Contrats pour la vérification d'adaptation d'exécution de modèles

Eric Cariou
Université de Pau et des Pays de l'Adour / LIUPPA
B.P. 1155, 64013 Pau Cedex, France
Eric.Cariou@univ-pau.fr

Mohamed Graiet
Institut Supérieur d'Informatique et de Mathématiques de Monastir
Monastir, Tunisie
Mohamed.Graiet@imag.fr

Abstract

One of the main goals of model-driven engineering is the manipulation of models as exclusive software artifacts. Model execution is in particular a means to substitute models for code. MDE is a promising discipline for building adaptable systems thanks to models at runtime. When the model at runtime is directly executed, the system becomes the model, then, this is the model that is adapted. We propose an approach based on previous execution contracts for defining adaptation contracts. Their goal is to verify that an executed model is valid or has an acceptable failsoft behavior accordingly to an execution environment. If not, the model has to be adapted. Based on some basic modeling rules and the ability to have only a partial knowledge on the model specification and on the execution environment, we can define a hierarchy of contrats, from strongest to weakest. We apply our approach on simplified state machines and using OCL for expressing the contracts.

Keywords: MDE, model execution, adaptation, verification, contracts, state machines, OCL

Résumé

Dans le contexte de l'IDM, l'exécution de modèles est un des moyens principaux pour supprimer le fossé entre le le code, c'est-à-dire le système développé, et le modèle. D'un autre côté, l'IDM ouvre des perspectives intéressantes pour l'adaptation logicielle avec la notion de *models@run.time*. Les modèles utilisés à l'exécution permettent de représenter l'état du système adaptable et de travailler sur ce modèle pour gérer l'adaptation du système. Quand on exécute un modèle, le système *est* le modèle et alors l'adaptation est réalisée directement sur le modèle. Dans cet article, nous étudions la notion de contrats d'adaptation qui sont basés sur nos contrats d'exécution. Ils ont pour but de vérifier si un modèle exécuté est adapté à un contexte d'exécution donné. A partir de quelques règles simples de modélisation ainsi que la possibilité de définir une connaissance limitée sur le comportement du modèle ou sur l'environnement d'exécution, nous pouvons définir des hiérarchies de contrats, des plus contraignants jusqu'à la validité de versions dégradées de modèles. Nous appliquons notre approche sur des machines à états UML simplifiées et en utilisant OCL pour exprimer les contrats.

Mots-clé : IDM, exécution de modèles, adaptation, vérification, contrats, machines à états, OCL

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. Cela peut être

réalisé par la génération de code à partir d'un modèle mais également par directement exécuter un modèle. D'un autre côté, l'IDM offre un intérêt particulier pour le développement de systèmes logiciels adaptatifs [6, 7, 15]. Cela est notamment dû à la possibilité de disposer de modèles à l'exécution (*models@run.time*) [2]. De tels modèles ont principalement pour but de représenter l'état du système adaptable et de raisonner sur la nécessité d'adaptation au niveau du modèle. Par exemple, [12] utilise des méta-modèles 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 (des modèles unifiés) toutes les informations requises pour l'adaptation. De plus, il permet également d'utiliser certains modèles définis à la conception directement à l'exécution. La contrepartie importante est qu'il faut assurer que les modèles représentent de manière cohérente et causale le système en cours d'exécution [2, 10, 14]. Dans un contexte d'exécution de modèles, le système *est* le modèle exécuté et donc, par principe, le modèle est la représentation exacte du système à l'exécution. Il n'est alors plus utile de mettre en œuvre des techniques relativement complexes pour assurer la cohérence entre le modèle et le système.

Dans cet article, nous investiguons une approche de vérification d'adaptation d'exécution de modèles en fonction de l'environnement d'exécution. Nous n'expliquons pas comment réaliser l'adaptation, c'est-à-dire comment adapter le modèle, mais nous déterminons s'il est nécessaire ou pas d'adapter un modèle exécuté selon un environnement d'exécution donné, ou, avant même d'exécuter le modèle, de savoir de manière statique si un modèle est adapté à un certain environnement d'exécution. Nous nous basons pour cela sur une approche par contrats reprenant nos contrats d'exécution de modèles [3]. A notre connaissance, il n'existe pas d'autres travaux s'étant intéressés à l'adaptation de l'exécution de modèles directement comme nous le faisons.

