Adaptation d'exécution de modèles par contrats

Eric Cariou [1], **Olivier Le Goaer** [1], **Franck Barbier** [1], **Samson Pierre** [1], **Mohamed Graiet** [2]

1. *Université de Pau et des Pays de l'Adour /* LIUPPA
   B.P. 1155, 64013 Pau Cedex, France
   {Prenom.Nom}@univ-pau.fr
2. *Institut Supérieur d'Informatique et de Mathématiques de Monastir*
   Avenue de la Corniche, Monastir 5000, Tunisie
   Mohamed.Graiet@imag.fr

RÉSUMÉ. *Dans le contexte de l'ingénierie des modèles, l'exécution de modèles est un des moyens principaux pour supprimer le fossé entre le code, c'est-à-dire le système développé, et le modèle. Pour adapter une exécution de modèles, nous proposons de directement adapter le modèle en cours d'exécution. Cela oblige à rajouter des éléments nécessaires à l'adaptation dans le modèle mais évite d'avoir un second modèle dédié à l'adaptation comme c'est le cas pour les méthodes de type models@run.time. Cet article présente une approche d'adaptation directe d'exécution de modèle par contrats. Si des conditions du contrat ne sont pas respectées, une action d'adaptation doit être entreprise. L'approche est illustrée par un i-DSML (interpreted Domain Specific Modeling Language) de machines à états et un exemple de signalisation ferroviaire avec comme problématique de s'assurer qu'un modèle est adapté à un environnement d'exécution donné.*

ABSTRACT. *In the model-driven engineering field, model execution is one of the main ways to fill the gap between the code (i.e. the developed system) and the model. For adapting a model execution, we propose to directly adapt the model under execution. This requires to add onto the model requisite elements for the adaptation but this avoids to have a second model dedicated to the adaptation as for models@run.time. This paper proposes a contract-based approach for directly adapting a model execution. If some conditions of the contract are not respected, an adaptation action has to be undertaken. The approach is illustrated through a state machine i-DSML (interpreted Domain Specific Modeling Language) and an example of a railway signaling based on the ability to ensure that a model is adapted with a given execution environment.*

MOTS-CLÉS : *IDM, exécution de modèles, adaptation, contrats, machines à états.*

KEYWORDS: *MDE, model execution, adaptation, contracts, state machines.*

DOI:10.3166/TSI.34.703-730

1. Introduction

Un des buts fondateurs de l'ingénierie des modèles (IDM) est de considérer les modèles comme les éléments principaux et productifs pour le développement d'applications. On distingue généralement deux approches productives des modèles : la génération de code à partir d'un modèle ou l'interprétation directe du contenu d'un modèle. Selon cette dernière approche, plus récente, on dispose d'un moteur d'exécution implémentant une sémantique opérationnelle capable d'interpréter un modèle. De tels modèles sont écrits dans des langages de modélisation (DSML pour *Domain-Specific Modeling Language*) intégrant la possibilité d'interpréter leur contenu. On parle alors de i-DSML pour *interpreted*-DSML (Clarke *et al.*, 2013).

Parallèlement, les problématiques d'adaptation font l'objet en ingénierie logicielle d'un intérêt toujours grandissant (Salehie, Tahvildari, 2009 ; Barbier *et al.*, 2015). La capacité d'un système à s'adapter à son environnement ou pour diverses raisons permet à ce système de continuer à assurer un service, potentiellement en mode dégradé, là où il aurait pu être nécessaire de l'arrêter, de le modifier et de le relancer. Beaucoup de techniques d'adaptation consistent à associer le système en cours d'exécution avec une entité qui observe son état et les changements de l'environnement d'exécution puis agit sur le système au besoin pour le modifier, c'est-à-dire l'adapter. Dans ce contexte, l'IDM apporte notamment des solutions intéressantes grâce à la possibilité de disposer de modèles à l'exécution (*models@run.time*) (Blair *et al.*, 2009). De tels modèles ont principalement pour but de représenter l'état du système adaptable et d'y raisonner sur la nécessité d'adaptation.

Il est naturellement possible d'appliquer de telles techniques si on veut adapter une exécution de modèles. Cette adaptation consiste principalement à modifier le modèle en cours d'exécution. En effet, pour une exécution de modèles, le comportement du système est en fait principalement intégré et représenté par le modèle qui est exécuté. À ce titre, adapter le système revient donc à adapter le modèle, c'est-à-dire le modifier. Dans ce cadre là, nous expliquons qu'il est possible de directement adapter l'exécution du modèle sans avoir besoin d'une entité extérieure observant et analysant le système. Le moteur d'exécution peut être directement étendu pour intégrer les problématiques d'adaptation et modifier directement le modèle exécuté.

Nous avons commencé à étudier l'adaptation directe de l'exécution de modèles dans (Cariou, Graiet, 2012 ; Cariou *et al.*, 2012 ; 2013 ; Pierre *et al.*, 2014). Nous y avons notamment caractérisé la notion d'i-DSML adaptable, c'est-à-dire un DSML dont les modèles sont exécutables mais également adaptables. Dans cet article, nous expliquons qu'une approche par contrat permet d'intégrer dans un moteur d'exécution les problématiques d'adaptation directe d'exécution de modèles. Nous illustrons cela par un cas d'étude concernant l'adéquation d'un modèle exécuté à son environnement d'exécution. En effet, une des raisons courantes d'adapter un système concerne la prise en compte d'un changement dans son environnement d'exécution. Nous montrons, sur un exemple de machines à états, comment assurer qu'un modèle est adapté à un environnement d'exécution donné. Cette vérification peut être faite aussi bien pendant

l'exécution du modèle que de manière statique en dehors de toute exécution quand l'ensemble des interactions attendues avec l'environnement est connu. En cas de non adaptation à un environnement d'exécution, nous montrons comment des propriétés dédiées à l'adaptation et directement intégrées dans le modèle permettent de le modifier pendant son exécution afin de prendre en compte des interactions initialement non prévues avec l'environnement. Ces éléments dédiés à l'adaptation permettent également ici de définir une adéquation à un environnement d'exécution en mode dégradé.

La section suivante discute des différentes méthodes pour adapter une exécution de modèle sur la base d'un exemple simple d'exécution d'un processus devant respecter des contraintes temporelles. La section 3 présente notre cas d'étude, celui d'un train spécifié par des machines à états et les différents environnements d'exécution associés. La section 4 définit les contrats d'adaptation en les appliquant à ce cas d'étude. Enfin, nous discutons des travaux connexes dans la section 5 avant de conclure.

2. Adaptation d'exécution de modèles

2.1. Adaptation par une approche models@run.time

Comme précisé en introduction, beaucoup de techniques d'adaptation consistent à associer le système en cours d'exécution avec une entité qui observe son état et les changements de l'environnement d'exécution puis agit sur le système au besoin pour le modifier, c'est-à-dire l'adapter. Dans ce contexte, l'IDM apporte des solutions intéressantes pour le développement de systèmes logiciels adaptatifs (Fleurey, Solberg, 2009 ; Floch *et al.*, 2006 ; Zhang, Cheng, 2006) grâce à la possibilité de disposer de modèles à l'exécution (*models@run.time*) (Blair *et al.*, 2009). De tels modèles ont principalement pour but de représenter l'état du système adaptable et d'y raisonner sur la nécessité d'adaptation. Par exemple, (Morin *et al.*, 2009) utilise des métamodèles pour représenter les fonctionnalités, l'architecture, le contexte et des raisonnements pour une application basée composants. L'IDM offre ici l'intérêt de représenter sous la même forme (c'est-à-dire des modèles unifiés) toutes les informations requises pour l'adaptation. De plus, elle permet d'utiliser certains modèles définis en phase de conception directement en phase d'exécution. La contrepartie importante est qu'il faut assurer que les modèles représentent de manière cohérente et causale le système en cours d'exécution (Blair *et al.*, 2009 ; Lehmann *et al.*, 2010 ; Vogel, Glese, 2011).

La figure 1a présente la relation entre un système en cours d'exécution et le modèle le représentant, modèle sur lequel la réflexion sur la nécessité d'adaptation est réalisée. Il s'agit d'une boucle entre le modèle et le système. Le modèle est mis à jour par rapport aux changements d'état du système, ce qui engendre une analyse sur la nécessité d'adaptation qui, le cas échéant, déclenche des modifications du système, et ainsi de suite. Ce modèle est géré par un moteur d'adaptation s'exécutant conjointement au système et interagissant avec lui.

La figure 1a est une vision générale et simplifiée des différentes boucles d'adaptation qui peuvent être implémentées en pratique. Une des boucles les plus connues est

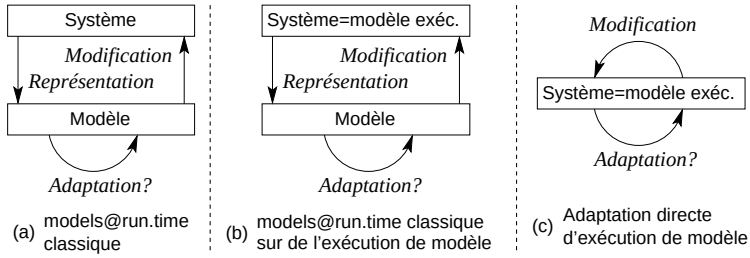


Figure 1. Variantes de boucles d'adaptation par des modèles

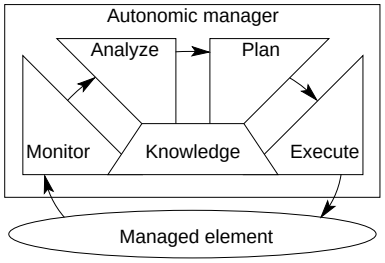


Figure 2. La boucle d'adaptation MAPE-K (Kephart, Chess, 2003 ; IBM, 2005)

MAPE-K définie par IBM (Kephart, Chess, 2003 ; IBM, 2005). Elle est représentée sur la figure 2. Un gestionnaire d'adaptation (*Autonomic manager*) est associé à l'élément du système que l'on veut adapter (*Managed element*). La boucle consiste à enchaîner un ensemble de phases. La première observe l'état courant de l'élément (*Monitor*) et met à jour la connaissance requise pour l'adaptation (*Knowledge*). Cette connaissance contient des informations sur l'état de l'élément du système ou bien encore des politiques d'adaptation qui pourront être appliquées. Cette connaissance est ensuite analysée pour savoir si une adaptation est requise (*Analyze*). Si oui, une modification de l'élément du système est planifiée (*Plan*) puis exécutée (*Execute*). L'acronyme MAPE-K reprend l'initiale de chacune de ces phases. Si cette boucle est implémentée comme une approche de type *models@run.time*, c'est la partie *Knowledge* qui pourra ici être constituée d'un ensemble de modèles.

