THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *24/09/2013* **par :**

El Arbi Aboussoror

Méthodes de diagnostic avancées dans la validation formelle des modèles

---

JURY

Mme Mireille Blay-Fornarino	Professeur	Rapporteur
M. Ferhat Khendek	Professeur	Rapporteur
M. Philippe Palanque	Professeur	Examinateur
M. Hubert Dubois	Chercheur CEA	Examinateur
Mme Ileana Ober	Maître de conférences	Directrice de thèse
M. Iulian Ober	Maître de conférences	Directeur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeurs de Thèse :
Ileana Ober et *Iulian Ober*

Rapporteurs :
Mireille Blay-Fornarino et *Ferhat Khendek*

Méthodes de diagnostic avancées dans la validation formelle des modèles.

El Arbi Aboussoror

Résumé

Malgré l'existence d'un nombre important d'approches et outils de vérification à base de modèles, leur utilisation dans l'industrie reste très limitée. Parmi les raisons qui expliquent ce décalage il y a l'exploitation, aujourd'hui difficile, des résultats du processus de vérification. Dans cette thèse, nous étudions l'utilisation des outils de vérification dans les processus actuels de modélisation de systèmes qui utilisent intensivement la validation à base de modèles. Nous établissons ensuite les limites des approches existantes, surtout en termes d'utilisabilité. A partir de cette étude, nous analysons les causes de l'état actuel des pratiques. Nous proposons une approche complète et outillée d'aide au diagnostic d'erreur qui améliore l'exploitation des résultats de vérification, en introduisant des techniques mettant à profit la visualisation d'information et l'ergonomie cognitive. En particulier, nous proposons un ensemble de recommandations pour la conception d'outils de diagnostic, un processus générique adaptable aux processus de validation intégrant une activité de diagnostic, ainsi qu'un framework basé sur les techniques de l'Ingénierie Dirigée par les Modèles (IDM) permettant une implémentation et une personnalisation rapide de visualisations.

Notre approche a été appliquée à une chaîne d'outils existante, qui intègre la validation de modèles UML et SysML de systèmes temps réel critiques. Une validation empirique des résultats a démontré une amélioration significative de l'utilisabilité de l'outil de diagnostic, après la prise en compte de nos préconisations.

Mots-clés : Vérification, SysML, UML, Ingénierie Dirigée par les Modèles, visualisation d'information

Abstract

A plethora of theoretical results are available which make possible the use of dynamic analysis and model-checking for software and system models expressed in high-level modeling languages like UML, SDL or AADL. Their usage is hindered by the complexity of information processing demanded from the modeller in order to apply them and to effectively exploit their results. Our thesis is that by improving the visual presentation of the analysis results, their exploitation can be highly improved. To support this thesis, we define a diagnostic trace analysis approach based on information visualisation and human factors techniques. This approach offers the basis for new types of scenario visualizations, improving diagnostic trace understanding.

Our contribution was implemented in an existing UML/SysML analyzer and was validated in a controlled experiment that shows a significant increase in the usability of our tool, both in terms of task performance speed and in terms of user satisfaction.

The pertinence of our approach is assessed through an evaluation, based on well-established evaluation mechanisms.In order to perform such an evaluation, we needed to adapt the notion of usability to the context of formal methods usability, and to adapt the evaluation process to our setting. The goal of this experiment was to see whether extending analysis tools with a well-designed event-based visualization would significantly improve analysis results exploitation and the results are meeting our expectations.

Keywords: Verification, SysML, UML, Model Driven Engineering, Information Visualisation

Remerciements

Je remercie tout d'abord mes directeurs de thèse, Ileana et Iulian Ober pour leur support scientifique et personnel tout au long de ma thèse, mes deux rapporteurs Mme Mireille Blay-Fornarino et M. Ferhat Khendek pour l'honneur qu'ils m'ont fait d'évaluer mes travaux de recherche.
Je remercie les examinateurs M. Philippe Palanque et M. Hubert Dubois pour leur participation à l'évaluation de mes travaux.
Mes remerciements vont aussi à Michelle Sibilla pour notre collaboration pendant mon stage de master et au début de ma thèse, et aussi à tous les collègues de l'équipe MACAO qui m'ont soutenu pendant ces années de thèse. Merci aux collègues de l'IRIT qui m'ont assisté dans l'expérience de validation, je pense à M.Antonio Serpa et Mlle Anke Brock ainsi qu'aux participants.
Je pense aussi particulièrement à tous ceux qui ont participé à cet effort : à ma belle-famille, mes amis, surtout El Ghali pour avoir relu mon manuscrit avec la plus grande attention.
Un grand merci à celles qui partagent tous les moments de ma vie, ma femme Sabrina et mes filles, Malika et sa future petite sœur.
Enfin je resterai infiniment reconnaissant envers ma mère Malika pour son effort incommensurable et continue d'éducation et de conseil, mon père Abdellah pour être un modèle de réussite, mon frère Badreddine pour son aide dans les moments critiques.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

La révolution numérique actuelle a pour racine le calcul numérique, c'est à dire le calcul automatique sur des informations encodées numériquement. Fondée sur l'information et non sur la matière et l'énergie, elle est plus proche des anciennes révolutions de l'écriture et de l'imprimerie.
Gérard Berry, Leçons Inaugurales du collège de France 2009

La révolution numérique est en train de passer vers une nouvelle phase où les algorithmes et les programmes ne sont plus l'objet central de la réflexion. L'informatique évolue aujourd'hui vers une vision plus globale basée sur les systèmes [150]. Ces systèmes qui embarquent des composants logiciels deviennent de plus en plus complexes. Ils sont soumis à de nombreuses contraintes. Contraintes de temps, de ressources ou d'adaptabilités qui se traduisent en un processus complexe d'ingénierie. A ces contraintes opérationnelles se rajoute le besoin de baisser les coûts tout en maintenant le même niveau de qualité voire l'améliorer.

Les processus d'ingénierie système, actuellement centrée sur des documents de spécification en langage naturel, ne permettent plus de gérer cette complexité en respectant toutes les contraintes [17].

Le paradigme de l'Ingénierie Dirigée par les Modèles (IDM) propose d'unifier tous les aspects du processus en utilisant les notions de modèle et de transformation [43]. L'IDM apporte un ensemble de techniques qui permet aujourd'hui de formuler des solutions rationnelles dans le domaine de l'ingénierie système. L'unification qu'apporte l'IDM constitue un important levier pour construire des chaînes intégrées et complètes de développement permettant une optimisation globale. C'est à dire une optimisation des coûts et des temps de développement qui porte sur toute une chaîne d'outils et de techniques d'ingénierie système.

Le génie logiciel profite aussi des avantages de l'IDM pour passer d'un processus centré sur le code vers un processus plus général, intégrant modélisation,

documentation et codage.

1.1 Contexte des travaux

L'ingénierie système est définie par l'Association Française d'Ingénierie Système (AFIS)[1] comme *"une démarche méthodologique générale qui englobe l'ensemble des activités adéquates pour concevoir, faire évoluer et vérifier un système apportant une solution économique et performante aux besoins d'un client tout en satisfaisant l'ensemble des parties prenantes"*. L'INCOSE (International Council on Systems Engineering) en donne une définition similaire. Les enjeux de l'ingénierie système sont donc pluriels, nous citons les plus importants [2] :

- une maîtrise de la complexité et des coûts inhérents aux produits et aux processus ;
- une meilleure anticipation en amont des problèmes et des risques ;
- un raccourcissement du temps de développement.

L'enjeu principal est la réduction des coûts en gardant ou en améliorant la qualité des produits.

1.1.1 L'ingénierie dirigée par les modèles

Les modèles sont utilisés depuis longtemps par nos ancêtres. Ils répondent au besoin de communication de l'homme qui, pour transporter du sens a besoin de le modéliser dans la réalité.

Ce type de modèles, bien qu'indispensables à la communication n'est pas la seule catégorie de modèles utilisés. En effet, les modèles peuvent être **contemplatifs** [44], ils ont alors pour seul objectif d'éclaircir un point de vue ou de communiquer des choix de conception. Ils peuvent aussi être **productifs**, c'est à dire qu'il peuvent faire l'objet d'un traitement automatique pour produire d'autre artefacts (e.g. pour générer du code).

D'autres types de modèles ont été utilisés dès le début de l'ère numérique. Par exemple, les fabricants des premières machines ont utilisé des modèles pour représenter leurs architectures. Plus récemment, dans le monde du génie logiciel ont peu cité les approches SADT (1977), Hood (1987) et OMT (1991) qui donneront naissance au langage UML adopté par l'OMG en 1997 et standardisé par l'ISO en 2000 [94].

[1] `http://www.afis.fr`
[2] Adapté du site de l'AFIS

Mais ce n'est qu'à partir de la proposition de l'approche Model Driven Architecture (MDA) par l'OMG [143] qu'une approche centrée sur les modèles a commencé à prendre du sens aux yeux des industriels du logiciel, comme en témoigne le nombre de projets de recherche lancés [4, 14]. L'approche MDA, centrée sur la séparation des préoccupations métier et des préoccupations de la plateforme d'exécution, a donné naissance en France à l'Action Spécifique CNRS et à un premier ouvrage collectif sur l'IDM [73].

L'IDM s'inscrit naturellement dans l'évolution de l'approche objet et des modèles à composants. L'IDM est construite autour de deux notions fondamentales que sont les modèles et les transformations. L'idée principale est que tout processus de production peut être vu comme un ensemble de modèles reliés par des transformations. Les modèles sont créés pour un objectif précis dans ce processus ; les transformations produisent de nouveaux modèles.
Il n'y a pas de consensus sur la définition des modèles, mais l'une des plus répandues est celle de Minsky [125] : *"pour un observateur B, un objet M(A) est un modèle d'un objet A si B peut utiliser M(A) pour répondre à des questions qu'il se pose sur A."*. Une définition plus complète est donnée par B. Selic dans [148], il y propose cinq caractéristiques qu'un modèle doit avoir :

- l'**abstraction** : un modèle doit abstraire les détails non pertinent pour un point de vue sur le système qu'il modélise ;
- la **compréhensibilité** : il doit être compréhensible par ses utilisateurs ;
- la **précision** : il doit proposer une représentation du système qui répond de manière réaliste aux questions que l'utilisateur (*observateur* dans la définition de Minsky) se pose sur le système ;
- la **prédiction** : le modèle peut être utilisé pour prédire des propriétés sur le système ;
- le **coût réduit** : Le but de la modélisation étant de réduire les coûts, un modèle doit avoir un coût abordable.

Les modèles font aujourd'hui partie des artefacts de base du génie logiciel sans toutefois être l'artefact de référence.

1.1.2 Apport de l'IDM au génie logiciel

L'objectif principal du génie logiciel est la rationalisation des coûts et de la qualité. L'obstacle majeur reste la complexité des domaines métiers et les contraintes des contextes d'utilisation de l'informatique. L'approche des composants a apporté

une réponse satisfaisante à ce problème dans la dernière décennie. L'évolution des technologies d'implémentation impliquent une migration constante des composants, mais ceux qui implémentent les aspects métiers devraient évoluer avec le métier et non pas avec chaque vague technologique. C'est cette vision qui a donné naissance à l'approche MDA.
Cette dernière est une séparation des aspects métiers et techniques dans le processus de développement du logiciel. L'IDM apporte ensuite une vision plus large où les différents aspects du génie logiciel, liés aux produits comme aux processus et aux intervenants, sont autant de modèles et de formalismes. Ces derniers sont liés par un ensemble de transformations.
Cette vision apporte un ensemble d'avantages indéniables :

- une **unification** dans les représentations et dans la gestion des activités ;
- une meilleure **séparation des préoccupations** métier et techniques ;
- un **découplage** entre les différents aspects d'un produit ;
- une concentration des efforts sur les **aspects métiers** ;
- la **génération** d'une partie ou de tous les aspects techniques.

Ceci marque un grand pas en avant dans l'automatisation et la standardisation des activités du génie logiciel. Il en découle ainsi une réutilisation des produits mais aussi des outils du génie logiciel.

L'apport de l'IDM au génie logiciel a donc un intérêt multiple : la gestion de l'hétérogénéité, la contribution d'un meilleur niveau de cohérence, ainsi que l'utilisation de l'abstraction formelle s'opposant aux abstractions descriptives et ambiguës (e.g. document textuel). Cet apport nécessite toutefois des prérequis qui sont, **l'utilisation des modèles comme référence centrale du processus d'ingénierie et le besoin d'une nouvelle génération d'outillage pour gérer les nouveaux types d'artefacts**. C'est à ces challenges principaux que la communauté des chercheurs et des industriels dans l'IDM tente de faire face.

1.1.3 Apport de l'IDM à l'ingénierie système

L'ingénierie système est actuellement dominée par des processus orientés documents [102]. De la définition du système jusqu'au retrait de service, les spécifications se font souvent dans le langage naturel et les vérifications se font par relecture croisée.

Plusieurs normes définissent l'état de la pratique dans le domaine de l'ingénierie système. La norme IEEE 1220 [88], la plus ancienne couvre les activités

allant de l'analyse des besoins à la réalisation du système. La norme EIA 632[25] couvre en plus, les activités d'intégration du système et de sa mise en production. La norme la plus récente et aussi la plus complète est la norme ISO 15288 [93] qui couvre tout le cycle de vie du système de la définition des besoins au retrait de service. Le modèle de référence CMMI [147] est complémentaire à la définition du processus d'ingénierie système par les normes mentionnée. Il permet au sein d'une approche standard de l'ingénierie système de mesurer le degré de maturité du processus et propose un registre de bonnes pratiques pour son amélioration.
Ces standards rationalisent les processus d'ingénierie système par la définition d'objectifs à atteindre à chaque phase du cycle. Ces objectifs sont exprimés dans des documents informels ce qui nécessite un effort important de vérification et de reprise après erreur.

L'IDM, par les concepts de métamodélisation et de transformation, permet d'augmenter le niveau de formalisation et d'automatisation des activités du processus d'ingénierie système. L'IDM permettra à terme une unification, une standardisation et une réutilisation des outils utilisés. L'unification permet la gestion de l'hétérogénéité et donc apporte un meilleur niveau de cohérence ainsi qu'une baisse des reprises tardives. L'utilisation d'abstractions formelles et non pas uniquement descriptives et ambiguës augmentera le niveau de formalisation des spécifications.
Rappelons que la condition principale pour réaliser cette avancée est de considérer les modèles comme l'artefact de référence et non les documents en langage naturel. Ceci nécessite une nouvelle génération de techniques supportées par un outillage de qualité industrielle.

1.1.4 Vers un processus d'ingénierie système orienté modèle

Réduire les coûts tout en gardant un degré de qualité élevé est aujourd'hui la préoccupation première des différents acteurs de l'ingénierie système. L'IDM est une approche prometteuse ; elle est examinée par de nombreux industriels. Par exemple, Airbus a lancé en 2005 avec ses partenaires le projet Topcased[3] pour explorer les possibilités d'une telle approche. Le constructeur Boeing quant à lui a initié le projet OSEE[4] pour la gestion du cycle de vie système basé sur des technologies de modélisation. De son côté, l'Agence Spatiale Européenne avait initié le projet ASSERT[5], un projet d'envergure pour développer un processus fiable

[3] http://www.topcased.org
[4] http://www.eclipse.org/osee/
[5] http://www.assert-project.net

d'ingénierie système. Enfin, la DARPA developpe au sein du programme Adaptative Vehicule Make (AVM)[6] un ensemble d'outils orienté modèle et ayant comme objectif d'améliorer de manière significative les processus d'ingénierie système.

La standardisation joue un rôle important dans le processus de dissémination de l'IDM dans l'industrie. Pour cela, les spécifications de l'OMG font référence. Plusieurs standards ont vu le jour sous l'égide de l'OMG, les plus utilisés dans le domaine de l'ingénierie système sont SysML, UML et MARTE. SysML couvre les activités de spécification du système et MARTE se propose d'introduire un langage de modélisation des aspects logiciels embarqués. MARTE couvre les aspects de spécification de contraintes non fonctionnelles comme la performance et l'ordonnancement. Ce standard permet aussi d'intégrer les outils de modélisation avec les outils d'analyses déjà existants via le package *Generic Quantitative Analysis Modeling (GQAM)* [15] qui propose un ensemble d'annotations permettant de spécifier les propriétés à analyser sur les modèles mais aussi de reporter les résultats d'analyse. Ces outils d'analyses sont largement adoptés par les acteurs de l'industrie et ont atteint un degré de maturité satisfaisant [15].

Dans le contexte de l'analyse de modèles, deux objectifs métiers sont à atteindre : une utilisation industrielle des techniques et des outils d'analyse, ainsi qu'une utilisation par des utilisateurs non expert dans ces techniques et outils. L'utilisation des outils serait ainsi confiée à un expert des techniques d'analyses qui va annoter ou transformer les modèles vers les formalismes d'analyses. Le modélisateur pourrait alors comprendre les résultats d'analyses et modifier le modèle en conséquence.

1.2 Validation des systèmes temps réel

Cette partie situe la validation des systèmes temps réel dans le processus d'ingénierie et détermine comment se fait cette validation en présentant les différentes approches et techniques employées.

1.2.1 Spécification et vérification des systèmes temps réel

Un système réactif [117] est un système qui interagit constamment avec son environnement. Contrairement à un système transformationnel qui produit un résultat à partir de données en entrée et s'arrête, un système réactif maintient en permanence une interaction avec son environnement. Ce type de système ne peut pas être modélisé en fonction uniquement de ses entrées et sorties, une notion d'*état*

[6] `http://www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__(AVM).aspx`

du système doit être prise en compte. La notion de calcul peut alors être vue comme une suite de changements d'états appelés *transitions*.
Par exemple, un système de gestion d'un réacteur nucléaire est un système réactif, puisqu'il doit en permanence interagir avec le réacteur et ses composantes physiques. Cet exemple a aussi la particularité d'être un système temps réel. Un système est dite *temps réel* si la validité des résultats produits dépend de leurs valeurs et des délais dans lesquels ils sont produits [42]. Un système temps réel doit donc réagir aux événements externes dans un délai précis.

La majorité des systèmes embarqués industriels sont des systèmes réactifs *critiques*. Un système est dit *critique* si son dysfonctionnement peut causer des dégâts matériels voire des pertes humaines. Un tel système est généralement caractérisé par :

- la concurrence dans l'exécution de ces composants : ces systèmes sont souvent réalisés comme un ensemble de composants interagissant entre-eux et avec l'environnent. C'est une des conditions nécessaires pour la réactivité aux événements de l'environnement ;
- le déterminisme du comportement observable : cette caractéristique est liée à la criticité ;
- le respect de contraintes temporelles (on parle alors de systèmes temps réel) ;
- la soumission à des contraintes de ressources.

Au vu de ces caractéristiques, de nombreuses contraintes vont façonner les processus de développement des systèmes temps réel critiques. **Dans la suite du manuscrit on parlera de systèmes temps réel pour signifier des systèmes temps réel, réactifs et critiques**. Il existe plusieurs approches pour la spécification des systèmes temps réels. Celles-ci sont basées sur des formalismes mathématiques de spécification de systèmes dynamiques.

Parmi ces formalismes il y a les systèmes de transitions comme les réseaux d'automates ou réseaux de Petri [142], les algèbres de processus [83, 123] ou encore les réseaux d'automates temporisés [23]. Une fois spécifiés dans un formalisme donné, les modèles des systèmes temps réels doivent être validés. Il faut donc démontrer que les contraintes non fonctionnelles sont satisfaites. L'exemple simple est de vérifier que le système produit des réponses à des stimulus de l'environnement dans un intervalle de temps défini. Pour cela, plusieurs approches existent et cohabitent souvent dans un même processus d'ingénierie industriel.

Pour vérifier les propriétés non fonctionnelles, les techniques de model checking sont largement utilisés, ils intervient plus tôt dans les cycles d'ingénierie et permettent d'éviter les reprises tardives qui correspondent à la majeure partie des

coûts de développement des systèmes embarqués [102]. Une autre approche peu utilisée dans le milieu industriel est la preuve. Celle-ci peut être appliquée même dans les cas où le système a un comportement décrit par un automate infini. Citons aussi la vérification à l'exécution [151](*Runtime Verification*) qui permet une vérification de propriétés sur le comportement réel d'un système.

1.2.2 Model Checking

Les méthodes de validation ont chacune des forces et des limitations. Il convient de savoir les utiliser dans le bon contexte, voire de les combiner. Les méthodes de test permettent de trouver un très grand nombre d'erreurs avec un faible coût de mise en œuvre comparé à des techniques plus formelles comme la démonstration. Cette dernière nécessite la connaissance d'approches mathématiques de preuve et d'outils d'assistance comme Coq [7].

Le model checking, créé indépendamment par Clarke et Emerson[55] d'une part et par Sifakis [139] d'autre part, apporte une approche exhaustive qui a l'avantage d'être totalement automatisable. C'est ce qui fait sa force mais aussi sa limitation. Le parcours de l'espace des comportements possibles est très coûteux en ressource de calcul et il est aussi impossible à réaliser sur certains types de modèles.

Le model checking définit un ensemble d'algorithmes de vérification d'une formule écrite en logique temporelle portant sur le comportement d'un système. Le comportement du système est spécifié sous la forme d'un système de transitions. Les propriétés à vérifier sont principalement de deux types [109] :

- les propriétés de sûreté : stipulent que quelque chose de mauvais n'arrivera jamais ;
- les propriétés de vivacité : stipulent que quelque chose de bon finit par arriver.

Les algorithmes de model checking auxquels nous nous intéressons, sont principalement des algorithmes de parcours de graphes avec une procédure de marquage.

