# Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD

Frédéric Jouault[1], Ciprian Teodorov[2], Jérôme Delatour[1], Luka Le Roux[2], Philippe Dhaussy[2]

[1] ESEO-Trame, Angers, France
[2] UEB, Lab-STICC Laboratory UMR CNRS 6285, ENSTA-Bretagne, France

## Résumé

La complexité des systèmes embarqués continue d'augmenter. Leur développement nécessite donc un processus de développement rigoureux pour répondre aux normes de certification très strictes. Dans ce contexte l'utilisation des méthodes formelles pour la vérification promet des gains importants en termes de fiabilité et temps de développement. En revanche, le fossé sémantique entre les langages utilisés par le domaine industriel et ceux pris en compte par les outils formels est une barrière réelle pour l'adoption de ces derniers. Dans cet article, nous introduisons une chaîne de transformation modulaire permettant l'obtention de modèles formels à partir d'un formalisme d'entrée basé sur UML. Les modèles ainsi obtenus permettent la vérification de propriétés.

## 1. Introduction

Dans le domaine des systèmes embarqués, les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. En raison de ces contraintes, les architectures logicielles embarquées sont soumises à un processus de certification qui nécessite un développement très rigoureux. Toutefois, en raison de la complexité croissante des systèmes, leur conception reste une tâche difficile.

Dans le but d'accroître la fiabilité des logiciels, les méthodes formelles contribuent à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. À cet effet, des méthodes de validation formelle de comportement de modèles ont été explorées depuis plusieurs années par de nombreuses équipes de recherche et expérimentées par des industriels [5]. Mais aujourd'hui, les logiciels embarqués intègrent des fonctionnalités de plus en plus complexes ce qui rend difficile la mise en œuvre de ces méthodes. Malgré la performance croissante des outils de model-checking, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel.

Pour la capture des besoins métiers et le développement des solutions embarqués les industriels utilisent des langages standardisés avec un grand pouvoir d'expression tel qu'UML ou SysML. Entre ces langages et les formalismes utilisés par les outils formels, le fossé sémantique est souvent trop large pour permettre l'analyse des modèles produits.

Pour contribuer à répondre à ce problème, nous présentons une chaîne de transformation automatique de modèles. Cette chaîne prend en entrée des modèles spécifiés dans un sous-ensemble du langage UML. A travers une série de transformations, un modèle formel exprimé

dans le langage Fiacre [2] est obtenu. Les deux problèmes majeurs qui devront être traités lors d'une telle transformation sont la **variabilité** des langages d'entrée et la **complexité** de la transformation. Pour résoudre le premier, nous introduisons un format pivot nommé *textual UML* (tUML) qui a pour vocation d'être un format explicite et non ambiguë pour la représentation de tous modèles conformes au standard UML. Un deuxième langage intermédiaire, nommé *ABCD*, est introduit pour rendre la chaîne de transformation robuste et modulaire. Ce langage permet, de cibler des langages formels comme, par exemple, Fiacre.
L'article est organisé comme suit. Dans la section 2, nous introduisons le formalisme UML et la vérification de modèles basée sur les contextes avant de donner une vue schématisée de notre approche. En section 3 nous présentons les principales restrictions imposées sur les modèles UML pour la vérification. Ensuite, l'article est orienté sur la description des deux langages intermédiaires (tUML section 4 et ABCD section 5) qui structurent notre chaîne de transformation. La section 6 conclue notre article en présentant des axes de développement futurs.

# 2. Modélisation et vérification

## 2.1 UML

Le langage de modélisation unifié UML[1], normalisé par l'OMG, est de plus en plus utilisé dans l'industrie. Il s'agit d'un langage semi-formel graphique qui est utilisable pour la capture des besoins durant les phases de spécification et de conception. Ce type de formalisme facilite le dialogue entre experts.
UML propose plus d'une douzaine de diagrammes permettant de modéliser les différents aspects structurels et comportementaux des systèmes. De plus, un mécanisme d'extensions basé sur des profils permet d'adapter UML à des besoins spécifiques. Parmi les profils standards, nous pouvons citer MARTE[2] pour les applications embarquées et temps réel ainsi que SysML[3] pour les aspects système.