2.2. Exemple d'adaptation d'exécution de modèles

Dans (Pierre *et al.*, 2014) nous présentons un exemple d'adaptation d'exécution de modèles définissant des processus. Un processus est composé d'un ensemble d'activités, celles-ci pouvant être regroupées dans des séquences s'exécutant en parallèle. La figure 3 présente un exemple de processus autour du développement d'un logiciel. Dans la partie haute, les activités actuellement en cours de réalisation sont les ellipses avec un fond gris. Elle concernent ici l'implémentation des fonctionnalités du logiciel (`CoreImpl`) et de la documentation utilisateur (`UserDoc`). L'activité d'implémentation des fonctionnalités sera suivie de leur test (`CoreTest`) puis de la phase

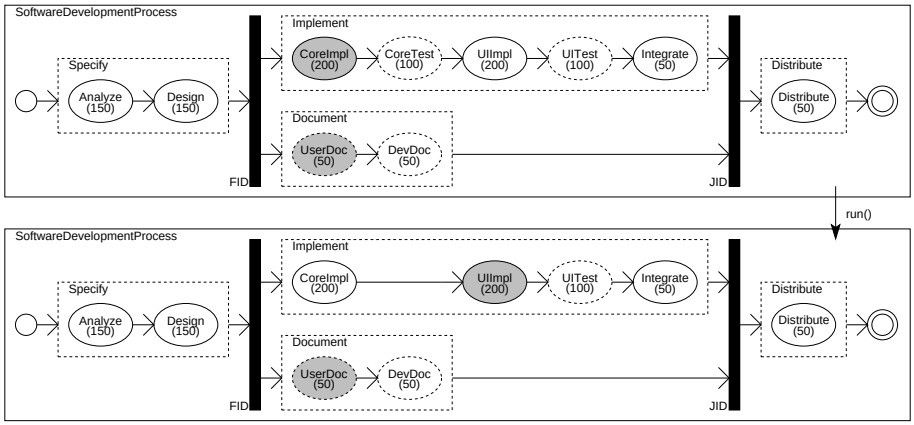


Figure 3. Adaptation d'un processus

d'implémentation de l'interface utilisateur (`UIImpl`). Chaque activité est prévue pour se réaliser en un certain temps prévisionnel (le nombre précisé dans chaque ellipse). Si on se rend compte que l'on a pris du retard dans le déroulement du processus, afin de regagner du temps, l'adaptation va consister ici à ne pas réaliser certaines activités. Bien sur, on ne peut pas choisir au hasard les activités à sauter, donc il faut que le concepteur du processus marque explicitement les activités qui ne sont pas strictement obligatoires dans son processus. Graphiquement, de telles activités sont représentées avec une ellipse en pointillé. Pour notre exemple, il s'agit des activités de documentation et de test (dont `CoreTest`).

Le passage de la partie haute à la partie basse de la figure 3 est un pas d'exécution du processus. Une activité qui était en cours s'est terminée et le moteur d'exécution doit activer la suivante. Cela se fait en exécutant l'opération `run()` du moteur. Ici, l'activité `CoreImpl` se terminait et il faut passer à son activité suivante. Mais comme il y avait du retard dans l'exécution du processus et que l'activité suivante `CoreTest` était marquée comme non obligatoire, celle-ci est supprimée du processus et on se retrouve donc avec la réalisation de l'activité `UIImpl` qui devient la suivante directe de `CoreImpl`.

2.3. Adaptation directe de l'exécution du modèle

Si l'on veut implémenter l'adaptation d'exécution de processus de notre exemple, il est bien entendu possible d'appliquer les principes des approches *models@run.time* en développant un moteur d'adaptation annexe qui se basera sur un modèle représentant le système en cours d'exécution. La particularité ici est simplement que le système en question est un moteur d'exécution interprétant un modèle. La figure 1b représente ce cas particulier. Dans ce contexte, on se retrouve avec deux modèles : le modèle en cours d'exécution (ici un processus) et le modèle servant à l'adaptation. Ce dernier représente le modèle en cours d'exécution avec des informations qui en sont extraites

ou déduites afin de gérer l'adaptation. Pour notre exemple, ce modèle sera quasiment une copie conforme du modèle en cours d'exécution. En effet, pour savoir si on est retard, il faut connaître les activités courantes ainsi que le temps prévisionnel des activités les précédant. Pour savoir si on peut supprimer une activité, il faut savoir si elle est obligatoire ou non. Toutes ces informations sont dans le modèle en cours d'exécution et représentent même la majeure partie du contenu du modèle. De plus, pour savoir si on doit réaliser l'adaptation, cela nécessite de la part du moteur d'adaptation « d'intercepter » l'appel de l'opération `run()` sur le moteur d'exécution puis ensuite d'aller au besoin modifier « de l'extérieur » via des fonctionnalités dédiées à cela, le modèle en cours d'exécution.

Tout cela est relativement complexe à réaliser et induit des redondances inutiles entre les modèles. Nous proposons donc une autre solution, consistant à directement adapter le modèle en cours d'exécution via un moteur d'exécution étendu. La figure 1c montre la nouvelle boucle d'adaptation : le contenu du modèle exécuté est analysé pour savoir si ce modèle doit être adapté et les modifications associées sont directement faites sur le modèle. Cela nécessite tout de même de rajouter au besoin dans le modèle des informations dédiées à l'adaptation. Cela peut complexifier plus ou moins grandement le modèle mais évite la nécessité de mettre en œuvre des techniques relativement complexes pour assurer la cohérence entre le modèle dédié à l'adaptation et le système. De plus, l'implémentation de l'adaptation peut se faire de manière relativement simple. En utilisant une approche par contrat par exemple, il suffit de rajouter sur les opérations d'exécution du moteur une précondition qui détermine si l'adaptation doit être faite ou pas. Ici, pour notre opération `run()`, cette précondition vérifiera que l'on n'est pas en retard. En cas de violation de la précondition, on modifiera si possible le modèle via une opération d'adaptation dédiée et directement appelée par le moteur d'exécution.

Pour cet exemple d'adaptation de processus, on constate aisément que l'adaptation directe est bien plus simple car il s'agit en effet de rajouter dans le moteur d'exécution une opération allant regarder la valeur d'un simple attribut booléen dans une activité (précisant si on peut la sauter ou pas) et modifiant directement le modèle le cas échéant. Une approche de type *models@run.time* engendrerait ici une complexité d'implémentation et de fonctionnement totalement inutile. Si l'approche classique d'adaptation via des techniques de *models@run.time* est quoi qu'il arrive utilisable pour adapter une exécution de modèles et pourrait éventuellement s'avérer plus intéressante dans certains rares cas, nous pensons que de manière générale l'approche à privilégier est celle de l'adaptation directe. Au delà de ce que nous venons de montrer avec notre exemple, nous avons également expliqué dans (Cariou *et al.*, 2013) qu'un i-DSML adaptable dont les modèles sont exécutables et adaptables peut être vu comme une extension directe et logique d'un i-DSML : des éléments dédiés à l'adaptation sont rajoutés dans le métamodèle du i-DSML et des opérations d'adaptation sont rajoutées directement dans le moteur d'exécution, étendant en cela la sémantique d'exécution par une sémantique d'adaptation.

Dans la suite de cet article, nous détaillons un autre exemple de i-DSML adaptable avec des machines à états et une adaptation ayant pour but de s'assurer que le modèle est adapté à son environnement d'exécution. Nous expliquons ensuite comment on peut facilement intégrer les opérations d'adaptation avec les opérations d'exécution dans le moteur via une approche par contrats.

3. Cas d'étude : adéquation d'une machine à états avec un environnement d'exécution

Dans cette section, nous détaillons un exemple d'adéquation d'un modèle à différents contextes d'exécution. Les explications sont données à partir de l'exemple de la machine à états d'un train pouvant circuler sur plusieurs types de voies ferrées [1]. Avant de présenter cet exemple, nous définissons le langage de modélisation (le DSML) que nous utilisons pour décrire la machine à états du train. Il s'agit de machines à états UML restreintes afin de rendre nos exemples plus simples et concis. Par rapport aux machines à états UML standard, nous conservons les fonctionnalités d'état composite (sans parallélisme), d'état historique et considérons des transitions associées à un événement, sans garde. Dans la section suivante, nous verrons comment modifier le modèle pour s'adapter à un environnement d'exécution non connu.

3.1. Un i-DSML de machines à états

Comme expliqué dans (Cariou *et al.*, 2013), le métamodèle d'un i-DSML adaptable doit permettre de définir des modèles auto-contenus permettant à la fois de gérer l'exécution et l'adaptation des modèles. À la base, un modèle exécutable doit contenir tous les éléments requis pour représenter son état complet pendant son exécution. Pour cela il contient une partie définissant ses éléments statiques et une extension dynamique spécifiant son état pendant l'exécution (Breton, Bézivin, 2001 ; Cariou *et al.*, 2011 ; Clarke *et al.*, 2013 ; Combemale *et al.*, 2012 ; Lehmann *et al.*, 2010). Au delà de ce contenu classique d'un modèle exécutable, l'adaptation du modèle pourra être facilitée ou guidée par des éléments particuliers qui sont définis dans la partie d'adaptation du métamodèle. Dans l'exemple de processus de la section 2.2, il s'agissait de marquer des activités. Cela se faisait avec un simple attribut booléen placé dans le méta-élément définissant une activité.

3.1.1. Définition du métamodèle

Notre métamodèle de machines à états est représenté dans la figure 4. Sa partie statique permet de définir l'ensemble des états et des transitions d'une machine à états. Elle contient classiquement les éléments suivants :

1. L'exemple de cet article est librement inspiré d'un système ferroviaire, avec une simplification de la réalité et des libertés par rapport à cette même réalité. Il ne faut donc pas le considérer comme le cas d'étude d'un système réel.