La section suivante présente le principe de nos contrats d'adaptation. La section 4 décrit un exemple concret pour une machine à états UML d'un train en détaillant plusieurs environnements d'exécution et les contraintes sur un modèle pour être adaptable. La section 4 décrit quelques contrats en utilisant OCL [13] comme support principal de leur écriture.

2 Contrats d'adaptation

La conception et la programmation par contrats [1, 9, 11] 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 s'assurer qu'ils fonctionnement correctement. Dans un contexte IDM, les invariants peuvent être les règles de bonne formation ou d'autres contraintes supplémentaires sur les éléments définissant un méta-modèle, et les opérations à spécifier sont les opérations de manipulation et de modification des modèles. Pour des opérations de transformations de modèles, on pourra alors définir des contrats de transformations [4, 5] et si ces opérations ont pour but de réaliser l'exécution d'un modèle, on définira des contrats d'exécution [3].

Nos contrats d'exécution [3] ont pour objectif de décrire une sémantique d'exécution dans un but de vérification. Globalement, l'idée est de s'assurer que le modèle est correctement exécuté, c'est-à-dire de vérifier que le moteur d'exécution fonctionne correctement. Un contrat d'adaptation sera un type particulier de contrat d'exécution : il vérifiera qu'une exécution est correcte, mais dans le sens où le modèle exécuté est cohérent par rapport à un environnement d'exécution donné. Pour donner un exemple, concernant une machine à états, lorsqu'un événement est à traiter, la vérification de l'exécution consiste à assurer que s'il existe une transition associée à cet événement et aux états actifs courants, alors cette transition est suivie.

D'un point de vue adaptation, on pourra vérifier par contre qu'il existe une transition pour cet événement afin d'assurer que tout événement (venant de et caractérisant l'environnement d'exécution) est traité et connu par la machine à états. La section suivante explicite ce problème avec notre exemple de train.

Dans [4], nous proposons une approche par contrats pour vérifier des transformations de modèles. Une exécution peut se ramener à une série de transformations de modèles : chaque pas d'exécution (par exemple suivre une transition en fonction de l'occurence d'un événement) fait évoluer le modèle, le modifie, c'est-à-dire, le transforme (par exemple pour une machine à états, les états actifs sont modifiés en suivant une transition). Ainsi, les contrats d'exécution de [3] et les contrats d'adaptation de cet article reprendront les principes de nos contrats de transformation. Un de ces principes est qu'un couple de pré/post-conditions peut-être exprimé sous la forme d'ensemble d'invariants. Cela est intéressant puisqu'une spécification par couple de pré et post-conditions est contraignante au niveau de leur évaluation : par principe, c'est le moteur d'exécution ou de transformation qui doit les vérifier et cela ne se fait donc qu'à l'exécution. Avec des invariants, il est également possible de les vérifier a posteriori, indépendamment du moteur, si ce dernier enregistre les modèles après chaque pas d'exécution.

3 Modèle exécuté et environnements associés

Dans cette section, nous détaillons les exigences requises sur un modèle exécuté pour pouvoir vérifier sa validité pour un contexte 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 décrivons le méta-modèle de machines à états simplifiées que nous utilisons pour décrire la machine à états du train. Dans [3], nous avions défini nos contrats d'exécution pour les machines à états UML standard complètes. Dans cet article, nous travaillons avec des machines à états simplifiées afin de rendre nos exemples plus simples et concis. Par rapport aux machines à états UML standard, nous nous restreignons aux fonctionnalités d'états composites, d'états historiques et ne considérons que des transitions simplement associées à un événement (sans garde). Ce nombre limité de fonctionnalités permet à la fois d'avoir un méta-modèle simple et de définir tout de même des exemples de machines à états non triviaux.