## 2.2 Vérification basée sur les contextes

Les techniques de vérification formelle type model-checking souffrent du problème de l'explosion de l'espace d'états des modèles, induite par la complexité interne du logiciel qui doit être validé. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent de façon asynchrone avec des environnements impliquant un grand nombre d'acteurs. Un moyen pour restreindre les comportements du modèle est de spécifier les comportements de son environnement. Ceux-ci correspondent à des phases opérationnelles bien identifiées, comme, par exemple, l'initialisation, les reconfigurations, les modes dégradés, etc.
Face à ces constats, l'outil OBP a été proposé [6] comme cadre structurant permettant à l'utilisateur de décrire formellement le système étudié en utilisant le langage formel Fiacre [2] ainsi que les contextes de preuves au travers de l'utilisation du langage de description de contextes, CDL.

[1] http://www.omg.org/spec/UML/2.4.1/
[2] http://www.omg.org/spec/MARTE/
[3] http://www.omg.org/spec/SysML/

Dans ce cadre, le langage Fiacre a été choisi pour sa vocation d'être un langage haut niveau conçu pour être proche des différents formalismes métier tel que AADL et UML [2]. Par contre, dans le cas d'UML notamment, la pratique montre l'existence d'un fossé sémantique assez important entre ces deux formalismes. Ce problème est d'autant plus important que les pratiques industrielles varient énormément en terme de méthodologie. Cette problématique bien identifiée est exacerbée par l'existence des points de variations sémantiques du langage UML.

## 2.3 Proposition

Notre objectif est de tirer parti des avantages de la vérification basée sur les contextes tout en partant de modèles définis en UML, plus faciles à définir que des modèles Fiacre. Il serait en théorie possible de développer des outils de vérification basés sur les contextes directement capables d'explorer des modèles UML. Cependant, le coût de tels développements serait relativement élevé. Cette approche devient d'autant plus impraticable que l'on multiplie les langages auxquels on veut l'appliquer (e.g., SysML, ou plus simplement une nouvelle version d'UML).

En conséquence, l'approche que nous avons retenue consiste à transformer les modèles UML définis par les utilisateurs en modèles Fiacre exploitables par les outils existants tels qu'OBP. L'utilisation d'une telle transformation ne résout cependant pas tous les problèmes. Par exemple, il faut aussi être capable de remonter les informations produites par les outils de vérification au niveau UML de manière à ce qu'elles puissent être facilement interprétables par le concepteur. De plus, la réalisation de cette transformation n'est pas triviale, ainsi que nous allons le voir par la suite.

## 2.4 Architecture de la transformation UML vers ABCD

Lors du passage d'un modèle UML à un modèle Fiacre, deux problèmes principaux à résoudre sont :

- La **variabilité** des outils d'édition des modèles UML qui n'implantent pas tous les mêmes métamodèles et format de sérialisation.
- Le **large fossé sémantique** entre UML et Fiacre avec en amont des objets actifs et en aval des automates temporisés.

Afin de prendre en compte ces deux aspects, l'architecture retenue se décompose en deux étages (voir Figure 1~~Figure 1~~) : un frontal (*front end*) et un terminal (*back end*).

L'étage frontal résout les problèmes de variabilité en transformant les formats des différents outils UML vers un format unique. Pour ce dernier, nous avons choisi l'environnement Eclipse UML 2.4 car il s'agit d'une implantation relativement fidèle du standard UML pour lequel il existe de nombreux outils (*e.g.*, l'éditeur Papyrus). Cet étage est principalement défini à l'aide du langage de transformation de modèles ATL et sa structure correspond à une version orientée de celle décrite dans [9].

L'étage terminal s'appuie sur le langage ABCD, intermédiaire entre UML et Fiacre, de manière à réaliser la traduction en deux phases plus simples.

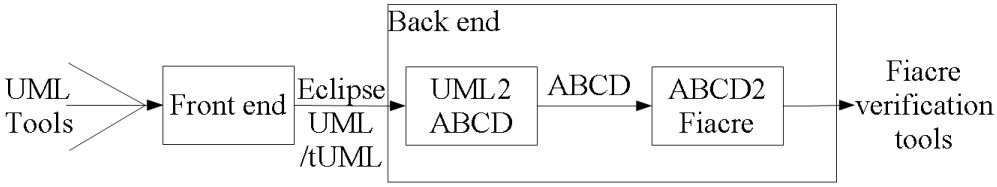


**Figure 1 Architecture de la transformation UML vers Fiacre**

# 3. Sous-ensemble UML

Ainsi qu'évoqué précédemment, nous nous intéressons à la modélisation de systèmes embarqués en UML afin de vérifier des propriétés sur nos modèles. Ceux-ci sont exploités à un niveau de conception générale. Le système à l'étude y est représenté comme un assemblage d'objets actifs.
Puisque le champ d'application d'UML est plus large que le domaine qui nous intéresse, nous n'avons pas besoin de tous les éléments du langage. De plus, notre objectif nécessite une sémantique précise qui permette d'interpréter de manière non ambiguë nos modèles. Or, le standard UML définit plusieurs sémantiques possibles [3,4] via des points de variation sémantique. Nous devons donc décider quelle variante d'UML nous conservons et résoudre toutes les incohérences.
En termes de diagrammes, nous en retenons trois qui sont typiquement utilisés pour la modélisation de systèmes embarqués :

