MSR 2017, 11ème Colloque sur la Modélisation des Systèmes Réactifs
15-17 novembre 2017, Marseille, France

Un langage graphique pour la modélisation et l'analyse des systèmes réactifs

Hamzat Olanrewaju Aliyu[1], Oumar Maïga[2] and Mamadou Kaba Traoré[3]
[1] Federal University of Technology, Minna, Nigeria
[2] Université des Sciences, des Techniques et des Technologies, Bamako, Mali
[3] Université Clermont Auvergne, Clermont-Ferrand, France
hamzat.aliyu@futminna.edu.ng, maigabababa78@yahoo.fr,
mamadou.traore@uca.fr

Abstract

Une étude exhaustive d'un système complexe nécessite souvent l'utilisation de plusieurs méthodes d'analyse (comme la simulation, l'analyse formelle, ou l'émulation) pour produire des réponses complémentaires aux questions qui se posent. La combinaison de multiples méthodes d'analyse offre plus de possibilités et de rigueur pour analyser un système que ne peut le faire chacune des méthodes prise individuellement. Si cet exercice permet d'aller vers une connaissance complète des systèmes complexes, son adoption pratique ne va pas de pair avec les avancées théoriques en matière de formalismes. Ce déficit rend nécessaire la lourde tâche de créer et de gérer plusieurs modèles du même système dans différents formalismes et pour différents types d'analyse. Nous proposons, dans cette communication, de réduire cette tâche à la spécification d'un modèle unique duquel pourront être conduites des analyses multiples, grâce à un langage graphique de haut niveau permettant de fédérer la simulation, l'analyse formelle et l'émulation.

1 Introduction

Les méthodes d'analyse des systèmes, telles que la simulation, l'analyse formelle et l'émulation ont été intensivement utilisées ces dernières années pour étudier et prévoir les propriétés et les comportements des systèmes réactifs. Les résultats de ces analyses révèlent des connaissances qui peuvent améliorer la compréhension d'un système existant ou soutenir un processus de conception de manière à éviter de couteuses (voire catastrophiques) erreurs. Les réponses à certaines questions que l'on se pose sur un système sont généralement obtenues en utilisant des méthodes d'analyse spécifiques. Par exemple l'étude des performances d'un système peut se faire par simulation, alors que la vérification de propriétés telles que la vivacité, la sécurité et l'équité sont mieux étudiées en utilisant des méthodes formelles. De même, l'émulation permet de vérifier des hypothèses temporelles

impliquant des interactions humaines. Une étude exhaustive d'un système complexe nécessite donc l'utilisation conjointe de plusieurs méthodes d'analyse pour produire des réponses complémentaires. Si cette idée de combinaison de méthodes d'analyse est séduisante, son adoption pratique se heurte à la difficile tâche de créer et de gérer plusieurs modèles du même système dans différents formalismes et pour différents types d'analyse. Un autre facteur bloquant est que la plupart des environnements d'analyse sont dédiés à une méthode d'analyse spécifique (i.e., simulation, ou analyse formelle, ou émulation) et sont généralement difficiles à étendre pour réaliser d'autres types d'analyse. Ainsi, une vaste connaissance de formalismes supportant la multitude de méthodes d'analyse est requise, pour pouvoir créer les différents modèles nécessaires, mais surtout un problème de cohérence se pose lorsqu'il faudra mettre à jour séparément ces modèles lorsque certaines parties du système changent.

L'objectif du travail présenté dans cet article est d'alléger les charges d'un utilisateur de méthodes d'analyse multiples, dans sa quête d'exhaustivité pour l'étude des systèmes complexes, grâce à un cadre qui fédère la simulation, l'analyse formelle et l'émulation. Ceci est rendu possible par la définition d'un langage de spécification unifié de haut niveau appelé HiLLS (pour High Level Language for Systems specification), supporté par des capacités de synthèse automatiques d'artéfacts requis par les différentes méthodes d'analyse.

Les sections 2 et 3 de cet article sont dédiées à la présentation des aspects syntaxiques et sémantiques de ce formalisme. La section 4 montre son application dans un cas simple. La section 5 discute du travail présenté, en situant HiLLS par rapport à l'existant. La section 6 conclut l'article et trace quelques perspectives de développement futur.

2 Construction syntaxique de HiLLS

Tout langage de modélisation peut se définir par sa syntaxe abstraite, et sa fonction de correspondance de ce dernier vers un domaine sémantique (Harrel & Rumpe, 2004), (Kleppe, 2008). Comme décrit par la Figure 1, une syntaxe concrète peut être attachée au langage, pour rendre la modélisation plus aisée. En réalité, rien n'exclut d'associer plusieurs syntaxes concrètes différentes aux mêmes concepts définis par la syntaxe abstraite. De manière similaire, il est possible d'associer plusieurs correspondances sémantiques différentes au même langage, dès lors que ces correspondances ne créent pas entre elles, des problèmes de cohérence. Par exemple, si les domaines sémantiques utilisés ont déjà des relations (comme, quand la sémantique d'un des domaines est déjà donnée dans un autre des domaines), les nouvelles correspondances établies ne doivent pas rentrer en conflit avec les anciennes. On parle d'alignement sémantique lorsque cette cohérence est garantie.

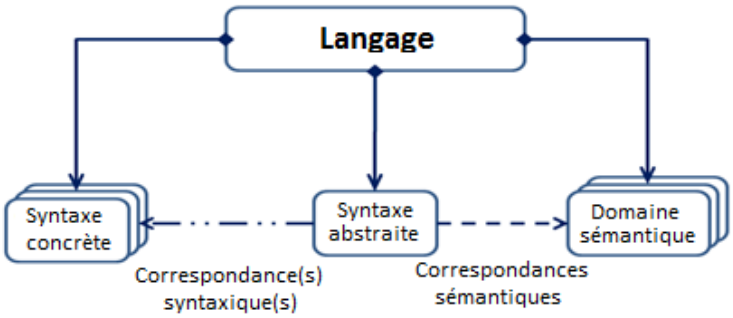

Figure 1 : Eléments constitutifs d'un langage de modélisation

Dans le cas particulier du langage HiLLS que nous proposons, l'originalité vient du fait que cet alignement sémantique concerne des domaines sémantiques n'appartenant pas au même univers

d'analyse (i.e., simulation, analyse formelle et émulation, respectivement). La syntaxe abstraite de HiLLS est construite par fusion de concepts issus des formalismes DEVS, Object-Z et UML, qui sont ensuite utilisés comme domaines sémantiques, respectivement pour la simulation, l'analyse formelle, et l'émulation. Nous donnons ci-après, en préalable au processus de construction syntaxique de HiLLS, un aperçu de chacun de ces formalismes et de leurs méta-modèles.

2.1 DEVS

DEVS (pour Discrete EVent systems Specification) est un formalisme enraciné dans la théorie des systèmes qui a été proposé au milieu des années 70 (Zeigler, 1976). Par système dynamique, la théorie des systèmes entend une entité dotée d'une structure et d'un comportement. La structure exprime, d'une part l'état dans lequel l'entité se trouve et les lois internes qui le gouvernent, et d'autre part, l'interface par laquelle elle interagit avec son environnement par réception de stimuli et émission de réponses. Le comportement se traduit par les traces d'évolution de cette interface dans le temps.

DEVS permet de représenter un système de deux façons. La première consiste à concevoir un modèle dit atomique, qui décrit le système comme un automate à états, i.e. un ensemble d'états S (fini ou infini), dont l'interface avec son environnement se définit par un ensemble d'entrées (X) et un ensemble de sortie (Y). L'état du système évolue le long d'une base de temps ($T \subseteq R+$), via des changements d'état internes, un processus de gestion du temps, une réactivité à l'environnement, et un mécanisme de génération d'évènements (i.e. de sorties). A tout instant, le système est dans un état donné, auquel on associe un état total (s,e), s étant l'état en question et e le temps écoulé dans cet état, dont il ne sort qu'après y avoir passé un temps qui correspond, soit à sa durée de vie initialement prévue (définie par une fonction ta, dite d'avancement du temps), soit à l'avènement d'un signal provenant de l'environnement du système (i.e., la réception d'une entrée). Le système change alors d'état en effectuant une transition externe (dans le cas d'une entrée reçue) ou interne (dans le cas où la durée de vie définie par la fonction ta sur l'état courant est écoulée). Ces deux types de transitions sont définis par deux fonctions, respectivement appelées fonction de transition externe (δ_{ext}) et fonction de transition interne (δ_{int}). Les réactions du système à son environnement sont traduites par une fonction de sortie (λ) qui n'est activée qu'en cas de transition interne.

DEVS fournit d'autre part la possibilité de décrire un système de façon hiérarchique, en concevant un modèle couplé. Ce dernier agrège des composants, qui peuvent être, soit des modèles atomiques, soit eux-mêmes des modèles couplés. Ces composants sont reliés par leurs ports de sortie et d'entrée ; ainsi, les évènements générés par un modèle deviennent des stimuli externes pour d'autres. Cette structure hiérarchique permet de modéliser un système de façon analytique (top-down) ou synthétique (bottom-up). Dans le premier cas, on commence par représenter le système avec un modèle couplé, puis on décompose chaque composant jusqu'à définir la base de la hiérarchie, i.e. les modèles atomiques. Dans l'autre cas, on conçoit ou on réutilise des modèles atomiques que l'on couple jusqu'à obtenir un modèle représentant de façon satisfaisante le système.

Enfin, la sémantique opérationnelle des modèles DEVS est définie par un protocole consistant en une hiérarchie d'automates abstraits (appelés Coordinateurs et Simulateurs), sous le contrôle d'un automate gestionnaire du temps simulé (appelé Coordinateur Racine). Les Simulateurs ont en charge de générer le comportement des modèles atomiques et les Coordinateurs celui des modèles couplés. La simulation s'effectue grâce à l'envoi de messages (de différents types, et tous estampillés par leur date de réalisation) entre les différents automates. Les algorithmes correspondant aux 3 types d'automate (Simulateur, Coordinateur et Racine) sont définis de manière abstraite (Chow 1996) afin de laisser le développeur libre de sa stratégie d'implémentation.

