techniques à des cas de «taille réelle» pose des problèmes de complexité difficiles à maîtriser,
- des mécanismes couramment utilisés en conception (héritage, migration, etc.) sont mal pris en compte.

En parallèle du travail important effectué afin de rendre les méthodes formelles exploitables dans un contexte de prototypage, il faut étudier leurs liens potentiels avec les méthodes de spécification utilisées dans l'industrie. Ce lien implique une utilisation sous-jacente des approches formelles (approche de type «push-button») dans des domaines d'application bien cernés **[194]**.

Dans ce contexte, mes travaux s'articulent selon deux axes de recherche :

- la mise en œuvre d'une méthodologie de prototypage d'application répar-

(3) Rapid Application Development.

(4) Par exemple, on peut associer des traitements aux places d'un réseaux de Petri; les transitions étant alors associées à des changements d'état. Inversement, on peut associer des traitements aux transitions et considérer les places comme des ressources. Ces deux utilisations parfaitement licites d'une même méthode formelle ne sont pas facilement interchangeables pour les utilisateurs, en particulier dès qu'il s'agit de vérifier un système.

tie exploitant les possibilités de méthodes formelles,
- la définition de techniques pour le prototypage d'Environnements de Génie Logiciel.

Mon premier objectif est de définir une méthodologie de prototypage prenant en compte à la fois l'aspect vérification (avec utilisation de techniques formelles lorsque c'est possible) et l'aspect génération de programmes dans le contexte des systèmes répartis. Pour cela, nous devons :

- exploiter les réseaux de Petri (c'est la méthode formelle que nous avons choisie) depuis un formalisme de haut niveau pour évaluer les propriétés puis retranscrire les résultats;
- réduire le fossé entre spécification et système en s'appuyant sur les techniques de génération automatique de programmes;
- réaliser des outils automatisant ces opérations afin d'évaluer l'apport des techniques mises en œuvre sur des problèmes de grande taille.

Le développement d'outils pour appliquer nos techniques à des exemples de taille réelle m'a amené à considérer un second objectif : la réalisation d'Ateliers de Génie Logiciel à très faible coût de développement. Dans ce cadre, je supervise des travaux visant à la définition d'outils de prototypage d'environnements de Génie Logiciel dont nous sommes les premiers consommateurs.

Le Chapitre 1 présente la problématique de mes axes de recherches. À partir des leçons issues de travaux et d'expérimentation sur la génération de code depuis les réseaux de Petri, il pose les principes d'une méthode de prototypage dans laquelle la notion de modèle joue un rôle central pour l'évaluation et pour la production de programmes. Il détaille également nos besoins en génération d'Ateliers de Génie Logiciel.

Le Chapitre 2 expose l'état de mes travaux sur MARS, la méthodologie de prototypage, en les comparants à d'autres études similaires. MARS utilise les possibilités des réseaux de Petri colorés à partir de deux formalismes de haut niveau. Le premier est consacré à la conception et la validation d'un système réparti construit par assemblage de composants élémentaires; le second est dédié à la génération d'applications réparties. Des règles de transformation permettent de passer du modèle Conceptuel au modèle Opérationnel. Il décrit ensuite les fonctions de l'environnement sur lequel nous expérimentons nos techniques et le compare à d'autres environnements caractéristiques.

Le Chapitre 3 présente FrameKit, une plate-forme de développement d'Ateliers de Génie Logiciel, en la comparant à des études comportant des traits similaires. FrameKit en outre comporte des outils de haut niveau dédiés à la production de générateurs de code, et de simulateurs.

Le Chapitre 4 conclut le présent mémoire et expose les perspectives de recherche de mes travaux sur les aspects vérification de spécifications et / ou programmes,

optimisation des programmes générés, ainsi que par le prototypage rapide d'Ateliers de Génie Logiciel.

Chapitre 1
Problématique et Motivations

1. Génération de code à partir de réseaux de Petri : les leçons

L'objectif de ma thèse **[163]**, soutenue en 1992, était l'étude de techniques de prototypage à partir de réseaux de Petri. À l'époque, le terme «prototypage» signifiait principalement «génération de code». Cela minimisait d'autres aspects (comme la fiabilité de la spécification, l'évaluation de la solution proposée, l'optimisation des performances du logiciel produit, etc.) qui seront développés plus loin. Ce point de vue se justifiait par la volonté d'exploiter en l'état les travaux sur la modélisation et la vérification de systèmes au moyen de réseaux de Petri **[96]**.

1.1. Approches d'implémentation centralisées et décentralisées d'un réseau de Petri

Deux approches de génération de code ont tout d'abord été étudiées par la communauté. La première vise à la production d'un «joueur de jeton» implémenté sur un composant matériel **[275, 258]** ou de manière logicielle **[215, 69, 211]** . Pour limiter le goulot d'étranglement au niveau du «joueur de jeton», des techniques sophistiquées de filtrage des transitions furent élaborées **[69, 211]** .

Cette approche n'est cependant pas exploitable pour la génération de code. En effet, même muni d'un algorithme de filtrage des transitions efficace, le joueur de jetons peut difficilement être divisé en tâches indépendantes car, pour évaluer les préconditions des transitions, il faut connaître le contexte global du système. L'exécuteur constitue donc un goulot d'étranglement qui ruine les performances sur de gros modèles.

Cette approche est cependant utilisée pour accélérer des séquences de simulation. Des outils comme Cabernet **[231]**, PEP **[126]** ou LOOPN **[180]** génèrent du code capable de piloter l'interface utilisateur en envoyant des ordres graphiques mettant en évidence les évolutions du modèle ainsi simulé.

Une seconde approche est de répartir complètement l'implémentation du réseau de Petri **[135, 273]** . Ainsi, dans le code généré, à chaque transition ou à chaque place du modèle correspond une tâche. L'évaluation des préconditions est pilotée par les tâches-transitions qui interrogent les tâches-places concernées. Pour

gérer les conflits, des mécanismes spécifiques basés sur l'estampillage des requêtes et un protocole complexe doivent être mis en œuvre.

Une telle approche ne conduit pas non plus à un code efficace : les surcoûts d'exécution lié à la gestion d'un nombre élevé de tâches et le protocole de résolution des conflits entre transitions partageant des places préconditions ruinent les performances.

1.2. L'approche hybride

Les stratégies mises en œuvre dans ma thèse s'articulaient autour de deux objectifs majeurs : le partitionnement d'un réseau de Petri en vue d'en obtenir une implémentation «hybride» (ni centralisée, ni complètement décentralisée) et la possibilité d'intégrer le code généré dans un environnement d'exécution externe préexistant.

1.2.1. Partitionnement pour la génération de code

Pour atteindre le premier objectif, j'ai défini puis mis en œuvre en collaboration avec J.F. Peyre un algorithme de partitionnement d'un modèle s'appuyant sur les invariants de places calculés sur un réseau de Petri «décoloré[(1)]» équivalent appelé «modèle structurel» **[161]**. Cet algorithme permet de caractériser des machines à états **[130]** communiquant par le biais de places d'interfaces (communication asynchrone **[264, 64]**) ou de transitions partagées (communications synchrones **[233, 16]**). Les machines à états sont ensuite implémentées par des tâches, les places d'interfaces par des zones de mémoire communes et les transitions partagées par des serveurs gérant les synchronisations.

Une telle approche permet, à partir des composants caractérisés lors du partitionnement (appelés *objets de génération*), de proposer un modèle générique d'architecture logicielle **[164, 166]** implémentable au moyen de différents langages de programmation et selon différentes stratégies. Ainsi, une partie importante du code généré est indépendante des réseaux de Petri traités.

Le modèle est partitionnable selon la technique élaborée s'il respecte quatre propriétés définies dans **[163]** (lorsque ces propriétés ne sont pas toutes vérifiées, l'algorithme de partitionnement échoue). Des transformations permettent cependant de débloquer la situation et de rendre le modèle partitionnable. Elles sont hélas difficiles à automatiser car leur application nécessite une connaissance de la sémantique de fonctionnement du système.

1.2.2. Intégration d'un modèle dans son environnement externe

Les travaux concernant la génération de code à partir de réseaux de Petri n'avaient pas vraiment exploré le problème de l'intégration du code produit avec un environnement d'exécution. La stratégie adoptée est en général d'associer des appels de procédures aux transitions : lorsqu'une transition est tirée, la

(1) La version «décolorée» d'un réseau de Petri coloré est ici le réseau de Petri Place/Transition structurellement équivalent obtenu sans dépliage des domaines de couleur. En fait, cela revient à dire que les invariants calculés ne prennent pas en compte les classes de couleur du modèle.

procédure correspondante est invoquée. L'avantage de ce modèle simple est qu'il se prête aussi bien à la simulation qu'à la génération de code.

L'application de cette technique suppose cependant que les procédures associées aux transitions n'ont pas d'effets de bord entre elles. Cette hypothèse est raisonnable dans le contexte des études choisies (implémentation matérielle ou conception d'Ateliers Flexibles) mais cesse de l'être lorsque l'on veut interfacer le code généré avec une bibliothèque graphique, un système d'exploitation ou tout autre environnement d'exécution contextuel. Comme les effets de bord entre les invocations des procédures ne sont pas exprimés dans le modèle (par exemple : le fait qu'il faut ouvrir un fichier avant d'y lire une information), l'opération de validation avant la génération de code perd alors tout son sens puisqu'elle ne peut les prendre en compte.

C'est pourquoi la notion de *composant externe* **[160, 74]** a été introduite pour modéliser une abstraction de l'environnement et de ses interfaces avec le système. Il est ainsi possible d'exprimer les dépendances entre services et générer des résultats cohérents à l'invocation de fonctions. La validation s'effectue sur le modèle complet et la génération de code sur la partie décrivant le système. L'environnement est réintégré par édition de lien au prototype ainsi généré.

1.3. Réalisations et expérimentations

À partir de ces travaux, un outil a été développé et diffusé sur Internet avec l'environnement CPN-AMI (versions 1.x) **[198]**. Cet outil a constitué l'une de nos contributions logicielles au projet européen IRENA **[146]**.

Ada83, le langage cible choisi pour l'expérimentation, supporte la gestion du parallélisme. Des aspects comme la communication entre tâches sont ainsi directement gérés par l'exécutif, même si cela se fait à un certain prix, les communications au sein du prototype étant essentiellement asynchrones[(2)]. Des applications (comprenant jusqu'à 500 tâches Ada) ont été générées pour des modèles de grande taille, les performances d'exécution du prototype restant parfaitement satisfaisantes.

Cet outil a permis d'expérimenter la génération de code à partir de réseaux de Petri, étudier l'impact de la définition / utilisation de bibliothèques de composants externes et d'analyser de différentes stratégies de répartition des prototypes. Deux observations ont été effectuées :

1) Le calcul des objets de génération à partir d'un partitionnement des réseaux de Petri n'est pas toujours possible. Outre les transformations proposées dans **[163]**, des travaux en vue de résoudre ce problème se poursuivent suivant deux directions :
   - À partir des invariants colorés, afin de saisir plus finement la sémantique du système ainsi modélisé **[38, 39, 230]** ;

(2) Ces travaux ont été menés avec la version 83 **[5]** du langage qui n'intègre pas de mécanisme de communication asynchrone comme les «protected records» d'Ada95 **[6]**, la mise en place d'un mécanisme basé sur l'existence de «tâches agents» **[242]** étant alors nécessaire.

- À partir des Arbres de Décision Binaires (BDD) **[195]**; le contexte est en général celui de la simulation, comme dans **[59]**.

Cependant, au-delà d'une certaine complexité, les algorithmes de partitionnement se heurtent systématiquement à un manque de compréhension de la sémantique du modèle. Une telle compréhension est le fait de l'utilisateur et, même si la recherche d'un partitionnement peut lui apporter des informations intéressantes, lui seul est capable de trancher. L'application de ces nouvelles stratégies n'est donc pas réellement automatisable.

2) La technique de partitionnement identifie parfois des configurations aux performances surprenantes. Considérons par exemple un système dans lequel plusieurs entités communiquent au moyen de sockets de communication Unix. Leur modélisation aboutit à un modèle complexe. Devant la diversité des techniques de modélisation d'un tel mécanisme, il est impossible à un générateur de code d'identifier clairement qu'un sous-modèle est une socket de communication. Des processus séquentiels supplémentaires (réalisant correctement l'objectif mais inutiles) vont être détectés lors du partitionnement et le générateur de code ne saura pas implémenter ces objets en utilisant les services systèmes adéquats.

Définir des règles de construction des réseaux de Petri apporterait une solution au problème (1) et peut-être, dans une certaine mesure au problème (2). De même, une utilisation des composants externes dans le cadre de gabarits de conception pourrait apporter une solution au problème (2) **[222]**. Cependant, la mise en œuvre de telles solutions impliquerait la définition d'une méthode d'utilisation lourde, pas transparente pour l'utilisateur, peu flexible et par conséquent peu conviviale.

1.4. Structuration des réseaux de Petri

Des problèmes similaires sont rencontrés en modélisation : comment bien structurer le modèle afin d'en faciliter la validation, identifier facilement les «bonnes» propriétés à rechercher ou tout simplement augmenter la lisibilité du modèle. Le principal problème identifié est l'absence de structuration et de construction de haut niveau dans les réseaux de Petri **[17, 78]**.

1.4.1. Extensions des réseaux de Petri

On se heurte aux limites de ce type de formalisme[(3)]. Très souples et très généralistes, les réseaux de Petri sont «l'assembleur» de la modélisation. La communauté a d'ailleurs identifié le problème et des travaux étudient l'extension de ce modèle pour y intégrer des fonctionnalités de haut niveau. Deux directions sont actuellement étudiées :

- L'extension des réseaux de Petri intègre des mécanismes de haut niveau, ou adaptés à un domaine d'utilisation **[49]**. L'introduction de mécanismes objets se fait généralement au niveau de la structure du réseau de Petri

(3) Un formalisme est une technique de représentation basée sur des conventions. Lorsque ces conventions reposent sur des fondements mathématiques permettant le calcul de propriétés, le formalisme est formel.

(objets hiérarchiques et mécanisme d'instanciation) **[236, 41, 278, 256, 20, 182]**. **[276]** propose une vision hiérarchique différente en considérant le marquage comme un sous-réseau de Petri à part entière décrivant le cycle de vie d'un objet. D'autres s'intéressent à la gestion du temps ou encore à la liaison avec des flux de production (work-flows) **[18]**.

- L'association des réseaux de Petri avec un «formalisme structurant» (en général orienté objet) est également considérée. Des travaux dans ce sens ont été menés avec HOOD dans **[82, 224]** , OOA dans **[206]** et Shlayer & Mellor dans **[181]**. À ce titre, signalons l'intéressante association de réseaux de Petri de haut niveau (objets et hiérarchiques) avec des Spécifications Algébriques dans CO-OPN **[42, 30]** . Ici, les réseaux de Petri structurent la spécification et fournissent un cadre pour la description du comportement des systèmes. Les spécifications algébriques sont dédiées au typage des informations et supportent l'étude des changements d'états des variables du système.

Ces deux approches sont complémentaires. La première vise le long terme sur les aspects validation car l'ajout de capacités nouvelles empêche le calcul d'un certain nombre de propriétés déjà connues sur des classes de réseaux de Petri plus simples (par exemple, certaines propriétés des réseaux de Petri Place / Transition ne sont toujours pas étendues à des réseaux de Petri Colorés). L'autre approche introduit de la structuration tout en conservant les capacités de validation de classes plus simples de réseaux de Petri, souvent au prix de manipulations délicates pour un non expert dans les deux formalismes (le structurant et les réseaux de Petri).

1.4.2. Encapsulation des réseaux de Petri

L'approche MARS (**M**ethod and **A**nalysis for **R**eliable **S**ystems), initialement introduite par P. Estraillier dans **[95, 15]** , tend vers l'association avec un formalisme structurant. Il s'agit d'*encapsuler* les réseaux de Petri afin de rendre leur utilisation transparente. L'activité de spécification s'effectue au moyen d'une représentation semi-formelle. La validation est réalisée sur une spécification synthétisée à partir du modèle semi-formel **[97]**. Les possibilités de cette représentation doivent donc comporter suffisamment de concepts permettant de guider les utilisateurs dans leur démarche de spécification tout en n'offrant pas de mécanisme susceptible d'engendrer des configurations que l'on ne sait pas valider.

Une telle démarche est également considérée par d'autres équipes de recherche. L'environnement SEA (System Engeneering and Animation), développé à l'Université de Paderborn **[139]** offre, dans une optique similaire, de modéliser des systèmes au moyen d'un langage graphique et hiérarchique **[159]**. Pour exécuter cette spécification, l'environnement transforme la spécification en réseaux de Petri Prédicat / Transition. SEA prend en charge les correspondances entre le formalisme de haut niveau et les réseaux de Petri Prédicat / Transition.

Une autre approche, plus centrée sur la validation, est proposée à l'Institut d'Informatique de l'Université d'Hildesheim avec PEP **[126]**. $B(PN)^2$ **[27, 100]** un langage textuel procédural, encapsule des réseaux de Petri. Nous évoquerons plus longuement ce travail dans le Chapitre 2.

De même, dans le projet PARSE (coopération entre l'Université de Sheffield Hallam, l'Université de New South Wales, l'Université de Sheffield, l'Université de Naple et la société IAG-Oxford Instuments), les réseaux de Petri sont également proposés pour valider des spécifications exprimées au moyen d'un langage de haut niveau **[84, 248]** . On ne peut parler ici d'une volonté délibérée d'encapsulation car le modèle de haut niveau a été conçu avant que la transformation ne soit envisagée. L'idée de cacher l'utilisation des réseaux de Petri à travers des outils de validation est par contre bien présente.

Ainsi, il apparaît plus facile pour les utilisateurs d'utiliser indirectement un formalisme comme les réseaux de Petri dans une méthode de prototypage si l'on souhaite obtenir un code efficace et optimal. C'est ce que je me suis efforcé de démontrer dans mes travaux depuis 1995, en particulier avec l'introduction de H-COSTAM (**H**ierarchical **CO**mmunicating **ST**ate m**A**chine **M**odel), une encapsulation des réseaux de Petri conçue pour la génération de code **[168]**. J'ai également travaillé avec A. Diagne sur une encapsulation similaire mais orientée vérification : OF-Class **[76]**. Ces travaux seront détaillés dans le Chapitre 2.

2. Prototypage et génération de code

Cette section présente les acceptions et les classifications les plus répandues du terme «prototypage». Elle décrit ensuite la vision actuelle d'une méthodologie de développement basée sur une démarche de prototypage. Mes travaux s'insèrent dans cette approche qui a de l'avenir **[24]** et l'applique au domaine des systèmes répartis.

2.1. Problématique du prototypage

Trois types d'objectifs distincts peuvent être identifiés dans le prototypage :

1) Le premier objectif concerne la définition du cahier des charges. Il est nécessaire que les ingénieurs comprennent correctement (i.e. sans interprétation abusive due à une mauvaise connaissance d'un domaine d'activité qui leur est étranger) les besoins d'un «client». À ce titre, un prototype du travail à effectuer facilite la compréhension mutuelle. On se contente à ce stade d'un prototype qui se comporte «comme» le système mais sans qu'aucune logique ne soit mise en œuvre, l'objectif étant de se mettre d'accord avec le «client» sur la manière dont le système à produire devra se comporter.
2) Le second objectif concerne l'expérimentation en cours de développement. Il est important de s'assurer de la pertinence d'une stratégie de réalisation avant d'avoir trop investi sur elle. En cas de succès, la stratégie est validée

et l'on dispose en général d'informations sur la complexité du problème, la manière de l'aborder, ainsi que sur l'évaluation du coût global du système.

3) Le troisième objectif consiste à limiter la dérive entre conception et implémentation afin de réduire la dérive potentielle introduite entre la spécification d'une solution et sa mise en œuvre.

Ces trois types d'objectifs ont une raison commune : obtenir des informations sur l'application en cours de développement. Cependant, les besoins et le type de données recherchées sont très différents car ces objectifs sont considérés à des stades distincts du cahier des charges.

Si on considère le type d'objectif (1), le prototype sera jeté dès qu'un consensus sera intervenu entre «développeurs» et «clients». L'objectif est d'appréhender le problème correctement le plus tôt possible afin d'éviter les erreurs provenant d'une mauvaise compréhension des spécifications de l'utilisateur, reconnues comme responsable de 70% des erreurs dans les logiciels **[142]**. Ainsi, un soin tout particulier sera apporté à la mise au point des caractéristiques fonctionnelles du produit, au détriment de la qualité du prototype.

Dans le cas du type d'objectif (2), le prototype obtenu peut être jeté ou conservé, selon le soin apporté à sa réalisation. La pérennité du prototype est en général directement liée à son degré de couverture (i.e. l'importance des points étudiés par rapport au projet global) mais l'objectif réel est de se prémunir le plus tôt possible des erreurs dans les choix d'implémentation.

Le prototype obtenu en poursuivant le type d'objectif (3) est conservé car l'intérêt porte sur le procédé de réalisation plus que sur le raffinement des spécifications.

Très tôt, différentes entreprises ont compris l'intérêt d'un processus de prototypage orienté selon l'un (ou plusieurs) des objectifs sus-mentionnés. Le mettre en œuvre à l'aide d'un ensemble cohérent d'outils permet d'économiser beaucoup de temps et d'argent pour un résultat plus fiable. Ce n'est d'ailleurs pas par hasard que, dans le domaine du matériel, la tradition de prototypage est très ancienne. Le coût de production de composants électroniques est si élevé qu'une erreur découverte trop tard engendre des coûts prohibitifs. Peu de compagnies ont pu s'offrir le luxe de se tromper plusieurs fois dans ce domaine[(4)].

Indépendamment de l'aspect fiabilité, le prototypage est également vu comme une technique permettant de réduire la durée de réalisation (et par conséquent, de commercialisation) d'un produit. Cela est important, en particulier dans les domaines ou cet aspect constitue un critère de rentabilité et d'efficacité par rapport à la concurrence. Donnons quelques exemples :

- La méthode REE **[48]** permet de valider des scénarii d'attaque ainsi que la pertinence des réponses apportées dans le cadre de systèmes militaires. C'est une bonne illustration des types d'objectifs (1) et (2). L'étude se situe

(4) Par exemple, le problème de la division flottante dans certaines séries du processeur Pentium a coûté plusieurs dizaines de millions de dollars à INTEL qui a finalement dû remplacer sur demande les processeurs de la série défectueuse.

ici essentiellement au niveau des spécifications pour la réalisation, car les outils mettant en œuvre cette méthode ne produisent pas de code;

- APPLBUILDER **[129]** se focalise sur la génération automatique d'interfaces utilisateurs. On touche à la fois le type d'objectif (1) car l'application peut être «déconnectée» de sa logique interne et le type d'objectif (3) car la qualité du code produit permet de l'intégrer à des composants logiciels produits selon des procédés plus classiques;
- La chaîne S.A.O. **[52]** permet à Aérospatiale de développer plus de 80% des logiciels embarqués (calculateurs de bord et interfaces utilisateurs des pilotes) à bord de son dernier avion, l'A-340. On est ici face à une excellente illustration des trois types d'objectifs : (1) puisque l'environnement permet de prototyper des tableaux de bord en accord avec des pilotes, (2) car les stratégies d'implémentation sont étudiées à un niveau opérationnel et (3) puisque le code produit (principalement de l'assembleur) est conservé dans sa majorité. Notons que S.A.O. est également utilisé pendant la phase de maintenance.

Notons que les démarches de prototypage relèvent le plus souvent d'approches méthodologiques et non plus d'une technique (même si parfois un produit a tendance à masquer cet aspect). Ainsi, le prototypage ne doit pas se réduire à de la génération de code ou à de la simulation. Il s'appuie certes sur elles mais doit offrir, pour être efficace, un cadre méthodologique strict.

2.2. Classification des approches de prototypage

Les différentes façons d'interpréter ou de décliner les trois types d'objectifs présentés dans la section précédente aboutissent bien évidemment à des visions très différentes du processus de prototypage. En insistant sur les types d'objectifs (1) et (2), le processus aboutit plutôt à une maquette jetable dont on ne conserve que les enseignements. En insistant sur des objectifs de type (3) le processus aboutit à l'obtention d'un produit opérationnel, utilisable directement.

En jouant sur les interprétations de ces types d'objectifs, **[133]** et **[14]** proposent la classification suivante : *jetable*, *incrémentale* et *par raffinements*.

L'approche jetable (aussi appelée prototypage par expérimentations) considère essentiellement des objectifs de type (1). Seul un sous-ensemble des fonctionnalités requises est implémenté, les informations concernant le cahier des charges devant être obtenues au plus faible coût. Ce type d'approche s'appuie sur des techniques comme :

- la simulation dans un environnement dédié comme PROTO **[46]**, DESIGN/CPN **[72]**, Cabernet **[231]** etc. Beaucoup de ces environnements de simulation permettent, outre la construction et l'animation d'un modèle, l'intégration de procédures externes, souvent utilisées pour obtenir des effets visuels réalistes dans des séquences de simulation (ce qui est le cas des trois outils mentionnés);
- la production d'un code capable de s'exécuter en dehors de l'environne-

ment qui l'a produit. Dans le cadre d'une approche jetable, de tels prototypes sont en général implémentés à l'aide d'outils spécialisés, comme HyperCard **[282]**, ou au moyen de langages disposant de larges bibliothèques de composants prédéfinis comme SmallTalk **[120]**.

L'approche jetable est désormais couramment utilisée y compris dans le domaine du matériel, où des standards (comme VHDL) ont été mis en place depuis longtemps. De nombreux environnements, y compris libre de droits (comme Alliance **[128]**, Université Paris VI ou Grape-II **[183]**, Université Catholique de Louvain), permettent d'évaluer la conception de composants matériels.

Dans l'approche incrémentale, de nouveaux modules sont successivement ajoutés à un «noyau» **[140]**. Ainsi, l'architecture initiale du prototype est conservée tout au long de la phase de développement, ce qui suppose qu'elle est pertinente. Si tel n'est pas le cas, la qualité du prototype ainsi obtenu en souffrira. On est ici confronté à un problème de type maintenance évolutive, ce qui est normal car cette démarche s'apparente à une vision «éclairée» du processus de développement (ce qui la rend applicable en l'absence d'outils sophistiqués).

La méthodologie OOMP (Object Oriented Mixed Prototyping) **[144]** est une bonne illustration de l'approche incrémentale. Le système est construit par implémentation successive de classes remplaçant au fur et à mesure une version générée automatiquement. On trouve ainsi dans un prototype des classes générées automatiquement (qui simulent un comportement donné) et une implémentation «manuelle» élaborée par le concepteur du système **[43]**. L'intégration dans une application d'une classe correctement implémentée ne doit pas engendrer de problème. Pour garantir cette caractéristique, une méthode et un outil de génération automatique de tests ont été mis au point **[229]**.

À l'inverse du prototypage incrémental, l'approche par raffinements (aussi appelée approche par évolution) tente de préserver la flexibilité du système et de ses fonctionnalités. Le prototype est enrichi au fur et à mesure d'études opérées sur des versions successives d'un couple <*spécification+prototype*> **[281]**. La démarche s'apparente plutôt à une vision du développement orienté sur la maintenance. Il est alors intéressant de bien archiver les différentes versions du système et les choix motivant le passage de l'une à l'autre.

2.3. L'importance du modèle

L'aspect développement ne correspond qu'à une partie du cycle de vie d'un logiciel. Les opérations de maintenance et d'évolution sont connues pour être extrêmement coûteuses... dans un contexte où la rapidité d'évolution des systèmes nous interdit de les ignorer.

Les méthodes de spécification modernes, qu'elles soient semi-formelles (OMT **[245]**, UML **[246]**, etc.) ou formelles (réseaux de Petri **[154]**, Spécifications algébriques **[109]**, B **[3]**, etc.) sont toutes basées sur la notion de *modèle*. Le modèle est une représentation abstraite du système exprimée au moyen d'un formalisme.

Pour adapter le prototypage et prendre en compte la dimension de maintenance, deux aspects doivent être considérés :

a) assurer l'absence de dérive entre la spécification d'un système et son implémentation, afin que ce qui est exprimé corresponde bien à ce qui est mis en œuvre. On retombe ici sur le type d'objectifs (3);

b) effectuer la maintenance, non sur le système lui-même, mais sur sa spécification. Cela permet d'appréhender plus facilement les difficultés introduites par des modifications en tenant compte de l'existant.

Pour atteindre le point (a), on peut considérer que le système est spécifié directement dans un langage de programmation ou au moyen d'un modèle disposant des mêmes constructions (comme dans Proteus **[98]** ou TRAPPER **[178]**), ce qui correspond alors à une vision «éclairée» du développement. Cette solution, si elle permet de décrire le système avec force détails, présente l'inconvénient majeur de contourner le problème réel de la spécification d'une solution. Si une telle spécification est programmée, alors c'est du développement et il n'y a pas de différence entre le modèle et le prototype.

On peut également considérer une approche reposant sur la génération de code à partir d'un formalisme de spécification approprié. Cette notion est intéressante car elle distingue les deux éléments importants : le modèle qui explicite le comportement du système et son implémentation qui les met en œuvre.

Le point (b) est important car il permet de maîtriser les évolutions d'un système complexe : lorsqu'on ajoute une nouvelle fonction, on dispose de tous les éléments pour vérifier qu'elle s'intègre correctement dans l'existant. Ici aussi, la génération de code est intéressante parce qu'elle permet de distinguer le modèle sur lequel de telles études seront faites, du prototype qui en est déduit. Si le modèle doit être distinct du système tout en exprimant correctement ses propriétés, il permet de comprendre le fonctionnement du système et de le valider. Il est ensuite à l'origine de la mise en œuvre, l'opération de codage étant alors vue comme un enrichissement du modèle.

Certains outils de développement (comme Rational/Rose **[240]**, Dom **[261]**, TeamWork **[53]**, Objecteering **[262]**, etc.) supportent ce genre d'approche. Les développeurs spécifient leur modèle selon un formalisme donné pour lequel l'outil peut générer un squelette d'application. Ces outils permettent de surcharger le modèle avec du code dans le langage cible de génération. Cette information ne fait pas partie du modèle en tant que tel, mais est exploitée pour produire le système complet (il est inséré dans le squelette de l'application). Ainsi, la gestion du code intégré au squelette de programme est directement assurée par l'outil au niveau du modèle.

Ainsi, à terme, on peut envisager que de plus en plus d'applications puissent être maintenues au niveau de leur modèle, ce qui est le cas pour l'outil S.A.O. **[52]** évoqué en Section 2.1. Si, de plus, le formalisme utilisé supporte la vérification, toute opération de maintenance peut être évaluée dans un premier temps

au niveau du modèle. Par exemple, on peut vérifier que les modifications n'entraînent pas de changement des propriétés de la version antérieure.

2.4. Le prototypage vu comme une méthodologie de conception/ réalisation

Les trois formes de prototypage (jetable, incrémental et par raffinement) ne considèrent pas le modèle de la même manière. Une explication est que les différents types de prototypages s'intègrent différemment dans le cycle de vie du logiciel (Figure 1).

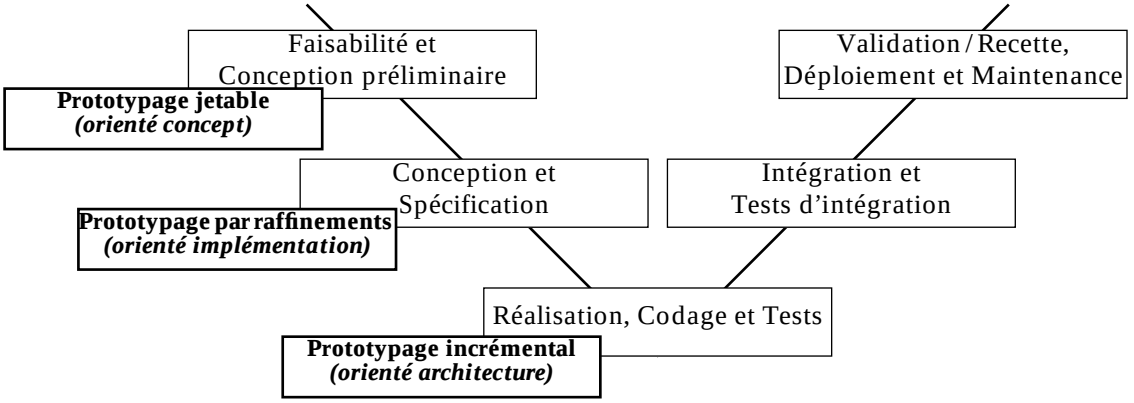

Figure 1 : Visions du prototypage et cycle de vie du logiciel.

L'approche jetable s'en désintéresse puisque le prototype n'est intéressant que par les informations qu'il permet de récolter pour le cahier des charges. On se place ici dans le cadre du prototypage de concepts. La vision incrémentale ne considère pas le modèle comme une entité distincte du prototype. Le prototypage est ici orienté architecture, ce qui explique que l'on peut l'assimiler à une vision éclairée du développement. L'approche par raffinement introduit une distinction entre le modèle et le prototype, car on se focalise sur l'aspect implémentation du système. C'est pourquoi la mise en œuvre du prototypage par raffinement implique l'existence d'outils sophistiqués, en particulier d'un générateur de code réalisant rapidement la transformation du modèle vers un programme dans un langage cible **[24]**. Cette condition rend réaliste un processus de développement itératif basé sur le modèle.

Ces trois visions du prototypage gagnent à être intégrées. On doit alors considérer le prototypage non comme une technique mais comme une méthodologie de développement qui comprend des opérations comme :

- la mise au point du cahier des charges (grâce à des maquettes),
- la conception préliminaire du système au moyen d'un modèle exprimé dans un formalisme adéquat,
- la mise au point du modèle,
- la génération automatique d'un prototype,
- l'évaluation du prototype.

La démarche, illustrée par la Figure 2, devient un cycle de vie par prototypage itératif **[194]**. Sur la base d'un cahier des charges, l'opération de modélisation permet d'obtenir un modèle formel du système. Une telle opération n'est pas automatisable, car c'est à ce niveau qu'un certain nombre de choix sont effectués par les concepteurs du système. Cependant, une approche de type gabarit de conception (*design patterns*) peut faciliter le travail de modélisation **[104, 222]** .

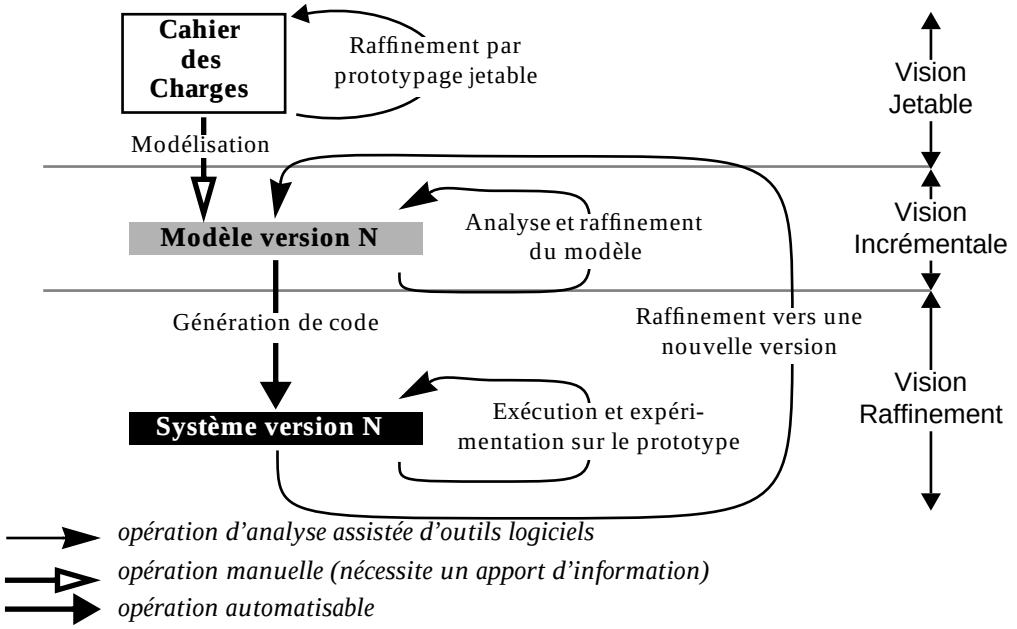

Figure 2 : Le prototypage en tant qu'approche de conception / développement.

Dès le modèle construit, les premières évaluations ont lieu pour vérifier que la solution modélisée est correcte et accroître la connaissance des concepteurs sur le système à réaliser. Elles sont menées par simulation ou, si le formalisme le permet, par analyse. Plusieurs versions du modèle peuvent être construite jusqu'à ce que les concepteurs soient satisfaits de sa fiabilité.

La génération de code produit automatiquement un prototype, c'est-à-dire une application conforme au modèle. Ce prototype permet de nouvelles expérimentations afin de mettre en évidence des caractéristiques attendues de manière plus empirique. À ce stade, le prototype a un avantage sur le modèle puisqu'il *est* lui-même l'application. Les résultats d'exécution ainsi obtenus n'en ont que plus de valeur : on peut ainsi vérifier certaines hypothèses faites pendant la simulation et / ou la validation.

Enfin, sur la base des résultats expérimentaux obtenus sur le prototype, on peut inférer une nouvelle version du modèle, plus riche, plus précise et plus réaliste. Le processus peut ainsi reboucler comme dans le prototypage par raffinements. Dans une telle approche, la traçabilité entre les différentes versions du système fournit des informations importantes sur les choix effectués et peut être assurée par un gestionnaire de versions.

2.5. Prototypage de systèmes répartis

L'approche méthodologique définie dans la section précédente facilite la conception et la réalisation de systèmes répartis, que leur importante complexité rendent délicats à mettre en œuvre. À ce titre, le choix du formalisme permettant de construire le modèle du système est très important. Les réseaux de Petri sont adéquats car : 1) ils proposent des concepts adaptées à la modélisation de ce type de systèmes 2) ils reposent sur des concepts mathématiques permettant le calcul de propriétés structurelles et comportementales.

Cependant, nous avons souligné en Section 1.3. le problème que soulevaient les réseaux de Petri quant à leur manipulation. À ce titre, l'approche que nous prônons dans notre travail repose sur des encapsulations de réseaux de Petri : H-COSTAM et OF-Class **[77]**.

Je présenterai dans le Chapitre 2 l'approche méthodologique que j'ai contribué à développer au sein de l'équipe Systèmes Répartis et Coopératifs pour mettre en œuvre une démarche de prototypage par raffinements.

3. Besoins en génération d'environnements

J'ai intégré le générateur de code développé pendant ma thèse dans l'environnement CPN-AMI1 **[198]**, reposant sur la plate-forme logicielle AMI **[23]**. Ce travail et les objectifs du projet IRENA ont également orienté mon travail de recherche vers la mise en œuvre de techniques visant à accélérer la production d'environnements de Génie Logiciel. L'objectif est d'utiliser ce savoir-faire pour produire un Atelier de Génie Logiciel mettant en œuvre la méthode de prototypage esquissée dans la section précédente. Dans le cadre d'un développement coopératif, une telle plate-forme simplifie la tâche des développeurs d'outils : ils peuvent exploiter des mécanismes communs accroissant l'interopérabilité des différents composants logiciels impliqués.

3.1. Des Ateliers aux Environnements de Génie Logiciel

L'objectif d'un Atelier de Génie Logiciel (AGL) est d'aider les ingénieurs de développement en supportant les aspects fastidieux des méthodes de développement **[252]**. Très vite, la complexité des méthodes aidant, un marché est apparu dans le domaine des AGL **[185]**. Cependant, ces outils dédiés ont longtemps souffert d'un manque d'adaptabilité. Une société utilisant une variante d'une méthode donnée pouvait difficilement utiliser l'AGL ne prévoyant pas cette variante. De même, le choix des outils de mise en œuvre n'était que rarement paramétrable : substituer à un compilateur piloté par l'AGL, un compilateur similaire mais d'une autre origine était difficile.

En réponse à cela, des propositions d'architectures et de normes sont apparues, comme ATIS **[34]** et ECMA-NIST **[88]** puis, plus récemment, CORBA **[208, 220]**, ODP **[218]** et TINA **[1]**. Les premières concernent les AGL. Les suivantes sont liées à d'autres besoins spécifiques : CORBA et ODP visent des applications

objets réparties, TINA se focalise sur les architectures de services de télécommunications (vente par correspondance, facturation etc.).

Les Ateliers de Génie Logiciel sont ainsi devenus des Environnements de Génie Logiciel (EGL) plus facilement adaptables : il est possible d'y insérer de nouveaux outils, de remplacer des composants logiciels par d'autres équivalents et d'adapter une méthode de développement. Cette adaptabilité permet le *prototypage d'Environnements de Génie logiciel*. Une telle approche facilite l'évaluation à moindre coût de l'efficacité et de la pertinence d'un outil dans des conditions réelles d'utilisation.

L'apport des normes est de séparer les éléments d'un EGL en deux parties : la *plate-forme d'accueil* et les *outils*. La plate-forme d'accueil offre des services communs (visualisation, stockage des données, communication...) et les outils se réduisent à des composants logiciels «enfichés» sur la plate-forme. Cette architecture, connue sous le sobriquet de «toaster», n'a pas cessé d'être raffinée depuis la parution de la première version du modèle de référence ECMA-NIST en 1990.

La prise en charge d'une nouvelle méthode par un Environnement de Génie Logiciel nécessite l'association de formalismes (mode de représentation, éventuellement graphique et / ou hiérarchique) et de services (fonctions offertes par un ou plusieurs outils). Dans les deux cas, l'intégration peut être faite *a priori* (développement spécifique pour l'environnement cible, ce qui implique la réalisation d'interfaces programmatiques pour faciliter cette tâche) ou *a posteriori* (récupération d'un logiciel développé pour un autre environnement, ce qui implique la mise en place de mécanismes permettant cette réutilisation à des conditions raisonnables).

Cette approche est également en vogue dans le domaine des systèmes d'exploitation. MacOS[(5)] **[272]** ou Windows-NT **[86]** dans le domaine de la Bureautique, peuvent désormais être considérés comme des plates-formes d'accueil, les services étant fournis par des applications utilisateur (éditeur de texte, tableur...). Les systèmes à base de micro-noyaux (Chorus **[71]**, MACH **[259]**...) peuvent également être vus comme des plates-formes d'accueil encapsulant l'architecture matérielle d'une machine.

Le développement dans de tels environnements est considérablement différent d'un développement classique. Des règles sont à respecter, des bibliothèques de fonctions sont à utiliser... En fait, il ne s'agit plus de construire un produit mais de le construire de manière *adaptée* à l'environnement cible. L'objectif est de faire en sorte qu'un logiciel fonctionnant sur une plate-forme d'accueil donnée puisse «facilement» être porté sur une autre plate-forme. On parle de portage mais aussi d'*intégration*.

(5) A ce titre, on peut considérer que MacOS, sur bien des aspects, est un précurseur dès 1984.

