Université de Bretagne Occidentale

Master 2 Rechercher en Informatique

Rapport d'étude bibliographie :

Transformation de modèles UML vers programmes Fiacre

HENG Sotharith

heng.sotharith@gmail.com

Encatrant : Philippe DHAUSSY

Laboratoire Lab-STICC ENSTA Bretagne, 2 rue François Verny,

BREST (France), 29806, Cedex 9.

philippe.dhaussy@ensta-bretagne.fr

Table des matières

Table des figures

1 Introduction

Dans ces dernières années, le domaine de la modélisation et de la validation formelle de logiciels est un enjeu majeur du génie logiciel et de nombreux travaux universitaires ont exploré différents formalismes de modélisation, langages et outils pour concevoir des composants logiciels. Et les méthodes formelles ont contribué, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de model-checking [1, 2] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils (model-checkers) ont été développés dans ce but [3, 4, 5].

Pour accroître la pénétration des formalismes, comme par exemple de type UML, dans les processus industriels d'ingénierie système et logicielle, il est encore nécessaire de pouvoir proposer aux utilisateurs des techniques opérationnels d'analyses de ses modèles. Ceci implique de disposer de transformation de modèles adéquats, paramétrables ou configurables permettant de générer, à partir des modèles de conception UML des codes formels pour différents outils selon le type d'analyse que l'on veut mener. Par exemple, pour la vérification d'exigences fonctionnelles, il est nécessaire de traduire les modèles dans des codes exploitables par un model-checker. Mais il est bien connu que la sémantique des modèles peut varier selon l'interprétation donné par l'utilisateur. Il est donc pertinent de concevoir des règles de transformation qui soient paramétrées par des choix sémantiques à définir.

Sachant les importances de la méthode formelle dans le procès de développement du logiciel, en inspirant de travaux effectués sur les techniques de transformation de modèle UML, l'objectif du stage proposé est donc de concevoir des transformations qui ciblent le langage pivot Fiacre. Ce langage est exploitable par des model-checkers: CADP [4], TINA [3] développé au LAAS, et OBP Explorer [15] développé à l'ENSTA Bretagne.

2 Langage Pivot Fiacre

Le langage pivot Fiacre est un langage formel de description qui s'inspire des nombreux travaux de recherche, à savoir V-Cotre [13], NTIF [14], ainsi que les décennies de recherche sur la théorie de la concurrence et la théorie du système en temps réel. Fiacre est conçu à la fois comme le langage ciblé de transformation des modèles à partir de différents modèles tels que PDL, AADL, DSL, ou UML, et que le langage ciblé Fiacre pourra exploité dans les boîtes à outils de vérification, à savoir CADP [4], TINA [3], et OBP Explorer [15] (figure 1). On n'a pas exploité les syntaxes et les sémantiques du langage Fiacre dans cette étude bibliographie, pour en savoir plus détail vous pouvez consulter [6].

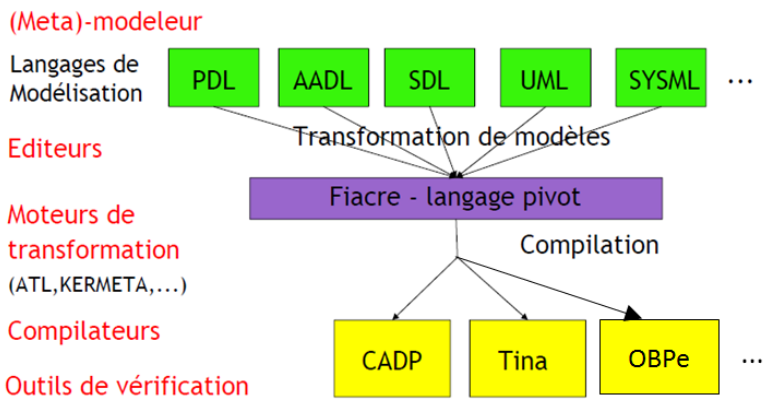

Figure 1 : Fiacre comme un langage ciblé de transformation

3 UML

Etant un langage de modélisation orientée objet, UML est largement adapté dans le procès de développements de logiciels. Les diagrammes UML sont représentés dans des vues distinctes pour modéliser les concepts particuliers du système. Généralement, ces diagrammes UML sont partagés en deux grands aspects : les aspects statiques et les aspects dynamiques. Les diagrammes statiques en UML, comme les digrammes de classes représentent les aspects statiques. Et les diagrammes dynamiques en UML, comme les diagrammes d'activités, ou les diagrammes d'états modélisent les aspects dynamiques.