3.1 Machines à états simplifiées

Comme expliqué dans [3], un méta-modèle représentant des modèles exécutables doit contenir tous les éléments requis pour représenter l'état complet d'un modèle pendant son exécution. Pour cela il contient une partie définissant les éléments statiques ou structurels d'un modèle et une extension dynamique spécifiant l'état du modèle à l'exécution. Notre méta-modèle de machines à états simplifiées est représenté sur la figure 1[2]. Sa partie structurelle contient les éléments suivants :

- 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 *état historique* référençant chacun un état de l'état composite auquel ils appartiennent. Tout composite

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

[2] L'élément `Kind` qu'il contient est un élément particulier qui sera introduit dans la section suivante.

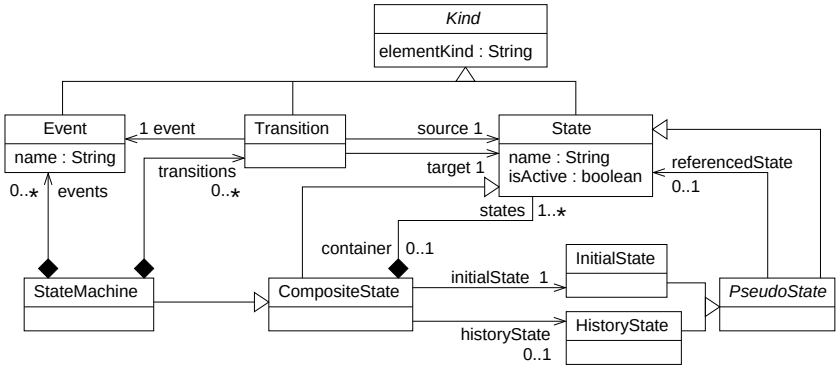


Figure 1: Méta-modèle de machines à états simplifiées

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 simplement 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 méta-modèle est simplement composée de deux éléments :

- L'attribut booléen *isActive* d'un état. Il précise si l'état est actif ou non.
- L'état référencé par un état historique qui référence le dernier état qui est (ou était) actif pour le composite dans lequel il est contenu.

Ces parties structurelles et dynamiques ne sont bien sûr pas suffisantes pour définir complètement le méta-modèle. Il faut en effet les compléter par des règles de bonne formation sous la forme d'invariants OCL. La figure 2 présente quelques-uns de ces invariants. Par exemple, l'invariant structurel `historyStatesInComposite` définit qu'un état référencé par un état historique (s'il existe) est un des états contenus dans l'état composite auquel cet état historique appartient. Un invariant de même nature précisera que l'état référencé par un état initial appartient au composite dans lequel est contenu cet état initial. L'invariant `containerForAllStatesExceptSM` spécifie lui que le seul type d'état composite sans container est une machine à états (car étant le sommet de la hiérarchie des états). Concernant la partie dynamique, l'invariant principal est `activeStateHierarchyConsistency` qui définit la cohérence de la hiérarchie d'états actifs : soit tous les états de la machine sont inactifs (la machine n'est pas en cours d'exécution), soit pour les états de la machine à états (au premier niveau de la hiérarchie), il y a un et un seul état actif, et si cet état est un composite alors lui aussi ne contient qu'un et un seul état actif et ainsi de suite jusqu'à la racine de la hiérarchie d'états.

3.2 Environnements d'exécution pour un train

La figure 3 représente trois types de signalisation ferroviaire. 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

```
- - Un état historique référence un état du composite auquel il appartient
context CompositeState inv historyStatesInComposite:
not self.historyState.oclIsUndefined() implies (
    self.states -> includes(self.historyState) and
    not self.historyState.referencedState.oclIsUndefined() implies
        self.states -> includes(self.historyState.referencedState) )

- - Une machine à états est le seul état à ne pas avoir de container
context State inv containerForAllStatesExceptSM:
if self.oclIsTypeOf(StateMachine)
then self.container.oclIsUndefined()
else not self.container.oclIsUndefined()
endif

- - Cohérence de la hiérarchie d'états actifs
context StateMachine inv activeStateHierarchyConsistency:
if self.isActive
then self.activeSubTree()
else self.unactiveSubTree()
endif

- - Vérifie qu'un état composite possède un et un seul état actif et si cet état est est aussi composite,
- - il contient lui aussi un seul état actif. Tous les autres états du composite sont inactifs.
context CompositeState def: activeSubTree() : Boolean =
let myStates = self.states -> reject ( s | s.oclIsTypeOf(PseudoState)) in
myStates -> select (s | s.isActive) -> size() = 1 and
myStates -> select (s | s.oclIsTypeOf(CompositeState)) -> forAll ( s |
    if s.isActive
    then s.oclAsType(CompositeState).activeSubTree()
    else s.oclAsType(CompositeState).unactiveSubTree()
    endif )

- - Vérifie que tous les états d'un composite sont inactifs
context CompositeState def: unactiveSubTree() : Boolean =
let myStates = self.states -> reject ( s | s.oclIsTypeOf(PseudoState)) in
myStates -> forAll ( s | not s.isActive) and
myStates -> select ( s | s.oclIsTypeOf(CompositeState)) -> forAll( s |
        s.oclAsType(CompositeState).unactiveSubTree())
```