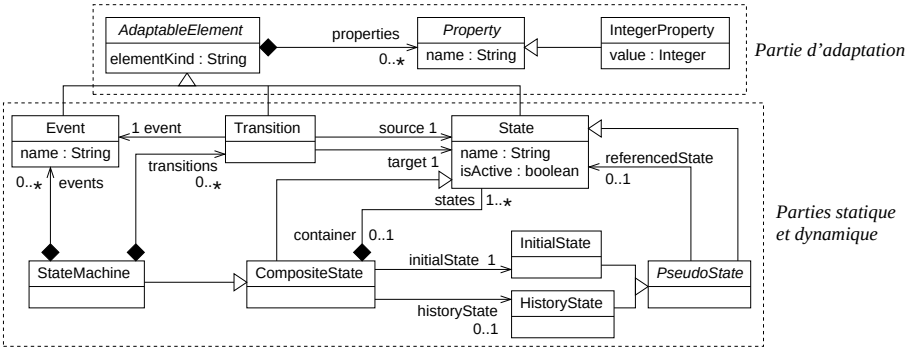


Figure 4. Métamodèle correspondant au i-DSML de machines à états

– Un *état* qui a un nom et est contenu dans un *état composite*;

– Deux types de *pseudo états* peuvent être définis : un *état initial* et un *état historique* (de type « profond » à la UML) référençant chacun un état de l'état composite auquel ils appartiennent. Tout composite contient obligatoirement un état initial et facultativement un état historique;

– Une *transition* entre un état source et un état cible, associée à un *événement* représenté par un nom;

– Une *machine à états* est un type particulier d'état composite (représentant la hiérarchie des états de la machine) avec en addition un ensemble d'événements et de transitions.

La partie dynamique du métamodèle est composée des deux éléments suivants qui permettent de représenter l'état courant d'une machine à états en cours d'exécution :

– L'attribut booléen `isActive` d'un état. Il précise si l'état est actif ou non;

– L'état référencé par un état historique (de type *deep history* en UML) qui référence le dernier état qui est (ou était) actif pour le composite dans lequel il est contenu.

Enfin, la partie d'adaptation est composée des deux éléments suivants :

– L'élément abstrait `AdaptableElement` qui est spécialisé par tous les éléments statiques et dynamiques du métamodèle. Il permet de définir, via l'attribut `elementKind`, un type pour les états, transitions ou événements d'une machine à états. En pratique, ce type est similaire à un stéréotype tel que défini dans les profils UML en permettant de préciser qu'un élément joue un rôle particulier;

– Chaque élément du métamodèle, via la spécialisation de `AdaptableElement`, contiendra un ensemble de *propriétés*. Ces propriétés peuvent être de différentes natures, mais dans un souci de concision, nous ne définissons ici que des *propriétés entières*. Une telle propriété est un couple <nom, valeur entière>. En pratique, une propriété est similaire à une valeur marquée telle que définie dans les profils UML.

En permettant de définir des informations supplémentaires sur des éléments, ces types et propriétés joueront un double rôle pour l'adaptation, au niveau vérification puis au niveau modification. Ainsi, on pourra d'abord comparer le type ou les valeurs d'un événement pour savoir s'il est connu et géré par la machine à états. Si ça n'est pas le cas, on pourra alors modifier la machine à états en conséquence en tirant parti de ces informations pour qu'elle s'adapte à cet événement inconnu. Ces vérifications et modifications seront détaillées dans la suite de l'article en section 4.

Notons que cette partie d'adaptation (`AdaptableElement` et les propriétés associées) est ici générique pour pouvoir être réutilisée au besoin dans différents contextes. En définissant des propriétés au sens large et sans être liées à un domaine métier particulier, on pourra les utiliser pour plusieurs types de modèles, pas seulement un train comme pour notre exemple. En complément, cette partie pourra être automatiquement ajoutée à tout métamodèle. En effet, il suffit de rajouter ces méta-éléments et de faire hériter tous les méta-éléments existants de `AdaptableElement`, ce qui se fait facilement via une transformation de métamodèles. Cette partie d'adaptation est ici volontairement réduite par soucis de concision des exemples. Si l'on veut quelque chose de plus complet et détaillé selon les besoins, on pourra bien entendu l'étendre au besoin ou réutiliser des résultats de travaux existants comme par exemple le métamodèle générique de (Fleurey, Solberg, 2009) qui permet de définir des propriétés, des conditions associées et des règles d'adaptation.

Les parties statiques, dynamiques et d'adaptation ne sont bien sûr pas suffisantes pour définir complètement le métamodèle. Il faut en effet compléter chacune d'elles de manière classique par des règles de bonne formation sous la forme d'invariants OCL. Par exemple, un invariant précisera qu'un état référencé par un état historique est forcément un état du même composite, un autre que si un composite est actif alors un et un seul de ses composants internes doit être actif ou bien encore que les propriétés associées à un élément ont un nom unique.

3.1.2. Machines à états spécialisées

Dans (Pierre *et al.*, 2014) nous expliquons que pour définir des actions d'adaptation génériques, il est généralement utile voire nécessaire de spécialiser un i-DSML en définissant des sous-types de modèle (Steel, Jézéquel, 2007 ; Guy *et al.*, 2012 ; Wuliang *et al.*, 2013). Par générique, nous entendons être totalement indépendant du contenu métier du modèle. Concrètement ici, pour nos machines à états, l'adaptation doit pouvoir se faire de manière automatique sur un modèle sans avoir besoin de savoir s'il représente le comportement d'un train, d'un ascenseur ou d'un four à micro-ondes.

Nous ajoutons donc ici une spécialisation complémentaire sur nos machines à états. Il s'agit d'intégrer le fait qu'un événement donné conduit toujours à un même état quelque soit le départ de la transition associée. La modélisation de notre train respectera cette contrainte. En effet, un signal d'une certaine couleur oblige le train à toujours rouler à une vitesse donnée, quelle que soit la vitesse courante du train. Par exemple, un feu rouge oblige systématiquement le train à s'arrêter ; aucun autre comportement n'est acceptable. Comme nous le verrons par la suite, sans cette spé-

cialisation, il ne sera pas possible de savoir comment automatiquement modifier le modèle pour l'adapter à un événement inconnu. Techniquement, cette spécialisation est définie en rajoutant des invariants OCL sur le métamodèle.

3.2. Environnements d'exécution pour un train

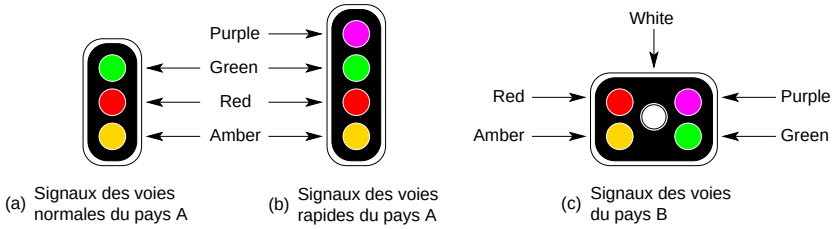


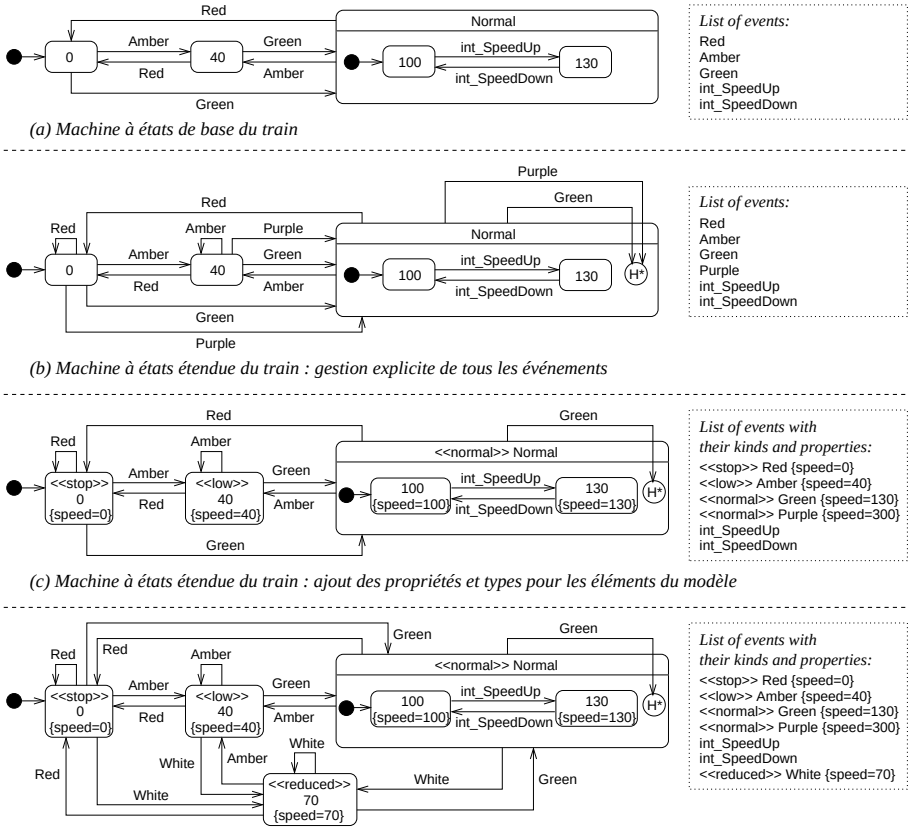
Figure 5. Trois types de signalisation ferroviaire

La figure 5 représente trois types de signalisation ferroviaire lumineuse. Le pays A possède deux types de voies ferrées : (a) des voies à vitesse normale (jusqu'à 130 km/h) et (b) des voies à haute vitesse (jusqu'à 300 km/h). La signalisation consiste en deux types de signaux : un panneau à 3 couleurs pour les voies normales et un panneau étendu à 4 couleurs pour les voies rapides. Les panneaux servent à préciser aux trains la vitesse à respecter en croisant le panneau. La couleur rouge correspond à un arrêt pour le train (vitesse de 0 km/h), la couleur orange à une vitesse lente (40 km/h), la couleur verte à une vitesse normale (jusqu'à 130 km/h) et la couleur mauve (seulement pour les sections rapides) à une très grande vitesse (jusqu'à 300 km/h). Dans notre exemple, nous définirons le modèle – la machine à états – pour un train du pays A à vitesse normale, c'est-à-dire ne pouvant pas rouler à plus de 130 km/h. Ceci dit, il peut rouler à 130 km/h sur une voie à haute vitesse.