UML offre trois notations complémentaires pour modéliser une vue opérationnelle du système [8, 9, 12]:

- Les diagrammes d'états assurent la représentation du comportement dynamique.
- Les collaborations spécifient l'échange des messages.
- Les diagrammes objet permettent de représenter la configuration initiale du système.

3.1 Diagrammes à traduire

En général, pour la traduction de modèle UML en un langage ciblé, les diagrammes de classes, les diagrammes d'états, et les diagrammes d'objet sont nécessaires [7, 12]. Les diagrammes de classes représentent la vue statique du système, les diagrammes d'états modélisent les comportements dynamiques du système, et les diagrammes d'objets représentent la configuration initiale du système. Dans la partie suivante, on va exploiter les règles de transformation des diagrammes de classes, des diagrammes d'états, et des diagrammes d'objets en Fiacre.

3.1.1 Diagrammes de classes

Dans la modélisation orientée objet, le diagramme de classes UML est qualifié comme le plus important élément parce qu'il représente la structure interne du système. Lors de transformation les éléments suivants sont pris en compte: les classes, les méthodes et événements, les attributs, le concept objet, les associations, et l'héritage [7, 9]. La figure 2 représente un exemple du diagramme de classes du système « *Home Alarm* » construit dans l'outil de modélisation Rhapsody.

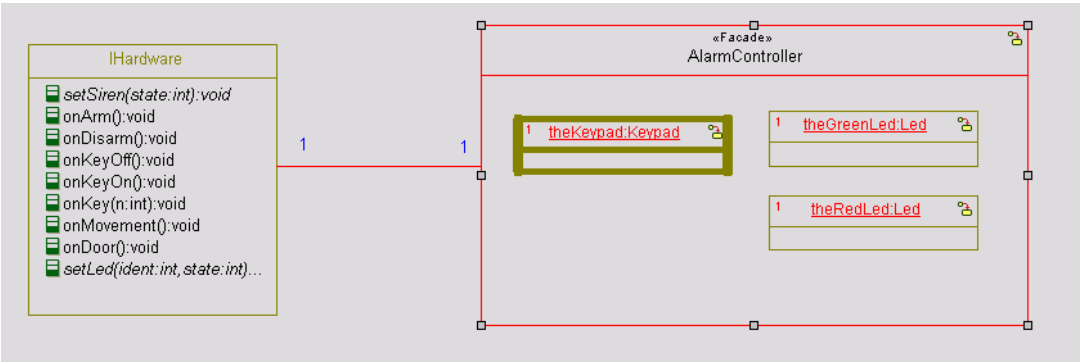

Figure 2 : Exemple de diagramme de classes

3.1.2 Diagrammes d'états

Les comportements dynamiques du système est représentés pas les diagrammes d'états (ou les machines à états). Les diagrammes d'états UML modélisent les comportements internes d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et

d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode). Lors de transformation les éléments suivants sont pris en compte: les états, les transitions, et les sous-machines d'états [7, 9]. La figure 3 est un exemple du diagramme d'états de la classe « *Led* ».

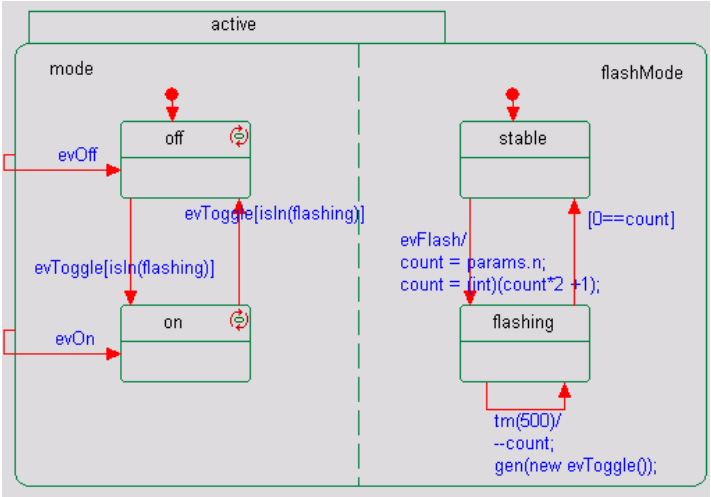

Figure 3 : Exemple de diagramme d'états

3.1.3 Diagrammes d'objets

Un diagramme d'objets représente les objets et leurs relations pour donner une vue fixé du système à un instant donné. Un objet est un instant d'une classe UML. Les objets réactifs communiquent entre eux via l'envoi asynchrone d'évènements et l'appel synchrone d'opérations. Le diagramme d'objets est plus concret que le digramme des classes et est utilisé pour modéliser la configuration initiale du système.