3.2. L'opération d'intégration

L'opération d'intégration consiste en l'adjonction d'une nouvelle fonctionnalité déjà opérationnelle dans un environnement pour lequel elle n'a pas forcément été spécifiquement développée. L'apparition du modèle de référence ECMA-NIST **[88]**, puis de CORBA **[220]**, a considérablement structuré cette opération, qui reste cependant une tâche délicate, tant les méthodes sont variées et les plates-formes d'accueil (système d'exploitation, langages...) différents. Cinq «axes» d'intégration ont été identifiés dans **[283]** :

- *présentation* (qui intègre à la fois les aspects visualisation mais aussi comportementaux des outils);
- *données* (le stockage mais aussi les techniques d'accès en vue de «comprendre» les informations stockées);
- *contrôle* (le pilotage d'outils);
- *procédés* (la définition de processus de traitement pour les méthodes);
- *plate-forme* (la plate-forme est vue ici comme une encapsulation des fonctions du système d'exploitation).

3.3. Plate-forme de prototypage d'environnements

Dès 1987, notre équipe s'était intéressée à la notion de plate-forme d'accueil pour factoriser le travail des développeurs d'outils sur les réseaux de Petri. En 1989, un premier prototype était opérationnel : AMI **[23]** sur lequel nous avons construit CPN-AMI1 (AMI pour les réseaux de Petri), diffusé sur Internet à partir de 1993 et utilisée par de nombreuses équipes universitaires.

En étudiant l'implémentation de la méthode MARS (ébauchée en Section 2. et détaillée dans le Chapitre 2), nous avons rapidement identifié le besoin d'une plate-forme d'accueil permettant la réalisation rapide d'Environnements de Génie Logiciel. Mais les mécanismes d'intégration d'AMI n'étant pas assez souples, sa configuration restait laborieuse. Ainsi, après avoir travaillé plusieurs années en collaboration avec K. Foughali, X. Bonnaire et J.L. Mounier sur la plate-forme AMI, j'ai été à l'origine du projet FrameKit **[169, 172]** .

FrameKit est la seconde génération de notre plate-forme d'accueil et tire les leçons de la longue expérience acquise avec AMI (tant dans son développement que dans sa diffusion) dont il est la suite logique. Nous avons élaboré de manière systématique des mécanismes permettant d'en faire une plate-forme d'accueil dédiée au prototypage d'environnement de Génie Logiciel, rendu possible par le faible coût d'intégration des formalismes et des outils.

FrameKit respecte la norme ECMA-NIST tout en l'interprétant, à la fois en fonction de nos besoins mais aussi pour permettre une implémentation simple et facilement portable. Notre plate-forme a été implémentée de manière à respecter les objectifs suivants :

- l'intégration de nouveaux formalismes,
- l'intégration de nouveaux services,

- des facilités d'administration d'entités comme les utilisateurs, les groupes d'utilisateurs, l'accessibilité aux services...
- un modèle de distribution type, permettant la diffusion d'éléments de la plate-forme, soit sous la forme de kits indépendants, soit sous la forme d'un paquetage. Un kit est une unité de diffusion de la plate-forme (correspondant à un ou plusieurs services); un paquetage regroupe plusieurs kits décrivant un ensemble cohérent de formalismes et les services qui leur sont associés.

Le Chapitre 3 détaille les choix effectués dans FrameKit et le résultat des premières expérimentations que nous avons faites. FrameKit a été utilisé pour construire la version 2 de l'environnement CPN-AMI **[199]**, disponible sur Internet depuis Mars 1997.

4. Synthèse

Dans ce chapitre, je me suis appliqué à brosser le chemin parcouru depuis ma thèse d'Université. Sur la base des expérimentations effectuées avec l'outil CPN / Tagada, implémentation de mon travail de thèse, j'ai observé les limites de la génération de code à partir d'un formalisme comme les réseaux de Petri. Cela m'a amené à reconsidérer ma vision du prototypage : la génération de code, qui y avait une importance prépondérante, s'est réduite à une simple opération technique au profit de l'activité de modélisation (description du système).

Par ailleurs, les travaux de mise en œuvre des résultats de ma thèse m'ont amené à m'intéresser aux environnements de Génie logiciel. Cela m'a conduit à envisager une autre forme de prototypage dédiée à la réalisation rapide (pour évaluation et plus si affinités) d'un environnement de Génie Logiciel complet qui est l'implémentation d'une méthodologie reposant sur divers formalismes.

Chapitre 2
Méthodes Formelles et Prototypage d'Applications Réparties

5. Introduction

Ce chapitre définit et expérimente une méthodologie de conception d'applications par prototypage. Le domaine d'application choisi est celui des systèmes répartis.

Dans ces travaux, j'essaye de concilier plusieurs objectifs :

- offrir des formalismes adéquats, manipulables par des ingénieurs et encapsulant des méthodes formelles,
- exploiter les avantages des méthodes formelles (ici, les réseaux de Petri),
- faciliter la mise en œuvre du système à partir de son modèle, notamment à travers la génération de code,
- prendre en compte des composants déjà existant et faciliter leur réutilisation.

Pour exploiter les possibilités offertes par les méthodes formelles tout en contournant les difficultés liées à leur manipulation, j'expose et justifie l'approche d'encapsulation adoptée. La méthode MARS, ses différents formalismes et ses étapes sont ensuite présentés. La mise en œuvre de MARS dans l'environnement CPN-AMI2 est enfin discutée et comparée à des environnements existants.

6. Encapsulation des réseaux de Petri

La modélisation revêt une importance capitale dans une approche de prototypage (Section 2.4.). Cette section montre comment la modélisation s'intègre au processus de prototypage. L'encapsulation d'une représentation formelle (ici, les réseaux de Petri) facilite le travail du concepteur d'un système et préserve les possibilités de validation.

6.1. Processus de modélisation

L'opération de modélisation permet d'obtenir et de structurer des informations sur un système (existant ou en cours de développement). Pour cela, on procède en plusieurs étapes résumées dans la Figure 3.

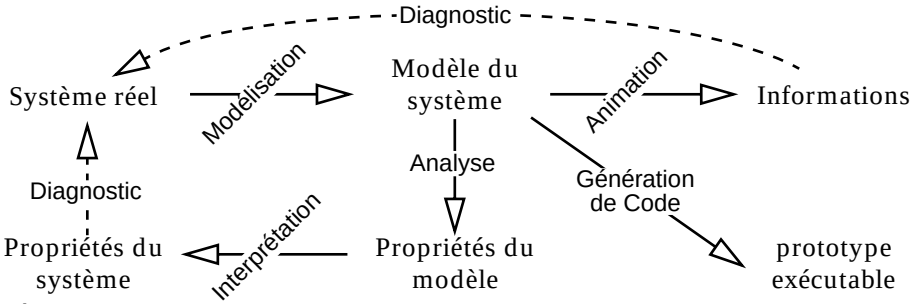

Figure 3 : Étapes du processus de modélisation.

Dans un premier temps, on décrit un modèle du système réel sur lequel on travaille (Modélisation). Lorsque le formalisme utilisé le permet (Estelle **[81]**, SA-RT **[134]**, etc.), on obtient des traces d'exécution (Animation), même s'il est souvent impossible de parcourir tout l'espace d'exécution du modèle. Les méthodes formelles permettent de calculer des propriétés de ce modèle (Analyse) pour en déduire des propriétés du système (Interprétation). Que les informations obtenues soient informelles (chemin Modélisation-Animation) ou formelles (chemin Modélisation-Analyse-Interprétation), il faudra, afin de pouvoir les exploiter, identifier leur incidence au niveau du système (Diagnostic).

Dans l'approche de prototypage décrite en Section 2.4., ce procédé revêt une importance particulière puisque le modèle est l'image d'une solution à un cahier des charges. Cette image est ensuite utilisée pour produire automatiquement l'application par génération de code.

6.2. Expérimentation avec les réseaux de Petri

Pour expérimenter et tester le générateur de code développé à la suite de ma thèse, j'ai développé des applications au niveau du réseau de Petri **[165]**. Les applications réalisées devaient être intégrées, après validation puis génération de code, dans un environnement d'exécution complexe (utilisation d'un démon supportant des actions graphiques sous X-OpenWin, exécution dans la plate-forme AMI évoquée en Section 1.2.2.). Ces études (menées suivant le schéma de la Figure 3) ont permis de valider la pertinence de l'approche basée sur les composants externes (évoquée en Section 1.2.2.) et d'aboutir aux observations suivantes :

a) L'utilisation d'une représentation formelle pour décrire une application n'est pas aisée car la contrepartie des concepts mathématiques permettant la vérification est l'absence de mécanismes de haut niveau (files, piles, notion d'instanciation propre au paradigme objet, etc.). Modéliser de tels mécanismes complique le modèle, ce qui pose alors les problèmes d'efficacité du générateur de code et de lisibilité de la spécification évoqués en Section 1.3.;

b) Il existe une contradiction entre les objectifs de validation et ceux de la génération de code. Lorsque l'on modélise un système en vue de le valider, on élague les informations inutiles par rapport à ce que l'on souhaite vali-

der tandis que pour générer du code, on en produit une description détaillée. Par exemple, pour valider un système du point de vue de ses communications, on aura tendance à ne modéliser les traitements que dans la mesure où ils affectent les communications afin de réduire au maximum un «bruit» pouvant nuire à l'analyse, ce qui est inacceptable du point de vue de la génération de code;

c) L'association entre propriétés attendues pour le système et les propriétés calculées sur le modèle n'est pas évidente car la signification des propriétés dépend de la manière dont le modèle à été construit. C'est un point où intervient le savoir faire du modélisateur.

Le problème posé par le point (b) est lié à la complexité des modèles lors de leur analyse. Plusieurs approches permettent de réduire cette complexité :

- Des techniques de réduction avec conservation des propriétés de vivacité ont été proposées dans **[25]** pour les réseaux de Petri Place / Transition, puis étendues dans **[131, 132]** pour les réseaux de Petri colorés. Ces techniques reposent sur des règles de transformation du modèle (fusion de places, agglomération de transitions...). Comme ces règles de transformation ignorent les aspects sémantiques du modèle, il est impossible d'analyser automatiquement les résultats produits sur des réseaux réduits;
- Des techniques de décoloration de réseaux de Petri utilisent les propriétés structurelles des réseaux de Petri afin de réduire les domaines de couleurs lorsqu'ils engendrent des états inutiles car équivalents à d'autres d'un point de vue vérification **[58]**. Hélas, comme dans le cas des réductions, l'information la disparition de couleurs ayant une sémantique de modélisation complique également l'analyse des résultats obtenus sur la spécification réduite.

Ainsi, l'utilisation directe des réseaux de Petri pose un problème. Il faudrait disposer de plusieurs modèles : un par aspect du système à valider (interopérabilité des composants du système, communication, terminaison, équité...) et une description détaillée pour la génération de code. La notion de «points de vues», identifiée dans ODP **[151]**, conforte ce type d'observation. Dans ODP, un système complexe est exprimé selon cinq vues (entreprise, information, traitement, ingénierie et technologie). Ces points de vues servent de base à la prise en compte de l'architecture du système complet **[267]**.

6.3. Vers une encapsulation des réseaux de Petri

L'extension des réseaux de Petri, comme l'association avec un formalisme structurant (tous deux évoqués en Section 1.3.), ne constituent pas des solutions aux trois problèmes évoqués dans la section précédente. L'extension des réseaux de Petri suppose l'abandon de propriétés que l'on ne sait pas encore calculer. La simulation devient alors la seule forme de validation possible tant que la théorie n'a pas intégré ces extensions.

L'association avec un formalisme structurant permet de gérer différents points de vues. Le problème réside dans la manipulation de plusieurs modèles. On peut imaginer que des liens étroits soient maintenus entre le modèle «structurant» et le modèle formel correspondant, pour permettre le passage automatique d'un niveau à l'autre, mais cela revient à encapsuler les réseaux de Petri.

L'encapsulation permet de cacher complètement les mécanismes sous-jacents. Le concepteur d'un système dispose d'un formalisme de haut niveau lui proposant des constructions types, exprimant une sémantique précise et permettant une description détaillée du système. Cette approche comporte des avantages :

- Pour la modélisation : les constructions de haut niveau augmentent la lisibilité le modèle. Il est par conséquent plus facile de traiter des systèmes complexes;
- Pour la génération de code : les problèmes liés à la reconnaissance de mécanismes sophistiqués simplement utilisables via l'environnement d'exécution disparaissent et les programmes produits s'en trouvent allégés. Les deux problèmes identifiés en Section 1.3. (performance du code généré et résistance de certains réseaux de Petri à un partitionnement) seraient ainsi résolus.

L'approche présentée en Figure 3 évolue alors vers celle de la Figure 4. Le système réel est modélisé à l'aide d'une représentation de haut niveau (1) qui peut être animée (2). Cette animation est intéressante pendant les premières phases de mise au point.

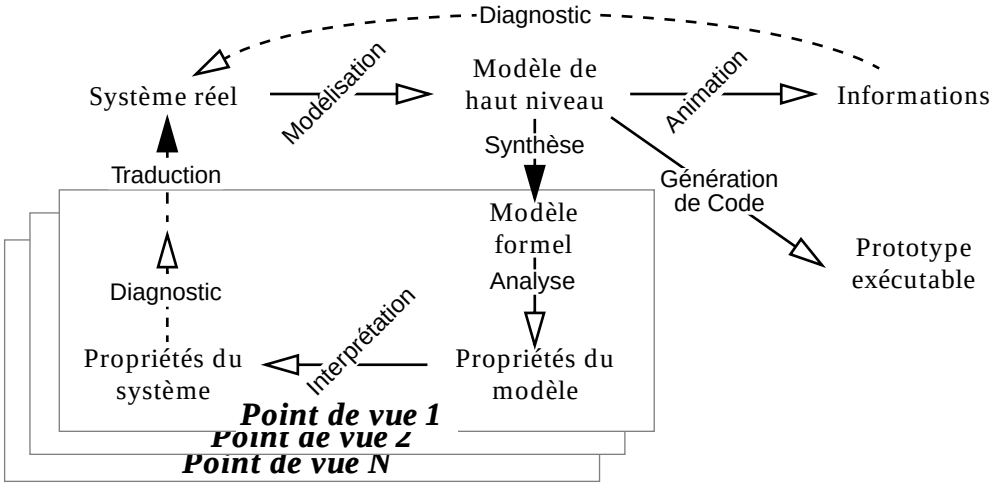

Figure 4 : Processus de modélisation avec encapsulation des réseaux de Petri.

Pour effectuer une validation formelle, on passe par la synthèse de réseaux de Petri. Plusieurs techniques sont envisageables en fonction des points de vue proposés. À chaque synthèse correspond un ensemble de propriétés pertinentes à valider (chemin Analyse-Interprétation), puis à traduire (chemin Diagnostic-Traduction). La traduction est un point délicat car des propriétés exprimées en fonc-

tion de réseaux de Petri doivent être remontées à l'utilisateur selon le formalisme qu'il manipule.

Cette approche, bien que complexe, constitue une solution aux problèmes (a), (b) et (c) évoqués en Section 6.2. Cependant, dans le cadre du prototypage, on doit aboutir à un modèle opérationnel (ou exécutable). Un tel niveau de description s'obtient par raffinements successifs, résultat de l'intégration progressive d'informations et de choix de conception, puis de réalisation.

6.4. Description conceptuelles et opérationnelles

Lorsque l'on conçoit un système, on est confronté à des problèmes conceptuels (choix d'une architecture pour l'application, choix de la répartition des services et définition des interactions entre les différents composants du logiciel, etc.) et à des problèmes opérationnels (mise en œuvre les fonctions du logiciel, choix de réalisation, etc.). La résolution de ces problèmes nécessite des informations de type distinct et difficile à mélanger. Les informations d'un niveau opérationnel ne s'accommodent pas des approximations nécessaires pour faciliter le travail de vérification (le parti pris du modélisateur). De même, certaines informations liées pour la vérification doivent être ignorées pour la génération de code. Ce point est bien intégré dans les méthodes objets comme OMT **[245]**, UML **[246]** ou Classe-relation **[75]** qui proposent différentes vues d'un système (par exemple, on modélise l'architecture avant de s'intéresser aux automates associés à chaque classe).

Ainsi, lors de l'élaboration d'un système par prototypage, on s'intéresse dans un premier temps aux aspects conceptuels avant d'intégrer des choix d'ordre opérationnel. Il est donc souhaitable de différencier dans le temps la phase de description conceptuelle du système, de la phase de description opérationnelle. De la sorte, nous proposons deux encapsulations des réseaux de Petri : un formalisme conceptuel, dédié à des aspects validation et un formalisme opérationnel ayant pour objectif la génération de code. Pour assurer une bonne traçabilité entre les niveaux conceptuels et opérationnels, ces formalismes reposent sur des principes communs qui seront exposés plus loin.

La Figure 5 illustre l'approche de développement par prototypage intégrant ces deux niveaux de description semi-formels. À partir du cahier des charges, un modèle conceptuel est construit (Modélisation). La présence d'un simulateur / débogueur est intéressante en phase de mise au point du modèle (Animation-Diagnostic).

La Synthèse automatique d'un modèle formel et le diagnostic sur les propriétés identifiées (Synthèse$_v$-Diagnostic) permet de vérifier des propriétés conceptuelles comme : «un service offert par un module est-il correctement utilisé ?», «l'exclusion mutuelle de telle ressource est-elle correctement assurée ?», «Combien le système a-t-il d'états de blocage ?» ou «tous les états de blocage ont-ils une signification identifiée par le concepteur ?».

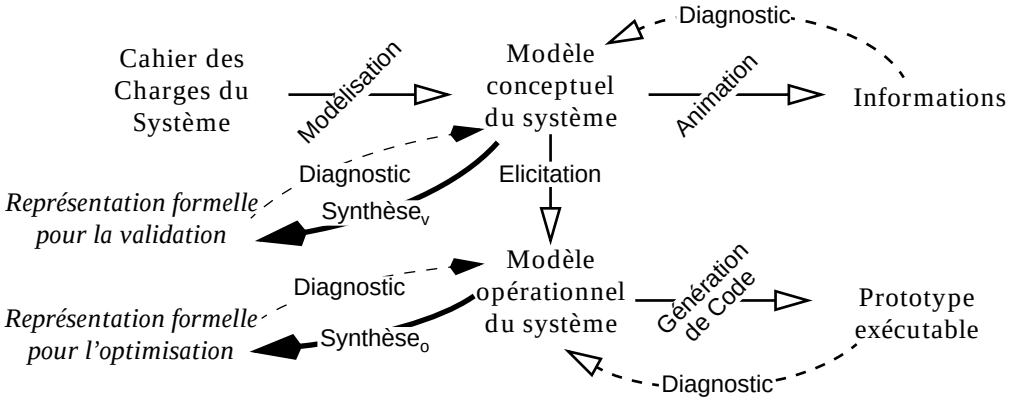

Figure 5 : Liens entre le niveau conceptuel et le niveau opérationnel dans une approche de développement par prototypage.

Une fois le modèle conceptuel stabilisé, le modèle opérationnel est construit à partir du modèle conceptuel. Cette opération d'*Elicitation* **[77]** nécessite un apport d'information de la part du concepteur (les choix qu'il effectue) et n'est donc pas complètement automatisable comme pour le passage du formalisme conceptuel vers le modèle formel. L'Elicitation d'une spécification peut survenir plusieurs fois dans la vie du logiciel : pendant sa réalisation, puis lors des évolutions de maintenance.

La synthèse d'une représentation formelle à partir du modèle opérationnel et le diagnostic sur les propriétés identifiées (Synthèse$_o$-Diagnostic) a pour objectif l'optimisation et permet de répondre à des questions comme : «est-il possible de répliquer sans risque une ressource ou un serveur selon un critère proposé par le concepteur ?» ou «si on réplique une ressource ou un serveur, quel serait le nombre optimal de répliques ?». De telles questions, exprimées par le concepteur du système sont intéressantes à vérifier car les réponses permettent de fournir au générateur de code les directives permettant d'exploiter la structure de l'application sans nuire à la sécurité de son exécution. Les programmes produits en seront d'autant plus efficaces.

Une telle démarche a l'avantage de composer le développement en étapes : le concepteur d'une application se pose les bonnes questions au bon moment. Ce type de démarche en deux temps est mise en œuvre dans l'environnement Proteus **[121, 122]** présenté en Section 8.2.3.

7. La méthode MARS

La méthodologie MARS **[96, 76]** s'inspire directement du schéma directeur exposé en Figure 5 et s'appuie sur trois formalismes (Figure 6) :

- Conceptuel : OF-Class (**O**bject **F**ormalism **Class**) **[76]**;
- Opérationnel : H-COSTAM (**H**ierarchical **CO**mmunicating **ST**ate m**A**chine **M**odel) **[168]**;
- Formel : Les réseaux de Petri bien formés **[56]**.

OF-Class et H-COSTAM intègrent des concepts objet tout en restant liés aux contraintes des réseaux de Petri. En effet, l'intégration de caractéristiques trop sophistiquées (par exemple, tout ce qui est lié à l'héritage) rendrait impossible l'utilisation de classes de réseaux de Petri sur lesquels le calcul de propriétés formelles est possible.

MARS a pour point d'entrée, soit le cahier des charges lui-même (le modèle conceptuel est directement utilisé comme modèle de spécification), soit la description d'un système avec un formalisme objet de type OMT ou UML. Trois types d'opérations reliant ces formalismes sont caractérisés :

1) Deux séries de Synthèses (Synthèse$_v$ et Synthèse$_o$) permettent d'obtenir des réseaux de Petri. Chaque transformation est dédiée à l'étude de propriétés précises sur le modèle formel et intègre le point de vue destiné à en faciliter l'évaluation;
2) L'Elicitation du système produit semi-automatiquement un modèle opérationnel à partir du modèle conceptuel;
3) La génération de code produit automatiquement des programmes exécutables.

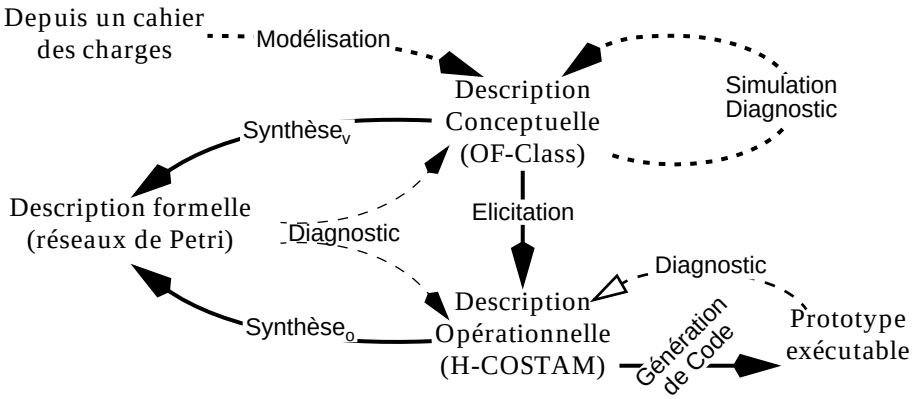

Figure 6 : Les formalismes et leurs relations dans la méthode MARS.

Le raffinement des descriptions est possible. Au niveau conceptuel, il s'appuie sur les résultats d'analyse (chemin Synthèse$_v$-Diagnostic) ou, le cas échéant, sur des simulations (chemin Simulation / Diagnostic). Au niveau opérationnel, le modèle peut être modifié en fonction de propriétés d'optimisation (chemin Synthèse$_o$-Diagnostic) ou l'analyse des résultats d'exécution du prototype (chemin Génération de code-Diagnostic).

7.1. Les formalismes de haut niveau

OF-Class et H-COSTAM présentent deux points communs pour faciliter l'élicitation unifier la démarche méthodologique :

- une organisation hiérarchique commune,
- une classification commune pour différencier les éléments de spécification appartenant au système de ceux de l'environnement d'exécution.

L'organisation hiérarchique d'OF-Class et H-COSTAM repose sur une classification des informations concernant des «modules» de spécification. On distingue de la sorte le niveau *macro* qui contient des informations relatives aux relations entre unités de modélisation et le niveau *micro* qui décrit le comportement d'une unité de modélisation. Ici, le terme «unité de modélisation» est à prendre au même sens qu'unité de programme quand on traite de compilation séparée. Une unité de modélisation correspond à une composante (potentiellement réutilisable) du système

Pour différencier le système de son environnement d'exécution, on sépare les unités de modélisation *interne* (celles qui décrivent un morceau du système) des unités de modélisation *externes* (celles qui décrivent un élément de l'environnement d'exécution). Toute unité de modélisation doit être décrite au niveau macro; la description micro est facultative pour les unités externes.

7.1.1. OF-Class

OF-Class a été définit par A. Diagne dans le cadre de sa thèse d'Université, soutenue en Juin 1997 **[78]**. Ses travaux s'appuient sur son expérience dans le projet européen IRENA (O-Formalisme Spoke) **[147]** et sur les travaux de H. Bachatène **[17]**. Ma contribution a OF-Class réside dans la structuration des différents composants de la spécification sur la base des principes décrits un an plus tôt avec H-COSTAM.

OF-Class est un formalisme dédié à la description conceptuelle de systèmes répartis. Il intègre les besoins identifiés dans le modèle de référence ODP (**O**pen **D**istributed **P**rocessing) **[151]** et propose des mécanismes pour décrire l'interfaçage des unités de modélisation. Il est associé aux réseaux de Petri colorés modulaires **[76]** qui constituent l'automate interne décrivant une unité de modélisation OF-Class et servent de support pour la vérification.

Structure d'un modèle OF-Class

Les interactions entre composantes d'un système réparti constituent un point délicat de leur mise en œuvre. C'est pourquoi OF-Class offre aux concepteurs un gabarit l'aidant à spécifier correctement la manière dont les unités de modélisation qu'il conçoit interagissent entre elles. Pour cela, il identifie des propriétés locales ou globales.

Les propriétés globales s'appliquent à un ensemble d'unités de modélisation et sont exprimées au moyen d'assertions sur l'évolution du système. Elles caractérisent des informations comme les états importants devant être atteints (par exemple, l'état de terminaison propre) ou des conditions de sécurité sur l'interopérabilité des unités de modélisation (absence d'interblocage ou de famine).

Les propriétés locales s'appliquent à une unité de modélisation. Elles peuvent être exprimées sous la forme d'invariants sur des ressources (comme dans RM-ODP) ou sous la forme de contraintes sur la disponibilité des services. Ces propriétés sont énoncées au niveau micro.

Niveau macro dans OF-Class

Le niveau macro dans OF-Class décrit les liens d'une unité de modélisation OF-Class avec son environnement extérieur (les autres unités avec lesquelles elle coopère). Deux types de liens sont caractérisés : les liens structurels composent des unités de modélisation et les liens dynamiques définissent les attentes sur les services liant ces entités entre elles :

- des *services offerts*, exportés par l'unité, sont composés d'un ensemble d'opérations sur lesquelles sont définies des contraintes contractuelles (précédence ou sémantique d'utilisation). C'est une vue cohérente et partielle du comportement de l'unité vis-à-vis de son environnement extérieur. L'ensemble des services offerts d'une unité constitue son gabarit d'utilisation;
- des *services requis*, importés par l'unité, doivent être offerts par son environnement d'exécution pour qu'elle soit opérationnelle. Leur description est similaire à celle des services offerts mais ils correspondent aux prérequis que l'unité fait sur la manière dont se comporte son environnement d'exécution.

Les composants sont connectés entre eux via les services exportés / requis. Ainsi, une unité A offre un service requis par une unité B. La sémantique des interactions du (ou des) service(s) liant A et B doit être cohérente : les prérequis exprimés au niveau de B correspondent à la réalité offerte par A.

Un service décrit un mécanisme de coopération et le protocole qui lui est associé. Les opérations d'un service identifient les communications nécessaires pour mettre en œuvre le service. Elles peuvent avoir les comportements suivants :

- Le *rendez-vous* impose une synchronisation forte entre les composants,
- La *RPC*[(1)] *synchrone* **[284]** correspond à une requête bloquante pour le client,
- La *RPC asynchrone* définit une requête qui n'est pas forcément bloquante pour le client.

Niveau micro dans OF-Class

Le «contenu» d'une unité de modélisation OF-Class est décrit au niveau micro. On y trouve la description de toutes les entités qui la composent, c'est-à-dire :

- Les *ressources* qui ne sont accessibles qu'à travers les opérations des services offerts. Les ressources peuvent être répliquées (chaque instance de l'unité possède sa propre copie) ou partagée (toutes les instances de l'unité accèdent à une copie unique);
- Les *traitements* associés aux opérations. Leur description est ici opérationnelle et non contractuelle comme au niveau macro. Deux opérations spéciales et non exportables permettent de gérer l'initialisation et la terminaison des instances de l'unité;

(1) Remote procedure Call.

- Les *triggers*, sont des traitements déclenchés en fonction de conditions sur les ressources de l'unité.

Les opérations et les triggers peuvent invoquer des services requis par l'unité à son environnement d'exécution (les autres unités). Chaque instance d'une unité OF-Class n'exécute qu'une seule opération à la fois mais, si un ou plusieurs triggers sont activables, leur exécution s'effectue en parallèle avec cette opération.

Exemple de modèle OF-Class

La Figure 7 présente la modélisation en OF-Class d'un composant de gestion de fichiers.

```
SGF isa OFCLASS
declaration {
    types {
        type_status is enumeration{OK, PB}
        type_booleen is enumeration {vrai, faux} ;
    }
}
macro-level
    ## Services importés du gestionnaire de disque
    imports {
    from Disque service Access {
        operation
            void : write (char : C in )
            invocation-mode synch
            default-return do continue
            void : read (char : C out )
            invocation-mode synch
            default-return do continue
    }
    ## Services exportés par l'unité
    exports {
    service File_Access
        operations { type_status : create (int : desc in_out)
                     type_status : open (int : desc in_out)
                     type_status : read (int : desc in; char : Buf in)
                     type_status : write (int : desc in; char : Buf out)
                     type_status : close (int : desc in)}
        ## Le manuel d'utilisation du service
        manual File_Access is { || {
                                  { open ; R is { read } } ;
                                  { create ; W is { write } } ;
                                  close }
                              } }
        invocation-mode { synch }
    }
}
micro-level
    resources {
    ## La clause default permet d'initialiser le composant a sa creation
        type_booleen opened_read default faux duplicated;
        type_booleen opened_write default faux duplicated;
    }
    instances {}
    operations {
        type_status : create (int : desc in_out) {
            #creation dynamique d'une instance
            constructor (desc);
            oself.opened_write := 'vrai';
            return 'OK';}
        ...
        type_status : close (int : desc in) { {
            if (oself.opened_write = 'faux') && (oself.opened_read = 'faux') then
                # le descripteur est déjà fermé
                return 'PB';
            else
                oself.opened_read := 'faux';
                oself.opened_write := 'faux';
                return 'OK';
            end if;}
        }
ENDOFCLASS
```

Définition d'un service importé

Définition d'un service exporté + gabarit d'utilisation

«contenu» d'un service

Figure 7 : Exemple de modèle OF-Class.

Le niveau macro définit les services de l'environnement attendus par le composant (lecture et écriture de caractères sans relation de précédence particulière

entre les deux opérations du service) et le service offert (manipulation d'un fichier en lecture ou en écriture).

Ici, le composant suppose que l'environnement offre deux opérations sur des caractères : write et read. Le service qu'il offre est composé de cinq opérations (`create`, `open`, `read`, `write`, `close`) et permet la manipulation d'un fichier en écriture (séquence d'invocation : `create {write} close`) ou en lecture (séquence d'invocation : `open {read} close`).

Au niveau micro, sont déclarées deux variables dont chaque instance d'unité aura sa propre copie (les variables sont «duplicated»). La définition des services create et close est également donnée (noter l'accès aux variables dupliquées pointées par «oself» qui désigne l'instance du composant en cours d'exécution).

7.1.2. H-COSTAM

H-COSTAM (Hierarchical COmmunicating STate mAchine Model) a été conçu, en collaboration avec W. El Kaim, pour la conception d'applications par prototypage (Section 1.3.). H-COSTAM se veut un formalisme adapté à la description opérationnelle de systèmes répartis. Ses caractéristiques principales sont :

- La hiérarchie pour exprimer des spécifications de grande taille lisibles;
- Un modèle de communication au sens de CSP **[141]**, mis en œuvre au moyen de mécanismes fortement typés à la fois par les données véhiculées et par leur comportement;
- La généricité(2) pour paramétrer des unités et faciliter leur réutilisation. Chaque instantiation d'une page générique correspond à une nouvelle copie particularisée de la hiérarchie d'origine.

Structure d'un modèle H-COSTAM

Une spécification H-COSTAM est composée de *pages* de type macro ou micro. Une page macro décrit les relations entre des entités qui sont soit des sous-systèmes (une référence à une autre page macro), soit des processus séquentiels (une référence à une page micro). Sous-systèmes et Processus séquentiels sont reliés entre eux via des media de communication. Un modèle H-COSTAM est représenté par un graphe orienté dont les nœuds correspondent aux pages. La racine est une page macro et les feuilles soit des pages macro marquées externes (qui décrivent un élément de l'environnement d'exécution de l'application), soit une page micro décrivant une machine à état du système. Une telle représentation ressemble, dans la gestion de la hiérarchie, aux graphes de processus tels qu'ils sont représentés dans des environnements TRAPPER **[178]** ou PARSE **[226, 123]** .

Un système de typage permet d'identifier de manière unique les informations manipulées dans un modèle H-COSTAM. La propagation des types se fait selon les règles suivantes :

1) Tout type défini est visible dans la page où il est défini,
2) Toute page référencée peut accéder aux types définis dans la page qui la

(2) Au sens Ada **[5]** du terme.

référence (l'englobe),

3) La généricité permet de déroger à la règle (2) car elle permet de redéfinir la liste des types importés depuis l'unité englobante.

Le niveau macro dans H-COSTAM

Comme dans OF-Class, le niveau macro permet de décrire la structure d'un système. Une page macro est à un sous-système non séquentiel possédant ses propres interfaces. Les composantes d'un tel sous-système peuvent être des sous-systèmes ou des processus séquentiels.

Les liens entre composantes d'une page macro sont exprimés au moyen de *media* de communication. On dénombre quatre types de media :

- Les *multi-rendez-vous* correspondent à une synchronisation entre N composantes. Cette synchronisation peut être gardée par un prédicat et permet éventuellement aux participants d'échanger des messages. Le traitement associé à un multi-rendez-vous est exécuté une fois à chaque synchronisation pour le compte de tous les clients impliqués;
- Les *liens* correspondent à un échange asynchrone de données régi par un comportement type : FIFO (l'ordre des messages transmis est préservé), LIFO (l'ordre des messages est inversé) ou aléatoire (les messages sont désynchronisés). Une capacité en nombre de messages peut être associée aux liens de communication;
- Les *RPC* (Remote Procedure Call) relient deux composants entre eux, l'un agissant comme un serveur et l'autre comme un client. Côté client, les RPC sont vues comme une action atomique (rendez-vous). Côté serveur, elle correspondent à deux FIFO, la première transmettant des paramètres en entrée, la seconde les résultats de la requête ainsi transmise;
- Les *constructeurs* (factories) véhiculent des messages particuliers provoquant la création de nouvelles instances d'un ou plusieurs processus **[104]**. Un message transmis à travers un constructeur est reçu par tous les destinataires reliés en sortie.

La communication entre un module (i.e. la page le décrivant) et son monde extérieur est exprimée au moyen de media. Ces objets représentent donc soit une entité de communication (connectée aux interfaces des composants) soit les interfaces de communication de la page (c'est-à-dire les représentants de media contenus dans des pages englobantes).

Le niveau micro dans H-COSTAM

Une page micro décrit un composant élémentaire : le processus. Comme les sous-systèmes (représentés par des pages macro), les processus sont reliés au monde extérieur via des interfaces définies par des media de communication.

Un processus représente un modèle de comportement instancié statiquement (instances définies dans la déclaration) ou dynamiquement (par le biais de mes-

sages provenant de constructeurs interfaces). Le contexte d'un processus est composé de variables dont chacune des instances possède une copie.

Une page micro contient un automate état-transition à la sémantique proche de celle d'un réseau de Petri. Cet automate, composé d'actions et d'états, décrit une machine à états (au sens de **[130]**) correspondant au comportement statique du processus. Les actions peuvent être gardées par des prédicats portant sur des variables du contexte d'exécution et/ou de messages provenant des media. Les actions peuvent également produire des messages en direction de media d'interface ou modifier le contenu de variables de contexte au moyen d'un ensemble d'opérateurs (identité, successeur et prédécesseur circulaire[3] d'ordre N, produit...). Le nombre de ces opérations est volontairement limité en fonction des contraintes définies dans les réseaux de Petri bien formés **[56]** (si des fonctions plus complexes sont nécessaires, elles peuvent être mise en œuvre au moyen de composants externes).

En interface d'entrée, les constructeurs sont connectés à l'état initial dans lequel doit être créée la nouvelle instance. La réception d'un message correspond à la création d'une nouvelle instance dont le contexte est issu du contenu du message. En interface de sortie, les constructeurs sont connectées aux actions qui, seules, peuvent générer des messages.

Exemple de modèle H-COSTAM

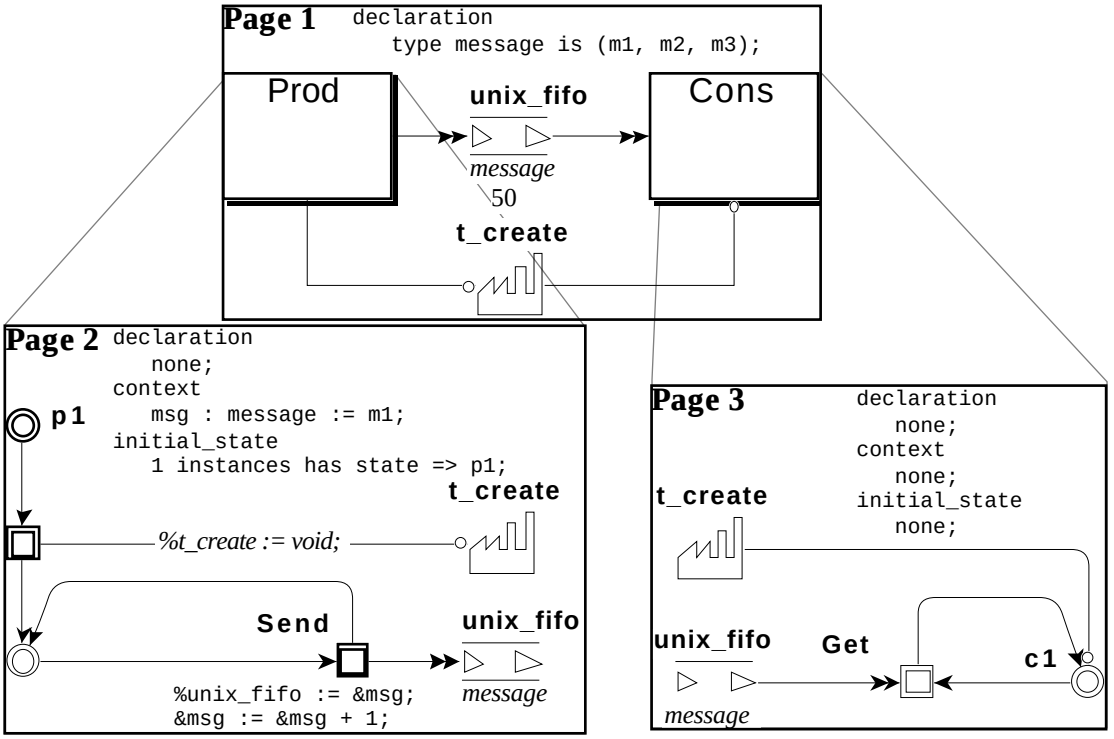

Figure 8 : Exemple de modèle H-COSTAM.

(3) le successeur de la dernière valeur est la première valeur. Cette contrainte est imposée par les réseaux de Petri bien formés **[56]** .

Un exemple simple de modèle est présenté en Figure 8 afin d'en illustrer les principaux aspects (un autre exemple, plus complexe, est traité en page 55). Ce modèle comporte trois pages : La page macro (page 1) décrit un sous-système composé de deux processus connectés au moyen de deux media : un constructeur et un lien FIFO.

La description micro du processus *Prod* (page 2) définit son comportement : il n'existe qu'une seule instance, créée statiquement, émettant un message dans le constructeur *t_create* puis bouclant indéfiniment pour envoyer des messages dans le lien *unix_fifo*. Le processus *Cons* (page 3) n'a initialement aucune instance (elle sera créée dès réception du message provenant de *Cons*) et consomme les messages émis par *Prod*.

7.1.3. Synthèse des caractéristiques d'OF-Class et H-COSTAM

OF-Class et H-COSTAM sont deux encapsulations de réseaux de Petri basées sur le paradigme objet. Ces formalismes différant principalement sur la manière dont sont décrites les interactions entre composants. D'une manière générale, OF-Class permet de vérifier la cohérence de l'assemblage de composants tandis que H-COSTAM est dédié à la génération de code.

Le Tableau 1 résume les différences et les similitudes entre les formalismes OF-Class et H-COSTAM.

Caractéristiques	OF-Class	H-COSTAM
Hiérarchie	Par assemblage de composants (approche plutôt bottom up)	Par composition de composants (approche plutôt top down)
Interface des composants	Description du gabarit des services. Différentiation entre services offerts et services requis (attentes sur l'environnement d'exécution).	Description au moyen d'un modèle de communication basé sur des mécanismes élémentaires fortement typés (les media).
Structure interne des composants élémentaires	Définition des ressources locales et des opérations permettant de les manipuler. Définition de triggers, traitements activés en fonction de conditions sur les ressources.	Description d'une machine à états séquentielle dotée d'un contexte d'exécution propre et instanciée statiquement ou dynamiquement.
Instanciation dynamique de composants	Par le biais de services précis. Le contrôle est défini au niveau micro d'une unité de modélisation OF-Class.	Par l'intermédiaire de messages spéciaux véhiculés par les constructeurs.
Communication entre les composants	Par le biais de services offerts / requis. Les services sont composés d'opérations unitaires auxquelles sont associées un gabarit d'utilisation.	Par l'intermédiaire de media de communication typés.
Types de communication supportés	RPC synchrones et asynchrones, Rendez-vous binaire.	Multi-rendez-vous, liens FIFO, LIFO ou Random et RPC.
Approche de réutilisation	Privilégiée par la forme de composition choisie (approche plutôt bottom-up). Réutilisation partielle au moyen de raffinements ou de simplifications	Assurée par la structuration hiérarchique associée à la généricité, qu'il s'agisse d'un sous-système ou d'un processus séquentiel.

Tableau 1: Synthèse des différences et similitudes entre OF-Class et H-COSTAM

7.2. Synthèse de réseaux de Petri

Les transformations ont pour objectif de produire une spécification formelle à partir d'une description de haut niveau à des fins d'analyse. Bien sûr, les transformations à partir d'OF-Class ou de H-COSTAM ne peuvent pas aboutir à un résultat similaire : elle n'ont pas le même objectif et les informations dont on dispose ne sont pas les mêmes, puisque c'est à ce niveau qu'intervient la notion de point de vue dans la modélisation. Plusieurs transformations doivent être mises en œuvre. Chacune exploite des informations distinctes (sur le modèle de haut niveau) adaptées à l'évaluation d'une propriété.

7.2.1. Principes des synthèses

Bien que toutes différentes, les synthèses de réseaux de Petri obéissent à des règles communes. Leurs différences résident essentiellement dans la manière d'exploiter les informations contenues dans le modèle source. En fonction des résultats recherchés, on privilégie ou on ignore telle ou telle information.