DEVS n'ayant pas de syntaxe concrète, plusieurs efforts ont été fournis dans la communauté scientifique pour proposer un langage lui servant d'interface de modélisation (Bergero & Kofman, 2011), (Bonaventura et al., 2013), (Capocchi et al., 2011), (Hamri & Zacharewicz, 2007), (Kim et al., 2009), (Sarjoughian & Zeigler, 1998). Nous proposons une approche de spécification centrée sur la

description des structures de données nécessaires pour capturer les concepts abstraits d'état, de transition d'état, de génération d'événements et de gestion de temps virtuel, que DEVS a définis. Un aspect particulièrement important est le passage que nous faisons de la notion d'espace d'états pouvant être fini ou infini, à la notion d'espace de variables conduisant toujours à une partition finie de l'espace d'états. Sans ce passage, la modélisation en termes d'automate à états serait impraticable de manière opérationnelle (en particulier, dans le cas d'un très grand nombre d'états et/ou de transitions). Le méta-modèle correspondant est présenté en Figure 2.

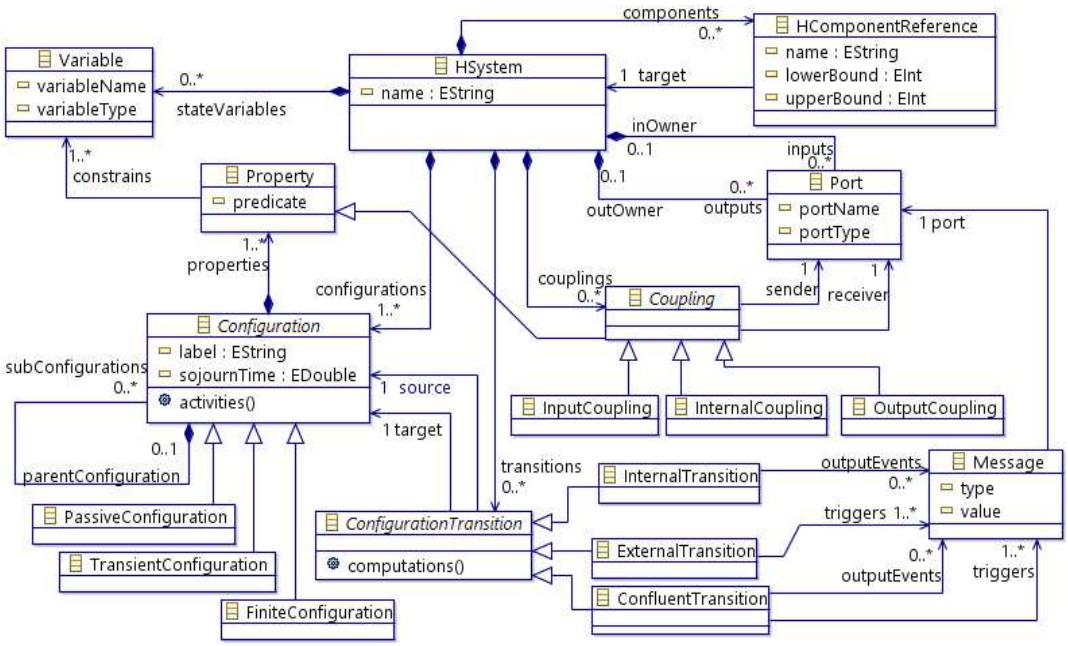

Figure 2 : Méta-modèle pour spécifier les modèles DEVS

La classe *HSystem* décrit un système, qui peut avoir zéro ou plusieurs variables d'état (*stateVariables*), des ports d'entrée (*inputs*), des ports de sortie (*outputs*), et des références à ses éventuels sous-composants (*components*), chacun étant un *HSystem*. Un système composite définit des couplages entre ses composants pour faciliter l'échange de messages entre eux (*Coupling*). Un couplage appartient à l'un des trois types: *InputCoupling*, *InternalCoupling* et *ExternalCoupling*. Le comportement d'un *HSystem* est décrit par les configurations et les transitions entre les configurations. Une *Configuration* est un regroupement d'états qui satisfont certaines propriétés communes définies sur les variables d'état. En d'autres termes, les configurations sont des sous-ensembles disjoints de l'espace d'état. Toute configuration est marquée par une étiquette unique et une durée de vie (*sojournTime*). Elle peut également avoir des activités, qui sont des opérations exécutées chaque fois que le système est dans la configuration, mais qui ne conduisent pas à une modification de la valeur d'une variable d'état, ni n'impliquent des opérations d'entrée et/ou de sortie (*activities*). Une configuration peut être classée en trois types en fonction de la valeur de son temps de séjour : *TransientConfiguration* (= 0), *PassiveConfiguration* (= +∞) et *FiniteConfiguration* (∈]0,+∞[). Enfin, un *HSystem* peut définir zéro ou plusieurs transitions de configuration (*ConfigurationTransition*) qui spécifient le comportement du système en termes d'évolution des configurations. *ConfigurationTransition* peut être l'un des trois types : *InternalTransition*, *ExternalTransition* et *ConfluentTransition*. Pour passer d'une configuration source à une configuration cible (de *source* à

target), les calculs associés à la transition correspondante doivent être exécutés (*computations*). Cette exécution entraîne la modification des valeurs de certaines variables d'état, ce qui permet aux nouvelles valeurs de satisfaire les contraintes de la configuration cible. Une *InternalTransition* se produit lorsque la durée de la configuration source expire ; elle peut donner lieu à des événements de sortie (*outputEvents*). Une *ExternalTransition* se produit lorsque les événements d'entrée sont reçus sur au moins l'un des ports d'entrée avant l'expiration du *sojournTime* de la configuration source. Une *ConfluentTransition* se produit lorsque les événements d'entrée sont reçus exactement à l'expiration du temps de séjour de la configuration source ; elle peut, elle aussi, donner lieu à des événements de sortie. Chaque événement (*Message*) est associé à une transition et fait référence à un port. Un message reçu fait référence au port d'entrée sur lequel il a été reçu tandis qu'un message de sortie généré dans le système fait référence à un port de sortie sur lequel il sera envoyé.

2.2 Object-Z

Object-Z (Smith, 2012] est une extension orientée objet du langage Z (Spivey, 1988). De ce fait, elle est, par construction, sémantiquement alignée avec UML sur les concepts d'objet, de classe, d'héritage et de polymorphisme. Ces concepts rendent Object-Z modulaire par rapport à Z. Cette modularité est concrétisée par la notion de classe de schéma (la notion initiale de schéma définie dans Z permet de définir en logique des prédicats, des variables et des propriétés logiques sur ces variables, y compris, celles permettant de décrire des opérations complexes). Une autre composante intéressante d'Object-Z, qui n'est pas présente dans Z, est son intégration avec la logique temporelle pour spécifier des invariants historiques d'un système.

Une spécification grammaticale de la syntaxe d'Object-Z est fournie dans (Smith, 2012). La présentation détaillée de tous les éléments de la grammaire dépasse le cadre de cette communication. Nous avons dérivé un méta-modèle simplifié des concepts les plus pertinents pour notre travail. Ce méta-modèle est partiellement présenté en Figure 3.

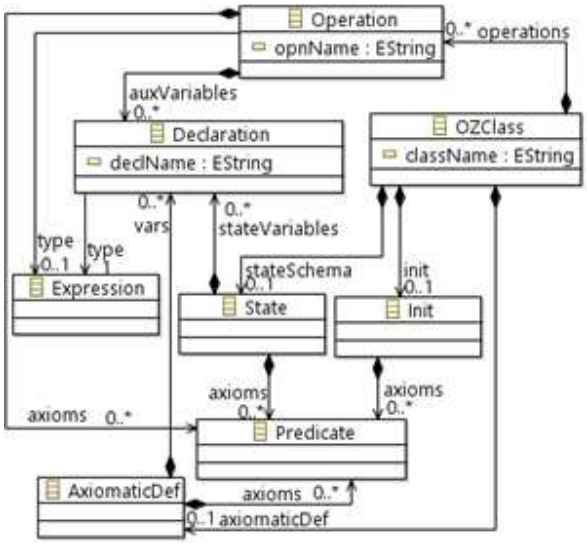

Figure 3 : Méta-modèle simplifié (et partiel) d'Object-Z

La classe *OZClass* dans le méta-modèle décrit la classe de schéma. Elle possède un identificateur (*className*), zéro ou un sous-schéma d'état (*stateSchema*) permettant de déclarer les variables d'état primaires et/ou secondaires (dont les types sont définis par des expressions), et zéro ou plusieurs sous-

schémas d'opération, chacun défini par des prédicats spécifiant des contraintes sur les variables déclarées. Un sous-schéma d'initialisation (*Init*) permet de spécifier les prédicats définissant l'état initial de la classe (i.e., l'état de toute instance de la classe, lors de sa création). Il est également possible d'associer à la classe de schéma, une définition axiomatique (*AxiomaticDef*), permettant de déclarer des constantes et des variables globales.

2.3 UML

UML (Rumbaugh et al., 2004), que l'on ne présente plus, est une notation orientée objet standard pour la conception de systèmes logiciels, indépendante de tout processus spécifique de développement et de toute technologie d'implémentation. Le méta-modèle proposé en Ecore dans (Budinsky et al., 2003) permet de capturer tous les concepts objets spécifiables en UML. Nous n'en retenons ici que les aspects utiles à notre travail, présentés en Figure 4. La classe *NamedElement* décrit toute entité ou relation identifiable par un nom unique (*name*). La classe *Classifier* unifie toutes les structures abstraites possibles, dont le concept de classe au sens de l'orienté objet (i.e., une entité autonome dotée d'attributs et d'opérations propres), la notion de type de donnée au sens classique (i.e., les types primitifs tels les entiers, réels, booléens, etc.), et celle de structures de données complexes (tels les tableaux, enregistrements, pointeurs/références, etc.). Ces concepts sont respectivement représentés par les classes *Class*, *DataType* et *TypedElement*. La classe *Package* décrit une organisation logique de plusieurs *classifiers*.