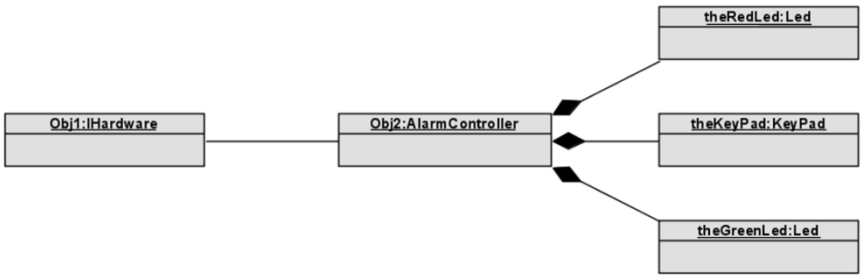

Figure 4 : Exemple de diagramme d'objets

4 Transformation UML en langage pivot Fiacre

Lors du processus de transformation UML en un langage formel Fiacre, la traduction de la structure du programme doit être effectuée en premier. Alors, les diagrammes de classes UML et les diagrammes d'objets vont être les premiers éléments à prendre en compte. On va exploiter la transformation des diagrammes de classes dans la partie suivante 4.1, des diagrammes d'états dans la partie 4.2, et des diagrammes d'objets dans la partie 4.3.

4.1 Traduction des diagrammes de classes

Un objet actif dans le modèle UML sera transformé en un processus Fiacre. Le comportement de cet objet actif est décrit par des méthodes inclus dans un diagramme de classe et dans un

diagramme d'état [7, 8]. Généralement, les classes UML peuvent être séparées en deux catégories : les classes actives possèdent des machines à états, et les classes inactives n'en possèdent pas. Les classes actives et inactives peuvent être des « singletons ». Un objet instancié d'une classe UML singleton sera interprétée par un processus avec une unique instance [8].

4.1.1 Classes

On a mentionné au-dessus que les classes UML sont différenciées en des classes actives et inactives. Chaque classe active est traduite par un processus autant de fois qu'elle sera instanciée. Dans cette partie, on va exploiter en détail leur transformation.

- **Classes actives**

Etant donné que la classe active UML possède des diagrammes d'états, il est donc nécessaire de les traduire directement en un processus Fiacre lors de son instanciation [7, 8]. On va exploiter la traduction des diagrammes d'états de chaque classe active dans la section 4.2. Dans l'exemple suivant, on montre la traduction d'une classe active en Fiacre:

```
process P() is
states state1, state2, state3
var v1 : int, v2: bool, v3: nat
init to state1
    from state1
    …
    to state2
    from state2
    …
    to state3
    from state3
    …
```

- **Classes inactives**

Etant donné que la classe inactive UML ne possède pas de diagrammes d'états, il sera donc traduit par un processus en Fiacre avec un seul état, et celui-ci demeurant vide (sans transitions) [8]. Dans le cas où tous ses attributs sont de types primitifs, on peut traduire la classe inactive UML par un type *record* en Fiacre [7]. On peut donc définir la règle de transformation des classes inactives UML en type *record* comme les suivants:

- Déclaration d'un type *record* contenant les attributs de la classe inactive correspondante.
- Création d'une variable du type précédemment déclaré dans le processus correspondant à la classe contenant.

Pour pouvoir lire et modifier les variables dans une classe inactive, il faut que la variable de type précédemment soit accessible globalement à partir du processus correspondant. Pour mieux compréhension, on reprend un exemple décrit dans [7] qui montre la traduction d'une classe inactive UML en IF2. Dans cet exemple, la classe C ne contient que des variables primitives et qui possède simplement des méthodes *get()* et *set()*.

```
Type C= record
        NbTour integer;
        NbState1 integer;
        NbState 2 integer;
        Endrecord;
```

```
Process B(0);
    Var Cobject C public;
    State state1 #start;
        …
        nexstate state2;
    endstate;
    state state2;
        …
    endstate;
endprocess;
```

Etant donné qu'il n'existe pas la notion de variable « public » dans Fiacre, on doit alors trouvez une solution équivalente qui permet de la modéliser. Pour cet instant, on pense à la déclarer en variable globale. On ne l'exploiter pas en détail ici, on va l'examiner et essayer de trouver la solution plus optimal pendent le stage.