1.2.3 Challenges du model checking

Les algorithmes de model checking sont basés sur un parcours de l'espace d'état. Pour des systèmes industriels cet espace est beaucoup trop grand pour pouvoir faire un parcours exhaustif. Par exemple, le système de contrôle des panneaux solaires d'un véhicule spatial comme l'ATV [64] est composé d'une vingtaine de

[7] The Coq Proof Assistant : `http://coq.inria.fr/`

blocs SysML et donne lieu à un espace d'états trop grand pour être vérifié avec les techniques actuelles.

Ce problème d'explosion de l'espace d'état est une limitation qui empêche l'application des méthodes de model checking sur des modèles de taille industrielle. Plusieurs techniques cherchent à contourner cette limitation. Nous pouvons citer les techniques d'abstraction [66], les techniques de model checking symboliques [119] (e.g. via l'utilisation de *diagrammes de décision binaires [40]*) et la réduction d'ordre partiel [75].
Il existe aussi d'autres pistes qui peuvent être explorées pour arriver à résoudre le problème de l'explosion de l'espace d'états [56]. Par exemple combiner les techniques de model checking avec d'autres techniques de vérification comme l'analyse statique ou la preuve, trouver des algorithmes efficaces ou encore utiliser la symétrie pour réduire l'espace d'états.

Une autre limitation est celle de l'interprétation des contre-exemples générés par le model checker. Ces contre-exemples détaillent le scénario qui mène à un état du système où la propriété spécifiée n'est pas satisfaite, des travaux de Clarke et al. [54] traite la question du contre-exemple, mais ici les auteurs s'intéresset à l'extraction des abstractions à partir d'un contre-exemple dans le cadre du model checking symbolique. Les travaux de Groce et al. [78] s'interessent à l'extraction à partir d'un contre-exemple de traces correctes qui permettraient de comprendre l'erreur.

Nous nous intéressons également aux contre-exemples, mais sous un angle qui vise une meilleure exploitation de ceux-ci dans le cadre des approches de validation qui partent de l'utilisation de langages de modélisation de haut-niveau, tels que UML/SysML. Cette limitation, qui porte sur le diagnostic des contre-exemples constitue l'objet de nos travaux.

1.3 Contenu du mémoire

Nous proposons dans cette thèse de rationaliser une partie des activités engagées dans le processus d'ingénierie système : le diagnostic dans la validation de modèles. Pour cela nous utiliserons l'IDM comme principal levier.

Ce mémoire est constitué de trois parties. Une première partie qui traite de **la problématique** et qui répond principalement à la question : pourquoi les traces générées par le model checker sont difficile à exploiter ? Une deuxième partie qui dresse l'**état de l'art** des outils de diagnostic. Enfin une troisième partie sur les **contributions**, qui apporte des solutions à la problématique du diagnostic pour la validation des modèles.

Chapitre 2

Problématique

Dans ce chapitre nous tenterons d'expliquer pourquoi, en dépit des possibilités de diagnostic des outils actuels, il reste difficile d'exploiter les résultats d'analyses.

2.1 Définition de la sémantique dans l'IDM

Dans un processus orienté modèle, les modèles constituent l'élément central. Les modèles de type *contemplatif* peuvent avoir une sémantique informelle, mais pour être traités de manière automatique, ils doivent être décrits avec une sémantique formelle.
Définir une sémantique formelle donne en effet la possibilité d'analyser les modèles et ceci très tôt dans le cycle d'ingénierie. Par exemple, nous pouvons explorer plusieurs alternatives d'un modèle d'architecture sur la base de propriétés non-fonctionnelles (e.g. la performance) avant d'initier la phase de réalisation d'un système.

Décrire la sémantique d'un langage de programmation ou de modélisation, c'est décrire avec précision la signification des concepts qui le constituent. Pour cela, nous pouvons utiliser le langage naturel comme cela est fait pour la spécification du standard UML [94] (dans sa version 2.4.1). Mais cette approche n'est pas suffisante pour la spécification formelle en vue de la validation des systèmes temps réel. Il faut alors utiliser un langage mathématique. Plusieurs formalismes (Réseaux de Petri, Chaînes de Markov, etc.) peuvent jouer ce rôle. Il existe plusieurs approches (décrites dans la suite) pour définir la sémantique d'un langage de modélisation dans l'IDM.

2.1.1 Taxonomie de la sémantique dans l'IDM

Définir la sémantique en IDM c'est *donner du sens aux concepts de la syntaxe abstraite*. Pour cela, nous devons définir ce sens de manière non ambiguë. Deux approches peuvent essentiellement être utilisées [53] :

- la première approche consiste à décrire une **sémantique opérationnelle** qui modélise le comportement opérationnel (i.e. en termes d'actions) des concepts du langage de modélisation. Pour cela nous pouvons utiliser des transformations endogènes de modèles qui implémentent un ensemble d'actions. Ces actions vont modifier un modèle d'exécution. Une telle approche est décrite dans [62] ;

- la seconde, appelée **sémantique dénotationnelle**, ou **sémantique par traduction** consiste à dénoter (i.e. traduire) les concepts du langage de modélisation vers d'autres concepts qui ont déjà une sémantique formelle.

Ces approches sont implémentées dans l'IDM en utilisant des métamodèles et des transformations de modèles.

2.1.2 Sémantique par traduction

La **sémantique par traduction** consiste à traduire les concepts d'un langage de modélisation dédié (*Domain Specific Modeling Language* (DSML)) vers des concepts qui ont une sémantique formelle. En général la traduction se fait vers des langages qui ont une sémantique formellement définie (e.g. les systèmes d'automates).
La sémantique par traduction présente de nombreux avantages. Dans le contexte de l'analyse de modèles, la traduction permet de réutiliser des techniques et des outils aujourd'hui très matures.

Par exemple l'outil de validation et de modélisation Topcased [14] traduit les modèles de haut niveau comme xSPEM [62] vers un formalisme de bas niveau basé sur les réseaux de Petri. D'autres outils utilisant cette approche seront présentés dans le chapitre état de l'art 3. Dans le contexte de nos travaux, nous utiliserons l'outil IFx-OMEGA [64], il traduit les modèles SysML/UML vers des modèles de bas niveau dans le formalisme IF [36]. Ces modèles de bas niveau seront ensuite validés par un model checker.

L'approche par traduction permet aussi de mutualiser les coûts de développement des outils d'analyses de modèle. Ainsi, seule la traduction vers le formalisme d'entrée de ces outils reste à la charge de celui qui veut réutiliser l'outil d'analyse. Quelques outils libres permettent aujourd'hui de réaliser cette mutualisation dans les coûts, nous pouvons citer SPIN [10] ou NuSMV2[6].

2.2 Limites des approches par traduction

Le fonctionnement d'une grande majorité des chaînes d'outils d'analyse formelle orientée modèles, consiste à traduire la sémantique du langage de modélisation de haut niveau vers un langage de bas niveau. Celui-ci bénéficie d'une sémantique formelle qui constitue une condition nécessaire à toute vérification formelle.

Une fois cette traduction effectuée l'outil d'analyse prend le relais et effectue l'analyse formelle sur les spécifications de bas niveau. Plusieurs outils utilisent cette approche. Par exemple vUML [114] utilise Promela [32], IFx-OMEGA [64] est basé sur IF [36], OpenEmbeDD offre un cadre d'intégration d'un ensemble de notations de haut niveau (e.g. UML, MARTE, SysML, etc...) et des transformations vers plusieurs formalismes de bas niveau (e.g. TINA [13], CADP [2] ou BIP [9]). Une description plus détaillée d'outils utilisant cette approche est présentée dans la partie état de l'art 3. Nous pouvons donc résumer cette approche dans les étapes suivantes :

1. annoter le model de haut niveau avec des contraintes non fonctionnelles ;
2. transformer ce modèle vers un formalisme d'analyse en générant les artéfacts en entrée de l'outil d'analyse ;
3. configurer et exécuter le processus d'analyse ;
4. exploiter les résultats d'analyse.

Au côté de ses avantages, l'approche par traduction souffre de quelques limitations. En effet, la difficulté d'exploiter des contre-exemples générés par l'outil d'analyse pour corriger les erreurs de spécification est une limitation importante. La traduction vers un nouveau langage introduit de nouvelles constructions sémantiques qui ne sont pas toujours présentes dans la sémantique de haut niveau. Ce *gap sémantique* est d'autant plus grand que la traduction se fait vers un formalisme de bas niveau très éloigné du formalisme de départ. Par exemple, traduire un formalisme de type système de transition comme les Statecharts vers un formalisme algébrique comme les algèbres de processus introduit un grand gap sémantique.

Ce gap sémantique est une mesure de l'effort cognitif et moteur nécessaire à l'utilisateur pour interagir avec un outil [87]. Par exemple, dans l'outil IFx-OMEGA [64], les contre-exemples générés dans le format XML peuvent être très difficiles à interpréter par le modélisateur SysML. Les deux niveaux ne sont donc pas *cognitivement équivalents*. Par équivalence cognitive nous signifions que les

charges cognitives induites sur l'utilisateur pour comprendre la même spécification dans les deux niveaux sont différentes. La figure 2.1 résume de manière schématique la différence d'abstraction induite par l'approche par traduction et sa conséquence pour l'utilisateur.

2.2.1 Problème de l'introduction du gap cognitif

La sémantique par traduction induit sur les outils un gap sémantique qui se traduit, pendant l'utilisation des outils par un gap cognitif. En effet, les utilisateurs du langage de modélisation de haut niveau n'étant pas initiés au langage de bas niveau se voient donner des résultats d'analyses (e.g. un contre-exemple) dans une syntaxe et une sémantique qu'ils ne connaissent pas. Il est donc impossible de comprendre la dynamique du système dans cette sémantique.

Comprendre la dynamique d'un scénario du système capturé dans le contre-exemple est une condition nécessaire au diagnostic. C'est principalement pour cette raison que les utilisateurs rencontre des difficultés pour exploiter les résultats d'analyses. Nous montrerons que ce problème d'exploitation des résultats d'analyse est en grande partie dû à l'absence de prise en compte du profil de l'utilisateur dans la conception des outils. Tant que cette préoccupation restera marginale dans la conception d'outils de diagnostic, l'adoption des méthodes d'analyses, par des utilisateurs de langage de modélisation restera très limitée.

2.2.2 Impact sur l'utilisabilité des outils de diagnostic

La question naturelle qui se pose est donc la suivante : "comment présenter les résultats d'analyse à l'utilisateur pour lui permettre de les exploiter ?"
Ce problème a été identifié dans plusieurs travaux [145, 82, 160], mais aucune solution élaborée n'a été développée en tenant compte des aspects cognitifs humains. Il est vrai que le support technologique déployé est important (e.g. dans [82, 160]) cependant l'accent est mis sur les aspects techniques, l'aspect cognitif humain n'étant que marginal.

Les approches actuelles ne semblent pas se préoccuper de l'efficacité des phases de diagnostic. En effet, nous n'avons pas trouvé dans la littérature des expériences qui évaluent l'efficacité des solutions proposées pour les activités de diagnostic. Les abstractions utilisateurs (e.g. les visualisations) ne sont pas conçues en utilisant un cadre théorique ou empirique reconnu.

Les travaux sur le model checking offrent un outillage de vérification configurable qui peut s'adapter à une multitude d'utilisations et de types de modèles. Ils offrent aussi un ensemble de techniques d'abstraction et d'optimisation pour outrepasser le problème de l'explosion des espaces d'états.

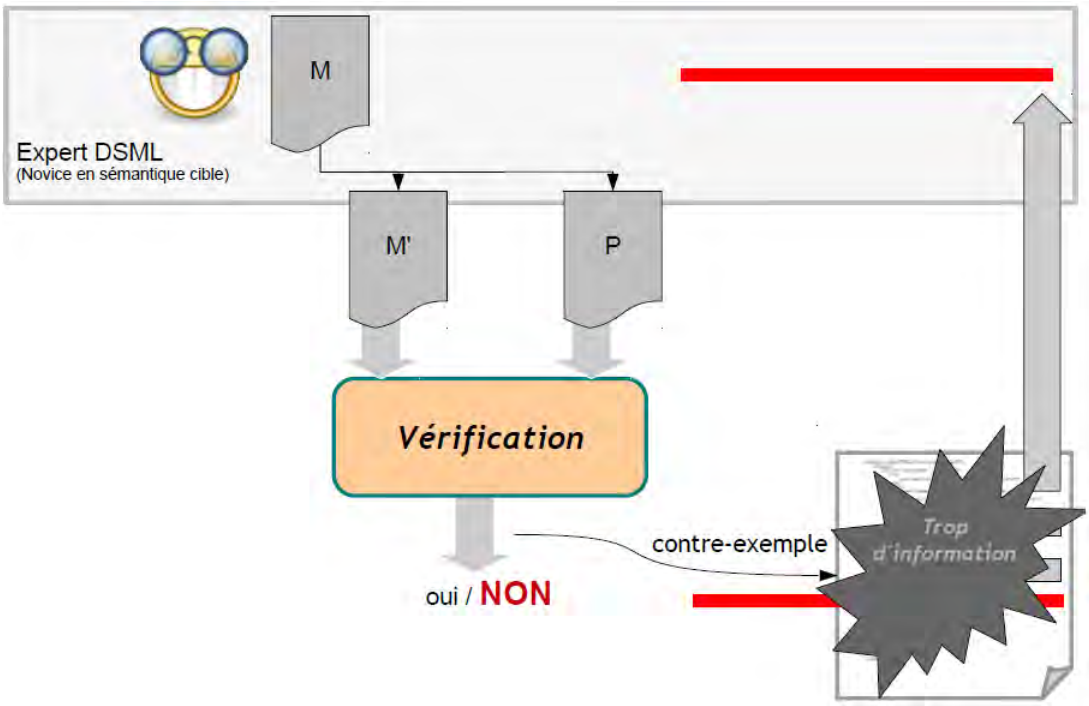


FIGURE 2.1 – Introduction d'un gap cognitif par l'approche de sémantique par traduction. Ce gap se traduit par un problème d'utilisabilité des outils de diagnostic. *M* symbolise le modèle de départ décrit dans le DSML de l'utilisateur, *M'* est le modèle résultat de la traduction et *P* est la propriété à vérifier.

Cependant les efforts déployés sur l'utilisabilité de ces outils sont largement minoritaires, ce qui limite l'adoption des techniques d'analyse formelle auprès des concepteurs de systèmes.

Notre travail s'insère dans la conception d'une approche outillée pour assister l'utilisateur de ces techniques formelles. Nous nous sommes restreints à l'activité de diagnostic dans le processus de validation. Cette partie est cruciale pour l'adoption des techniques d'analyse. L'utilisateur qui n'arrive pas à comprendre les causes d'une erreur de spécification portée par un contre-exemple est incapable de corriger sa spécification ou d'appliquer les nombreuses techniques d'optimisation et d'abstraction disponibles.

Il nous semble donc primordial d'aider l'utilisateur à tirer un maximum d'information de cette phase en amont du processus de validation. Ceci va guider nos travaux vers une approche plus large qui prend en compte les spécificités du système cognitif humain.

2.3 Analyse du problème : perception et cognition

Dans cette section nous allons nous intéresser au problème du gap cognitif chez l'utilisateur du DSML. Ceci nous permettra de comprendre comment l'utilisateur interagit avec les informations de diagnostic et pourquoi cette interaction n'est pas efficace.

2.3.1 Introduction à la structure du système perceptif et cognitif humain

Pour une meilleure analyse, il convient tout d'abord de découvrir la structure du système perceptif et cognitif chez l'homme. Comprendre sa structure avec un modèle simple permet dans un premier temps de comprendre un grand nombre de dysfonctionnements liés à l'exécution d'une tâche par l'utilisateur. Ce système est constitué essentiellement de quatre parties :

- **l'œil** : qui est l'organe sensoriel responsable de la capture de l'information externe, présentée sur un écran dans notre contexte ;

- **le processeur visuel** : traite l'information envoyée par l'œil ; Ce traitement est très rapide (quelques millisecondes), automatique et parallèle. En plus, il ne nécessite aucun effort conscient de la part de l'utilisateur ;

- **le processeur cognitif** : donne un sens à l'information traitée et renvoyée par le processeur visuel. Contrairement à ce dernier, le processeur cognitif fonctionne de manière séquentielle contrôlée par l'utilisateur et nécessite donc un effort de sa part ;

- **la mémoire** : est chargée de stocker les souvenirs. Une partie de ces souvenirs constitue l'expertise acquise. En réalité il y a plusieurs types de mémoires. La mémoire iconique ou sensorielle qui est intégrée dans le processeur visuelle. La mémoire de travail, intégrée dans le processeur cognitif et très impliquée dans les processus de types résolution de problème (e.g. tâche de diagnostic). Et pour finir, la mémoire long terme.

Ce modèle schématisé dans la figure 2.2, est un modèle simplifié basé sur le modèle de Kieras et Meyer [120]. Card et al. proposent un modèle plus complet dans [45].

2.3.2 Fonctionnalités et spécificités des composants du système de perception/cognition humain

A partir de la décomposition du système perceptif et cognitif proposée ci-dessus, nous tenterons de comprendre le déroulement d'une tâche cognitive de type résolution de problème. Pour cela, nous allons nous intéresser au rôle de chaque composant dans ce type de tâche où l'œil est considéré comme un transporteur fiable de l'information.

2.3.3 Le processeur visuel : la perception

Le processeur visuel traite l'information affichée. Il construit des formes dites de haut niveau à partir des objets affichés. Pour utiliser correctement les capacités considérables de ce processeur il convient de bien en saisir le fonctionnement. Pour cela plusieurs modèles théoriques ont été proposés. Kofka [105] propose la théorie de la forme connu aussi sous le nom de la *Gestalt* (forme en allemand). Cette théorie stipule que la perception est un processus de synthèse qui construit des formes de haut niveau à partir des objets que nous percevons. Cette théorie qui rassemble un ensemble de principes permet d'expliquer le caractère parallèle de certains traitements, comme par exemple la détection de regroupements dans des diagrammes. Cette détection se fait de manière automatique par le système perceptif selon le *principe de proximité* énoncé par la Gestalt et qui suggère que des objets à proximité les uns des autres sont groupés dans une forme unique.

Un autre modèle fondamental est celui de Bertin[34]. Bertin propose dans la *sémiologie graphique* une modélisation systématique des formes visuelles. Il définit pour cela un ensemble de huit variables visuelles qui décrivent l'espace des

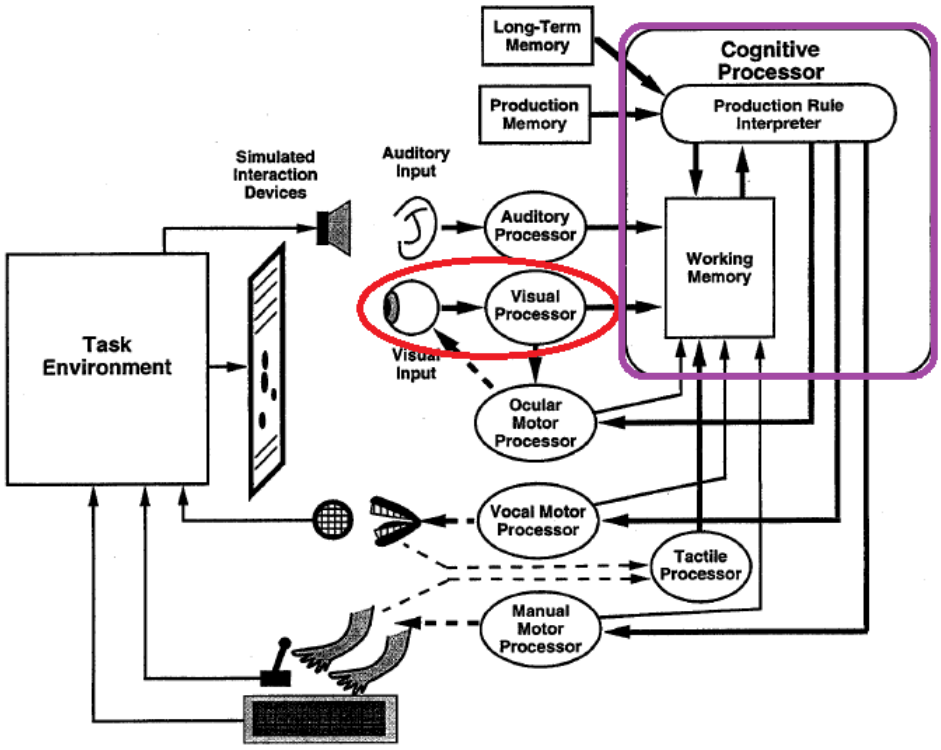


FIGURE 2.2 – Modèle des systèmes cognitif (encadré rectangle) et perceptif (encadré ovale) chez l'humain selon [120]

formes visuelles. Il décrit aussi la manière d'utiliser ces variables en fonction de l'information à communiquer.

2.3.4 Le processeur cognitif : mémoire et cognition

Le processeur cognitif est constitué de la mémoire et des processus cognitifs qui participent à la compréhension et à l'apprentissage. La tâche de diagnostic peut être décomposée en plusieurs objectifs à atteindre par l'utilisateur. Celui-ci doit d'abord comprendre le comportement stocké dans le scénario à diagnostiquer puis localiser l'erreur.

Dans la phase de compréhension, plusieurs composants du modèle cognitif interviennent. Tout d'abord la mémoire de travail charge les informations nécessaires à la compréhension. Ensuite, le modèle mental donne une sémantique aux éléments stockés et les met en relation. La mémoire de travail, outil indispensable dans la phase de compréhension, souffre d'une limitation importante. En effet, pour les éléments de type visuel, la mémoire de travail ne peut emmagasiner que cinq à sept éléments visuels [121]. Ces éléments sont aussi appelés *mnèmes* et sont définis par les connaissances acquises de l'utilisateur. Par exemple, pour un enfant qui apprend l'alphabet et qui ne connaît pas encore de mots, le mot *MODELE* comporte six *mnèmes*, pour un adulte ce n'est qu'un seul *mnème*.