Figure 2: Quelques invariants OCL associés au méta-modèle de machine à états

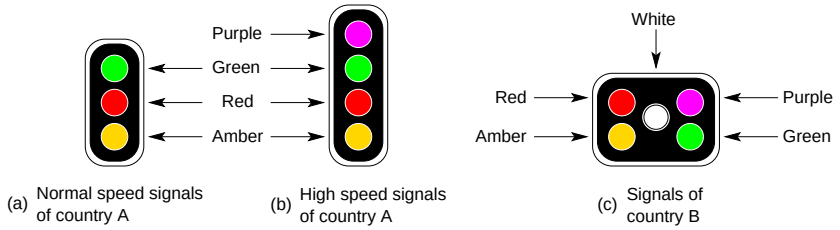
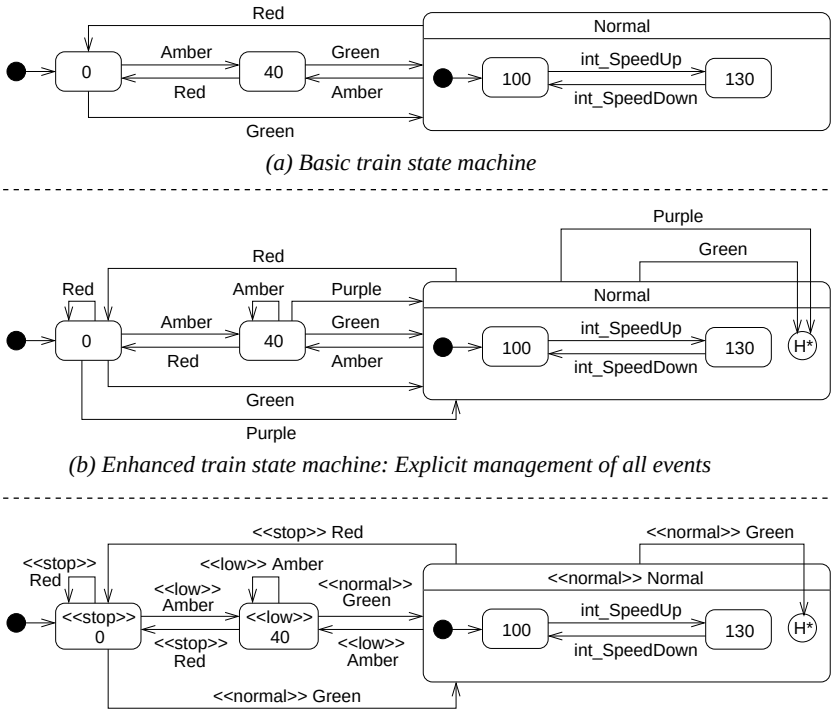


Figure 3: Trois types de signalisation ferroviaire

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. La couleur rouge correspond à un arrêt (0 km/h), la orange à une vitesse lente (30 km/h), la verte à une vitesse normale (jusqu'à 120 km/h) et la 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



(a) Basic train state machine

(b) Enhanced train state machine: Explicit management of all events

(c) Enhanced train state machine: Explicit management of all kinds of events

Figure 4: Variantes de machines à états pour un train

La figure 4, partie (a), 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 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.