- Le **diagramme de classes** permet d'une part de définir les classes actives (i.e., les types des objets actifs) avec la liste des messages qu'ils peuvent recevoir et émettre ainsi que leurs données internes. D'autre part, ce diagramme sert aussi à modéliser les structures de données passives. La Figure 2~~Figure 2~~ donne un exemple avec la classe *Button* dont les instances peuvent recevoir les messages *button_pressed*, *button_released* et *problems_detected* et émettre le message *pressed* que la classe *ControlsManager* peut recevoir.
- Le **diagramme de structure composite** représente le système sous la forme d'objets actifs reliés par des connecteurs correspondant à des canaux de communication. L'exemple de la Figure 4~~Figure 4~~ montre que notre système à l'étude (*SUS)* comporte deux instances *theOnButton* et *theOffButton* de la classe active *Button* qui sont connectées à une instance de la classe active *ControlsManager*.
- Le **diagramme d'état** modélise le comportement d'un objet actif sous la forme d'une machine à états. Chaque bouton instance de la classe *Button* a, dans notre exemple, le comportement représenté à la Figure 4~~figure 4~~. Son comportement commence dans l'état relâché *RELEASED* et passe à l'état pressé *PRESSED* lorsque le message *button_pressed* est reçu. Le bouton envoie le message *pressed* au *ControlsManager* lorsqu'il passe à l'état pressé puis toutes les cent millisecondes tant qu'il y reste. Enfin, quand l'un des messages *button_released* ou *problems_detected* est reçu dans l'état pressé, le bouton repasse dans l'état relâché.

**Commentaire [1]:** en réalité, sous le forme de "parts"... dont le lien avec les objets actifs est assez délicat à expliquer succintement

Même en restreignant l'utilisation d'UML à ces trois diagrammes, il reste des redondances et des ambiguïtés. Nous avons donc décidé d'ajouter les contraintes suivantes :

- La communication entre objets actifs se fait par échange de signaux asynchrones.
- Des méthodes peuvent être utilisées à l'intérieur d'un même objet actif ou pour les objets passifs.

- Les ports de communication ne sont pas actifs. C'est-à-dire qu'ils ne servent qu'à faire passer des messages sans filtrage.

Nous excluons notamment l'utilisation de méthodes pour la communication entre objets actifs car cela serait redondant avec l'utilisation des signaux.

Il est important de noter ici qu'il existe un sous-ensemble d'UML appelé fUML dont la sémantique est standardisée de manière relativement précise. Cependant, ce sous-ensemble ne correspond pas aux pratiques de modélisation de systèmes embarqués les plus répandues. Notamment, fUML n'offre pas les classes structurées, ni les interfaces et utilise les activités pour modéliser le comportement plutôt que les machines à état. Cela étant dit, notre sous-ensemble d'UML n'est pas le seul possible et nous n'excluons pas qu'il soit possible d'utiliser fUML ou tout autre sous-ensemble pertinent d'UML pour la modélisation des systèmes embarqués.