Dans le contexte de la validation de systèmes critiques avec la méthode du model checking, la quantité d'information générée par l'activité de parcours exhaustif du graphe d'états est importante. Même si le contre-exemple généré, ne correspond qu'à une partie du graphe atteignable, cette quantité reste très au-delà des capacités de notre mémoire de travail.

La mémoire de travail se trouve donc en surcharge. Ce phénomène s'observe chez le novice (qui ne connaît pas la sémantique de bas niveau) et porte le nom de **surcharge cognitive**.

2.3.4.1 Théorie de la charge cognitive

Ce phénomène de surcharge cognitive a été relevé par Sweller pour la première fois dans [48] et depuis il sert à expliquer un bon nombre de limitations, en ergonomie logicielle mais aussi dans des environnements d'apprentissages classiques [127].
Par exemple, cette théorie peut expliquer l'inefficacité de certains outils de diagnostic par animation des modèles de comportement. En effet, les approches existantes n'offrent qu'une simulation pas-à-pas après le retour du contre-exemple. Dans un contexte industriel, il n'est pas suffisant de faire du pas-à-pas puisque l'utilisateur doit mémoriser une grande partie des informations liée aux états successifs du modèle animé.

Le novice n'a pas de modèle mental causal complet contrairement à l'expert en sémantique de bas niveau. Quand une erreur dans le système est détectée, l'expert, grâce à son modèle mental, remonte les différentes chaînes causales pour trouver l'erreur dans le résultat d'analyse. Cette erreur sera utilisée par l'expert pour rechercher l'erreur de conception dans le modèle de départ.

Assister l'utilisateur novice à construire un modèle mental ou lui fournir par exemple des indications serait une piste prometteuse pour diminuer la charge cognitive inhérente au processus de diagnostic.
Le rôle de la mémoire de travail est primordial dans le bon déroulement de toute tâche cognitive. Pour que l'utilisateur puisse exécuter la tâche de diagnostic de manière efficace, il faut que sa mémoire de travail puisse gérer l'information qui lui est déléguée par le processeur visuel.

2.3.4.2 Charge cognitive et expertise

Ce problème de la charge cognitive n'est pas observé chez l'expert [153]. Par exemple les joueurs d'échecs professionnels peuvent stocker la configuration complète d'un échiquier en mémoire de travail [49]. Les éléments de cette configuration dépassent largement la limite théorique de la mémoire de travail. Mais ceci est possible grâce à un phénomène observé uniquement chez les experts : *le regroupement*.
L'expert utilise le *regroupement* pour passer outre la limitation de sa mémoire de travail. Pour cela, il utilise les capacités disproportionnées de la mémoire à long terme, par rapport à la mémoire de travail, pour pré-traiter l'information envoyée par le processeur visuel.
L'expert fait des regroupements pour permettre le stockage en mémoire de travail, condition nécessaire à tout traitement cognitif.

2.3.5 Implications pour la conception d'outils de diagnostic

A la lumière du modèle cognitif présenté ci-dessus, il devient aisé de comprendre les limitations des outils de diagnostic actuels. L'utilisation des connaissances actuelles du fonctionnement du système perceptif et cognitif humain peuvent expliquer les problèmes d'utilisabilité de tels outils. Cette compréhension fournit aussi un levier important pour la conception d'outils ergonomiques. Nous parlons ici d'ergonomie cognitive, c'est-à-dire des techniques qui permettent d'améliorer l'efficacité de l'utilisateur pour une tâche cognitive donnée.

Il est important de noter que cette optimisation de l'ergonomie se fait pour un profil d'utilisateur et une tâche donnés. L'étude de ces aspects permet de mettre en exergue un ensemble de contraintes à respecter et des recommandations à suivre

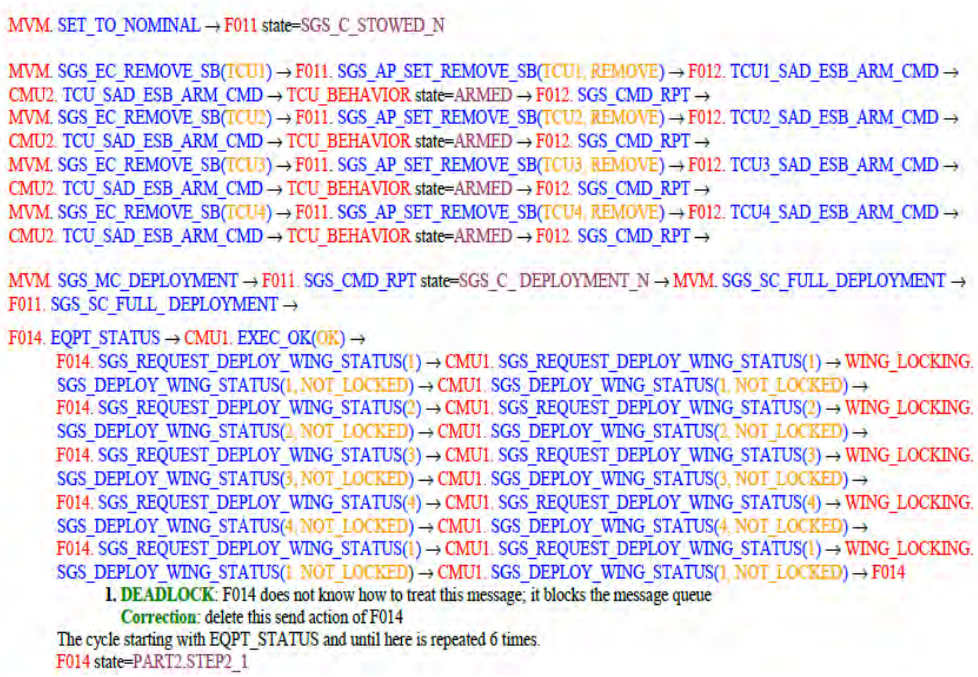


FIGURE 2.3 – Exemple d'une partie d'un aide mémoire créé manuellement à partir d'un contre-exemple. L'utilisateur a rajouté un code de couleur pour comprendre le comportement stocké dans le scénario d'erreur.

dans la conception des outils. Si ces contraintes ne sont pas prises en compte, le diagnostic d'erreur prend trop de temps et demande un effort considérable de la part de l'utilisateur. Par exemple le diagnostic avec la version classique de l'outil IFx-OMEGA, se trouve dans ce cas de figure. En effet, même pour un utilisateur initié à l'outil et sur des modèles de taille importante, il est nécessaire d'utiliser un aide mémoire externe [161] pour le diagnostic. La figure 2.3 montre un exemple réel d'aide mémoire sous forme d'une trace coloriée à l'aide d'un outil de traitement de texte.

Ne pas prendre en compte les aspects cognitifs pousse dans certains cas à mimer la conception des débogueurs des langages de programmation, sans aucune preuve empirique de leur efficacité [126]. Par exemple, quelques outils miment les interfaces des langages de programmation comme Yakindu (cf. figure 2.4) qui se base sur le même agencement de l'interface de débogage pour les Statecharts que pour les langages comme Java.

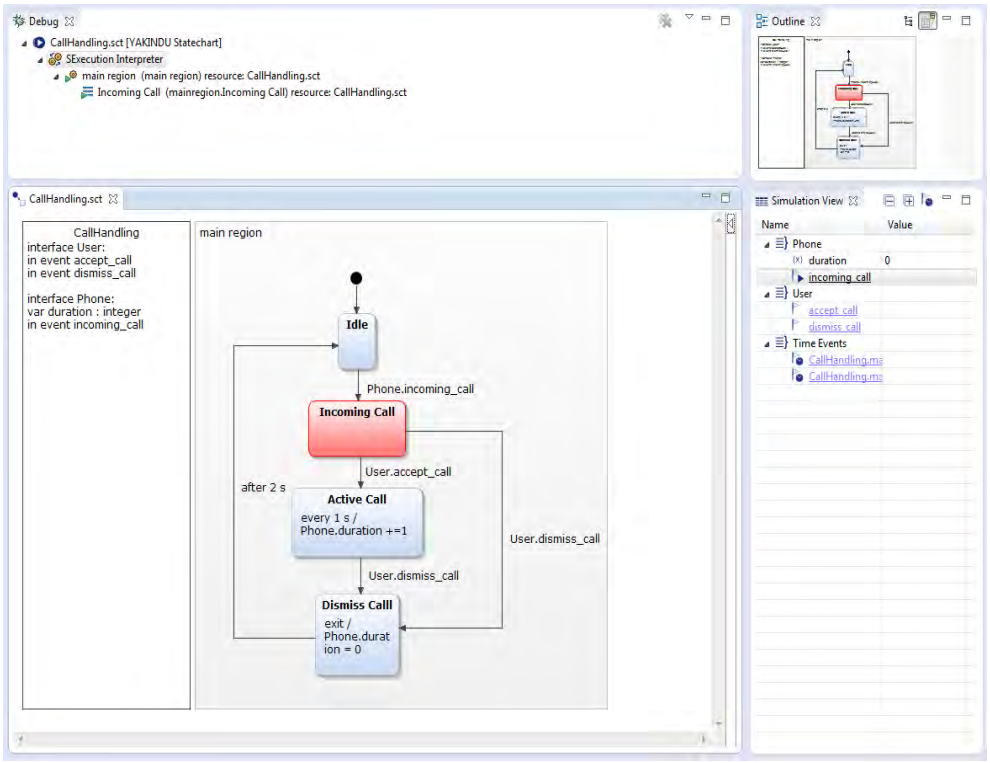


FIGURE 2.4 – Interface de simulation de modèle calquée sur l'interface de débogage des langages objet (e.g. Eclipse JDT)

2.4 Résumé de la problématique

Les problèmes de diagnostic et d'adoption des méthodes d'analyses sont essentiellement dus à la non prise en compte des spécificités cognitives et perceptives humaines. Les concepteurs d'outils de diagnostic n'intègrent pas de phases d'études des processus cognitifs initiés dans le contexte du diagnostic. La question qui se pose est donc la suivante : "**comment permettre à un utilisateur non expert en sémantique de bas niveau d'explorer et de comprendre un contre-exemple ?**"
Il s'agit donc essentiellement d'un problème d'ergonomie. Il existe en fait deux types d'ergonomies. Une ergonomie dite physique, qui prend en compte les spécificités physiologiques de l'utilisateur pour améliorer son efficacité, et une ergonomie cognitive, qui s'intéresse aux aspects cognitifs. C'est ce dernier type d'ergonomie qui manque aujourd'hui dans les outils de diagnostic.

2.5 Objectifs

Comprendre le fonctionnement du système cognitif de l'utilisateur des formalismes de haut niveau a permis de comprendre l'origine du problème de diagnostic. Les objectifs que nous allons poursuivre dans nos travaux seront le suivants :

- prendre en compte le système cognitif humain. Plus particulièrement, assister l'utilisateur dans l'exploration et la compréhension de la quantité importante d'information générée pendant le diagnostic ;
- proposer une approche outillée offrant un cadre de diagnostic plus efficace. Il s'agit de proposer un processus générique et réutilisable avec n'importe quel outil d'analyse, et un framework de conception et d'extension d'outils d'analyse ;
- appliquer l'approche et l'outillage pour en évaluer l'efficacité.

Ces objectifs sont quantifiables, et une mesure de l'efficacité de notre approche sera développée dans la section évaluation 4.4.

Chapitre 3

État de l'art

3.1 Techniques et outils pour le diagnostic dans la validation des modèles

Deux approches existent dans les processus de validation des systèmes critiques. Une première approche que nous qualifierons d'*approche bas niveau* utilise des formalismes de spécification dits de bas niveaux. Ces formalismes de spécification sont basés sur des formalismes mathématiques pour la spécification de systèmes dynamiques. Par exemple nous pouvons citer : les systèmes de transitions comme les réseaux d'automates ou réseaux de Petri, les algèbres de processus [83, 123] ou encore les réseaux d'automates temporisés [23]. Une seconde approche dite *approche haut niveau* se base sur une représentation des systèmes dans un formalisme de haut niveau permettant d'exprimer directement les concepts métier manipulés par l'utilisateur. Il s'agit dans ce cas d'utiliser des formalismes plus souples, permettant d'avoir une expressivité plus accessible pour le concepteur métier du système. Ces formalismes en plus d'être plus abordables offrent aussi des mécanismes d'extension (e.g. les profils UML) pour capturer la sémantique propre au domaine du concepteur. Le concepteur, qui est souvent un expert métier pourra plus aisément spécifier des systèmes critiques et comprendre les spécifications écrites par d'autres concepteurs du même domaine. Chaque approche apporte avec le lot de ses avantages une série de limitations qu'il faudra prendre en compte dans le choix du processus de spécification et de validation de systèmes critiques. Ces spécificités propres à chacune des deux approches seront abordées dans la suite de ce chapitre.

Dans la première catégorie des formalismes de spécification, on trouve les outils de model checking. Ceux-ci offrent la possibilité à l'utilisateur de vérifier automatiquement une propriété spécifiée sur un modèle du système. En pratique, ce

processus nécessite l'intervention de l'utilisateur dans le cas où la propriété n'est pas satisfaite sur le modèle ou quand le processus de vérification ne s'arrête pas. Dans le premier cas le model checker fournit une trace d'exécution menant à un état du système où la propriété est violée [111], dans le second l'utilisateur doit effectuer des changements sur la spécification ou sur la stratégie de vérification pour éviter l'explosion de l'espace d'états. Une technique prometteuse consiste à appliquer des abstractions qui préservent les propriétés du modèle tout en minimisant l'espace d'état à explorer [58]. La section 3.2 résume les fonctionnalités de diagnostic offertes par quelques outils de model checking.

La trace générée par le model checker ne permet pas de diagnostiquer facilement, ou d'une manière systématique, les erreurs d'une spécification de haut niveau. Il est en effet difficile pour un utilisateur d'un langage de spécification de haut niveau de comprendre la trace d'exécution de bas niveau. Ceci est largement dû au gap cognitif induit par la différence dans la sémantique d'exécution des deux niveaux. Ce point a été abordé en détail dans la section 2. Plusieurs travaux se sont donc dirigée vers la compréhension de ces traces par l'utilisateur des notations de haut niveau. Ces techniques peuvent être regroupées en quatre catégories :

- les techniques de *diagnostic par* ***animation*** des modèles sources ;
- les techniques de *diagnostic par* ***annotation*** des modèles sources avec des résultats extrait des contre exemples ;
- les techniques de *diagnostic par* ***visualization*** *des contre-exemples* ;
- les techniques de *diagnostic par* ***extension*** de l'outil de bas niveau (extension souvent écrite dans un langage de programmation).

Dans la section 3.3 nous passerons en revue les travaux basés sur ces différentes techniques. Les outils cités ont été sélectionnés sur la base des outils présentés dans [146, 59] complétée de quelques travaux récents qui intègrent des mécanismes de diagnostic. Notre état de l'art ne se veut pas exhaustif, mais représentatif de l'état des pratiques.
Nous avons étudiés ces contributions selon les critères suivants :

- Le(s) formalisme(s) utilisés pour la spécification du système
- Le(s) formalisme(s) utilisés pour la spécification des propriétés à vérifier
- L'existence d'un support utilisateur pour la spécification (e.g. éditeur de texte avec vérification de syntaxe)
- L'existence d'un support utilisateur pour le diagnostic

3.2 Le diagnostic dans les outils de Model Checking

3.2.1 Uppaal

Uppaal [31, 16] est un outil de spécification et de vérification des systèmes à base de réseaux d'automates temporisés [112]. Uppaal utilise le formalisme des automates temporisés proposé par Alur et Dill [23] avec des extensions. Ces extensions permettent de spécifier des constantes et des variables mais aussi des mécanismes de synchronisation entre automates. La synchronisation d'automates permet de décrire la communication entre deux processus, on parle alors de synchronisation binaire. Mais elles permettent aussi de synchroniser plusieurs processus avec une synchronisation de type broadcast. L'utilisateur peut aussi spécifier des fonctions spécifiques dans une syntaxe proche de celle du langage C. Uppaal permet de spécifier des propriétés de sûreté ou de vivacité. Cette spécification se fait à l'aide d'un langage de la logique temporelle temporisée, TCTL [22] (Timed Computation Tree Logic).

Uppaal offre une interface graphique pour la spécification des réseaux d'automates ainsi que pour la vérification automatique. Cette interface permet aussi de saisir les paramètres de configuration du model checker [31].

Les traces de diagnostic sont générées dans des fichiers au format textuel. Mais pour aider l'utilisateur dans le diagnostic de ces traces d'erreurs, deux types de visualisations sont offerts (cf. figure 3.1) : une vue systèmes qui affiche les automates des processus instanciés et une vue qui affiche les messages échangés entre les processus. La première vue (en haut à droite de la figure 3.1) est de type réseaux graphiques [34] et affiche les automates respectifs des processus, les états actifs ainsi que la valeurs des variables et horloges. La seconde vue (en bas à droite de la figure 3.1) affiche les messages de synchronisation échangés par les processus actifs ainsi que les états de contrôles actifs dans les automates. Cette dernière vue utilise un formalisme similaire aux diagrammes de séquences UML. Ces deux vues sont animées par l'utilisateur via une interface de contrôle selon le mode de simulation pas-à-pas. Pour les modèles trop larges, le simulateur d'Uppaal peut être utilisé en ligne de commande. Uppaal est utilisé dans des contextes industriels et plusieurs travaux ont été publiés dans ce cadre [141, 81].

3.2.2 Symbolic Model Verifier (SMV)

SMV est historiquement le premier outil de model checking symbolique qui a permis la validation exhaustive de propriétés spécifiées dans la logique CTL [55] (Computation Tree Logic) sur des systèmes de taille conséquente [57]. SMV utilise

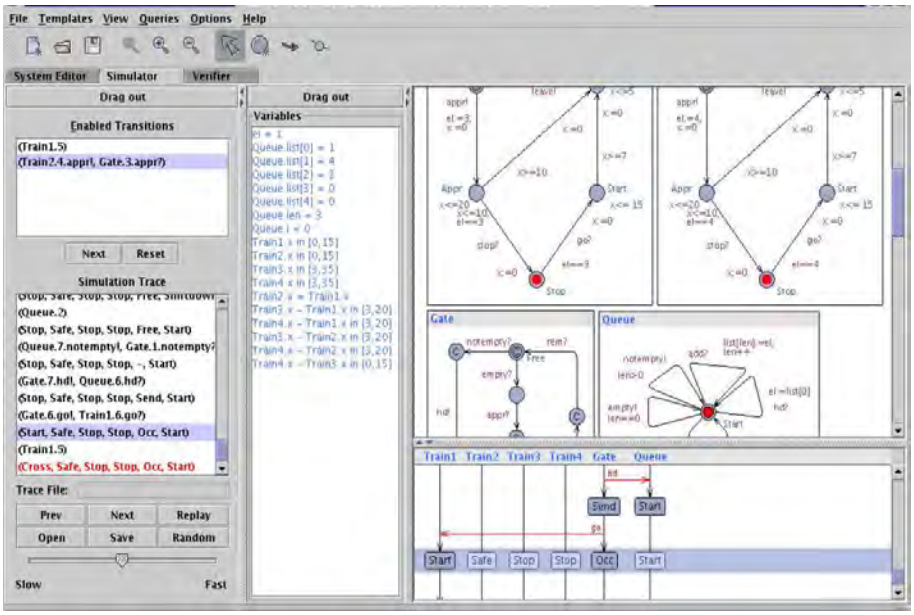


FIGURE 3.1 – Dans Uppaal, le support au diagnostic est fait par l' animation des automates de départ et par la visualisation des variables et des messages échangés.

un formalisme déclaratif [119] qui permet de décrire des machines à états grâce à des *modules* ou *processus* avec *variables* et *transitions*. Ce formalisme, dépourvu de la notion d'état de contrôle, peut sembler déroutant pour l'utilisateur habitué aux langages de spécification basés sur des états. Pour définir des états de contrôle il faudra passer par la notion de *variable*.

SMV ne fournit pas d'interface graphique pour la spécification des modules, seul le format textuel est supporté. L'utilisateur décrit le système dans un fichier et le soumet à une vérification automatique du model checker. Spécifier un système de plusieurs automates communicants avec SMV revient à créer un module ou processus pour chaque automate et des variables partagées pour permettre la communication binaire entre les automates. La notion de module permet de spécifier un comportement séquentiel pour le réseau d'automates tandis que la notion de processus permet d'avoir des automates qui communiquent et évoluent de manière asynchrone. Le formalisme choisi pour la spécification des propriétés dans SMV est CTL [55]. La syntaxe est très proche de la syntaxe mathématique de CTL ce qui rend la spécification très intuitive pour l'utilisateur initié. Pour la vérification automatique, l'outil n'offre pas d'interface graphique. La vérification doit être lancée en ligne de commande. En cas d'erreur, le model checker génère une trace de diagnostic pour aider l'utilisateur à identifier et corriger l'erreur de

```
specification is false

AG (proc1.state = entering -> AF proc1.s... is false:

.semaphore = 0
.proc1.state = idle
.proc2.state = idle

next state:

[executing process.proc1]

next state:

.proc1.state = entering

AF proc1.state = critical is false:

[executing process .proc2]
```

FIGURE 3.2 – Trace de diagnostic généré par le model checker SMV. Seuls les éléments qui changent sont listés dans le fichier.

spécification. Cette trace est générée au format textuel (cf. figure 3.2). Pour simplifier ces traces et en minimiser la taille, seuls les éléments qui changent d'un pas à l'autre sont listés. Aucun outil de visualisation n'est fournit par SMV pour permettre d'explorer et d'analyser ces traces d'erreur. L'outil est aussi dépourvu de fonctionnalités de simulation contrôlée par l'utilisateur.