Le pays B possède lui une signalisation différente, avec des panneaux rectangulaires à 5 couleurs (c). Les couleurs identiques à celles des panneaux de A ont le même type de signification mais peuvent correspondre à des valeurs de vitesse légèrement différentes. Le rouge correspond à un arrêt (0 km/h), l'orange à une vitesse lente (30 km/h), le vert à une vitesse normale (jusqu'à 120 km/h) et le mauve à une très haute vitesse (jusqu'à 350 km/h). La couleur blanche quant à elle a une signification inconnue pour le conducteur du train du pays A.

3.3. Machine à états d'un train

La figure 6a représente la machine à états d'un train à vitesse normale du pays A. Le nom d'un état correspond à la vitesse du train et les événements sur les transitions correspondent, sauf cas particuliers, aux couleurs des signaux croisés par le train. Par exemple, une couleur orange fait passer le train dans un état à une vitesse de 40 km/h, quelque soit la vitesse à laquelle il roulait précédemment. L'état composite nommé `Normal` sert à définir les vitesses en marche normale : le train peut rouler à 100 ou



(a) Machine à états de base du train

(b) Machine à états étendue du train : gestion explicite de tous les événements

(c) Machine à états étendue du train : ajout des propriétés et types pour les éléments du modèle

(d) Adaptation de la machine à états du train : addition automatique d'un état et des transitions associées

Figure 6. Variantes de machines à états pour un train

130 km/h et c'est le conducteur du train qui décide entre ces deux vitesses. Les événements `int_Speed[Up/Down]` sont donc des événements internes déclenchés par le conducteur, c'est-à-dire par le train, contrairement aux événements `Red`, `Amber` et `Green` qui correspondent à des événements extérieurs, c'est-à-dire à des interactions avec l'environnement d'exécution. Ce sont bien entendu ces événements extérieurs qui nous intéressent pour l'adaptation.

3.4. Modélisation explicite des interactions avec l'environnement d'exécution

Bien que correcte, la spécification de notre machine à états pose des problèmes en ce qui concerne l'adaptation à un environnement d'exécution. Cela est dû à l'information implicite qu'elle contient. Par exemple, si un signal mauve est croisé par le train, comme la machine à états ne contient aucune transition associée, alors l'état actif ne change pas. Si on suppose qu'un signal mauve ne peut être croisé qu'après un signal vert, alors le train sera dans un état de vitesse normale et le restera après avoir croisé

un signal mauve (il roulera donc à 100 ou 130 km/h sur une voie rapide). C'est le comportement attendu, mais il est implicitement spécifié. Par contre, si le train traverse la frontière et roule sur les voies ferrées du pays B et qu'il croise un signal blanc (inconnu par le conducteur et n'étant pas géré par la machine à états) alors le train restera à la même vitesse que celle à laquelle il roulait : le signal est purement et simplement ignoré car inconnu. Or il est évident qu'ignorer systématiquement un signal inconnu n'est pas un comportement opportun. La règle que nous imposons donc est que l'intégralité des interactions prévues avec l'environnement d'exécution est explicitement et systématiquement prise en compte. Ainsi, il devient aisé de différencier une interaction non prévue (comme le signal blanc) d'une interaction connue (un signal rouge, orange, vert ou mauve) quand bien même elle ne modifie pas la vitesse du train.

Pour nos machines à états, nous pouvons notamment appliquer deux solutions pour implémenter cette définition explicite des interactions. La première consiste à disposer à l'exécution, au sein du moteur, de la liste explicite des événements attendus et de vérifier avant de traiter l'occurrence d'un événement que ce dernier est bien dans cette liste. L'inconvénient de cette approche est qu'elle nécessite de paramétrer le moteur pour lui passer, en plus du modèle à exécuter, la liste des événements attendus. La seconde solution consiste à disposer d'une machine à états qui définit explicitement une transition partant de chaque état pour chacun des événements de cette liste (en théorie des automates, on parlerait d'automate complet). Cette solution nécessite de modifier la structure de la machine à états en ajoutant les transitions manquantes. Cette modification peut être faite de manière automatique par une simple transformation de modèles. En effet, comme un événement mène toujours au même état (cf. spécialisation expliquée en section 3.1.2), il suffit de se baser sur les transitions existantes pour les dupliquer en les faisant partir des états requis. Ainsi, cette restructuration ne nécessite pas ici de travail supplémentaire au concepteur de la machine à états. L'intérêt de cette deuxième solution est également que toute l'information requise (ici l'ensemble des interactions attendues) est directement embarquée dans le modèle, ce qui correspond à l'esprit des i-DSML où tout ce qui est requis est auto-contenu dans le modèle. C'est ce second choix que nous mettons en œuvre dans cet article.

La figure 6b complète ainsi la machine à états précédente en ajoutant explicitement à partir de chaque état une transition pour chacune des quatre couleurs de signal du pays A. Pour un état composite, un état historique est ajouté et est la cible des auto-transitions pour assurer de rester à la même vitesse quand on croise un signal ne faisant pas changer la vitesse du train. Les états composites permettent aussi de factoriser des transitions en une seule. Pour l'état composite `Normal`, toutes les transitions partent directement de cet état et s'appliquent donc à ses états internes de 100 et 130 km/h. Dès lors, même si le train ne peut toujours pas rouler à plus de 130 km/h, l'événement mauve est correctement géré. La machine à états est valide pour les deux types de voies du pays A : chaque couleur de signal (rouge, orange, vert ou mauve) est explicitement associée à une transition et est donc explicitement gérée. Par contre, la couleur blanche inconnue ne correspond pour aucun état à aucune transition. Comme les événements connus sont explicitement et systématiquement gérés, on sait alors que la couleur blanche correspond à un événement inattendu. En d'autres termes, on

se rend compte en croisant un signal de couleur blanche que le modèle n'est pas adapté au contexte, à l'environnement d'exécution. Il faudra alors prendre une décision : par exemple soit modifier le modèle pour l'adapter, soit charger un autre modèle qui lui est adapté à l'environnement (c'est-à-dire qui gère le signal de couleur blanche), soit arrêter tout bonnement l'exécution de la machine à états.

3.5. Types d'éléments pour la définition d'un modèle dégradé

Vérifier que chaque état possède une transition pour chaque événement attendu (c'est-à-dire généré par l'environnement d'exécution) n'est pas suffisant. Il faut également vérifier que les événements sont correctement gérés. De manière générale, y compris pour les exécutions d'autres types de modèles que les machines à états, s'assurer que les interactions avec l'environnement sont correctement gérées est bien entendu une tâche des plus ardues. Néanmoins, dans notre contexte de machines à états spécialisées, il est possible de faire, parmi d'autres, un choix de vérification qui consiste à s'assurer qu'un événement d'un certain type conduit à activer un état du même type. Par exemple, la couleur orange d'un signal correspond à une demande de vitesse lente et doit mener à un état de vitesse lente. Pour le pays A, une vitesse de 40 km/h est considérée comme lente. Pour le pays B par contre, c'est une vitesse de 30 km/h. Ces deux vitesses ne sont pas identiques mais restent relativement proches. Cela fait qu'un train de A roulant dans le pays B se trouvera par défaut à 40 km/h au lieu des 30 km/h requis. Cette vitesse sera suffisamment proche de celle attendue pour être acceptable ou, en d'autres termes, pour être considérée comme un comportement dégradé acceptable d'un point de vue vitesse.

Ces types d'états ou d'événements seront définis grâce aux éléments d'adaptation introduits dans le métamodèle. Chaque état ou événement sera marqué par un type défini grâce à l'attribut hérité `elementKind`. Ainsi, il est possible de marquer tous les éléments d'une machine à états avec un type en plus de la valeur exacte de l'élément. Cette double précision permettra de choisir entre une vérification exacte ou lâche se basant sur des types d'éléments. Par exemple, des états de vitesses de 30 ou 40 km/h, différentes mais suffisamment proches pour être considérées comme similaires, seront alors marqués comme de type « lent ». Concernant la valeur exacte d'un élément, on pourra soit simplement se baser sur son nom, soit être plus précis en utilisant des propriétés valuées. Ici par exemple, pour notre train, chaque état ou événement pourra posséder une propriété valuée précisant la vitesse associée.

La figure 6c représente la modification de notre machine à états du train à vitesse normale de A en intégrant des types d'éléments (ces types d'éléments sont représentés avec la notation << ... >> des stéréotypes UML). L'événement rouge est un événement de type arrêt, l'événement orange de type lent et les événements vert et mauve sont tous les deux considérés comme de type normal. Les états sont eux aussi marqués par des types : l'état 0 km/h est un état de type arrêt, l'état 40 km/h un état de type lent et l'état composite normal un état de type normal. À chaque état ou événement est associé une propriété de vitesse (nommée `speed`). Pour un état, cela correspond

à la vitesse à laquelle roule le train et pour un événement, la vitesse à laquelle doit rouler le train lorsque l'événement est généré, c'est-à-dire lorsque le signal associé est croisé. Par exemple, l'état 40 et l'événement `Amber` auront tous les deux une propriété `speed` positionnée à 40 km/h. Ces propriétés sont notées avec la syntaxe entre accolades des valeurs marquées des profils UML.