4.1.2 Attributs

Les attributs de chaque classe seront traduits par les variables locaux dans chaque processus Fiacre correspondant. Les attributs peuvent être différenciés en deux types, ce sont les attributs de types primitifs, qui seront traduits directement par des variables simples correspondantes aux types basiques en Fiacre (int, nat, bool, array,…), et les attributs qui sont les références vers des instances des autres classes qui seront traduits par le *pid* [7, 8].

4.1.3 Opération

On peut différencier les opérations dans le modèle UML en deux catégories importantes : les opérations primitives et les *triggered* opérations. Lors de ses traductions en Fiacre, deux signaux sont déclarés pour chaque opération, ces signaux correspondent à l'appel et au retour de l'opération [8].

- **Opération primitive**

Normalement, l'appel d'une opération primitive est effectué dans le corps d'une méthode. Cet appel constitue un nouveau processus parce qu'il est traité par le receveur indépendamment de son état. Lors de traduction en Fiacre, un processus est donc créé pour chaque opération primitive [8]. Quand on effectue une opération primitive, un signal est envoyé par le processus appelant vers le processus qui effectue l'opération. Et quand l'opération termine, le processus appelé envoie un signal de retourne ver le processus appelant. Ainsi une route temporisée relie chaque processus opération au processus qui peut l'appeler [8].

- **Triggered opération**

Les *triggered* opérations sont synchrones, et leurs appels sont bloquants [8, 10]. Lors de transformation l'appel de *triggered* opération en Fiacre, on choisit de le traduire par des envois de signaux asynchrones (ou événements en UML) avec une procédure d'acquittement [8, 10]. Il est possible d'envoyer le signal synchrone dans Fiacre mais il n'est pas utilisé très souvent. Les envois de signaux asynchrones dans Fiacre sont effectués via des FIFO. On traduit chaque *trigerred* opération UML par un signal en Fiacre. Lorsqu'un processus reçoit une *triggered* opération, il envoie l'acquittement de réception au processus émetteur dans une transition entrante dans un état stable. Un état stable, est un état dans lequel aucune transition ne peut être tirée sans la réception d'un nouveau message.

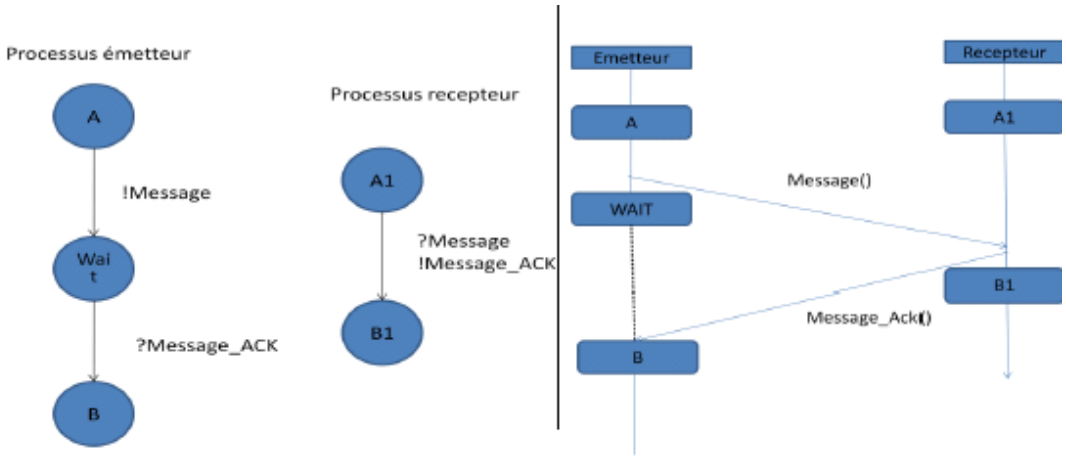

Figure 5 : Triggered opération sous forme de machine à états et son diagramme de séquence

4.1.4 Evénements

Un événement est quelque chose qui se produit pendant l'exécution d'un système ; contrairement aux opérations, ses envoyés sont indépendants de la définition des classes (*signal asynchrone*). Pour le traduire en Fiacre, un signal est déclaré pour chaque événement susceptible d'être reçu [8, 12].