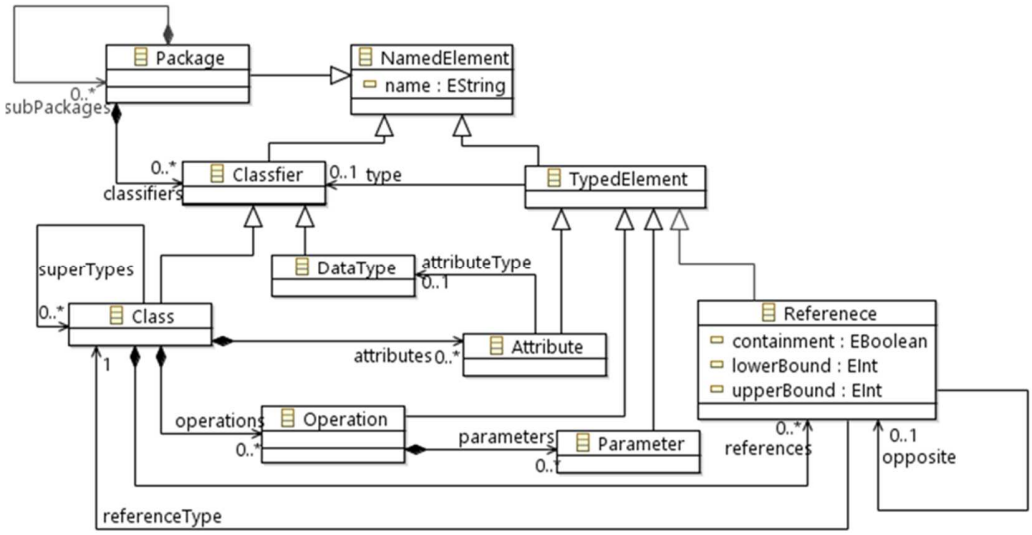

Figure 4 : Méta-modèle simplifié (et partiel) d'UML (Budinsky et al., 2003)

2.4 Syntaxe abstraite de HiLLS

Pour garantir un alignement sémantique par construction, le méta-modèle définissant la syntaxe abstraite de HiLLS est créé par tissage des méta-modèles de DEVS, d'Object-Z et d'UML. Il existe plusieurs techniques de tissage de méta-modèles (Emerson & Sztipanovits, 2006). Celle par héritage, que nous adoptons dans ce travail, suggère l'introduction de nouvelles classes et relations pour combiner deux méta-modèles décrivant des domaines distincts mais connexes.

Nous introduisons une classe d'interface, *HClassifier*, pour amorcer l'intégration des concepts basés sur DEVS et Object-Z comme le montre la Figure 5. Par conséquent, les classes *HSystem* et *OZClass* deviennent des types de *HClassifier* mais aucune relation n'est encore établie entre leurs

composants. *HClassifier* aligne les deux concepts sur une sémantique commune : celle de représenter un système dynamique. Cette introduction unifie donc le concept de système dans les deux univers, mais introduit de possibles redondances dans les concepts et relations associées à cette notion de système dans chacun des univers. C'est le cas de la classe *Variable*, redondante avec la classe *Declaration*, cette dernière apparaissant dans un mécanisme plus rigoureux (i.e., la relation *stateSchema*) pour spécifier l'espace d'état. La classe *Variable* est donc marquée pour suppression. Pour filtrer les concepts et relations redondants, nous appliquons une autre technique de tissage de méta-modèles appelée raffinement de classe. Elle consiste à utiliser un fragment d'un méta-modèle pour fournir une spécification détaillée d'un concept considérablement abstrait dans un autre méta-modèle. Par exemple, l'attribut *predicate* de la classe *Property* est en fait une contrainte sur les variables d'état ; la classe *Property* est donc raffinée par la classe *Predicate*, qui offre une large gamme de constructions pour spécifier les prédicats logiques. De manière analogue, la classe *Configuration* est raffinée par la classe *Expression*, à la fois à travers son attribut *sojournTime* (spécifié par une expression) et sa méthode *activities* (spécifiée par une séquence d'expressions). En effet, l'utilisateur spécifie par des expressions. De manière similaire, la classe *ConfigurationTransition* est elle aussi raffinée par la classe *Expression* (la méthode *computation* s'apparente à la méthode *activities*). Un autre raffinement est celui de la classe *Port* par la classe *Declaration*, à travers la spécification détaillée des attributs *name* et *type*.

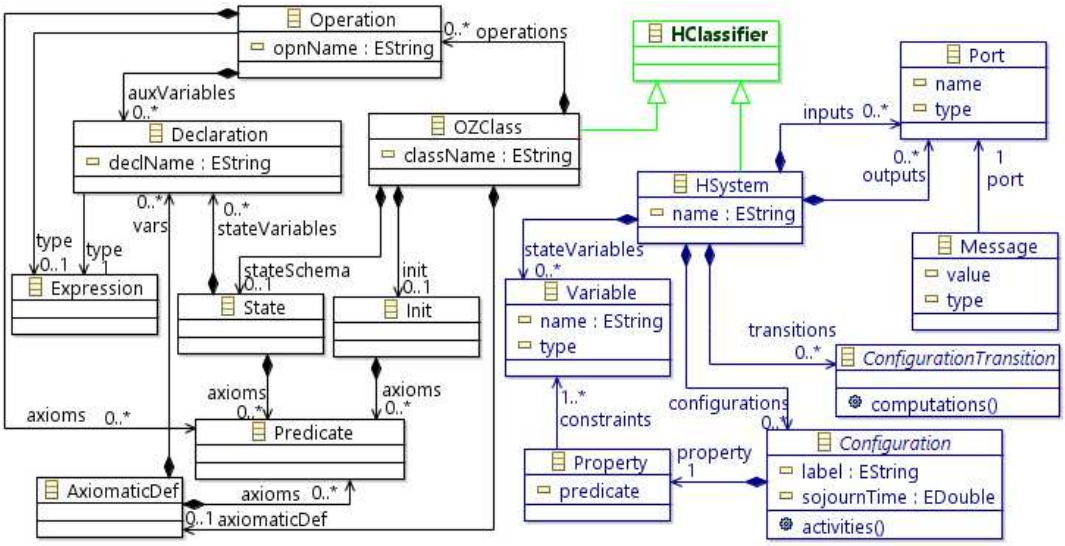

Figure 5 : Interfaçage des méta-modèles pour DEVS et Object-Z

La Figure 6 présente le méta-modèle résultant de l'application complète de la technique de raffinement de classe au méta-modèle de la Figure 5. Il convient de noter que la classe *OZClass* y a été renommée *HClass* pour plus d'uniformité dans le système de nommage, et que toutes les formes d'expression et d'opération n'y sont pas citées (dans les blocs « *enumeration* ») afin d'alléger la lecture du méta-modèle, par ailleurs suffisamment chargé. Par ailleurs, le méta-modèle complet de HiLLS intègre également un raffinement des aspects liés à la logique temporelle.

La classe *HClassifier* est aussi l'entrée pour le tissage de ce méta-modèle avec le méta-modèle d'UML. En effet, la mère de toutes les classes dans le méta-modèle d'UML est la classe *Classifier*. C'est cette classe (renommée *HClassifier*) que nous utilisons. Ce faisant, les concepts de composition, d'agrégation, d'héritage et d'interfaçage sont acquis dans HiLLS. Plusieurs contraintes OCL sont spécifiées pour identifier les règles de construction d'un modèle HiLLS, à partir de son méta-modèle.

Par exemple, la contrainte OCL suivante spécifie qu'aucune transition interne ou confluente ne peut partir d'une configuration passive :
self.oclIsTypeOf(InternalTransition) or self.oclIsTypeOf(ConfluentTransition) implies not self.source.oclIsTypeOf(PassiveConfiguration)