L'intérêt de cette nouvelle spécification est que cette machine à états, si l'on se base sur des vérifications lâches (c'est-à-dire sur les types des éléments) et non pas exactes (sur les valeurs précises des éléments), permet au train A de rouler indifféremment sur les deux voies de A (le signal mauve est géré comme un signal vert grâce au type normal) et sur les voies de B (à l'exception du signal blanc, toujours inconnu). Pour B, on pourra en effet par exemple marquer l'état de 30 km/h comme un état lent et on pourra s'assurer que le signal orange, de type lent, conduit bien à un état lent quand bien même il s'agit de l'état lent de A, à 40 km/h, et non pas exactement de l'état lent de B, à 30 km/h. Ainsi, on peut considérer que cette machine à états de A pour un train à vitesse normale est une version dégradée par rapport à une machine à états de référence d'un train du pays B. En d'autres termes, cette machine à états de A est adaptée à tous les environnements d'exécution de A et de B (exceptée la couleur blanche pour B, encore une fois).

3.6. Caractérisation d'un modèle pour une vérification de son adaptation à un environnement

Dans les discussions ci-dessus, nous avons présenté une manière de définir un modèle pour vérifier son adéquation à un environnement d'exécution. En généralisant ce que nous avons appliqué ici à nos machines à états de train, nous pouvons définir une caractérisation plus globale qui pourra être réutilisée dans d'autres contextes selon les besoins et si elle convient. Cette caractérisation est la suivante :

Comportement du système : le but principal du modèle est de spécifier le comportement du système (les états d'une machine à états par exemple);

Interactions avec l'environnement d'exécution : le modèle doit intégrer la spécification la plus complète et exhaustive possible des interactions avec l'environnement d'exécution, les spécifications implicites sont à rendre explicites (par exemple, chaque événement attendu est associé à une transition pour tout état);

Connaissance exacte ou limitée des éléments : un élément peut être défini avec un double niveau de précision (voire plusieurs autres niveaux), le premier étant sa valeur exacte et le second un type plus général et abstrait;

Connaissance totale ou partielle des interactions : l'ensemble des interactions possibles entre l'environnement d'exécution et le système peuvent être statiquement connues ou certaines interactions peuvent être « découvertes » à l'exécution.

Si l'ensemble des interactions est fini et déterminé à l'avance, alors il est possible statiquement (avant et indépendamment de l'exécution) de vérifier qu'un modèle est

adapté à un environnement d'exécution donné. Il suffit de vérifier qu'il traite de manière systématique toutes les interactions (dans notre exemple de machine à états, on vérifiera qu'un événement est associé à une transition partant de tout état de la machine).

Tableau 1. Quatre combinaisons de vérification d'adaptation

	Comportement exact	**Type de comportement**
Interaction exacte	Un comportement exact est valide pour des interactions précisément définies. Le modèle est un modèle de référence pour cet environnement d'exécution.	Un comportement dégradé est valide pour un environnement d'exécution précis.
Type d'interaction	Le comportement du modèle est directement valide pour un environnement d'exécution proche par rapport à l'environnement pour lequel il représente un modèle de référence.	Un comportement dégradé est valide pour un environnement d'exécution partiellement connu.

Concernant la connaissance requise sur un élément, elle est soit en mode exact, soit en mode limité (via un type d'élément). Comme il y a d'un côté la définition du comportement et de l'autre celle des interactions, cela donne au total quatre combinaisons possibles. Chaque combinaison définit un niveau de vérification d'adéquation du modèle à l'environnement d'exécution, du plus complet (connaissance exacte du comportement et des interactions) au plus lâche (connaissance limitée via les types de comportement et d'interactions), formant ainsi une hiérarchie de niveaux de vérification. Ces niveaux de vérification seront implémentés par des contrats comme nous le verrons dans la section suivante. La table 1 résume ces quatre combinaisons.

3.7. Rendre adaptable un modèle exécutable pour la gestion des interactions

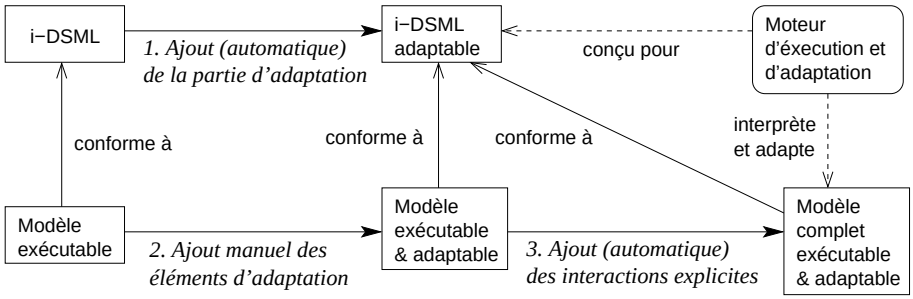


Figure 7. Processus possible pour rendre adaptable un modèle exécutable

La figure 7 montre un processus possible qui permet de passer d'un modèle exécutable à un modèle exécutable adaptable respectant les contraintes de spécifications

des interactions avec l'environnement que nous venons de présenter. Ce processus est celui qu'on peut appliquer sur notre i-DSML de machines à états. Il sera potentiellement différent pour d'autres i-DSML, avec par exemple des transformations plus ou moins automatiques voire inutiles (si toutes les interactions sont par principe déjà spécifiées dans le modèle initial ou gérées avec des définitions extérieures au modèle par exemple).

Comme point de départ, nous avons un modèle exécutable conforme à son i-DSML (par exemple notre train de la figure 6a). La première transformation consiste à automatiquement ajouter la partie d'adaptation sur le métamodèle, aboutissant alors à notre i-DSML adaptable de la figure 4. Le modèle instancié est ensuite enrichi par le concepteur pour ajouter les informations qui serviront à l'adaptation. Cette opération n'est pas automatisable car elle dépend du contenu métier du modèle. Par exemple, c'est le concepteur qui sait qu'un événement orange correspond à un type lent associé à une vitesse de 40 km/h, c'est donc à lui de rajouter ces informations sur le modèle. Une fois le modèle enrichi avec ces informations, il est automatiquement transformé pour expliciter toutes les interactions (ce qui donne la figure 6c pour notre train). Le moteur d'exécution, qui intègre également les politiques d'adaptation et qui est défini pour le i-DSML adaptable, va alors se charger d'interpréter et d'adapter au besoin ce modèle.

4. Adaptation d'exécution de modèles par contrats

Dans cette section, nous expliquons comment mettre en œuvre des contrats dédiés à l'adaptation et comment les utiliser pour réaliser une adaptation [2]. Après avoir rappelé les principes de la conception par contrats et défini les contrats d'adaptation, nous appliquons ces contrats sur notre cas d'étude de la section précédente, par rapport aux différents niveaux de vérification d'adaptation et présentons des actions d'adaptation associées.

4.1. Conception par contrats

La conception et la programmation par contrats (Meyer, 1992 ; Beugnard *et al.*, 1999 ; Le Traon *et al.*, 2006) est une approche bien connue de vérification d'exécution d'éléments logiciels. De manière traditionnelle, une approche par contrats consiste à spécifier des éléments logiciels via des invariants et leurs opérations via des pré et des post-conditions. Le but est de spécifier ce que font ces éléments, comment les utiliser correctement et de s'assurer qu'ils fonctionnent correctement. Les invariants spécifient les contraintes que l'élément logiciel doit respecter en permanence. La pré-condition d'une opération définit l'état que le système doit respecter ou les contraintes sur les

2. Les différents modèles et contrats présentés dans cet article ainsi que le moteur d'exécution et d'adaptation du i-DSML adaptable de machines à états sont disponibles à l'adresse suivante (avec également la présentation d'une trace d'exécution/adaptation) :
`http://ecariou.perso.univ-pau.fr/adapt-iDSML/`

paramètres de l'opération pour que celle-ci puisse être appelée. La post-condition spécifie l'état que le système s'engage à respecter après l'appel. Si la pré-condition n'est pas respectée, la post-condition n'est pas assurée.

La conception par contrats appliquée à l'ingénierie des modèles consiste à spécifier des contraintes sur des modèles et les opérations les manipulant. Les invariants sur les modèles peuvent être les règles de bonne formation ou d'autres contraintes supplémentaires sur les éléments définissant un métamodèle. Concernant les opérations sur les modèles, la plupart sont des opérations de transformations de modèles visant à produire un nouveau modèle à partir d'un premier ou à modifier le modèle courant. À ce titre, dans (Cariou *et al.*, 2004) nous avons défini le concept de contrats de transformations de modèles. La pré-condition d'une opération de transformation spécifie les contraintes à respecter par un modèle source pour pouvoir être transformé et la post-condition, les contraintes à respecter par le modèle cible pour être considéré comme un résultat valide de la transformation. Ces dernières contraintes se décomposent en deux groupes : d'un coté des contraintes s'appliquant uniquement sur le modèle cible et de l'autre des contraintes sur le contenu du modèle cible par rapport aux éléments du modèle source. Les contrats peuvent bien sur être utilisés pour vérifier d'autres manipulations de modèles que des transformations. Par exemple, dans (Cariou *et al.*, 2011), nous les utilisons pour vérifier des exécutions de modèles. Toutefois, une exécution peut être ramenée à une série de transformations endogènes. En effet, chaque pas d'exécution modifie le modèle en cours d'exécution, et plus exactement, modifie les éléments de sa partie dynamique pour changer son état courant (Cariou *et al.*, 2013).

4.2. Contrats d'adaptation

Dans (Cariou *et al.*, 2013), dans un contexte d'adaptation d'exécution de modèles, nous expliquons qu'une sémantique d'adaptation est une sémantique d'exécution spécifique. En effet, la sémantique d'exécution définit le fonctionnement nominal, c'est-à-dire lorsque tout se passe comme prévu alors que de son côté, la sémantique d'adaptation s'intéresse aux cas anormaux ou qui posent problème.

Par exemple, pour nos machines à états, la sémantique d'exécution consiste à définir comment suivre les transitions associées aux états actifs courants et à un événement généré par l'environnement. La sémantique d'adaptation quant à elle a ici pour but de gérer les cas où l'on rencontre un événement inconnu : dans un premier temps, il faut déterminer que cet événement est inconnu et dans un deuxième temps, il faut appliquer une action d'adaptation.