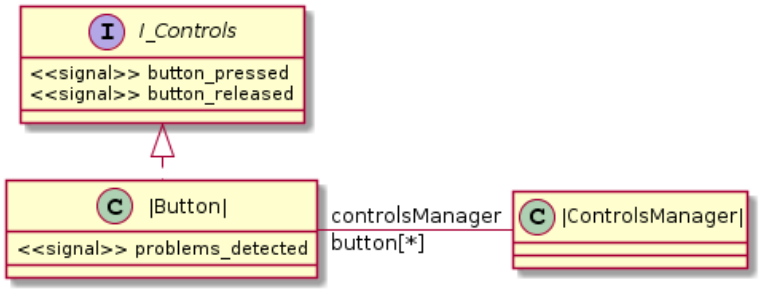


Figure 2 Exemple de diagramme avec deux classes active pour bouton et contrôleur

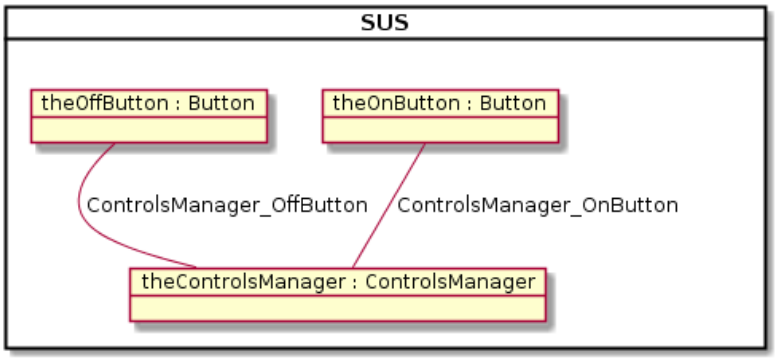


Figure 3 Diagramme de structure composite d'un système avec deux boutons et un contrôleur

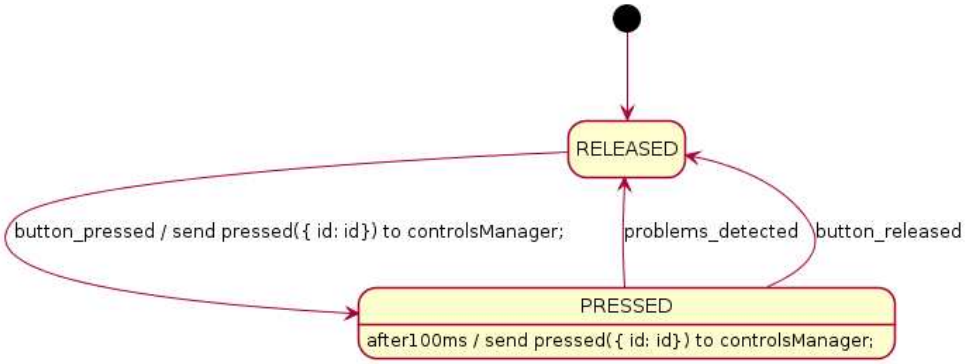


**Figure 4 Diagramme d'état du comportement des boutons**

# 4. tUML

En ingénierie dirigée par les modèles, les modèles ne sont pas que de simples dessins (ou diagrammes). Au contraire, le plus important est de pouvoir les traiter automatiquement (i.e., les transformer). Par exemple, transformer des modèles peut avoir pour objectif de générer du code. D'autres applications qui nous intéressent plus particulièrement ici permettent de vérifier des propriétés sur des modèles. Pour rendre ces transformations possibles, chaque modèle se conforme à un langage précis et syntaxiquement non ambigu appelé métamodèle. Un diagramme n'est alors qu'une représentation d'une partie d'un modèle.

En conséquence, bien que nous ayons décrit ci-dessus notre sous-ensemble d'UML en termes de diagrammes, sa définition précise est en réalité une restriction du métamodèle UML standard[4]. Dans le cas d'UML, chaque diagramme ne représente qu'une partie relativement réduite du modèle sous-jacent. Ceci entraîne deux problèmes principaux :

1. **Pas de vue de l'intégralité du modèle.** Chaque aspect a sa propre syntaxe, et les différents diagrammes ne sont reliés entre eux que par des références qui apparaissent textuellement.
2. **Détails cachés.** Les éditeurs UML graphiques cachent certaines propriétés du modèle en dehors du diagramme (e.g., dans des feuilles de propriétés accessibles par une séquence de clics) et parmi l'ensemble des propriétés définies dans UML. Or nous n'en utilisons qu'une partie qui se retrouve diluée.

Afin de répondre à ces problèmes, nous avons défini une syntaxe textuelle pour notre sous-ensemble d'UML. Nous l'appelons tUML pour *textual* UML. Cette représentation n'a pas pour objectif de remplacer les diagrammes graphiques pour l'utilisateur. tUML sert principalement à faire le lien avec tout éditeur en amont, et à simplifier l'analyse du modèle. Les avantages de tUML sont :

- **Les détails sont visibles.** La représentation textuelle intègre tous les éléments du modèle sans en cacher.
- **L'ensemble est homogène** car l'intégralité du modèle se retrouve consolidée, sans avoir besoin de consulter plusieurs diagrammes.
- **La structure du modèle est respectée** puisque tUML suit, de très près, le métamodèle UML.

[4] http://www.omg.org/spec/UML/2.4.1/

- **La redondance est réduite** car il n'est notamment pas nécessaire de répéter des informations sur plusieurs diagrammes.

Le principal inconvénient de tUML est qu'il est relativement verbeux : en rendant tous les détails visibles, l'essentiel est dilué. Nous gagnons une vue de l'intégralité du modèle, mais ce n'est pas vraiment une vue d'ensemble. Néanmoins, dans le futur, les modèles tUML sont voués à être générés à partir des modèles construits à l'aide des différents éditeurs UML.