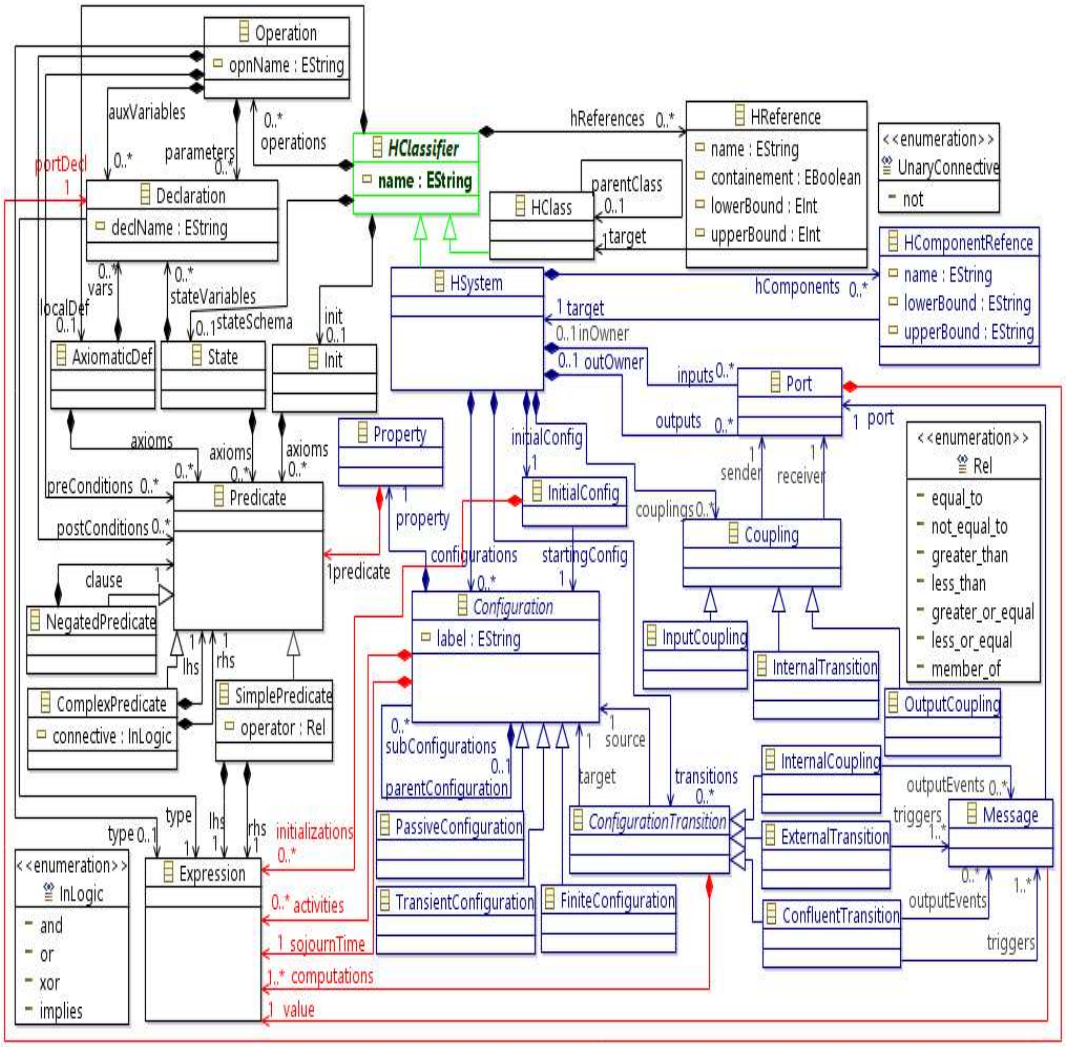

Figure 6 : Méta-modèle simplifié de HiLLS

2.5 Syntaxe concrète de HiLLS

Afin d'offrir une interface de modélisation conviviale, une syntaxe concrète, empruntant des représentations graphiques largement reconnues dans les communautés de modélisation, a été définie pour HiLLS. La Figure 7 en présente les éléments principaux sous une forme tabulaire.

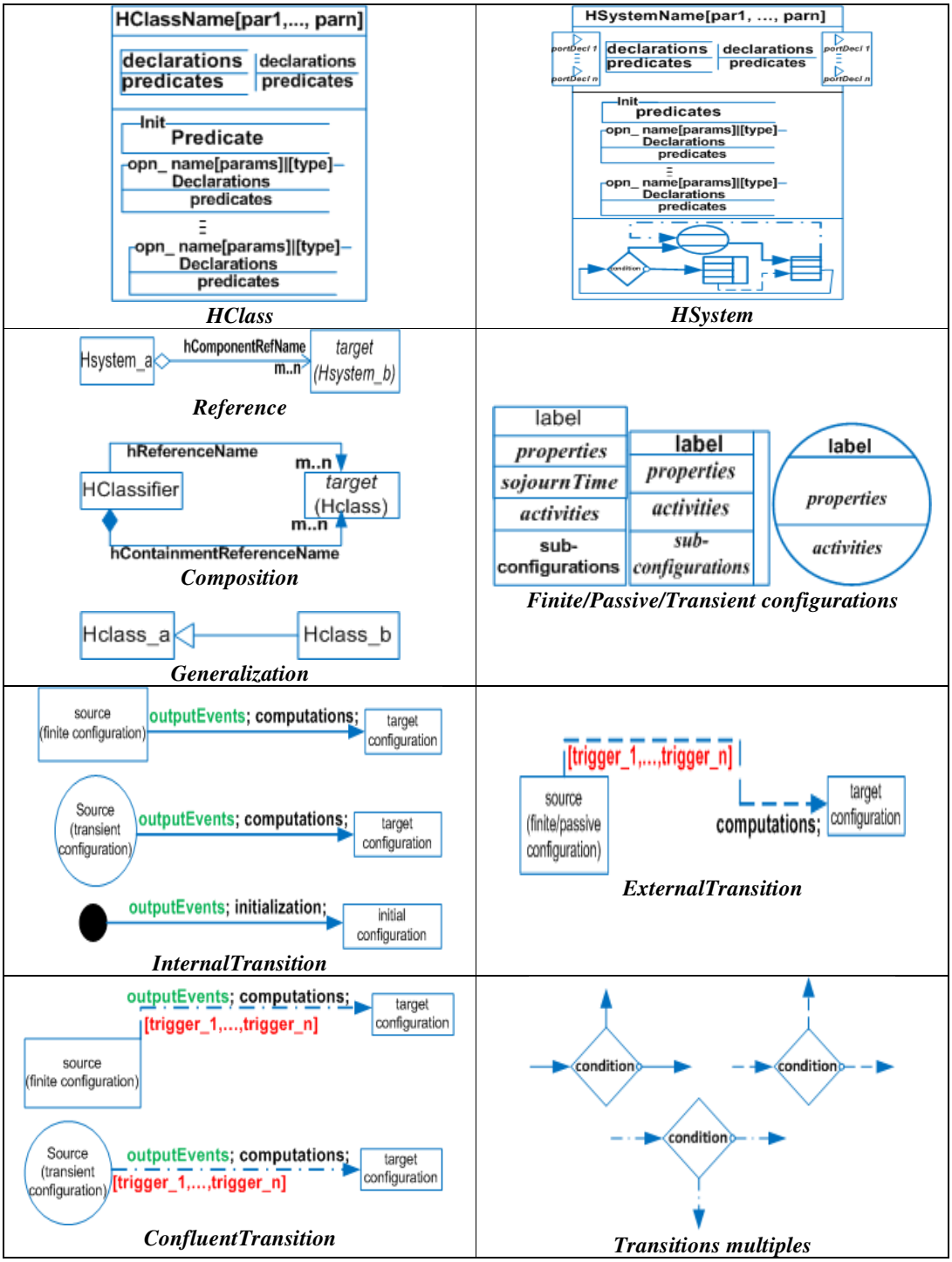

Figure 7 : Eléments de la syntaxe concrète de HiLLS

Le concept de *HClass* est représenté par une boîte à triple compartiments, à l'image du symbole utilisé pour les classes UML. Ces compartiments sont dédiés respectivement, au nom de la classe (avec possibilité d'y indiquer les éventuels paramètres), au schéma d'état, et aux schémas d'opération.

La notation pour le concept de *HSystem* étend celle pour *HClass*, en adjoignant un compartiment supplémentaire dans lequel le diagramme de transition de configuration du système apparait. De plus, les interfaces d'entrée et de sortie sont désignées par des compartiments latéraux (respectivement à gauche et à droite des compartiments principaux), où une flèche est utilisée pour représenter chaque port, dont la déclaration suit le format « portName: portType ». Les concepts sous-jacents de référence, de composition et de généralisation sont représentés de manière identique à ce qui est fait dans les diagrammes de classe d'UML.

Le concept de *FiniteConfiguration* est représenté par une boîte à cinq compartiments, dédiés respectivement à son label, le prédicat définissant ses propriétés, sa durée de vie, ses activités, et ses éventuelles sous-configurations (les sous-configurations sont à une configuration, ce que les sous-états sont à un état composite dans le diagramme d'état d'UML). De cette représentation, dérive celle du concept de *PassiveConfiguration* où le compartiment réservé à la durée de vie est supprimé et remplacé par une barre verticale sur le côté droit de la configuration. La représentation du concept de *TransientConfiguration* en dérive également, avec un remplacement de la boîte par un cercle, et la suppression du compartiment réservé à la durée de vie.

Le concept d'*InternalTransition* est représenté par une flèche solide partant du côté droit de la configuration source vers le côté gauche de la configuration cible. Les événements de sortie (*outputEvents*) et les calculs nécessaires durant la transition (*computations*) sont indiqués sur la flèche. La réception d'un outputEvent sur un port de sortie nommé outputPortName est spécifiée par l'expression *outputPortName!outputEvent*. Le respect des côtés pour le départ et l'arrivée d'une transition est strict pour les configurations présentant des côtés (ce qui n'est pas le cas de la transition transiente, pour laquelle n'importe quel point peut être utilisé pour accrocher la flèche de transition).

Le concept d'*ExternalTransition* est représenté par une flèche en pointillé partant du haut ou du bas de la configuration source vers le côté gauche de la configuration cible. Les conditions d'activation ou gardes (*trigger*) et les calculs nécessaires durant la transition (*computations*) sont indiqués sur la flèche. Une garde possible est la réception d'un inputEvent sur un port d'entrée, spécifiée pour un port nommé inputPortName, par l'expression *inputPortName?inputEvent*.

Le concept de *ConfluentTransition* est représenté par une flèche en points-et-traits partant de l'angle supérieur ou inférieur droit de la configuration source vers le côté gauche de la configuration cible. Les gardes, événements de sortie, et calculs nécessaires sont indiqués sur la flèche.

Il est possible de représenter plusieurs transitions partant d'une même configuration source, et ayant des caractéristiques communes, par une seule transition partant de cette source. Un nœud de décision est utilisé, pour définir les conditions permettant de déterminer le chemin à suivre vers l'une des configurations cibles concernées. Le nœud de décision est représenté par un losange ayant un flux d'entrée et deux flux de sortie. Le flux de sortie marqué d'un petit cercle indique le chemin pris lorsque la condition spécifiée dans le nœud de décision est vraie, l'autre sortie étant alors l'alternative.

3 Sémantiques de HiLLS

L'essence des trois sémantiques envisagées pour HiLLS est dans la philosophie de gestion du temps. Il s'agit de gérer respectivement un temps virtuel (simulation), un temps réel (émulation) et un temps abstrait (analyse formelle).