En termes de vérification, les contrats d'exécution permettent de s'assurer que l'exécution se déroule normalement. Dit autrement, ils s'assurent que le moteur d'exécution fonctionne correctement et qu'il a bien implémenté la sémantique d'exécution attendue, ici qu'une transition qui doit être suivie l'a bien été. Les contrats d'adaptation que nous définissons dans cet article sont différents et auront pour principal objectif de définir s'il y a besoin ou non d'adapter l'exécution du modèle et si cette

adaptation s'est bien déroulée. Pour nos machines à états, ils permettront notamment de s'assurer qu'un événement rencontré est bien connu et géré par la machine à états, c'est-à-dire que le modèle est en adéquation avec l'environnement courant d'exécution.

4.3. Vérification statique

Pour notre exemple de machines à états, il est possible de faire un ensemble de vérifications statiques, c'est-à-dire en dehors de l'exécution du modèle. Leur but principal est de s'assurer qu'un modèle est valide pour un certain environnement d'exécution (cet environnement étant représenté par l'ensemble des événements attendus) et si le modèle gère correctement cet environnement. L'idée est de pouvoir déterminer à l'avance, avant d'exécuter un modèle, si tel ou tel modèle est utilisable pour tel ou tel environnement, éventuellement en mode dégradé. Nous détaillons ici quelques exemples de ces contrats statiques. Techniquement, ces contrats sont implémentés par des invariants OCL qui seront vérifiés sur le modèle.

4.3.1. Vérification statique du traitement des événements

La figure 8a présente l'invariant OCL qui vérifie que la machine à états définit une transition partant de chaque état (ou d'un de ses super-états) pour chacun des événements attendus de l'environnement. Cet invariant `existsTransitionFor-AllExpectedEvents` est défini à la ligne 20. Les événements attendus forment l'ensemble `expectedEvents` (ligne 2), contenant ici les couleurs rouge, orange et verte. Pour notre exemple de machine à états (figure 6), tous les états de la variante (b) respectent cet invariant mais pas ceux de la variante (a), comme expliqué précédemment.

Notons que seul le contenu de l'ensemble `expectedEvents` est spécifique au modèle (celui de notre train et de son environnement d'exécution) alors que tout le reste de ce qui forme l'invariant est générique (c'est-à-dire défini uniquement par rapport au métamodèle de machines à états) et peut s'appliquer sur n'importe quelle machine à états, qu'elle modélise un train ou quelque chose d'autre. Par conséquent, pour une vérification concernant un autre modèle ou un autre environnement, il suffira juste de changer le contenu de `expectedEvents`.

L'invariant assurant qu'un type d'événement est géré pour chaque état suivra la même logique et la même structure : il suffit de remplacer la vérification de la valeur de l'attribut `name` par celle de l'attribut `elementKind`.

4.3.2. Gestion correcte des événements

Comme expliqué dans la section précédente, un choix de gestion correcte des événements consiste dans notre cas à s'assurer qu'une occurrence d'un certain événement active l'état associé attendu dans la machine à états. Les contrats réalisant ce genre de vérification peuvent être implémentés comme des extensions des contrats de vérification de traitement d'événements (comme celui présenté dans la section 4.3.1) : en

```
-- l'ensemble fini des couleurs de signaux
context State def: expectedEvents : Set(String) = Set{'Red', 'Amber', 'Green'}

-- renvoie la machine à états unique dans le modèle
context State def: theSM : StateMachine = StateMachine.allInstances() -> first()

-- vérifie que pour une couleur, il existe une transition partant de l'état courant
-- ou d'un de ses super-états
context State def: existsTransitionFor(eventName : String) : Boolean =
self.theSM.transitions -> exists ( t |
   if (t.event.name = eventName and t.source = self) then true
   else
      -- pas de container : on a atteint le sommet de la hiérarchie sans trouver
      -- de transition
      if self.container.oclIsUndefined() then false
      else self.container.existsTransitionFor(eventName)
      endif
   endif )

context State inv existsTransitionForAllExpectedEvents:
self.oclIsTypeOf(State) or self.oclIsTypeOf(CompositeState) implies
self.expectedEvents -> forAll( evt | self.existsTransitionFor(evt))
```

(a) Invariant vérifiant le traitement des événements

```
-- un état ou un de ses super-états est d'un certain type
context State def: isGeneralElementKind(eltKind : String) : Boolean =
if (self.elementKind = eltKind) then true
else
   if self.container.oclIsUndefined() then false
   else self.container.isGeneralElementKind(eltKind)
   endif
endif

-- un état ou un de ses super-états a une transition associée avec le type de
-- l'événement et mène à un état du même type que l'événement
context State def: existsTransitionFor(event : Event) : Boolean =
self.theSM.transitions -> exists ( t |
   if (t.event.elementKind = event.elementKind)
      and (t.source = self)
      and (t.target.isGeneralElementKind(event.elementKind))
   then true
   else if self.container.oclIsUndefined() then false
        else self.container.existsTransitionFor(event)
        endif
   endif )
```

(b) Une partie de la vérification des types d'événements et des états cible

```
-- vérifie l'égalité de deux propriétés
context Property def: isEquals(p : Property) : Boolean =
   self.name = p.name and
   if (self.oclTypeOf(IntegerProperty) and p.oclTypeOf(IntegerProperty))
   then self.oclAsType(IntegerProperty).value = p.oclAsType(IntegerProperty).value
   else false
   endif

-- un élément possède au moins les mêmes propriétés que celui en paramètre
context AdaptableElement def: hasAtLeastSamePropertiesOf(elt : AdaptableElement)
                                                                          : Boolean =
elt.properties -> forAll ( p |
    self.properties -> exists ( pSelf | p.isEquals(pSelf))
```

(c) Vérification d'égalité de propriétés

Figure 8. Vérification statique du traitement de chaque événement par chaque état

plus de vérifier que l'événement est connu, on vérifiera qu'il est correctement géré. Pour une vérification lâche des états et des événements, il s'agit de vérifier qu'un type donné d'événement conduit à un état de même type. La figure 8b détaille certaines modifications sur la partie (a) pour implémenter cette vérification.

Pour une vérification exacte des événements et des états, il s'agit par contre de vérifier que les propriétés d'un événement sont identiques ou compatibles avec celles de l'état qui est systématiquement la cible des transitions associées avec cet événement. Cette comparaison de propriétés est réalisée de manière automatique indépendamment des propriétés mais pour l'exemple de notre train, cela reviendra en pratique à s'assurer que la propriété `speed` est présente avec la même valeur pour un état et l'événement menant à cet état. La figure 8c présente les fonctions OCL réalisant ces comparaisons. La fonction `hasAtLeastSamePropertiesOf` (ligne 10) vérifie que l'élément courant possède au moins les mêmes propriétés que l'élément passé en paramètre (ici, on s'en servira pour vérifier qu'un état possède les mêmes propriétés que celle d'un événement). Pour cela, elle s'appuie sur la fonction `isEquals` qui vérifie l'égalité de nom et de valeur de propriétés dans le cadre des valeurs entières.

4.4. Vérification dynamique et adaptation pendant l'exécution

La vérification dynamique consiste à ajouter des pré et post-conditions aux opérations du moteur d'exécution. Ces pré et post-conditions feront des vérifications pendant l'exécution du modèle pour s'assurer que le modèle est dans un état d'exécution correct ou si déclencher une action d'adaptation est nécessaire. Sur l'exemple de nos machines à états, nous décrivons le schéma général d'implémentation des contrats en dynamique puis nous caractérisons l'adaptation que nous réalisons avec des exemples à l'appui.

4.4.1. Principes d'implémentation

Le métamodèle de machines à états et les règles de bonne formation associées ont été implémentés dans l'environnement EMF [3] en Ecore et OCL. Nous avons implémenté un moteur d'exécution et d'adaptation dédié à nos machines à états en Kermeta [4]. Kermeta, outre ses capacités de manipulation de modèles, intègre une approche orientée contrat. Il est possible de définir des invariants sur les éléments d'un métamodèle ainsi que des pré et des post-conditions sur les opérations manipulant les modèles. Le choix particulier de ces langages (Kermeta et OCL) n'est pas important car l'implémentation des contrats peut être réalisée de la même façon avec d'autres langages de contraintes ou de programmation.

Concernant la vérification pendant l'exécution, nous nous basons sur le schéma qui suit. L'exécution d'un modèle est réalisée à travers une séquence de pas d'exécution. Chacun de ces pas correspond à l'appel d'une opération d'exécution. Par exemple,

3. `http://www.eclipse.org/modeling/emf/`
4. `http://www.kermeta.org`

pour notre métamodèle de machines à états, nous aurons l'opération `runToCompletion(Event)` qui traite l'occurrence d'un événement et fait évoluer le modèle en suivant les transitions si cela est requis par rapport à l'événement et les états actifs courants. Cette opération sera attachée à un méta-élément particulier du métamodèle, ici `StateMachine`. Cet attachement peut se faire en ajoutant virtuellement (en « tissant ») cette méthode à l'élément `StateMachine` grâce aux mécanismes d'aspects de Kermeta. Ainsi, on peut facilement définir un comportement associé à un métamodèle structurel sans pour autant modifier ce métamodèle directement. Ces opérations d'exécution sont enrichies par des pré-conditions. Ce sont elles qui intégreront les vérifications d'adaptation à l'environnement d'exécution. Si une pré-condition est violée, alors l'appel de l'opération d'exécution n'est pas réalisée et on exécutera à la place une action d'adaptation. Celle-ci pourra par exemple relancer la vérification en mode lâche si on était en mode exact ou bien encore modifier le modèle en fonction de l'événement inconnu.

```
aspect class StateMachine {
   operation runToCompletion(event : Event)
   pre existingTransition is do
       // vérifier qu'il existe une transition partant des états actifs courants
   end
   is do
      // corps de l'opération : suivre la transition trouvée en activant l'état
      // cible et en modifiant en conséquence la hiérarchie d'états actifs
   end
}
```

```
(...)
do
   // traite l'occurrence d'un événement
   sm.run_to_completion(event)
rescue (err : ConstraintViolatedPre)
   // la pré-condition est violée : on doit prendre une décision d'adaptation
   sm.adaptationAction(event)
end
(...)
```

Figure 9. Schéma général d'intégration des contrats d'adaptation en Kermeta

La figure 9 donne le schéma général de cette mise en œuvre en Kermeta pour l'opération `runToCompletion(Event)` de notre métamodèle de machines à états. Dans la partie du haut, cette opération est définie. On voit qu'elle a été rajoutée par aspect à la méta-classe `StateMachine` et qu'elle contient une pré-condition nommée `existingTransition`. C'est cette dernière qui vérifie, selon le niveau choisi (exact ou lâche), s'il existe une transition partant des états actifs courants pour l'événement passé en paramètre de l'opération. Si une telle transition n'existe pas, alors vu la construction de notre modèle, cela implique que l'événement n'est pas géré par la machine à états. Dans la partie du bas de la figure, on observe un appel de cette opération sur l'objet `sm` qui représente ici la machine à états exécutée. Cet objet est une instance de la méta-classe `StateMachine`. En cas de non respect de la pré-condition, une exception est levée. Le traitement de cette exception dans la clause Kermeta `rescue` va consister à prendre une décision d'adaptation via l'appel de l'opération `adaptationAction(Event)`.