4.1.5 Collaboration

Quand deux classes sont reliées par une collaboration, leurs instances peuvent s'échanger des messages. Pour préciser qu'une classe collabore avec une autre, une variable dans le processus généré par cette classe doit contenir l'identifiant de la classe qui collabore. La traduction de collaboration en Fiacre peut se faire comme suivante [8, 12]:

- Pour chaque collaboration, deux ports sont créées entre processus correspondants aux classes liées (pour les deux sens d'envoi). Ces ports sont traversés par des signaux d'envois d'événements et d'appel d'opérations.
- Pour une classe donnée, une variable d'accès est donc déclarée pour chaque classe à laquelle elle est liée. Sa taille est déterminée par la multiplicité portée par la collaboration.

4.1.6 Gestion des concepts Objet

Les concepts d'objets sont importants dans le langage modélisation orienté objet UML, il faut donc les interpréter en Fiacre. Cependant, la traduction des concepts d'objets en Fiacre est compliqué et difficile, il faut donc limiter les concepts supportés lors de sa traduction. Donc, l'héritage des machines d'états et la surcharge des opérations ne sont pas considérés [12].

- **Héritage**

Etant donné qu'il n'existe pas la notion d'héritage dans le langage Fiacre, il est donc nécessaire de définir des mécanismes spécifiques permettant la traduction de notion héritage des classes UML en Fiacre. Lorsque l'héritage des machines d'états et la surcharge des opérations ne sont pas pris en compte pendant la traduction, alors la méthode la plus simple consiste à recopier les propriétés de la classe mère dans chaque classe fille [7, 9]. Pour les classes héritant d'une classe abstraite, on réplique les associations et attributs qui proviennent de cette classe abstraite [8].

- **Accès concurrent aux attributs**

Il peut provoquer les problèmes de l'accès concurrent aux variables partagées car les variables des objets réactifs dans les diagrammes UML peuvent être modifiés mutuellement ou la modification des valeurs des variables se fait par l'intermédiaire des opérations primitives. Pour garantir la non-concurrence, il suffit donc de garantir qu'une primitive opération ne possède qu'une seule instance en même temps [8, 12].

- **Variables temporaires**

Pendent la traduction en Fiacre, seulement les variables temporaires suffisantes pour recevoir un message à la fois sont créées [8, 12]. On crée donc une collection de variables temporaires typées distinctes pour stocker les valeurs des paramètres des chaque message susceptible d'être reçu.

- **Processus opération**

Nous décrivons ici comment fonctionne plus précisément le processus opération créé lors de la présence d'une opération primitive.

  - **Paramètres de l'opération**

Lorsqu'un processus reçoit un appel d'opération primitive, les paramètres de ce message sont entrés au constructeur du processus opération créé. Par conséquent, à chaque paramètre d'un appel d'opération est associé un paramètre dans le processus qui exécute cette opération [8].

  - **Accès aux variables du processus classe**

Généralement, une opération primitive peut modifier les variables de l'objet qui l'appelle. De ce fait, le processus opération a besoin de la référence du processus qui l'a appelé [8, 12].

  - **Retour d'une valeur**

Les références de l'appelant sont indiquées au processus opération afin qu'il envoie l'acquittement [8, 12].

4.2 Traduction des diagrammes d'états

Un diagramme d'états UML est nécessaire pour définir le comportement d'un objet actif UML. Le diagramme d'état sera traduit directement en automate en Fiacre [7]. Chaque état dans le diagramme d'état correspond à un état dans automate en Fiacre.

4.2.1 Sous-machines d'états

Il est possible que les machines d'états contiennent des sous-machines. Quand une machine d'états arrive dans un état dans lequel il possède une sous-machine d'état à invoquer, alors cette machine reste en attente jusqu'à la sous-machine d'états termine ou il reçue un signal externe. La figure au-dessous représente ce cas [7]:

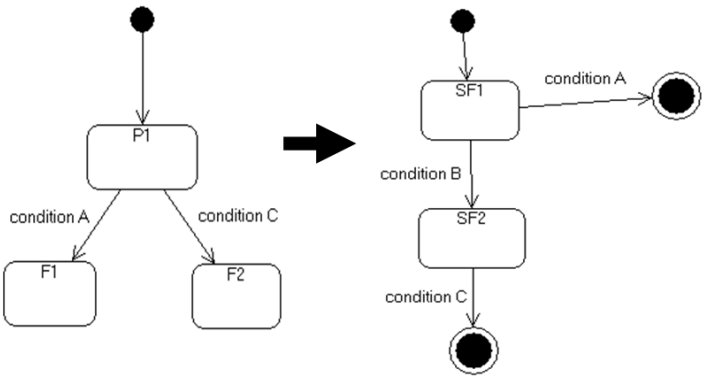

Figure 6 : Un exemple de machine d'états avec sous-machine d'états

La traduction de sous-machine d'états UML en Fiacre peut être réalisée comme suivant [8]:

- Un processus est créé pour chaque sous-machine d'états correspondant.
- Le processus correspondant à sous-machine d'états sera appelé lorsqu'on arrive dans un état qui possède une sous-machine d'état, et on place la machine d'état mère dans un état d'attente.
- La machine d'état mère passera en paramètre son *pid* lors de la création de la sous-machine d'état pour recevoir les messages d'acquittement et continuer son exécution le cas échéant. Il faut également prévoir le cas où le processus mère puisse passer à un autre état s'il reçoit un signal externe. Dans ce cas, dans le état *wait* du processus mère, il faut prévoir un état qui contiendrait un *kill()* du processus fils et le passage à un autre état.

4.2.2 Les transitions UML

Dans le diagramme d'états, le passage d'un état à un autre état s'effectue par des transitions. Ces transitions peuvent être déclenchées par des signaux, des événements, des conditions, ou par des gardes temporisées. La traduction des transitions UML en Fiacre peut se faire de la manière suivante :

- Lorsque dans une transition, nous avons une condition, on crée un état « S » dont les transitions sortantes traduisent les éléments de la condition. Dans l'article de (Philippe DHAUSSY, [8]), « S » est un état instable dans IF2. Etant donné que dans Fiacre n'existe pas la notion de l'état instable, on va représenter l'état « S » par un état simple dont son transition vers les autres états peut effectuer sans attendre la réception un ou des signaux.

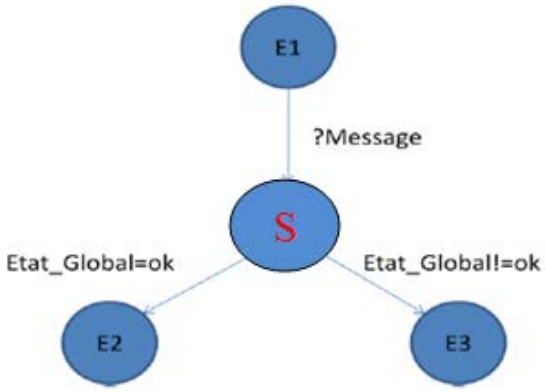

Figure 7 : La traduction de transition UML en Fiacre

4.2.3 Actions en entrée et en sortie d'un état

Normalement, il pourra avoir des actions à invoquer dans l'état d'arrivée de la transition. De ce fait, on doit alors prendre en compte les actions de chaque transition entrante créé. Cependant, la traduction des actions en entrée peut s'avérer laborieuse pour deux raisons principales [7]:

- Dans le cas de plusieurs transitions entrantes dans un même état, il faut que toutes les transitions fassent les mêmes opérations.
- Dans le cas de plusieurs transitions entrantes engendrer une multiplication substantielle des états, ils peuvent être dans l'état d'attente.

La figure au-dessus nous montre les problèmes lors les actions en entrée d'un état.

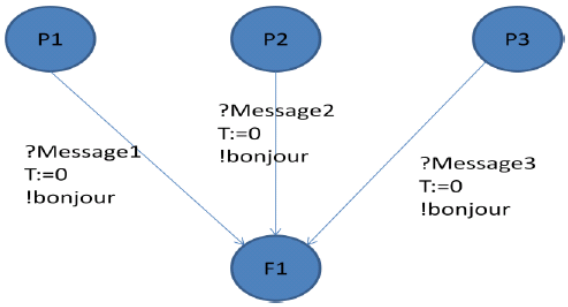

Figure 8 : Les actions en entrée d'un état

La traduction de ces modèles en Fiacre peut se faire en mettant un état « S » juste avant l'état « destination ». Dans [7], « S » est un état instable en IF2. Comme on a discuté dans la partie précédente qu'il n'existe pas la notion état instable en Fiacre, on peut donc utiliser la même solution en représentant « S » par un état simple en Fiacre dont son transition vers les autres états peut effectuer sans attendre la réception un ou des signaux.