Le code de la Figure 5~~Figure 5~~ illustre l'utilisation de tUML en l'appliquant à l'exemple du bouton présenté ci-dessus. La première partie (avant *stateMachine*) correspond au diagramme de classe de la Figure 2~~Figure 2~~. La seconde partie correspond au diagramme d'état de la Figure 4~~Figure 4~~. Nous notons notamment que la machine à état se situe à l'intérieur de la classe, ce qui n'était pas visible sur les diagrammes précédents, de même que le langage dans lequel les actions sont spécifiées.

```
class |Button| behavesAs SM implements I_Controls
        receives
                problems_detected_R(problems_detected) {
        private itsControlsManager[1-1] : ControlsManager in ControlsManager_Button;
        private id[1-1] : Integer;

        stateMachine SM {
                region MainRegion {
                        Initial -> RELEASED;
                        RELEASED -> PRESSED : button_pressed_SE /
                                opaqueBehavior = 'send pressed(id) to itsControlsManager;'
                                        in ABCD; ;
                        PRESSED -> PRESSED : after100ms /
                                opaqueBehavior = 'send pressed(id) to itsControlsManager;'
                                        in ABCD; ;
                        PRESSED -> RELEASED : problems_detected_SE /;
                        PRESSED -> RELEASED : button_released_SE /;
                        initial pseudoState Initial;
                }
        }
}
```

**Figure 5 Comportement du bouton, listing tUML**

La Figure 6~~Figure 6~~ donne une vue d'ensemble des outils que nous avons construit autour de tUML : conversion depuis des éditeurs tels que Rhapsody (Papyrus étant supporté nativement), diagnostic du modèle, transformation vers PlantUML pour visualisation et enfin transformation vers ABCD pour vérification.

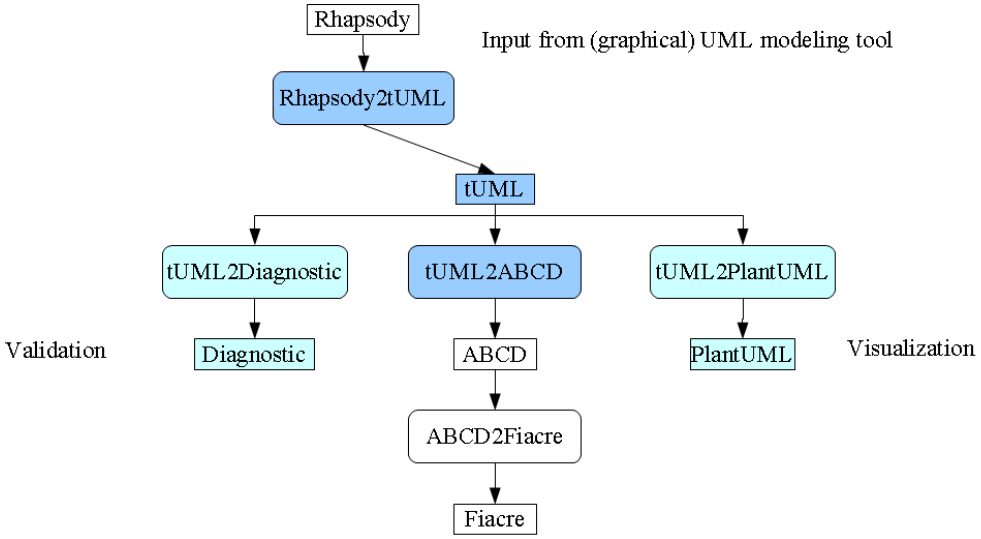


**Figure 6 Vue d'ensemble de l'outillage tUML**

Dans le but de simplifier, dans un premier temps, la traduction des modèles UML vers les outils de vérification, notre transformation tUML vers ABCD ajoute les restrictions suivantes à notre sous-ensemble UML : pas d'états composites, pas de région orthogonale (tout le parallélisme est modélisé par des objets actifs explicitement identifiés) et le langage d'action est ABCD. Ces restrictions ont vocation à être levées progressivement lors des améliorations ultérieures de nos outils.