Dans cette implémentation par contrat, nous n'avons parlé que des pré-conditions pour intégrer l'adaptation. Il est bien entendu possible d'utiliser également des post-conditions. Elles auront pour but de s'assurer qu'une opération s'est bien déroulée. S'il s'agit d'une opération d'exécution, la post-condition vérifiera que le modèle est dans un état correct après le pas d'exécution réalisé, sinon, il faudra déclencher une action d'adaptation. S'il s'agit d'une opération ou action d'adaptation, la post-condition vérifiera que l'adaptation s'est correctement déroulée.

4.4.2. Événement inattendu à l'exécution

Si un événement est inattendu à l'exécution, c'est-à-dire si la pré-condition de la méthode `runToCompletion(Event)` est violée, alors le moteur va exécuter une action d'adaptation. Cette action peut, entre autres, réaliser une des choses suivantes :

– Stopper l'exécution de la machine à états car on ne sait pas traiter l'événement;

– Charger un modèle de référence adapté au nouveau contexte d'exécution rencontré;

– Si on était dans un mode de vérification exact, on peut basculer dans un mode de vérification lâche et regarder si dans ce cas l'événement est géré ou pas;

– Modifier la machine à états pour prendre en compte l'événement inconnu.

À titre d'exemple, considérons le cas suivant : on exécute la machine à états du train de A dans la version (c) de la figure 6 et un signal mauve est croisé. L'événement associé à ce signal viole la pré-condition dans un mode de vérification exact mais pas dans un mode de vérification lâche. En effet, cet événement est défini comme un événement de type normal et une transition associée à l'événement vert sera trouvée et considérée comme valide car cet événement est également du type normal, tout comme le mauve. On voit bien avec cet exemple l'intérêt pendant l'exécution de pouvoir modifier la politique d'adaptation, ici le mode de vérification de l'adaptation à l'environnement.

Maintenant, supposons que la machine à états exécutée est toujours celle de la version (c) de la figure 6 et que le train, circulant sur les voies de B, croise un signal blanc. Quel que soit le mode de vérification, l'événement blanc est inconnu pour le train. Une action d'adaptation peut alors consister à modifier la machine à états pour prendre en compte le nouvel événement. Cela est réalisable si l'événement vient avec des informations supplémentaires, comme son type et ses propriétés. Ici, on supposera que l'événement blanc correspond à une vitesse réduite de 70 km/h. Il sera alors associé avec le type `reduced` et une propriété `speed` valant 70. L'action d'adaptation que nous avons implémentée pour adapter le modèle réalise le traitement qui suit. Dans un premier temps, on recherche sur le modèle s'il n'existe pas un état ayant le même type et les mêmes propriétés que l'événement. S'il est trouvé, alors, grâce au fait qu'un événement donné mène toujours au même état, on considère que cet état est l'état cible de ce nouvel événement. On rajoute alors des transitions associées à cet événement partant de tous les états (ou de leurs composites) et menant à cet état.

En revanche, si un tel état n'est pas trouvé, alors on ajoute dans la machine à états un tout nouvel état dédié ayant le même type et les mêmes propriétés que l'événement. On rajoute ensuite les transitions associées à cet événement partant de tous les états (ou de leurs composites) et menant à ce nouvel état. On rajoute également, en se basant sur les transitions existantes, des transitions partant du nouvel état, associées à chaque événement existant et menant à l'état de chaque événement. Dans le cas de notre événement blanc, il n'existe aucun état compatible avec le type réduit et la vitesse de 70 km/h. Comme le montre la variante (d) de la figure 6, un nouvel état 70 a donc été ajouté avec toutes les transitions associées menant à cet état ou en partant. L'événement blanc avec son type et ses propriétés a également été ajouté à la liste des événements gérés par la machine à états. Une fois le modèle modifié, on peut vérifier que la modification s'est faite correctement. Pour cela, on pourra intégrer les invariants détaillés en section 4.3 dans la post-condition de l'opération réalisant l'adaptation.

Notons que cette modification du modèle se fait de manière totalement automatique et générique par le moteur, il n'a aucunement besoin de savoir ce que représente le modèle (un train ici) ni les propriétés des événements (une vitesse ici). Cela est rendu possible par le fait que les machines à états sont contraintes, à la fois parce que se sont des automates complets pour expliciter les interactions avec l'environnement (pour tout événement connu, chaque état doit définir une transition l'ayant comme point de départ) et par la spécialisation définie en section 3.1.2 précisant qu'un événement donné mène toujours au même état cible. C'est grâce à ces contraintes que l'on peut modifier automatiquement une machine à états sans avoir aucunement savoir ce qu'elle représente. Sinon, il faudrait définir des adaptations dédiées à un domaine métier particulier (le comportement d'un train par exemple), de manière ad-hoc et donc non utilisables dans d'autres contextes. Dans la même logique, notons également qu'au lieu de passer par une propriété valuée, on aurait pu définir directement dans les méta-éléments définissant un état ou un événement un attribut nommé `speed` et de type entier. Il aurait alors représenté directement une vitesse. Si cela aurait simplifié la définition des modèles, d'un autre coté, cela aurait restreint bien plus les possibilités d'adaptation puisqu'on aurait explicitement vérifié la valeur de cet attribut. L'adaptation n'aurait donc été possible que pour des machines à états définissant des comportements impliquant une vitesse.

Dans (Cariou *et al.*, 2013), nous avons expliqué que contrairement à l'exécution qui ne modifie que la partie dynamique d'un modèle (ici les états actifs courants mais sans modifier structurellement les états ou les transitions), l'adaptation peut modifier toutes les parties du modèle ainsi que les sémantiques d'exécution et d'adaptation associées. Ici, les deux choix d'actions d'adaptation que nous avons détaillées modifient soit la sémantique d'adaptation utilisée par le moteur en substituant un mode de vérification par un autre, soit le contenu structurel du modèle correspondant aux éléments de la partie statique du métamodèle.

4.5. Caractérisation de l'adaptation de machines à états

Dans (Barbier *et al.*, 2015), nous avons décrit un ensemble de critères qui permettent de caractériser une adaptation. Voici la caractérisation de l'adaptation des machines à états que nous avons décrite dans cet article :

Fonctionnelle *vs* non-fonctionnelle (qualité de service) : les deux cas sont traités puisque l'ajout d'états et de transitions permet d'assurer la continuité de fonctionnement d'une machine à états dans un autre contexte d'exécution mais en même temps nous pouvons basculer dans un mode dégradé relevant là plus d'une problématique de qualité de service.

Prévue à l'avance *vs* non-anticipée : cette caractéristique se rapporte au fait que les adaptations sont ou pas « pré-câblées » dans le système dès le départ. Ici, ça n'est pas le cas puisque la modification d'un modèle pendant son exécution dépendra des interactions avec l'environnement et que ces interactions ne sont pas forcément connues à l'avance. Nous sommes donc dans un cas d'adaptation non-anticipée.

Prédictible *vs* non-déterministe : l'adaptation est prédictible dans le sens où la modification de la machine à états suit un algorithme bien précis et répétable.

Auto-adaptation *vs* par un tiers : nous faisons ici de l'auto-adaptation puisque le moteur adapte lui-même et seul l'exécution en cours, sans l'aide d'une tierce entité.

Générique *vs* métier : comme nous l'avons expliqué, l'adaptation est ici générique car étant totalement indépendante du contenu métier du modèle.

5. Travaux connexes

Il existe à notre connaissance très peu de travaux s'étant intéressés à l'adaptation d'exécution de modèles dans un contexte d'ingénierie des modèles. Comme nous l'avons expliqué en début d'article, on peut appliquer deux méthodes principales : soit directement adapter le modèle exécuté comme cela est fait dans cet article, soit appliquer des méthodes existantes d'adaptation d'un système comme par exemple l'adaptation par les *models@run.time*.

L'adaptation directe de l'exécution de modèle a été étudiée dans nos précédents articles (Cariou, Graiet, 2012 ; Cariou *et al.*, 2012 ; 2013 ; Pierre *et al.*, 2014). Concernant l'application des *models@run.time*, nous pouvons citer un autre de nos précédents articles qui présente MOCAS (Ballagny *et al.*, 2009). MOCAS est une plateforme permettant l'adaptation d'une machine à états UML en cours d'exécution. Cette machine à états est observée par une autre machine à états qui a la charge de gérer l'adaptation de la première. Conceptuellement, la première représente la partie métier de l'application et la seconde est un agent chargé de l'adaptation de la partie métier. Ainsi, nous avons avec MOCAS une plateforme permettant à la fois d'exécuter une machine à états et de l'adapter via un modèle l'observant dans l'esprit des principes

des *models@run.time* où un système est représenté par un modèle. La limite notable de cette approche est que le système et le modèle ne sont pas bien dissociés. La logique métier et la logique d'adaptation via les deux machines à états restent entremêlées.