Le processus de synthèse d'un réseau de Petri se déroule en deux phases :

1) Le niveau micro sert de base pour la construction des modules élémentaires destinés à être assemblés. Pour cela, on s'inspire de la méthode proposée dans **[138]**. Des automates sont déduits des traitements et des domaines de couleurs déduits des types définis. Les variables locales deviennent des valuations d'arcs et les variables partagées sont stockées dans des places. Les ports de communications (opérations dans OF-Class, media dans H-COSTAM) sont laissés pendants en vue de l'assemblage du module;
2) La structure du système, contenue au niveau macro, est alors utilisée pour assembler les différents modules obtenus lors de la phase précédente. Des gabarits paramétrés sont associés à chaque mécanisme de communication défini dans le formalisme source (opérateurs dans OF-Class, media dans H-COSTAM). Une fois particularisés (en fonction de paramètres comme le type de message véhiculé, ou le comportement défini par l'utilisateur), ces gabarits constituent des sous-réseaux connectés aux modules précédemment synthétisés. Le modèle de communication permet d'assembler les sous-réseaux en s'appuyant sur des techniques reconnues, que les interactions soient synchrones **[25]** ou asynchrones **[264]**.

Les réseaux de Petri synthétisés de la sorte sont «plats» et ne sont pas destinés à être observés par le concepteur d'une application. Pour faciliter l'analyse en réduisant l'explosion combinatoire souvent liée à des réseaux de Petri de grande taille, on considère que :

- La synthèse selon un point de vue permet de réduire la taille des gabarits dans les cas ou l'information qu'ils représentent n'est pas pertinente. L'information ignorée prend place sous une forme réduite (par exemple, regrouper un ensemble de transitions neutres vis-à-vis de la propriété recherchée ou réduire à une place les mécanismes de communication asynchrone dont le comportement n'affecte pas la propriété que l'on cherche à

valider).

- Les techniques de réduction proposée dans **[25]** (réseaux place / transition) et **[131, 132]** (réseaux colorés) peuvent être appliquées une fois le réseau de Petri synthétisé.

Dans le cas du modèle conceptuel, certaines propriétés peuvent être vérifiées sur des réseaux de Petri modulaires, synthétisés à partir d'informations parcellaires. Par exemple, il est possible de générer un réseau de Petri à partir du gabarit d'un service. La comparaison du gabarit d'un service offert par rapport au gabarit des hypothèses faites dans l'unité où il est importé est une opération intéressante qui ne nécessite qu'un travail sur deux modèles de taille réduite : il faut vérifier que gabarit attendu par l'unité qui importe le service «est inclus» dans celui du service tel qu'il est offert. L'inclusion est ici celle de deux grammaires. Par exemple, la grammaire du service de manipulation d'un fichier en lecture seulement doit être inclue dans celle correspondant à la manipulation d'un fichier en lecture et écriture.

7.2.2. Exploitation des résultats

Le résultat d'une opération d'analyse sur le réseau de Petri doit être exprimé à celui qui spécifie le système dans les termes du formalisme source. Pour cela, il faut maîtriser les transformations et maintenir la correspondance entre entités du modèle de haut niveau et celles du modèle formel.

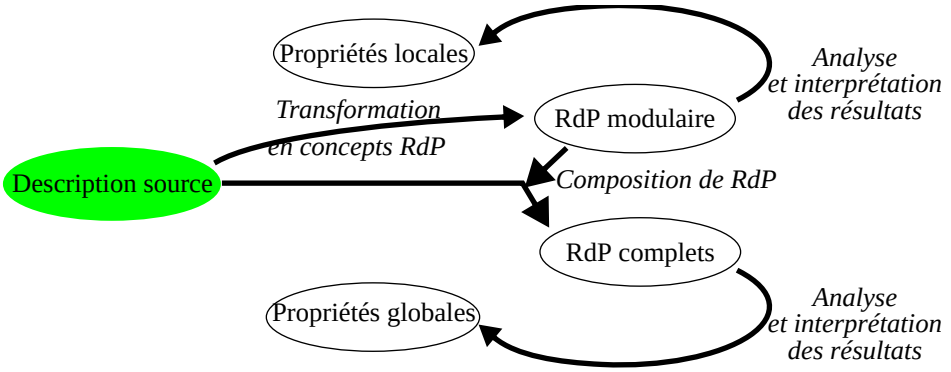

Figure 9 : Le cycle d'analyse dans la méthodologie MARS.

La Figure 9 illustre le processus d'analyse dans la méthodologie MARS. Les résultats peuvent être calculés soit sur des réseaux de Petri modulaires, soit sur les modèles complets. Dans le premier cas, les informations propriétés sont locales à un seul composant, dans le second cas, elles sont globales et concernent le système.

Analyse sur le modèle conceptuel

L'analyse des réseaux de Petri produits à partir d'OF-Class porte actuellement sur la recherche de propriétés de sûreté et de fiabilité **[80]**. Elle s'effectue en deux temps :

- Pour chaque module obtenu dans la phase 1 de la transformation, un

Chaos-Free Failure Divergences Model **[277]** (CFDM) est calculé. Ce modèle est une version réduite du graphe d'accessibilité d'un composant ne comprenant que les actions observables (i.e. interagissant avec l'environnement), les actions divergentes (introduisant des boucles infinies), les états de blocage et les erreurs d'interaction. Le CFDM est obtenu en intégrant les hypothèses des modules important les services de l'unité étudiée (services requis). Un tel modèle donne un point de vue intéressant sur la qualité d'un composant;

- L'exploitation des CFDM permet d'obtenir des réseaux de Petri réduit ne comprenant que les actions observables. Ces modèles constituent une signature du composant et peuvent être liées à un autre composant en vue d'une validation globale.

L'utilisation d'un model checker permet également d'appliquer des formules de logique temporelle sur le graphe d'accessibilité du modèle (des expérimentations ont été effectuées avec Prod **[279]**). La définition de labels au niveau de la spécification OF-Class permet de repérer des états du système de les analyser. On peut par exemple vérifier des scenarii provenant du cahier des charges calculant les chemins qui séparent deux états dans le graphe des marquages accessibles.

Analyse sur le modèle opérationnel

Au niveau opérationnel, l'analyse des réseaux de Petri synthétisés a pour objectif principal l'optimisation du code généré. Une analyse locale peut être intéressante pour calculer des bornes afin de dimensionner des ressources locales mais l'analyse globale est plus intéressante. Elle permet :

- de dimensionner des ressources, par exemple d'identifier les cas où il est impossible de borner le nombre de messages stockés dans un lien de communication. Cela permet de sélectionner une stratégie d'implémentation.

  La recherche d'une telle information s'apparente à un calcul de borne sur un modèle pour lequel les liens de communication sont modélisés au moyen d'une place unique. Cette approximation permet d'identifier correctement les places pour lesquels le calcul nous intéresse. Lorsque le modèle produit est coloré, un dépliage (éventuellement limité à certaines classes de couleurs) permet d'affiner cette information en exploitant la sémantique des messages véhiculés.
- de rechercher des configurations de type pipe-line entre les différents processus du système. Cela est intéressant pour réaliser une bonne stratégie d'allocation des processus sur une architecture matérielle donnée.

  Cette information est fournie par certains invariants de places effectués sur le modèle structurel (voir Section 1.2.1.) d'un réseau de Petri pour lequel :
  - les communications synchrones et les constructeurs sont modélisés au moyen d'une place unique;
  - dans les automates des processus, les états qui ne sont pas liés à des

communications sont agglomérés.

Les invariants de places qui nous intéressent traversent plusieurs processus en identifiant les media de communication séparant les étages du pipeline. Un exemple de ce type d'analyse est présenté page 55.

- de rechercher comment répliquer (si cela est possible) des ressources pour augmenter le parallélisme du système et déduire des critères d'allocation des processus sur une architecture matérielle. Cette étude s'effectue sur un modèle pour lequel les liens de communication et les variables partagées sont modélisés au moyen d'une place seulement. On étudie la manière dont un tel modèle se déplie par rapport aux couleurs impliquées dans la communication (sémantique du message véhiculé). Un exemple de ce type d'analyse est présenté page 55.

7.3. L'Elicitation

Au contraire de la synthèse de réseaux de Petri, le passage d'une spécification conceptuelle à une spécification opérationnelle n'est pas une tâche complètement automatisable. Il ne s'agit pas uniquement d'interpréter une information présente dans le modèle source; il faut intégrer de nouvelles informations fournies par le concepteur du système. Les aspects suivants sont pris en compte :

- Il faut définir les équivalences entre les concepts d'OF-Class et ceux de H-COSTAM. Certaines équivalences, les mécanismes de communication, dépendent de choix effectués par l'utilisateur;
- La notion de trigger n'a pas d'équivalent dans H-COSTAM. Cependant, les mécanismes imaginés peuvent être exprimés en H-COSTAM;
- Les informations relatives aux gabarits d'utilisation doivent être écartés car leur utilisation concerne la vérification;
- L'environnement d'exécution (les composants externes) ne sont pas traités de la même manière que le système. Seul le gabarit d'utilisation décrit en OF-Class est traduit en H-COSTAM.

L'élicitation peut survenir plusieurs fois dans le cycle de vie du logiciel : une fois lors de la construction du système, puis lors de l'ajout de nouvelles fonctions. Il est donc intéressant d'avoir une procédure «accompagnée» qui pourrait être mise en œuvre au moyen d'un jeu de questions / réponses. Le Tableau 2 récapitule les principales règles utilisées pendant la phase d'élicitation en indiquant comment intervient l'utilisateur.

Une étude préliminaire sur l'implémentation de la phase d'élicitation a été effectuée par O. Sy pendant son stage de DEA **[271]**.

7.4. La génération de code

Dans le processus de prototypage, la génération de code joue un rôle central car elle est le lien entre le modèle et le système. Il s'agit d'une transformation dont le

	OF-Class	Équivalent H-COSTAM	degré d'automatisation
Niveau macro	Opération rendez-vous	Multi-rendez-vous (binaires)	oui
	Opération RPC synchrones	RPC	oui
	Opération RPC asynchrones	Deux liens, le premier pour l'appel, le second pour le résultat. Si l'usager souhaite que l'ordre des requêtes soit préservé, les liens seront des FIFO, dans le cas contraire, il s'agira de liens random. Le type des messages échangés est déduit des paramètres de l'opération.	semi[a]
	Gabarit d'utilisation	Écarté pour les unités faisant partie du système. Exploité pour construire l'automate micro des composants externes.	oui
Niveau micro	Automate interne	S'il s'agit d'un automate séquentiel, production d'un processus H-COSTAM. Dans le cas contraire, déduction d'un sous-système H-COSTAM composé de plusieurs processus.	semi[b]
	Triggers	Production d'un processus pour gérer les trigger et d'un processus par trigger. Le gestionnaire de trigger crée dynamiquement les instances qui exécuteront les trigger une fois qu'ils sont activés.	oui
	Services	Les services sont «dissous» dans l'automate (ou les automates) produits à partir de la micro description. On peut envisager une structuration en processus basée sur celle des services à des fins de lisibilité.	oui
	Variables, manipulation de variables et ressources	Transformation en variables de contexte ou en liens de communication (qui modélisent alors un stockage).	oui

Tableau 2: Quelques règles de traduction d'OF-Class vers H-COSTAM.

a. L'usager doit spécifier son attente concernant la transmission des messages (ordre d'émission maintenu ou non).
b. Le concepteur du système doit valider les choix de décomposition effectués. Le cas échéant, plusieurs possibilités lui seront proposées.

résultat peut être raffiné et optimisé en fonction des données issues d'exécutions ou des informations dont on dispose sur l'architecture cible.

Pour produire des programmes dans différents langages cibles, nous proposons une architecture générique basée sur les constructions de H-COSTAM permettant d'encapsuler proprement les interactions avec un environnement de communication ou de répartition. Ainsi, le concepteur d'un système peut choisir, en fonction de critères qui lui sont propres, d'utiliser un environnement particulier supporté par le générateur de code.

Le modèle H-COSTAM contient aussi les informations utiles pour le calcul du placement des entités logicielles en tenant compte des contraintes de l'architecture matérielle d'exécution (nombre et vitesse des processeurs, topologie du réseau, etc.). Pour effectuer un tel calcul, nous nous appuyons, soit sur l'exploitation de résultats théoriques de la Recherche Opérationnelle, soit sur l'évaluation de propriétés sur un réseau de Petri synthétisé depuis le modèle H-COSTAM.

7.4.1. Points clefs pour la génération de code

Pour générer une application répartie, il faut s'intéresser à deux aspects : les programmes en eux-mêmes et le placement des composantes (données et processus) de l'application sur une architecture cible. Certaines études traitent du problème de la répartition au moment de la génération de code **[40]**, mais changer l'alloca-

tion des tâches impose de regénérer l'application entière. Une telle approche n'est en général considérée que dans des cas particulier : une architecture multi-processeurs auto-configurable et / ou un langage extrêmement lié à une philosophie d'utilisation particulière. C'est le cas de **[40]** qui s'intéresse à la génération de programmes Occam sur une architecture transputers : lors de son initialisation, l'application construit l'architecture logique appropriée à son exécution sur le réseau physique de transputers.

L'utilisation croissante d'environnements médiateurs (middleware) pour la communication ou la répartition assure la transparence dans la manipulation des ressources système et permet la production de programmes portables. On distingue dans **[90, 91]** deux approches dans la répartition d'une application : l'*équilibrage de charge* et l'*équilibrage d'application*.

Equilibrage de charge

L'équilibrage de charge est effectué pendant l'exécution de l'application. Les tâches qui s'exécutent sont affectées à un processeur en fonction de critères de charge. Deux utilisations de l'équilibrage de charge sont proposées dans **[124]** :

- *L'équilibrage de charge dynamique* reposant sur la migration de composants logiciels pendant l'exécution du prototype en fonction de critères de charge des processeurs. On intègre un environnement de répartition de charge par édition de liens. Il est cependant difficile de le configurer finement en utilisant les informations dont on dispose sur l'architecture logicielle. Le problème vient du fait que l'équilibrage de charge dynamique exploite principalement les informations relatives à la charge des machines et gère moins bien celles relatives à l'architecture de l'application (en particulier lorsque celle-ci ne respecte pas de modèles d'exécution simples). Le modèle de processus communiquant que nous obtenons (les machines à états) n'est pas adapté aux langages de configuration proposés dans ce type d'environnement dont l'utilisation risque donc de se limiter à celle d'une boîte à outils de communication (ce qui est dommage).
- *L'équilibrage de charge statique* repose sur la définition d'une charge théorique associée à chaque processus de l'application afin de définir les processeurs qui seront utilisés pour l'exécution du prototype. Si l'on dispose de directives permettant (en fonction de critères déduits de l'architecture logicielle de l'application) d'associer des composantes logicielles à des processeurs virtuels et d'en calculer la charge théorique, la correspondance processeurs virtuels / processeurs réels tiendra compte de critères difficiles à exploiter dans l'approche précédente. Cependant, on perdra les avantages de la migration. Pour disposer d'un moyen d'évaluer la charge théorique associée à chaque processus du système, il faudrait spécifier au niveau d'H-COSTAM les informations relatives à l'exécution du système (obtenues après exécution du prototype).

Equilibrage d'application

L'équilibrage d'application exploite l'architecture logicielle du système qui constitue son atout majeur. Prenons l'exemple d'un système composé de N couples producteur / consommateur partageant un mécanisme de communication (Figure 10, à gauche). Le mécanisme de communication peut être répliqué sur plusieurs sites puisque les communications n'ont lieu qu'au sein d'un couple. La règle de partitionnement suivante peut ainsi être déduite : n_i couples producteur / consommateur + une copie du mécanisme de communication[(4)] sont affectés sur le site i (Figure 10, à droite).

Les systèmes d'équilibrage de charge ne peuvent en général exploiter de telles informations sur la sémantique d'une application. il faut également vérifier que de telles assertions n'engendrent que des exécutions correctes. Dans exemple de la Figure 10, si on divise un couple producteur consommateur (i.e. le producteur sur un site et le consommateur correspondant sur un autre), les messages émis ne seront jamais reçus par le consommateur mais resteront bloqués sur la copie du media localisée avec le producteur. Une telle exécution est incorrecte. Les réseaux de Petri permettent de vérifier que ce type de placement est correct.

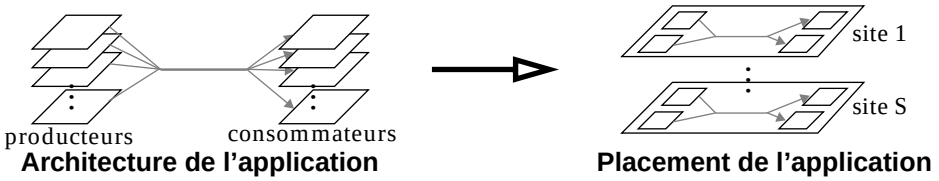

Figure 10 : Exemple d'architecture logicielle et d'un placement adéquat.

Citons trois études exploitant la structure d'une spécification en vue d'en optimiser l'exécution. Dans **[99]**, le système logiciel et son architecture matérielle sont décrits au moyen de réseaux de Petri : le modèle de l'architecture logicielle correspond à un P-Net et les dépendances entre ressources sont représentées au moyen d'un R-Net. La correspondance est manuellement effectuée au moyen d'un PRM-Net (**P**rogram **R**esource **M**apping). Une autre étude de ce type (basée sur d'autres formalismes) a été menée avec l'environnement de prototypage Parallel PROTO **[47]** : le placement (essentiellement manuel) de composants de la spécification d'un système logiciel sur une architecture matérielle permet d'obtenir des informations sur la charge induite des différents processeurs. L'approche présentée dans **[192]** privilégie le respect de contraintes temps réel. Pour cela, un échéancier est calculé sur la base des informations contenues dans la spécification (principalement son architecture et des directives de priorité).

La démarche élaborée en collaboration avec William El Kaïm peut exploiter les stratégies de répartition proposées. Pour cela, on différencie le code généré des directives de placement. Le processus de génération de code produit deux éléments distincts : un prototype auto-configurable et un fichier de configuration contenant des informations sur le placement des entités logicielles qui le compo-

(4) Avec $\sum_{i = machines} n_i = N$.

sent **[89]**. Deux opérations distinctes sont donc contenues dans la génération de code : la phase de génération (décrite en Section 7.4.3.) et la phase de placement (présentée et illustrée par un exemple en Section 7.4.6.). Cette dernière utilise, outre l'architecture logicielle du système, une description matérielle de l'architecture cible décrite au moyen du formalisme HADEL (Section 7.4.4.).

7.4.2. Le processus de génération de code

La production de programmes repose sur deux outils : un générateur de code et un outil de placement. L'exploitation des résultats d'exécution permet de raffiner le placement pour chaque architecture d'exécution. La dissociation entre le prototype et les données d'affectation des tâches le composant permet, pour une cible donnée (machine plus système d'exploitation), d'expérimenter plusieurs placements en fonction d'architectures matérielles (sous ensemble de machines utilisées) et/ou logicielles (affectation des machines à état sur les processeurs) sans recompilation.

Le processus de génération de code est composé des étapes illustrées par la Figure 11. Le prototype est généré à partir de l'architecture logicielle du système exprimée en H-COSTAM. Le placement est calculé à partir de l'architecture logicielle du système et de l'architecture matérielle cible d'exécution **[203]**. L'exécution du prototype produira des traces qui, après évaluation automatique ou manuelle, pourront être exploitées par l'outil de placement. Cette stratégie est similaire à celle proposée dans l'environnement de développement Proteus **[204, 12]**.

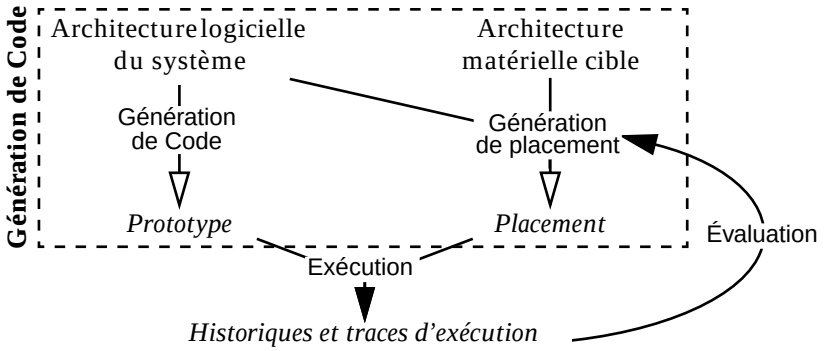

Figure 11 : Processus de Génération de code.

Cette vision intègre de manière raisonnable les différentes stratégies de répartition :

- L'équilibrage d'application est ici directement applicable. Différents placements peuvent être générés pour différentes architectures cibles d'exécution. L'intérêt est de prendre en compte les caractéristiques de l'architecture logicielles en fonction des contraintes matérielles auxquelles on est soumis.
- L'équilibrage de charge statique peut être vu comme une variation de l'équilibrage d'application. Pour cela, le fichier généré défini des directives de placement sous la forme de prérequis (nombre de sites virtuels souhai-

tés et placement sur ces sites). Une association est effectuée en début d'exécution sur la base de ces informations afin d'associer une machine à chaque processeur virtuel.

- L'équilibrage de charge dynamique réalise au démarrage de l'application la même correspondance site virtuel / site réel qu'en équilibrage de charge statique. Ensuite, l'évolution dynamique du placement passe par le respect de contraintes propres à l'application (par exemple, déplacement groupé de processus fortement couplés). Ces contraintes sont déduites de l'architecture et assurées par l'environnement d'équilibrage de charge.

Pour mettre en œuvre l'équilibrage d'application, il suffit de disposer d'un environnement de communication comme MPI **[209]**, PVM **[111, 112]** , GLADE **[4]** ou NMS **[103]** (des expérimentations avec NMS ont été menées sur une version étendue de l'outil CPN / Tagada générant du code Ada réparti). En revanche, la mise en œuvre de l'équilibrage de charge repose sur l'utilisation d'un environnement de répartition comme LSF (version commerciale d'Utopia) **[286, 234]** , Condor **[70, 188]** ou GATOS **[101]** en plus de l'environnement de communication(5).

Cette approche de la génération de code se place résolument dans le cadre d'une démarche de prototypage par raffinement. Elle permet d'expérimenter ces différentes approches en fonction des besoins et de la capacité des environnements de communication ou de répartition choisis.

7.4.3. La phase de Génération

La notion d'architecture générique d'un prototype **[168]** (voir Section 1.2.1.) se prête parfaitement, à la génération de code depuis H-COSTAM. Cette architecture constitue un schéma d'implémentation pour des générateurs de code, le seul prérequis exigé étant que les langages cibles soient structurés (plutôt de type impératifs ou objets) et permettent la manipulation de tâches, que ce soit intrinsèquement (comme dans Ada ou Java) ou via le système d'exploitation (par exemple, C ou C++ sous Unix).

La Figure 12 présente l'architecture générique d'un prototype généré à partir d'un modèle H-COSTAM. Le prototype s'appuie sur un exécutif comprenant des services de communication, d'exécution à distance et des mécanismes d'évaluation de charge et de migration (si on souhaite réaliser l'équilibrage de charge). Un tel exécutif doit être couplé avec des environnements existant.

Une bonne encapsulation des services de l'exécutif devrait permettre au concepteur d'un système de choisir le sien en fonction de critères identifiés. Par exemple, une comparaison entre MPI et PVM **[113]** identifie que le premier est plus performant pour de grands systèmes multi-processeurs alors que PVM semble plus avantageux dans le cas d'environnements hétérogènes.

(5) Les environnement de répartition offrent d'ailleurs des services de communication.

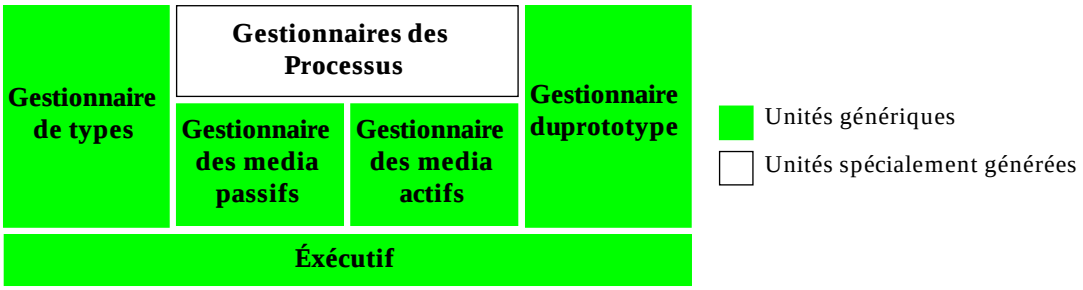

Figure 12 : Architecture générique d'un prototype.

Un gestionnaire de types supporte la définition des types déclarés dans le modèle et leur associe les opérations de manipulation prédéfinies dans H-COSTAM. Isoler la définition des types permet de faciliter l'évolution du générateur de code en fonction des extensions de H-COSTAM.

Le gestionnaire de prototype supporte tout ce qui a trait à l'initialisation et la terminaison du système. C'est lui qui utilise les informations de placement du fichier de configuration pour construire la topologie de l'application en s'appuyant sur les services de l'exécutif. Il supervise également la création des instances des processus (qu'elle soit statique ou dynamique).

Un gestionnaire de processus sera généré pour chaque processus élémentaire du modèle H-COSTAM. Ces gestionnaires permettent la manipulation (création, migration, destruction) d'instances du processus qu'ils implémentent sur requête du gestionnaire du Prototype.

Deux types de gestionnaires sont associés aux media selon qu'ils soient actifs (multi-rendez-vous, RPC) ou passifs (liens). Les media actifs sont gérés par des serveurs alors que les media passifs correspondent à des bibliothèques de fonctions manipulant des données.

La mise en œuvre des constructeurs (factory) est répartie entre le gestionnaire du prototype et les gestionnaires de processus. Le gestionnaire du prototype est notifié de chaque création dynamique de tâches, ce qui lui permet de connaître l'état courant du système (c'est utile pour détecter la fin de l'application et procéder à l'arrêt des différents gestionnaires). Il propage le message au gestionnaire de processus concerné qui effectue la création.

Afin de pouvoir répartir facilement différentes composantes du système sous le contrôle des gestionnaires de prototype, le mécanisme de *tâche transparente à la répartition* (TTR) est utilisé. De ce mécanisme, développé en collaboration avec P. Sens et présenté dans **[162]**, on ne retiendra que le fait que toute tâche contient, en plus de son comportement réel, un comportement de mandataire (proxy). La communication entre tâches s'effectue alors de manière identique, qu'elles soient locales ou distantes, ce qui est le cas lors de l'expérimentation de plusieurs placements successifs sans recompilation. Les différentes tâches du prototype sont alors placées sur un site d'exécution au démarrage du système en

fonction des informations présentes dans le fichier de placement. Cette stratégie a été expérimentée avec Ada **[36]** et avec C **[83]** sous Unix.

Le mécanisme de TTR autorise également une forme simplifiée de migration pour les tâches issues de machines à états. Leur contexte se réduit à un ensemble de variables locales pouvant être véhiculé par messages. Il suffit de demander à une telle tâche son contexte, de le transmettre au site d'accueil, puis de la commuter en mode mandataire.

Certains modules de cette architecture sont génériques : ils sont quasiment indépendants (à quelques structures de données près) du modèle H-COSTAM source. Bien sûr, plusieurs stratégies d'implémentations peuvent être envisagées (par exemple, réparti ou centralisé). Le concepteur d'un système choisit une stratégie au lancement du générateur de code.

7.4.4. Description d'une architecture matérielle

La phase de placement produit un fichier de configuration décrivant la localisation des entités d'un prototype (tâches et media de communication) sur une architecture cible. Nous avons donc besoin d'un formalisme permettant de décrire une architecture matérielle.

Les systèmes multi-processeurs actuels sont concurrencés par des réseaux de stations de travail ou y sont connectés **[10]**. On parle de réseaux hybrides mais aussi de machines multi-processeurs faiblement ou fortement couplés en fonction du débit des liens de communications **[50]**. La description de ces réseaux hybrides doit prendre en compte deux aspects : le grain fin (processeurs fortement couplés, i.e. machines multi-processeurs) et le gros grain (processeurs faiblement couplés, i.e. mono-processeurs ou multi-processeurs vus comme une entité unique). Les différences entre ces deux niveaux sont liées d'une part aux caractéristiques des machines et de leurs liens de communication, d'autre part à l'apparition d'informations nouvelles (par exemple, au niveau fin, la manière dont la mémoire est organisée).

Un certain nombre d'études se focalisent sur la gestion de la configuration (et reconfiguration) du système pour des environnements comme CONIC **[196]** ou pour les langages de programmation parallèles comme Argus **[189]**, Emerald **[155]** et DURRA **[19]**. Le problème majeur de ces études est que la description matérielle du système est difficilement dissociable du langage de programmation.

La description introduite dans Parallel PROTO **[47]** est volontairement simplifiée afin de ne prendre en compte que les besoins précis propres aux types d'applications traitées (les systèmes d'informations). Elle est indépendante d'une langage de programmation et repose sur l'association de mémoires à des processeurs via des bus. Ce modèle est dédié à la description d'une architecture gros grain, la mémoire étant ici vue comme une zone de stockage pour des bases d'informations.

De même, SySl (**Sy**stem **S**tructure **l**anguage) intègre les aspects matériel d'un système en intégrant **[263]**. Le paradigme objet y est utilisé pour décrire les classes d'objets matériels et leurs propriétés. Par exemple, la classe «machine» possède des attributs décrivant ses caractéristiques techniques (mémoire, processeur etc.). Chaque instance de machine présente dans une architecture opérationnelle dispose donc de caractéristiques propres définies d'après un modèle.

L'environnement Proteus **[122]** propose un modèle générique **[186]** permettant d'exprimer les caractéristiques d'une architecture de type super calculateur (Cray, MP-1, CM5, etc.). Ces informations sont utilisées par un simulateur et un allocateur pour optimiser les performances d'exécution sur une machine cible. Nous reparlerons de cette démarche en Section 8.2.3.).

7.4.5. Description d'une architecture matérielle avec HADEL

Nous nous sommes inspirés des approches de Parallel PROTO, Proteus et SySl pour définir HADEL (**HA**rdware **DE**scription **L**anguage) **[90]**. Les objectifs d'HADEL sont :

- la simplicité d'utilisation : pour cela, une visualisation graphique et hiérarchique a été considérée pour la mise en œuvre;
- le regroupement d'un minimum d'informations utiles sur la description d'un système : à cette fin, nous définissons des classes pour lesquelles certaines rubriques doivent être renseignées;
- la prise en compte d'une description au niveau gros grain comme au niveau grain fin : cela est réalisé au moyen d'un formalisme graphique et hiérarchique;
- l'extensibilité : il est possible d'envisager, à terme, l'ajout de nouvelles rubriques ou de nouvelles classes.

La description gros grain d'une architecture matérielle

Au niveau gros grain, HADEL permet de relier des mono-processeurs et des multi-processeurs au moyen de liens. Les mono-processeurs sont issus d'une classe pour lesquelles on doit renseigner les informations suivantes :

- un identificateur,
- le type de CPU (pour prendre en compte le format des exécutables),
- la puissance,
- la capacité mémoire.

Les multi-processeurs correspondent à un lien vers une description grain fin. Les liens auxquels est associé un débit relient les machines.

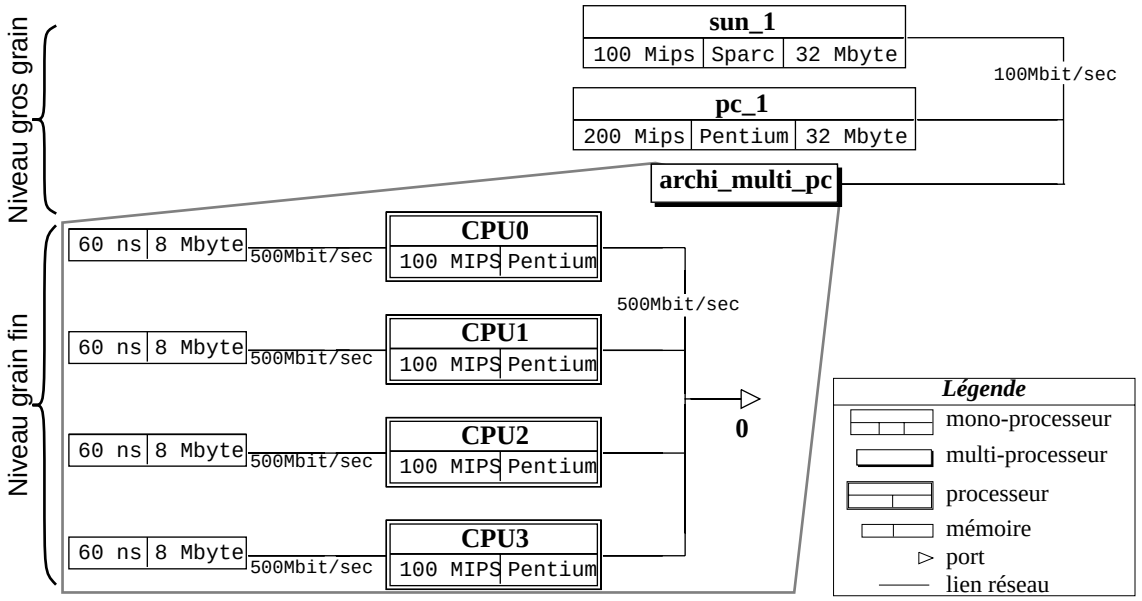

Figure 13 : Exemple de description d'architecture avec HADEL.

La description grain fin d'une architecture matérielle

Au niveau grain fin, HADEL permet de relier des processeurs à des ports ou à des mémoires au moyen de liens. Les processeurs sont caractérisés par un identificateur, leur type et leur puissance. Les mémoires sont également caractérisés par un identificateur, leur taille et leur vitesse. Comme pour les brins réseau dans le niveau gros grain, les bus sont caractérisés par leur débit.

Exemple de description d'une architecture matérielle

La Figure 13 présente un exemple de description d'une architecture avec HADEL. Il s'agit d'un réseau de trois machines (niveau gros grain) dont l'une, multi-processeur, est détaillée (grain fin). Les trois machines sont connectées par un réseau de 100 Moctets par seconde. La machine multi-processeur est composée de quatre cartes identiques comprenant un Pentium et une mémoire locale connectés via un bus privé de 500 Mbit par seconde. Les cartes sont elles-mêmes reliées par un bus à 500 Mbit par seconde.

7.4.6. La phase de placement

La phase de placement calcule un placement des entités logicielles du prototype sur une architecture matérielle. Deux approches sont actuellement étudiées : l'exploitation de résultats théoriques de la Recherche Opérationnelle ou l'utilisation des réseaux de Petri pour vérifier la présence de configurations type sur lesquelles des «recettes» sont applicables.

Exploitation de résultats théoriques sur le placement

Dans les études théoriques menées sur le placement **[33, 217]** ou l'ordonnancement **[184, 63]** , les tâches (appelées *tâches élémentaires* dans ce document) correspondent à des processus atomiques consommant des données et produisant des

résultats. Dans les modèles qui nous intéressent, les tâches élémentaires ont une durée d'exécution et sont reliées par des contraintes de précédences. Deux coûts de communication sont associés à ces contraintes : un *coût de communication local* (lecteur et écrivain du message sont sur le même site) et un *coût de communication distant* (lecteur et écrivain sont situés sur deux sites différents).

Par rapport à ces modèles, H-COSTAM propose un niveau de granularité plus élevé puisque les processus peuvent émettre ou recevoir des messages dans des séquences d'actions ou des boucles. On peut cependant transformer certaines spécifications H-COSTAM en un modèle de tâches élémentaires par découpage des processus selon les règles suivantes :

1) Les processus sont instanciés en fonction du nombre d'instances statiques et dynamiques (nombre à évaluer);
2) Le découpage processus à états instanciés en tâches élémentaires s'effectue au niveau des points de communication. Dans un processus H-COSTAM, chaque séquence d'actions sans communication, encadrant la réception d'un message et/ou l'émission d'un message devient une tâche élémentaire;
3) Les actions reliées à un multi-rendez-vous sont fusionnées en une tâche élémentaire unique;
4) Pour les tâches élémentaires qui ne sont pas associées à un multi-rendez-vous, des contraintes de précédences «artificielles» sont déduites du découpage en tâches élémentaires d'un processus. Un coût de communication distant ∞ (destiné à forcer un regroupement des tâches issues d'une même instance de processus H-COSTAM), et un coût de communication local nul, sont associés à ces précédences. On peut affecter un autre coût si on autorise la migration de tâches;
5) Des contraintes de précédences «réelles» sont déduites des liens de communication entre processus. Des contraintes de précédences «réelles» sont également déduites des liens entre les processus et la tâche élémentaire correspondant à un multi-rendez-vous. Des coûts de communications locaux et distants leur sont attribués[(6)];
6) Selon les modèles, les boucles sont soit dépliées, soit ignorées, soit conservées. Par exemple, on peut aisément imaginer que des «petites boucles» soient dépliées et de longues boucles maintenues pour étudier un régime stationnaire. On peut imaginer qu'elles soient purement et simplement ignorées si les heuristiques appliquées ne les traitent pas et que l'on est incapable d'estimer le nombre d'itérations (ou s'il est trop important).

La Figure 14 présente un exemple d'un tel découpage lorsque des communications sont situées dans des séquences d'actions. Nous considérons (en haut) deux processus H-COSTAM instanciés une seule fois et communicant via un lien de communication et un multi-rendez-vous. Le dépliage selon la règle (1) nous

(6) Ces valeurs sont par exemple déduites des flux (nombre de messages, taille des messages) observés pendant des exécutions pilote.

donne deux fois une instance. Les tâches élémentaires 1, 2, 3, 4 et 5 sont issues de l'application de la règle (2) tandis que la tâche r est le résultat de l'application de (3). La contrainte 1→2 est déduite de la règle (4) et les autres contraintes sont issues de la règle (5).

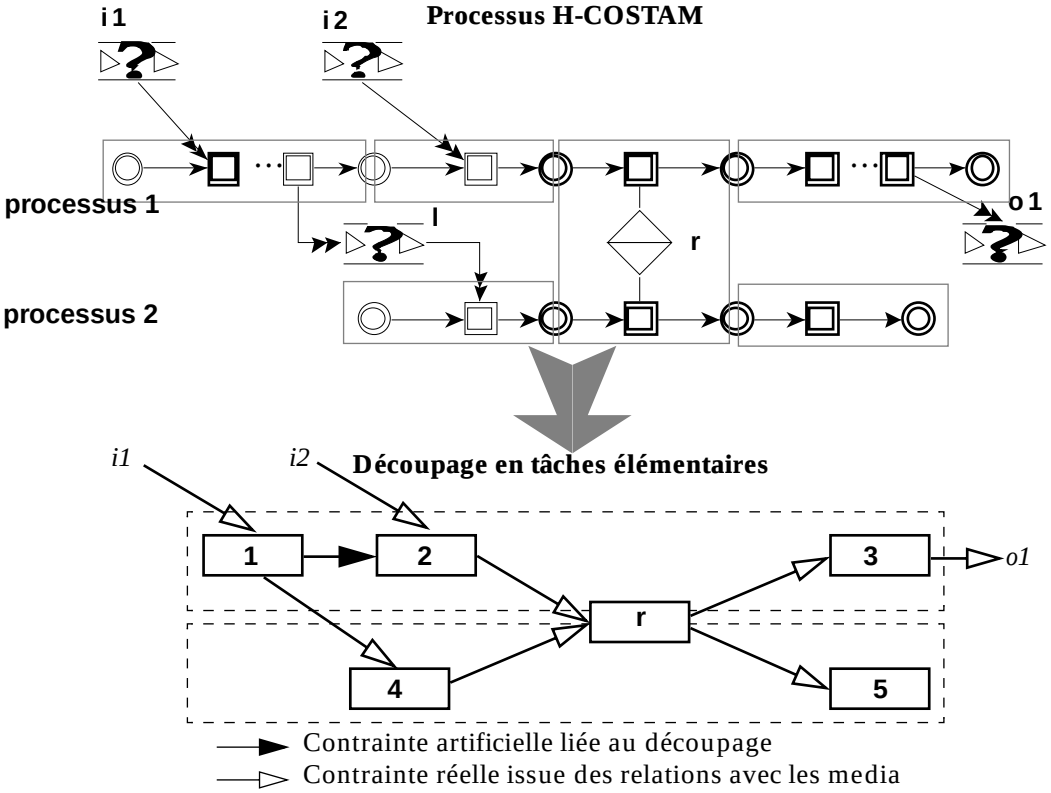

Figure 14 : Exemple de découpage de deux processus H-COSTAM effectuant plusieurs communications pendant leur exécution.

Une étude a commencé sur ce thème en collaboration avec C. Hanen et A. Munier. Le problème dans son ensemble est complexe mais deux axes de recherches sont envisagés :

- L'étude de transformations applicables à un sous-ensemble de H-COSTAM pour produire des modèles sur lesquelles des heuristiques connues sont applicables;
- L'étude d'extensions des modèles existants permettant d'intégrer les contraintes engendrées par une description de type H-COSTAM.

Utilisation des réseaux de Petri pour vérifier la présence de configurations type

Les réseaux de Petri issus de modèles H-COSTAM permettent de vérifier la présence de configurations type facilitant le travail de placement des composants du système par application de «recettes». Le concepteur pose des hypothèses que l'on essaye de vérifier pendant la phase de placement. Pour l'instant, deux familles de règles de transformation d'un modèle H-COSTAM sont élaborées pour :

- vérifier la présence d'une configuration de type pipe-line,
- vérifier la réplicabilité de certains media de communication.

Nous présentons dans un premier temps les règles de transformation ainsi que les propriétés attendues sur le réseau de Petri produit avant d'illustrer l'approche par un exemple.

Pour vérifier la présence d'une configuration de type pipe-line, les règles suivantes doivent être appliquées au sous-modèle H-COSTAM sur lequel on souhaite vérifier l'hypothèse :

1) Chaque processus ou sous-système impliqué dans le pipe-line pressenti est représenté par une transition lorsqu'il ne comporte pas de traitement potentiellement divergent (une boucle). Les comportements potentiellement divergents doivent être synthétisés en réseaux de Petri;
2) Les media de communication sont représentés par des places reliées aux transitions représentant les processus, conformément au modèle H-COSTAM. Les arcs reliant les places représentant un lien de communication sont unidirectionnels, ceux reliant les places correspondant aux multi-rendez-vous ou aux RPC sont bidirectionnels;
3) Une place supplémentaire nommée «vérification hypothèse» est ajoutée pour relier entre elles les deux extrémités du pipe-line pressenti.

Le réseau de Petri ainsi obtenu ne décrit qu'un modèle de communication. Si l'hypothèse énoncée est vérifiée, la place supplémentaire introduite par la règle (3) engendre un ou plusieurs invariants de places couvrant toutes les transitions (i.e. les étages) du réseau de Petri (i.e. du pipe-line).

Pour vérifier qu'un ensemble de media est bien réplicable en fonction des valeurs d'un type donné, les règles suivantes doivent être appliquées au sous-modèle H-COSTAM auquel s'applique l'hypothèse que l'on souhaite vérifier :

1) L'automate de contrôle des processus impliqués est transformé en réseaux de Petri. On ne traduit cependant en classes de couleurs que les types concernés par l'hypothèse que l'on cherche à vérifier. Les conditions associées aux actions H-COSTAM sont reproduites quand elles concernent des variables d'un type traduit en classes de couleurs. Les autres préconditions sont positionnées à vrai;
2) Les liens de communication sont traduits en places et les RPC sont transformées en couples de deux places (transmission des paramètres et attente du résultat). Ces media sont connectés conformément au modèle H-COSTAM aux processus qu'ils relient;
3) Les transitions connectées à un multi-rendez-vous sont fusionnées.