En majorité, les utilisateurs d'UML en retiennent sa syntaxe graphique plutôt que son métamodèle. Ils seront donc peut-être surpris par tUML. Cependant, ce n'est pas la première fois qu'une syntaxe textuelle est définie pour UML. L'OMG a notamment standardisé plusieurs "cousins" de tUML :

- **XMI**[5] définit une traduction systématique de tout métamodèle en schéma XML, ainsi que de tout modèle en document XML. Il s'agit d'un format dédié à l'échange de modèles entre outils et il n'est pas adapté à un lecteur humain.
- **HUTN**[6] semble combler ce manque puisqu'il promet la dérivation automatique d'une syntaxe utilisable par un être humain à partir de n'importe quel métamodèle. Cependant, cette généricité mène à des syntaxes relativement verbeuses.
- **Alf**[7] définit une syntaxe textuelle standard adaptée à fUML. L'OMG a donc bien compris que HUTN n'est pas approprié pour ce type d'application. Malheureusement, Alf est limité au sous-ensemble fUML qui n'est pas adapté à l'embarqué et se trouve donc être relativement éloigné de notre sous-ensemble (voir section précédente). En revanche, une extension possible de notre travail sur tUML serait de l'aligner avec Alf et de le proposer à l'OMG.

[5] http://www.omg.org/spec/XMI/
[6] http://www.omg.org/spec/HUTN/
[7] http://www.omg.org/spec/ALF/

Par ailleurs, parmi les propositions non standards d'UML textuel[8], nous pouvons mentionner :

- **TextUML**[9] est relativement proche de tUML mais ne suit pas le métamodèle UML d'aussi près (notamment pour son langage d'action).
- **PlantUML**[10] a l'avantage d'être moins verbeux et plus permissif que les approches citées précédemment. Cependant, son but est uniquement de générer automatiquement des diagrammes et il ne suit pas le métamodèle standard.

# 5. ABCD

Afin de maîtriser le fossé sémantique entre les modèles tUML et le langage d'automates temporisés Fiacre [2], nous avons défini un langage appelé ABCD[11]. Le but de cette représentation intermédiaire est d'offrir un format pivot formel et stable qui permet la conversion de différents formalismes d'entrée (e.g., UML, SysML, SDL) vers des outils de vérification de modèles tel qu'OBP.

L'usage d'un langage pivot dans la chaîne de transformation la rend plus modulaire. De plus cela réduit le nombre de transformations unitaires à écrire entre les formalismes d'entrée et celui ciblé. Le langage ABCD peut-être vu de deux manières différentes et complémentaires. D'une part, ABCD permet de descendre le niveau d'abstraction des langages d'entrée vers le formalisme d'automates de Fiacre. D'autre part, il remonte le niveau d'abstraction du Fiacre vers les langages d'entrée.

Les unités comportementales du modèle sont représentées par la notion de processus, équivalente à celle d'automate temporisé. Un opérateur de composition asynchrone de processus permet de faire émerger le comportement du modèle. La principale différence par rapport à Fiacre est qu'ABCD offre un ensemble de primitives plus riches ainsi que certaines facilités d'expression (i.e., sucre syntaxique). Voici quelques exemples notables :

- la communication par envoi asynchrone de messages à travers des "event pool" ;
- des horloges (timers) comme primitives, qui peuvent être soit discrètes (permettant de simuler l'écoulement du temps sous la formes d'instant discrets), soit continues (permettant de simuler l'écoulement du temps suivant une loi monotone et continue telle que celle des automates temporisés) ;
- des primitives de contrôle d'accès aux ressources partagées telles que les sémaphores et les sémaphores d'exclusion mutuelle.

En terme de facilité d'expression on note notamment :

- le nommage explicite des instances dans la composition de processus ;
- la création d'associations explicites entre les variables et les ports de communication dans la composition de processus ;
- la notion explicite de type de canal de communication (channeltype) permettant la spécification des différentes sémantiques de communication.

[8] http://modeling-languages.com/uml-tools/#textual
[9] http://sourceforge.net/apps/mediawiki/textuml/
[10] http://plantuml.sourceforge.net/
[11] ABCD est un acronyme de **A**rchitecture, **B**ehavior, **C**ommunication and **D**ata