3.2.3 NuSMV 2

NuSMV 2 [52] est un outil de model checking symbolique fondé sur les diagrammes de décision binaire (Binary Decision Diagrams) [40] et la satisfaction booléenne (SAT) [27]. Il est en fait une ré-implémentation de SMV. Cet outil rajoute plusieurs nouveautés aux fonctionnalités existantes dans SMV. Ainsi, nous trouvons plusieurs interfaces graphiques, et un support pour la spécification de propriétés dans la logique LTL (Linear Temporal Logic [113]). S'inspirant de son prédécesseur SMV, la spécification des systèmes à vérifier se fait avec une extension du langage de spécification de SMV. Une interface graphique permet néanmoins d'appréhender plus facilement cette tâche. Les propriétés à vérifier sur le système sont spécifiées dans la logique LTL ou CTL. Leur vérification est basée sur des algorithmes de model checking et de satisfaction booléenne (SAT). L'utilisateur peut

choisir la méthode à utiliser pour la vérification. En cas de non satisfaction des propriétés spécifiées l'utilisateur a la possibilité de lancer une simulation en mode pas-à-pas ou en mode aléatoire. Il peut aussi enregistrer le résultat de la simulation sous forme d'une trace qu'il pourra rejouer ultérieurement. Pour le diagnostic, l'outil offre la possibilité de naviguer dans la trace avec des commandes textuelles. Il permet aussi d'apporter des modifications dans les traces (e.g. changer la valeur d'une variable). Mais il ne fournit pas dans la version actuelle (2.5) de support graphique pour en faciliter la compréhension et la manipulation.

3.2.4 CPN Tools

CPN Tools[97] est la nouvelle version de Design/CPN développé à l'université d'Aarhus au Danemark. CPN Tools est utilisé par une large communauté d'universitaires et d'industriels ce qui a permis son développement. CPN Tools permet de spécifier les systèmes temps réel dans le formalisme des réseaux de Petri colorés [98]. La spécification peut se faire sous forme de réseaux hiérarchiques, ce qui en facilite la description et la compréhension.

Le langage utilisé pour la spécification est propre à l'outil mais le vérificateur syntaxique facilite grandement cette tâche. Le système peut aussi être spécifié de manière visuelle grâce à l'éditeur graphique. Les propriétés peuvent être spécifiées dans la logique CTL. L'interface utilisateur permet une fois encore de spécifier les propriétés de manière conviviale.

L'outil offre deux modes de simulations : la simulation pas-à-pas et la simulation automatique. Dans chacun des modes, l'interface graphique assiste l'utilisateur dans la compréhension du comportement spécifié par un système d'activation par coloriages des composants comme le montre la figure 3.3. Des critères d'arrêt peuvent être spécifiés pour contrôler le mode automatique. La vérification automatique génère une trace au format textuel, mais un mécanisme d'extension de l'outil permet d'associer des fonctions utilisateurs à des changements d'états du système dans une trace donnée. Ces extensions doivent être écrites dans le langage SML [124]. Dans sa version actuelle (i.e. v3.4) CPN Tools offre de nouvelles fonctionnalités de visualisation. L'utilisateur peut utiliser une librairie pour créer des interfaces de visualisation qu'il peut connecter au simulateur pour une meilleure compréhension des traces. Il est à noté que pour exploiter cette fonctionnalité, des connaissances de la sémantique des réseaux de Petri colorés sont nécessaires.

3.2.5 SPIN

SPIN [85, 32, 10] est un outil très mature, utilisé par un large panel d'universitaires et d'industriels. Il est largement documenté et propose plusieurs extensions. SPIN

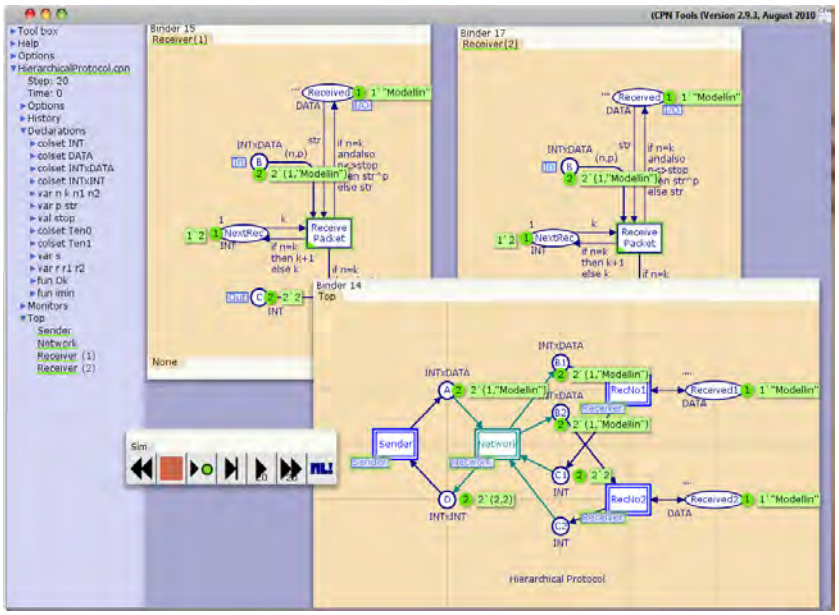


FIGURE 3.3 – Interface de simulation dans l'outil CPN Tools (v2.9). L'interface assiste l'utilisateur dans la compréhension du comportement par un système d'activation par coloriages

est utilisé pour la spécification de réseaux d'automates communicants par canaux bornés. Le langage de spécification (PROMELA) est spécifique à l'outil mais il est largement inspiré du langage C.

SPIN permet de spécifier et de vérifier des propriétés de la logique PLTL [70]. La vérification bénéficie de nombreuses optimisations, ce qui permet de vérifier des systèmes de taille industrielle [84, 144]. SPIN bénéficie de plusieurs intégrations à des environnements de développement comme Eclipse [107]. Cette intégration rend l'outil encore plus abordable. Comme la plupart des autres outils, la simulation dans SPIN offre deux modes, un mode pas-à pas où l'utilisateur peut visualiser l'envoi de messages entre processus et un mode automatique qui permet une exploration plus rapide de l'espace d'états. Comme Uppal, SPIN offre une vue (fig. 3.4) similaire aux *Message Sequence Chart* [96] de SDL pour guider l'utilisateur dans la simulation [108].

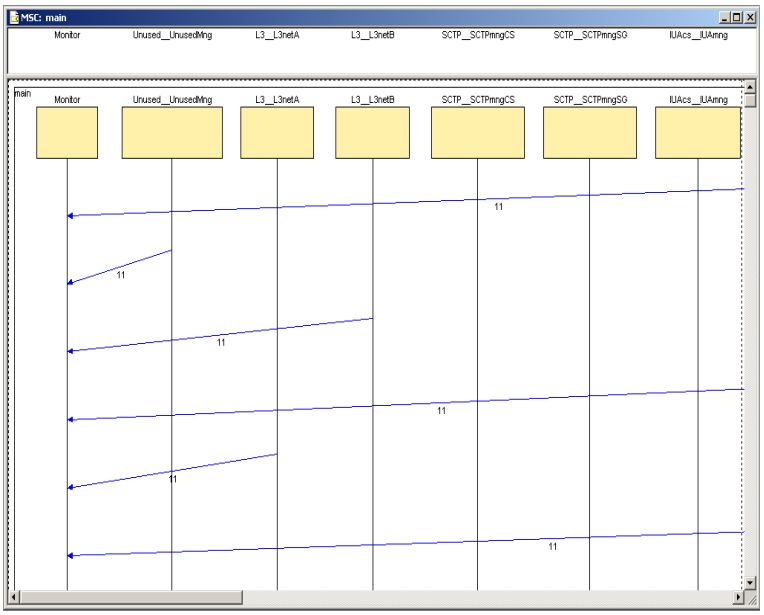


FIGURE 3.4 – Trace de diagnostic généré par le model checker SPIN et affichée sous forme de MSC grâce à une extension de l'outil [108].

3.3 Le diagnostic dans les outils de validation de modèles de haut niveau

Avec l'utilisation de plus en plus répandue des techniques IDM, et plus précisément de modèles exprimés dans des formalismes de haut niveau (e.g. UML), le besoin d'utiliser des méthodes de validation est apparu. Plusieurs travaux précurseurs montrent l'intérêt de la validation des modèles de haut niveau [37, 71]. Ces méthodes ont pour objectif principal d'augmenter la confiance de l'utilisateur dans les modèles produits. Au même titre que les processus utilisés dans la certification des applications critiques (e.g. la norme DO-178), les approches de validation de modèles visent à apporter au cycle d'ingénierie un cadre formel ou semi-formel pour atteindre des objectifs formulés en termes de contraintes. Les approches de diagnostic citées dans la précédente section apportent une solution dans un cas précis d'utilisation. Il s'agit d'approches d'ingénierie où les tâches de validation sont attribuées à une équipe d'experts dans les méthodes formelles. Cette équipe se doit d'avoir une expertise opérationnelle sur les formalismes et outils de bas niveau utilisés pour spécifier et valider les modèles. Ces contraintes d'organisation et de cout limitent grandement l'adoption de cette approche. D'autres problèmes se posent et limitent l'utilisation d'une telle approche. Par exemple, les problèmes

qui découlent des transformations entre formalismes doivent être traités au sein de ce type d'organisation dans les équipes de conception. Ces problèmes ont été abordés dans la section 2.
Dans le cas où une telle organisation des équipes ne peut pas être mise en place il faut utiliser une approche qui s'adapte aux utilisateurs du formalisme de haut niveau. Cette section dresse un état de l'art des outils et techniques qui abordent la validation de modèles exprimés dans des langages de haut niveau.

3.3.1 IFx-OMEGA

La plateforme IFx-OMEGA[1] offre une panoplie d'outils autour de la spécification et de la validation de systèmes temps réel embarqués [130]. La spécification du système à vérifier se fait avec un profil UML, appelé OMEGA. Ce profil est disponible pour deux outils largement utilisés [130] : IBM Rhapsody et Papyrus. Le profil définit un ensemble de stéréotypes pour la spécification des systèmes temps réel. Les propriétés temporelles ou/et temporisées sont spécifiées grâce à des machines à états UML. Ce qui fait une des forces de cet outil puisqu'il n'introduit pas un nouveau formalisme pour la spécification des propriétés. L'outil réutilise donc le formalisme connu par le concepteur. Bien entendu, une syntaxe spécifique pour les actions doit être utilisée [77]. La technique des automates observateurs est utilisée par de nombreux outils (e.g. Uppaal) et permet de réduire le comportement d'un automate en le synchronisant avec un autre automate dit *automate observateur*.

Grâce à ce mécanisme de spécification, l'utilisateur peut utiliser la boite à outils IF avec n'importe quel éditeur UML pour peu qu'il puisse exporter au format XMI. Il faudra aussi importer le profil OMEGA dans son outil.

Le processus de vérification commence par une compilation de la spécification de haut niveau vers une spécification intermédiaire dans le langage IF [37] (cf. figure 3.5). Le langage IF est basé sur des réseaux d'automates temporisés [36]. A partir de cette spécification l'outil génère une spécification exécutable qui lui permet d'explorer l'espace d'états du système spécifié. Cet exécutable permet aussi de faire une vérification automatique ou une simulation interactive.

Les traces de simulation peuvent être enregistrées au format XML. En cas d'erreur de spécification détectée par le model checker, des contre-exemples sont générés au format XML. Ces contre-exemples peuvent être chargés dans une interface graphique de simulation qui permet à l'utilisateur de les rejouer sous forme de scénarios d'exécution. Cette simulation peut se faire en mode automatique avec gestion de point d'arrêt, ou en mode interactif via un panneau de contrôle, comme

[1] `http://www.irit.fr/ifx`

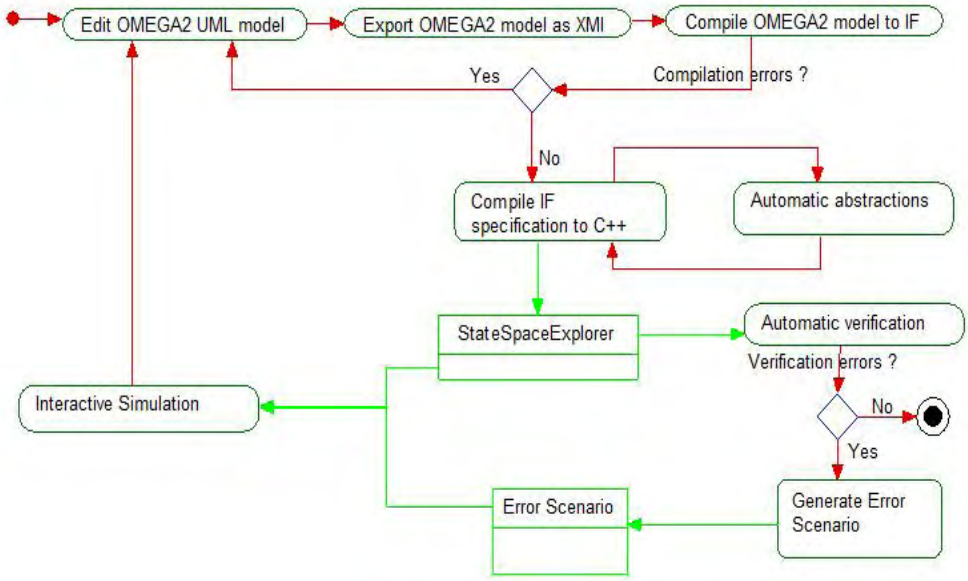


FIGURE 3.5 – Processus de validation de la plateforme IFx-OMEGA.

le montre la figure 3.6. Le processus de validation d'IFx-OMEGA a été appliqué à plusieurs cas d'étude, à titre non exhaustif, il y a lieu de citer Ariane-5 [132], MARS [133] et SGS [64].

3.3.2 vUML

vUML est un outil de validation de modèle UML basé sur une sémantique par transformation [114]. La spécification de haut niveau est exprimée dans le langage UML augmenté d'une sémantique formelle décrite dans [137]. Ces modèles UML sont ensuite traduits vers le langage PROMELA, formalisme d'entrée du model checker SPIN [32]. Après vérification par le model checker, vUML transforme les contre-exemples en diagrammes de séquences UML. Pour le diagnostic, vUML reprend ainsi l'approche implémentée sur SPIN et citée plus haut [108].

Les contre-exemples sont affichés sous forme de diagrammes de séquences (cf. figure 3.7). L'approche est présentée dans [114] sur un exemple simple, celui du dîner des philosophes [68]. Il est à noter que même sur un exemple de petite taille il n'est pas aisé de comprendre l'erreur à partir du diagramme de séquence car il n'offre pas de fonctionnalités d'exploration (e.g. visualisation des états, filtrage des processus ou messages).

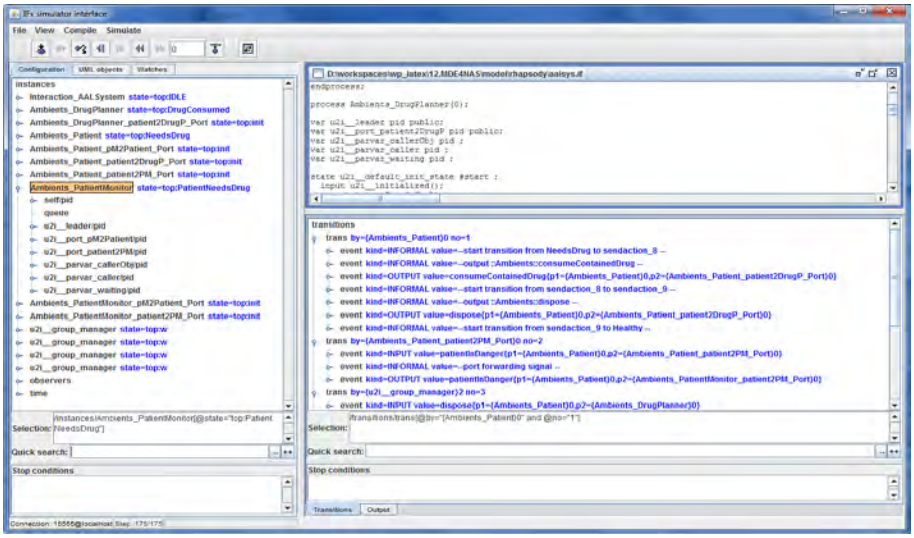


FIGURE 3.6 – Interface de simulation de l'outil IFx-OMEGA. L'interface permet le diagnostic des scénarios d'exécution en visualisant les états successifs du système sous forme d'arbre.

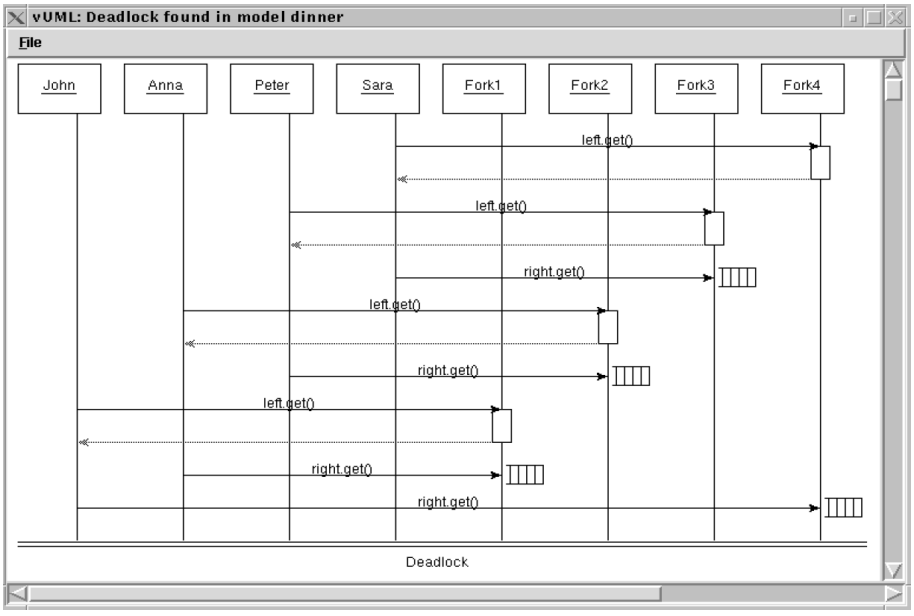


FIGURE 3.7 – Interface de visualisation des contre-exemples avec l'outil vUML à base de diagrammes de séquence UML.

3.3.3 Barber et al.

Dans [28] Barber et al., proposent un outil pour l'analyse dynamique d'architecture logicielle. Ils abordent les deux problèmes qui sont la traduction d'une architecture métier vers une spécification formelle pour faire du model checking et l'interprétation des contre-exemples par l'utilisateur.

Les auteurs proposent une approche automatisée basée sur le model checker SPIN et une traduction des résultats vers une notation appelée *Architecture Trace Diagram*. Cette représentation est très proche des notations MSC [96] ou diagrammes de séquence UML. Cette représentation reformule le contre-exemple généré par le model checker SPIN dans une sémantique proche de celle du métier. Il ne s'agit pas d'une abstraction sémantique qui serait au même niveau que le langage de départ mais seulement d'une traduction des concepts du langage Promela en termes utilisés par l'utilisateur au niveau du métier. Par exemple le transfert de donnée entre des processus Promela est remplacé par le terme *invocation de service* (familier à l'utilisateur).

3.3.4 Zalila et al.

Combemale et al. dans [61] introduisent un cadre de métamodélisation qui opérationnalise les DSML. Ce cadre permet d'introduire une approche de diagnostic des erreurs par animation des modèles de haut niveau [63]. Zalila et al. dans [160] rajoutent une extension (Query DMM) pour spécifier des requêtes TOCL sur le modèle métier et les traduire ensuite vers le formalisme de vérification de bas niveau, à savoir les réseaux de Petri. Ces travaux constituent un pas en avant vers un *diagnostic orienté métier*. L'approche permet en effet de traduire les résultats vers le formalisme métier. L'approche peut être appliquée à tout DSML pourvu d'une sémantique opérationnelle endogène. En effet l'existence d'un modèle d'exécution pour le haut niveau (e.g. xSPEM [60]) est nécessaire pour la transformation qui génère des traces d'exécution haut niveau à partir des contre-exemples générés par le processus de vérification.

3.3.5 Hegedus et al.

Hegedus et al. [82] proposent une des approches les plus élaborées (avec celle de Zalia et al.) ; elle consiste à utiliser des *transformations orientées changement* [33] pour améliorer la spécification de la transformation de retour. Les travaux présentés dans [82] sont appliqués à la traduction des modèles BPEL [129] vers les réseaux de Petri. L'idée générale repose sur le report des changements dans le modèle de bas niveau vers le modèle de haut niveau via l'utilisation d'une transformation orientée changement. L'apport principal est la réduction de la complexité

des transformations. L'approche a été testée avec succès pour la traduction des modèles BPEL vers les réseaux de Petri et elle est en théorie généralisable à tous les formalismes de type système à événements discrets.

3.3.6 RT-SIMEX

RT-SIMEX [67] [2] est un projet de recherche collaboratif financé par l'Agence Nationale de la Recherche (ANR). RT-SIMEX n'a pas pour objectif principal de développer un environnement de validation des modèles comme IFx-OMEGA. Le but principal du projet est de concevoir un socle pour la rétro-ingénierie de résultat d'exécution sur des plate-formes physiques. La partie du projet qui nous intéresse est la partie *Analyse de traces d'exécution* visible sur l'architecture globale de l'outil de la figure 3.8. Les traces d'exécution à analyser sont extraites d'une série d'exécutions sur différentes plate-formes temps réel. Un des scénarios retenu par le projet RT-SIMEX est l'exécution d'un système de vidéo surveillance. Les traces générées par l'instrumentation des plate-formes d'exécution sont transformées pour être visualisées par l'utilisateur par animation des modèles de départ (cf. figure 3.9). Les visualisations offertes par l'outil permettent de voir les résultats d'analyse directement sur le modèle utilisateur spécifié avec le profil MARTE [15]. Elles montrent les différences entre la spécification temporelle et l'exécution effective sur des plate-formes physiques. Même si l'outil n'intègre pas de fonctionnalité de validation par vérification exhaustive, il utilise les techniques de diagnostic d'annotation des modèles sources.