Une fois le modèle obtenu, on effectue un dépliage et l'on étudie les composantes connexes dans le réseau P/T correspondant. Si les media que l'on considère sont bien réplicables, on doit avoir une composante connexe par valeur possible

de la classe de couleur issue du type. Chacune de ces composantes contiendra une copie du ou des media concernés par l'hypothèse.

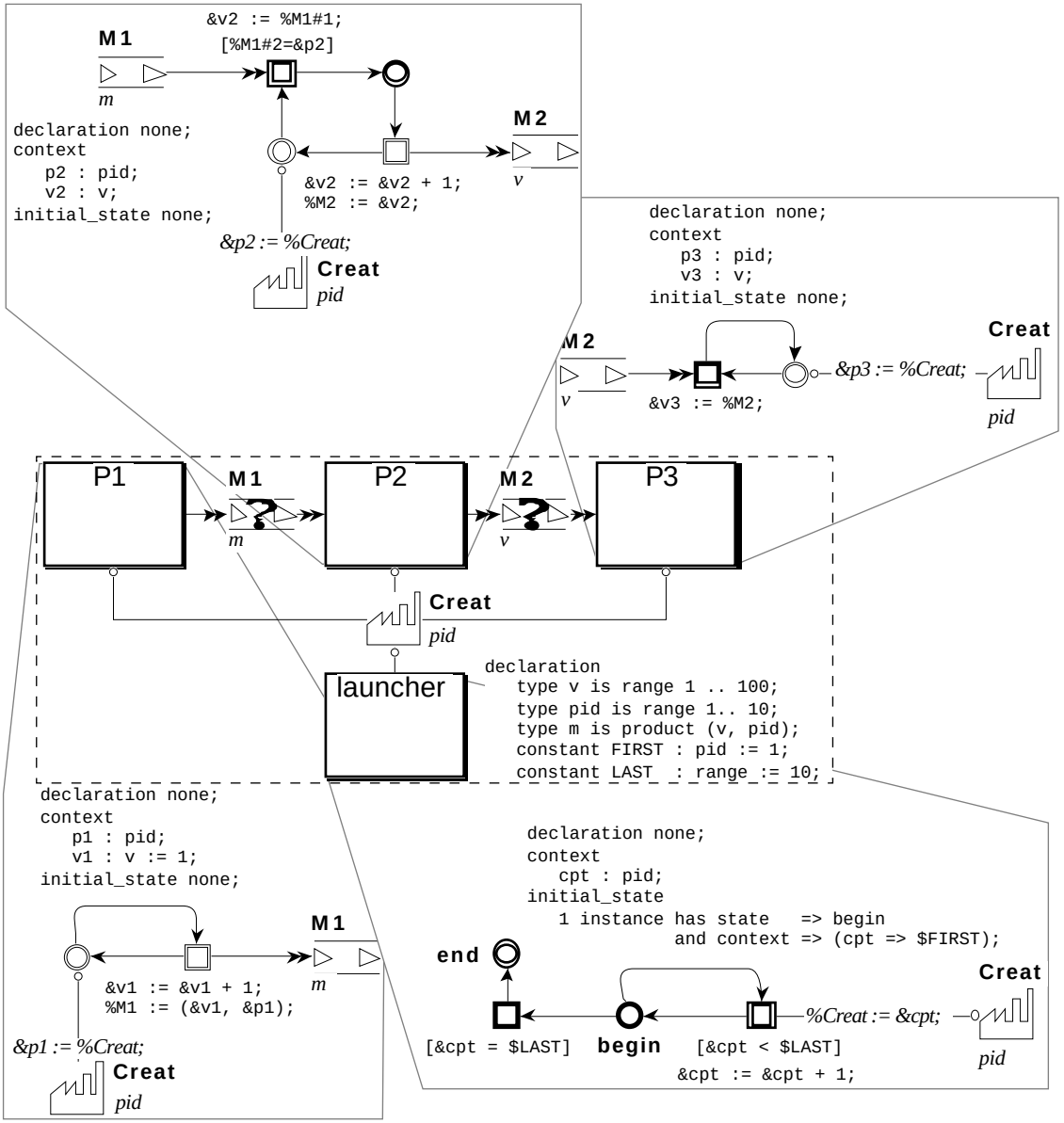

Figure 15 : Modèle d'exemple pour le placement.

Exemple d'analyse pour le placement des entités d'un système

Le modèle de la Figure 15 permet d'illustrer l'utilisation de ces transformations. Le système est composé de quatre classes de processus :

- *P1*, qui produit dans *M1* un message au format <v, pid> où v est une valeur calculée et pid le numéro d'identification de l'instance qui travaille,
- *P2* qui récupère depuis *M1* les messages provenant d'une instance de processus *P1* ayant la même identité, incrémente la valeur ainsi reçue puis l'écrit dans *M2*,

- *P3* qui consomme les valeurs contenues dans *M2* quel que soit l'émetteur,
- *launcher* qui a pour charge l'initialisation du système complet, c'est-à-dire la création des triplets de processus *P1*, *P2* et *P3* (10 instances chacun).

Nous nous proposons de vérifier les hypothèses suivantes :

a) les processus *P1* / *P2* / *P3* forment un pipe-line,

b) le media *M1*, situé entre *P1* et *P2* est réplicable selon le type pid.

Le modèle construit pour évaluer l'hypothèse (a) est indiqué en Figure 16. Le processus *launcher* et le media *Creat* ne sont pas représentés. Ils ne sont pas concernés par l'hypothèse et leur présence introduit des informations superflues.

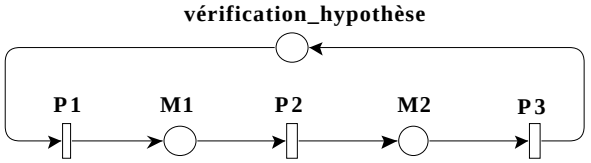

Figure 16 : Réseau de Petri permettant de vérifier la présence d'un pipe-line reliant P1, P2 et P3 dans le modèle de la Figure 15.

Ce réseau de Petri contient un invariant trivial[7] qui démontre l'existence du Pipe-line puisqu'il couvre tous les étages :

$$M1 + M2 + vérification\_hypothèse = constante$$

Le modèle construit pour évaluer l'hypothèse (b) est présenté en Figure 17. Les processus *P3* et *launcher*, ainsi que *M1* et *Creat*, ne sont pas représentés pour les mêmes raisons que précédemment. Notons que des classes de couleurs inutiles pour la vérification de notre hypothèse ont été écartées : la place représentant M1 contient donc des jetons simples de type *pid*.

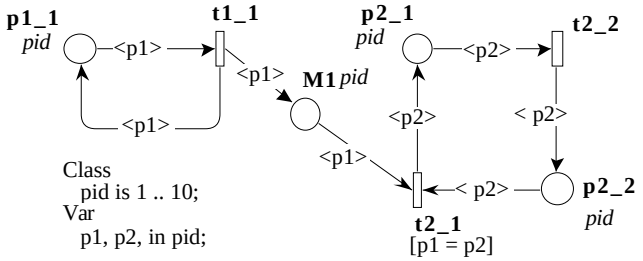

Figure 17 : Réseau de Petri permettant d'évaluer la réplicabilité du media de communication M1 dans le modèle de la Figure 15.

Le dépliage d'un tel modèle aboutit à un réseau P / T ayant 10 composantes connexes[8] (voir Figure 17). La réplication de *M1* sur plusieurs sites peut être réalisée conformément aux contraintes structurelles de l'application.

Ainsi, à l'issue de cette analyse, nous disposons de deux caractéristiques importantes sur le modèle de la Figure 15 :

- Il existe un pipe-line constitué par *P1*, *P2* et *P3*. On peut facilement imagi-

(7) Calculé avec GreatSPN dans CPN-AMI2.

(8) Calculé avec l'outil de dépliage de CPN-AMI2.

ner de placer chaque étage du pipe-line sur une machine différente;

- Le lien de communication M1 est réplicable selon la «direction» indiqué par le type pid. On peut donc placer des groupes *P1* / copie de *M1* / *P2*, pour des sous-ensembles disjoints de valeurs du type pid, sur des machines distinctes.

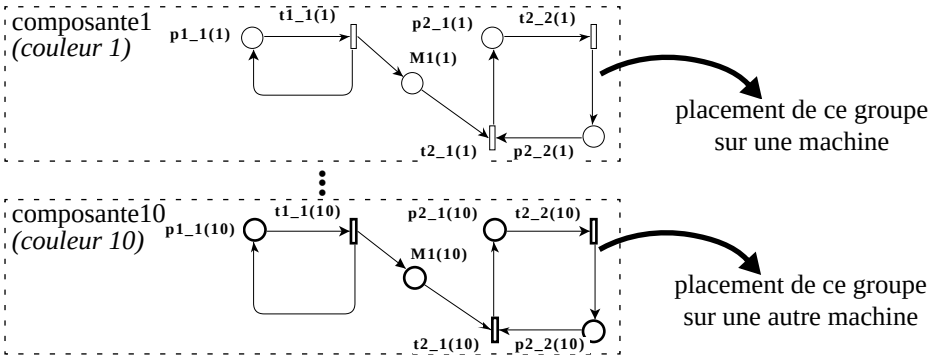

Figure 18 : Dépliage du modèle de la Figure 17 (seules deux composantes ont été représentées).

En combinant ces deux observations, et considérant que l'on dispose de 4 processeurs, on propose le placement indiqué dans le Tableau 3 :

- En exploitant l'hypothèse (b), on constitue trois groupes $P1_{ID}$ / copie de $M1_{ID}$[9] / $P2_{ID}$ (où ID représente un sous-ensemble des valeurs du type *pid*) que l'on place sur des processeurs distincts; les différentes copies de M1 sont implémentées au moyen de segments de mémoire locale et aucun mécanisme de cohérence n'est à mettre en œuvre entre les différentes copies
- En exploitant l'hypothèse (a), les instances de *P3* (qui consomment du CPU vers la fin de l'exécution) sont placées sur le quatrième processeur en même temps que l'unique instance de *launcher* (qui consomme du CPU au début de l'exécution). Une unique copie de *M2* est gérée par un gestionnaire de media passif sur le processeur 4.

numéro de processeur	launcher	P1	P2	P3	M1	M2
1		instances numérotées de 1 à 3	instances numérotées de 1 à 3		1 copie (éventuellement 3)	
2		instances numérotées de 4 à 7	instances numérotées de 4 à 7		1 copie (éventuellement 4)	
3		instances numérotées de 8 à 10	instances numérotées de 8 à 10		1 copie (éventuellement 3)	
4	1 instance			toutes les instances		1 copie

Tableau 3: Proposition de placement des éléments du modèle H-COSTAM de la Figure 15.

(9) par M1, il faut entendre, le gestionnaire de media passif associé à M1

On peut également discuter le nombre de copies de *M1* sur chaque site : une seule, partagée par tous les couples *P1* / *P2* présents sur le site, ou une copie par couple *P1* / *P2* présent. Du point de vue de l'exécution, la seconde solution améliore les performances en supprimant les synchronisations liées à la gestion d'une ressource critique.

8. Mise en œuvre de la Méthode MARS

Pour éprouver la méthodologie MARS en l'utilisant sur des spécifications de taille «réelle», nous développons CPN-AMI2 **[199]**, un Environnement de Génie Logiciel pilote. CPN-AMI2 nous permet d'évaluer la pertinence de MARS et de l'adapter en fonction des problèmes rencontrés.

CPN-AMI2 est utilisé en enseignement (DEA Systèmes Informatiques) et pour des expérimentations dans le cadre de contrats industriels (action VAMOS **[79]** dans le projet FORMA et le projet CNET / CARISMA).

8.1. CPN-AMI2 : un Environnement de Génie Logiciel pour MARS

CPN-AMI2 a été bâti au-dessus de la plate-forme FrameKit **[169]** qui sera détaillée dans le Chapitre 3. Pour cela, nous avons paramétré les formalismes dans l'interface utilisateur Macao **[207]** et réalisé les services associés.

8.1.1. Formalismes de CPN-AMI2

Six formalismes ont été définis pour implémenter les services de CPN-AMI2. Quatre d'entre eux sont dédiés à la construction de modèles :

- *OF-Class* : la version actuelle revêt une forme textuelle,
- *H-COSTAM* : la version actuelle est hiérarchique,
- *AMI-Net* : hérité de CPN-AMI1 **[198]**, cette description correspond sémantiquement aux réseaux de Petri bien formés **[56]** auxquels des contraintes syntaxiques ont été définies afin d'en faciliter l'analyse **[35]**,
- *HADEL* : la version actuelle est hiérarchique sur deux niveaux (gros grain et grain fin).

Deux autres formalismes sont dédiés à la visualisation de résultats. *ReachabilityGraph* décrit les entités d'un graphe des marquages accessibles (éventuellement symbolique). *BranchingProcess* décrit le résultat du dépliage McMillan **[200]** d'un réseau de Petri Place / Transition.

8.1.2. Enchaînements des formalismes dans CPN-AMI2

Le point d'entrée de la méthode MARS est constitué par OF-Class. C'est donc aussi le point d'entrée privilégié de CPN-AMI2, même s'il est possible de travailler directement au niveau de H-COSTAM ou des réseaux de Petri.

Des services sont associés aux quatre formalismes de CPN-AMI2 dédiés à la construction de modèles. Ces services effectuent des calculs sur le modèle et fournissent un résultat à l'utilisateur. Des *services de liaison* produisent à partir du

modèle source, un ou plusieurs modèles cibles exprimés dans un autre formalisme. Par exemple, le service d'Elicitation (Figure 19) produit un modèle H-COSTAM à partir d'une spécification OF-Class.

Le passage d'un formalisme à un autre dans CPN-AMI2 se fait uniquement via les services de liaison (élicitation, synthèse, génération d'application et de directives de placement). Dans le cas du formalisme HADEL, les architectures sont stockées dans une base de données et sélectionnées lorsque la génération de placement est demandée (Figure 19).

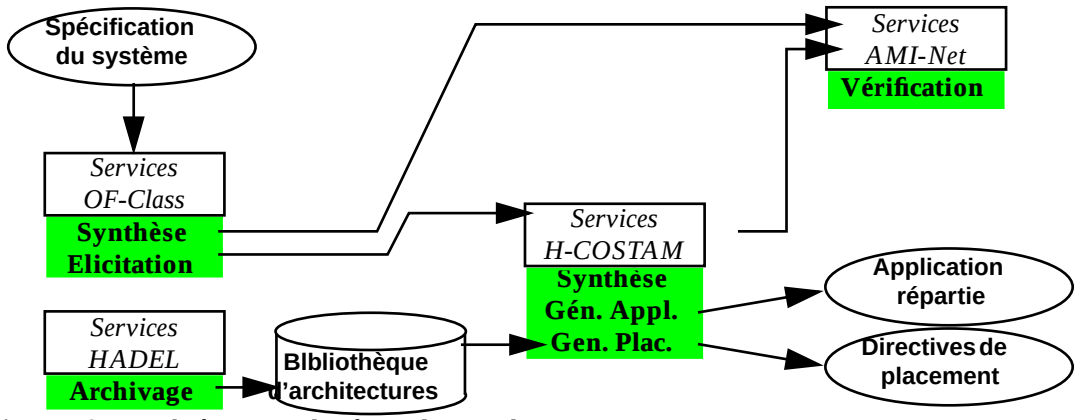

Figure 19 : Enchaînement des formalismes dans CPN-AMI2.

Des services de vérification permettent le calcul de propriétés sur les réseaux de Petri (à terme, les résultats devraient être retranscrits dans les termes des formalismes de haut niveau). Depuis H-COSTAM, deux services sont dédiés à la production d'une application répartie et de directives de placement (respectivement «Gen. Appl.» et «Gen. Plac.» dans la Figure 19).

8.1.3. Les services dans CPN-AMI2

Hormis HADEL, pour lequel c'est inutile, un simulateur / débogueur est prévu à tous les niveaux (conceptuel, opérationnel et formel). Celui animant des réseaux de Petri est hérité de CPN-AMI1 et peut s'avérer utile dans une phase d'expérimentation. Celui d'OF-Class doit permettre au concepteur d'une application d'animer une unité en fonction des prérequis sur son environnement, puis un ensemble d'unités afin d'étudier plus finement leur comportement. Cet outil est un préalable à la mise au point du système mais ne doit pas se substituer à la validation formelle. Enfin, un simulateur au niveau H-COSTAM peut fournir des informations sur le comportement du prototype avant qu'il n'ait été généré (par exemple, dimensionnement de certaines ressources, comptage des messages échangés entre entités H-COSTAM, étude de scénario, etc.).

L'analyse des propriétés structurelles d'un modèle OF-Class n'est pas encore effectuée de manière transparente mais des essais ont été réalisés pour l'évaluation de formules de logique temporelle (model checking). L'analyse du graphe d'accessibilité d'un modèle OF-Class, assurée par l'outil PROD **[279]**, se fait sans

visualisation du réseau de Petri sous-jacent. L'utilisateur identifie au moyen de labels des portions de spécifications dans ses formules de logique temporelle.

Formalisme	Services	Classe de RdP		Etat d'avancement				
		P/T	Colorés	Prévu	Étude préliminaire	Partiellement opérationnel	Version prototype	Opérationnel
AMI-Net	Vérification syntaxique	✓	✓					✓
	Simulation	✓	✓					✓
	Formule booléenne sur le graphe des marquages accessibles	✓[a]						✓
	Borne de places	✓						✓
	Invariants de places	✓	✓					✓
	Invariants de transitions	✓						✓
	Verrous et trappes	✓						✓
	Invariants de places et transition (par dépliage)		✓					✓
	Verrous et trappes (par dépliage)		✓					✓
	Caractérisation de propriétés linéaires	✓						✓
	Recherche de marquages morts	✓					✓	
	Dépliage de réseaux de Petri colorés		✓					✓
	Dépliage McMillan	✓[a]	✓[a]					✓
	Génération du Graphe des marquages accessibles	✓	✓					✓
	Génération du Graphe des marquages symboliques		✓				✓	
	Evaluation de formule de logique temporelle	✓	✓					✓
	Placement des objets du réseau de Petri	✓	✓					✓
OF-Class	Vérification syntaxique							✓
	Synthèse de réseaux de Petri Colorés							✓
	Analyse de l'espace d'état (réseaux de Petri masqués)					✓		
	Elicitation				✓			
	Simulation				✓			
H-COSTAM	Vérification syntaxique						✓	
	Synthèse de réseaux de Petri					✓		
	Simulation			✓				
	Génération de Code				✓			
	Génération de placement				✓			
	Optimisation (usage transparent des réseaux de Petri)				✓			

Tableau 4: Services de CPN-AMI2.

Formalisme	Services	Classe de RdP		Etat d'avancement				
		P/T	Colorés	Prévu	Étude préliminaire	Partiellement opérationnel	Version prototype	Opérationnel
HADEL	**Vérification syntaxique**						✓	
	Gestion de bibliothèques d'architectures (archivage)			✓				

Tableau 4: Services de CPN-AMI2.

a. Réseaux de Petri 1-bornés seulement.

La liste des services actuellement disponibles dans CPN-AMI2 est indiquée dans le Tableau 4. Pour les services disponibles sur les réseaux de Petri, nous indiquons la classe à laquelle ils s'appliquent. Nous indiquons également l'état d'avancement de leur développement.

Certains services ont été développés dans notre équipe ou importés depuis CPN-AMI1 (basé sur la plate-forme AMI **[23]**). D'autres sont rendus par des outils provenant d'autres universités[(10)] :

- PROD **[279]**, développé à l'Université Technologique d'Helsinki, permet d'explorer le graphe d'accessibilité d'un réseau de Petri et de l'explorer au moyen d'un langage d'interrogation permettant de vérifier des contraintes temporelles linéaires et arborescentes. Nous l'utilisons également pour restituer le graphe d'accessibilité (intérêt d'ordre pédagogique). Nous avons intégré cet outil de manière à offrir des services d'analyse à la fois pour les réseaux de Petri (formalisme AMI-Net) et pour les OF-Class;
- GreatSPN **[57, 60]**, développé à l'Université de Turin, dont nous avons importé les modules de calcul d'invariants P/T (places et transitions), de verrous, de trappes et de construction du graphe des marquages symboliques;
- La production d'une description sous la forme de processus arborescents d'un réseau de Petri (algorithme de dépliage d'Ezparza, Römer et Vogler, **[93]**, dérivé de celui de McMillan **[200]**) issu de l'environnement PEP **[28]**, développé à l'Université d'Hildesheim;
- L'outil de construction de graphes dot **[176, 177]**, développé aux AT&T Bell-Laboratories permet de placer correctement certains des résultats produits (graphe d'accessibilité, réseaux de Petri colorés dépliés ou description d'un réseau de Petri en processus arborescents).

CPN-AMI2 est disponible sur Internet depuis Mars 1997 (la version 1 l'a été à partir de Novembre 1993).

(10) les conditions de leur intégration sera discutée en Section 13.1.

8.2. Autres environnements de prototypage

Cette section est dédiée à la description de quelques environnements ou outils ayant des objectifs de type modélisation / vérification et / ou prototypage.

Nous y présentons brièvement les caractéristiques d'outils basés sur les réseaux de Petri et dégageons PEP qui, par bien des aspects, comporte des similitudes avec notre approche. Nous nous intéressons également à d'autres environnements de développement / prototypage d'applications parallèles : Proteus, TRAPPER et PARSE.

8.2.1. Outils et environnements basés sur les réseaux de Petri

Dans le Tableau 5, nous avons regroupé les caractéristiques des principaux outils réseaux de Petri dont nous avons eu connaissance. Ils sont identifiés dans la base de référence officielle maintenue par l'Université d'Aarhus **[1]**, certains sont également l'objet d'une évaluation selon des critères d'utilisation industrielle dans **[269]**.

Nous avons retenu, dans notre classification, trois critères :

- la classe de réseaux de Petri supportés : Place / Transition, Colorés, temporisés, stochastiques ou hiérarchiques;
- les fonctions d'évaluation de modèles : essentiellement de l'animation, de la simulation ou d'autres fonctions comme la construction du graphe d'accessibilité (model checking) ou l'analyse de performances;
- les fonctions de validation de modèles : essentiellement de l'analyse structurelle (calcul d'invariants), parfois agrémentée de réductions au sens de **[25]** ou de la détection de classes particulières de réseaux (machines à états, réseaux à choix libre...).

Nom de l'outil	Classe de RdP					Évaluation					Validation			
	P/T	Colorés	Stochastiques	Gest. temps	Hiérarchiques	Animation	Simulation	Vérification de modèle[a]	Anal. perf.[b]	Autres	P-Invariants	T-Invariants	Réductions	Autres[c]
ALPHA/Sim [9]		✓			✓	✓	✓		✓					
ANARCO/PAREDE[d] [237]	✓	✓		✓				✓			✓	✓	✓	
Artifex [13]		✓		✓		✓	✓		✓					
CodeSign [94]		✓		✓	✓	✓	✓							
CO-OPN [30]		✓			✓		✓							
Design/CPN [72]		✓		✓	✓	✓	✓	✓	✓	✓[e]				
DSPNexpress [187]			✓			✓		✓	M		✓			
EDS Petri Net Tool [1]		✓	✓	✓		✓	✓		✓					✓

Tableau 5: Classification des outils et environnements basés sur les réseaux de Petri.

Nom de l'outil	Classe de RdP					Évaluation					Validation			
	P/T	Colorés	Stochastiques	Gest. temps	Hiérarchiques	Animation	Simulation	Vérification de modèle[a]	Anal. perf.[b]	Autres	P-Invariants	T-Invariants	Réductions	Autres[c]
Elsir [7]		✓		✓	✓	✓	✓							
ExSpect [18]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[f]	✓	✓		✓
F-net [1]				✓		✓	✓		M		✓	✓		✓
GreatSPN [60]	✓	✓	✓			✓	✓	✓	✓		✓	✓		
HiQPN-Tool [22]		✓	✓		✓	✓		✓	M		✓	✓		
INA [243]	✓	✓		✓				✓	M		✓	✓	✓	✓
Looping [1]		✓				✓	✓							
Netmate [216]	✓	✓		✓		✓	✓				✓	✓		
PED[g] [37]	✓			✓	✓									
PETRI Maker [1]	✓					✓	✓	✓	✓		✓	✓		
PetriSim [260]	✓	✓		✓		✓			M					
PNTalk [153]		✓		✓		✓	✓		✓					
POSES++ [115]	✓	✓	✓	✓				✓						
PROD [279]	✓	✓						✓						
SEA [139]		✓[h]			✓	✓	✓							
SPN2MGM [136, 137]			✓				✓		M					
STROBOSCOPE [1]				✓			✓		M		✓	✓		
SURF-2 [270]			✓					✓	M		✓	✓		
SYROCO [257]		✓		✓			✓		✓					
THORN/DE [253]		✓		✓	✓		✓		✓					
TimeNET [156]	✓		✓	✓		✓	✓		M		✓			
Visual SimNet [1]	✓	✓	✓	✓		✓	✓	✓	M		✓	✓		✓
WebSPN [1]			✓						M					

Tableau 5: Classification des outils et environnements basés sur les réseaux de Petri.

a. Model Checking.
b. **[1]** différencie l'analyse «simple» de l'analyse «avancée», qui repose sur une approche markovienne, ce que nous notons par la lettre M.
c. Comme la détection de classes particulières de réseaux de Petri
d. Il s'agit de deux applications MS / DOS différentes développés par la même équipe et partageant une interface graphique commune : GraPETRI.
e. Association de procédures externes en C ou en ML aux transitions.
f. Association de procédures externes en C aux transitions.
g. PED est un éditeur qui sait exporter des fichiers de description au format d'outils comme INA ou PROD.
h. Les réseaux de Petri colorés à prédicats servent de noyau à un environnement de simulation supportant des classes de modèles hiérarchiques.

Les outils basés sur les réseaux de Petri se divisent en deux catégories. La première propose des modèles hiérarchiques de haut niveau, plus conviviaux d'uti-

lisation (comme Design/CPN, POSE++). Le prix à payer est souvent l'absence de fonctions de validation formelle. Ainsi, ces outils proposent essentiellement de la simulation, à l'exception notable d'ExSpect (il est possible de valider des modules) et HiQPN (des sous-modèles devant respecter certaines contraintes sont associés aux places, l'analyse est réalisée sur un modèle «aplati» **[21]**).

L'autre catégorie dispose en général de meilleures capacités d'évaluation et de validation mais repose sur des classes de réseaux de Petri plus simples (comme F-Net ou INA). Ainsi, des caractéristiques «agréables», comme la hiérarchie, ne sont plus présentes pour les modèles manipulés.

8.2.2. PEP

L'environnement PEP (**P**rogramming **E**nvironment based on **P**etri Nets) est développé par l'Institut d'Informatique de l'Université d'Hildesheim **[28, 228]** . Leur approche repose sur une encapsulation des réseaux de Petri au moyen d'un langage impératif : B(PN)² (pour **B**asic **P**etri **N**et **P**rogramming **N**otation) **[27]**. Ce langage possède une sémantique de composition proche de celle des réseaux de Petri. Il permet de définir des séquences, des choix non déterministes, des itérations et possède un opérateur de parallélisme. Les dernières extensions intègrent également la notion de procédure **[100]**. Une surcouche de B(PN)² a été récemment proposée : PFA (**P**arallel **F**inite **A**utomata) **[127]**.

PEP synthétise des réseaux de Petri P/T ou colorés à partir d'une description en B(PN)². Ce langage peut également être transformé en termes d'une algèbre de processus dérivée de CCS **[29]**. Cette algèbre, comme les réseaux de Petri, peut également être directement utilisée par un usager de PEP.

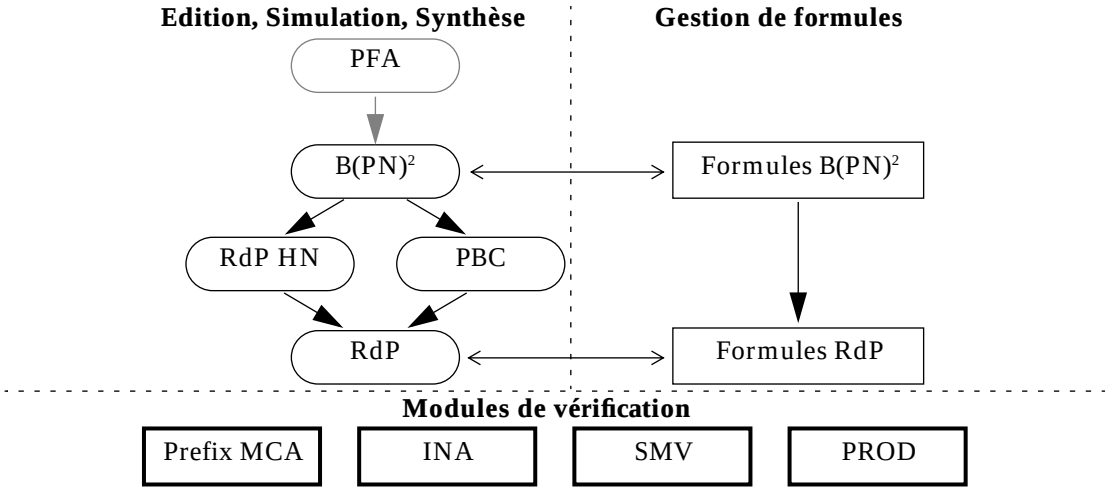

Figure 20 : Architecture de PEP.

PEP offre des possibilités de simulation aux niveaux B(PN)² et réseaux de Petri. Il permet également de vérifier des formules de logique, soit au niveau du programme B(PN)², soit au niveau des réseaux de Petri (au moyen d'équivalences). Au niveau des réseaux de Petri, Prefix MCA, un composant de vérification par model checking **[92]**, basé sur des techniques de dépliage (de réseaux de Petri

colorés) efficaces **[93]** a été développé. Plus récemment, des liens ont également été établis avec d'autres outils disponibles sur Internet : INA **[266]**, PROD **[279]**, et SMV **[65]**, enrichissant ainsi les possibilités de PEP.

Certaines références mentionnent également de la génération de code en langage C à partir de spécifications B(PN)^2 **[201]**, mais il semble qu'elle soit surtout utilisée afin d'accélérer la simulation. De fait, les programmes produits sont difficiles à intégrer à un environnement d'exécution pré-existant.

8.2.3. Proteus

Proteus est un environnement de développement pour la mise au point d'applications réparties **[12, 121, 122]** . Ce projet regroupe l'University of North Carolina Chapel Hill, le Kestrel Institute, et la Duke University dans le cadre d'un contrat avec le DoD. L'objectif est de définir un langage de haut niveau permettant à des ingénieurs de mettre au point des algorithmes, de les particulariser par raffinements successifs en fonction de contraintes architecturales d'une machine hôte, puis de synthétiser le code adapté cette machine.

Le processus de prototypage mis en œuvre est illustré en Figure 21. Durant la phase de spécification, on construit des programmes sans réfléchir aux caractéristiques d'une architecture cible. Plusieurs versions peuvent apparaître; l'exemple en propose trois, l'une d'elles (P') étant par exemple abandonnée parce que jugée mal optimisée. Durant la phase de raffinement, on intègre les contraintes sur l'architecture cible. Ici, du C + des caractéristiques vectorielles (Pv) ou du C + communication par messages avec deux variations, PVM (S1) et thread Posix (S2). La phase de traduction génère des programmes optimisés.

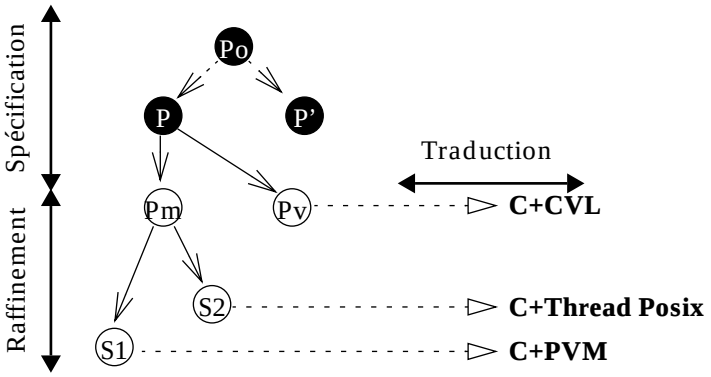

Figure 21 : Processus de prototypage dans Proteus.

L'environnement repose sur Proteus, un langage impératif comprenant des fonctions de base, des structures de données complexes et des constructions permettant d'exprimer du parallélisme **[98]** :

- au niveau des données (data-parallelism) lorsque des éléments d'un agrégat (tableau, article) sont potentiellement parallélisables;
- au niveau des traitement (process-parallelism) lorsque des séquences de code peuvent être composées ou pour des processus distincts.

L'architecture de Proteus est présentée en Figure 22. Le module *Elaboration* regroupe les outils mettant en œuvre la démarche illustrée par la Figure 21. L'historique des modifications effectuées pendant les phases de spécification et de raffinement est conservé dans un référentiel qui stocke également les programmes obtenus par traduction. Si la spécification est effectuée manuellement, les phases de raffinement et de traduction sont complètement automatisées. La traduction est implémentée pour la machine parallèle virtuelle de CVL (C Vector Library). L'exploitation de machines virtuelles basées sur la communication par messages ou la mémoire répartie est prévue.

Le module *Exécution* permet ensuite d'évaluer les programmes par simulation des programmes Proteus. Pour cela, un exécuteur séquentiel simulant le parallélisme permet d'obtenir des historiques d'exécution précis.

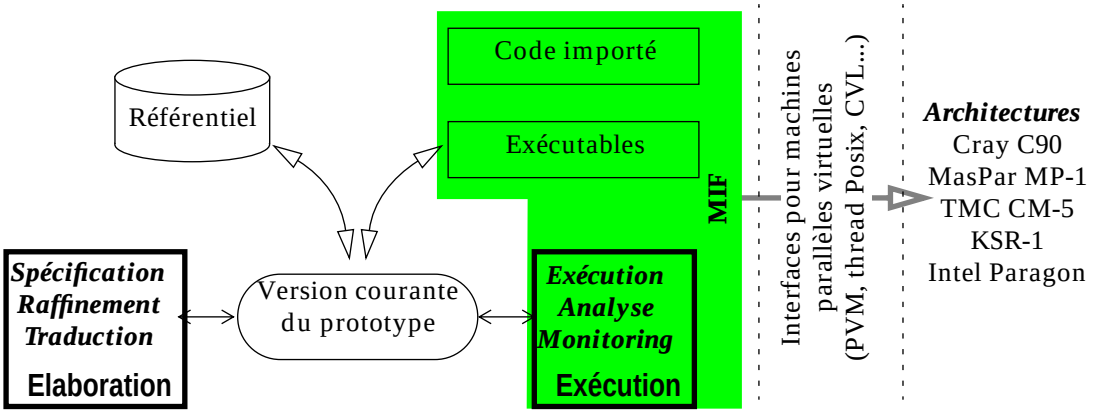

Figure 22 : Architecture de l'environnement Proteus.

Le simulateur est exploité pour effectuer de la prédiction de performances. Un modèle générique paramétré adapté à leurs besoins : LogP-HMM **[186]**, a été élaboré. Il intègre de deux extensions du modèle PRAM **[102]** : LogP **[73]** (qui intègre les temps de communications) et P-HMM **[280]** (qui exploite les caractéristiques d'une mémoire hiérarchisée).

Le *MIF* (Module Interconnection Facility) permet d'intégrer des programmes écrits dans différents langages, en particulier des programmes Proteus et des programmes C déjà traduits et optimisés. Il permet également à l'interpréteur d'exploiter les super-calculateurs pour les parties de code déjà écrites, afin d'accélérer les séquences de simulation.

8.2.4. TRAPPER

TRAPPER est un environnement de spécification, animation et configuration de systèmes parallèles **[178, 251]** . Il a été développé conjointement par le laboratoire de calcul parallèle du GMD (Bonn) et le groupe de recherches de Daimler-Benz (situé à Berlin). Le projet dans sa phase initiale est terminé depuis la fin de 1994 mais reste utilisé dans le cadre de projets de recherches. Une version est même distribuée commercialement **[117]**.

L'environnement TRAPPER, dont l'architecture est représentée en Figure 23, est composé d'une interface commune à quatre outils :

- *DesignTool* pour la construction de graphes de processus décrivant une application. Cette représentation d'un système parallèle présente des points communs avec H-COSTAM, en particulier dans la gestion de la hiérarchie (par structuration en sous-systèmes). Par contre, les relations entre composants correspondent à des contraintes de précédence et les processus élémentaires sont décrits au moyen de programmes C ou OCCAM;
- *ConfigTool* pour définir le placement des processus sur une architecture matérielle. Il permet de décrire une architecture matérielle d'une manière voisine à ce que nous proposons dans le niveau gros grain d'HADEL et d'associer, manuellement ou automatiquement, les tâches aux processeurs. L'heuristique «iterated 2-Opt» **[214]** a été implémentée pour le placement automatique;
- *VisTool* pour animer le modèle de l'application pendant son exécution;
- *PerfTool* pour analyser le comportement du matériel pendant l'exécution du système.

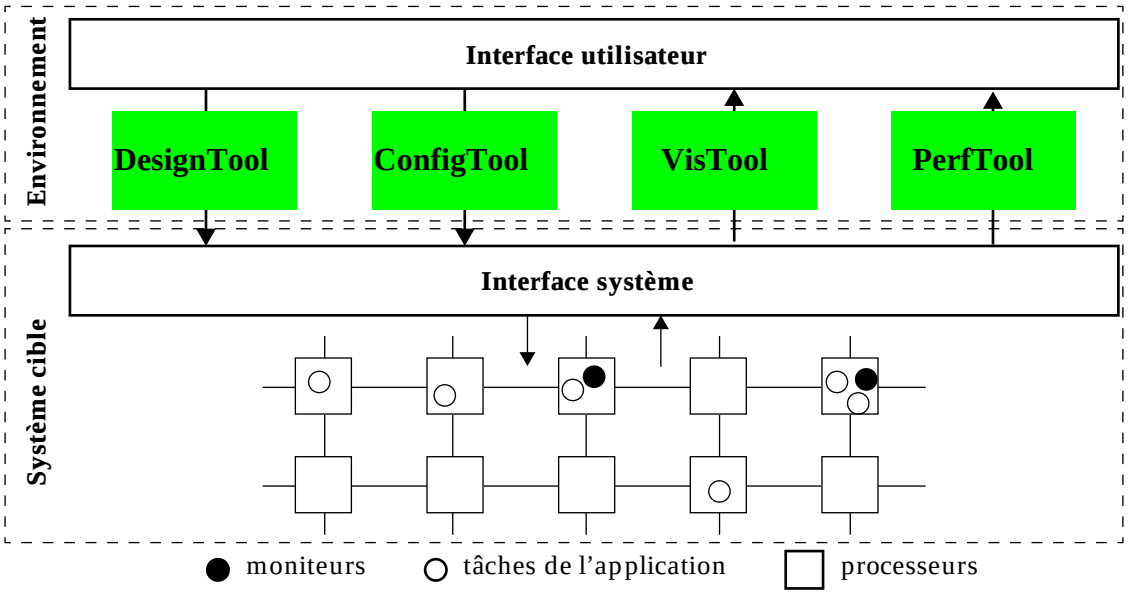

Figure 23 : Architecture de l'environnement TRAPPER.

La génération de code proposée dans TRAPPER permet l'assemblage de programmes correspondant aux différentes tâches en fonction de l'architecture logicielle de l'application. Le code généré repose sur PVM **[112]** pour ce qui est des communications. Pour surveiller le comportement du système, des tâches particulières (moniteurs) sont créées et utilisent les possibilités de PVM. Des options de génération permettent de produire les processus moniteurs en fonction de ce que l'on veut étudier sur l'application.

VisTool et PerfTool constituent les principaux outils de monitoring de TRAPPER. Ils implémentent des techniques sophistiquées de visualisation de la

charge des processeurs, et de surveillance des communications entre tâches de l'application pour la mettre au point et l'optimiser.

TRAPPER a permis d'étudier par simulation / exécution le comportement d'applications sur différentes architectures (par exemple, un réseau de T9000 dans **[249]**).

8.2.5. PARSE

L'environnement PARSE (**PAR**allel **S**oftware **E**ngineering) est le résultat d'une collaboration entamée en 1991 **[225]**. Ce projet regroupe à l'heure actuelle les universités de Sheffield Hallam et Sheffield (Angleterre), les Universités de Wollongong et New South Wales (Australie) le département d'Informatique et Systèmes de l'Université de Naple et un partenaire industriel australien. L'objectif du projet est la définition d'une méthode et d'outils pour le développement d'une large variété d'applications parallèles. L'approche suivie doit intégrer des objectifs de Génie Logiciel comme la portabilité et la définition d'abstractions indépendantes d'un langage de programmation. A ce titre, un grand nombre d'études sont en cours dans les domaines couverts par le projet :

- *Définition de règles de modélisation et de développement* : la méthodologie proposée passe par l'élaboration, puis le raffinement d'une description logicielle d'un système;
- *Définition de notations* : les formalismes proposés intègrent la démarche objet et possèdent une structure hiérarchique similaire à celle d'H-COSTAM **[226, 123]** . Les processus constituent également un élément de base pour la construction de leurs modèles mais ils sont classés en trois types : serveurs de fonctions, serveurs de données ou processus de contrôles. Les deux premiers correspondent à des objets passifs, le dernier définit des traitements de coordination (des autres types d'objets) potentiellement réutilisables;
- *Outillage* : en vue d'assurer un outillage cohérent, BSL (Behaviour Specification Language), une description textuelle standardisée des modèles a été mise au point **[247]**. Cette représentation sert de point d'entrée à tous les outils de leur environnement, qu'il s'agisse d'outils de dessin, de vérification ou de génération de code.
- *Étude du comportement des applications* : différentes techniques d'analyse sont proposées et reposent essentiellement sur de la simulation ou de la réutilisation de composants «sûrs». Cependant, des travaux plus récents ont abouti, dans certains cas, à établir un lien vers les réseaux de Petri stochastiques **[84, 248]** . Il semble que l'automatisation de ces transformations soit en cours et qu'un lien étroit avec GreatSPN soit envisagé pour réaliser l'analyse stochastique des spécifications PARSE.
- *Génération de code* : plusieurs générateurs de code ont été élaborés à partir du format BSL. Les générateurs de code produits fournissent des programmes en langage OCCAM ou du DISC (DIStributed C). Des études sont éga-

lement en cours sur le co-développement matériel / logiciel (codesign).

PARSE a été utilisé dans différents projets, plusieurs d'entre eux utilisant des architectures parallèles à base de transputer.

8.3. Synthèse

Par rapport aux outils réseaux de Petri «classiques», notre approche offre à la fois un confort d'utilisation appréciable et l'exploitation de techniques formelles pour la vérification. Les concepts manipulables par l'utilisateur au moyen de langages de spécification de haut niveau s'apparentent à la démarche objet mais restent validables formellement grâce à l'encapsulation de réseaux de Petri. La Génération de Code assure la production d'un prototype conforme à la spécification.