3.1 Sémantique opérationnelle pour simulation

La sémantique opérationnelle de HiLLS pour la simulation, est définie par DEVS. Ce dernier s'apparente à un automate temporel doté d'une unique horloge (appelée e, pour *elapsed time*), qui se réinitialise automatiquement entre chaque transition d'état, mais s'en distingue par le fait que l'espace d'état total est toujours infini, et que le nombre de transitions possibles peut également être infini. De manière formelle, un modèle DEVS atomique est défini par la structure suivante:

$M = <X, Y, S, \delta int, \delta ext, \delta conf, \lambda, ta>$, où

- X est l'ensemble des entrées, et Y est l'ensemble des sorties.
- S est l'ensemble des états possibles du système.
- ta est la fonction d'avancement du temps, qui donne la durée durant laquelle le système reste dans un état donné avant de faire une transition interne.
- δ_{int} est la fonction de transition interne.
- δ_{ext} est la fonction de transition externe.
- δ_{conf} est la fonction de conflit, i.e., la collision entre transitions externe et interne.
- λ est la fonction de sortie, invoquée durant chaque transition interne, qui détermine les évènements générés par le système, en fonction de son état courant.

Un modèle DEVS couplé est défini par la structure suivante :

$M = <X, Y, D, \{Md / d \in D\}, EIC, EOC, IC>$, où

- X et Y sont définis comme précédemment.
- D est l'ensemble des composants, chacun d'eux étant un modèle couplé ou atomique.
- EIC (External Input Coupling) définit le couplage entre les ports d'entrée du modèle couplé et certains ports d'entrée de certains composants.
- EOC (External Output Coupling) définit le couplage entre les ports de sortie des composants et les ports de sortie du modèle couplé.
- IC (Internal Coupling) définit le couplage entre composants (sorties vers entrées).

Le tableau 1 présente, de manière informelle, les règles de traduction de HiLLS vers DEVS. Nous ne présentons pas ici les détails d'implémentation de cette mise en correspondance, que nous avons réalisés avec ATLAS (Aliyu et al., 2016), (Aliyu & Traoré, 2015). Toutefois, un extrait est montré par la Figure 8. Les modèles DEVS sont certes générés dans notre environnement Java de simulation DEVS, appelé SimStudio (Traoré, 2008), mais il est possible, à condition de réécrire les règles, d'adapter cette transformation à tout autre environnement pour DEVS.

Tableau 1 : Correspondance entre HiLLS et DEVS

Modèle HiLLS (HSystem)	*Modèle DEVS (M)*
Produit cartésien des domaines, respectivement des ports d'entrée et de sortie	X, Y
Produit cartésien des domaines des variables d'état	S
Ensemble des relations, respectivement de type *InternalTransition*, *ExternalTransition* et *ConfluentTransition*	δ_{int}, δ_{ext}, δ_{conf}
Ensemble des relations définissant les événements de sortie	λ
Ensemble des relations définissant les *sojournTime* des configurations	ta
Ensemble des *HSystem* composant le modèle	D
Ensemble des prédicats définissant les couplages, respectivement dans *InputCoupling*, *OutputCoupling*, et *InternalCoupling*	EIC, EOC, IC

```
rule HSystem2AtomicDEVS {
 from--HSystem without components -> Atomic DEVS
  hsystem : HiLLS!HSystem(hsystem.hcomponents->isEmpty())
 to
  atomicDEVS : DEVS!AtomicDEVS (
    name <- hsystem.name,
    iports <- hsystem.inputs->collect(p|thisModule.HiLLSPort2DEVSInput(p)),
    oports <- hsystem.outputs->collect(q|thisModule.HiLLSPort2DEVSOutput(q)),
    stateVars <- hsystem.stateSchema.declarations->collect(v|thisModule.HiLLSVar2DEVSVar(v)),
    phases <- hsystem.configurations->collect(ph|thisModule.HiLLSConfig2DEVS_Phase(ph)),
    deltaInt <- hsystem.transitions->collect(dInt|thisModule.HiLLSTrans2DEVSdeltaInt(dInt)),
    deltaExt <-hsystem.transitions->collect(dExt|thisModule.HiLLSTrans2DEVSdeltaExt(dExt)),
    deltaConf <-hsystem.transitions->collect(dConf|thisModule.HiLLSTran2DEVSdeltaConf(dConf))
  )
}
lazy rule HiLLSPort2DEVSInput {
  from
    hillsPort : HiLLS!Port
  to  --HiLLS port -> DEVS input port
    devs_input : DEVS!IPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
  )
}
lazy rule HiLLSTrans2DEVSdeltaInt {
  from --InternalTransition-> DeltaInt
    intTrans: HiLLS!InternalTransition
  to
    deltaInt : DEVS!DeltaInt (
    sourcePhase <- intTrans.source,
    targetPhase <- intTrans.target,
    outputs <- intTrans.outputEvents
  )
}
lazy rule HiLLSPort2DEVSOutput {
  from
    hillsPort : HiLLS!Port
  to --HiLLS port -> DEVS output port
    devs_output : DEVS!OPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
 )
}

lazy rule HiLLSTrans2DEVSdeltaExt {
  from --ExternalTransition-> DeltaExt
    outTrans: HiLLS!ExternalTransition
  to
    deltaInt : DEVS!DeltaExt (
    sourcePhase <-  outTrans.source,
    targetPhase <-  outTrans.target,
    inputs <- outTrans.triggers
  )
}
```

Figure 8 : Extraits d'implémentation ATL de la transformation de HiLLS en DEVS

3.2 Sémantique opérationnelle pour émulation

Afin que la spécification HiLLS puisse être utilisée également pour synthétiser le code d'un système logiciel, nous proposons un schéma d'exécution de modèle HiLLS en mode temps réel, i.e., en utilisant l'horloge réelle de l'ordinateur, et non une horloge virtuelle fournissant un temps simulé. Ceci s'appuie sur un motif récurrent de conception qui est une adaptation du design pattern Observer (Gamma et al., 1995) autorisant des interactions asynchrones. Cette sémantique est décrite par le diagramme de classe UML donné en Figure 9.

Les composants logiciels majeurs sont les classes *Subject*, *Observer*, *Notifier*, *Notification* et *ConcreteNotification*. Les classes *ConcreteAtomicSystem* et *ConcreteCoupledSystem* correspondent aux systèmes décrits en HiLLS, qui héritent du cadre générique ainsi défini. En raison de leurs relations d'héritage avec *AbstractSystem*, les classes *AbstractCoupledSystem* et *AbstractAtomicSystem* (qui correspondent respectivement à un modèle HiLLS composite ou non) implémentent l'interface *Observer*. Par conséquent, les deux peuvent être influencés par les notifications des objets qu'ils observent. Le port générique décrit les ports d'entrée et de sortie d'un système. Le paramètre générique T modélise le type d'événements admissible dans le port et doit être fourni lors de l'instanciation. En raison de ses relations avec *Subject* et *Observer*, un port peut être un observateur alors qu'il est lui-même une entité observable. La relation sujet-observateur est utilisée pour implémenter une connexion permettant à deux ports d'échanger des messages/événements, de sorte

que chaque fois qu'un message est placé sur un port source, il est immédiatement transmis au port cible. Ceci est réalisé ici par le fait que le port cible est enregistré dans la liste des observateurs du port source. A chaque changement de valeur de ce dernier, une notification est faite à tous les ports cibles concernés par un couplage avec lui.

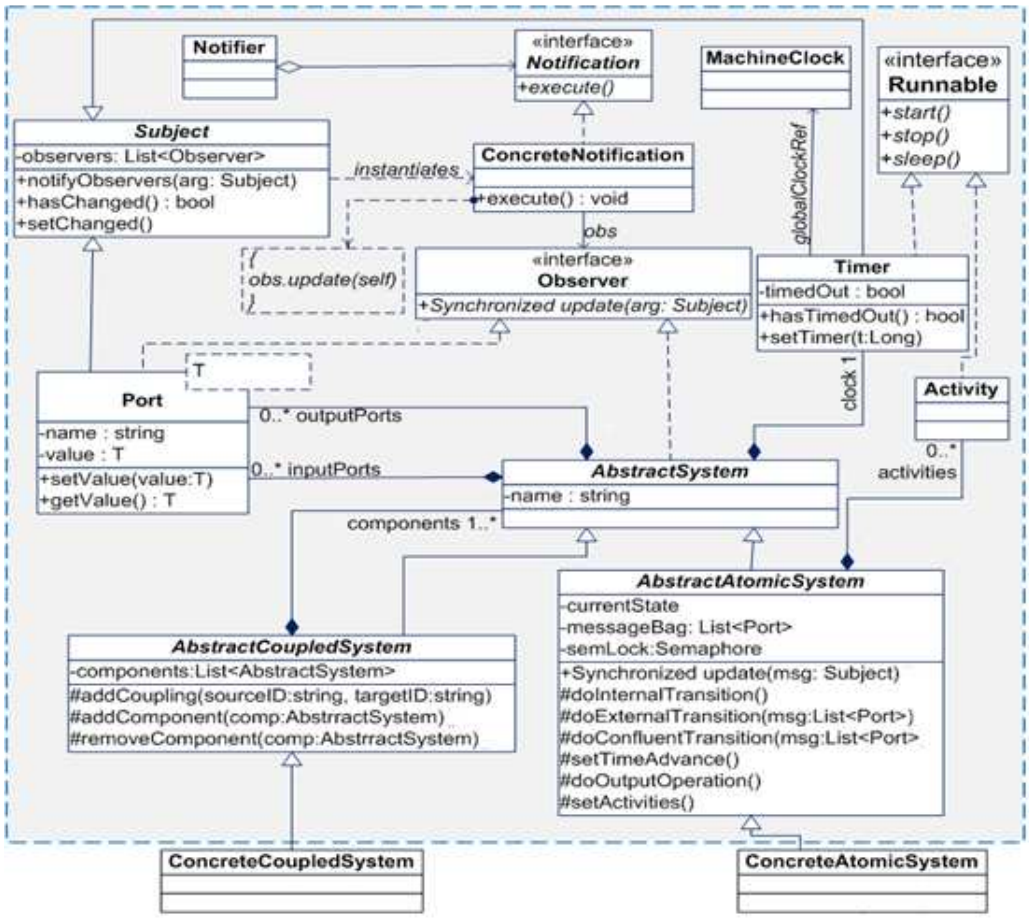

Figure 9 : Schéma d'émulation de HiLLS

Les couplages de port sont spécifiés en fournissant les noms des ports source et cible à la méthode *addCoupling*. Un port peut évidemment être impliqué dans plusieurs couplages à la fois, et peut jouer le rôle de source dans certaines relations tout en jouant le rôle de cible dans d'autres. Un *AbstractAtomic* est enregistré dans la liste des observateurs de tous ses ports d'entrée. Ainsi, la réception d'un événement d'entrée sur un port d'entrée déclenche naturellement une transition externe dans *l'AbstractAtomic* (opération *doExternalTransition*). Un port d'entrée avertit donc automatiquement son système hôte chaque fois qu'il reçoit un événement. La méthode *upDate* de l'*Observer* implémente le protocole d'échange de messages. En effet, une transition est invoquée dans *AtomicSystem* chaque fois que cette méthode est sollicitée, i.e., quand l'entité reçoit des notifications des sujets qu'elle observe (i.e., de son horloge et de ses ports d'entrée). Étant donné que les notifications sont indépendantes, il est possible de recevoir plusieurs messages simultanément. Tous les messages instantanément reçus sont donc d'abord groupés en un sac unique, dont le traitement est