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

La spécification de notre machine à états semble correcte mais en fait elle pose un problème car elle contient de l'information et des éléments implicites. 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. Bien entendu, il est évident qu'ignorer systématiquement un signal inconnu n'est pas un comportement adéquat. 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. L'idée est d'être capable de différencier une interaction non prévue d'une interaction connue. Par exemple, pour nos machines à états, nous pouvons appliquer deux solutions. 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'occurence d'un événement que ce dernier est bien dans cette liste. 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. Notons qu'il est possible de transformer facilement et automatiquement n'importe quelle machine à états afin qu'elle respecte cette structuration. L'intérêt est également que toute l'information requise (ici l'ensemble des interactions attendues) est directement embarquée dans le modèle. C'est ce second choix que nous appliquerons dans cet article.

La variante (b) de la figure 4 complète ainsi la machine à états précédente en ajoutant explicitement à partir de chaque état une transition pour chacune des 4 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[3]. 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.

Maintenant, 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 asssociée à une transition donc est explicitement gérée. Par contre, la couleur blanche inconnue ne corre-

[3]Si on considérait des machines à états UML standard, c'est-à-dire permettant d'associer des opérations à des états, il faudrait ajouter en plus un mécanisme permettant d'éviter qu'on appelle les opérations associées aux états pour les auto-transitions ajoutées lors de l'explicitation de toutes les interactions.

spond 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, soit arrêter l'exécution de la machine à états car elle se trouve dans une situation indéfinie et non prévue.

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, il est possible de faire, parmi d'autres, un choix de vérification simple qui consiste à s'assurer qu'un événement d'un certain type aboutit à activer un état du même type. Concrètement, 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 lente est une vitesse de 40 km/h. Pour le pays B par contre, c'est une vitesse de 30 km/h. Ces deux vitesses sont relativement proches mais pas identiques. 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 ce qui ne correspond pas exactement à la même vitesse mais tout de même à une vitesse suffisament proche pour être acceptable (en d'autres termes, pour être un comportement dégradé acceptable en terme de vitesse).

Nous proposons donc de rajouter une notion autour des événements et des états, celles de "type de". Des vitesses de 30 ou 40 km/h, suffisament proches pour être considérées comme similaires seront alors marquées comme de "type lent". Pouvoir définir des types d'éléments se fait grâce à l'extension basique du méta-modèle (figure 1) consistant à définir un élément `Kind` possédant un attribut `elementKind` de type chaîne, et de faire hériter tous les éléments du méta-modèle de cet élément. Au besoin, on pourra simplement ajouter les liens d'héritage sur certains éléments du méta-modèle, mais pas tous. Ceci dit, ajouter cet héritage sur tous les éléments peut être réalisé par une transformation automatique pour n'importe quel méta-modèle. 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 relâchée se basant sur des types d'éléments.

La figure 4, partie (c), 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 notion en `<<...>>` des stéréotypes UML). L'événement rouge est un événement de type arrêt, le 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. L'intérêt de cette spécification est que cette machine à états, si l'on se base sur des vérifications relâchées (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 (sauf si le train rencontre un 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 acceptable 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 (toujours à l'exception du la couleur blanche pour B).

3.6 Caractérisation d'un modèle adaptable pour de la vérification

En nous basant sur les discussions précédentes et en les généralisant au delà des machines à états de nos exemples, nous pouvons caractériser un modèle exécutable et adaptable de la façon 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 (chaque événement attendu est associé à une transition pour tout état par exemple).

Connaissance exacte ou limitée des éléments : un élément peut être défini avec un double niveau de précision, 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 connus ou certaines interactions peuvent être "découvertes" à l'exécution.

Si l'ensemble des interactions est fini et connu 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).

Concernant la connaissance requise sur un élément, elle est soit en mode exact, soit en mode limité (via un type d'élément) et comme nous avons d'un côté le comportement et de l'autre les interactions, cela donne au total 4 combinaisons possibles. Chaque combinaison définit un niveau de vérification, du plus complet (connaissance exacte du comportement et des interactions) au plus relâché (connaissance limitée via les types de comportement et d'interactions), formant ainsi une hiérarchie de niveaux de vérification, c'est-à-dire concrètement une hiérarchie de types de contrats. La table 1 décrit ces 4 combinaisons.