L'environnement PEP procède d'une approche d'encapsulation similaire à la nôtre. Sa force réside dans ses capacités de validation. La connexion avec différents outils développés dans d'autres universités accroît ses possibilités dans ce domaine. Par contre, l'aspect génération de code est moins poussé que dans nos travaux, ce qui fait de PEP un environnement de modélisation et de preuve plus qu'un environnement de prototypage. Les travaux actuels de cette équipe, qui semble s'intéresser aux liens entre les réseaux de Petri et des langages comme SDL, VHDL et OCCAM **[126]**, renforcent cette orientation.

Proteus est un environnement bien instrumenté et éprouvé sur plusieurs études de cas **[122]**. Il découle d'une démarche de prototypage, mais certains choix diffèrent des nôtres à cause du type d'applications visé (calcul à hautes performances pour super-calculateurs). Proteus se focalise sur la partie réalisation et optimisation. L'usager n'a pas vraiment la possibilité de vérifier que son algorithme fonctionne correctement autrement que par simulation. Cette étude sépare nettement les phases de conception et d'implémentation : dans un premier temps, l'usager se focalise sur l'exploitation du parallélisme et décrit son application avec le langage Proteus. Ensuite, il effectue des choix de réalisation et les intègre sous la forme de directives plus spécifiques. Cette démarche est similaire à celle que nous proposons avec deux modèles de haut niveau, l'un couvrant les aspects conception, l'autre les aspects réalisation.

TRAPPER se situe sur un plan similaire à celui de Proteus. Cependant, l'accent est mis sur les différentes manières de générer le code. TRAPPER est plus un environnement bien instrumenté qu'une approche dans la conception d'une application parallèle (approche présente dans Proteus). L'aide fournie concerne essentiellement la mise au point des programmes directement dans un langage de programmation (puisque c'est ainsi que sont décrits les composants élémentaires d'un système dans cet environnement) et le placement des différents composants sur une architecture donnée. Outre les aspects modélisation et mise en œuvre du système par génération de code, PARSE propose un lien vers une technique de description formelle en vue de la validation du système. Cela en fait un

environnement qui couvre largement le cycle de développement du logiciel, depuis la conception jusqu'à la mise en œuvre.

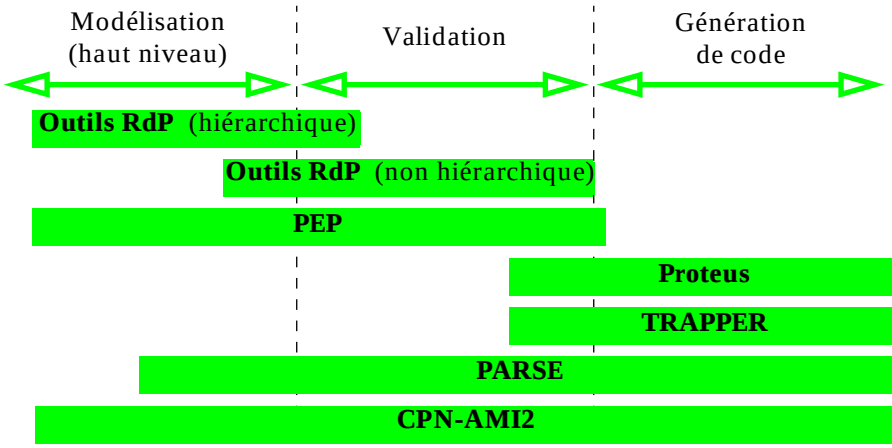

Figure 24 : Comparaison des approches proposées par les différents environnements présentés.

Si nous considérons nos objectifs principaux (modélisation de haut niveau, validation et génération de code), on peut considérer que les outils réseaux de Petri couvrent des besoins situés en amont de la phase de développement, Proteus et TRAPPER des besoins situés en aval de la phase de développement tandis que CPN-AMI2 et PARSE couvrent l'ensemble du cycle de conception. Cette vision est illustrée par la Figure 24 :

- les outils basés sur des réseaux de Petri de haut niveau couvrent bien l'aspect modélisation mais en général mal l'aspect validation (validation par simulation);
- les outils basés sur des réseaux de Petri non hiérarchiques couvrent en général mieux les aspects validation mais la modélisation sur de systèmes de grande taille reste délicate;
- l'environnement PEP couvre bien l'aspect modélisation (grâce aux différentes encapsulations qu'il propose) et s'appuie sur des réseaux de Petri suffisamment analysables pour supporter de manière satisfaisante la validation des systèmes spécifiés;
- Proteus et TRAPPER sont essentiellement des environnements de prototypage par programmation; ils traitent donc surtout les aspects liés à la production de programmes dans le domaine ciblé. Les aspects validation, réduits à de la simulation, sont moins bien couverts. Il est difficile de parler de modélisation dans le cas de Proteus puisque les programmes correspondent directement à une mise en œuvre du système. Le concept de modèle est encore plus réduit dans TRAPPER puisque le formalisme de haut niveau propose essentiellement un paradigme d'arrangement des processus séquentiels, directement écrits dans un langage de programmation;
- PARSE et CPN-AMI2 visent à couvrir de la manière la plus complète possible les trois aspects. Comme PEP, ils permettent une modélisation aisée (formalismes de haut niveau guidant l'utilisateur) tout en maintenant des

capacités de validation grâce à des transformations en réseaux de Petri. Ils proposent également de la synthèse de programmes.

PARSE et CPN-AMI2 se situent au même niveau de sophistication dans une démarche de prototypage. Cependant, le domaine d'applications réparties visées par PARSE est extrêmement vaste (des systèmes temps réels aux protocoles en passant par les bases de données réparties). De ce fait, un certain nombre de résultats ne sont pas généralisables. CPN-AMI, qui couvre un domaine d'application plus restreint, propose des solutions plus homogènes.

9. Conclusion

Dans ce chapitre, j'ai présenté MARS, notre démarche méthodologique pour la spécification et l'analyse d'applications parallèles, et CPN-AMI, l'environnement logiciel qui la met en œuvre.

La méthodologie MARS exploite les possibilités des réseaux de Petri sur les aspects analyse et vérification. Cependant, il existe un problème lié à ce formalisme dès qu'il s'agit de manipuler de grosses applications : complexité des modèles, structuration des modèles, difficulté d'interprétation des propriétés etc. Pour le contourner, nous encapsulons les réseaux de Petri afin de les rendre accessibles via des formalismes de plus haut niveau, à la manière de ce qui se fait dans des projets comme PEP (sur des aspects vérification), PARSE (sur la démarche), PROTEUS et TRAPPER (sur des aspects génération de programmes). On contrôle donc :

- La manière dont les réseaux de Petri sont synthétisés à partir de la spécification de haut niveau et selon des «partis pris» adaptés aux propriétés que l'on cherche à valider;
- L'architecture des applications produites à partir de telles spécifications qui se déduit des constructions offertes par les formalismes de haut niveau, l'objectif étant ici de guider les concepteurs d'applications parallèles.

CPN-AMI2, l'implémentation de MARS, a pour objectif d'offrir aux ingénieurs des outils adaptés à la conception de systèmes répartis exploitant la puissance des réseaux de Petri. L'originalité de MARS et CPN-AMI2 par rapport à d'autres approches et d'autres environnements consiste à mieux couvrir le cycle de vie du logiciel, depuis la spécification (avec un fort potentiel de vérification), jusqu'à la génération de programmes (avec exploitation de propriétés formelles pour optimiser les programmes générés). Une telle approche est rendue possible par une limitation du domaine d'application (les applications logicielles réparties sans contraintes de type temps réel).

L'expérimentation de MARS a permis de traiter des exemples qui n'auraient pas pu l'être directement avec les réseaux de Petri. Cependant, elle ne fait pas appel à des techniques spécifiques aux réseaux de Petri. On peut donc envisager d'étendre une telle démarche à d'autres méthodes formelles.

Chapitre 3
Prototypage d'Environnements de Génie Logiciel

10. Introduction

Dès 1987, notre équipe s'est intéressée à la notion de plate-forme d'accueil. A partir de 1989 un premier prototype était opérationnel : AMI **[23]** sur lequel nous avons construit CPN-AMI (AMI pour les réseaux de Petri). CPN-AMI a été diffusé à partir de 1993 sur Internet et a été utilisée par de nombreuses équipes universitaires. L'expérience acquise avec cet environnement de Génie Logiciel nous a amené en 1995 à élaborer FrameKit **[169, 172]** qui est donc une version plus puissante de notre plate-forme d'accueil.

Ce chapitre présente cette plate-forme d'accueil dédiée à la réalisation rapide d'environnements de Génie Logiciel. Nous poursuivons plusieurs objectifs :

- mettre en œuvre la méthodologie MARS (présentée dans le Chapitre 2) afin de l'exploiter dans le contexte d'applications de grande taille;
- exporter un savoir-faire sous la forme d'outils auprès de partenaires universitaires dans le cadre de projets communs;
- importer le savoir-faire de partenaires universitaires pour enrichir notre méthodologie de techniques et d'outils dont nous ne maîtrisons pas directement la théorie sous-jacente;
- disposer d'un environnement de prototypage d'Environnements de Génie Logiciel disponible pour des coopérations dans le cadre de projets communs.

Une plate-forme d'accueil permet la manipulation de modèles graphiques et hiérarchiques, l'application de services sur ces modèles et le stockage des résultats obtenus, et dispose de fonctions permettant une administration simple d'entités comme les utilisateurs, les services etc.

Un environnement produit à partir d'une telle plate-forme d'accueil est paramétré par des formalismes et des outils associés à ces formalismes (Figure 25). Ainsi, la création d'un Environnement de Génie Logiciel dédié à une méthodologie se résume à l'intégration des formalismes qu'elle requiert et des outils les manipulant.

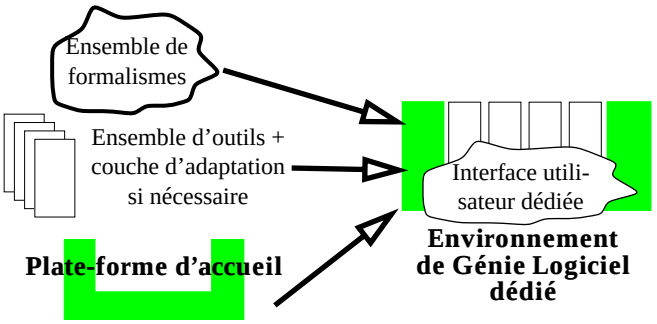

Figure 25 : Processus de production d'un environnement dédié à partir de la plate-forme.

Pour assurer ces différents objectifs, notre plate-forme d'accueil dispose :

- d'une interface utilisateur paramétrable (Macao **[207]**) pour étudier, mettre au point puis utiliser des formalismes graphiques et hiérarchiques,
- de bibliothèques de composants facilitant le développement d'applications dans le contexte de cette plate-forme afin d'offrir aux chercheurs un environnement de développement cohérent,
- de mécanismes d'intégration permettant d'importer des logiciels développés par d'autres,
- de techniques de développement multi-cibles pour assurer une bonne diffusion du «produit». Cette caractéristique doit concerner aussi bien la plate-forme elle-même que les composants offerts aux programmeurs d'outils,
- de mécanismes d'administration et de diffusion assurant une administration simple et proposant un gabarit pour faciliter la diffusion d'outils.

Nous identifions dans un premier temps nos besoins, avant de développer l'architecture des différents composants de FrameKit. Enfin, nous discutons de nos expérimentations avec la construction de CPN-AMI2, l'environnement de Génie Logiciel qui implémente la méthode MARS.

11. Environnements de développement d'AGL : besoins et choix

Au démarrage du projet FrameKit, nous avions identifié différents besoins qui seront détaillés dans les sections suivantes :

- Définir rapidement de nouveaux formalismes graphiques et hiérarchiques afin de les tester et de les finaliser le plus tôt possible;
- Intégrer rapidement des outils pour les tester et les évaluer. Différents niveaux d'intégration sont souhaitables : il est raisonnable qu'une intégration plus fine et plus coûteuse succède à une première intégration peu ergonomique mais destinée à évaluer l'intérêt du nouveau composant logiciel. Il faut également maintenir l'unité de l'environnement ainsi produit, son utilisation doit être la plus simple possible et ne pas requérir de con-

naissances autres que celles liées aux méthodes implémentées;

- Offrir des bibliothèques de fonctions pour le développement de nouveaux outils destinés à fonctionner dans FrameKit, pour faciliter le travail des développeurs;
- Mettre en place des mécanismes d'administration des entités de la plate-forme (une demande des usagers de CPN-AMI 1) comme la gestion d'utilisateurs, un procédé de mise à jour automatique (nouveaux outils, patches correctifs) etc.;
- Proposer des mécanismes pour la définition de standards propres à un ou plusieurs formalismes afin de faciliter l'échange et la réutilisation de données et / ou de résultats;
- Offrir des fonctions évoluées comme la possibilité de construire des générateurs de code, des outils de traduction entre représentations ou des simulateurs dédiés à un formalisme à partir d'un noyau couvrant une partie du travail;
- Disposer d'un modèle de diffusion pour la construction de composants cohérents distribuables via Internet (un outil, un environnement implémentant une méthode etc.).

A partir de ces besoins, nous avons complètement redéveloppé la plate-forme d'accueil. Macao, l'interface utilisateur de CPN-AMI1, a été étendue de manière à servir d'éditeur graphique et de frontal pour FrameKit.

11.1.Formalismes et modèles

Les représentations graphiques ou textuelles constituent des entités majeures pour un environnement de Génie Logiciel. Leur manipulation pose différents problèmes de visualisation, de stockage et de représentation. Le modèle de référence ECMA-NIST **[88]** définit la notion de services pour l'interface utilisateur (User Interface Services) pour identifier les fonctions réalisant l'interaction avec les usagers d'un Environnement de Génie logiciel. On y trouve la manipulation de représentations graphiques et / ou textuelles, la gestion des dialogues avec l'usager ou la sélection des actions à effectuer.

La notion de formalisme est liée à la représentation et à la sémantique d'informations, qu'il s'agisse de données en entrée ou de résultats calculés. Nous définissons un formalisme comme étant *l'ensemble des règles de présentation d'une description*. Une telle représentation peut se restreindre à un aspect visuel ou s'étendre jusqu'à la définition de grammaires (via leur BNF). La sémantique de ce qui est représenté est exprimée au moyen de conventions entre utilisateurs d'un formalisme.

Les réseaux de Petri, HOOD ou OMT sont des formalismes :

- Les réseaux de Petri **[154]** constituent un formalisme graphique doté d'une sémantique stricte mais simple, basée des objets graphiques et des informations de type textuel (référence à des variables et à des domaines de cou-

leurs, valuations d'arcs, etc.).

- HOOD **[143]** est un formalisme permettant de représenter la structure de programmes. Comme dans le cas des réseaux de Petri, il est composé d'entité graphiques représentant des éléments de programmes et utilise la hiérarchie comme un moyen d'augmenter la lisibilité des spécifications.
- OMT **[245]** est une combinaison de trois formalismes décrivant différents aspects d'une solution à un problème logiciel. Le modèle objet décrit la structure d'un système, le modèle dynamique permet de définir les règles dévolutions d'une entité de base (la classe) et le modèle fonctionnel explicite les flux d'informations entre les différentes entités du système.

Ces trois formalismes sont souvent représentés de manière graphique mais il en existe aussi des représentations textuelles(1). Une grammaire (par exemple, la BNF d'un langage de programmation) n'est pas autre chose qu'un formalisme. Les mécanismes de description proposé dans Macao et FrameKit traitent les représentations textuelles comme un cas particulier de représentations graphiques.

Nous pouvons maintenant définir un modèle comme étant une «instance» d'un formalisme, c'est-à-dire *l'expression d'une connaissance (spécification de la solution à un problème, expression de propriétés etc.) réalisée selon un formalisme.*

11.2.Services pour l'interface utilisateur

Les «services pour l'interface utilisateur» du modèle ECMA-NIST ont été diversement interprétés. En général, ils sont identifiés à un environnement de développement graphique normatif et prérequis à l'intégration des outils dans un environnement de Génie Logiciel. C'est le choix effectué dans le cadre du projet IRENA **[244]**. Les outils intégrés dans l'Atelier de Génie Logiciel délivré devaient exploiter les services de l'environnement graphique X-MOTIF afin d'être compatibles avec la plate-forme qui y était définie. Dans le même ordre d'idées, Ptolemy requiert des outils graphiques qu'ils utilisent des services basés sur tcl / tk **[238]**, ETI que l'interface se fasse au moyen d'un navigateur Internet **[197]** et PCIS exige des outils qu'ils utilisent les bibliothèques d'OpenLook ou MOTIF **[232]**. Dans le cas de PCIS, des règles sont cependant énoncées en vue d'uniformiser les interfaces utilisateurs des outils.

L'avantage d'un tel choix est qu'il repose sur des environnements sophistiqués et qu'il laisse toute liberté dans l'implémentation des outils. Cependant, les environnements graphiques ne sont en général pas suffisamment contraints : si l'on veut maintenir une certaine uniformité dans la réactivité des outils, il faut définir des règles d'usage. L'expérience du projet Ptolemy est intéressante : après s'être contenté dans un premier temps d'une boîte à outils basée sur X11 pour la construction d'interfaces utilisateurs **[221]**, un environnement de développement plus sophistiqué, Tycho **[145]**, a été conçu. Basé sur Java et tcl / tk, Tycho est dédié à la construction d'interfaces utilisateurs (graphiques ou non) dans l'envi-

(1) Ces descriptions textuelles servent en général de format standard pour l'échange d'informations.

ronnement Ptolemy. Outre des fonctions graphiques de base assurant la portabilité des interfaces ainsi produites, Tycho propose des «guides» (sous la forme de concepts architecturaux) que les développeurs doivent respecter. Ces guides ont pour objectif principal l'uniformisation de la réactivité.

Si cette approche offre des possibilités permettant de couvrir des besoins très différents (c'est le cas dans Ptolemy), il faut redévelopper un nouvel éditeur graphique pour chaque nouveau formalisme. Ce travail, long et coûteux, est inacceptable si on se place dans un contexte de prototypage d'Environnements de Génie Logiciel où il est intéressant de tester le nouveau formalisme dès que possible. Pour cela, une utilisation précoce (avec l'éditeur graphique approprié) est nécessaire alors même qu'il n'est pas encore stabilisé.

C'est pourquoi nous avons choisi de regrouper les services pour l'interface utilisateur dans un outil dédié : Macao **[207]**, conçu et développé par Jean-Luc Mounier. Macao est un éditeur graphique générique paramétré par la description du formalisme qu'il doit manipuler. L'intérêt majeur de cette approche est que la création d'un éditeur graphique pour un nouveau formalisme se réduit à la déclaration des entités le composant, ce qui constitue un travail beaucoup plus simple que la réalisation d'un nouvel éditeur.

11.3. Aide au développement d'applications

L'aide au développement d'applications dans une plate-forme d'accueil est importante pour l'enrichissement des environnements ainsi créés. Par exemple, Design/CPN et des environnements comme PEP ou Ptolemy proposent des techniques pour faciliter le développement par les utilisateurs.

L'approche de PEP est peu directive car cet environnement vise l'exploitation d'outils théoriques. Tout repose sur la définition d'un format interne d'échange délibérément extensible **[125]**. Ce format de représentation est lié à des fonctions de base sur les réseaux de Petri, regroupées dans la Petri Box Calculus. Un autre format de représentation intégrant des caractéristiques graphiques a été mis en place afin de centraliser l'accès aux services d'interfaçage avec l'utilisateur. Dans tous les cas, les communications entre composants sont effectuées au moyen de fichiers.

Ptolemy **[239]** est un environnement de prototypage de systèmes matériel/logiciels (codesign) développé à l'Université de Berkeley. Il offre une batterie impressionnante de fonctions évoluées destinées à faciliter le développement de nouveaux services. L'une des caractéristiques les plus intéressantes est la mise à disposition de bibliothèques graphiques et d'un guide de conception d'une interface utilisateur **[145]**. De nombreuses interfaces programmatiques sont également définies en C++ pour manipuler les entités gérées dans la plate-forme **[45]**.

Le point fort de Design/CPN est un mécanisme sophistiqué de «macro-fonctions» permettant d'exploiter très finement les possibilités de simulation de

l'outil : OA[(2)] **[202]**. Cependant, l'utilisation directe des fonctions d'OA est extrêmement complexe et des environnements de simulation ont été élaborés par dessus. Par exemple, Mimic/CPN **[241]** est une bibliothèque de fonctions permettant la manipulation d'objets graphiques. Elle permet d'afficher ou de cacher des connecteurs, déplacer ou aligner des zones d'écran, sauvegarder/restaurer/changer l'apparence d'objets. On peut également s'en servir pour contrôler l'animation de la spécification.

Dans FrameKit, la conception de nouveaux formalismes passe par la définition d'un jeu de paramètres pour l'interface Utilisateur Macao qui est générique et propose un modèle de réactivité type. C'est plus simple que l'approche de Ptolemy car aucune programmation n'est requise, mais cela se fait au prix d'une réactivité plus élémentaire.

Pour faciliter la manipulation des modèles, des interfaces programmatiques sont proposées dans différents langages de Programmation (à présent, Ada et C). La plate-forme décharge également les outils de la gestion de points importants comme la cohérence des résultats, le respect de séquences d'invocation des outils, la gestion des interruptions de services par un utilisateur, etc.

11.4.Intégration d'outils

Les Environnements de Génie Logiciel proposent aux utilisateurs une technique d'accès «universel» aux services qu'ils offrent. Ces services sont réalisés par des composants logiciels. L'opération d'intégration vise à assurer la correspondance entre ces outils et les services qu'ils apportent aux utilisateurs. L'opération d'intégration est définie pour toutes les architectures logicielles extensibles, qu'il s'agisse d'ECMA-NIST **[88]** pour les Ateliers de Génie Logiciel, de CORBA **[208]** pour les applications réparties ou TINA **[2]** pour les services de télécommunications.

Selon **[283, 274, 232]** , l'intégration d'outils dans un environnement complexe doit tenir compte de cinq «axes» d'intégration :

- *Intégration au niveau des données* : il s'agit de préciser des mécanismes pour leur stockage (où sont conservées les données échangées entre composants logiciels) et leur représentation (comment un composant logiciel peut-il «comprendre» des informations qu'il n'a pas produites);
- *Intégration au niveau de la présentation* : il s'agit de normaliser l'apparence et le comportement («look and feel») des composantes d'un environnement, principalement pour assurer une plus grande facilité d'utilisation;
- *Intégration au niveau du contrôle* : il s'agit d'uniformiser les mécanismes permettant de stimuler les fonctions d'un outil en vue de favoriser la coopération entre composants d'un environnement;
- *Intégration au niveau procédé* : il s'agit de définir de manière externe (i.e. non codée dans l'environnement) les règles imposées par une méthode;

(2) OA est l'acronyme de «Meta Software's Open Architecture».

- *Intégration au niveau plate-forme* : il s'agit d'encapsuler les services de «bas niveau» offerts par le système d'exploitation.

Certains environnements offrent une vision partielle, mais souvent intéressante de l'intégration. C'est le cas de Bast **[107]** (développé à l'Ecole Polytechnique Fédérale de Lausanne) dédié au développement de systèmes répartis résistants aux pannes. Le principe d'utilisation de Bast repose sur l'assemblage de composants logiciels selon des gabarits de conception **[105]**. L'environnement est vu ici comme un ensemble de bibliothèques proposant l'implémentation de mécanismes types adaptés à la construction de systèmes répartis. L'intégration dans Bast repose sur les possibilités du langage Smalltalk (une implémentation en Java est en cours de réalisation) et exploite la portabilité au niveau du langage **[106]**.

Sur des bases plus proches du modèle ECMA-NIST, les environnements PEP et Ptolemy basent l'intégration sur l'aide au développement d'outils offerte par un environnement de programmation, quitte à en privilégier certains aspects. Ptolemy se focalise sur les aspects données, présentation et plate-forme alors que PEP privilégie plutôt les aspects données et plate-forme, l'homogénéité des représentations graphiques utilisées permettant de réduire l'intégration par la présentation à son strict minimum.

Dans FrameKit, nous avons pris en compte les cinq axes d'intégration en privilégiant les fonctions permettant le prototypage d'Environnements de Génie Logiciel. Nous interprétons ces axes en leur adjoignant des hypothèses fortes permettant d'en simplifier l'interprétation :

- Au niveau présentation, la définition d'une interface utilisateur générique, si elle constitue une vision extrêmement stricte de l'axe tel qu'il était initialement identifié dans ECMA-NIST, permet d'avoir une réactivité commune pour tous les outils, même si cela se fait au prix de quelques simplifications;
- Au niveau données, la définition de services minimaux d'accès et de représentation simplifie considérablement la tâche d'analyse et de communication, même si cela se fait au prix de possibilités plus restreintes (par exemple, se restreindre à l'utilisation de mécanismes de représentation de données structurées élémentaires);
- Au niveau contrôle, le choix d'associer une «ligne de commande» (au sens d'un système d'exploitation) à chaque service permet de simplifier le modèle d'intégration, même si cela écarte les applications ne pouvant être invoquées sous cette forme (i.e. les applications qui ne peuvent être dissociées de leur interface utilisateur);
- Au niveau procédé, le choix de s'appuyer sur des mécanismes de droits d'accès (utilisateurs, groupes d'utilisateurs) et d'ordonnancement des services offre des possibilités satisfaisantes, même si la notion «d'aide intelligente à l'utilisateur» en est complètement absente;
- Au niveau plate-forme, on choisit de se ramener à une encapsulation impli-

cite des services du système d'exploitation **[274]**. Cette encapsulation se fait via des interfaces programmatiques et se trouve «diluée» dans les autres axes.

11.5. Standardisation des données/résultats dans une plate-forme

Tous les outils complexes (composés de plusieurs programmes coopérants) et les environnements de Génie Logiciel s'appuient désormais sur une standardisation des échanges de données et de résultats **[208]** (parfois posés sur un système de description de données structurées, les metadonnées dans **[191]**). Par exemple, les différents exécutables composant GreatSPN **[60]** partagent un même format d'entrée pour les données et les résultats. Le programme assurant les fonctions de l'interface utilisateur exporte les données en entrée dans ce format et visualise les résultats qu'il récupère après invocation du module spécifique assurant la fonction sélectionnée (la communication se fait par fichiers). Ce format standard est ici un élément majeur du mécanisme de communication entre les différents modules de cet outil complexe.

Le fonctionnement de l'environnement SEA **[139]** repose également sur un format standard utilisé à d'autres fins. Il s'agit ici d'une forme «canonique» de représentation interne d'un système (les réseaux de Petri Prédicat/Transition) exploitée pour exprimer des spécifications dans une classe plus élaborée de réseaux de Petri. Ce standard est exploité par une batterie de fonctions de simulation et d'animation de spécifications décrites dans des formalismes de plus haut niveau.

PEP **[28]** s'inspire à la fois de SAE et de GreatSPN. Une représentation de base : les LL-Nets **[26]** (réseaux de Petri Place/Transition 1-bornés) sert à la fois de «langage d'assemblage» mais aussi de base pour l'exportation vers d'autres formats de représentations définis pour des outils importés (INA, PROD et SMV).

AMI **[23]** dispose d'un langage d'échanges dédié (CAMI) qui joue le rôle de format d'échange standard. Il s'agit du seul format manipulé par la plate-forme d'accueil et l'interface utilisateur.

Cependant, l'usage des LL-Nets dans PEP ou de CAMI dans AMI pose des problèmes d'ordre différents :

- S'il est bien adapté à l'expression d'une large classe de réseaux de Petri, LL-Net est extrêmement lié aux fonctions offertes par la Petri Box Calculus. Cela peut rendre délicat l'évolution du standard de représentation comme celui de la Petri Box Calculus.
- CAMI est si générique qu'il épouse parfaitement toute forme de représentation, mais de manière souvent inadaptée par rapport au formalisme (par exemple, une représentation en matrice creuse serait plus adaptée pour les réseaux de Petri). Ici, ce n'est pas l'adaptation fine du langage à son utilisation qui est à l'origine du problème mais au contraire son côté polymorphe.

Citons également les travaux du groupe de travail CoFI (The Common Framework Initiative) qui s'intéresse à l'interopérabilité d'outils travaillant sur des spécifications algébriques **[67]**. L'objectif est de disposer d'une solution technique sous la forme d'un langage permettant de décrire les modèles soumis aux outils et les résultats qu'ils fournissent **[32]**. Des expérimentations dans le cadre du projet SALSA ont été menées entre ASSPEGIQUE+ **[61, 31]** et LP **[108]**.

Il est impossible d'évoquer les standards de représentation des données sans parler d'IDL **[150]**, mais ce langage est trop lié à des structures de données (donc, des choix d'implémentation). Dans le cas de notre problème, quelle représentation devrions-nous choisir pour exprimer nos formalismes? On observe le même problème avec le langage EXPRESS **[149]**, défini pour le standard STEP (ISO-10303). Ce langage dont la philosophie est voisine de celle d'IDL permet de décrire des structures de données **[8]**. L'une de ses caractéristiques est la possibilité de définir des algorithmes de vérification de contraintes sur ces données (par exemple, qu'une liste circulaire reste toujours circulaire).

L'expérience montre qu'il est difficile pour une plate-forme d'accueil d'offrir un standard d'échange générique et adapté à tous les usages. Il est alors intéressant d'associer à ce format d'échange une bibliothèque de fonctions adaptée à certaines formes de représentations. Un bon exemple de cette stratégie est la mise en place de formalismes(3) prédéfinis dans Ptolemy, comme le «Synchronous Dataflow Domain» pour lesquels des bibliothèques optimisées reposent sur les mécanismes de base de l'environnement **[54]**. Ce travail se justifie pour des formalismes fortement utilisés dans les domaines d'application couverts par le projet.

Cette observation nous a conduit à proposer deux niveaux de standardisation dans FrameKit : primaires (c'est-à-dire propres à la plate-forme) et secondaires (liés à un formalisme). Les standards primaires sont constitués d'une version évoluée du langage CAMI pour tout ce qui est échange de données et expression des résultats. Les standards secondaires définissent les conventions devant être respectées par tous les outils associés à un formalisme comme :

- Les formats de représentation des modèles. Ainsi, on peut factoriser la vérification des modèles en la confiant à un outil dédié acceptant du CAMI en entrée et produisant le format adapté, directement exploité par les outils qui ignorent alors CAMI;
- Les formats de représentation des résultats, ce qui permet à des outils d'exploiter les résultats produits par d'autres;

La réalisation de standards secondaires relève d'une démarche similaire à celle du groupe CoFI avec les spécifications algébriques puisqu'il s'agit de proposer des formats et des mécanismes d'échange dédiés à un formalisme. De même, l'Université Humbolt (Berlin) a récemment proposé pour les réseaux de Petri un

(3) Dans Ptolemy, les formalismes sont appelés «domaines».

format de représentation auquel sont associés des librairies de manipulation **[158]**.

Dans FrameKit, la mise en œuvre des standards secondaires repose sur les principes proposés par ***M***eta***S***cribe, un générateur de moteurs de transformations (un traducteur d'un format vers un autre). ***M***eta***S***cribe sera présenté en Section 12.3.

11.6. Éléments pour le développement d'outils complexes

Pour que FrameKit devienne un outil de prototypage d'environnements de Génie Logiciel, nous travaillons à la réalisation de fonctions de haut niveau permettant la mise en œuvre d'outils complexes. Nous visons actuellement deux directions : la génération d'applications et la simulation / déboguage de spécifications.

La génération d'applications est principalement une transformation d'une spécification en entrée en un programme source. On dégage délibérément deux aspects du problème : la réalisation de la transformation elle-même (i.e. les règles de transformation) et l'expression syntaxique du résultat (i.e. le langage cible). Si les règles de transformations sont suffisamment dissociées des règles d'expression du résultat, cela favorise la réutilisabilité. L'objectif est de pouvoir ainsi définir des gabarits de transformations réutilisables. Nous verrons en Section 12.3. comment ***M***eta***S***cribe facilite la construction d'un générateur de programmes.

Un simulateur débogueur peut se diviser en deux éléments distincts : la gestion des fonctions de base (exécution pas à pas, exécution jusqu'à un point d'arrêt, pose de points d'arrêts etc.) et un «moteur d'exécution», c'est-à-dire l'implémentation des règles d'exécution du formalisme concerné. Des travaux prospectifs sont en cours.

12. FrameKit

L'architecture de FrameKit est indiquée en Figure 26. L'utilisateur accède à l'environnement via une interface générique : Macao. Elle permet l'édition de modèles graphiques et hiérarchiques et se comporte également comme le frontal de la plate-forme d'accueil en centralisant tous les accès. La plate-forme d'accueil offre les services de base d'un environnement de Génie Logiciel, c'est-à-dire l'administration (gestion des utilisateurs, des services des droits d'accès, de la cohérence des données sur lesquelles travaillent les outils etc.) et l'interaction avec des outils.

Les outils ne font pas partie de FrameKit : ils offrent un ensemble de services caractérisant un environnement dédié. La construction de CPN-AMI2 en est l'illustration : nous avons déclaré dans FrameKit un ensemble de formalismes et d'outils (certains provenant d'autres universités) afin de spécialiser une «instance» dédiée là la mise en œuvre de la méthode MARS.

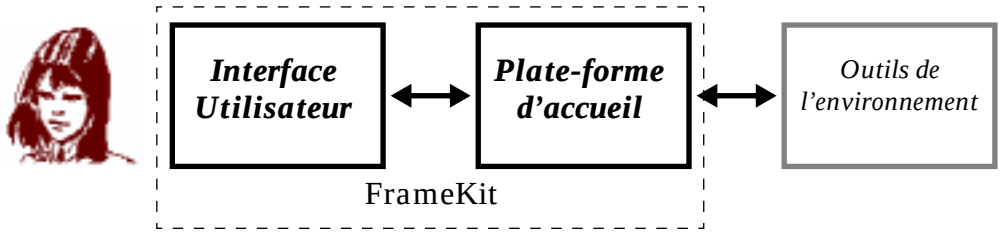

Figure 26 : Composantes de la plate-forme FrameKit.

Meta***S***cribe est un générateur de moteurs de transformations permettant de produire facilement des outils de transformation de résultats vers différents formats cible, mais aussi de faciliter la construction de générateurs de programmes. Cet outil produit des moteurs de transformation utilisant les interfaces programmatiques de FrameKit; ils sont donc directement utilisables dans cet environnement.

Un modèle générique de simulateur, paramétré par un «moteur d'exécution» propre à un formalisme, facilite la mise au point de nouveaux simulateurs. L'objectif est de pouvoir disposer à faible coût d'un moyen d'observer le comportement d'un formalisme «exécutable».

12.1.L'interface utilisateur

Outre sa fonction d'éditeur de graphes, Macao est également le frontal de FrameKit : les opérations d'édition, connexion, accès aux services et affichage des résultats sont assurés par le même logiciel. Ainsi, dès que l'éditeur associé à un nouveau formalisme est opérationnel, on peut déclarer ce formalisme dans la plate-forme et y intégrer des outils sans développement supplémentaire.

12.1.1. Formalismes et modèles dans Macao

L'utilisation de techniques objets se prête bien à la représentation de formalismes et modèles graphiques **[68]**. A ce titre, dans Macao, la création d'un formalisme se résume à la déclaration des différentes classes d'entités qui le composent. Toute classe d'entité déclarée est soit un nœud, soit une boîte, soit un connecteur. Chaque classe d'entités est caractérisé par un identificateur et contient des attributs (informations textuelles associées aux instances).

Les connecteurs permettent de relier les boîtes et nœuds entre eux. On peut définir des règles de connectivité dans le formalisme (par exemple, un connecteur de type «arc inhibiteur» relie un nœud de type «place» à un nœud de type «transition»).

Les boîtes sont des nœuds qui représentent un lien vers une *page*. Une page correspond à un sous-ensemble de la spécification et peut être caractérisée par un formalisme différent. La Figure 27.a illustre ce mécanisme. Dans une page «supérieure», une boîte référence une page «inférieure» contenant une partie de la description.

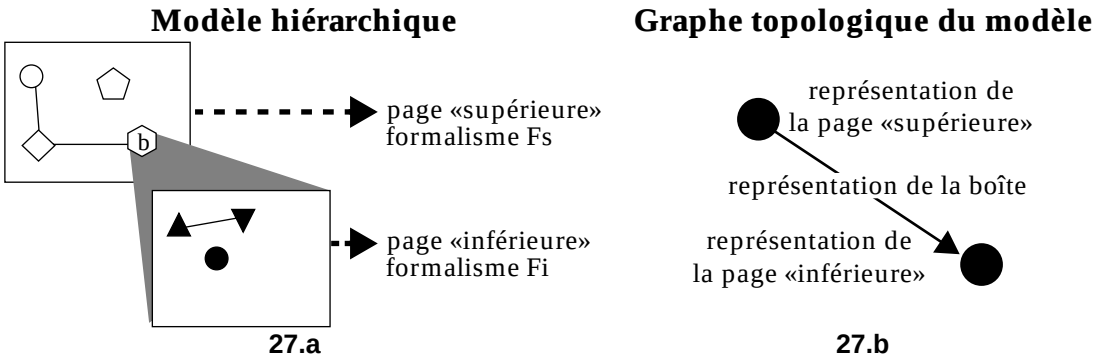

Figure 27 : Organisation des modèles hiérarchiques.

Un formalisme hiérarchique est donc vu par Macao comme un assemblage de formalismes. L'un de ces formalismes est dit «racine»; il correspond au «point d'entrée» pour l'usager lorsqu'il crée un modèle. Par exemple, H-COSTAM est la composition de deux formalismes, l'un correspondant au niveau macro, l'autre au niveau micro. Le formalisme racine est celui du niveau macro.

Pour les formalismes hiérarchiques, Macao gère à la fois la description des différentes pages qui composent le modèle mais aussi un *graphe topologique* (Figure 27.b) décrivant son organisation hiérarchique.

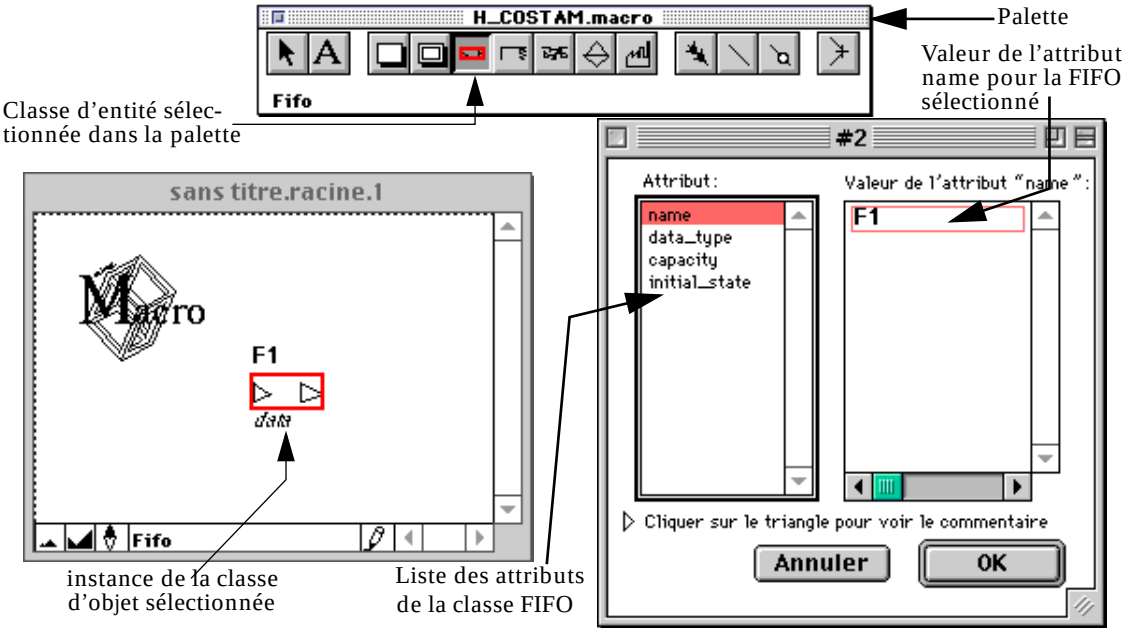

Figure 28 : Visualisation d'un formalisme dans Macao.

En tant qu'éditeur de graphes, Macao considère à la fois les aspects déclaratifs (les classes d'objets-nœuds, objets-boîtes ou d'objets-connecteurs et les attributs qui leur sont associés) et esthétiques (leur forme). Les attributs des classes d'objets dans un formalisme sont considérés par Macao comme des chaînes de caractères sur lesquelles aucune vérification de conformité n'est effectuée (par exemple, le respect d'une grammaire), ce travail étant effectué par l'invocation

d'un outil intégré dans la plate-forme. FrameKit permet d'attribuer un statut particulier à un tel outil afin de rendre transparente son invocation lors d'une demande de service.

La Figure 28 illustre, à travers la manière dont les formalismes sont visualisés dans Macao, le mécanisme de description sous-jacent. La palette (en haut) propose à l'utilisateur les classes d'entités potentiellement utilisables dans le formalisme. Des instances de ces classes sont ajoutées dans une fenêtre de modèle (en bas à gauche). Les attributs de chaque instance de classe sont remplis via une fenêtre de saisie (en bas à droite) qui permet l'édition des attributs déclarés pour la classe correspondante. Ici, l'objet sélectionné, de type FIFO, comporte quatre attributs (name, data_type, capacity, initial_state). La valeur de l'attribut name est «F1».

12.1.2. Macao en tant que frontal de FrameKit

Outre ses fonctions d'éditeur de graphes, Macao permet d'accéder à FrameKit. Dans ce mode, un certain nombre de services supplémentaires sont disponibles pour les outils et la plate-forme. Ces services peuvent se classer comme suit :

- mécanismes de connexion,
- manipulation des services accessibles à l'utilisateur,
- manipulation de boîtes de dialogue,
- transmission de résultats.

Les mécanismes de connexion permettent à Macao et FrameKit d'échanger l'identité d'un utilisateur ainsi que les informations concernant les modèles qu'il a ouverts. La plate-forme, sur la base de ces informations, peut alors identifier les services accessibles par cet utilisateur pour ces formalismes.

Macao dispose d'un mécanisme de sélection de services (au moyen de menus hiérarchiques). Il permet à FrameKit de créer des entrées, de les masquer (le service est temporairement inaccessible) et les démasquer.

Les services de manipulation de boîtes de dialogues ou de transmission des résultats sont destinés aux outils. Ils leur permettent d'interroger l'utilisateur et de présenter ses résultats, que ce soit de manière textuelle (affichage des résultats dans une fenêtre d'historique) ou graphique (par exemple en créant un nouveau modèle ou en désignant des objets dans le modèle traité).

Macao et FrameKit communiquent au moyen d'un canal de communication véhiculant des messages au format CAMI. CAMI est un langage de description divisé en plusieurs sous-groupes, ayant des objectifs différents : description des services disponibles, activation de dialogues types, transmission des résultats, connexion, transmission des modèles et invocation de services. Le sens de transmission des commandes appartenant à ces différents sous-groupes est indiqué dans la Figure 29.