De plus, le langage ABCD est conçu d'une manière complémentaire et composable par rapport au langage CDL qui permet la description de contextes de preuve, ainsi que par rapport à un langage de contraintes (i.e., le langage CCSL[12] intégré dans la normé UML-Marte [1]).
À un autre niveau, le langage ABCD peut aussi être vu comme une plate-forme d'exécution abstraite permettant de représenter à haut niveau les détails d'exécution d'une plate-forme physique. De plus, le non-déterminisme permet de représenter non seulement une exécution du modèle sur une plate-forme mais un ensemble d'exécutions suivant une sémantique d'entrelacement concurrent de comportements. Dans ce cadre il faut mettre en évidence l'intérêt réel d'étudier un langage de contraintes tel que CCSL [1] pour pouvoir réduire l'ensemble des comportements abstraits à travers un formalisme complémentaire à celui utilisé pour la modélisation. Ceci correspond effectivement à une phase de déploiement de l'application sur une plate-forme donnée.
Pour illustrer la syntaxe ABCD nous nous rapportons à l'exemple du bouton présenté plus tôt. Le code de la Figure 7~~Figure 7~~ présente les types de messages échangés par le système. La Figure 8~~Figure 8~~ représente la composition parallèle d'objets actifs encodée en ABCD, en particulier les 3 canaux définis sont des «event pools» et non pas de canaux de synchronisation. L'encodage ABCD du diagramme d'états de la Figure 4~~Figure 4~~ est présenté dans la Figure 9~~Figure 9~~. On note déjà, pour cet exemple simple, un gain (de facteur 2) en terme de lignes de code par rapport au code Fiacre équivalent. De plus, avec l'utilisation de concepts UML plus riches (tel que la possibilité de garder un événement dans la queue de réception), ce gain ne peut qu'augmenter.

```
type ButtonEvents is union
        problems_detected   //localy defined event
|       button_pressed
|       button_released
end union

type ControlsManagerEvents is union
        pressed of int
end union

channeltype ToControlsManager is eventpool buffersize: 2 of ControlsManagerEvents
channeltype ToButton is eventpool buffersize: 2 of ButtonEvents
```

**Figure 7 ABCD types pour l'exemple des boutons**

```
channel itsOnButton : ToButton
channel itsOffButton : ToButton
channel itsControlsManager : ToControlsManager
par
        [itsOnButton]           theOnButton:Button(1)       [itsControlsManager]
||      [itsOffButton]          theOffButton:Button(2)      [itsControlsManager]
||      [itsControlsManager]    theControlsManager:ControlsManager
end par
```

**Figure 8 Composition de processus ABCD, correspondant au diagramme de structure composite.**

[12] CCSL est un acronyme de Clock Constraint Specification Language

```
process Button
        port
                myEventPool : in ButtonEvents;
                itsControlsManager : out ControlsManagerEvents;
        param
                id : int
is
init to RELEASED
from RELEASED
        receive ButtonEvents:button_pressed from myEventPool;
        send ControlsManagerEvents:pressed(id) to itsControlsManager;
        to PRESSED
from PRESSED
        select
                wait[100,100];
                send ControlsManagerEvents:pressed(id) to itsControlsManager;
                to PRESSED
        []      receive ButtonEvents:problems_detected from myEventPool;
                to RELEASED
        []      receive ButtonEvents:button_released from myEventPool;
                to RELEASED
        end select
```

**Figure 9 L'encodage en processus ABCD de la diagramme d'états de la classe Button.**

# 6. Conclusion

Dans le but de permettre l'analyse, par des techniques formelles, de modèles exprimés dans des langages semi-formels (UML, AADL, SysML, …), nous avons introduit une chaîne de transformation prenant un sous-ensemble d'UML en entrée et ciblant le langage Fiacre en sortie. Pour ce faire, nous avons introduit deux langages intermédiaires : tUML en tant que pivot d'entrée et ABCD pour faciliter le franchissement du fossé sémantique. Cette combinaison nous permet de mieux maîtriser la transformation et d'apporter de la modularité à l'approche.

Le cas d'étude d'un régulateur de vitesse [8] décrit en UML a ainsi pu être transformé vers Fiacre et analysé [7] par l'outil OBP [6]. En termes de perspectives, la chaîne de transformation est toujours en développement pour, d'un coté lever certaines des restrictions posées sur le langage d'entrée et, de l'autre, permettre de cibler d'autres outils de vérifications. Plus particulièrement, dans le cadre du « *context-aware model checking* » mis en œuvre par l'outillage OBP, il nous faut étendre la transformation pour permettre de modéliser les contextes au même niveau que le système. Nous travaillons aussi sur les aspects pratiques qu'impliquent les différentes méthodologies tant en amont qu'en aval.

# 7. BIBLIOGRAPHIE


[1] C.André ; Syntax and semantics of the clock constraint specification language CCSL. *Technical Report 6925, INRIA* (2009)

[2] B.Berthomieu , J.P.Bodeveix , P.Farail , M.Filali , H.Garavel , P.Gaufillet , F.Lang , F.Vernadat ; Fiacre: an intermediate language for model verification in the TOPCASED environment. *Embedded Real Time Software and Systems (ERTS$^2$)*, 2008.