4 Exemples de contrats d'adaptation

Dans cette section, nous donnons des exemples concrets d'implémentation de contrats, principalement basés sur OCL, pour notre méta-modèle de machine à états et son exemple de train.

```
- - 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
contex 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 supers-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 associé avec l'événement exact ou son type
- - 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.name = event.name) or (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

Figure 5: Vérification statique pour le traitement de chaque événement par chaque état

	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.

Table 1: Quatre combinaisons de vérification d'adaptation

4.1 Vérification statique du traitement des événements

La figure 5, partie (a), présente l'invariant OCL `existsTransitionForAllExpectedEvents` qui assure qu'il existe une transition partant de chaque état (ou d'un de ses super-états) pour chacun des événements attendus. Ces événements forment l'ensemble `expectedEvents`, contenant ici les couleurs rouge, orange et verte. Pour notre exemple de machine à états (figure 4), 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 (défini uniquement par rapport au méta-modèle de machines à états) et peut s'appliquer sur n'importe quel modèle de train ou autre. Pour une vérification concernant un autre modèle ou un autre environnement, il suffira juste de changer le contenu de `expectedEvents`.

Cet invariant peut être statiquement vérifié sur n'importe quelle machine à états avant de l'exécuter afin de s'assurer que le modèle saura gérer toutes les interactions avec l'environnement d'exécution, c'est-à-dire s'assurer que le modèle est adapté à cet environnement d'exécution. Cet invariant peut aussi être intégré dans le moteur d'exécution. En effet, si on détecte qu'un événement n'est pas géré par la machine à états courante, on peut l'adapter (la modifier ou charger une machine à états de référence) pour prendre cet événement en compte en plus des événements déjà gérés. Dans la post-condition de l'opération réalisant cette adaptation, on pourra intégrer et vérifier les contraintes de cet invariant pour assurer que la machine à états est bien adaptée à la nouvelle liste d'événements à gérer.

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`.

4.2 Gestion correcte des événements

Comme expliqué dans la section précédente, un choix de gestion correcte des événements consiste à s'assurer qu'une occurence 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 de l'exemple précédent) : en plus de vérifier que l'événement est connu, on vérifiera qu'il est correctement géré. Pour une vérification relâchée des états et des événements, il s'agit de vérifier

qu'un type donné d'événement conduit au même type d'état (la figure 5 partie (b) détaille certains modifications sur la partie (a) pour implémenter cette vérification).

Pour une vérification exacte des événements et des états, il s'agit de s'assurer que le modèle est équivalent à un modèle de référence pour cet environnement d'exécution. Pour une transformation, nous appelons un contract de non modification un contrat qui vérifie que certaines parties du modèle ne sont pas modifiées [4]. Ce type de contrat peut être partiellement généré grâce à nos outils. Vérifier qu'un modèle est équivalent à un modèle de référence peut être réalisé en se basant sur les mêmes principes : sans rentrer ici dans des problématiques d'équivalence sémantique de machines à états, on pourra simplement vérifier que certaines parties des deux modèles sont structurellement identiques.

4.3 Evénement inattendu à l'exécution

Nous avons implémenté un moteur d'exécution de 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 méta-modèle ainsi que pré et des post-conditions sur les opérations manipulant les modèles. Dans notre moteur, l'opération traitant l'occurence d'un événement possède une pré-condition qui vérifie, selon le niveau désiré (exact ou relâché), qu'il existe bien une transition associée aux états actifs pour cet événement ou ce type d'événement. Si ce n'est pas le cas, alors cela signifie que la machine à états n'est pas adapté au contexte d'exécution courant. Il faut alors, au choix, modifier le modèle pour l'adapter, ou, comme nous le faisons dans notre implémentation, arrêter l'exécution du modèle car on s'est retrouvé dans une situation inattendue. Notons que par arrêt d'exécution, nous parlons bien de stopper l'exécution du moteur et non pas d'activer l'état 0 km/h de la machine à états du train de notre exemple. Cela est dû au fait que pour avoir un traitement générique de l'occurence d'un événement inconnu, on ne peut pas se baser sur le contenu d'un modèle, notamment ici savoir qu'il existe un état représentant un arrêt (ce qui ne sera pas le cas de toutes façons dans tous les modèles).

La figure 6 montre un extrait du code Kermeta de notre moteur. Dans la partie basse, le code appelant `newEvent` (l'opération traitant une occurence d'événement) gère une exception en cas de violation de la pré-condition de cette opération. `sm` représente la machine à états en cours d'exécution et `event` l'occurence de l'événement à traiter. Cette pré-condition s'assure qu'il existe bien une transition associée à cet événement partant d'un des états de la hiérarchie des états actifs (en commençant par le plus bas, retourné par l'appel de `getLeafState`) via l'appel de `getTriggerableTransition`. Cette opération recherche une transition associée à l'événement (via son nom ou son type selon le niveau de vérification) pour un état donné. Si la transition n'est pas trouvée, la même recherche est faite récursivement sur l'état composite contenant cet état, jusqu'à atteindre le sommet de la hiérarchie. Le niveau de vérification est défini par la variable `verificationLevel` pouvant prendre 3 valeurs : `none` (pas de vérification), `weak` (vérification du type de l'événement) ou `exact` (vérification du nom de l'événement). L'opération `getEventKind` retourne, à partir de la liste des événements de la machine à états, le type associé à un événement dont on connaît le nom.

Pour notre exemple de machine à états de train de la figure 4, dans sa variante (c), un événement mauve viole la précondition dans un mode de vérification exacte mais pas dans un mode de vérification relâchée car il 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

[4] `http://www.kermeta.org`