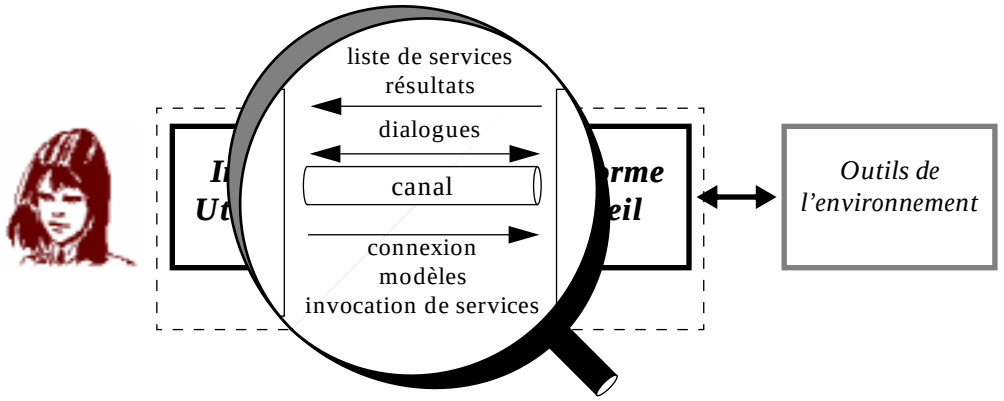

Figure 29 : Interactions entre l'interface utilisateur Macao et la plate-forme d'accueil FrameKit.

12.1.3. Formalismes et modèles dans FrameKit

Pour FrameKit, les formalismes sont vus comme des identificateurs permettant de les référencer et leur associer des services. La plate-f7orme doit être «neutre» vis à vis des formalismes et des modèles qu'elle manipule. A cette fin, nous distinguons les informations *esthétiques*, *syntaxiques* et *sémantiques*.

Les informations d'ordre esthétique concernent l'interface utilisateur et sont paramétrables : que les transitions d'un réseau de Petri soient représentées par un carré ou un rectangle ne changent rien à leur signification (il ne s'agit que d'une convention graphique permettant de les identifier facilement dans un schéma). Ainsi, l'apparence des objets composant un formalisme n'est gérée qu'au niveau de l'interface utilisateur. Chaque classe d'objets d'un formalisme doit avoir une apparence type, éventuellement modifiable au niveau d'un modèle (par exemple, la taille).

Les informations d'ordre syntaxique correspondent à l'expression d'une connaissance dans le formalisme associé et doivent être véhiculées de l'interface utilisateur vers les outils. Le support de cette information est polymorphe puisque ni l'interface utilisateur ni la plate-forme ne doivent être modifiés lors de la création d'un nouveau formalisme. Pour cela, nous utilisons une description à base de messages : CAMI-LDM[(4)] **[170]**. CAMI-LDM permet de véhiculer la description d'un modèle sous la forme de messages construits par Macao, transmis par la plate-forme et exploités par les outils.

Une telle solution est voisine de celle adoptée pour Tycho **[145]** dans lequel TIM (Tycho Information Model) propose un meta-modèle de description des formalismes d'où un format d'échange est déduit pour transmettre des modèles. Cette solution résout à la fois le problème de la spécification d'un formalisme et celui de la transmission des modèles. On couvre à la fois les aspects intégration par la présentation et intégration par les données selon la classification de Wasserman

(4) CAMI-LDM est le sous-ensemble du langage CAMI dédié à la description d'un modèle. Il est également identifié sous l'acronyme LSE (Langage de Spécification des Enoncés) dans la thèse de K.Foughali **[103]** .

[283] et de PCIS **[232]**. Elle offre également les «bonnes» caractéristiques identifiées pour PEP dans **[125]** : structuration, extensibilité et flexibilité.

Enfin, l'aspect sémantique d'un formalisme réside dans la mise en place de conventions entre utilisateurs et outils. Elle est exprimée implicitement dans la définition des entités qui composent le formalisme (par exemple, une classe dans OMT a une sémantique très précise interprétée de la même manière par l'outil et ses utilisateurs).

12.2.La plate-forme d'accueil et ses outils d'administration

La plate-forme d'accueil est constituée d'un ensemble de programmes et de bibliothèques devant assurer l'invocation de composants logiciels (les outils), leur transmettre des données, et exploiter leur résultats (récupération, stockage etc.). Les services d'administration (comme la gestion des usagers et des groupes, l'installation / mise à jour d'outils ou de composants de la plate-forme, etc.) sont réalisés au moyen d'outils spécialisés accessibles aux administrateurs.

12.2.1. Architecture de la plate-forme d'accueil

L'un des choix principaux dans la mise en œuvre de FrameKit est d'identifier les outils sous la forme d'un programme à invoquer («ligne de commande» du système d'exploitation cible). Ce choix impose les contraintes suivantes :

- les fonctions d'interface utilisateur des composants logiciels que nous intégrons doivent pouvoir être «déconnectées», Macao étant la seule interface de FrameKit;
- le système d'exploitation cible doit offrir des mécanismes de gestion des processus utilisé par le broker (au sens CORBA du terme) de FrameKit.

Ainsi, chaque service activé via l'interface utilisateur est transformé, pour invocation, en une ligne de commande comprenant des paramètres de gestions propre à FrameKit (transmettant des informations sur l'état de la plate-forme) et les paramètres de l'outil. Le langage de programmation des outils importe peu : l'important est qu'ils utilisent, pour communiquer, le standard de communication de FrameKit (CAMI).

Pour accéder aux services offerts par un environnement, les usagers doivent s'identifier. Ainsi, la plate-forme supporte la gestion de connexions et de sessions. Une connexion correspond à la durée pendant laquelle un usager authentifié accède aux services via la plate-forme. Pendant une connexion, il applique des services sur des modèles. Une session est constituée d'une suite d'invocations de services sur un modèle et peut se dérouler sur plusieurs connexions. Une session prend fin : 1) lorsque le modèle est détruit, 2) lorsqu'il est modifié (une nouvelle session est instantanément recréée). Dans les deux cas, toutes les informations concernant l'évolution de la session sont détruites.

La plate-forme d'accueil est composée de deux programmes (Figure 30) : un *serveur de connexion* et un *serveur de session*. Le serveur de connexion assure la gestion d'une connexion. Il authentifie l'usager lorsqu'il se connecte puis pilote les

gestionnaires de sessions pour créer, reprendre, suspendre et détruire les sessions. Le serveur de session prend en charge une session. A la création, il doit élaborer son état initial composé de la liste des services accessibles, calculée en fonction de l'identité de l'utilisateur et du formalisme du modèle associé. Il maintient ensuite l'état courant de la session (certains services peuvent n'être invocables qu'une seule fois ou requérir au préalable l'invocation d'autres services). Lors de l'invocation d'un service, il calcule la commande correspondante, exécute l'outil et dialogue avec lui (échanges avec l'utilisateur, émission des résultats, etc.). Enfin, il assure la destruction des données relatives à la session en cas de terminaison (la modification d'un modèle entraîne la destruction de tous les résultats préalablement calculés qui deviennent obsolètes).

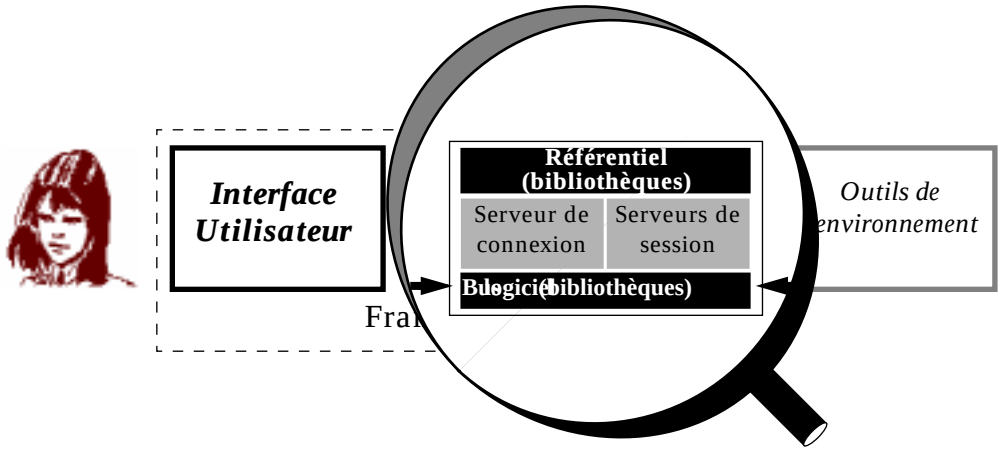

Figure 30 : Détail des éléments de la plate-forme d'accueil FrameKit.

Pour chaque usager connecté, FrameKit crée un serveur de connexion et un serveur de session par session ouverte. Les différents processus communiquent via un bus logiciel et exploitent des informations stockées dans le référentiel de FrameKit. Le bus logiciel et le référentiel sont constitués de bibliothèques encapsulant les fonctions offertes, l'objectif étant d'assurer leur portabilité.

Les modèles et les résultats obtenus par application des services sont décrits au moyen d'objets persistants stockés dans le référentiel. Le mécanisme utilisé est semblable à celui (basé sur PCTE) proposé pour les Ateliers de Génie Logiciel étudiés dans **[87, 191]** .

12.2.2. Mécanismes d'intégration des outils dans FrameKit

Nous distinguons deux types d'intégration. L'intégration *a priori* s'applique à un outil développé spécialement pour fonctionner dans FrameKit. Au contraire, l'intégration *a posteriori* vise les outils développés hors de cet environnement.

Enregistrement d'un service

Les deux types d'intégration (a priori et a posteriori) ont en commun une phase de déclaration dans la plate-forme qui définit les informations permettant la construction du menu de service transmis par le serveur de session à l'interface utilisateur. Ces informations permettent d'associer une signature aux services.

Le terme signature regroupe ici l'association avec l'outil correspondant (vision du broker de CORBA **[220]**) et un typage en fonction des formalismes en entrée et en sortie (au sens du langage GDMO **[148]**). Ce mécanisme simple supporte cependant la polymorphie(5) **[152]** et la composition de services **[219]**.

Les informations décrivant un service sont stockées dans des fichiers de configuration contenant :

- le nom externe du service (dans différents langues) qui sera transmis tel quel à l'interface utilisateur;
- le nom interne du service (indépendant de la langue de travail) ;
- le nom du fichier exécutable à invoquer et les paramètres à transmettre lors de l'invocation afin que le serveur de session puisse construire la ligne de commande d'invocation de l'outil. Il est possible de lier les paramètres à des options dans le menu de services; il ne seront alors transmis que si l'option est sélectionnée;
- les droits d'accès (par utilisateur, groupe ou architecture cible(6)) qui sont évalués de manière statique lors de la première connexion de l'utilisateur afin de déterminer la liste des services auxquels il a droit;
- les conditions de service, permettant d'interdire temporairement l'accès à un service (par exemple, soumettre son invocation à celle d'un autre service ou en fonction de la valeur de «variables de sessions» positionnées par les outils).

Les services peuvent être regroupés dans des sous-menus en vue d'obtenir une hiérarchie cohérente pour l'accès par l'utilisateur. Les menus de services sont construits par morceaux, à partir d'une liste de descriptions associée à un formalisme (les services s'appliquent à des modèles et sont donc liés à un formalisme). La mise en œuvre de l'enregistrement des services de FrameKit est détaillé dans **[174]**.

Structure d'un outil développé pour FrameKit

Pour faciliter le développement des outils et cacher les mécanismes sous-jacents en vue de faciliter leur évolution ainsi que leur développement sur plusieurs cibles, FrameKit propose trois interfaces programmatiques qui assurent :

- la communication avec l'interface utilisateur au moyen de messages transitant à travers le bus logiciel (dialogue avec l'utilisateur, transmission de résultats etc.);
- l'accès aux données stockées dans le référentiel de FrameKit. Pour assurer la transparence de l'accès aux données, un système de référence spécifique permet l'accès à différents espaces de travail (au sens de COO dans **[118]**) dans le référentiel au moyen de clefs construites dynamiquement pendant l'exécution. Trois modes de persistance sont proposés :
  - les données liées à un modèle (par exemple, des résultats calculés par

(5) Les services polymorphes sont applicables à plusieurs formalismes.

(6) Utile lorsqu'un outil n'est pas disponible sur toutes les architectures cible de distribution.

application de services) qui disparaissent lorsque ce dernier est détruit ou modifié;

- les données liées à un utilisateur (par exemple, des préférences d'utilisation d'un outil) qui sont accessibles à cet utilisateur indépendamment du modèle sur lequel il applique un service;
- les données globales à la plate-forme (par exemple, des bibliothèques partagées) qui sont accessibles à tout utilisateur.

On trouve aussi dans cette interface les fonctions de manipulation de modèles de «bas niveau», c'est-à-dire encapsulant la stratégie de description au format CAMI, ce qui permet au concepteur d'application de ne pas connaître ce langage de description.

- l'accès aux mécanismes de communication évolués de la plate-forme (invocation d'un autre outil, encapsulation de fonctions rendues par le système d'exploitation etc.). Cette interface programmatique intègre, entre autres, un modèle de coopération client / serveur avec imbrication des transactions (ou invocation de services) similaire à ce qui est proposé dans COO **[51]**. Ainsi, l'outil invoqué ne sait pas s'il est directement connecté à l'interface utilisateur (un humain a invoqué le service correspondant) ou si l'invocation est effectuée par un autre outil.

La Figure 31 présente la structure attendue d'une application dans FrameKit. Le développeur d'un outil n'a plus qu'à se consacrer à l'aspect algorithmique du problème. Un ensemble de bibliothèques de base (principalement la gestion de structures de données complexes) est également à la disposition du programmeur. Notons également que le programme principal («main») fait partie de la bibliothèque de FrameKit. Ainsi, les procédures d'initialisation et de terminaison d'un outil développé au moyen de ces bibliothèques de fonctions sont prises en charge par l'environnement avec souplesse.

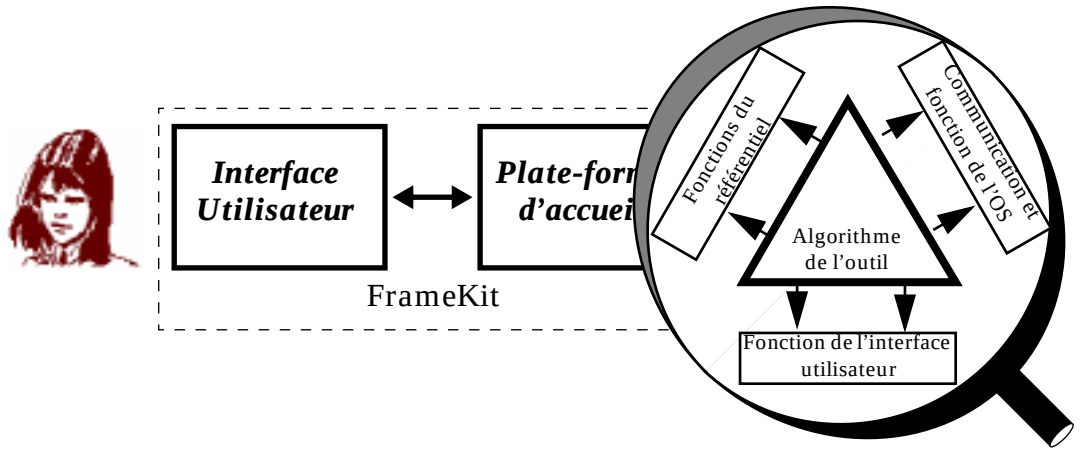

Figure 31 : Architecture d'un outil développé pour fonctionner dans FrameKit.

Structure d'un outil intégré dans FrameKit

Les outils que nous pouvons intégrer a posteriori dans FrameKit doivent pouvoir être «déconnectés» de leur interface utilisateur. Ils sont ainsi vus comme des éléments logiciels échangeant des informations avec la plate-forme d'accueil, principalement au moyen de fichiers contenant des informations (la description des modèles de travail et des résultats calculés) décrites avec le langage CAMI. Pour comprendre ce standard, l'outil ainsi intégré doit être associé à un *pilote* qui traduira les informations en entrée (au format CAMI) dans le format requis par l'outil et inversement.

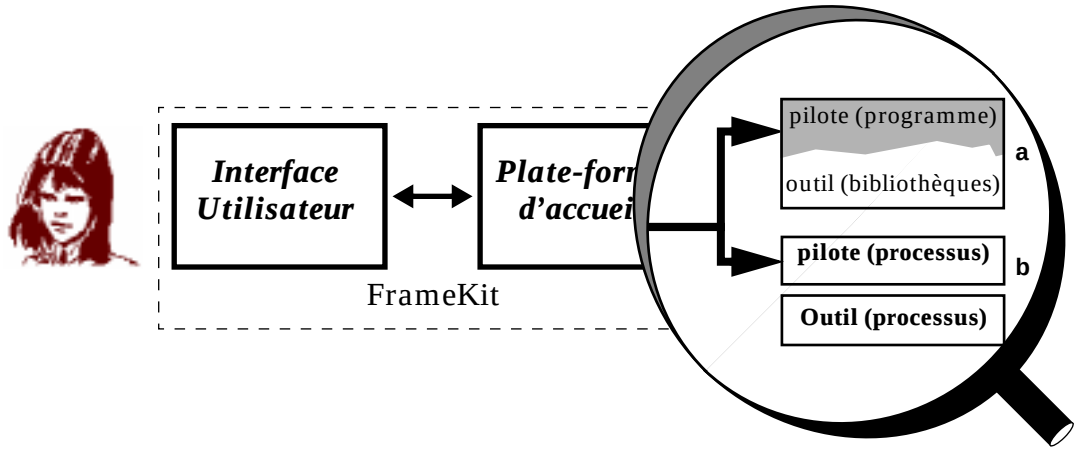

Figure 32 : Architecture d'un outil intégré a posteriori.

Deux visions de ce pilote, illustrées dans la Figure 32, sont considérées :

- la première (Figure 32.a) est applicable si nous disposons de l'outil sous la forme de fichiers sources ou de bibliothèques de fonctions. Le pilote est alors un programme qui fait appel aux fonctions de l'outil; la communication se fait en général au moyen de structures de données dans le segment de données de la tâche ainsi constituée;
- la seconde (Figure 32.b) suppose que nous ne disposons que du fichier exécutable et des informations sur les formats en entrée et en sortie. Le pilote est alors une tâche indépendante de l'outil qui est considéré par la plate-forme comme l'exécutable à invoquer lorsque le (ou les) service(s) correspondant(s) est (sont) invoqué(s). Comme précédemment, le pilote traduit les informations dans un format traitable par l'outil, invoque celui-ci puis retraduit les informations qu'il a calculées. La communication entre le pilote et l'outil peut s'effectuer via n'importe lequel des mécanismes offerts par le système d'exploitation cible.

Les interfaces programmatiques de FrameKit

Qu'il s'agisse d'un outil développé spécialement ou du pilote d'une application intégrée a posteriori, on se ramène donc toujours au développement d'éléments respectant les caractéristiques énoncées pour fonctionner dans FrameKit. Il est

ainsi aisé de s'appuyer sur les interfaces programmatiques de FrameKit qui ont été développées dans différents langages de programmation :

- Ada, qui est le langage de développement de la plate-forme,
- C, qui est le langage de développement traditionnel dans le monde Unix,
- Shell, qui s'est avéré extrêmement pratique pour effectuer rapidement un travail d'intégration préliminaire afin d'évaluer l'intérêt de l'outil avant d'investir sur des mécanismes plus complexes. Avec l'expérience, shell s'est également avéré un bon langage pour intégrer des outils conçus pour des environnements ayant une philosophie similaire à celle de FrameKit (par exemple, certains modules issus de PEP et GreatSPN).

12.2.3. Administration d'un environnement construit sur FrameKit

Pour être opérationnelle, la plate-forme d'accueil doit maintenir un ensemble de données d'administration (description des utilisateurs, description des services et formalismes déclarés etc.). L'accès à ces données au moyen d'outils dédiés permet de faciliter l'administration de l'ensemble et, en particulier, d'en assurer la transparence. Ce point correspond à un problème rencontré dans la diffusion de CPN-AMI 1, AMI ne disposant pas de tels mécanismes.

Les outils d'administration de FrameKit assurent :

- la gestion des utilisateurs,
- la gestion des services,
- la gestion des données internes de la plate-forme.

Dans leur conception et leur mode de fonctionnement, les outils d'administration de FrameKit ne diffèrent en rien des autres outils (en particulier, ils respectent l'architecture présentée en Figure 31). Les services qu'ils offrent sont regroupés dans un menu d'administration qui n'est accessible qu'aux utilisateurs effectuant l'administration de la plate-forme.

Gestion des utilisateurs dans FrameKit

Nous regroupons sous cette dénomination la gestion de l'ensemble des données propres à un utilisateur. Ces données permettent de les identifier et de les regrouper en «classes» auxquelles des droits particuliers peuvent être accordés. Cela constitue ainsi un moyen de différencier des «rôles» au sens donné par ECMA -NIST. C'est par exemple le cas avec les services d'administration qui ne sont accessibles qu'aux membres du groupe «Admin»[(7)].

Ce service permet la définition des données caractérisant un utilisateur (nom, mot de passe, espace de stockage de ses données de travail) leur regroupement potentiel en groupes d'utilisateurs et l'accès à une messagerie dédiée aux messages d'administration.

(7) Ce groupe est une convention. Il ne peut être détruit et est automatiquement créé à l'installation.

Gestion des services dans FrameKit

Nous regroupons sous cette dénomination la gestion des informations caractérisant une configuration de FrameKit, c'est-à-dire : les formalismes et les services installés. L'ensemble des données à gérer est à la fois complexe par la quantité des conventions qu'elle implique et la diversité des informations qui y sont associées (fichiers de description des formalismes, fichiers de description des services, fichiers de description d'une configuration etc.)

C'est pourquoi dans FrameKit, toutes ces données respectent un modèle de diffusion élaboré en vue de faciliter la distribution modulaire d'éléments de configuration d'un Environnement de Génie Logiciel. Ce modèle de diffusion repose sur un concept clef : le *kit*, qui est un composant élémentaire pour l'installation. Il existe quatre type de kits :

- les kits *plates-formes* contiennent l'intégralité d'une plate-forme FrameKit, à l'exception de l'interface utilisateur qui est un logiciel à part,
- les kits *formalismes* contiennent l'ensemble des données permettant de déclarer un formalisme dans la plate-forme,
- les kits *outils* contiennent l'intégralité des informations caractérisant un outil (description des services qui lui sont associés, exécutables, etc.),
- les kit *spéciaux* contiennent un ensemble de données non particularisées (formalismes, outils, exécutables de la plate-forme) et ont pour raison principale la diffusion de corrections portant sur un ensemble d'éléments différents (patches).

Quelque soit son type, un kit regroupe un ensemble cohérent d'informations permettant l'exécution des fonctions qui lui sont associées. Les kits sont automatiquement installés par un outil d'administration de FrameKit.

Puisque la diffusion de FrameKit est multi-cibles, les kits plates-formes sont associés à une architecture cible. De même, les kits outils sont associés à une architecture et à un formalisme. Enfin, les kits formalismes peuvent être associés à une architecture, par exemple s'ils contiennent un outil vérificateur (outil «caché», automatiquement invoqué si besoin est avant l'exécution de tout service).

La Figure 33 illustre l'exploitation potentielle de ce modèle de distribution. On y considère un Environnement de Génie Logiciel dont les composantes sont développées sur différents sites. Il est alors possible d'imaginer que, sur un site de diffusion, les composants divers (sous forme de kits) soient mis à jours de manière décentralisée. De même, le client n'importe et n'installe que les éléments requis pour une utilisation donnée.

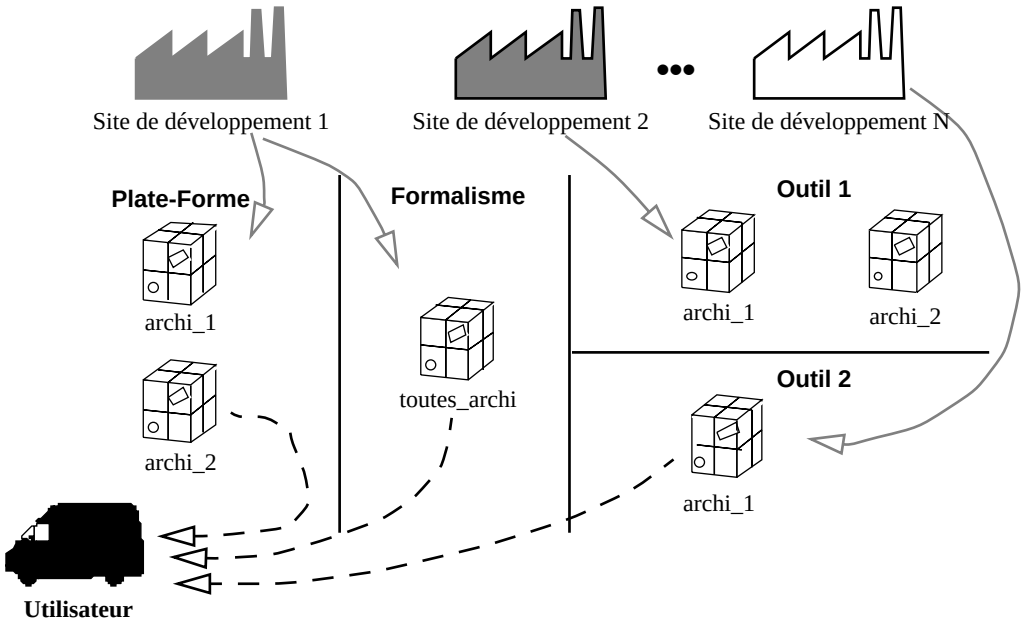

Figure 33 : Utilisation du modèle de distribution de FrameKit.

Gestion des données internes de FrameKit

Nous regroupons sous cette dénomination la gestion d'un certain nombre d'éléments internes de la plate-forme FrameKit comme les statistiques d'utilisation ou les caches associés aux menus de services (en vue d'accélérer la phase de connexion). Ces données sont construites par la plate-forme et leur consultation (données statistiques d'utilisation) ou leur destruction (caches) n'est possible qu'à travers des outils dédiés.

12.3. *M*eta*S*cribe

Meta***S***cribe est un générateur de moteurs de transformations. Sa réalisation répond à différents besoins :

- faciliter la mise en œuvre des standards secondaires dans FrameKit (voir Section 11.5.),
- supporter le développement de générateurs de programmes (voir Section 11.6.).

Dans les deux cas, un mécanisme de description de formats de représentations et la possibilité de réaliser des transformations entre formats d'échanges doivent être mis en œuvre. ***M***eta***S***cribe propose un gabarit pour la définition des formats en entrée et facilite la production de moteurs de réécriture adaptés aux deux cas de figure sus-mentionnés.

Meta***S***cribe couvre un problème voisin de celui identifié dans la conception en co-développement matériel/logiciel (codesign) de systèmes embarqués pour lesquels des processeurs spéciaux sont construits en petite série (en général à base de FPGA). Le problème est celui de la particularisation à faible coût d'un

compilateur en vue de la production d'un code spécifique lié aux composants matériels. Pour cela, on utilise des compilateurs multi-cibles (*retargetable compilers*) que **[11]** classifie comme suit :

- les compilateurs automatiquement adaptables, ce qui suppose que le compilateur contient la description de tous les jeux d'instructions qu'il sera susceptible d'utiliser;
- les compilateurs adaptables par l'utilisateur, pour lesquels un jeu d'instruction réduit est prédéfini en vue d'être paramétré. L'usager qui souhaite particulariser son compilateur redéfinit alors le codage de ce jeu d'instructions;
- les compilateurs adaptables par un développeur, qui permettent non seulement l'adaptation du jeu d'instruction mais aussi la définition de règles d'optimisation propres au composant matériel cible.

Meta***S***cribe propose, dans son domaine, une réponse aux deux derniers points mais en intégrant également des caractéristiques propres à celle des générateurs d'analyseurs lexicaux et syntaxiques comme le couple flex / bison **[227, 85]** .

D'un générateur d'analyseurs lexicaux et syntaxiques, il reprend à la fois la description d'un format en entrée et la possibilité de définir des règles qui seront appliquées selon un schéma d'analyse des données fournies en entrée du moteur de réécriture. ***M***eta***S***cribe est orienté vers l'analyse de représentations graphiques ou hiérarchiques et utilise donc des mécanismes de description du format en entrée différents de ceux utilisés dans un analyseur lexical ou syntaxique. Par contre, l'utilisateur peut parfaitement définir, à l'aide d'un *patron sémantique,* non seulement le schéma d'analyse des entrées mais aussi les constructions sémantiques qui constituent des éléments du format de sortie indépendamment de leur expression syntaxique.

D'un compilateur multi-cibles, il reprend la possibilité de paramétrer une «sortie» en fonction d'un format défini dans un *patron syntaxique*. Il est ainsi possible d'associer aux éléments sémantiques identifiés dans un patron sémantique un sucre syntaxique permettant de les exprimer. Le paramétrage du format en sortie s'apparente à l'hypergénéricité **[75]** d'un générateur de code[(8)].

Meta***S***cribe permet de paramétrer le format d'une spécification en entrée et dissocie les aspects sémantiques et syntaxiques de la description d'un moteur de réécriture, ce qui les rend potentiellement réutilisables. Ce point sera détaillé en Section 12.3.1.

Application à la mise en œuvre de standards secondaires

La mise en œuvre de standards secondaires associés à un formalisme permet d'unifier la représentation des résultats par des outils d'origine distincte. Ainsi, à un formalisme, on pourra associer par application de services une collection de

(8) L'hypergénéricité du générateur de code est décrite via des fichiers de configurations dans l'outil Objectering **[262]** (formalisme en entrée : Classe-relation) ou à l'aide du langage Tcl dans l'outil S-CASE **[210]** (formalisme en entrée : UML).

résultats décrits selon un format unique (Figure 34). De tels résultats peuvent alors être exploités par d'autres outils sans que ceux-ci se préoccupent de leur origine. Un résultat susceptible d'être calculé par plusieurs outils peut être exploité par d'autres sans que l'on se préoccupe de formats «propriétaires». La multiplicité d'outils calculant un même résultat peut être liée à un changement de version, d'architecture ou d'invocations différentes en fonction de critères de performance liées aux caractéristiques du modèle source.

Lorsque les outils n'admettent pas directement en entrée le format choisi pour représenter ce standard secondaire, on peut construire un moteur de réécriture effectuant la traduction de ce format vers le format exigé. Ce moteur sera appliqué avant exploitation des résultats. De même, différents moteurs de réécriture peuvent aboutir à des formats de visualisation différents. Dans la Figure 34, on trouve deux formats : la production d'un rapport (exportation d'un document exploitable par un traitement de texte) ou le pilotage d'une interface utilisateur (par exemple, des directives graphiques de visualisation en langage CAMI).

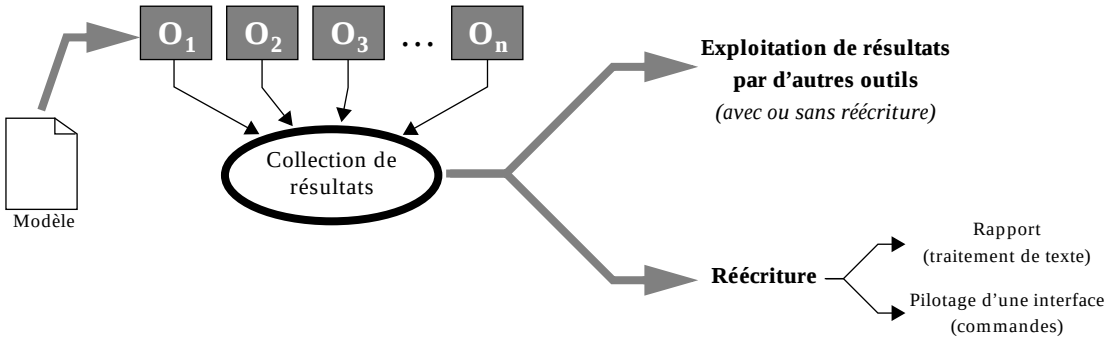

Figure 34 : Utilisation de moteurs de réécriture pour supporter la gestion de standards secondaires.

Application à la génération de code

La génération de code vise à la production de programmes par transformation d'une spécification initiale décrite selon un formalisme exploité par le moteur de transformation. L'approche proposée par ***M***eta***S***cribe convient parfaitement à ce problème (Figure 35). A partir d'un modèle source, on peut identifier les règles sémantiques produisant une description en mémoire des programmes générés en fonction de concepts issus d'un paradigme de programmation (i.e. des arbres d'expression de programmes). Ensuite, un patron syntaxique appliqué à cette vision conceptuelle permet de l'exprimer avec la syntaxe d'un langage de programmation.

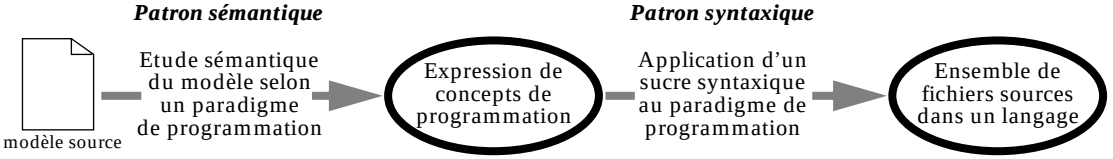

Figure 35 : Utilisation de ***M***eta***S***cribe pour la construction d'un générateur de programmes.

12.3.1. Principes de *M*eta*S*cribe

Un moteur de transformation est un outil prenant en entrée un modèle dans un formalisme source, et produisant un autre modèle dans un formalisme cible (Figure 36). Les modèles soumis au moteur de réécriture sont décrits au moyen d'un langage proposé par ***M***eta***S***cribe. Les modèles cibles sont produits au format ASCII.

Figure 36 : Fonctionnement d'un moteur de transformation.

Entités composant un moteur de réécriture

La première composante nécessaire à la définition d'un moteur de réécriture est le *formalisme source*. De cette description, on déduit les entités potentiellement présentes dans le modèle à traiter. La description du *formalisme cible* (en sortie), quant à elle, est «diluée» dans la définition des règles de transformation.

Nous avons dissocié les aspects sémantiques et syntaxiques d'une transformation. Le premier concerne la transformation en elle-même, le second correspond au sucre syntaxique appliqué pour représenter le format de sortie. Pour cela, nous introduisons deux types de patrons :

- Un *patron sémantique* est associé au formalisme source. Il définit les règles de transformations sémantiques applicables entre le formalisme en entrée et le formalisme en sortie.
- Un *patron syntaxique* est associé à un patron sémantique. Il décrit les expressions syntaxiques applicables pour la forme sémantique définie dans le patron sémantique correspondant; c'est-à-dire la structure de ce qui est produit dans le formalisme cible.

Meta***S***cribe génère un moteur de transformation à partir du triplet <*formalisme source, patron sémantique, patron syntaxique*> le caractérisant. Ce mécanisme permet de réutiliser les différents composants : à un formalisme source, on peut associer plusieurs patrons sémantiques correspondant à des transformations dans des formalismes cibles de classes différentes. De même, à un couple <*formalisme, patron sémantique*>, on peut associer plusieurs patrons syntaxiques, chacun correspondant à un habillage syntaxique différent des éléments produits par le patron sémantique.

La Figure 37 illustre les possibilités de réutilisation des différents composants décrivant un moteur de transformation. Nous y considérons la génération de code à partir d'un réseau de Petri pour un langage procédural classique ou un langage objet. On construit deux types de patrons sémantiques : le premier pro-

duit des expressions adaptées aux langages procéduraux tandis que le second génère des expressions adaptées aux langages objets. Dans chacun des cas, on peut appliquer des patrons syntaxiques différents selon que l'on souhaite générer du C, de l'Ada83, du C++ ou du Java.

Les entités caractérisant une transformation (la description du formalisme source, le patron sémantique et le patron syntaxique) sont définies de manière modulaire : ainsi, deux patrons sémantiques, syntaxiques, ou deux formalismes peuvent partager des définitions communes. Dans l'exemple donné précédemment, il est évident que le patron syntaxique associé au langage C++ empruntera beaucoup au patron syntaxique du C. De même, le patron sémantique pour les langages objets peut s'appuyer, pour ce qui concerne des constructions «classiques», sur des éléments appartenant au patron sémantique des langages procéduraux.

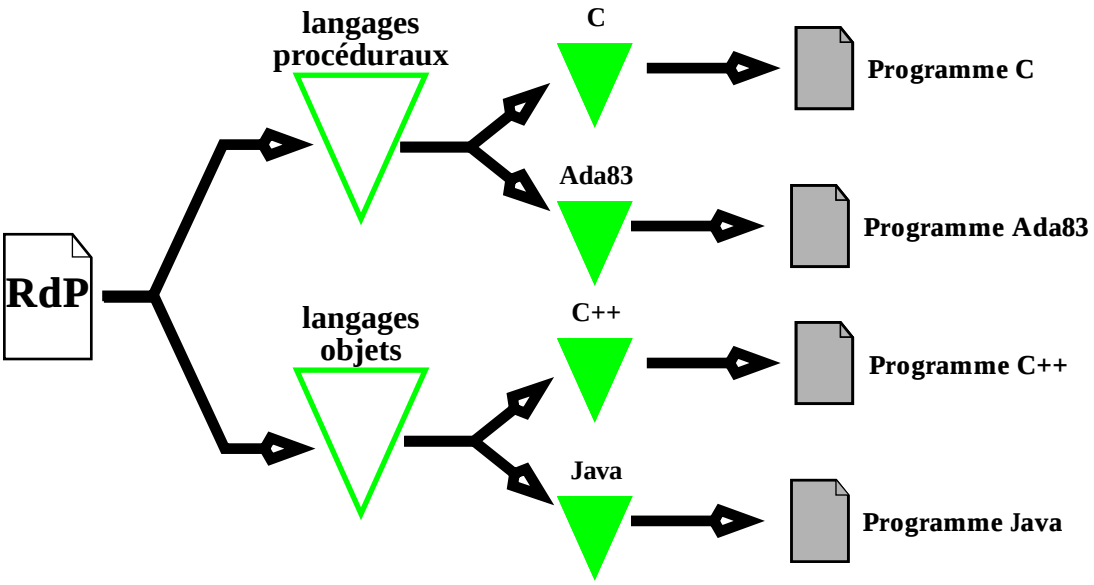

Figure 37 : Exemple de transformations d'un réseau de Petri dans plusieurs formalismes cibles appartenant à des classes différentes. Si tous les patrons sont définis, on peut obtenir ainsi quatre moteurs de transformations à partir d'un réseau de Petri.

Structure d'un moteur de transformation généré avec MetaScribe

La structure d'un moteur de transformation généré avec ***M***eta***S***cribe est donnée en Figure 38. La transformation se déroule en trois temps :

- décodage du modèle et construction d'une représentation en mémoire,
- application sur ce modèle du patron sémantique (PSm dans la Figure) qui produit des arbres d'expression en mémoire (les arbres sémantiques),
- application du patron syntaxique (PSy dans la figure) sur les arbres sémantiques pour y appliquer le sucre syntaxique correspondant.

Les modèles fournis au moteur de transformation doivent être décrits selon un format précis indépendant du formalisme. L'interprétation des entités d'un modèle se fait à partir de la description du formalisme.

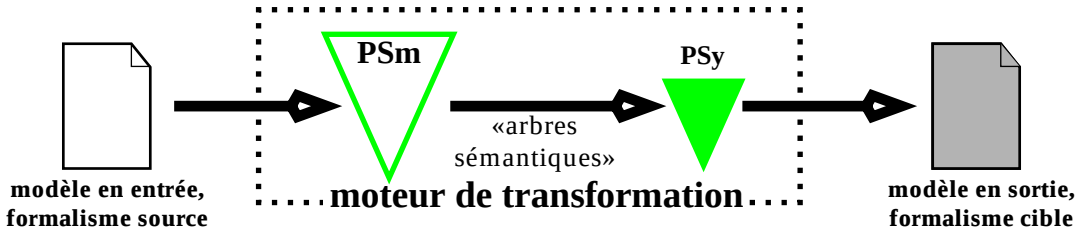

Figure 38 : Les étapes du traitement dans un moteur de réécriture obtenu grâce à ***M***eta***S***cribe.

Les patrons sémantiques sont composés de règles qui décrivent les mécanismes de transformation devant aboutir à la production d'une représentation sémantique dans le formalisme cible. Cette représentation s'exprime sous la forme d'arbres d'expressions (arbres sémantiques) similaires arbres de syntaxe abstraite de l'environnement ASIS **[250]** mais dont les étiquettes sont paramétrables. Les nœuds de ces arbres sont étiquetés par des chaînes de caractères et / ou par des constructeurs ayant une signification dans le formalisme cible.

Pour appliquer des règles syntaxiques aux arbres produits par application des règles d'un patron sémantique, on associe un ensemble de règles syntaxiques aux constructeurs définis dans le patron sémantique, ces derniers constituant le lien entre le patron sémantique et le patron syntaxique.

Dans la section suivante, nous allons décrire les principes de la mise en œuvre de ***M***eta***S***cribe. Les détails de la mise en œuvre et une illustration de l'utilisation de ***M***eta***S***cribe sont donnés dans **[173]**.

12.3.2. Mise en œuvre de *M*eta*S*cribe

Quatre langages ont été définis pour permettre la description des différentes entités nécessaires à la caractérisation d'un moteur de transformation (Figure 39) :

- MSF (***M***eta***S***cribe **F**ormalism) pour déclarer les entités d'un formalisme,
- MSM (***M***eta***S***cribe **M**odel) pour décrire un modèle en fonction des entités déclarées dans la description au format MSF du formalisme associé,
- MSSM (***M***eta***S***cribe **S**e**M**antic pattern) pour expliciter les règles de production d'une description sémantique dans la représentation cible,
- MSST (***M***eta***S***cribe **S**yn**T**actic pattern) pour définir le sucre syntaxique applicable aux expressions produites par application d'un patron sémantique.

Meta***S***cribe est un générateur de code produisant des programmes Ada à partir des règles contenues dans les patrons sémantiques et syntaxiques. Les vérifications liées au fort typage de ce langage sont utilisées pour garantir la sécurité du fonctionnement du moteur de transformation en assurant le contrôle dynamique des entités manipulées pendant l'exécution[(9)]. Le moteur de transformation ainsi obtenu produit des fichiers ASCII et s'intègre tel quel dans FrameKit (respect des

(9) Le générateur de code de ***M***eta***S***cribe est ainsi déchargé de cette tâche, ce qui ne serait pas toujours le cas pour d'autres langages de programmation.

hypothèses de fonctionnement et utilisation des interfaces programmatiques). Ce moteur est dédié à la transformation de modèles (au format MSM) conformes au formalisme source (décrit en MSF).

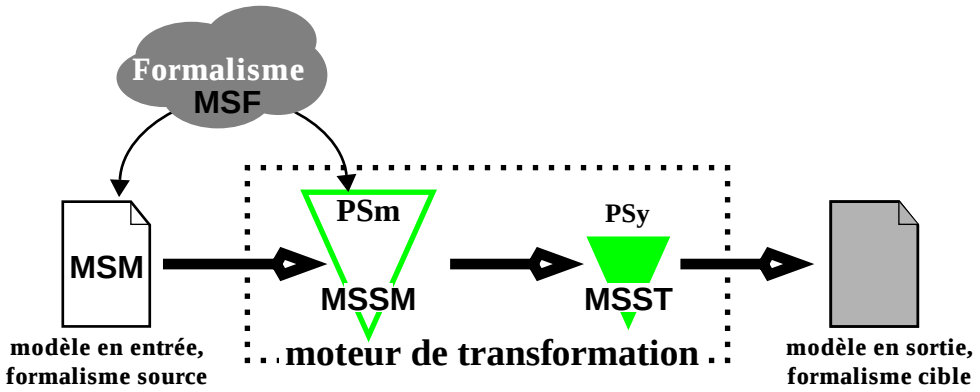

Figure 39 : Les entités manipulées par ***M***eta***S***cribe et leurs langages de description.