Bien que ne concernant pas directement le contexte de l'ingénierie des modèles, nous pouvons également citer un domaine qui présente des liens avec l'adaptation d'exécution de modèles. Il s'agit de la définition de processus qui sont interprétés à l'exécution et dont le but est d'ordonnancer ou orchestrer un ensemble de fonctionnalités ou services métier. On retrouve ici les principes d'un i-DSML dont les modèles exécutables sont interprétés par un moteur d'exécution. Des travaux ont été menés pour adapter ces processus pendant l'exécution, par exemple dans le domaine des workflows (Rinderle *et al.*, 2004) ou bien encore de l'orchestration ou la chorégraphie de services Web (Karastoyanova *et al.*, 2005 ; Mosincat, Binder, 2008 ; Leonardo *et al.*, 2013). L'utilité de l'adaptation peut être de remplacer un service par un service équivalent pour des raisons de qualité de service ou d'erreurs lors d'appel d'un service. La particularité de ces travaux est de se concentrer uniquement sur l'adaptation pour un contexte et des buts d'adaptation précis. Dans cet article, nous proposons au contraire un cadre général que l'on peut appliquer à tout i-DSML. Néanmoins, ces travaux montrent des exemples concrets d'adaptation pour des processus équivalents à nos modèles exécutables. Ils pourraient donc servir de base à la définition d'i-DSML adaptables, comme cela a été le cas pour notre exemple de processus de la section 2.2.

6. Conclusion et perspectives

Nous avons présenté dans cet article une approche d'adaptation d'exécution de modèles par contrats. Son principe fondamental est d'adapter directement le modèle en cours d'exécution sans passer par une représentation externe de ce qui est nécessaire à l'adaptation. Cela nécessite de rajouter dans le modèle des informations dédiées à l'adaptation afin qu'il soit auto-contenu. Un métamodèle définissant des modèles à la fois exécutables et adaptables est alors appelé un i-DSML (pour *interpreted Domain-Specific Modeling Language*) adaptable. L'implémentation de l'adaptation se fait en étendant les opérations d'exécution du moteur d'exécution par des contrats et notamment des pré-conditions. Celles-ci ont pour objectif de vérifier que l'exécution du modèle se déroule correctement et si ça n'est pas le cas, de déclencher des opérations d'adaptation.

Nous avons appliqué notre approche sur un cas d'étude se basant sur un i-DSML de machines à états inspirées de celles d'UML et avec un exemple illustratif de signalisation ferroviaire et de comportement de train. Le but des contrats ici est de s'assurer que les interactions avec l'environnement d'exécution sont bien gérées par le modèle. Si ça n'est pas le cas, des actions d'adaptation doivent être réalisées comme charger un nouveau modèle ou bien modifier le modèle en cours d'exécution grâce à des propriétés dédiées à l'adaptation. Cette modification du modèle se fait de manière totalement générique, sans avoir besoin de savoir ce que représente le modèle (un train ou autre chose). Il faut ici expliciter en intégralité la liste des interactions attendues avec l'envi-

ronnement d'exécution afin de pouvoir détecter si une interaction est connue ou pas. Il est également possible de définir une adéquation avec un environnement d'exécution en mode dégradé avec des vérifications lâches. Avec l'explicitation des interactions, on peut également, de manière statique avant même d'exécuter un modèle, déterminer si celui-ci est adapté à un certain environnement d'exécution. Le moteur d'exécution et d'adaptation pour ce i-DSML adaptable a été implémenté en Kermeta. Les contrats ont été intégrés dans le moteur ou écrits en OCL en ce qui concerne les vérifications statiques.

Les perspectives autour des i-DSML adaptables sont multiples. Nous pouvons par exemple citer les suivantes. Il faudrait étudier du point de vue adaptation d'autres i-DSML et des cas d'études plus complexes. Nous avons notamment pour objectif de travailler avec des machines à états UML standard et complètes. Nous étendrons alors nos outils existants pour y intégrer les politiques d'adaptation définies. SimUML [5] est un outil de simulation de machines à états UML tandis que PauWare Engine [6] est une bibliothèque Java permettant d'exécuter des machines à états UML sur n'importe quel support Java, y compris le système Android.

L'adaptation se fait concrètement dans le moteur en appelant des opérations d'adaptation en cas de non-respect des contrats au niveau des opérations d'exécution. Cette manière d'entrelacer ou d'orchestrer des opérations d'adaptation et d'exécution pourrait gagner à être définie à un plus haut niveau d'abstraction que d'être « codée en dur » dans un moteur. Pour cela, nous allons définir un i-DSML ayant pour objectif de décrire l'orchestration et le tissage des opérations d'exécution avec les vérifications et les actions d'adaptation. Ce i-DSML devra permettre également de modifier la politique d'adaptation pendant l'exécution (Cariou *et al.*, 2013). Par exemple, nous avons vu que l'on pouvait passer d'un mode de vérification exact à un mode de vérification lâche, ce qui est un changement dans la politique d'adaptation. L'idée clé ici est que deux modèles seront exécutés : d'un côté celui du i-DSLM considéré (comme nos machines à état de train) et de l'autre, le modèle d'orchestration défini via cet autre i-DSML et qui du coup pourra se modifier lui-même à l'exécution. Ce modèle d'orchestration est donc lui aussi écrit dans un i-DSML adaptable puisqu'il est interprété et modifiable pendant l'exécution. La définition de ce i-DSLM adaptable pourra se baser sur les travaux d'adaptation d'orchestration de services dont nous avons parlé dans la section des travaux connexes.

Bibliographie

Ballagny C., Hameurlain N., Barbier F. (2009). MOCAS: A State-Based Component Model for Self-Adaptation. In *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '09)*. IEEE Computer Society.

5. http://sourceforge.net/projects/simuml/

6. http://www.pauware.com/

Barbier F., Cariou E., Goaer O. L., Pierre S. (2015). Software Adaptation: Classification and a Case Study with State Chart XML. *IEEE Software*, vol. 32, n° 5, p. 68-76.

Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D. (1999). Making Components Contract Aware. *IEEE Computer*, vol. 32, n° 7, p. 38–45.

Blair G. S., Bencomo N., France R. B. (2009). Models@run.time. *IEEE Computer*, vol. 42, n° 10, p. 22-27.

Breton E., Bézivin J. (2001). Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)*. ACM.

Cariou E., Ballagny C., Feugas A., Barbier F. (2011). Contracts for Model Execution Verification. In *Seventh European Conference on Modelling Foundations and Applications (ECMFA '11)*, vol. 6698 of LNCS, p. 3–18. Springer.

Cariou E., Goaer O. L., Barbier F., Pierre S. (2013). Characterization of Adaptable Interpreted-DSML. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*, vol. 7949 of LNCS, p. 37–53. Springer.

Cariou E., Graiet M. (2012). Contrats pour la vérification d'adaptation d'exécution de modèles. In *1ère Conférence en IngénieriE du Logiciel (CIEL 2012)*.

Cariou E., Le Goaer O., Barbier F. (2012). Model Execution Adaptation? In *7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012*. ACM Digital Library.

Cariou E., Marvie R., Seinturier L., Duchien L. (2004). *OCL for the Specification of Model Transformation Contracts*. Workshop OCL and Model Driven Engineering, UML 2004.

Clarke P. J., Wu Y., Allen A. A., Hernandez F., Allison M., France R. (2013). Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. In M. Mernik (Ed.), chap. 9: Towards Dynamic Semantics for Synthesizing Interpreted DSMLs. IGI Global.

Combemale B., Crégut X., Pantel M. (2012). A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. IEEE.

Fleurey F., Solberg A. (2009). A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, vol. 5795 of LNCS. Springer.

Floch J., Hallsteinsen S., Stav E., Eliassen F., Lund K., Gjørven E. (2006). Using Architecture Models for Runtime Adaptability. *IEEE Software*, vol. 23, n° 2, p. 62-70.

Guy C., Combemale B., Derrien S., Steel J., Jézéquel J.-M. (2012). On Model Subtyping. In *8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, vol. 7349 of LNCS, p. 400–415. Springer.

IBM. (2005). *An architectural blueprint for autonomic computing, third edition*. Rapport technique. International Business Machines (IBM) corporation.

Karastoyanova D., Houspanossian A., Cilia M., Leymann F., Buchmann A. P. (2005). Extending BPEL for Run Time Adaptability. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, p. 15–26. IEEE Computer Society.

Kephart J., Chess D. (2003). The vision of autonomic computing. *IEEE Computer*, vol. 36, n° 1, p. 41-50.

Lehmann G., Blumendorf M., Trollmann F., Albayrak S. (2010). Meta-Modeling Runtime Models. In *Models@run.time Workshop at MoDELS 2010*, vol. 6627 of LNCS. Springer.

Leonardo L., Oliva G., Nogueira G., Gerosa M. A., Kon F., Milojicic D. (2013). A Systematic Literature Review of Service Choreography Adaptation. *Service Oriented Computing and Applications*, vol. 7, n° 3, p. 199–216.

Le Traon Y., Baudry B., Jézéquel J.-M. (2006). Design by Contract to improve Software Vigilance. *IEEE Transaction on Software Engineering*, vol. 32, n° 8.

Meyer B. (1992). Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, n° 10, p. 40–52.

Morin B., Barais O., Jézéquel J.-M., Fleurey F., Solberg A. (2009). Models@Run.time to Support Dynamic Adaptation. *IEEE Computer*, vol. 42, n° 10, p. 44-51.

Mosincat A. D., Binder W. (2008). Transparent Runtime Adaptability for BPEL Processes. In *6th International Conference on Service Oriented Computing (ICSOC 2008)*, vol. 5364 of LNCS, p. 241–255. Springer.

Pierre S., Cariou E., Goaer O. L., Barbier F. (2014). A Family-based Framework for i-DSML Adaptation. In *10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*, vol. 8569, p. 164–179. Springer.

Rinderle S., Reichert M., Dadam P. (2004). Correctness criteria for dynamic changes in workflow systems—a survey. *Data & Knowledge Engineering*, vol. 50, n° 1, p. 9–34.

Salehie M., Tahvildari L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, vol. 4, p. 14:1–14:42.

Steel J., Jézéquel J.-M. (2007). On model typing. *Software and System Modeling*, vol. 6, n° 4, p. 401-413.

Vogel T., Glese H. (2011). Language and Framework Requirements for Adaptation Models. In *Models@run.time Workshop at MODELS 2011*.

Wuliang S., Combemale B., Derrien S., France R. (2013). Contract-Aware Substitutability of Modeling Languages. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*, vol. 7949 of LNCS. Springer.

Zhang J., Cheng B. H. C. (2006). Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE 2006)*. ACM.

Reçu le 29/09/2013
Accepté le 29/06/2015