3.3.7 Barringer et al.

Baringer et al. proposent dans [30] un langage dédié (DSL) appelé TraceContract et embarqué dans le langage de programmation Scala [12].

Tracecontract implémente un ensemble de primitives de la logique temporelle. Ce langage peut être utilisé pour analyser les contre-exemples [29] mais n'offre pas de cadre général pour le diagnostic. En particulier, cette approche ne couvre pas les préoccupations du rendu des résultats d'analyse à l'utilisateur. En effet, après analyse le résultat est affiché de manière textuelle (cf. figure 3.10). Cette approche est comparable à celle de Zalila et al. qui utilise un formalisme (TOCL) qui est toutefois moins flexible qu'un langage fonctionnel objet comme Scala.

[2] `http://www.rtsimex.org/`

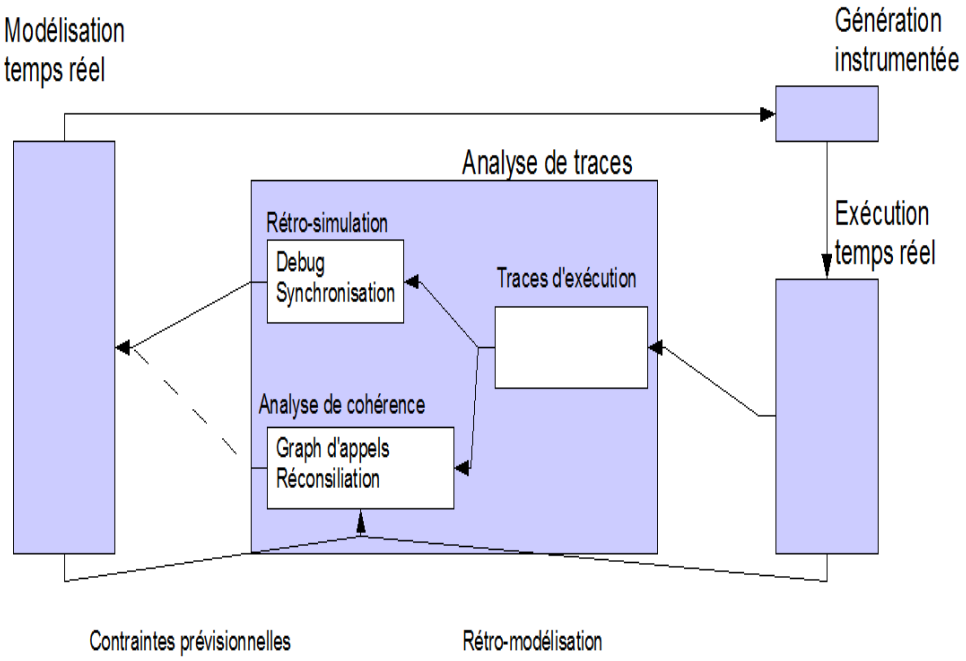


FIGURE 3.8 – Architecture fonctionnelle de l'outil RT-SIMEX[8]

```
*** Safety error:
  Monitor: CommandRequirements
  Property 'SS violated
  Violating event number 4:
    SUCCESS(STOP_DRIVING)
  Error trace:
    3=SUCCESS(STOP_DRIVING)
    4=SUCCESS(STOP_DRIVING)

*** Liveness error:
  Monitor: CommandRequirements
  Property 'CS violated - missing event
  Error trace:
    2=COMMAND(TAKE_PICTURE)
```

FIGURE 3.10 – Résultat d'analyse d'une trace de simulation avec le DSL Trace-Contract [29]

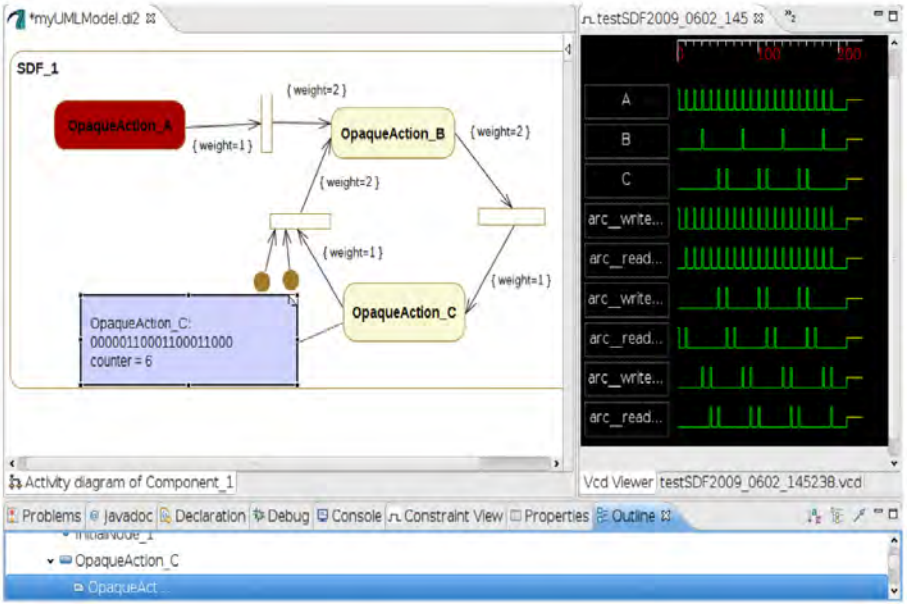


FIGURE 3.9 – Interface de diagnostic de l'outil RT-SIMEX. Le diagnostic des erreurs dans les traces d'exécution se fait par animation des modèles sources (ici un digramme d'activité).

3.4 Synthèse et discussion

3.4.1 Synthèse des approches de diagnostic

Une avancée considérable des outils de model checking a permis leur intégration à des processus d'ingénierie dirigée par les modèles. Avec l'avènement des techniques d'IDM cette intégration a pris un pas en avant en systématisant et outillant cette intégration. Un des patrons d'intégration les plus utilisés est l'intégration par traduction d'un langage de haut niveau.

Cette intégration permet une mutualisation des développements et de la maintenance des outils d'analyse, ce qui apporte un avantage indéniable pour les aspects financiers des processus d'ingénierie système. Les outils open-source comme par exemple SPIN [10] ou NuSMV2 [6] sont un levier considérable pour les processus de vérification par traduction. Sur le plan de l'approche, il est naturel de vouloir réutiliser les efforts de recherches ainsi que les résultats dans le domaine de l'analyse de modèle. Par exemple la communauté du model checking a réalisé des avancées considérables et les résultats sont applicables à un très grand nombre de systèmes dynamiques.

Pour donner encore plus de légitimité à l'approche par traduction beaucoup de travaux récents se focalisent sur la préservation de la sémantique pendant la traduction qui est un gage que ce que l'on a vérifié correspond bien à ce que l'on a spécifié. Ceci est vital dans le processus de certification des outils de vérification et de génération de code. Si une preuve a été formalisée, nous pouvons dire que les 2 niveaux de spécifications (e.g. OMEGA) et la spécification formelle de bas niveau (e.g. IF) sont sémantiquement équivalent (notion de bisimulation forte), mais ils ne seront pas *cognitivement équivalents*. Par équivalence cognitive nous signifions que la charge cognitive (i.e. l'effort utilisateur nécessaire) induite pour comprendre la même spécification dans les deux niveaux est loin d'être la même. Il en découle que la vitesse et la précision avec lesquelles l'utilisateur traite l'information présentée est différente. Cette mesure, appelée *efficacité cognitive* [110] est détaillée dans la section problématique 2.

Aucun des outils que nous avons cité n'intègre de mécanisme pour répondre de **manière systématique** à ce problème de gap cognitif introduit par l'approche de sémantique par traduction. Ce gap peut être considérable, par exemple dans les approches qui traduisent des modèles UML vers des formalismes algébriques de type CSP comme [21]. Ce problème d'adéquation cognitive se traduit en problème d'utilisabilité au niveau des interfaces graphiques de l'environnement intégré de modélisation. Bien que des travaux [145, 82, 160] identifient ce problème, aucune solution élaborée n'a été développée en prenant en compte les aspects cognitifs chez l'utilisateur. En, effet, l'accent est mis sur les aspects techniques, l'aspect

cognitif humain n'étant que marginal.
Le tableau 3.1 compare les différentes techniques de diagnostic décrites dans ce chapitre.

	Spécification		Vérification		Techniques de diagnostic (cf. section 3.1)			
Outils et contributions	**Formalisme**	**Assistance utilisateur**	**Formalisme de modèle**	**Assistance**	**Annotation**	**Animation**	**Visualization**	**Extension**
Uppaal [31]	Réseaux d'automates temporisés	GUI	TCTL	GUI	✓	✓	×	×
SMV [119]	Langage SMV	×	CTL	×	N/A	×	×	×
NuSMV 2 [52]	Extension du langage SMV	✓	CTL/LTL	✓	N/A	×	×	×
CPN Tools[97]	Petrinet colorés	✓	CTL	✓	N/A	✓	×	✓
SPIN [85]	PROMELA	×	PLTL	×	N/A	×	✓	×
IFx-OMEGA [130]	OMEGA	✓	OMEGA	✓	×	×	×	×
vUML [114]	Formal Statecharts	×	Aucun	×	×	×	✓	×
Barber et al. [28]	UML	✓	Aucun	×	×	×	✓	×
Zalila et al. [160]	Spécifique au DSML	✓	TOCL	✓	×	✓	×	✓
Hegeus et al. [82]	BPEL	✓	Petrinet	×	✓	✓	×	×
RT-SIMEX [67]	MARTE	✓	MARTE	✓	✓	✓	✓	×
Barringer et al. [30]	TraceContract DSL	✓	TraceContract	✓	×	×	×	×

TABLE 3.1 – Tableau synthétique comparant les différentes techniques de diagnostic décrites dans ce chapitre.
Assistance : tout mécanisme d'assistance de l'utilisateur, par exemple une interface de spécification avec vérification de syntaxe

Chapitre 4

Contributions

Notre contribution sera proposée en trois volets :

- **un ensemble de recommandations** pour la mise en œuvre d'outils d'analyse cognitivement efficace ;
- **un processus** qui intègre ces recommandations et offre une approche pratique pour l'application itérative des recommandations proposées [18] ;
- **une chaîne d'outils génériques** qui permet d'implémenter des outils d'analyse ou d'étendre des outils de modélisation avec des fonctionnalités d'analyse[19].

Ce chapitre est organisé de la manière suivante : après une présentation de l'information de diagnostic nous proposons une série de **recommandations pour la conception d'outils de diagnostic** basées sur des travaux fondateurs dans plusieurs domaines (psychologie cognitive, visualisation d'informations, etc.). Nous proposons ensuite un **processus et un outillage** pour permettre l'implémentation d'outils qui intègrent cet ensemble de recommandations support. Enfin, dans la partie évaluation nous proposons un **protocole de validation** des outils produits par le processus. La partie support outillé est définie dans une démarche IDM et repose sur notre framework Metaviz [18].

L'approche que nous avons suivie est orientée utilisateur et préconisée dans la norme ISO 9241-210 [92]. Nous avons posé comme but général de produire une approche qui met l'accent sur l'utilisabilité, définie dans [90] comme : *un système est utilisable s'il permet à l'utilisateur de réaliser sa tâche dans un contexte précis avec efficacité, efficience et satisfaction.*
Cette définition donne en fait une mesure de l'utilisabilité avec trois facteurs qui sont :

- l'efficacité : la rapidité de l'utilisateur dans l'exécution d'une tâche spécifique ;
- l'efficience : l'ensemble des ressources consommées par l'utilisateur pour réaliser la tâche (e.g. le temps) ;
- la satisfaction : qui est une mesure subjective de l'impression de l'utilisateur.

4.1 Principes pour le diagnostic de modèle

4.1.1 Le diagnostic : définitions et positionnement

Le vocabulaire standard ISO/IEC/IEEE de l'ingénierie des systèmes et du logiciel [95] définit le **diagnostic** comme étant : **la détection et l'isolation d'erreur ou de panne.** Il définit aussi **un manuel de diagnostic** comme étant un document qui donne les informations nécessaires à l'exécution de procédures d'identification des erreurs et de leur corrections. Enfin **un outil** est un logiciel qui fournit le support pour un cycle de vie système ou logiciel. De ces définitions standard, nous pouvons déduire une définition pour un outil de diagnostic :

Un outil de diagnostic est un logiciel qui fournit un support pour le diagnostic. Il fournit donc les informations nécessaires à l'exécution de procédures automatiques ou semi-automatiques d'identification des erreurs et à leur correction.

4.1.1.1 Profil de l'utilisateur

Dans la partie problématique, nous avons vu que la source du problème est due à la non prise en compte de spécificités dans le profil de l'utilisateur (e.g. pas de familiarité avec la notation de bas niveau).
En effet, donner une sémantique d'exécution formelle au modèle permet de valider la conception de l'utilisateur de manière automatique en utilisant les algorithmes de model checking sur cette sémantique formelle. Mais dans le cas de discordances entre la spécification utilisateur et le modèle, l'utilisateur n'est pas en mesure de comprendre les retours de l'outil. Ces retours sont très souvent exprimés dans une notation de bas niveau. L'utilisateur n'étant pas expert sur la sémantique de bas niveau, il est bloqué dans le processus et ne pourra pas corriger ces modèles.

En réalité nous sommes en présence d'un profil **utilisateur expert dans une notation de haut niveau mais novice dans la notation de bas niveau utilisée par l'outil d'analyse.**

4.1.1.2 Modélisation de la tâche de diagnostic

Il est important de comprendre le but que l'utilisateur cherche à atteindre via l'utilisation d'un outil. Ceci permet de développer des interfaces de diagnostic qui permettent à l'utilisateur d'accéder à l'information nécessaire pour réaliser une tâche. Dans notre contexte l'utilisateur veut trouver la cause de l'erreur, rapportée par l'outil d'analyse (e.g. un contre-exemple). Ceci correspond au but principal que l'utilisateur voudrait atteindre via l'utilisation d'un outil de diagnostic. Ce but se décompose en sous-tâches de la manière suivante :

- comprendre la trace ou une partie, i.e. avoir une vue globale du scénario (e.g. le contrôleur électronique d'une valve lui envoie des messages et la valve y répond en changeant d'état, elle passe par exemple d'un état ouvert à un état fermé ;
- comprendre l'erreur embarquée dans la trace (e.g. la valve a reçu deux messages d'ouverture dans un intervalle de moins de 5s) ;
- comprendre la cause de l'erreur. Comprendre les liens de causalité entre un état corrompu du système et les causes directes voire indirectes difficiles à trouver.

4.1.1.3 Information recherchée par l'utilisateur

Pour réaliser les tâches désignées précédemment, l'utilisateur a besoin d'un ensemble d'informations. Pour définir cette information, il faut trouver les questions que se pose l'utilisateur pour la réalisation de chaque sous-tâche. **Comprendre le résultat** renvoyé par les outils d'analyse, qui dans notre cas est une trace suppose de comprendre :

- quelles sont les entités du système qui participent au scénario de la trace ?
- quelles sont les interactions entre ces entités ?

Pour **comprendre l'erreur** rapportée par la trace, il faut comprendre :

- la configuration du système qui correspond à un état non désiré ;
- le détail de cette configuration, i.e. les états respectifs des entités participantes.

Et pour **comprendre la cause de l'erreur** rapportée dans la trace, il faut comprendre les liens de causalité entre des événements d'intérêt.

Nous en déduisons que l'information à mettre à disposition de l'utilisateur doit porter sur :

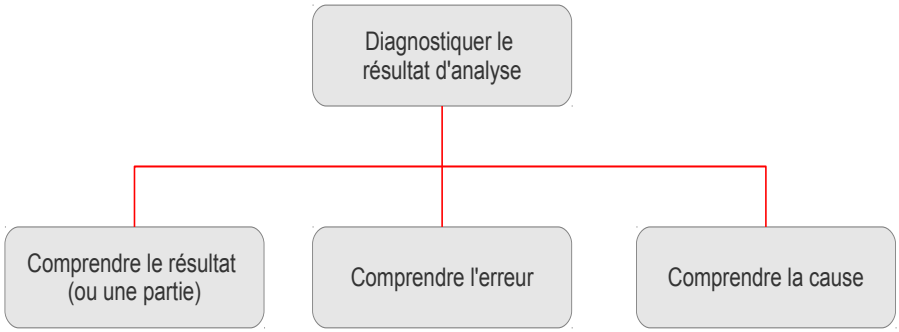


FIGURE 4.1 – Hiérarchie des tâches à exécuter par l'utilisateur

- les **configurations du système** : l'ensemble des objets et leurs propriétés (e.g. états, valeurs des variables) ;
- le **scénario** : les transitions exécutées qui permettent au système de passer d'une configuration à l'autre.

Cette information peut être rassemblée dans la notion de **trace**. Une trace est définie comme une suite de paires *(transition exécutée, configuration cible)*. La trace donne une information sur les transitions extraites des scénarios et une information sur les objets activés et les messages échangés extraits des configurations successives du système.

4.1.1.4 Métamodèles utilisés dans le processus de diagnostic

L'information nécessaire à l'utilisateur dans le contexte de diagnostic est représentée par un ensemble de métamodèles. Dans notre cas, ces métamodèles sont inspirés de la définition de la sémantique opérationnelle du langage IF [36].

Métamodèle des configurations du système Un système passe par une suite de configurations successives représentant l'ensemble des objets activés et leurs pro-

priétés. Le métamodèle (cf. figure 4.2) *IFConfiguration.ecore* représente cette information.

Métamodèle des scénarios Ce métamodèle (cf. figure 4.3) représente la séquence de transitions exécutée durant un scénario de simulation ou un contre-exemple. Il est utilisé pour injecter (i.e transformer vers l'espace technologique IDM [100]) les scénarios rapportés par le model checker.

Métamodèle de trace Pour représenter l'information nécessaire à la tâche de diagnostic, un métamodèle de trace a été développé. Ce métamodèle a pour vocation de représenter les informations relatives aux configurations successives d'un système et les transitions exécutées. Une trace est donc représentée par un ensemble de couples (Transition, Configuration) représenté dans la figure 4.4 par la classe **TConfig**. Ce couple représente la transition exécutée (**Transition**) ainsi que la configuration du système qui en résulte. Nous appelons cette configuration, configuration cible (*Taget Configuration*), elle est représentée par la classe **IFConfig**. Cette dernière classe représente la configuration du système à un instant donné, elle fait partie du métamodèle de configuration.

4.1.2 Recommandations pour la conception d'interfaces cognitivement efficaces

Cette section discute les modèles et théories fondateurs dans la notation graphique, la visualisation graphique et l'ergonomie des interfaces graphique. Le but est de les prendre comme base pour extraire un ensemble de recommandations pour la conception d'outils de diagnostic efficaces.

4.1.2.1 Physics of Notation

Dans son article fondateur sur les notations logicielle, Daniel Moody [126] apporte une réponse au problème du codage de l'information nécessaire à l'utilisateur pour effectuer une tâche de manière efficace. L'efficacité est à comprendre au sens d'efficacité cognitive. Nous pouvons la mesurer avec la vitesse et l'effort perceptif et cognitif avec lesquels l'utilisateur accomplit la tâche et l'effort perceptif et cognitif nécessaire à sa réalisation. Le but de ces travaux est donc de proposer un ensemble de recommandations pour optimiser les notations logicielles pour un meilleur traitement par l'humain. Il est aujourd'hui largement accepté que la notation graphique permet d'exploiter la puissance de notre système de perception d'une manière plus grande qu'une

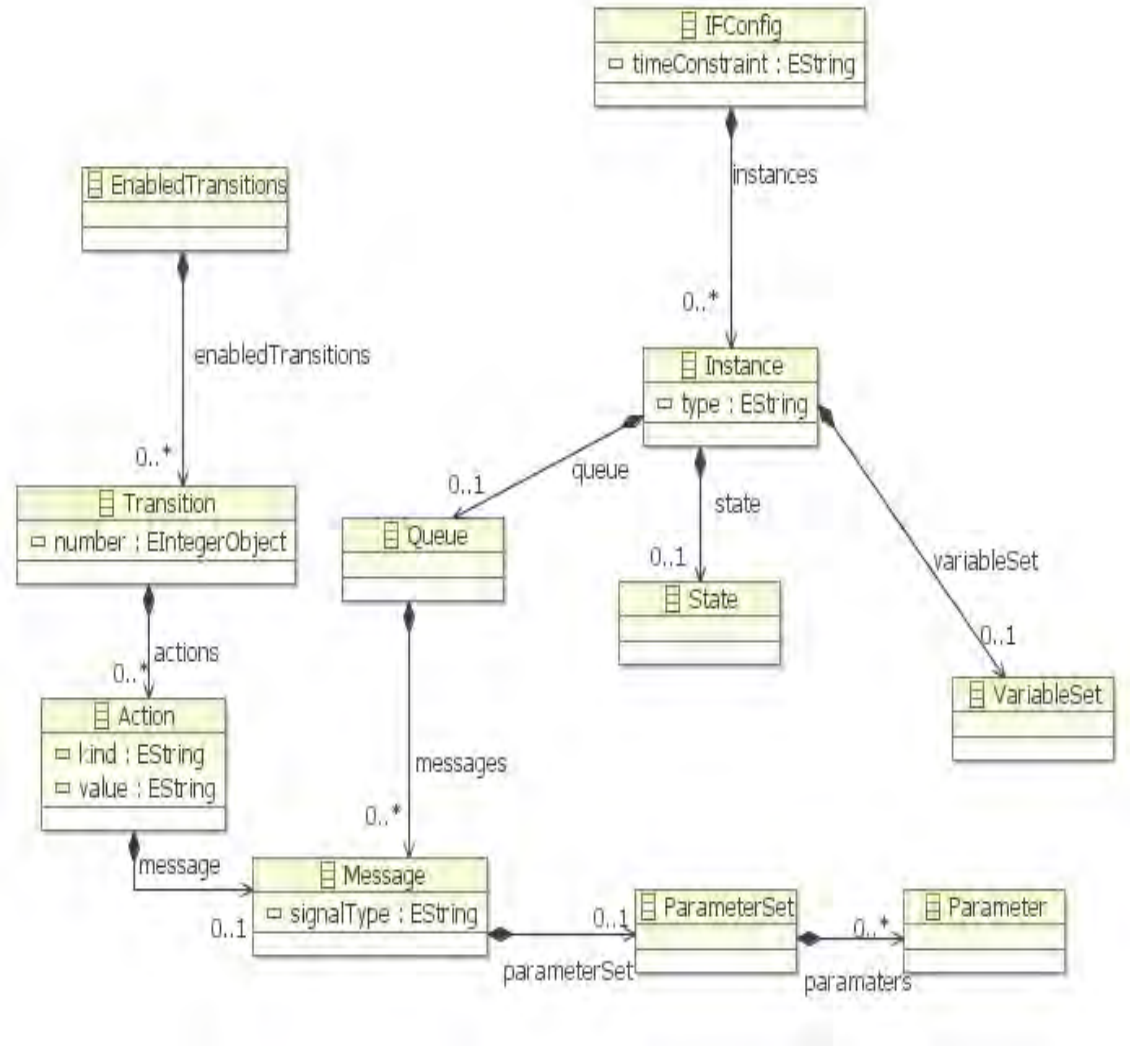


FIGURE 4.2 – Extrait du métamodèle des configurations du système, utilisé pour injecter les configurations successives envoyées par le simulateur à l'interface de diagnostic.