Pour représenter les formalismes et les modèles, il était impossible d'utiliser tels quels les mécanismes offerts par Macao qui sont trop rudimentaires. Aux notions de nœuds et de connecteurs auxquels sont associés des attributs, il faut ajouter un typage de ces attributs, ce qui permet de vérifier leur valeur et rend possible leur manipulation (ce qui est important pour définir les règles sémantiques).

Principes du langage MSF

MSF s'apparente à TIM dans l'environnement Tycho **[145]** et est tout à fait adapté à la description de formalismes graphiques et hiérarchiques. Les formalismes que nous avons retenus sont potentiellement graphiques et hiérarchiques (comme les réseaux de Petri ou H-COSTAM). La description d'un formalisme textuel n'est qu'un cas particulier facile à élaborer.

MSF permet de déclarer des composants élémentaires :

- les *attributs globaux* sont des informations associées au modèle;
- les *nœuds* sont des objets typés auxquels sont potentiellement associés des informations définies au moyen d'attributs. Chaque instance de nœud dans un modèle sera caractérisée par un identificateur (unique) et les valeurs associées à ses attributs;
- les *liens* sont des objets typés auxquels sont potentiellement associés des attributs. Chaque instance de lien dans un modèle relie entre eux N nœuds ($N \geq 2$); elle sera caractérisée par un identificateur (unique) et les valeurs associées à ses attributs;
- les *constructions sources prédéfinies* correspondent aux constructions syntaxiques licites que l'on peut trouver dans les attributs correspondant à des arbres d'expression (qui seront définis plus loin).

Les nœuds ne peuvent être reliés qu'au moyen d'un lien. Pour accroître les possibilités de contrôle dynamique, on exprime des règles de connectivité des nœuds permettant de préciser :

- Le nombre maximum de liens d'un type donné auquel il peut être relié,
- la direction des connexions acceptées. Les choix possibles sont : en entrée, en sortie ou sans direction. Dans les deux premiers cas, le lien est orienté et ne peut relier plus de deux nœuds. Dans le dernier cas, il n'est pas orienté et aucune limite n'est donnée aux nombre de nœuds qu'il relie.

Les attributs, qu'ils soient globaux, associés à un nœud ou à un lien, sont typés comme suit :

- *integer* : l'attribut contient une valeur entière,
- *string* : l'attribut contient une chaîne de caractère,
- *text* : l'attribut contient une suite de chaînes de caractères,
- *expression* : l'attribut contient un arbre d'expression(10),
- *boolean* : l'attribut contient une valeur booléenne(11).

Noeuds, liens, attributs et constructions sources prédéfinies constituent le lien entre un formalisme et le patron sémantique qui lui est associé.

Principes du langage MSM

Le langage MSM permet de décrire un modèle. C'est un mode de représentation neutre par rapport au formalisme et le format de description fourni en entrée d'un moteur de transformation produit par ***M***eta***S***cribe.

Ce langage décrit un modèle par rapport à son formalisme en identifiant les instances de chaque classe d'objet et leurs relations. Si une entité non déclarée dans le formalisme apparaît, le moteur de transformation trace l'erreur et stoppe son exécution. La description MSM n'est pas directement manipulée par ***M***eta***S***cribe; il est exploité par les moteurs de transformation générés.

Principes du langage MSSM

Le langage MSSM permet de décrire un patron sémantique composé de trois types d'entités : les *constructeurs prédéfinis*, les *règles sémantiques* et les *arbres sémantiques statiques*. Les constructeurs prédéfinis identifient les concepts manipulés dans le formalisme cible (pour un langage de programmation, il s'agit des instructions potentielles). Les règles sémantiques constituent le noyau des mécanismes de transformation tandis que les arbres sémantiques statiques sont des macro paramétrées définissant des constantes.

Une règle sémantique manipule les entités définies dans le formalisme source, c'est-à-dire :

- accès aux nœuds et liens d'un modèle,
- accès aux attributs, qu'il soient globaux ou associés aux nœuds ou liens du

(10) Cet arbre correspond typiquement à une expression compilée.

(11) Ce booléen est utilisable dans les expressions conditionnelles du langage MSSM.

modèle.

Une règle sémantique construit un arbre sémantique selon un format choisi, qui constitue une norme pour le patron sémantique considéré. Pour cela, il est possible :

- d'utiliser les constructeurs prédéfinis du patron sémantique. Ces constructeurs caractérisent des concepts propres au formalisme cible. Il sont référencés dans les nœuds des arbres sémantiques et servent de lien avec le patron syntaxique associé;
- d'invoquer d'autres règles sémantiques;
- de référencer des arbres statiques,
- d'appliquer une règle syntaxique à un arbre sémantique pour produire un texte dans un fichier.

Règles sémantiques et arbres statiques sont paramétrables. Les paramètres sont typés et peuvent être :

- une chaîne de caractères quelconque,
- un entier quelconque,
- un arbre syntaxique au format décrit dans le langage MSM,
- un arbre sémantique au format de ceux produits par une règle sémantique ou un arbre statique,
- un constructeur prédéfini du patron sémantique,
- une référence à une entité du modèle (nœud ou lien).

Les règles sémantiques sont typées par le type de valeur qu'elles rendent qui peuvent être :

- des chaînes de caractères,
- des entiers,
- des arbres sémantiques,
- des constructeurs prédéfinis,
- aucune valeur.

Il existe deux types de règles sémantiques : les *règles principales* qui seront toutes appliquées par le moteur de réécriture, et les *règles secondaires* qui correspondent à des «sous-programmes» invoqués depuis une autre règle. Un patron sémantique doit contenir au moins une règle principale. Par définition, les règles principales ne rendent aucune valeur.

Le corps d'une règle sémantique manipule les entités d'un modèle en vue de produire les arbres décrivant le résultat de la transformation. Ces arbres sont d'arité quelconque. Leurs nœuds contiennent *au moins une* des trois informations suivantes : un constructeur déclaré dans le patron sémantique, une chaîne de caractères ou un entier.

Principes du langage MSST

Le langage MSST permet de décrire un patron syntaxique. Un patron syntaxique est composé de règles s'appliquant aux arbres sémantique issus de l'application d'un patron sémantique sur un modèle. Le lien entre un patron sémantique et un patron syntaxique est effectué à l'aide des constructeurs prédéfinis déclarés dans le patron sémantique : à chacun d'eux doit correspondre une règle d'écriture dans le patron syntaxique qui lui est associé.

Il existe deux types de règles syntaxiques :

- Les *règles externes* : il y en a autant que de constructeurs prédéfinis déclarés dans le patron sémantique associé[12];
- Les *règles internes* : elles ne sont pas associées à des constructeurs prédéfinis du patron sémantique associé et ne peuvent qu'être invoquées par d'autres règles.

Une règle externe est en général «frontal» de plusieurs règles internes. Cela permet de générer des constructions compliquées ne pouvant être produites par la seule règle externe.

On applique toujours un patron syntaxique à un arbre sémantique et l'on range toujours le résultat dans un fichier ASCII. Aussi, ces deux paramètres (l'arbre et le fichier) ne sont jamais déclarés explicitement dans une règle. On les manipule comme suit :

- les lectures concernent un arbre sémantique,
- les écritures s'appliquent sur un fichier.

12.4.Environnement de simulation générique

Ce travail, qui a démarré fin 1997 en collaboration avec Alioune Diagne, vise à doter FrameKit d'un environnement de développement de simulateurs permettant d'animer «facilement» un nouveau formalisme.

Pour cela, nous avons identifié l'architecture type d'un simulateur (Figure 40). Trois composants se dégagent :

- un système de pilotage offre des fonctions type (manipulation pour l'utilisateur de notions comme les «pas d'exécution», «points d'arrêts» etc.). Ce système de pilotage est en liaison avec l'interface utilisateur via la plateforme d'accueil;
- un moteur d'exécution décrit les règles d'évolution du formalisme considéré;
- une bibliothèque de structures de données propose un représentation en mémoire d'un modèle à partir de la description externe de son formalisme.

Le système de pilotage et la bibliothèque de structures de données sont génériques. Seul le moteur d'exécution est dépendant du formalisme considéré. Nous réutilisons les bibliothèques de manipulation en mémoire d'un modèle au for-

(12) Il n'est pas exclu que certaines de ces règles soient vides, par exemple pour ignorer un constructeur prédéfini.

mat MSM développées pour ***M***eta***S***cribe car cette technologie est adaptée au paramétrage des structures de données en fonction des besoins du formalisme à simuler (via le couple MSF / MSM).

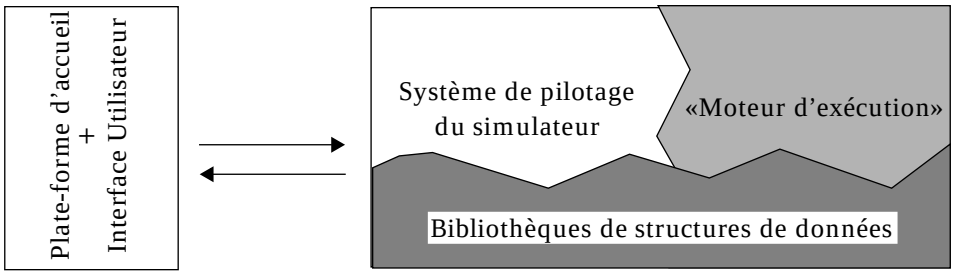

Figure 40 : Architecture type d'un simulateur.

Nous effectuons actuellement l'étude de faisabilité à travers l'implémentation d'un prototype que nous expérimenterons sur les formalismes de CPN-AMI2 dans le cadre du stage de DEA de Daniel Bondard.

13. Mise en œuvre et expérimentation

Le travail de développement sur FrameKit a démarré en mars 1995, en collaboration avec Jean-Luc Mounier et un premier prototype de la plate-forme, intégrant quelques outils, a été présentée un an plus tard à la conférence TACAS'96 (Tools and Algorithms for the Construction and Analysis of Systems). FrameKit est diffusée depuis mars 1997 sur Internet avec CPN-AMI 2.

FrameKit, développée en Ada, fonctionne sous Unix (systèmes SunOS, Solaris et un prototype sous Linux). Elle est distribuée sur Internet à l'adresse `<http://www-src.lip6.fr/framekit>` et a été utilisée dans le cadre de différents projets :

- CPN-AMI2, présenté en Section 8.1. et dont nous analyserons le retour d'expérience;
- CARISMA, pour le développement ponctuel d'une maquette de simulateur pour le formalisme AF-Class **[114]**. En moins d'une semaine de travail, nous avons montré à nos partenaires l'allure du formalisme et expliqué, par animation d'une spécification selon quelques scénarios prédéfinis, ses modalités de fonctionnement.
- VaMoS, pour le développement d'un environnement basé sur le formalisme Moka **[285]**. Ce travail a pour objectif la création du formalisme, l'intégration et l'utilisation de différents outils (principalement PROD **[279]** et Trio **[110]**).

13.1. Expérimentation avec CPN-AMI2

CPN-AMI2 est l'environnement opérationnel que nous avons construit sur FrameKit pour implémenter la méthode MARS (voir Section 7.). Une partie de cet

environnement (celle reposant sur les réseaux de Petri colorés) est diffusée sur Internet à <http://www-src.lip6.fr/cpn-ami>.

CPN-AMI2 est constitué d'une collection d'outils conçus pour FrameKit ou intégrés a posteriori. Comme nous l'avons déjà mentionné, plusieurs d'entre eux ont été développés dans d'autres universités. Nous n'en maîtrisions donc pas les sources.

Le Tableau 6 récapitule le temps passé pour l'intégration de ces outils. Nous avons ici considéré le temps du travail aboutissant à la première intégration permettant d'évaluer l'intérêt de l'outil. Ne figurent dans ce tableau récapitulatif ni les temps de développement des applications, ni les temps de «peaufinage» des intégrations lorsque l'évaluation a été positive.

Les applications sont classées en deux catégories : celles que nous avons importées (intégration a posteriori) et celles que nous avons développées pour FrameKit (utilisation des API spécialisées). Nous avons également distingué le temps d'adaptation de ces outils (écriture des pilotes ou, exceptionnellement, de modification des sources de l'application) et de déclaration dans la plate-forme (description des services offerts par l'outil).

S'il n'est pas surprenant que les applications développées avec les interfaces programmatiques de FrameKit n'aient pas posé de problème, la rapidité d'intégration des outils que nous avons importés est satisfaisante. Par exemple, en moins d'une demi journée de travail, nous disposions d'une première intégration de l'outil dot, développé aux laboratoires AT&T Bell **[176]** et n'effectuant que le placement des nœuds d'un graphe (le routage des arcs entre ces nœuds a ensuite été traité). Elle nous a permis d'évaluer l'intérêt de cet outil et un travail d'adaptation plus élaboré a suivi.

Si l'intégration d'un grand nombre d'outils s'est effectuée assez rapidement, certains ont posé des problèmes particuliers. Il s'agit de CPN / Desir (notre simulateur de réseaux de Petri développé sur la plate-forme AMI) et, dans une moindre mesure, de GreatSPN 2 et PROD. La forte interactivité de CPN / Desir explique la longueur du travail car de nouveaux mécanismes ont été mis en place dans FrameKit. Le développeur qui a intégré GreatSPN 2 utilisait FrameKit pour la première fois, ce qui explique également la durée de son travail. Enfin, la complexité de l'outil PROD (tant du point de vue ergonomique que du point de vue technique) justifie la lenteur de l'intégration.

Notons que, dans tous les cas, l'intégration de ces outils nous a permis d'importer un savoir-faire représentant une quantité de travail inenvisageable pour notre équipe. Nous avons pu importer et expérimenter de nombreuses technologies; en ce sens, l'utilisation de FrameKit a donné toute satisfaction.

Formalisme	Outil intégré	Type d'application		Temps d'intégration		Remarques
		Importée	développée pour FrameKit	Adaptation	Déclaration des services	
AMI-Net	**GreatSPN (version 1.6)**	✓		6 h	10 mn	Intégration effectuée à partir de fichiers exécutables uniquement. Une large partie de l'interfaçage est effectué en script shell.
	GreatSPN (version 2.0)	✓		7 j	20 mn	Intégration effectuée à partir de fichiers exécutables uniquement. Quelques difficultés ont été rencontrées lors de l'écriture des traducteurs entre les formats GreatSPN2 et CAMI, notamment dues à l'apprentissage de la plate-forme par le développeur.
	CPN/Desir (simulateur)	✓[a]		15 j	1 h	Outil hautement interactif. A nécessité une mise à jour de certains mécanismes dans la plate-forme. Certains éléments de l'outil ont également été transformés en C-ANSI.
	BooleanCondition		✓	30 mn	10 mn	Intégration au moyen d'un script shell.
	CPNverifier	✓[a]		1 h	10 mn	Combinaison de trois outils pilotés par un programme en script shell.
	CPNunfolder		✓	30 mn	10 mn	Pilotage via un programme en script shell.
	CPNinvariant	✓[a]		4 h	10 mn	Intégré tel quel après recompilation avec les API de FrameKit
	PROD (version 3.2)	✓		3 j	20 mn	Outil d'un fonctionnement très complexe. Intégration au moyen d'un pilote spécialement programmé en Ada.
	Prefix	✓		6h	10 mn	Intégration à partir de fichiers exécutables seulement
	PetriBDD	✓[b]		4 h	10 mn	Intégration au moyen d'un script shell.
	PrettyGraph (outil dot)	✓		3 h	10 mn	Intégration au moyen d'un pilote spécialement programmé en Ada de l'outil dot **[176]**.
	LinearCharacterization		✓	/	10 mn	Développement de l'outil en C avec les API de FrameKit
	Calcul de bornes	✓[a]		2 h	10 mn	Intégration au moyen d'un script
	Invariants	✓		8 h	10 mn	Enchaînement d'outils (déplieur + GreatSPN) piloté par un script shell
	Renew net importation	✓		5h	10mn	Invocation d'un programme Java issu de l'outil renew **[179]**.
OF-Class	**OFCverifier**		✓	/	10 mn	Développement de l'outil en C avec les API de FrameKit.
	PN-loader		✓	/	10 mn	Développement de l'outil en Ada avec les API de FrameKit.
	PROD	✓		/	10 mn	Factorisation de l'intégration pour les AMI-Nets. Seuls certaines fonctions qui n'ont pas de sens ne sont pas disponibles.

Tableau 6: Récapitulatif des durées d'intégration des outils dans CPN-AMI2.

Formalisme	Outil intégré	Type d'application		Temps d'intégration		Remarques
		Importée	développée pour FrameKit	Adaptation	Déclaration des services	
H-COSTAM	**HCMverifier**		✓	/	10 mn	Développement l'outil en Ada avec les API de FrameKit.
	HCM2PN (prototype)		✓	/	10 mn	Génération de l'outil par MetaScribe.
HADEL	**HADELverifier**		✓	/	10 m,	Développement de l'outil en Ada avec les API de FrameKit par des étudiants de Maîtrise.

Tableau 6: Récapitulatif des durées d'intégration des outils dans CPN-AMI2.

a. Il s'agit en fait d'une importation depuis l'environnement CPN-AMI 1.3, reposant sur la plate-forme AMI.
b. Il s'agit d'un outil développé dans notre laboratoire sans objectif d'intégration initial.

13.2.Le projet BioMedScape

Notre savoir-faire dans la réalisation de plates-formes logicielles m'a amené à participer au projet BioMedScape. Ce projet a été lancé en janvier 1996, grâce à un soutien de l'INSERM dans le cadre d'une convention liant cet organisme aux Universités de Franche-Comté (Laboratoire d'Histologie) et Pierre et Marie Curie (LIP6).

Son objectif principal est de réaliser une expérience pilote de communication électronique adaptée aux chercheurs d'une discipline donnée et comportant la mise au point d'outils de recherche maniables et originaux. La mise à disposition de ces outils à leurs utilisateurs est effectuée au moyen d'une plate-forme Internet intégrant des outils de capitalisation, de diffusion et d'analyse de données issues de la recherche **[171]**.

Ainsi, l'un des points importants du projet concerne la réalisation d'une plate-forme logicielle ayant des objectifs voisins de ceux de FrameKit. L'une des différences notables est constituée par l'utilisation de navigateurs Internet comme interface utilisateur. Les objectifs et principes mis en œuvre dans ce projet sont similaires à ceux proposés dans la plate-forme ETI (Electronic Tool Integration) **[268]**. Cependant, contrairement à ETI qui vise l'intégration d'outils via Internet, la plate-forme BioMedScape est plutôt une boîte à outils pour le développement d'outils. L'autre différence avec ETI est la volonté délibérée d'utiliser des mécanismes simples et requérant une faible bande passante.

La plate-forme permet d'authentifier les utilisateurs et de gérer des services (au sens FrameKit du terme). Le mécanisme de sélection des services disponibles pour un utilisateur repose ici sur deux notions : les domaines (i.e. la discipline scientifique) auquel il est inscrit et les groupes de travail (regroupement d'individus sur un projet donné) auquel il est abonné. La plate-forme BioMedScape

gère également des espaces de données accessibles selon les mêmes critères. Enfin, un mécanisme permet de lier n'importe quel type de données entres elles. L'activation de ces liens entraîne le lancement de l'outil qui est chargé de gérer la donnée correspondante.

Une discipline-pilote (la Neuroendocrinologie) a été ciblée en vue de servir de test et différents outils dédiés ont été élaborés puis intégrés dans la plate-forme en vue d'offrir trois services de base :

- un outil permettant de créer et maintenir des bases de données bibliographiques spécifiques, ainsi qu'une banque de données de références;
- un outil de recherche adapté supportant des requêtes simples (recherche d'occurrences d'un mot, recherche par proximité...) et des combinaisons de requêtes simples (conjonctions, disjonctions). Des "graphes" de concepts ont ensuite été élaborés pour permettre des recherches sur la base de critères anatomiques, physiologiques, biochimiques et pathologiques du domaine. Ces arbres seront maintenus par des scientifiques du domaine;
- un outil permettant les échanges "en ligne", c'est-à-dire des forums. L'une des particularités de ces forums est que des liens peuvent être réalisés sur d'autres informations gérées par d'autres outils (référence bibliographique etc.).

Au terme de deux années de travail, un prototype de BioMedScape (la plate-forme et ses trois services) a été présenté au forum d'Obernai, "nouvelles méthodes de communication et d'édition scientifique", organisé par l'INSERM.

14. Conclusion

Dans ce chapitre, je me suis attaché à identifier nos besoins en termes de génération d'Environnements de Génie Logiciel en nous positionnant par rapport au modèle de référence ECMA-NIST d'une part, et à l'étude d'environnements ou d'outils existants d'autre part (Ptolemy, PEP, Bast, GreatSPN, Design/CPN etc.). L'objectif de ce travail étant la production rapide de prototypes destinés à l'évaluation d'une méthodologie ou à sa démonstration, nous avons proposé un certain nombre d'hypothèses simplificatrices nous permettant d'interpréter les caractéristiques des architectures logicielles actuellement proposées en vue d'en simplifier l'implémentation.

J'ai ensuite présenté FrameKit, la solution que nous proposons au prototypage d'Environnements de Génie Logiciel. FrameKit est un ensemble de logiciels permettant :

- la production d'éditeurs graphiques dédiés à un formalisme,
- la gestion d'un Environnement de Génie Logiciel support d'une méthode et exploité par plusieurs utilisateurs,
- l'intégration et le développement d'outils.

Enfin, j'ai décrit les résultats obtenus dans la production de CPN-AMI 2 : un environnement de Génie Logiciel implémentant la méthode MARS. Les techniques mises en œuvre dans FrameKit nous ont permis d'importer un grand nombre d'outils de l'extérieur, ajoutant ainsi une valeur importante à l'environnement de départ. Ces intégrations se sont effectuées à très faible coût.

Chapitre 4
Conclusion et Perspectives

15. Conclusion

Les méthodologies et les environnements de Génie Logiciel les implémentant ont pour objectif de guider les développeurs d'applications complexes. Cependant, ces environnements se focalisent sur la description de solutions et n'offrent pas de proposition satisfaisante du point de vue de la vérification. Les ingénieurs en sont donc réduits à tester leurs applications une fois leur implémentation achevée et/ou au moyen de méthodes «exploratoires» non exhaustives basées sur des jeux de tests.

L'implémentation d'une solution exprimée dans les formalismes habituellement proposés entraîne souvent des dérives importantes liées à une mauvaise interprétation de la spécification initiale (de la solution). La génération de programmes présente une solution expérimentée avec succès dans des contextes industriels parfois critiques (par exemple, l'avionique). De plus, cette technique permet d'utiliser la spécification dans la phase de maintenance de l'application.

Pour évaluer la spécification d'une solution, les méthodes formelles permettent de traquer des caractéristiques structurelles et comportementales du système ayant un impact sur son exécution. Cependant, leur complexité de mise en œuvre est un frein qui empêche actuellement leur utilisation de manière généralisée.

Les aspects définition et implémentation sont indissociables si l'on se place dans un contexte de prototypage. Une fois le système élaboré puis vérifié, il est important de l'évaluer dans son environnement d'exécution, ce qui nécessite la mise en œuvre d'outils adaptés. Pour aider l'ingénieur, il est nécessaire de mettre en œuvre des environnements supportant la phase de conception, d'évaluation (au moyen de méthodes formelles) et d'implémentation (par génération d'applications).

Dans ce contexte, j'ai défini et partiellement implémenté une méthodologie associant des caractéristiques de «convivialité» des langages de spécification actuellement utilisés aux capacités de preuve formelle apportées par certaines méthodes formelles (dans mon cas, les réseaux de Petri). Pour atteindre cet objectif, j'ai dû travailler sur les méthodologies en elles-mêmes mais aussi sur

leur implémentation par prototypage pour évaluer leur impact en traitant des cas d'études «réalistes». Ainsi, du prototypage d'applications, mes travaux se sont étendus au prototypage d'Environnements de Génie Logiciel capables de produire ces applications.

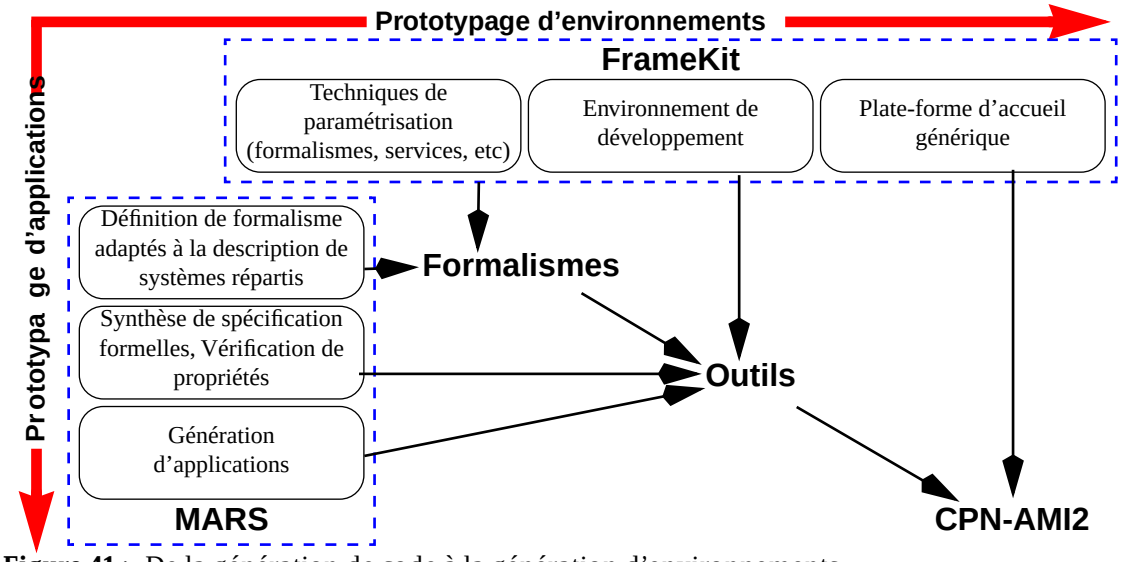

Figure 41 : De la génération de code à la génération d'environnements.

Cette démarche est illustrée par la Figure 41. Verticalement, le point de vue prototypage d'applications aboutit à la définition de méthodes (ici MARS) basées sur des formalismes adaptés, des techniques de vérification sur ces formalismes et de la génération d'applications. Horizontalement, le point de vue prototypage d'environnements permet la réalisation, à faible coût, d'Environnements de Génie Logiciel prototypes dédiés à de nouvelles méthodes de développement. La rencontre de ces deux axes de recherches est naturellement la mise en œuvre (encore partielle) de la méthode MARS à l'aide de FrameKit.

Dans les deux cas, nous nous plaçons dans une optique de prototypage par raffinements. L'application finale est obtenue par itérations successives d'un processus «modélisation/évaluation du modèle/génération de code», et les spécifications d'un Environnement de Génie Logiciel sont le résultat d'un processus de sélection des formalismes et outils adaptés à une méthode. L'aspect maintenance est traité dans les deux cas : le modèle d'un système peut évoluer pour intégrer de nouvelles contraintes (la nouvelle version de l'application est obtenue par génération de code) et l'Environnement de Génie Logiciel peut s'enrichir de nouveaux outils au fur et à mesure que la méthode associée évolue.

16. Perspectives

Les perspectives de mes travaux de recherche s'articulent sur quatre points : les techniques de modélisation, les techniques d'évaluation et vérification de modè-

les, la génération d'application et le prototypage d'Environnements de Génie Logiciel.

Techniques de modélisation

Nous avons expérimenté de manière satisfaisante l'encapsulation des réseaux de Petri avec des formalismes comme OF-Class (orienté preuve) et H-COSTAM (orienté génération de programmes). Cependant, l'existence de deux formalismes pose un problème ergonomique : si la phase d'élicitation correspond bien à une nécessité (introduction progressive d'informations concernant l'implémentation), il est préférable qu'elle soit réalisée à travers un seul et unique formalisme. De plus, pour que les méthodes formelles deviennent utilisables dans un contexte industriel, il est important d'adopter des langages de spécifications dérivés de ceux utilisés dans l'industrie.

Le co-développement logiciel matériel est une technique couramment employée dans le domaine des systèmes embarqués. Certains projets, comme PARSE **[225]** et Chinook **[62, 55]** proposent des formalismes ayant des liens de parenté avec H-COSTAM. Il serait intéressant d'étudier dans quelle mesure certains éléments de l'approche MARS seraient adaptables à une telle catégorie de systèmes.

Techniques d'évaluation et de vérification de modèles

L'expérimentation de MARS nous a conduit à observer certaines limites liées à l'utilisation des réseaux de Petri. Par exemple, si ceux-ci sont parfaitement adaptés à la spécification / vérification de comportements, ils sont moins appropriés à la description de structures de données (on observe en général une explosion de la complexité des graphes d'accessibilité correspondant). D'autres formalismes comme les spécifications algébriques **[109]**, devraient, sur ce point, apporter des solutions satisfaisantes. De même, d'autres méthodes formelles (π-calcul **[205]**, B **[3]**, etc.) permettraient d'étudier de nouveaux aspects de la dynamique d'un système. Dans un contexte où plusieurs méthodes formelles sont utilisées, l'encapsulation au moyen d'un langage de haut niveau devient extrêmement profitable car il ouvre l'accès à différentes techniques de preuves.

L'exploitation des propriétés issues de l'analyse du modèle formel est un problème qui ouvre un champ de perspectives important. Lorsqu'une propriété intéressante par rapport au modèle (et au parti pris de modélisation) n'est pas vérifiée, il est souvent difficile d'identifier simplement les causes de son échec. Ce point devient particulièrement délicat lorsque les techniques utilisées sont encapsulées par des formalismes de haut niveau où la réponse obtenue est encore du type «oui / non» (dans le meilleur des cas, on peut produire un contre-exemple). Étudier les mécanismes permettant de traduire, dans les termes du formalisme de haut niveau, les raisons de l'échec d'une propriété est également une condition nécessaire à la popularisation des approches multi-formalismes.

Génération d'application

Les techniques issues de la Recherche Opérationnelle sont pertinentes pour exploiter l'architecture d'un système et proposer des directives de placement

afin d'en optimiser leur exécution. Ce point est en cours d'étude en collaboration avec C. Hanen et A. Munier. Nous envisageons dans un premier temps de définir un lien entre les modèles utilisés actuellement et les informations dont on dispose au niveau d'une spécification semi-formelle. À terme, il sera peut-être possible de proposer des modèles plus proches de ces méthodes que ceux dont nous disposons actuellement.

Un autre problème des méthodes de conception / développement est la prise en compte de l'existant. L'approche par composants externes (représentés dans la spécification au moyen d'une abstraction) offre une solution à leur prise en compte dans un nouveau système. Cependant, pour s'assurer de la conformité d'un composant logiciel son abstraction, un retour aux techniques traditionnelles de tests est nécessaire. L'exploitation de cette abstraction pour générer automatiquement des séquences de tests de conformité et de robustesse **[44]** est une perspective intéressante. Des travaux sur ce sujet ont été entamés en collaboration avec Didier Buchs (École Polytechnique Fédérale de Lausanne) et Alioune Diagne.

Prototypage d'Environnements de Génie Logiciel

FrameKit nous a donné toute satisfaction lors de la construction de CPN-AMI 2. Cependant, il serait intéressant de vérifier qu'il est également adapté à d'autres besoins, en exportant une version «de développement» à destination d'autres utilisateurs. À ce titre, le développement conjoint, avec le CERT-ONERA dans le cadre du projet FORMA, d'un environnement logiciel basé sur le formalisme Moka **[285]** (évoqué en Section 13.) nous permettra d'éprouver et d'enrichir FrameKit. De même des collaborations sont prévues avec l'équipe de Génie Logiciel de l'École Polytechnique Fédérale de Lausanne afin d'utiliser FrameKit comme plate-forme d'intégration pour certains éléments de la prochaine génération de l'environnement CO-OPN **[30]**.

Pour mieux composer les méthodologies, nous étudions l'intégration dans FrameKit de mécanismes permettant de regrouper des modèles (dans un ou plusieurs formalismes). Cela permettra de mieux gérer les liens entre les différentes méthodes d'une méthodologie.

Références Bibliographiques

[1] Aarhus University, "Petri Nets Tools Database", <http://www.daimi.aau.dk/PetriNets/tools/db.html>

[2] C. Abarca, P. Farley, J. Forslöw, J. C. García, T. Hamada, P. F. Hansen, S. Hogg, H. Kamata, L. Kristiansen, C. A. Licciardi, H.Mulder, E. Utsunomiya & M. Yates, "Service Architecture , Version 5.0", ed. L. Kristiansen (TINA Consortium), June 1997

[3] J.R. Abrial, "The B-book", Cambridge University Press, 1995

[4] ACT-Europe, "GLADE user manual", <http://www.act-europe.fr/glade_guide.html>

[5] Ada Join Program Office, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A", U.S. Department of Defense, The Pentagon, Washington, January 1983

[6] Ada 9X Mapping/Revision Team, "Ada 95 Reference Manual and Rationale, ANSI/ISO/IEC-8652", Intermetrics, Inc. 733 Concord Avenue Cambridge, Massachusetts 02138 1995

[7] ADV technologies, "elsir home page", <http://worldserver.oleane.com/adv/elsir.htm>

[8] Y. Ait-Ameur, F. Besnard, P. Girard, G. Pierra & J-C. Potier, "Format Specification and MetaProgramming in the EXPRESS language", in proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE), 1995

[9] ALPHATEC, "ALPHA/Sim:A General Purpose, Discrete-Event Simulation Tool", <http://www.alphatech.com/alpha.htm>

[10] T.E. Anderson, D.E. Culler & D.A. Patterson, "A case for NOW (Network Of Workstations)", IEEE Micro, Vol 15(1), pp. 54-64, February 1995

[11] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang & A. Wang, "Challenges in Code Generation for Embedded Processors", Chapter 3, pp. 48-64, in "Code Generation for Embedded Processors", P. Marwedel and G. Goossens editors, Kluwer Academic Publishers, ISBN 0-7923-9577-8, 1995

[12] ARPA ProtoTech, "Home page of the Proteus Programming System", <http://www.cs.unc.edu/Research/proteus/>

[13] ARTIS S.R.L., "Artifex Home Page", <http://www.artis.it>

[14] S. Asur & S.Hufnagel, "Taxonomy of Rapid-Prototyping Methods and Tools" 4th IEEE International Workshop on Rapid System Prototyping, Research Triangle Park Institute, USA, IEEE comp Soc Press, N° 93TH0567-8, pp 42-56, 1993

[15] H. Bachatène & P. Estraillier, "Specification of large distributed systems integrating oriented objects concepts", in proceedings of the 12th World Computer Congress IFIP 92 "From research to Practice", Madrid, Spain, September 1992

[16] H. Bachatène & J.M. Couvreur, "A Reference Model for Modular Colored Petri Nets", in proceedings of IEEE/System, Man and Cybernetics International Conference, Le Touquet, France, October 1993

[17] H. Bachatène, "Une approche modulaire intégrant les réseaux de Petri colorés pour la spécification de systèmes distribués", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Septembre 1994

[18] Bakkenist Management Consultants, "ExSpect 4.2 User Manual", 1994

[19] M. Barbacci, D. Doubleday, C.Weinstock, M.Gardner & R.Lichotta, "Building Fault Tolerant Distributed Applications With Durra", in proceedings of International Workshop in Configurable Distributed Systems, IEEE press, England, 1992

[20] R. Bastide, "Approaches in unifying Petri nets and the Object-oriented Approach",1st Workshop on Object-oriented Programming and Models of Concurrency, Torino, Italy, 1995

[21] F. Bause, "Queueing Petri Nets - a formalism for the combined qualitative and quantitative analysis of systems",in proceedings of PNPM, IEEE Press, pp14-23, 1993

[22] F. Bause, P. Buchholz & P. Kemper, "QPN-Tool - A Tool for Hierarchical Analysis of Queueing Petri Nets", report of the Fachbereichs Informatik der Universität Dortmund (Germany), 1995

[23] J.M. Bernard & J.L. Mounier, "Conception et Mise en Oeuvre d'un environnement système pour la modélisation, l'analyse et la réalisation de systèmes informatiques", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 1990

[24] L. Bersntein, "Importance of Software Prototyping", in Testing Techniques Newsletter, on-line edition, oftware Research,Inc, February 1998.

[25] G. Berthelot, "Transformation and decomposition of Nets", in Advances in Petri Nets : Part I, LNCS, vol 254, W.Brauer, W.Reisig & G.Rosenberg Eds, Springer Verlag, pp 359-376, September 1996

[26] E. Best, R. Devillers & J. G. Hall, "The Box Calculus: a New Causal Algebra with Multi-Label Communication", Advances in Petri Nets, SpringerVerlag, LNCS 609, 1992

[27] E. Best & R. Hopkins, "B(PN)^2 - a Basic Petri Net Programming Notation", in proceedings of PARLE, LNCS vol 694, pp 379-390, Springer Verlag, June 1993

[28] E. Best & B. Grahlman, "PEP : Documentation and user guide", Universtät Hildesheim, 1995

[29] E. Best & M. Koutny, "A Refined View of the Box Calculus", in proceesings of the 16th International Conference on Application and Theory of Petri Nets, Torino, LNCS, Vol 935, 103-118, Springer Verlag, June 1995

[30] O. Biberstein, D. Buchs & N. Guelfi, "CO-OPN/2: A Concurrent Object-Oriented Formalism," in proceedings of the Second IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Chapman and Hall, pp. 57-72, UK, July 1997

[31] M. Bidoit, C. Choppy & F. Voisin, "ASSPEGIQUE+: an integrated specification environment providing inter-operability of tools", in proceedings of Algebraic Methodology and Software Technology (AMAST'96), L.N.C.S. 1101, Springer Verlag, pp 555-558, 1996

[32] M. Bidoit, C. Choppy, F. Voisin, "Interchange format for inter-operability of tools and translation. The Salsa and Asspegique+/LP experience", Recent Trends in Data Type Specification", Selected Papers of the 11th Workshop on Specification of Abstract Data Types (WADT-COMPASS), Springer-Verlag L.N.C.S. 1130, pages 102-124, 1996

[33] A. Billionnet, M.C. Costa & A. Sutter, "Les problèmes de placement dans les systèmes distribués", Techniques et Science Informatique, vol 8, N°4, 1989

[34] E. Black, "ATIS, CIS, PCTE and the Software BlackPlane", in proceedings of the 4th International Conference on Software Engineering and its Applications, pp 601-615, Toulouse, France, November 1991

[35] X. Bonnaire, C. Dutheillet & S. Haddad, "SANDRINE : an Analysis System for the Declaration of AMI-Nets, version 1.4", Rapport de l'Institut Blaise Pascal, Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France, October 1993

[36] C. Boussand, E. Cavillon & F. Kordon, "TEPACAP : Transformation Et réPartition d'une Application Codée automatiquement en Ada Pour unix", Rapport IBP / MASI N°92 / 22, Mai 1992

[37] Brandenburg Technical University, "PED home page", <http://www-dssz.Informatik.TU-Cottbus.DE/~wwwdssz/ped.html>

[38] F. Bréant & J.F. Peyre, "OCCAM Prototyping of Massively Parallel Applications using colored Petri Nets", 7th IEEE Int. Parallel Processing Symposium, Newport Beach, California, 1993

[39] F. Bréant & J.F. Peyre, " A New Prototyping Method of Massively Parallel Applications using Colored Petri Nets", Transputer Research and Applications, IO Press, Vancouver, Canada, vol 6, pp 83-98, 1993

[40] F. Bréant, "Décomposition de réseaux de Petri colorés. Modélisation d'architectures parallèles. Application au protypage sur des réseaux", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, 1993

[41] D. Buchs, J .Flumet & P. Racloz, "Producing Prototypes from CO-OPN Specifications", 3rd Int. Workshop on Rapid System Prototyping, Research Triangle Park, North Carolina, USA IEEE comp Soc Press N°92TH0503-3, pp 77-93, 1992

[42] D. Buchs, J. Hulaas, P. Racloz, M. Buffo, J. Flumet & E. Urland, "DANS, Structured Algebraic Net Development Systems for CO-OPN", in proceedings of the 16th International Conference on Theory and Applications of Petri Nets, LNCS vol 935, pp 45-53, Torino, Italy, June 1995

[43] D. Buchs & J. Hulaas, "Incremental Object-Oriented Implementation of Concurrent Systems Based on Prototyping of Formal Specifications", in proceedings of the SITAR Workshop, Biel, Switzerland, pp 141-145, October 1995

[44] D.Buchs, A. Diagne & F.Kordon, "Testing Prototypes Validity to Enhance Code Reuse", to appear of the 9th International Workshop on Rapid System Prototyping, J.Becker Ed, pp 6-12, IEEE comp Soc Press 98TB100237, belgium, June 1998

[45] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", International Journal of Computer Simulation, special issue on "Simulation Software Development", vol. 4, pp. 155-182, April 1994

[46] C. Burns, "PROTO - A software Requirements Specification, Analysis and validation tool", 1st IEEE International Workshop on Rapid System Prototyping, Research Triangle Park, USA, IEEE comp Soc Press, N° 91TH0380-6, pp 196-203, 1990

[47] C. Burns, "Parallel PROTO - A prototyping tool for analyzing and validating sequential and parallel processing software requirements", 2nd International IEEE Workshop on Rapid System Prototyping, Research Triangle Park Institute, USA,IEEE comp Soc Press N° 92TH0454-9, pp 151-160, June 1991

[48] C. Burns, "REE - A Requirements Engineering Environment for Analyzing & Validating Software and System Requirements", 4th IEEE International Workshop on Rapid System Prototyping, Research Triangle Park, USA, IEEE comp Soc Press, N°93TH0567-8, pp 188-193, 1993

[49] P. Buchholz, "Hierarchical High Level Petri Nets for Complex System Analysis", Proceedings of the 15th International Conference on Application and Theory of Petri Nets (LNCS, spinger Verlag), Zaragoza, Spain, LNCS vol. 815, pp. 119-138, June 1994

[50] G. Cabilic & I. Puaut, "Répartition de charge dans Stardust: un environnement pour l'exécution d'applications parallèles en milieu hétérogène", Actes de l'école thématique CNRS placement dynamique et répartition de charge, Presqu'île de Giens, pp. 167-174, Juillet 1996

[51] G. Canals, F. Charoy, G. Godart & P. Molli, "P-Root & COO : Building a Cooperatrive Software Development Environment", in proceedings of the 7th INternational conference on Software Engineering Environments (IEEE Computer Society Press), Mars 1995

[52] V. Cassigneul, "S.A.O. presentation", rapport technique de l'Aérospatiale N°463.097 / 91, Toulouse, 1991

[53] Cayenne Software, "Teamwork, documentation technique ", version 6, 1997

[54] M. J. Chen, "Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy", ERL Technical Report UCB/ERL No. 94/16, University of California, Berkeley, CA 94720, May, 1994

[55] Chinook Team, "The Chinook Project Home Page", <http://www.cs.washington.edu/research/chinook>

[56] G. Chiola, C. Dutheillet, G. Franceschini & S. Haddad, "On Well-Formed Coloured Nets and their Symbolic Reachability Graph", High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991

[57] G. Chiola, "GreatSPN 1.5 Software Architecture", In Proceedings of the 5th International Conference Modeling Techniques and Tools for Computer Performance Evaluation, Torino (Italy), February 1991

[58] G. Chiola & G. Franceschinis, "Structural colour simplification in Well-Formed coloured nets", in proceedings of the 4th International Workshop on Petri Nets and Performance Models, Melbourne, Australia, December 1991

[59] G. Chiola & A. Ferscha, "Distributed Simulation of Timed Petri Nets: Exploiting the net structure to Obtain Efficiency" Advances in Petri Nets, Lecture Notes in Computer Science, vol 691, pp 146-165, Springer Verlag - M. Ajmone Marsan Ed, 1993

[60] G. Chiola, G. Franceschinis, R. Gaeta & M. Ribaudo, " GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets", in Performance Evaluation, special issue on Performance Modeling Tools, 24(1&2), pp47-68, November 1995

[61] C. Choppy, "Spécifications algébriques: Prototypage et validation", Habilitation à diriger des recherches, Université Paris-Sud, 1994

[62] P. Chou, R. Ortega & G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", in proceedings of the International Symposium on System Synthesis, pp 22-27, Cannes, France, September 13-15, 1995

[63] P.Chrétienne & C. Picouleau, "Scheduling with communication delays: a survey", in P. Chrétienne, E.G Coffman, J.K Lenstra, and Z. Liu, eds, Scheduling Theory and its Applications, pp 65-89, John Wiley Ltd, 1995

[64] S. Christensen & L. Petrucci, "Towards a Modular Analysis of Coloured Petri Nets", LNCS, Vol. 60, pp 113-133. Springer Verlag, 1992

[65] E. Clarke, K. McMillan, S. Campos & V. Hartonas-Garmhausen, "Symbolic Model Checking", in proceedings of CAV, New Brunswick, LNCS 1102, Springer Verlag, 1996

[66] CNES/ESA, "Rapport de la Commission d'enquête Ariane 501", Juillet 1996

[67] CoFI group, "CoFI: The Common Framework Initiative for Algebraic Specification and Development", <http://www.brics.dk/Projects/CoFI/>

[68] D. Collins. Designing Object-Oriented User Interfaces. Benjamin/Cummings, 1995

[69] J.M. Colom, M. Silva & J.L. Villarroel, "On software implementation of Petri Nets and colored Petri Nets using high-level concurrent languages", 7th Workshop on Application and Theory of Petri Nets, pp207, 1986

[70] Condor Team, "Condor Version 6.0 Manual", University of Wisconsin–Madison, USA, http://www.cs.wisc.edu/condor/manual/condor-V6-Manual.pdf, May 1998

[71] G. Coulson & G. Blair, "Microkernel Support for Continuous Media in Distributed Systems", Computer Networks and ISDN Systems, Special Issue on Multimedia, 1994; also available as internal report MPG-93-04, Computing Dept., Lancaster University.