```
operation getEventKind(sm : StateMachine, eventName : String) : String is do
    var event : Event
    event := sm.events.detect{ e | e.name == eventName }
    // no event found, then there is no element kind associated: return a specific value
    if (event == void) then result := "____noElementKind_"
    else result := event.elementKind
    end
end

operation getTriggerableTransition(sm : StateMachine, state : State, eventName : String) : Transition
is do
    var trans : Transition

    if (eventVerificationLevel == Level.weak)
    then trans := sm.transitions.detect { t | t.source == state and (
        t.event.name == eventName or t.event.elementKind == getEventKind(sm, eventName) ) }
    else trans := sm.transitions.detect { t | t.source == state and t.event.name == eventName }
    end

    if (trans != void) then
        result := trans
    else
        if (state.container != void)
        then
            var cont : State
            cont ?= state.container
            result := getTriggerableTransition(sm, cont, eventName)
        else
            result := void
        end
    end
end

operation newEvent(eventName : String, sm : StateMachine)
    pre existingTransition is do
        if (eventVerificationLevel != Level.none) then
            getTriggerableTransition(sm, getLeafActiveState(sm), eventName) != void
        end
    end

is do
(...)
end
```

```
(...)
do
    // process the new event occurence
    newEvent(event, sm)
rescue (err : ConstraintViolatedPre)
    stdio.writeln(" The event "+event+" is not processed ! ")
    // do the adaptation or stop the execution here
    (...)
end
```

Figure 6: Vérification associée à l'occurence de l'événement à l'exécution

est également du type normal comme le mauve. Un événement blanc viole quant à lui la précondition pour les deux types de vérification.

5 Conclusion

Nous avons présenté une approche de vérification d'adaptation d'exécution de modèles par contrats, reprenant nos contrats d'exécution. L'intérêt pour l'adaptation de directement exécuter un modèle est que le modèle représentant le système n'a plus besoin d'être maintenu en cohérence avec le système, comme cela est nécessaire dans la plupart des approches d'adaptation utilisant des *models@run.time*. En effet, le modèle est alors par principe le système. Ceci étant, pour être capable de différencier et traiter correctement les interactions avec l'environnement d'exécution, nous avons montré que le modèle doit intégrer le plus précisément et exhaustivement possible la spécification des interactions avec cet environnement. Un modèle est alors adapté à un environnement d'exécution donné s'il est capable de gérer explicitement toutes les interactions. Si cet ensemble d'interactions est connu et fini, on peut même déterminer statiquement (avant toute exécution) si ce modèle est adapté ou pas à cet environnement.

Afin de pouvoir utiliser des comportements dégradés acceptables par rapport à un certain environnement, nous avons proposé de définir un double niveau de connaissance sur les éléments formant le comportement du système ou définissant les interactions : soit leur valeur exacte, soit une valeur plus globale, un "type de valeur". Ainsi, nous pouvons définir une hiérarchie de contrats d'adaptation, du plus exigeant au plus relâché.

Nous avons appliqué notre approche sur des machines à états UML simplifées avec un exemple simple de signalisation ferroviaire et de comportement de train. Nous avons implémenté en OCL, et au sein du moteur d'exécution de ces machines à états, plusieurs sortes de contrats et vérifications.