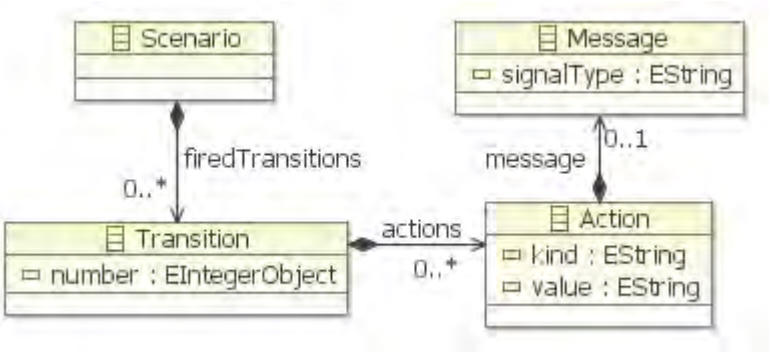


FIGURE 4.3 – Métamodèle des scénarios, utilisé pour injecter les configurations XML renvoyées par l'explorateur de l'espace d'état (le simulateur) IF.

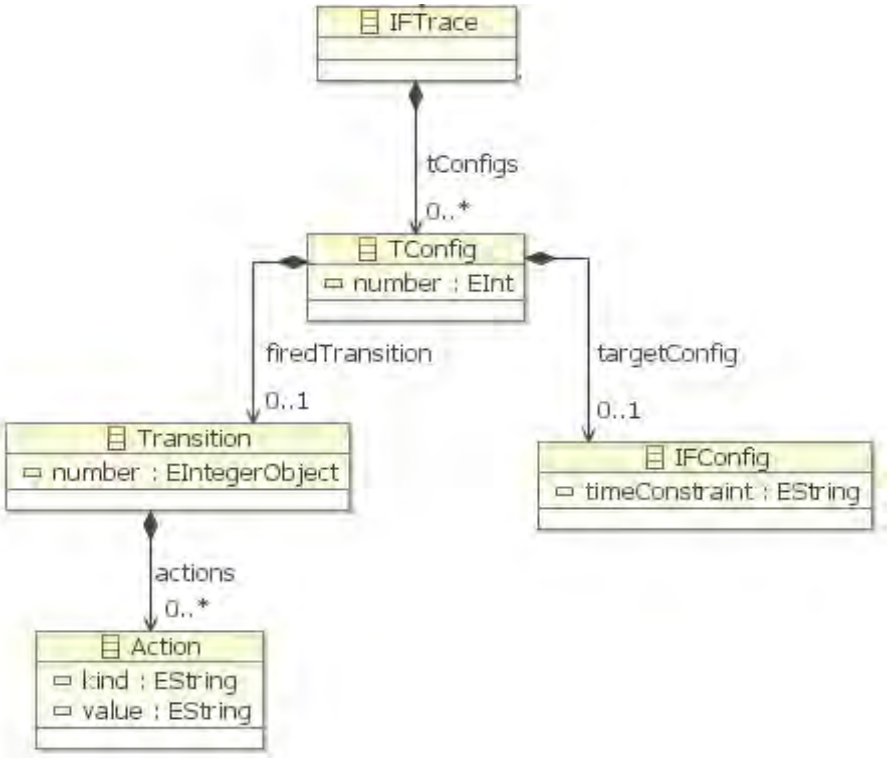


FIGURE 4.4 – Métamodèle de la trace, utilisé pour injecter les traces XML rapportées par le model checker IF.

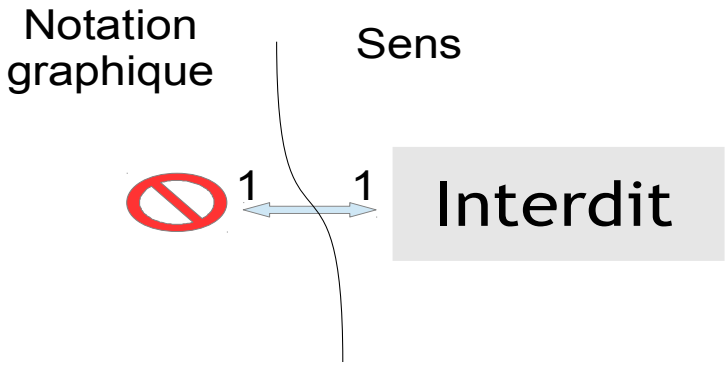


FIGURE 4.5 – Principe de clarté sémiotique : un élément graphique dénote un sens unique (ici la notation graphique de l'interdiction).

notation textuelle [110]. Exploiter les possibilités des mécanismes de perception chez l'humain nécessite d'encoder l'information à communiquer de manière précise. Cela revient à donner des valeurs bien choisies à l'ensemble des variables visuelles définies par Bertin dans ses travaux fondateurs sur la sémiologie graphique [34]. En effet Bertin définit huit variables qui correspondent aux dimensions de l'espace de conception des notations visuelles. Il s'agit de la position, de la taille, de la valeur (ou luminosité), du grain (répétition d'un motif), de la couleur, de l'orientation spatiale et de la forme. Ces variables peuvent être utilisées pour caractériser ou pour construire n'importe quelle notation graphique. Moody, propose alors un ensemble de principes qui vont permettre au concepteur de contraindre ces variables et de produire des notations ou visualisations effectives. Ces principes sont décrits dans la suite.

Clarté sémiotique Ce premier principe, emprunté à la sémiologie graphique de Bertin, stipule que les éléments visuels d'une notation doivent avoir un sens univoque. Un élément ne doit pas dénoter plusieurs éléments sémantiques. Et inversement, il ne doit pas y avoir plusieurs éléments graphiques pour dénoter le même sens (cf. figure 4.5 pour un exemple).

Discrimination perceptive Ce principe fait référence à l'utilisation des variables de Bertin. La facilité de différenciation entre deux éléments d'une notation augmente avec le nombre de variables visuelles utilisées. Moody propose une mesure de la

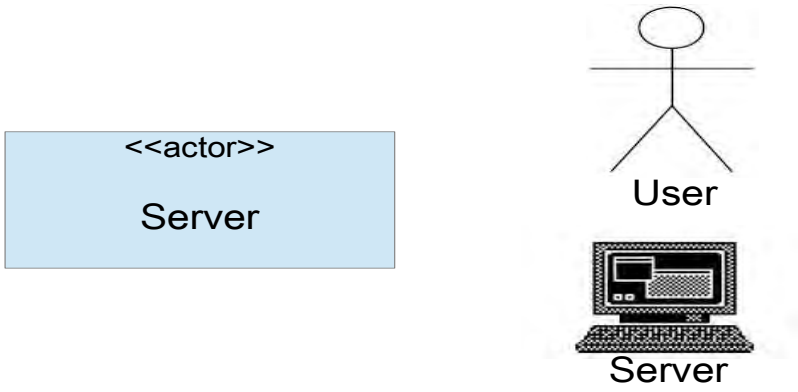


FIGURE 4.6 – Principe de transparence sémantique : le sens doit apparaître facilement à partir de la forme visuelle. Dans UML l'acteur dénote un rôle joué par une entité externe au système modélisé, l'utilisation d'une image (à droite) permet de comprendre facilement la nature de l'acteur.

distance visuelle entre ces éléments par le nombre des variables visuelles. La forme est à utiliser en priorité puisqu'elle est la plus discriminante des 8 variables de Bertin.

Transparence sémantique Les éléments graphiques d'une notation doivent avoir une **sémantique immédiate**. Le sens doit apparaître facilement à partir de la forme visuelle. Un bon exemple de l'application de cette recommandation est la notation d'acteur dans UML (cf figure 4.6). Cependant, cette notation peut être déroutante dans le cas d'acteurs non humains. Pour cela la spécification propose d'utiliser une notation rectangulaire avec un stéréotype *actor* ou une icône.

Gestion de la complexité Toute notation graphique doit proposer un mécanisme explicite de gestion de la complexité des diagrammes. La complexité se mesure simplement par le nombre d'éléments dans un diagramme. Nous avons déjà mentionné la limitation de la mémoire de travail chez l'humain. Ce principe permet de respecter cette limitation. Ce principe peut être appliqué de deux manières, soit par la modularisation des diagrammes, soit par la hiérarchisation. La première est une décomposition horizontale, i.e. au même niveau d'abstraction, la seconde est une décomposition verticale, i.e. elle étale le diagramme sur plusieurs niveaux d'abstraction permettant ainsi une exploration *top-down* qui à son tour favorise la compréhension [128]. Par exemple les Statecharts fournissent un mécanisme de gestion de la

complexité via les machines à états hiérarchiques.

Intégration cognitive Les diagrammes présentés doivent s'intégrer de manière cognitive et perceptive. L'intégration cognitive [79] consiste à donner à l'utilisateur un support conceptuel pour intégrer plusieurs diagrammes hétérogènes. L'intégration perceptive ou principe de contiguïté spatiale [118] permet de mieux gérer l'attention de l'utilisateur et d'éviter l'effet d'attention partagée [47] qui diminue l'efficacité d'une représentation si l'intégration de l'information est laissée à la charge de l'utilisateur. Ce principe, est l'un des principes à appliquer en priorité pour diminuer la charge cognitive.

Expressivité visuelle Ce principe fait référence aux nombre de variables visuelles de Bertin utilisées par la notation. Plus ce nombre est élevé plus la notation est expressive. Une notation expressive permet aussi de réduire la charge cognitive.

Double encodage Cette recommandation traite de la bonne utilisation du texte dans une notation graphique. Elle est basée sur la théorie du double encodage [136]. Ce principe permet de rendre l'encodage de l'information plus élaboré, ce qui en facilite la compréhension.

Économie graphique Ce principe stipule que le nombre de symboles utilisés par une notation graphique doit prendre en compte la charge cognitive induite sur l'utilisateur et ménager sa limitation de perception [121]. Il est donc à la charge du concepteur de trouver le bon équilibre entre ce principe et celui de l'expressivité visuelle.

Adéquation cognitive Ce principe est l'un des plus négligés dans les outils actuels. Il stipule que la notation dépend essentiellement de deux facteurs : le profil utilisateur et la tâche qu'il exécute. En effet le niveau d'expertise influe sur la perception des diagrammes. D'une part l'expert différencie plus facilement les symboles de la notation [38] et gère mieux la complexité par le système de *regroupement* [104]. D'autre part, le novice doit maintenir en mémoire de travail la signification des symboles [158]. Ces différences doivent être prises en compte par les concepteurs de la notation, pour produire différentes variantes d'une notation (appelées aussi dialectes [126]).

Dans le contexte de nos travaux, l'utilisateur a un profil de novice dans la notation utilisée par l'outil d'analyse. Il est donc primordial d'extraire un sous ensemble de

recommandations qui permettront une optimisation de la notation pour le novice, ce sont ces recommandations que nous appliquerons en priorité. Ce sous ensemble de recommandations prend aussi en considération les choix de conception des outils d'analyse actuels (décrits dans la partie état de l'art 3). Il n'est cependant pas possible d'optimiser la notation pour tous les types de profils, *l'effet de renversement dû à l'expertise* [103] stipule qu'une notation optimisée pour le novice risque d'entraver la réalisation des tâches chez un utilisateur expert. Le but de ces recommandations sera largement centré sur la réduction de la charge cognitive induite par la nature des informations rapportées par l'outils d'analyse.

4.1.2.2 Critères ergonomiques

La norme ISO 9241-110 :2006 [91] et la norme française AFNOR Z67-133.1 [74] définissent un ensemble de critères d'ergonomie qui permet d'évaluer et de concevoir des interfaces ergonomiques. Ces critères sont les suivants :

- **compatibilité** : l'interface du système doit être adaptée au profil de l'utilisateur. Ce qui rejoint le principe d'adéquation cognitive décrit ci-dessus 4.1.2.1 ;
- **homogénéité** : l'interface doit être homogène dans son apparence, mais aussi dans la logique d'utilisation ;
- **guidage** : donner à l'utilisateur des moyens pour se situer dans la logique de navigation de l'interface. Ce critère est similaire au critère d'intégration cognitive chez Moody ;
- **souplesse** : l'interface doit pouvoir s'adapter à plusieurs modes d'exécution d'une même tâche ;
- **contrôle explicite** : offrir un mécanisme à l'utilisateur pour lui permettre de garder le contrôle sur le système même pendant les tâches de fond ;
- **gestion des erreurs** : inclure dans le système un mécanisme de gestion des erreurs. Les messages d'erreurs doivent être facilement compréhensibles et leur correction doit être guidée ;
- **concision** : réduire la charge de perception et de mémorisation. Réduire le nombre d'étapes nécessaires pour réaliser une tâche. Ceci rejoint la recommandation *d'économie graphique de Moody.*

4.1.2.3 Visualisation d'information

La visualisation d'information est l'utilisation de représentations numériques, interactives et visuelles de données abstraites pour amplifier les capacités cognitives humaines [46]. La visualisation d'information diffère de la visualisation scientifique par l'aspect abstrait (i.e. non physique) des données représentées, comme la représentation des pays en fonction de leur produit intérieur brut. La visualisation d'information propose un ensemble de techniques qui permettent d'explorer une grande masse d'information, souvent multidimensionnelle. Pour cela elle tente d'exploiter les capacités visuelles, perceptives et cognitives chez l'humain d'une part et les capacités de représentation et de calcul des processeurs graphiques et des moyens d'affichage. Ces techniques apportent de nombreux avantages [50, 46] parmi lesquels :

- l'augmentation des capacités cognitives [110] ;
- la réduction du temps de recherche [155] ;
- la réalisation d'une partie du traitement de l'information, normalement inférée à partir d'une représentation textuelle [86].

Pour réaliser ces avantages, il est nécessaire de respecter un ensemble de recommandations [50]. Ces recommandations consistent principalement à réduire tout traitement cognitif inutile à l'exploration et la compréhension d'un espace de données. Cela passe par l'application des règles sur le fonctionnement du système perceptif humain. Par exemple les principes de la Gestalt [105] ou de la gestion de l'attention [159]. Mais il passe aussi par la prise en compte du contexte de l'utilisateur. L'étude de ce contexte doit s'intégrer dans le processus de conception des visualisations [106].

4.1.3 Recommandations pour la conception d'un système de diagnostic

Comme expliqué dans la partie problématique 2, la limitation des outils de diagnostic est essentiellement due à l'introduction d'un gap sémantique, qui se traduit par une distance cognitive chez l'utilisateur. Nous avons aussi étudié les limitations du matériel cognitif chez l'utilisateur et surtout chez le novice.

Dans les paragraphes précédents nous avons exposé un ensemble de recommandations extraites de travaux fondateurs dans plusieurs domaines. Le but de cette section est de contextualiser ces travaux pour les outils de diagnostic. Parmi ces recommandations, celles qui auront le plus d'impact sur l'utilisabilité des outils de

diagnostic sont celles qui **réduisent la complexité de l'information présentée à l'utilisateur et la charge cognitive**. La complexité renvoie à l'ensemble de l'information utilisée durant la tâche de diagnostic, information qu'il faut réduire au minimum. La charge cognitive renvoie à l'effort de perception et de mémorisation de l'utilisateur.

Il est aussi primordial lors de l'intégration de ces recommandations dans un processus commun de prendre en compte leurs interactions. En effet certaines recommandations, appliquées indépendamment des autres peuvent donner un effet inverse à l'effet escompté [126]. Par exemple les recommandations d'économie graphique et d'expressivité de la notation doivent être prises en compte ensemble pour trouver le bon équilibre entre économie et richesse de la notation. Le choix se fera en fonction du profil utilisateur et du contexte d'utilisation (i.e. la tâche). Cet ensemble de recommandations va donc répondre essentiellement à deux questions :

- quelles recommandations appliquer dans le cas du diagnostic avec un utilisateur novice dans la notation de bas niveau ?
- dans quel ordre appliquer les recommandations pour bénéficier de la meilleure efficacité cognitive (puisque des recommandations appliquées indépendamment peuvent s'annihiler) ?

4.1.3.1 Diminuer la charge cognitive

Pour cela on va utiliser la puissance du processeur perceptif afin d'alléger la charge de traitement engagée par le processeur cognitif. Nous proposons un ensemble de recommandations qui va permettre de déporter une partie de l'effort de mémorisation (maintien en mémoire de travail) sur le processeur perceptif. Ce dernier processeur, contrairement au processeur cognitif, traite une très grande quantité d'information de manière rapide et automatique (sans effort).

Recommandation n° 1 : gestion de l'attention La première étape dans l'assistance au diagnostic est d'attirer l'attention de l'utilisateur vers ce qui est pertinent (i.e. nécessaire au diagnostic d'une erreur). Il faut donc gérer son attention [159], pour cela nous pouvons utiliser des couleurs (ex. changement d'état, envoie de message) ou des mouvements. Nous retrouvons ici le principe **d'expressivité visuelle** de Moody [126].

Recommandation n° 2 : syntaxe intuitive (ou à sémantique immédiate) Une fois l'attention de l'utilisateur attirée sur un symbole graphique, il faut que la

dénotation faite par l'utilisateur n'induise aucun effort de sa part. Nous parlons de *syntaxe à sémantique immédiate* i.e. l'utilisateur comprend le sens du symbole sans effort de mémoire [89]. Il faut donc avoir une syntaxe compréhensible sans effort par l'utilisateur. C'est le principe de **transparence sémantique** chez Moody. Nous pouvons utiliser une syntaxe proche de celle utilisée par le langage de haut niveau. Ce qui n'est pas toujours possible, par exemple dans les cas ou les langages de haut niveau et bas niveau sont très différents (e.g. traduction d'UML vers une algèbre de processus).

Recommandation n° 3 : contiguïté de l'information pertinente Cette recommandation découle du principe d'**intégration cognitive** de Moody et de la loi de proximité de Gestalt[105].
Elle fournit un mécanisme d'organisation du diagramme pour éviter une attention partagée entre plusieurs diagrammes et minimiser par conséquent la charge de mémorisation. Les informations pertinentes ne doivent pas être dispersées dans des diagrammes différents mais être présentées de manière contiguë. Nous pouvons soit les présenter dans un seul diagramme soit fournir un mécanisme de navigation efficace [138].

Recommandation n° 4 : persistance de l'information Une fois l'information affichée de manière compatible avec le profil de l'utilisateur, il faut s'assurer qu'elle soit persistante. En effet, il arrive que durant l'interaction de l'utilisateur une partie ou la totalité de l'information pertinente ne soit plus accessible. Ceci oblige l'utilisateur à maintenir en mémoire cette information. La rétention de l'information pertinente sur plusieurs pas de l'interaction est primordiale à la baisse de la charge cognitive.

Recommandation n° 5 : réduction de l'information La persistance de l'information risque de mener à des interfaces trop chargées. Il faut donc contrebalancer cette recommandation. C'est le but de la dernière recommandation, qui propose de réduire l'information à ce qui est pertinent. Par exemple pour les configurations du système nous pouvons afficher uniquement les objets qui ont subi un changement d'un pas de la simulation à l'autre. Le reste de l'information devra néanmoins rester accessible mais via un contrôle rapide de l'utilisateur, ceci pour ne pas aller à l'encontre de la recommandation sur la persistance de l'information.

Cet ensemble de recommandations va constituer la base du processus de conception d'outils de diagnostic. Ce processus est décrit dans la section suivante.

4.2 Processus et outils pour le diagnostic de modèle

Dans cette section[1] le but est double :

- définir un processus pour supporter les recommandations proposées dans le cadre général détaillé dans la section 4.1. Ce processus devra permettre une application itérative et un prototypage rapide des outils. Le caractère itératif du processus est très important pour permettre la réalisation d'une évaluation à chaque fin d'itération afin de mesurer l'impact de l'application de chaque recommandation ;

- proposer un outillage qui supporte ce processus. L'outillage devra permettre une implémentation orientée modèle. Les outils conçus sur cet outillage devront être facilement extensibles.