différencié dans la méthode *update* selon la nature de ces messages. Par ailleurs, un système dispose d'une horloge, qui se sert à son tour de l'horloge de la machine sur laquelle elle s'exécute pour gérer les avancées temporelles de ses états et l'exécution de ses activités. Le temporisateur est une entité observable en vertu de son héritage de la classe *Subject*, et son seul observateur est son système hôte. Dans chaque nouvel état, le système initialise son horloge pour surveiller l'avancement du temps de son état courant. Une fois que la durée de vie de l'état courant est écoulée, l'horloge place son attribut *timedOut* sur true et envoie automatiquement une notification au système, déclenchant ainsi une transition interne (opération *doInternalTransition*). Si la notification de l'horloge coïncide avec celle d'un port d'entrée, une transition confluente est déclenchée (opération *doConfluentTransition*). La méthode *setActivities* peut être utilisée pour définir et associer des activités à chaque état. Ces dernières sont eécutées, chaque fois que le système se trouve dans cet état.

Toutes les méthodes dans les classes *AtomicSystem* et *CoupledSystem* sont abstraites. Par conséquent, les classes concrètes décrites en HiLLS les mettent en œuvre en fournissant les éléments spécifiques. Nous avons réalisé une implémentation Java de ce cadre générique, au sein de notre package SimStudio. Le lecteur intéressé par plus de détails, peut se référer à (Aliyu, 2016).

3.3 Sémantique logique

Au niveau logique, une spécification HiLLS correspond à un automate temporel (Alur & Dill, 1994) ou TA (Timed Automata), ayant les caractéristiques suivantes : (1) chaque configuration est un nœud de l'automate, et l'horloge unique e est explicitement ramenée à 0 sur chaque transition ; (2) les activités sont abstraites et n'ont pas lieu d'apparaître ; (3) la durée de vie d'un état n'est autre qu'une variable d'état parmi les autres ; (4) les prédicats sur ces variables représentent les invariants d'état de l'automate ; et (5) lors des transitions, les prédicats sur les variables sont des gardes lorsqu'il s'agit de conditions pre, et des assignations de valeur lorsqu'il s'agit de conditions post. Le modèle TA ainsi généré peut être soumis à des outils comme UPPAL (Behrman et al., 2004) pour analyse formelle.

Par ailleurs, les déclarations et instructions étant toutes réalisées sous forme de schémas, le système tout entier se résume à une classe de schéma composée de schémas d'état (les attributs), de schémas d'opération, et d'une série de schémas représentant les transitions de configuration (où chaque configuration source est une condition pre, et chaque configuration cible est une condition post). Le modèle Object-Z ainsi généré peut être formellement analysé par les outils de la suite CZT (Malik & Utting, 2005).

4 Exemple illustratif

Notre exemple commence par un simple feu tricolore de signalisation routière, susceptible de subir des pannes et des opérations de remise en service. En mode nominal, le feu affiche de manière cyclique les lumières verte, orange et rouge, et à des intervalles de temps constants par couleur de lumière (supposons, de manière fictive, que les longueurs respectives de ces intervalles sont 120 secondes, 40 secondes, et 60 secondes). La figure 10 correspond au diagramme de transition de configurations (que nous appelons dynamique de comportement) du feu tricolore. Six configurations sont considérées : AfV (affichage de la lumière verte), AfN (affichage de la lumière orange), AfR (affichage de la lumière rouge), AfN (extinction de toutes les lumières), DRE (réaction à une mise hors service), et DRA (réaction à une mise en route). La variable s codifie chaque configuration (afin de pouvoir les manipuler symboliquement dans les calculs). La variable σ mémorise, dans chaque configuration, la durée de vie restante (différente, dans certains cas, de la durée de vie initiale de la configuration). Les paramètres pv, po et pr sont les durées normales respectives d'affichage des lumières. L'activité nulle (notée $\varnothing$) correspond à une situation de pure attente dans une configuration donnée (dans la configuration DRE, une activité d'alerte est déclenchée). Le prédicat de la

configuration AfV est (s=0)∧(σ∈ [0, pv]). On peut constater qu'il s'agit bien d'un ensemble d'états, chacun pouvant être défini par une valeur spécifique allouée à σ avec la même valeur de s. Les prédicats des autres configurations sont définis de manière similaire. On notera que DRA est, à l'instar de DRE, une configuration transiente (c'est à dessein que nous montrons les deux possibilités équivalentes de représentation graphique). Lors de ses transitions internes, le feu tricolore envoie sur son port de sortie *out*, les valeurs vert, orange, rouge, ou noir, selon les circonstances, et assigne à σ la bonne valeur pour la prochaine configuration (po, pr, pv, 0 ou +∞). Par contre, lorsque le feu reçoit une valeur *on* sur son port d'entrée *in*, alors qu'il est déjà en marche (e.g., dans la configuration AfV), il réagit en revenant dans la même configuration pour signifier qu'il reste en phase d'affichage de la lumière verte, mais en soustrayant de sa durée de vie restante, la valeur de l'horloge implicite (i.e., le temps écoulé dans AfV avant réception de cette entrée). La même chose se passe dans les autres phases AfO et AfR, mais nous ne les indiquons pas sur la Figure 9 pour éviter de la surcharger.

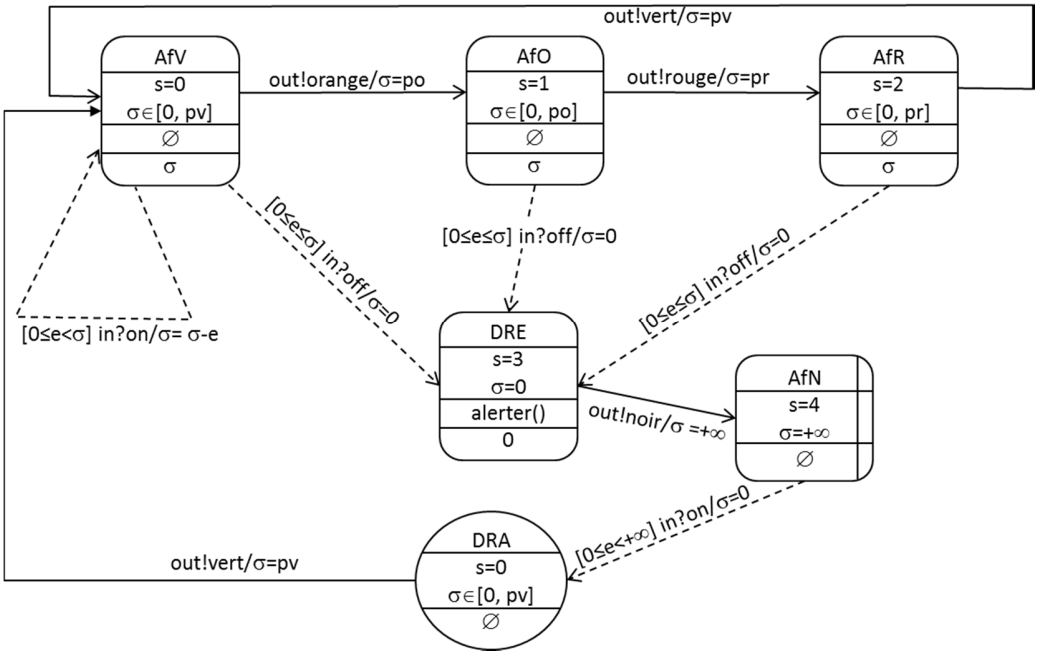

Figure 10 : Exemple de dynamique de comportement en HiLLS

La Figure 11 donne l'architecture complète du système de feu tricolore, qui encapsule la dynamique de comportement dans un diagramme de classe. Les variables d'état sont des attributs de la classe, les paramètres en sont les arguments d'instanciation (au sens de l'orienté objet), et les variables d'entrée (respectivement de sortie) en sont les ports d'entrée (respectivement de sortie). Les activités sont des méthodes de la classe. Des relations de composition ou d'agrégation sont possibles entre le système et certains composants utilisés comme simples ressources (e.g., une sonnerie est sollicitée pour réaliser les opérations d'alerte). De même, on s'autorise l'établissement de relation d'héritage entre systèmes (par exemple, la possibilité de dériver des types de feu différents selon les zones pays), ainsi que la possibilité d'implémenter des interfaces (par exemple, une classe abstraite Lumière dont les fonctionnalités afficher et clignoter sont à implémenter).