Le travail présenté dans cet article constitue une toute première investigation de l'adaptation de modèles exécutés. La prochaine étape est de définir et tester des cas d'études plus complexes et réalistes. La forme des contraintes à respecter, en OCL et se basant sur le contenu du modèle en cours d'exécution, pourrait être un peu limitée dans certains cas. Pour une certaine politique d'adaptation, on pourrait avoir besoin de se baser sur des conditions appliquées à des propriétés et des règles associées comme cela est fait dans [6] qui définit un méta-modèle générique pour cela. Il serait intéressant d'intégrer ce méta-modèle à celui de nos machines à états. De manière plus générale, il serait intéressant d'étudier concrètement d'autres types de vérification d'adaptation pour d'autres types de modèles exécutables que les machines à états. Toujours concernant l'adaptation, il faudrait également et bien entendu définir des actions d'adaptation, c'est-à-dire exprimer comment un modèle doit être modifié ou transformé pendant son exécution en fonction de conditions, notamment par rapport à l'environnement d'exécution.

Enfin, une autre perspective serait d'appliquer des démarches par contrats pour du *models@run.time* plus classique, là où le modèle représente de manière cohérente une partie de l'état du système et n'est plus directement exécuté comme nous le faisons. Par exemple, nous pourrions reprendre nos travaux de vérification de [8] pour les adapter en version à contrats à l'exécution. Il s'agirait concrètement ici de vérifier la configuration dynamique de services Web en déterminant par exemple si un service est adapté à des besoins fonctionnels et non-fonctionnels.

References

[1] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[2] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[3] Eric Cariou, Cyril Ballagny, Alexandre Feugas, and Franck Barbier. Contracts for Model Execution Verification. In *Seventh European Conference on Modelling Foundations and Applications (ECMFA '11)*, volume 6698 of *LNCS*, pages 3–18. Springer, 2011.

[4] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL Contracts for the Verification of Model Transformations. In *Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages at MoDELS 2009*, volume 24. Electronic Communications of the EASST, 2009.

[5] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the Specification of Model Transformation Contracts. Workshop OCL and Model Driven Engineering at UML 2004, 2004.

[6] Franck Fleurey and Arnor Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, volume 5795 of *LNCS*, pages 606–621. Springer, 2009.

[7] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

[8] Lazhar Hamel, Mohamed Graiet, Mourad Kmimech, Mohamed Tahar Bhiri, and Walid Gaaloul. Verifying Composite Service Transactional Behavior with EVENT-B. In *the 5th European Conference on Software Architecture (ECSA'11)*, volume 6903 of *LNCS*, pages 67–74. Springer, 2011.

[9] Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Design by Contract to improve Software Vigilance. *IEEE Transaction on Software Engineering*, 32(8), 2006.

[10] Grzegorz Lehmann, Marco Blumendorf, Frank Trollmann, and Sahin Albayrak. Meta-Modeling Runtime Models. In *5th International Workshop on Models@run.time at MODELS 2010*, volume 6627 of *LNCS*, pages 209–223. Springer, 2010.

[11] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, 1992.

[12] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@run.time to Support Dynamic Adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[13] OMG. Object Constraint Language (OCL) Specification, version 2.2, 2010. `http://www.omg.org/spec/OCL/2.2/`.

[14] Thomas Vogel and Holger Glese. Language and Framework Requirements for Adaptation Models. In *6th Workshop on Models@run.time at MODELS 2011*, 2011.

[15] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *the 28th international conference on Software engineering (ICSE '06)*, pages 371–380. ACM, 2006.