4.2.1 Processus générique de construction et de personnalisation des visualisations

Pour assister l'utilisateur dans la tâche de diagnostic, nous proposons dans cette section un processus générique. Ce processus est générique puisqu'il a vocation à s'intégrer à n'importe quel outil de validation de modèle. Ce processus est construit autour de deux notions : la notion de **tâche utilisateur** et la **visualisation d'information**. Exécuter ce processus revient à bien cerner la tâche de l'utilisateur et à en déduire un ensemble de visualisations qui va l'assister dans cette tâche. Le but de ces visualisations sera de détecter les erreurs et d'en comprendre les causes.

Le point de départ pour définir les activités de notre processus sont les travaux de Card dans [46]. Il y décrit quatre étapes principales pour la conception de visualisations (cf. figure 4.7). La première étape consiste à rassembler les données brutes (e.g. sous format textuel). Ces données seront ensuite transformées en information porteuse de sens pour l'utilisateur. Dans cette étape il s'agit de filtrer, d'agréger et de transformer les données brutes pour leur donner une forme et du sens pour l'utilisateur. Dans une troisième étape, cette information sera transformée vers une structure *visuelle* par exemple un graphe ou un arbre. Il ne s'agit pas encore de graphique visuel exposé à l'utilisateur mais juste de structures visualizable. C'est uniquement dans une dernière étape que nous rajoutons les informations nécessaires

[1] Les travaux présentés dans cette section ont fait l'objet de deux publications [18, 19]

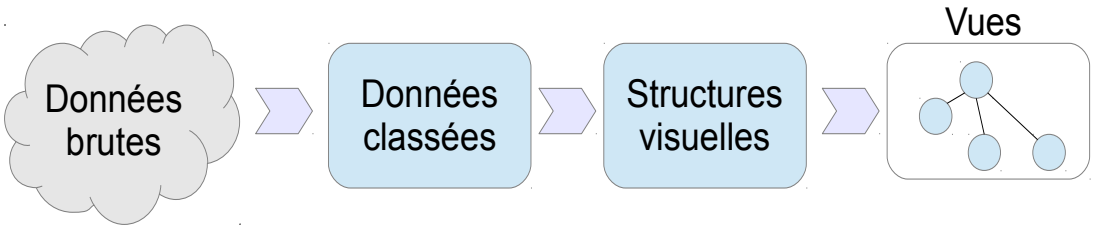


FIGURE 4.7 – Modèle de référence pour la visualisation d'information [46]

à la visualisation de ces structures. Comme par exemple les positions des objets via l'exécution d'un algorithme de positionnement.

Nous venons d'identifier quatre états de la donnée dans le pipeline de visualisation. Ces étapes sont en réalité une caractérisation des visualisations que nous allons utiliser pour la construction de nos visualisations.

4.2.1.1 Processus orienté utilisateur pour la construction de visualisation de diagnostic

Pour être efficaces, les visualisations doivent être construites à partir d'une bonne compréhension de la tâche de l'utilisateur ainsi que de son profil [116, 24]. C'est dans cet esprit que notre processus de conception de visualisation commence par une activité de définition et de compréhension de la tâche utilisateur. Les étapes de ce processus sont définie ci-dessous, son application sera décrite dans la section *application du processus* 4.3.

Analyse de la tâche : la tâche principale est le diagnostic. Cette tâche se décompose selon le type d'erreur à diagnostiquer en différentes sous-tâches. Par exemple, une des erreurs récurrentes dans les modèles OMEGA est l'oubli (dans les machines à états) de consommation de tous les messages envoyés à un objet. La tâche utilisateur dans ce contexte sera d'*explorer les envois et réceptions de messages par les objets.*

Définition d'une stratégie d'amélioration : une fois la tâche définie, il faut

comprendre les raisons de l'inefficacité de l'utilisateur dans son contexte. Cette étape est primordiale, et consiste à utiliser les recommandations détaillées plus haut (4.1.3) pour comprendre l'échec ou les causes de performances dégradées de l'utilisateur. Ensuite il s'agit de définir, grâce aux mêmes recommandations, une stratégie d'amélioration de l'interface de diagnostic.

Choix d'une technique de visualisation : le but de cette activité est de choisir dans la littérature et les outils existants une technique de visualisation qui permette d'implanter la stratégie d'amélioration définie précédemment. On pourra pour cela utiliser les taxonomies de visualisations [51, 106] pour réutiliser une ou plusieurs techniques adaptées au contexte de l'utilisateur.

Dans le cas où aucune technique satisfaisante n'a été trouvée, il faut en créer une. C'est le but de l'activité suivante de caractérisation d'une nouvelle visualisation.

Caractérisation de visualisation : cette activité consiste à concevoir et construire une nouvelle visualisation. Pour y arriver il faut tout d'abord définir l'architecture d'une visualisation d'information. Nous fournissons pour cela un framework de visualisation orienté modèle, défini dans la section 4.2.3.

Évaluation : il s'agit de mesurer l'amélioration apportée par l'exécution de ce processus. Plusieurs types d'évaluations peuvent être appliquées. Dans la section 4.4 nous discutons de ces types et proposons un protocole d'évaluation adapté au contexte de diagnostic.

4.2.2 Intégration à une plate-forme de modélisation et de vérification (IFx-OMEGA)

IFx-OMEGA permet de modéliser et de valider des système temps réel (cf. section 3). L'outil fournit un processus utilisateur bien défini (cf figure 4.8) qui prévoit une activité de *simulation interactive*. Nous retrouvons cette activité dans la plupart des outils actuels. Le processus que nous proposons s'intègre dans cette activité par raffinage. La figure 4.9 montre l'intégration du processus générique dans le processus général de IFx-OMEGA.

4.2.2.1 Outils de modélisation/validation extensible : *IFx-Workbench*

Avant de pouvoir étendre l'outil nous avons porté une partie de l'outillage IFx-OMEGA sur le standard OSGi[2].

OSGi définit un modèle à composants dynamique et orienté services, exécutable sur la machine virtuelle Java.

[2] `http://www.osgi.org`

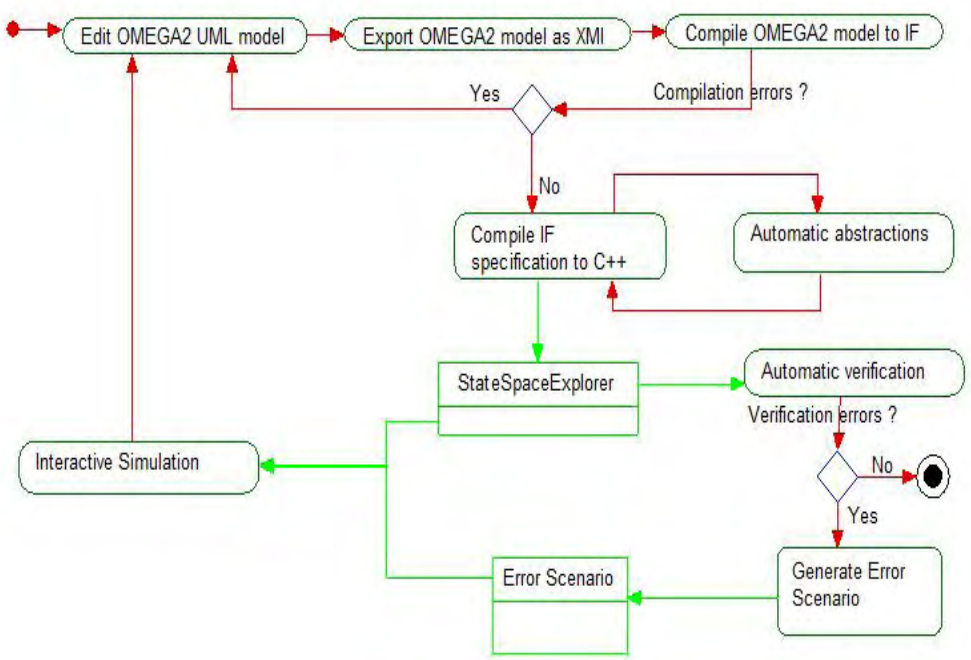


FIGURE 4.8 – Processus de validation de la plate-forme IFx-OMEGA.

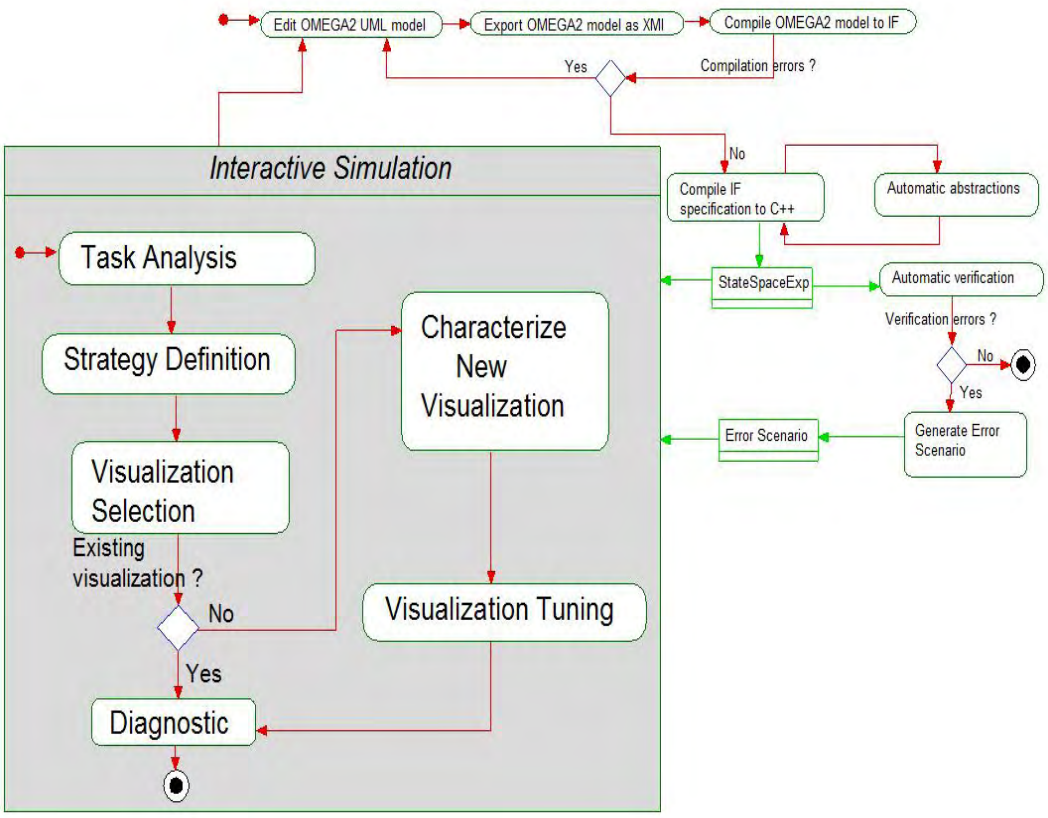


FIGURE 4.9 – Intégration du processus générique de construction de visualisation de diagnostic dans la plate-forme IFx-OMEGA.

L'utilisation de ce standard pour l'implémentation d'une partie de l'outil répond à la problématique essentielle d'extension rapide. En effet le besoin d'étendre rapidement l'outil par un ensemble de visualisations impose d'avoir une plate-forme d'intégration flexible. OSGi est la réponse la plus mature à ce besoin.

Le choix d'OSGi répond aussi au besoin de réutilisation des implémentations de référence des techniques de l'IDM (i.e. Ecore[3], ATL[1], etc...).
Cette nouvelle implémentation de l'outil IFx-OMEGA, appelée *IFx-Workbench* offre un ensemble de fonctionnalités :

- un répertoire de modèles avec des fonctionnalités de gestion (création, sérialisation, etc...) ;
- un répertoire de vues utilisateurs pour gérer les vues de diagnostic ;
- des vues de diagnostic (e.g. TreeViewers, IFConfig, IFTrace, Graphcom) ;
- des injecteurs XML pour les scénarios, les traces et les configurations ;
- un explorateur de fichiers ;
- des menus de diagnostic contextuels ;
- une interface de programmation (API) pour la gestion des transformations de modèles.

Les figures 4.11, 4.12 et 4.13 donnent un aperçu de ces fonctionnalités.

L'outillage doit être facilement étendu. Dès qu'une nouvelle erreur récurrente est répertoriée, le processus proposé permet de savoir quel type d'interface développer pour la diagnostiquer. Le rajout des artéfacts nécessaires à l'implémentation de cette interface (transformations, metamodèles et vues), est alors supporté par les mécanismes de modularité de d'IFx-Workbench.

Rajouter une visualisation dynamiquement revient à créer un nouveau *plugin* Metaviz (i.e. une chaîne de transformation d'un modèle de simulation vers un modèle de viewer) et de rajouter un menu contextuel permettant de lancer la visualisation. L'architecture fonctionnelle de l'outil est décrite dans la figure 4.10.

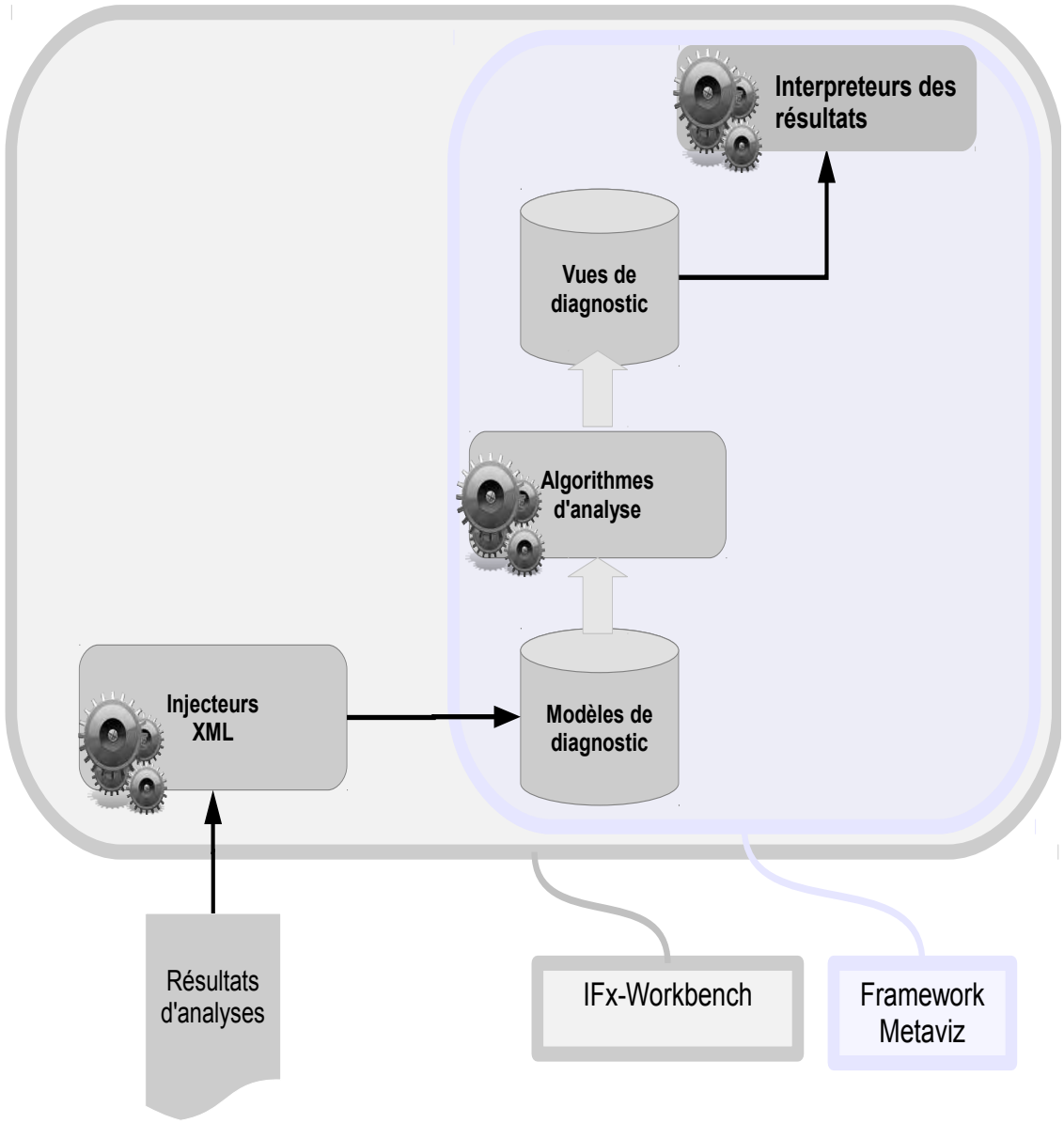


FIGURE 4.10 – Architecture fonctionnelle de l'outil IFx-Workbench. L'outil est essentiellement composé d'un ensemble d'injecteurs de fichier XML vers les modèles de diagnostic (e.g. modèle de trace) et des implémentations des visualisations sur la base du framework Metaviz.