[72] CPN-Group at the University of Aarhus, "Design/CPN On Line", <http://www.daimi.aau.dk/designCPN/>

[73] D. Culler, R. Karp, D.Patterson, A.Sahay, K.Schauser, E.Santos, R.Subramonian & T.von Eicken, "LogP: Towards a realistic model of parallel computation", in proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993

[74] D. Dampier, Luqi, V. Berzins, "Automated Merging of Software Prototypes", Journal of System Integration, Vol. 4, No. 1, pp. 33-49, February 1994

[75] P. Desfray, "Modélisation par Objets", Masson 1996

[76] A. Diagne & P. Estraillier, "Formal Specification and Design of Distributed Systems" in proceedings of the 1st IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), Paris, France, March 1996

[77] A. Diagne & F. Kordon, "A Multi Formalisms Prototyping Approach from Formal Description to Implementation of Distributed Systems", in proceedings of the 7th International Workshop on Rapid System Prototyping, N.Kanopoulos Ed, pp 102-107, IEEE comp Soc Press, Greece, June 1996

[78] A. Diagne, "Une Approche Multi-Formalismes de Spécification de Systèmes Répartis : Transformation de Composants Modulaires en Réseaux de Petri", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Juin 1997

[79] A. Diagne, P. Estraillier, F. Kordon, I. Vernier, J. Cazin, M. Doche, C. Seguin & V. Wiels, "Spécification et validation modulaires de systèmes avioniques : Projet forma - opération vamos, rapport final", Technical Report 4/3599.00/DERI, ONERA - CERT, 2, avenue Edouard Belin, BP 4025, 31055 Toulouse, cedex 04, 1997

[80] A. Diagne & F. Kordon, "From formal specification to optimized implementation of distributed systems : A multi-formalism approach", Rapport de Recherche LIP6 1997/039, Université Paris 6 - CNRS, December 1997

[81] M.Diaz, J-P.Ansart, J-P.Courtiat, P.Azema & V.Chari Editors, "The Formal description Technique Estelle", North-Holland, 1989

[82] R. Di Giovanni, "Petri Nets and Software Engineering : HOOD Nets", 11th International Conference on Application and Theory of Petri Nets, Paris, France, 1990

[83] J. Diot, "Génération de programmes parallèles en langage C à partir d'un modèle de machines à états communicantes", mémoire de stage de DEA, Septembre 1996

[84] S. Donatelli, N. Mazzocca & S. Russo, "A CASE system for Petri net modelling of CSP-like programs", Transputer Communications, Wiley&Sons, Vol.3, No.1, January 1996

[85] C. Donnelly & R. Stallman, "Bison: The YACC-compatible Parser Generator", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/bison_toc.html>, November 1995

[86] G. Eckel, "Inside Windows Nt Workstation", New Riders Publisher, ISBN: 1562055836, February 1996

[87] D.E. Eckhardt Jr., M.J. Jipping, C.J. Wild, S.J. Zeil & C.C. Roberts, "Open Environments To Support Systems Engineering Tool Integration : A Study Using the Portable Common Tool Environment (PCTE)", NASA Technical Memorandum 4489, September 1993

[88] ECMA, "A Reference Model for Frameworks of Stoftware Engineerings Environments", ECMA report number TR/55 (version 3), NIST Report, April 1993

[89] W. El Kaim & F.Kordon, "An Integrated Framework for Rapid System Prototyping And Automatic Code Distribution", in proceedings of the 5th IEEE International Workshop on Rapid System Prototyping, Grenoble, France, IEEE Comp Soc Press N° 94TH0633-8, pp 52-61, June 1994

[90] W. El Kaïm, "Méthode de structuration, de placement et d'exécution de composants logiciels d'une application distribuée", dans "Placement et répartition de charge : application aux systèmes parallèles et répartis", G. Bernard, J. Chassin de Kergommeaux, B. Folliot, C. Roucairol Eds., Collection didactique INRIA, pp 325-343, Décembre 1996

[91] W. El Kaïm, "Structuration, Placement et Exécution de Composants Logiciels dans les Applications Réparties ou Parallèles : mise en œuvre avec des applications construites selon le paradigme client-serveur sur des architectures matérielles hybrides", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 1997

[92] J. Esparza, "Model Checking Using Net Unfoldings", in Science of Computer Programming, N°23, pp151-195, ELSEVIER, 1994.

[93] J. Esparza, S. Römer & W. Vogler, "An Improvement of McMillan's Unfolding Algorithm", in proceedings of Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1055, pp 87-106, Springer Verlag, March 1996

[94] R. Esser, "CodeSign version 1.0, Concepts and Tutorial", Report of the Eidgenössische Technische Hochschule Zürich, 1996

[95] P. Estraillier, "Modélisation, Analyse et Réalisation de Systèmes Distribués", Rapport d'habilitation à diriger des recherches, Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 1991

[96] P. Estraillier & C. Girault, "Applying Petri Net Theory to the Modelling, Analysis and Prototyping of Distributed Systems", Proceedings of the IEEE International Workshop on emerging technologies and factory automation - State of the art and future directions, Cairns, Australia, August 1992

[97] P. Estraillier & F. Kordon, "Structuration of large scale Petri Nets: an association with higher level formalisms for the design of multi-agent systems", proceedings of the IEEE International Conference on Systems, Man and Cybernetics pp 3198-3203, Beijing China, October 1996

[98] R.E. Faith, L.S. Nyland, D.W. Palmer and J.F. Prins, "The Proteus NS grammar," Technical Report TR94-029, UNC, 1994.

[99] A. Ferscha, "Modeling Mappings of Parallel computations onto Parallel Architecture with the PRM-Net Model", in proceedings of IFIP Working Group 10.3, Conference on Decentralized Systems, Lyon, France, pp. 215-242, December 1989

[100] H. Fleinschhack & B. Grahlmann, "A Petri Net Semantics for B(PN)^2 with procedures", in proceedings of PDSE'97, 1997

[101] B. Folliot, "Methodes et Outils de Partage de Charge pour la Conception et la Mise en Œuvre d'Applications dans les Systèmes Répartis Hétérogènes", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, Paris, France, Novembre 1992

[102] S. Fortune & J. Wyllie, "Parallelism in random access machines", in proceedings of the 10th ACM Symposium on Theory of Com-puting, pp. 114–118, 1978

[103] K. Foughali, "Conception et Réalisation d'une plate-forme d'intégration distribuée : application à l'Atelier de Génie Logiciel AMI", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, Paris, France, Février 1995

[104] E. Gamma, R. Helm, R. Johnson & M. Vlissides"Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-63361-2, 1995

[105] B. Garbinato, P. Felber & R. Guerraoui, "Modeling Protocols as Objects for Structuring Reliable Distributed Systems", in Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDS'97), Phoenix (Arizona), January 1997

[106] B. Garbinato & R. Guerraoui, "Bast, A Framework for Reliable Distributed Computing ", Technical Report, Operating Systems Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology, June 1997

[107] B. Garbinato, "A Framework of Patterns for Fault-Tolerant Distributed Computing", <http://lsewww.epfl.ch/Research/Bast>

[108] S.Garland & J. Guttag, "An overview of LP, the Larch Prover", in proceefings of the 3rd International Conference on Rewriting Techniques and Applications, LNCS 355, Springer Verlag, pp 137-151, 1989

[109] M.C. Gaudel, "Algebraic Specifications", Chapter 22 in "Software Engineer's Reference Book", John Mc Dermid ed, Butterworth, 1991

[110] G. Ghezzi, D. Mandrioli & A. Morzenti, "A model parametric real-time logic", ACM transactions on programming languages and systems, vol 14(4), pp 521-573, October 1992

[111] G. Geist &V. Sunderam, "Network-based concurrent computing on the PVM system", Concurrency: Practice & Experience, 4 (4), pp293-311, 1992

[112] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek & V. Sunderam, "PVM: Parralel Virtual Machine, A Users' Guide and Tutorial for Networked Parrallel Computing", MIT Press, 1994

[113] G. Geist, J. Kohl & P. Papadopoulos, "PVM and MPI: a Comparison of Features", Calculateurs Paralleles, Vol 8 N° 2, 1996

[114] M-P. Gervais & A. Diagne, "Enhancing Telecommunication Service Engineering with Mobile Agent Techology and Formal Methods", to appear in IEEE Communications Magazine, July 1998

[115] Gesellschaft für Prozeßautomation & Consulting, "POSES++ Online Documentation", <http://www.tu-chemnitz.de/ftp-home/pub/Local/simulation/poses%2b%2b/www/english/docu.htm>

[116] W. Gibbs, "Software's Chronic Crisis," in Scientific American, pp. 86-95, September 1994

[117] GMD, "the TRAPPER Home Page", <http://www.gmd.de/SCAI/lab/trapper/trapper_home.html>

[118] G. Godart, G. Canals, F. Charoy, P. Molli & H. Skaf, "Designing and Implementing COO : Design Process, Architectural Style, Lessons Learned", in proceedings of the International Conference on Software Engineering, 1996

[119] J. Goguen, "Requirements Engineering as the Reconciliation of Social and Technical Issues," in Requirements Engineering: Social and Technical Issues, M. Jirotka and J. Goguen eds., Academic Press, pp. 165-200, London, 1994

[120] A. Goldberg & D. Robson, "Smalltalk-80 : the language and its implementation", Addison-Wesley, 1989

[121] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif & J. Riely, "Specification and Development of Parallel Algorithms with the Proteus System", in Specification of Parallel Algorithms, G. Blelloch, M. Chandy, S. Jagannathan, eds., AMS, pp 383-399, 1995

[122] A. Goldberg, J. Prins, J. Reif, R. Faith, Z. Li, P. Mills,L. Nyland, D. Palmer, J. Riely & S. Westfold, "The Proteus System for the Development of Parallel Applications", in Prototyping Languages and Prototyping Technology, M. Harrison, ed., Springer-Verlag, 1996

[123] I. Gorton, J.P. Gray & I.E. Jelly, "Object Based Modelling of Parallel Programs", in IEEE Parallel and Distributed Technology Journal, Vol 3, No. 2, 1995, IEEE Computer Society Press, 1995

[124] A. Goscinski, "Distributed Operating Systems : the logical design", chapter 8, Addison-Wesley, 1991

[125] B. Grahlmann, "Petri Net File Formats", in proceedings of the 3rd Workshop Algorithmen und Werkzeuge für Petrinetze, Karlsruhe, Germany, 1996

[126] B. Grahlmann & E. Best, "PEP - More than a Petri Net Tool", in proceedings of TACAS'96, LNCS vol 1055, Springer Verlag, April 1996

[127] B. Grahlmann, "The PEP Tool", tool presentation at the 18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997

[128] A.Greiner, L.Lucas, F.Wajsbürt, "Designing a high complexity microprocessor using the alliance CAD system", proceedings of the 7th annual IEEE ASIC conference and exibit (ASIC'94), Rochester, USA, September 1994.

[129] K. Grönbœk, A. Hviid & R. Trigg, "APPLBUILDER - An object oriented application generator supporting rapid prototyping", International conference on Software engineering & its applications, Toulouse, France, 1991

[130] M. Hack, "Extended State-Machine Allocatable Nets (ESMA), an extension of free Choice Petri Net results", technical report of the MIT, project MAC, Computation Structures Group, Memo 78-1, 1974

[131] S. Haddad, "Une catégorie régulière de réseaux de Petri de haut-niveau : définition, propriétés et réduction", Thèse de l'Université Pierre & Marie Curie, Juin 1987

[132] S. Haddad, "A Reduction Theory for Coloured Nets", LNCS : High Level Petri Nets. Theory and Application. Edited by K. Jensen , G. Rozenberg, Springer Verlag 1991

[133] H. Hallmann, "A process model for prototyping", International conference on Software engineering & its applications, Toulouse, France, 1991

[134] D. Hatley & I. Pirbhai, "Strategies for real-time system specification", Donset house publishing Co, 1988

[135] D. Hauschildt, "A Petri Net Implementation", Fachbereich Informatik, Universität Hamburg, Hamburg, February 1987

[136] B. Haverkort, "SPN2MGM: Tool support for matrix-geometric stochastic Petri nets", in proceedings of the Second International Computer Performance and Dependability Symposium, IEEE Computer Society Press, pp 219-228 , 1996

[137] B. Haverkort & A.Ost, "SPN2MGM v1 - short manual", <http://www-lvs.informatik.rwth-aachen.de/tools/manual/manual.html>

[138] M. Heiner, "Petri Net Based Software Validation, Prospects and Limitations" Technical Report TR92-022, International Computer Science Institute, Berkeley, California, USA, March 1992

[139] O. Heitbreder, B. Kleinjohann, E. Kleinjohann & J. Tacken, "Intelligent Design Assistance with SEA", IEEE International Symposium and Workshop on Systems Engineering of Computer Based Systems (ECBS '97), Monterey, CA (USA), March 1997

[140] S. Hekmatpour & D. Ince, "Software Prototyping, formal Methods and VDM", Addison-Wesley, 1988

[141] C.A.R. Hoare, "Communicating Sequential Processes", Printice-hall International editor in computer science, C.A.Hoare series editor, 1985

[142] G. Holzmann, "Early Fault Detection Tools", proccedings of Tools and Algorithms for the Construction and Analysis of Systems, Springer Verlag - Lecture Notes in Computer Sciences, vol 1055, pp 1-13, 1996

[143] HOOD Technical Group, "HOOD Reference Manual, release 4", June 1995

[144] J.Hulaas, "An incremental Prototyping Methodology for Distributed Systems Bases on Formal Specifications", Thèse N°1664, département d'informatique de l'École Polytechnique Fédérale de Lausanne, 1997

[145] C. Hylands, E. Lee & H. Reekie, "The Tycho User Interface System", The 5th Annual Tcl/Tk Workshop '97, Boston, Massachusetts, pp 149-157, July 14-17, 1997

[146] IRENA Consortium, "Documentation utilisateur de l'outil CPN/Tagada (Traduction, Analyse et Génération de code ADA pour les réseaux de Petri Colorés)", référence DOC/MASI/1213, in deliverable of the IRENA project, année 3, Juin 1995

[147] IRENA Consortium, "Le O-Formalisme SPOKE : Un Outil de Vérification de Spécifications Modulaires", référence DOC/MASI/1215, in deliverable of the IRENA project, année 3, Juin 1995

[148] ISO International Standard 10165-4, "Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects", 1992

[149] ISO International Standard 10303-11, "Part 11: EXPRESS Language reference Manual", 1994

[150] ISO / IEC DIS 14750, "Information technology -- Open Distributed Processing -- Interface Definition Language", 1998

[151] ITU X.901 & ISO / IEC 10746-1, "The Reference Model of Open Distributed Programming : Overview, and Guide to Use", 1995

[152] I. Jacobson, M. Christerson, P. Johnsson & G. Overgaard, "Object oriented Software Engineering: A Use Case Driven Approach", ACM Press, Addisson-Wesley, 1992

[153] V. Janousek, "PNtalk: Object Orientation in Petri nets", in proccedings of European Simulation Multiconference ESM'95, Prague, pp 196-200, June 1995

[154] K. Jensen, "Coloured Petri Nets. Basic concepts, analysis method and practical use (vol 1)", EATC monographs on Theoretical Computer Science, Springer Verlag 1992

[155] E. Jul, H. Levy, N. Hutchinson & A. Black, "Fine Grained Mobility in the Emerald System", ACM Transactions on Computer System, 6(1), pp 109-133, 1988

[156] C. Kelling, "TimeNET-SIM - a Parallel Simulator for Stochastic Petri Nets", in proceedings of the 28th Annual Simulation Symposium, pp. 250-258, Phoenix, USA, 1995

[157] J.Kerr & R. Hunter, "Inside RAD", McGraw Hill, 1995

[158] E.Kindler & M. Weber, "Te Petri net kernel", technical Report, Humbolt University, September 1998, also in <http://www.informatik.hu-berlin.de/~kindler/PN-Kern/PNK-engl.html>

[159] B. Kleinjohann, E. Kleinjohann & J. Tacken, "The SEA Language for System Engineering and Animation", 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan, LNCS 1091, Springer Verlag, pages 307-326, 1996

[160] F. Kordon & P. Estraillier, "Complex Systems Rapid Prototyping and Environment Abstraction", proceedings of the 2nd "International Workshop on Rapid System Prototyping" N.Kanopoulos Ed, IEEE comp Soc Press 92TH0454-9, Triangle Park Institute, June 1991

[161] F. Kordon & J.F. Peyre, "Process decomposition for Rapid Prototyping of Parallel systems", 6th International Symposium on Computer and Information Science, Kener, Antalya, Turkie, October 1991

[162] F. Kordon & P. Sens, "Répartir des programmes Ada sur un ensemble homogène de machines Unix, une expérience de réalisation", Conférence Internationale Francophone Ada-France : "Ada, premier bilan d'utilisation", Novembre 1991

[163] F. Kordon, "Prototypage de systèmes parallèles à partir de réseaux de Petri colorés, application au langage Ada dans un environnement centralisé ou réparti", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Mai 1992

[164] F. Kordon, "On the use of structuration rules for Rapid Prototyping", Proceeding of the workshop “concurrency, specification & programming”, Berlin, November 1992

[165] F. Kordon, "Expérience d'utilisation de techniques de spécification et prototypage pour développer une application Ada intégrée dans un environnement logiciel", Rapport IBP / MASI N°93 / 36, Juin 1993

[166] F. Kordon, "A generic prototype model for distributed systems based on high level object oriented specification", proceedings of the 4th "International Workshop on Rapid System Prototyping", N.Kanopoulos Ed, IEEE comp Soc Press 93TH0567-6, pp 194-204, June 1993

[167] F.Kordon, "Prototypage automatique: de la maquette au système final", in proceedings of "Colloque sur la Conception de Systèmes", CNIT, Paris, Avril 1995

[168] F. Kordon & W. El Kaim "H-COSTAM : a Hierarchical Communicating State-machine Model for Generic Prototyping", 6th IEEE International Workshop on Rapid System Prototyping, Triangle Park Institute, USA, IEEE comp Soc Press N°95CS8078, pp 131-137, June 1995

[169] F. Kordon & J-L. Mounier, "FrameKit and the prototyping of CASE environments", 8th IEEE International Workshop on Rapid System Prototyping, Research Triangle Park Institute, IEEE comp Soc Press N° 97TB100155, pp 91-97, June 1997

[170] F. Kordon, "FrameKit version 1.4 : Manuel de Programmation", Documentation technique du laboratoire LIP6, Novembre 1997,

[171] F. Kordon, N. Poizot, G.Filliatreau & C. Kordon, "BioMedScape : A Web based Environment for Diffusion and Analysis of Research Data in Discrete areas of Life-Sciences", proceedings of the International Conference for Computer Communications - ICCC'97, pp 149-154, France, November 1997

[172] F. Kordon & J-L. Mounier, "FrameKit, an Ada Framework for a Fast Implementation of CASE Environments", to appear in proceedings of the ACM / SIGAda ASSET'98 symposium, July 1998

[173] F. Kordon, "MetaScribe : un outil pour la génération de moteurs de réécriture", Rapport LIP6, 98 / 37, Juillet 1998, aussi disponible sur <http://www.lip6.fr/reports/lip6.1998.037.html>

[174] F. Kordon, "Enregistrement des services dans la plate-forme d'accueil FrameKit", Rapport LIP6, 98 / 38, Juillet 1998, aussi sur <http://www.lip6.fr/reports/lip6.1998.038.html>

[175] G.Kurpis & C.Booth, "The new IEEE Standard Dictionnary of Electrical and Electronic Terms", New York, 1993

[176] E. Koutsofios & S.C. North, "Drawing graphs with dot", Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, September 1991. dot User's Manual.

[177] E. Koutsofios & S.C. North, "Drawing graphs with dot", Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[178] O. Krämer-Fuhrmann, L. Schäfers & C. Scheidler, "TRAPPER - A Graphical Programming Environment for Parallel Systems", in proceedings of New Computing Techniques in Physics Research III, Third International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, World Scientific Publishing Co., pages 3-15, October 1993

[179] O. Kummer, "Simulating Synchronous Channels and Net Instances", in proceedings of the Vth Workshop Algorithmen und Werkzeuge für Petrinetze J. Desel, P. Kemper, E. Kindler, A. Oberweis (eds.), pp 73-78 Fachbereich Informatik, Universität Dortmund, 1998

[180] C. Lakos, "LOOPN User Manual", Technical Report TR95-1, Computer Science Department,University of Tasmania, Australia, 1995

[181] C. Lakos & C.D. Keen, "An Open Software Engineering Environment Based on Object Petri Nets", Technical Report TR95-6, Computer Science Department,University of Tasmania, Australia, 1995

[182] C. Lakos, "The consistent use of names and polymorphism in the definition of objects Petri nets", 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan, LNCS 1091, Springer Verlag, 1996

[183] R. Lauwereins, M.Engels, M.Adé & J.A.Peperstraete, Grape-II: A System-Level Prototyping Environment for DSP Applications, IEEE Computer, Vol.24, No. 2, pp. 35-43, February 1995

[184] E.L Lawler, J.K Lenstra, A.H.G Rinooy Kan & D.B Shmoys, "Sequencing and scheduling: algorithms and complexity", in Handbook of Operations Research and Management Science, North-Holland, Amsterdam, 1993

[185] N. Leveson, "Software Engineering: Stretching the Limits of Complexity", Communications of the ACM, Vol 40(2), pp 129-131, February 1997

[186] Z. Li, P. Mills, & J. Reif, "Models and Resource Metrics for Parallel and Distributed Computation", in Journal of Parallel Algorithms and Applications, Vol.9, No.4, December 1995

[187] C. Lindemann, "Performance Modeling with Deterministic and Stochastic Petri Nets", John Wiley & Sons, 1997

[188] M. Litzkow, M. Livny, & M. Mutka, "Condor - A Hunter of Idle Workstations", Proceedings of the 8th International Conference of Distributed Computing Systems, pp. 104-111, June, 1988

[189] B. Liskov, "Distributed Programming in Argus", Communications of the ACM, 31(3), pp 300-312, March 1988

[190] Luqi, "Software Evolution Through Rapid Prototyping," in Computer, pp. 13-25, May 1989

[191] F.Long, E.Morris, "An Overview of PCTE: A Basis for Portable Commun Tool Environment", Technical report CMU / SEI-93-TR-1, ESC-TR-93-175, March 1993

[192] Luqi, M. Shing & J. Brockett, "Real-time scheduling for software prototyping", 4th IEEE International Workshop on Rapid System Prototyping, Triangle Park Institute, USA, IEEE comp Soc Press, N° 93TH0567-8, pp 150-163, 1993

[193] Luqi & J. Goguen, "Some Suggestions for Progress in Software Analysis, Synthesis and Certification," in proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, pp. 501-507, Skokie, USA, 1994

[194] Luqi & J. Goguen, "Formal Methods: Promises and Problems", IEEE Software, Vol 14, N°1, pp 75-85, January 1997

[195] K. McMillan, "Symbolic Model Checking", Kluwer Academic, 1993

[196] J. Magee, J. Kramer & M. Sloman, "Constructing Distributed System in CONIC", IEEE Transactions on Software Engineering, SE-15(6), June 1989

[197] T. Margaria, V. Braun & J. Kreileder, "Interacting with ETI: a user session", in international Journal on Software Tools for Technology Transfervol 1997-1, pp 49-63, Spinger Verlag, 1997

[198] MARS-Team, "The CPN-AMI environment version 1.3", MASI lab, Institut Blaise Pascal, Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, aussi sur <ftp://anonymous@ftp.lip6.fr/lip6/softs/ami/binaries/package/cpn-ami.1-3.tar>, June 1994

[199] MARS-Team, "The CPN-AMI environment version 2.3", <http://www-src.lip6.fr/cpn-ami>, November 1998

[200] K.L. McMillan, "A Technique of a State Space Search Based on Unfolding", in Formal Methods in Design, volume 6(1), pp 45-65, 1995

[201] S. Melzer, "Design and Implementation of a C-Code Generator for $B(PN)^2$", in "PEP : Programming Environment Based on Petri Nets", Hildeshermer Informatik-Berichte 14 / 95, 1995

[202] Meta Software, "Design / CPN Internal Functions : Programmer's Reference Version 2.0", Meta Software Corporation, 125 CambridgePark Drive, Cambridge, MA 02140 U.S.A., 1993

[203] P. Mills, L. Nyland, J. Prins & J. Reif, "Prototyping High-Performance Parallel Computing Applications in Proteus", in proceedings of 1992 DARPA Software Technology Conference, pp 433-442, April 1992

[204] P. Mills, L. Nyland, J. Prins & J. Reif, "Software Issues in High-Performance Computing and a Framework for the Development of HPC Applications", in Computer Science Agendas for High Perfromance Computing, U. Vishkin ed, ACM, 1994

[205] R. Milner, "The polyadic pi-calculus: a tutorial", in "logic and algebra of specification", F.L. Bauer, W. Brauer & H. Schwichtenberg Eds, Springer Verlag, pp 203-246, 1993

[206] D. Moldt, "OOA and Petri Nets for System Specification", in proceedings of Workshop Object-Oriented Programming and Models of Concurrency, G.Agha & F.De Cindio Eds, University of Torino, June 1995

[207] J-L. Mounier, "the Macao Home page", <http://www-src.lip6.fr/macao>

[208] T. Mowbray & R. Zahavu, "The Essential CORBA: Systems Integration Using Distributed Objects", John Wiley & Sons, 1995

[209] MPI Forum, "MPI: A message-passing interface standard. International", in Journal of Supercomputer Application, 8 (3/4), pp165-416, 1994

[210] MultiQuest Corporation, "S-CASE 3.0 User's guide", 1931 N Meachanm Road, Suite 318, Schaumburg, IL 60173, USA, 1996

[211] T. Murata, N. Komoda & K. Matsumoto, "A Petri Nets based Factory Automation controller for flexible and maintainable control specification", IEEE Trans on Industrial Electronics, vol IE-33, N°1, February 1986

[212] T. Murata, "Petri nets : properties, analysis and applications ", proceedings of the IEEE, vol 6, n°1, pp 39-50, January 1990

[213] S.C. Murphy, P. Gunningberg & J.P.J. Kelly, "Implementing protocols with Multiple Specifications: Experiences with Estelle, LOTOS and SDL", 9th IFIP WG 6.1 International Symposium on Protocol Specifications, Testing and Verification, Enschede, The Netherlands, June 1989

[214] C.S.R. Murthy & V. Rajaraman, "Task Assignment in a Multiprocessor System", Microprocessing and Microprogramming, 26:63-71, 1989

[215] R. Nelson, L. Haibt & P. Sheridan, "Casting Petri Nets into Programs", IEEE Transactions on Software Engineering, vol 9, pp 590-602, 1983

[216] Netmate Team, "Netmate home page", <http://www.e-technik.uni-kl.de/litz/ENGLISH/software/netmate.htm>

[217] M. Norman & P. Tanisch, "Models of machines and computation for mapping in multicomputers", in ACM computing Surveys, vol 25, n°3, pp 263-302, September 1993

[218] ODP, "The Reference Model of Open Distributed Programming, Overview, and Guide to Use" Draft ITU-T, Recommendation X.901, 1995

[219] OMG, "Join Object Service Submission", document N°93.2.1, February 1993

[220] OMG, "The common Object Request Broker: Architecture and Specification", rev 2.0 + update, July 1996

[221] J. K. Ousterhout, "An X11 Toolkit Bassed on the Tcl Language," USENIX Conference Proceedings, 1991

[222] C. Ozanne, "Conception d'applications client-serveur: modèles d'architecture fonctionnelle et opérationnelle", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Avril 1996

[223] M. Palmer, "Guidelines for the development and approval of STEP application protocoles", Technical report, ISO TC184/SC4/WG4 N511, 1995

[224] M. Paludetto, "Sur la commande de procédés industriels : une méthodologie basée objets et réseaux de Petri", Thèse de l'Université Paul Sabatier, Toulouse, France, 1991

[225] PARSE-project, "The PARSE Project", <http://www.dcs.shef.ac.uk/~prc/parse.html>

[226] PARSE-project, "Definition of the PARSE Process Graph Notation version 2", Technical Report PARSE-TR-2b, Computer Science department, University of Wollongong, Australia, March 1994

[227] V. Paxson, "Flex: A fast scanner generator, Edition 2.5", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/flex_toc.html>, March 1995

[228] PEP Team, "PEP Home page", <http://www.informatik.uni-hildesheim.de/~pep/HomePage.html>

[229] C. Péraire, S. Barbey, and D. Buchs, "Test Selection for Object-Oriented Software Based on Formal Specification", In Proceedings of PROCOMET98, N.Y, USA, 8-12 June. 1998

[230] J-F. Peyre, "Résolution paramétrée de systèmes linéaires ; applications au calcul d'invariants positifs dans les réseaux colorés, à la validation formelle et à la génération de code", PhD thesis, Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France, March 1993

[231] M. Pezzè & C. Ghezzi, "Cabernet : a customizable Environment for the Specification and Analysis of Real-Time Systems", Report of the Dipartimento di Elettronica e dell'Informazione Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, 1992

[232] PCIS Architecture Team, "PCIS Architecture Framework Definition and Rationale - Framework Definition and Rationale", Deliverable DA3, version 1.0, December 1993

[233] G. Pinna, "Petri Nets and Their Composition Problem". PhD Thesis: TD-2/90, Dipartimento di Informatica, Universita degli Studi di Pisa, Mars 1990

[234] Platform Computing Corporation, "LSF 3.0 User's Guide", Fourth Edition, December 1996

[235] R. Podorozhny & L. Osterweil, "The Criticality of Modeling Formalisms in Software Design Method Comparison", in Proceedings of the 19th International Conference on Software Engineering, pp 303-313, Boston, USA, May 1997

[236] L. Pomello, "Refinement of Concurrent Systems based on Local State Transformations", Stepwise Refinement of Distributed Systems, REX Workshop, Mook, The Netherlands, LNCS N° 430, Springer Verlag, 1990

[237] Pontificia Universidade Catolica do Rio, "ANARCO Home page", <http://www.ele.puc-rio.br/~menasche/anarco.html>

[238] Ptolemy project, "Ptolemy Programmer's Manual", University of California berkeley, 1996

[239] Ptolemy project, "the Ptolemy Home Page", <http://ptolemy.berkeley.edu/>

[240] T. Quatrani, "Visual Modeling with Rational Rose and UML", Addison-Wesley, ISBN: 0-201-31016-3, 1998

[241] J. L. Rasmussen & M. Singh, "Mimic/CPN : A Graphic Animation Utility for Design/CPN", Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, December 1995

[242] M. Richard-Foy, "L'optimisation des tâches passives en Ada", La lettre Ada, Nº 48-49, Novembre 1991

[243] S. Roch & P. Starke, "Integrated Net Analyser INA", <http://www.informatik.hu-berlin.de/~starke/ina.html>

[244] V. Roustan, "Integration des outils dans les ARES", SPEC/ARES/2002/V7, dans le délivrable IRENA année 1, Décembre 1992

[245] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani& W. Lorensen, "OMT : Modélisation et conception orientées objet", Masson & Prentice hall, 1994

[246] J. Rumbaugh, I. Jacobson & G. Booch, Unified Modeling Language Reference Manual, ISBN: 0-201-30998-X, Addison Wesley, est. publication December 1997

[247] S. Russo, C. Savy & I. Jelly, "From Textual Representation of PARSE Designs to Perti nets", joint Technical Report N° 7/95, Computing Research Centre, Sheffield Hallam University and University of Naples, April 1995

[248] S. Russo, C. Savy, I. Jelly & P. Collingwood, "Petri net modelling of PARSE designs", in proceedings of EuroPar96, Lecture Notes in Computer Science - Springer Verlag, August 1996

[249] L. Schäfers, C. Scheidler, T. Born & W. Obelöer, "Monitoring the T9000 - The TRAPPER Approach", in proceedings of the World Transputer Congress (WTC 94), Cernobbio, Italy, September 1994

[250] S. Rybin, A. Strohmeier & E. Zueff, "ASIS for GNAT:Goals, Problems and Implementation Strategy", In M. Toussaint (Ed), Second International Eurospace - Ada-Europe Symposium Proceedings, LNCS no 1031, Springer Verlag, pp 139-151, 1995

[251] L. Schäfers, C. Scheidler & O. Krämer-Fuhrmann, "Software Engineering for Parallel Systems: The TRAPPER Approach", in proceefings of the 28th Hawaiian International Conference on System Sciences, Hawaii, USA, January 1995

[252] D. Schefström, "System Development Environments : Contemporary Concepts", in Tool Integration : environment and framework, Edited by D.Schefström & G. van den Broek, John Wiley & Sons, 1993

[253] S. Schöf, M. Sonnenschein & R. Wieting, "Efficient Simulation of THOR Nets", in proceedings of the 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, LNCS vol 935, pp 412-431, Springer Verlag, June 1995

[254] M. Shaw & D. Garlan, "Formulations and Formalisms in Software Architecture", in Computer Science Today: Recent Trends and Developments, J. van Leeuwen (Ed), LNCS Sprinver-Verlag, pp 307-323, 1996

[255] S. Shtil & V. Bhatia, "Rapid Prototypong Technology Cuts System Verification Time and Cost by More Than Half", in proceedings of the Design Conference, January 1998

[256] C. Sibertin-Blanc, "Cooperative Nets", 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain, 1994

[257] C. Sibertin-Blanc, "SYROCO Reference Manual, version 7", rapport IRIT, Université Toulouse 1, Mai 1997

[258] M. Silva & S. Vellila, "Programmable logic controllers and Petri Nets", International symposium IFAC-IFIP on Software Computer Control, Madrid, Spain, 1982

[259] A. Silberschatz and P. Galvin, "Operating Systems Concepts", Chapter 20, pages 659-689, Addison-Wesley Publishing Company, 4rd Edition 1994

[260] J. Sklenar, "PetriSim version 2 User Guide", <http://www.cis.um.edu.mt/~jskl/manual2.html>

[261] Sligos, "Manuel utilisateur de DOM", 1995

[262] SofTeam, "Manuel utilisateur d'*Objecter*ing v4", 1996

[263] I. Sommerville & R. Thomson, "Configuration Specification Using a System Structure Language", International Workshop in Configurable Distributed Systems, IEEE Press, England, pp. 80-89, 1992

[264] Y. Souissi, "Compositions of nets via a communication medium", Proceedings of the 10th International Conference on Application and Theory of Petri Nets, Bonn, Germany, 1989

[265] Standish Group International, "Chaos 97 technical report", Internal report, available on <http://www.standishgroup.com/chaos.html>, 1995

[266] P. Starke, "INA: Integrated Net Analyzer", Handbuch, 1992

[267] J. B. Stefani, "Open Distributed Processing: an Architectural Basis for Information Networks", Computer Communications, Vol. 18, n° 11, pp849-862, November 1995

[268] B. Steffen, T. Margaria & V. Braun, "The Electronic Tool Integration platform: concepts and design", in international Journal on Software Tools for Technology Transfervol 1997-1, pp 9-30, Spinger Verlag, 1997

[269] H.Störrle, "An Evaluation of High-End Tools for Petri Nets", technical report of the LUM, University of Müchen, germany, <http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/survey.ps.gz>, April 1998,

[270] SURF Team, "SURF2 User Guide", rapport du laboratoire LAAS, <http://www.laas.fr/surf/binary/surf2-doc.ps.Z>

[271] O. Sy, "Elicitation d'une modélisation conceptuelle en une modélisation opérationnelle", rapport du DEA systèmes Informatique, Septembre 1997

[272] D. Sydow, "Macintosh Programming Techniques", 2nd edition, M&T Books, 1996

[273] D. Taubner, "On the implementation of Petri Nets", Advances in Petri Nets, Lecture Notes in Computer Science, vol 340, pp 418-440, Springer Verlag - G.Rozenberg, H. Genrich and C. Roucairol Eds, 1988

[274] M. Thomas & B. Nejmeh, "Definitions of Tool Integration for Environments", IEEE Software 9(2), pp 29-35, March 1992

[275] R. Valette, M. Courvoisier, J.M. Bigou & J. Alburkerque, "A Petri Net based programmable logic controller", 1st International Conference on Computer Application in Production and Engineering, Amsterdam, April 1983

[276] R. Valk, "Petri Nets as Dynamical Objects", proceedings of 1st Workshop on Object-oriented Programming and Models of Concurrency, Torino, Italy, 1995

[277] A. Valmari, "Compositional Analysis with Place-bordered Subnets", Proceedings of the 15th International Conference on Application and Theory of Petri Nets (LNCS, spinger Verlag), Zaragoza, Spain, June 1994

[278] K. van Hee & P. Verkoulen, "Integration of a Data Model and Petri Nets", in proceedings of 12th International Conference on Application and Theory of Petri Nets, pp410-431, Gjerns, Denmark, 26-28 June 1991

[279] K. Varpaaniemi, J. Halme, K.Hiekkanen & T.Pyssysalo, "PROD reference manual", Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995

[280] J. Vitter & E. Shriver, "Algorithms for parallel memory II: Hierarchical multilevel memories", Algorithmica, 1993

[281] R. Vonk, "Prototypage : l'utilisation efficace de la technologie CASE", MASSON & Prentice Hall, Paris, 1992

[282] the Waite Group, "Tricks of the HyperTalk Masters", ISBN 0-672-48431-5, Hayden Books, 1989

[283] A. Wasserman, "Tool Integration in Software Engineering Environments". In Fred Long, editor, Software Engineering Environments, Springer-Verlag, LNCS vol 467, pp 137-149, 1990

[284] W.E. Weihl, "Remote Procedure Call", chapter 4 in "Distributed systems", edited by S. Mullender, ACM press, 1989

[285] V. Wiels, "Modularité pour la conception et la validation formelle de systèmes", PhD thesis, ENSAE - ONERA / CERT, October 1997

[286] S. Zhou, X. Theng, J. Wang & P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems", Software - Practice and Experience, 23(12), pp 1305-1336, 1993

Résumé

Dans le contexte actuel où le développement et la maintenance d'applications industrielles deviennent sans cesse plus délicats, l'un des problèmes est l'évaluation des systèmes que l'on construit. Pour faciliter le travail des concepteurs, des méthodes de Génie Logiciel ont été introduites puis mises en œuvre au moyen d'Ateliers de Génie Logiciel. Elles permettent d'identifier les problèmes et leurs solutions au moyen de modèles décrits dans des formalismes dédiés.

Cependant, une fois les solutions décrites, leur implémentation engendre une dérive importante liée en général aux choix de réalisation. Le prototypage, défini par l'IEEE comme une approche privilégiant le développement de prototypes dès les premières étapes, apporte une solution à ce problème de dérive. Pour être efficace, une telle démarche doit s'intégrer dans un processus de développement.

Les travaux présentés dans ce document s'articulent sur deux axes : la mise en œuvre d'une méthodologie de prototypage d'application répartie exploitant les possibilités de méthodes formelles et la définition de techniques de prototypage d'Environnements de Génie Logiciel (EGL). Le second axe permet, entre autres, d'implémenter et d'évaluer les résultats fournis par le premier.

Dans les deux cas, nous nous plaçons dans une optique de prototypage par raffinements. L'application répartie ou l'EGL sont obtenus par itérations successives du processus de modélisation / évaluation / construction. Dans les deux cas, un environnement supporte et facilite les tâches d'élaboration du système. La méthode de conception d'applications réparties par prototypage a été expérimentée avec succès dans le cadre d'études industrielles et les techniques de prototypage d'EGL expérimentées avec succès dans FrameKit.

Abstract

Implementation and maintenance of industrial applications become more and more complex. In this context, a problem is the evaluation of complex systems. To help designers, software engineering methods and Computer Aided Software Engineering Environments (CASEE) have been introduced. These methods and tools are suitable to identify problems and describe solutions.

However, the specification of a solution and its implementation may be quite different due to implementation choices. Prototyping is defined by IEEE as an approach promoting prototype implementation as soon as possible. This approach is a solution to avoid the shift between a specification and its implementation. To be efficient, such an approach has to be integrated in the development process.

The work presented in this document focuses on two axes: the implementation of a prototyping based methodology for distributed systems relying on formal methods and the definitions of techniques for a quick implementation of CASEE. The second axis enables the implementation of the first axis.

In both cases, our approach is prototyping oriented. Distributed applications, as well as CASEE, are obtained from successive refinements of a modeling / evaluating / building procedure. In both cases, the process is supported by an environment that eases design and implementation. The prototyping based method for the design of distributed systems has been successfully experimented on industrial case studies. CASEE prototyping approach has been successfully implemented in FrameKit.