Etendons maintenant l'exemple à un système plus global, dans lequel un contrôleur observe en permanence la sortie du feu tricolore ainsi décrit, et lui envoie des signaux pour le contrôler. Comme le montre la Figure 11, le diagramme de transition de ce système global ne comporte qu'une unique

configuration passive, dont le prédicat indique que tout signal émis par le composant contrôleur (attribut c) sur son port de sortie *sig* est reçu sur le port d'entrée *in* du composant feu (attribut f), et que toute sortie du feu est observé par le contrôleur. Le système global est fermé, et donc ne possède ni entrée, ni sortie.

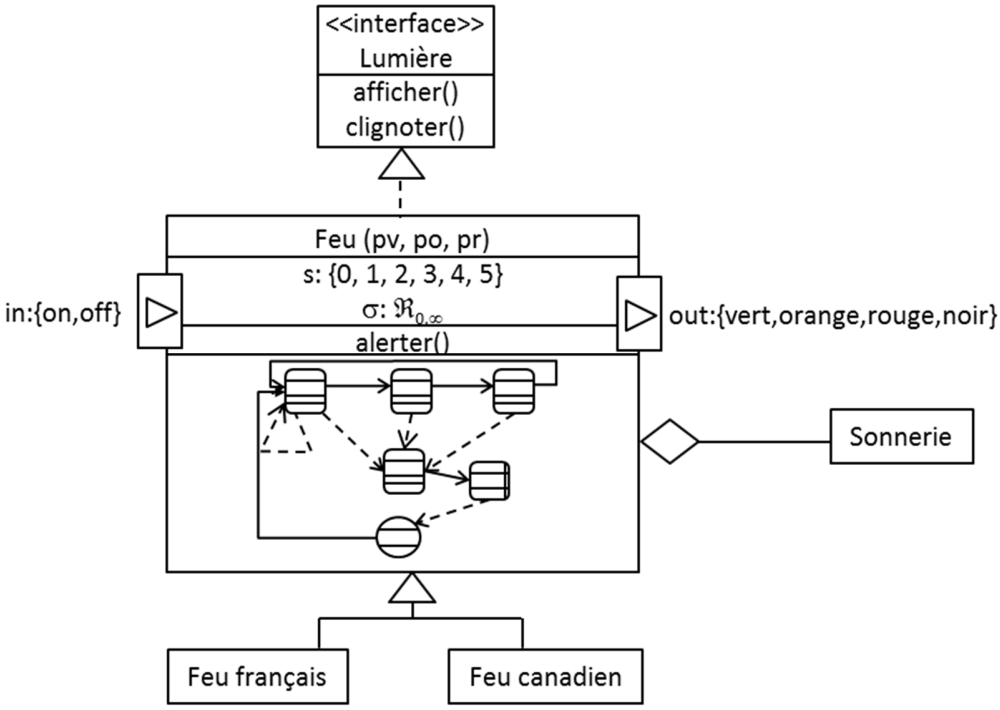

Figure 11 : Exemple de structure en HiLLS

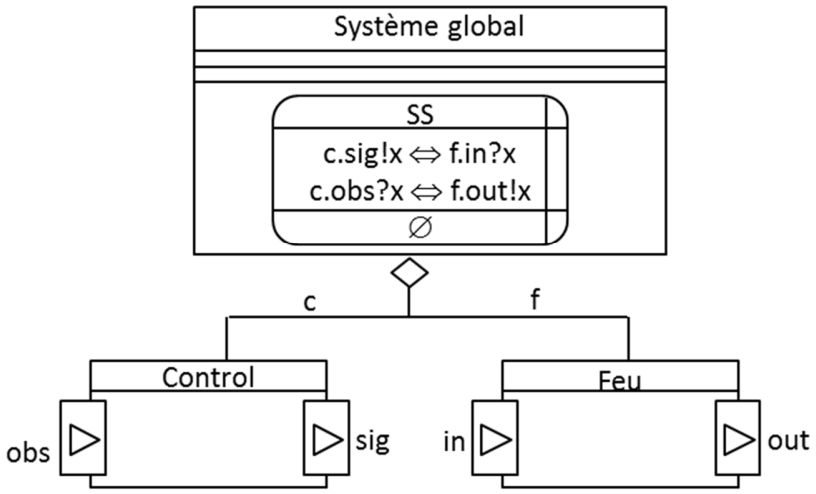

Figure 11 : Exemple de dynamique de structure en HiLLS (structure statique)

Dans le cas plus complexe d'un système dont la structure peut changer dans le temps, le diagramme de transition comporterait plusieurs configurations, chacune décrivant à travers son prédicat, la nouvelle structure du système. Les transitions exprimeraient le comportement dynamique de cette structure. Nous parlons alors de dynamique de structure. Un exemple serait le cas où le contrôleur et le feu de la Figure 11 ne sont en interaction que 16 heures sur 24, par jour. L'unique configuration SS serait alors remplacée par un diagramme de transitions internes cycliques entre deux configurations, l'une correspondant à SS, mais dont la durée de vie est 16 (et non $+\infty$), et l'autre dont le prédicat est nul (signifiant ainsi l'absence de couplage entre le contrôleur et le feu).

5 Discussion

Si la modélisation multi-formalismes est aujourd'hui largement pratiquée, la modélisation multi-analyses est plutôt une approche nouvelle. Clarifions ces propos, en faisant la différence entre l'analyse multiple et la modélisation multi-analyse.

Dans le premier cas, il s'agit de conduire divers types d'analyse sur un système (i.e., simulation, émulation, et méthode formelle), afin de couvrir le plus grand spectre de connaissances dérivables du système. Deux types d'approche existent en la matière :

- Construire plusieurs modèles d'un même système, chacun dédié à une forme spécifique d'analyse (e.g., trois modèles, dont un de simulation, un pour l'analyse formelle, et un correspondant à un prototype logiciel). Il existe une pléthore de formalismes spécifiques à chacun des types d'analyse considérés.
- Elaborer une méthode de transformation des modèles écrits dans un formalisme dédié à u type d'analyse donné, en modèles adaptés à un autre type d'analyse. Par exemple, plusieurs travaux proposent des approches de transformation de modèle de simulation en modèle pour analyse formelle. L'Ingénierie Dirigée par les Modèles (IDM) offre un cadre méthodologique générique pour réaliser ces transformations (Bézivin, 2006).

Dans le second cas, il s'agit de faire en sorte que les divers types d'analyse envisagés, puissent se faire sur un modèle unique (et non à partir de plusieurs modèles) du système étudié. Il s'agit donc d'une forme particulière d'analyse multiple, qui ne mobilise qu'une seule spécification. Les langages mobilisés sont des formalismes « pivot », pouvant être connectés à différents outils d'analyse. Deux types d'approche existent également dans ce cas :

- Intégrer, dans un formalisme dédié initialement à un type d'analyse, des capacités pour mener un autre type d'analyse. Par exemple, des formalismes, comme les Réseaux de Pétri, CADP, ou les langages synchrones, adaptés à l'analyse formelle de propriété, ont également été étendus à une sémantique opérationnelle permettant de les simuler (Zhou & Dicesare, 1993), (Garavel & Hermanns, 2002), (Berry & Gonthier, 1992).
- Intégrer par construction plusieurs formes d'analyse dans un formalisme érigé à dessein. C'est l'approche que nous avons adoptée. En plus de se différencier de la précédente, par le fait que le formalisme résultant ne part pas d'un langage initialement dédié à un type d'analyse, mais est conçu dès le départ pour intégrer plusieurs formes d'analyse, notre démarche porte sur une intégration triple (simulation, émulation et analyse formelle). La littérature montre plutôt des intégrations par pair de méthodes : simulation et analyse formelle (Kuhn et al., 2003), (Trojet et al. 2009), (Yacoub et al., 2014), (Traoré 2006), émulation et analyse formelle (Lano et al., 2004), (Lilius & Paltor, 1999), (Laleau et al., 2010), (Mentré 2016), (Bousse et al., 2012), (Kinoshita et al., 2014), ou simulation et émulation (Risco-Martın et al., 2007), (Schamai et al., 2009), (Shaikh & Vangheluwe, 2011), (Sqali, 2012). Nous n'avons pas connaissance de travaux dans la littérature, qui fusionnent les trois types d'analyse.

6 Conclusion

Cet article se situe dans le cadre d'un travail d'intégration de méthodes multiples d'analyse pour l'ingénierie des systèmes dirigée par modèle. Les trois méthodes d'analyse considérées sont la simulation, l'analyse formelle et l'émulation. L'objectif est d'exploiter la synergie de ces méthodes pour permettre des analyses complémentaires des propriétés statiques et dynamiques des systèmes complexes. Une propriété indésirable découverte dans l'analyse d'un système signale un problème de sûreté dans la conception du système. Une telle découverte doit être faite à un stade précoce de développement pour prévenir les erreurs coûteuses dans le système final. De même, une propriété souhaitée découverte en amont de la conception, atteste de la validité des choix de conception. Selon les propriétés à découvrir, c'est la simulation, l'analyse formelle ou l'émulation qu'il est adéquat de mettre en œuvre. Peu d'experts maîtrisent à la fois les langages et techniques spécifiques à ces différentes méthodes. Par ailleurs, les modèles conçus en ayant exclusivement à l'esprit une de ces méthodes s'avèrent en général peu réutilisables en l'état, pour la mise en œuvre des autres méthodes. En général, un effort de transformation de modèle est requis, voire une nouvelle description du système sous étude dans des formes nouvelles mieux adaptées. Cette tâche peut être très ardue et sujette à erreurs.

Dans notre tentative de proposer une solution efficace à cette difficulté, nous avons envisagé un langage qui, par construction, réalise l'alignement sémantique entre trois formalismes respectivement représentatifs des pratiques de simulation, d'analyse formelle et d'émulation. De cette sorte, un seul modèle exprimé dans ce langage permettrait d'obtenir par correspondance sémantique les artefacts adaptés aux méthodes d'analyse ainsi intégrées. Un autre apport important est que ce langage possède une syntaxe concrète graphique, permettant de communiquer plus facilement les modèles construits entre les parties prenantes.