[3] M. L. Crane, J. Dingel ; UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling* (2007): 415-435.

[4] H. Fecher, J. Schönborn, M. Kyas, W.P. de Roever ; 29 New Unclarities in the Semantics of UML 2.0 State Machines. *Formal Methods and Software Engineering, Lecture Notes in Computer Science* (2005).
[5] J.-P. Queille, J. Sifakis ; Specification and verification of concurrent systems in CESAR. *Proceedings of the 5th Colloquium on International Symposium on Programming. London: Springer-Verlag*, 1982. 337-351.
[6] P. Dhaussy, F. Boniol et J.-C. Roger ; Reducing state explosion with context modeling for model-checking. *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, États-Unis, 2011.
[7] P. Dhaussy, L. Le Roux, C. Teodorov ; Vérification formelle de propriétés : Application de l'outil OBP au cas d'étude CCS; *Revue Génie Logiciel*, n°109, Juin 2014.
[8] L. Le Roux, J. Delatour et P. Dhaussy ; Modélisation UML d'un régulateur de vitesse automobile*. Revue Génie Logiciel*, n°109, Juin 2014.
[9] R. Chenouard et F. Jouault. Automatically discovering hidden transformation chaining constraints. *In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)* : 92-106.

**Frédéric Jouault** est chargé de recherche à l'ESEO, école d'ingénieur à Angers. Il a obtenu son doctorat de l'Université de Nantes, en 2006. Ses thématiques de recherche se situent dans le domaine de l'ingénierie dirigée par les modèles, avec un fort accent sur la transformation de modèles, la méta-modélisation, les langages dédiés, la définition de syntaxes concrètes, ainsi que leurs applications à la rétro-ingénierie et l'interopérabilité d'outils. Il a créé le langage de transformation de modèles ATL au cours de son doctorat et dirige le projet Eclipse.org sur la transformation de modèles.

**Ciprian Teodorov** est actuellement post-doctorant au laboratoire Lab-STICC UMR CNRS 6285 à l'ENSTA-Bretagne. Ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles pour l'industrialisation d'outils de preuve par model-checking pour les systèmes embarqués et systèmes sur puce. Avant intégrer l'équipe IDM du pôle STIC de l'ENSTA-Bretagne il a été ingénieur CAO à Dolphin Integration. En 2011 il a obtenu le titre de docteur en informatique de l'Université de Bretagne Occidentale pour ses travaux sur une approche dirigée par les modèles pour la synthèse physique de circuits émergents nanométriques.

**Jérôme Delatour** est enseignant-chercheur à l'ESEO (école d'ingénieur, http://www.eseo.fr/). Il est le responsable de l'équipe de recherche TRAME (TRAnsformations de Modèles pour l'Embarqué, http://trame.eseo.fr/). Les activités de recherche de Jérôme sont axées sur la modélisation, ainsi que sur le développement d'applications temps réels à l'aide de l'Ingénierie Dirigée par les Modèles. Dans ce contexte, il a notamment participé à la normalisation du profil UML MARTE (*http://omgmarte.org/*).

**Luka Leroux** est ingénieur de recherche. Après un Master recherche à l'université de Rennes I, il a rejoint en 2011 l'équipe du pôle STIC de l'ENSTA-Bretagne. Ses recherches sont orientées dans le domaine de modélisation des logiciels embarqués et les techniques de vérifications formelles d'exigences.

**Philippe Dhaussy** est directeur du pôle STIC de l'ENSTA-Bretagne et enseignant-chercheur HDR. Son expertise et ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles et les techniques de validation formelle pour le développement de logiciels embarqués et temps réel. Diplômé ingénieur (1978) de l'Institut Supérieur d'Électronique du Nord (ISEN), il a obtenu le titre de docteur (1994) à Telecom

Bretagne. Après un parcours industriel (1980-1991), il a rejoint l'ENSTA-Bretagne et a contribué à la création du groupe de recherche en modélisation de systèmes et logiciels embarqués, intégré au laboratoire Lab-STICC UMR CNRS 6285.

## Table of Contents


**Résumé** ..... **1**

**1. Introduction** ..... **1**

**2. Modélisation et vérification** ..... **2**

**2.1 UML** ..... **2**

**2.2 Vérification basée sur les contextes** ..... **2**

**2.3 Proposition** ..... **3**

**2.4 Architecture de la transformation UML vers ABCD** ..... **3**

**3. Sous-ensemble UML** ..... **4**

**4. tUML** ..... **6**

**5. ABCD** ..... **9**

**6. Conclusion** ..... **11**