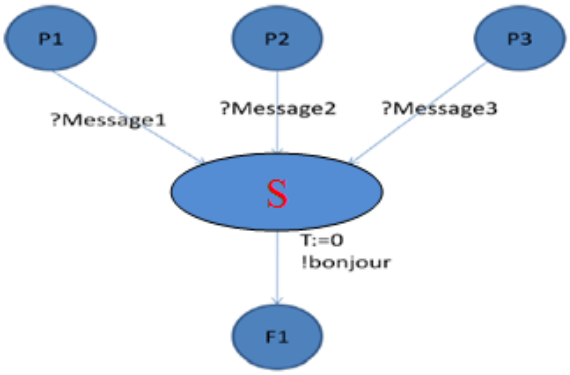

Figure 9 : Traduction des actions en entrée d'un état en Fiacre

Alors, on est sûr que les actions en entrée (ici T=0 envoi du message « bonjour ») seront effectués sans délai de temps, c'est-à-dire juste après la réception des événements Message{i}. Ainsi la sémantique du système sera respectée.

En revanche, nous ne pouvons pas effectuer la même transformation pour les actions en sortie d'un état [7]. En effet, s'il existe plusieurs transitions sortantes, celles-ci ont forcément des facteurs déclenchant différents et peuvent avoir des états de destination différents. Or, si nous créons un état « S » de sortie, et que nous forçons l'ensemble des transitions à passer par cet état et à faire les mêmes actions, la terminaison de la transition sera le même pour tous.

Nous ne pouvons alors donc plus différencier les différentes transitions possibles. Ainsi, lorsqu'un état possède des actions en sortie d'un état, nous les inscrivons dans les transitions directement.

4.2.4 Appel d'opérations

En général, il peut avoir des appels des opérations dans les actions en entrée et en sortie des états. Le processus qui appelle une opération doit attendre que celle-ci soit terminée avant de pouvoir continuer son évolution.

Voici au travers d'un exemple, l'implémentation que nous proposons pour gérer l'opération FAIRE_QQCH (Boolean …) dans une transition [7]:

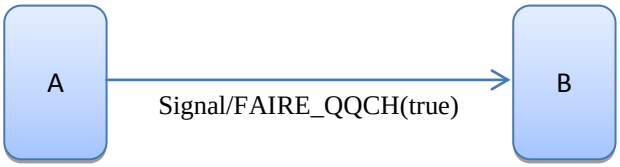

Figure 10 : Appel d'opération dans une transition en UML

On les traduit en Fiacre par deux processus. L'un va gérer l'évolution globale de la machine d'états, l'autre va effectuer l'opération. Une fois que le processus a effectué l'opération et la termine, il envoie un message d'acquittement au processus appelant pour que celui-ci puisse continuer son exécution.

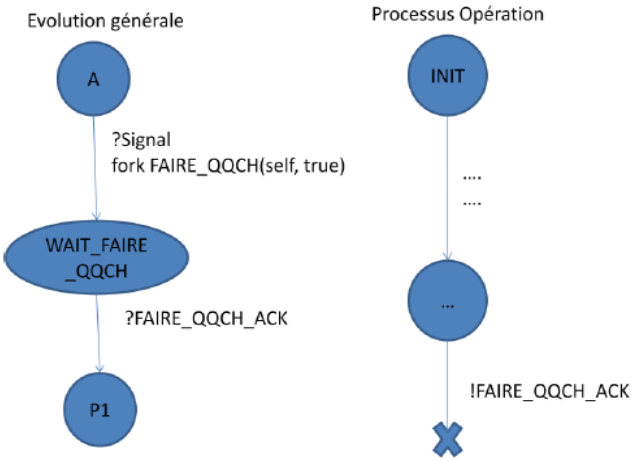

Figure 11 : Traduction d'un appel d'opération

Nous déduisons alors les règles de transformation suivantes [7, 8] :

- Pour chaque opération nous créons systématiquement un processus qui se chargera d'effectuer les actions correspondantes. A la fin de l'opération, il enverra un message d'acquittement au processus appelant et s'arrêtera.
- Si dans une transition, un appel d'opération est présent, le processus courant d'évolution crée le processus correspondant et se place dans un état d'attente de fin de l'opération. Lors de la création, il passera son *pid* en premier paramètre formel et

ensuite les autres paramètres nécessaires dans l'ordre original. Le processus courant ne pourra poursuivre son évolution qu'après réception du message d'acquittement.