Il reste encore un long chemin à parcourir pour mettre ce langage à la portée des utilisateurs potentiels. Son utilisabilité reste à être évaluée sur des systèmes complexes à grande échelle. Un environnement logiciel complet, offrant une couche de modélisation à travers une interface graphique conviviale, est un préalable à une large adoption. A cela doit s'ajouter tous les utilitaires d'une suite logicielle, des moyens de compilation à ceux de génération de code dans divers langages de programmation. Pour cela, nous comptons profiter de l'infrastructure IDM de la plate-forme Eclipse.

Remerciements

Ce travail a été effectué dans le cadre de deux thèses de Doctorat, de manière complémentaire l'une à l'autre. Les auteurs adressent leurs vifs remerciements à NITDA (National Information Technologies Development Agency, Nigeria) pour leur soutien financier à la thèse de Hamzat O. Aliyu, et aux Rectorats des Universités de Bamako, pour leur soutien financier à la thèse de Oumar Maïga.

References

(Aliyu & Traoré, 2015) Aliyu, H.O., and Traoré, M.K. 2015. Toward an Integrated Framework for the Simulation, Formal Analysis and Enactment of Discrete Event Systems Models. In Proceedings of the Winter Simulation Conference, Huntington Beach, CA, USA. IEEE Press (pp. 3090-3091).

(Aliyu et al., 2016) Aliyu, H.O., Maïga, O., and Traoré, M.K. 2016. The High Level Language for System Specification: A Model-Driven Approach to Systems Engineering. International Journal of Modeling, Simulation, and Scientific Computing, 7(1): 1641003.

(Aliyu, 2016) Aliyu, H.O. (2016). An Integrative Framework for Model-Driven Systems Engineering: Towards the Co-Evolution of Simulation, Formal Analysis and Enactment Methodologies for Discrete Event Systems. PhD thesis, Clermont Ferrand 2, France.

(Alur & Dill, 1994) Alur, R., and Dill, D.L. 1994 A Theory of Timed Automata. In Theoretical Computer Science, 126: 183-235

(Behrman et al., 2004) Behrmann, G., David, A., and Larsen, K. G. 2004. A Tutorial on Uppaal. In Formal methods for the design of real-time systems (pp. 200-236). Springer Berlin Heidelberg.

(Bergero & Kofman, 2011) Bergero, F., and Kofman, E. 2011. PowerDEVS: a Tool for Hybrid System Modeling and Real-Time Simulation. Simulation, 87(1-2): 113-132.

(Berry & Gonthier, 1992) Berry, G., and Gonthier, G. 1992. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. Science of Computer Programming, 19(2).

(Bézivin, 2006) Bézivin, J. 2006. Model Driven Engineering: an Emerging Technical Space. Generative and Transformational Techniques in Software Engineering. Springer Berlin (pp. 36-64).

(Bonaventura et al., 2013) Bonaventura, M., Wainer, G.A., and Castro, R. 2013. Graphical Modeling and Simulation of Discrete-Event Systems with CD++ Builder. Simulation, 89(1): 4-27.

(Bousse et al., 2012) Bousse, E., Mentré, D., Combemale, B., Baudry, B., and Katsuragi, T. 2012. Aligning SysML with the B Method to Provide V&V for Systems Engineering. In Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. ACM (pp. 11-16).

(Budinsky, 2003) Budinsky, F. 2003. Eclipse Modeling Framework: a Developer's Guide. Addison-Wesley Professional.

(Capocchi et al., 2011) Capocchi, L., Santucci, J.F., Poggi, B., and Nicolai, C. 2011. DEVSimPy: A collaborative Python software for modeling and simulation of DEVS systems. In 2nd International Track on Collaborative Modeling & Simulation. IEEE (p. 6).

(Chow 1996) Chow A. 1996. Parallel DEVS: a Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulator. SCS Transaction on Simulation 13(2): 55-102.

(Emerson & Sztipanovits, 2006) Emerson, M., and Sztipanovits, J. 2006. Techniques for Metamodel Composition. In OOPSLA – 6th Workshop on Domain Specific Modeling (pp. 123-139).

(Gamma et al., 1995) Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design patterns. Elements of reusable object-oriented software. Addison-Wesley.

(Garavel & Hermanns, 2002) Garavel, H., and Hermanns H. 2002. On Combining Functional Verification and Performance Evaluation using CADP. Proceedings of the 11th International Symposium of Formal Methods Europe. Copenhagen, Denmark.

(Hamri & Zacharewicz, 2007) Hamri, M.E.A., and Zacharewicz, G. 2007. LSIS-DME: An Environment for Modeling and Simulation of DEVS Specifications. In AIS-CMS International Modeling and Simulation Multiconference (pp. 55-60).

(Harel & Rumpe, 2004) Harel, D., and Rumpe, B. 2004. Meaningful Modeling: What's the Semantics of "semantics?". Computer, 37(10): 64-72.

(Kim et al., 2009) Kim, S., Sarjoughian, H.S., and Elamvazhuthi, V. 2009. DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring. In Proceedings of the 2009 Spring Simulation Multiconference . Society for Computer Simulation International (p. 161).

(Kinoshita et al., 2014) Kinoshita, S., Nishimura, H., Takamura, H., and Mizuguchi, D. 2014. Describing Software Specification by Combining SysML with the B Method. In Software Reliability Engineering Workshops (ISSREW), IEEE (pp. 146-151).

(Kleppe, 2008) Kleppe, A. 2008. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Pearson Education.

(Kuhn et al., 2003) Kuhn, D.R., Craigen, D., and Saaltink, M. 2003. Practical Application of Formal Methods in Modeling and Simulation. In Summer Computer Simulation Conference. Society for Computer Simulation International (pp. 726-731).

(Laleau et al., 2010) Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., and Tatibouet, B. 2010. A First Attempt to Combine SysML Requirements Diagrams and B. Innovations in Systems and Software Engineering, 6(1-2): 47-54.

(Lano et al., 2004) Lano, K., Clark, D., and Androutsopoulos, K. 2004. UML to B: Formal Verification of Object-Oriented Models. Integrated Formal Methods. Springer Berlin (pp. 187-206).

(Lilius & Paltor, 1999) Lilius, J., and Paltor, I.P. 1999. vUML: A Tool for Verifying UML Models. In Automated Software Engineering. IEEE (pp. 255-258).

(Malik & Utting, 2005) Malik, P., and Utting, M. 2005. CZT: A Framework for Z Tools. In International Conference of B and Z Users. Springer Berlin (pp. 65-84).

(Mentré 2016) Mentré, D. 2016. SysML2B: Automatic Tool for B Project Graphical Architecture Design Using SysML. In International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. Springer International Publishing (pp. 308-311).

(Pnueli, 1977) Pnueli, A. 1977. The Temporal Logic of Programs. In 18th Annual Symposium on Foundations of Computer Science, IEEE (pp. 46-57).

(Risco-Martın et al., 2007) Risco-Martın, J.L., Mittal, S., Zeigler, B.P., and Jesús, M. 2007. From UML State Charts to DEVS State Machines using XML. In Proceedings of the Workshop on Multi-Paradigm Modeling: Concepts and Tools, Nashville, TN.

(Rumbaugh et al., 2004) Rumbaugh, J., Jacobson, I., and Booch, G. 2004. Unified Modeling Language Reference Manual, The. Pearson Higher Education.

(Sarjoughian & Zeigler, 1998) Sarjoughian, H.S., and Zeigler, BP. 1998. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. Simulation Series, 30: 29-36.

(Schamai et al., 2009) Schamai, W., Fritzson, P., Paredis, C., and Pop, A. 2009. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior using Graphical Notations. In Proceedings of the 7th International Modelica Conference. Linköping University Electronic Press (pp. 612-621).

(Shaikh & Vangheluwe, 2011) Shaikh, R., and Vangheluwe, H. 2011. Transforming UML2. 0 Class Diagrams and Statecharts to Atomic DEVS. In Proceedings of DEVS Integrative M&S Symposium. Society for Computer Simulation International (pp. 205-212).

(Smith, 2012) Smith, G. 2012. The Object-Z specification language (Vol. 1). Springer Science.

(Spivey, 1988) Spivey, J.M. 1988. Understanding Z: a Specification Language and its Formal Semantics, 3. Cambridge University Press.

(Sqali, 2012) Sqali H.M. 2012. Utilisation du Formalisme DEVS pour la Validation de Comportements des Systèmes à partir des Scénarios UML. PhD Thesis, Aix-Marseille, France.

(Traoré, 2008) Traoré, M.K. 2008. SimStudio: a Next Generation Modeling and Simulation Framework. In Proceedings of the 1st International Conference on Simulation Tools and Techniques, Communications, Networks and Systems Workshop (p. 67).

(Traoré 2006) Traoré, M.K. 2006. Analyzing Static and Temporal Properties of Simulation Models. In Proceedings of the 38th Conference on Winter Simulation (pp. 897-904).

(Trojet et al. 2009) Trojet, M. W., Frydman, C., & Hamri, M. E. A. 2009. Practical Application of Lightweight Z in DEVS Framework. In Proceedings of the 2009 Spring Simulation Multiconference. Society for Computer Simulation International (p. 154).

(Yacoub et al., 2014) Yacoub, A., Hamri, A., and Frydman, C. 2014. A Method for Improving the Verification and Validation of Systems by the Combined Use of Simulation and Formal Methods. In Proceedings of the 18th International Symposium on Distributed Simulation and Real Time Applications. ACM/IEEE (pp. 155-162).

(Zeigler 1976) Zeigler B.P. Theory of Modeling and Simulation. Wiley & Sons.

(Zeigler et al., 2000) Zeigler, B.P., Praehofer, H., and Kim, T.G. 2000. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic press.

(Zhou & Dicesare, 1993) Zhou, M., and Dicesare, F. 1993. Petri Net Synthesis for Discrete Event Control of Manufacturing Systems. Kluwer Academic Publishers. ISBN 0-7923-9289-2.