4.3 Traduction des diagrammes d'objets

Lors de transformation UML en Fiacre, le digramme d'objets est le premier élément à traduire parce qu'il représente la configuration initiale du système. Une classe d'un objet actif et son diagramme d'état sera traduite pour construire le code du processus Fiacre. La relation entre les objets sera traduite par sa collaboration. Et le ou les diagrammes d'objets seront exploites pour identifier le nombre de processus à instancier dans la partie Component de Fiacre [12].

5 Conclusion

Comme vu dans cette étude bibliographie, les diagrammes de classes, les diagrammes d'états et les diagrammes d'objets sont nécessaires et suffisants pour la traduction des vues statique et des comportements dynamiques du modèle UML. Les diagrammes d'objet représentent la configuration initiale du système. Dans la section 4.1.6, l'héritage des machines d'états et la surcharge des opérations ne sont pas considérés car ceux-ci vont faire augmenter la complexité du processus de transformation

Finalement, on va s'inspirer de ces règles pour la traduction du modèle UML Rhapsody en Fiacre. Le dispositif de lecture des fichiers XMI du modèle UML Rhapsody va être utilisé pour leur analyse. On va aussi examiner et trouver la mécanique la plus optimale qui permet de modéliser la notion de variable « *public* » en Fiacre comme on a discuté dans la partie 4.1.1. On a aussi présenté la notion de « *pid* » dans le procès de transformation. Pour cet instance, on n'a pas exploité comment on peut le déclarer et le référencer en Fiacre, parce que dans les références de langage Fiacre ne couvrent pas cette notion, on va les étudier en détail pendent le stage.

Références

[1] Queille J.-P., Sifakis J., « Specification and verification of concurrent systems in CESAR », Proceedings of the 5th Colloquium on International Symposium on Programming, Springer- Verlag, London, UK, p. 337-351, 1982.

[2] Clarke E., Emerson E., Sistla A., « Automatic verification of finite-state concurrent systems using temporal logic specifications », ACM Trans. Program. Lang. Syst., vol. 8, n° 2, p. 244- 263, 1986.

[3] Berthomieu B., Ribet P.-O., Verdanat F., « The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », International Journal of Production Research, 2004.

[4] Fernandez J.-C., Garavel H., Kerbrat A., Mounier L., Mateescu R., Sighireanu M., « CADP : A Protocol Validation and Verification Toolbox », CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification, Springer-Verlag, London, UK, p. 437-440, 1996.

[5] Larsen K. G., Pettersson P., Yi W., « UPPAAL in a Nutshell », International Journal on Software Tools for Technology Transfer, vol. 1, n° 1-2, p. 134-152, 1997.

[6] Bernard Berthomieu, Frédéric Lang, « Le langage pivot asynchrone Fiacre », réunion WP3 Topcased - LAAS - Toulouse, 13 novembre 2007

[7] Maxime FROMENTIN, « Identification des mécanismes de transformation entre UML et IF (V1) », 7 novembre 2008

[8] Philippe DHAUSSY, « Plugin UML_to_IF », version 0: 08/10/2008

[9] Philippe DHAUSSY, « La transformation UML vers IF Base de transparents », version du 21 mai 2005

[10] Julien AUVRAY, Mémoire de projet de fin d'études « Validation formelle de modèles UML », promotion 2007

[11] Maxime FROMENTIN, « Identification des mécanismes de transformation entre UML et IF (V4) », version du 26 février 2009

[12] Habart OLIVIER, Rapport de PFE « Modélisation et validation formelles de propriétés en environnement », 23 juillet 2004

[13] B. Berthomieu, P.-O. Ribet, F. Vernadat, J. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Fi-lali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert, «Towards the verification of real-time systems in avionics: the Cotre approach », volume 80 of Electronic Notes in Theoretical Computer Science, pages 201–216. Elsevier, June 2003.

[14] Hubert Garavel and Fr´ed´eric Lang, «NTIF: A general symbolic model for communicating sequential processes with data », volume 2529 of Lecture Notes in Computer Science, pages 276–291. Springer Verlag, November 2002.

[15] Philippe Dhaussy, Jean-Charles Roger, « OBPe (OBP Explorer) Document », http://www.obpcdl.org, 26 octobre 2011