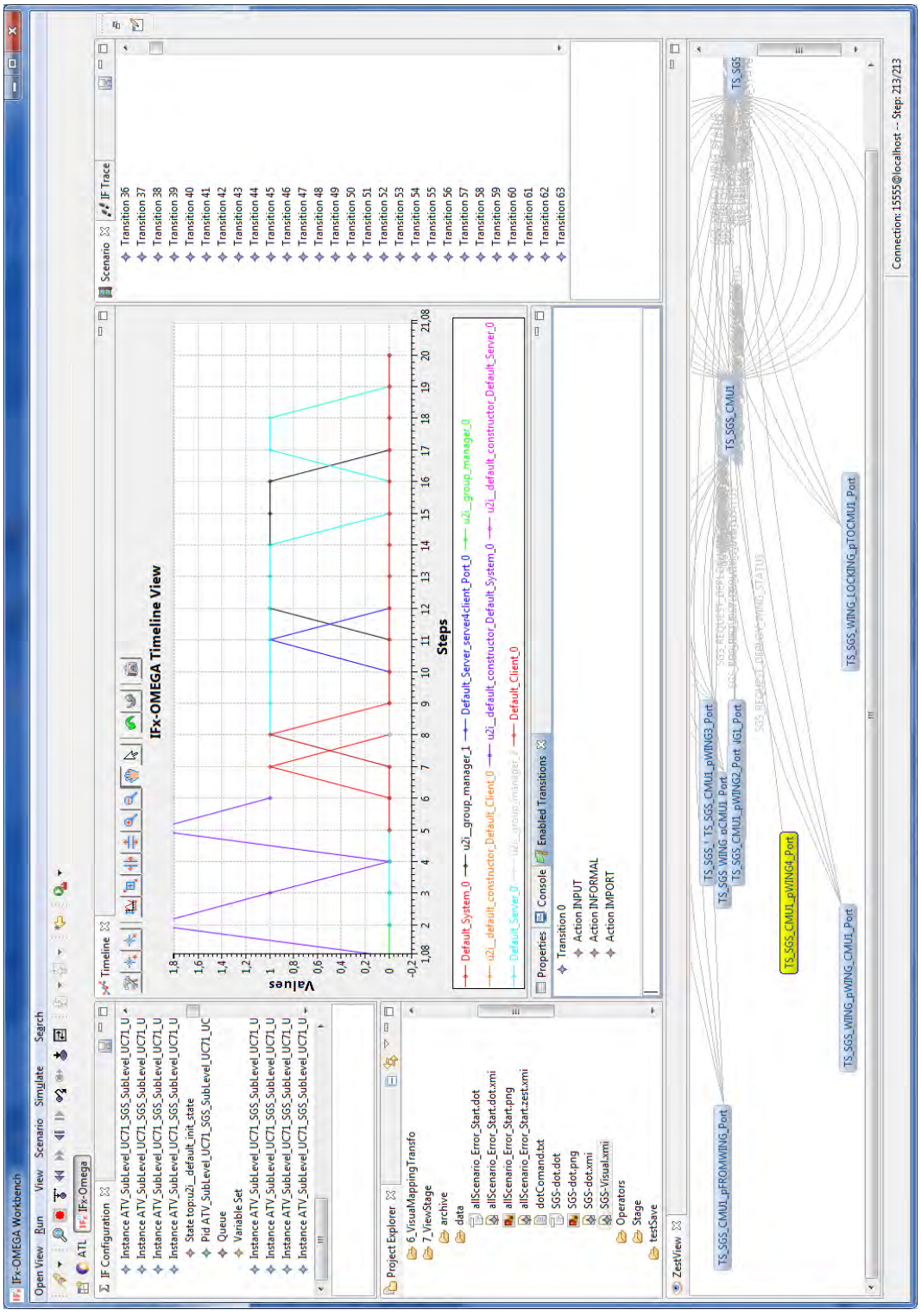


FIGURE 4.11 – L'outil *IFx-Workbench*

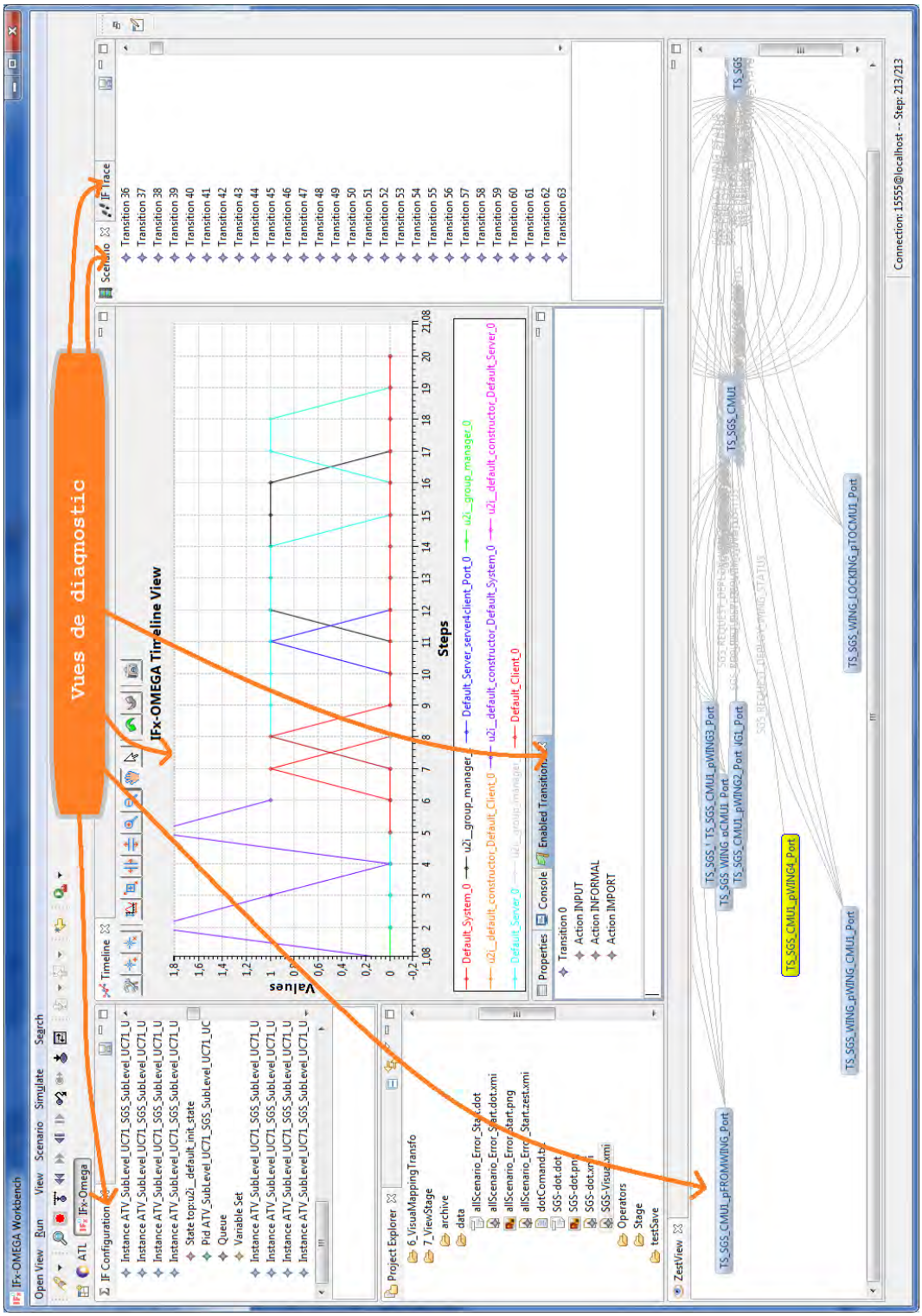


FIGURE 4.12 – Affichage des vues de diagnostic dans *IFx-Workbench*

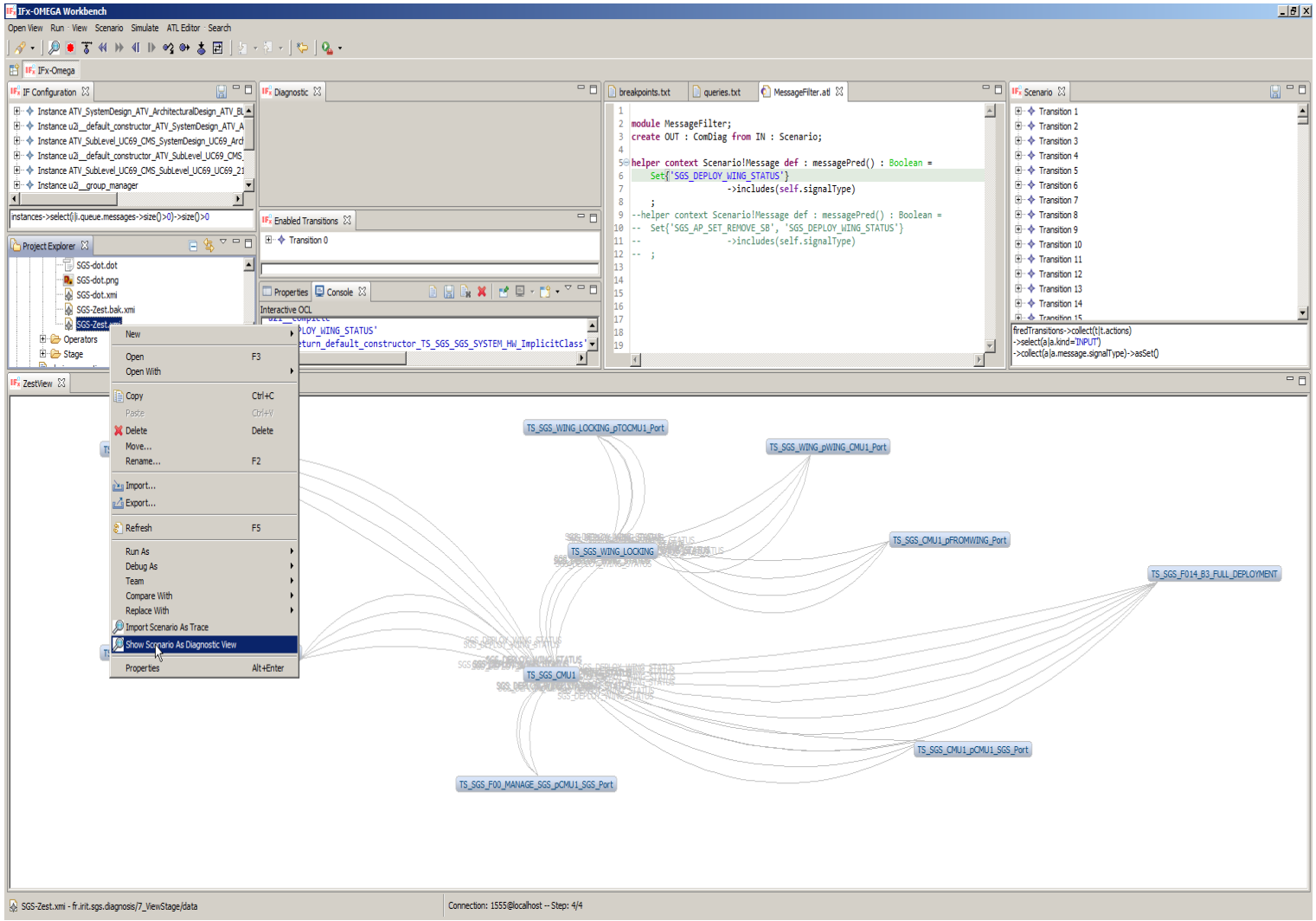


FIGURE 4.13 – Explorateur de modèle sérialisé au format XMI avec menus contextuels de diagnostic

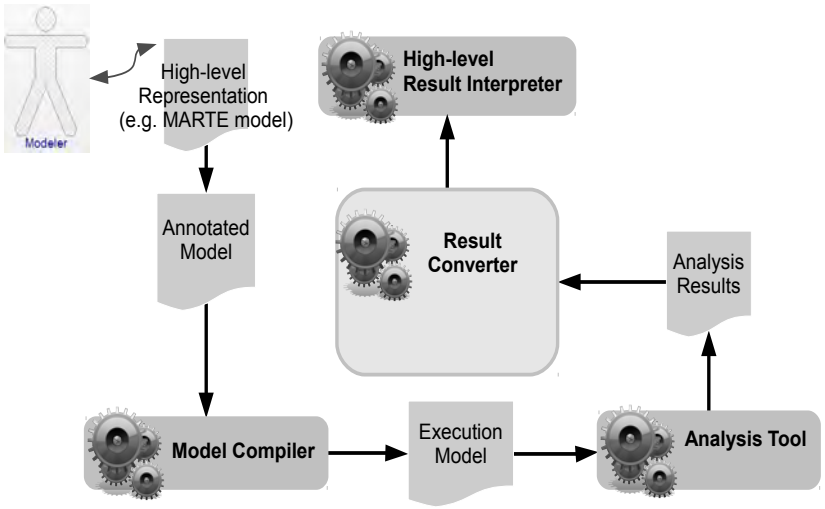


FIGURE 4.14 – Approche générique pour l'exploitation des résultats d'analyse de modèles. Les modèles annotés sont traduits vers un formalisme d'analyse, les résultats sont ensuite interprétés de manière visuelle.

4.2.2.2 Extension de IFx-Workbench avec les fonctionnalités de diagnostic par visualisation

Durant l'activité de vérification automatique l'utilisateur génère et manipule des contre-exemples et des scénarios de simulation. Ces scénarios sont exprimés dans la sémantique de bas niveau (i.e. celle du model checker IF). L'extension que nous proposons traduit l'information recherchée par l'utilisateur dans une sémantique de haut niveau. Pour étendre IFx-Wokbench avec des fonctionnalités d'analyse des résultats nous avons implémenté l'architecture générique proposée dans [7] et représentée dans la figure 4.14.

4.2.3 Framework de visualization de modèle (Metaviz)

Pour supporter le processus proposé dans la section précédente nous avons développé un framework orienté modèle : Metaviz.

Metaviz assiste l'utilisateur à partir de l'activité choix d'une technique de visualisation jusqu'à l'étape d'évaluation, pour laquelle, un protocole est défini plus loin en section 4.4. Le framework ne couvre pas actuellement l'étape de définition de la tâche qui peut être informelle (documentation) ou basée sur des approches de modélisation de la tâche utilisateur [45].

4.2.3.1 Modèles de références utilisés

Pour concevoir ce framework nous nous sommes basés sur les travaux fondateurs de Card dans [46]. Card propose un modèle d'organisation du flux de données pour la construction de visualisations : le *Data Reference Model (DRM)*.

Plusieurs modèles de référence ont depuis été proposés (e.g. [116, 51]), la plupart sont une variante du modèle de flux de données proposé par Card.

Ces modèles donnent une taxonomie claire de l'espace des visualisations mais ne permettent pas une implémentation directe. En particulier ils ne proposent pas de mécanisme de séparation des préoccupations, principe classique et très efficace pour réduire la complexité des implémentations. La raison simple en est que ces modèles sont souvent proposés pour classifier les visualisations de la littérature. Il n'ont pas, par conséquent une vocation de framework d'implémentation.

Dans notre contexte nous avons besoin d'un modèle qui soit assez raffiné pour permettre une implémentation des visualisations. Le DRM caractérisant les données a besoin d'un raffinage qui caractériserait aussi les transformations successives de ces données.

Le modèle proposé par H. Chi [51] fournit ce raffinage. Ce modèle appelé **Data State Reference Model (DSRM)**, caractérise les données manipulées le long du pipeline de visualisation ainsi que les opérateurs qui les manipulent. Il est composé d'un ensemble d'**étages** qui rassemblent les données d'un niveau d'abstraction ainsi que d'un ensemble d'**opérateurs** pour transformer les données d'un étage vers un autre.

Implémenter une visualisation sur la base de ce modèle donne une vision claire sur les résultats intermédiaires des opérateurs. Il permet aussi de définir les étages de manière itérative. La séparation des préoccupations avec le système des étages et des opérateurs permet d'avoir une approche de type MDA [122, 143] rendant ainsi la réutilisation possible des étages et des opérateurs.

4.2.3.2 Architecture fonctionnelle

Les différents composants du framework Metaviz ont été conçus selon le modèle *DSRM*. Nous retrouvons ainsi les notions d'étages et d'opérateurs. Cette mise en correspondance apporte un découpage entre les données du métier relatives à l'utilisateur et celles relatives aux résultats d'analyse, permettant ainsi la réutilisation des données et des opérateurs.

L'architecture proposée consiste en une séparation des préoccupations dans le pipeline de visualisation rendu possible par la synergie entre les techniques de l'IDM et le modèle de référence *DSRM*. La figure 4.15 montre les différentes composantes

fonctionnelles du framework. Les paragraphes suivants décrivent ces composants.
L'étage des données rassemble les données de départ, à savoir les traces, les configurations successives du système à diagnostiquer et toute autre information nécessaire. Cet étage dépend du contexte de l'utilisateur (i.e. la tâche et le profil) et est défini à partir de la connaissance de ce contexte.

Les opérateurs de données manipulent les données de départ sans introduire de nouveaux types de données. Il s'agit essentiellement de fonctions de filtrage (non représentées sur la figure 4.15).

Les opérateurs d'analyse réalisent des fonctions d'analyse sur les données de départ. Les techniques d'analyse de données (analyse du graphe de communication, résumé de trace, détection de patrons, etc...) sont implémentées avec ces opérateurs. Ces opérateurs constituent une librairie de techniques d'analyse réutilisables. Librairie que nous pouvons organiser avec une taxonomie comme celle proposée par Andrienko et al. dans [24] : *vue globale, abstraction, recherche de patrons, etc..*

Les étages d'analyse sont le résultat de l'application des opérateurs d'analyse aux données. Ils constituent un ensemble de structures de données réutilisables. Parmi les structures les plus utilisés il y a les graphes de communications, les patrons de communication inter-processus et les statistiques ou résumés de trace.

Les opérateurs de visualisation produisent, à partir des étages d'analyse des structures visualisables, comme les graphes ou les arbres.

Les étages de visualisation préparent les données visualisables pour un ensembles de librairies graphiques (e.g. un graphe étiqueté). Séparer cet étage de celui de l'analyse permet de réutiliser les 2 types d'étages séparément. En particulier nous pouvons réutiliser un étage de visualisation pour afficher les résultats venant de plusieurs étages d'analyse. I. Bull [41] donne un ensemble de structures réutilisables dans cet étage.
L'intérêt de cet étage est de permettre la réutilisation de données entre plusieurs outils de visualisation, ce qui équivaut à un modèle de type *Platform Independent Model* dans les termes du MDA [122].

Les opérateurs de mapping visuel permettent de rajouter les informations spécifiques à une librairie graphique donnée. Il s'agit par exemple d'informations de positionnement et de gestion des attributs visuels (au sens de Bertin [34]). Ces in-

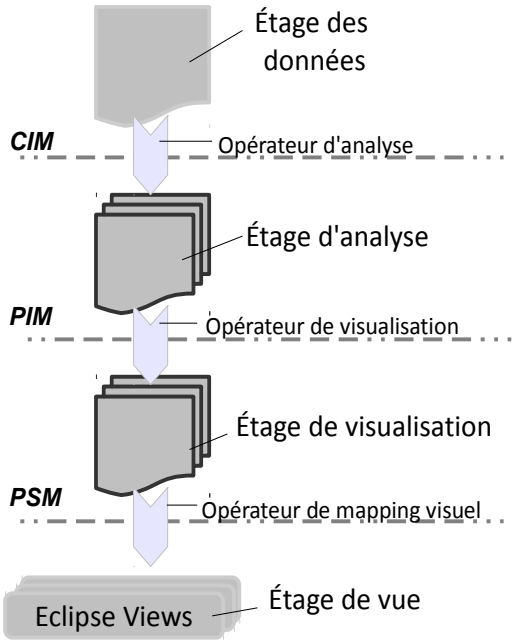


FIGURE 4.15 – Architecture fonctionnelle du framework Metaviz.
CIM (Computation Independent Model), PIM (Platform Independent Model) et PSM (Platform Specific Model) font référence aux trois modèles préconisés par l'approche MDA[122]

formations sont généralement spécifiques à la plate-forme d'exécution qui dans notre cas est la librairie graphique utilisée pour l'affichage du résultat final.

Les étages des vues cet étage est spécifique à la librairie graphique. Nous définissons pour chaque librairie ou outils utilisés (e.g. Graphviz [5]) un étage qui rassemble les données spécifiques, ce qui correspond à un modèle de type *Platform Specific Model* dans la terminologie du MDA [122].

4.2.3.3 Utilisation du framework Metaviz

Utiliser le framework, c'est réutiliser un ensemble de metamodèles (par instanciation) et de transformations. L'utilisateur peut aussi enrichir le framework avec sa propre librairie de metamodèles et de transformations.

Les étages sont implémentés avec des metamodèles Ecore [3], et les opérateurs par des transformations compatibles avec Ecore. Nous avons utilisé le langage ATL [99, 1]. L'approche déclarative des langages conforme à QVT [134] apporte une facilité d'implémentation et d'évolution des opérateurs ainsi que de leur documentation. En effet, les correspondances entre les éléments des étages source et cible d'un opérateur sont explicites dans le code de la transformation. La figure 4.16 montre la correspondance entre le modèle de référence *DSRM* et le framework *Metaviz*.

4.2.4 Conception d'outils de diagnostic par visualisation

Dans cette section, nous allons utiliser le framework Metaviz pour construire un ensemble de visualisations de diagnostic pour l'outil IFx-Workbench.

4.2.4.1 Construction de visualisations

Pour illustrer l'utilisation de notre framework, nous nous proposons de construire un ensemble d'outils dans le cadre d'une étude de cas industriel. Nous allons exécuter une partie du processus décrit dans la section 4.2.1, celle correspondante à la caractérisation des visualisations. Un exemple complet d'exécution du processus sera donné dans la section 4.3.

L'étude de cas concerne un vaisseau spatial de type cargo. Il s'agit du véhicule automatique de transfert européen ou *ATV* pour *Automated Transfer Vehicle*. L'*ATV* est développé par Astrium pour le compte de l'Agence Spatiale Européenne (ESA) et a pour but de ravitailler la Station Spatiale Internationale (ISS). La figure 4.17 montre une photo de L'*ATV* avec ses 4 ailes déployées. Nous nous intéressons plus

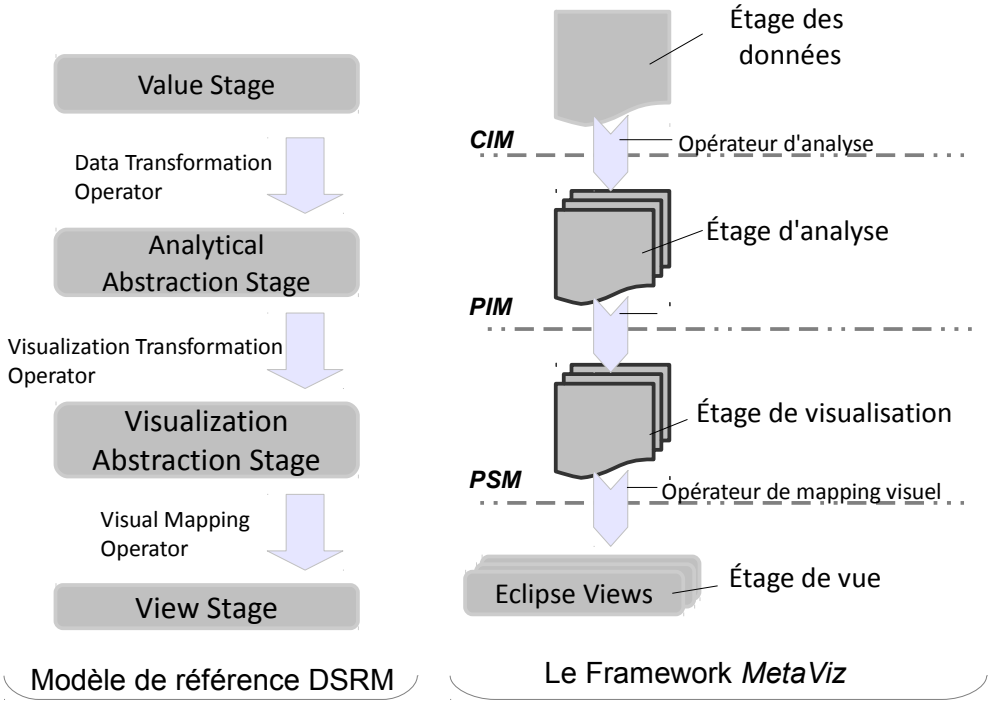


FIGURE 4.16 – Correspondance entre Metaviz et le DSRM [51]

FIGURE 4.17 – Le Jules Verne, le premier ATV en approche de l'ISS le 31 mars 2008[4]

précisément à une partie du logiciel embarqué sur l'ATV : le composant *Solar Generation Subsystem (SGS)*. La fonction première du composant *SGS* est de réaliser le déploiement et la rotation des 4 ailes de l'ATV. Il est à noter que l'étude de cas porte sur un modèle extrait par rétro-ingénierie et non pas sur le système opérationnel effectivement embarqué sur l'ATV.

4.2.4.2 Définition et extraction de modèles de traces

Le pipeline des visualisations construites sur la base du framework Metaviz part de l'étage de données. Cet étage est implémenté par un ensemble de metamodèles, il s'agit des métamodèles proposés dans la section 4.1.1.4 : le métamodèle des configurations (*IFConfig.ecore*), le métamodèle des traces (*IFTrace.ecore*) et le métamodèle des scénarios (*Scenario.ecore*).

Ces métamodèles doivent être instanciés à partir des données fournies par l'outil d'analyse utilisé, à savoir, IFx-OMEGA dans notre cas.

IFx-OMEGA fournit des résultats au format XML. Il a fallu instancier les modèles à partir de ces résultats en utilisant la technique d'injection [100] des artefacts XML vers un métamodèle XML. Un ensemble de trois transformations (une pour chacun

des trois métamodèles) sont chargées de créer les modèles de données à partir des modèles XML.

4.2.4.3 Visualisation Graphcom

La complexité du composant SGS conduit à une énorme quantité d'information à analyser pendant les phases de diagnostic. Les fichiers XML générés, représentant les scénarios d'exécution, atteignent 50 milles lignes. Une partie des résultats est rapportée au niveau du langage utilisé par l'utilisateur comme cela a été publié dans [131]. Mais ce retour d'information de diagnostic ne permet pas de voir l'information dont a besoin l'utilisateur. Le besoin réel de l'utilisateur consiste à voir l'ensemble des interactions entre les objets instanciés dans le composant SGS [64]. En effet, l'outil IFx-OMEGA ne permet pas de voir le graphe des messages échangés durant un scénario.

Cette visualisation doit être créée à partir d'une combinaison d'informations venant du scénario mais aussi de l'explorateur du graphe d'état. Ce graphe de communication est utilisé par exemple pour détecter les groupes d'objets pour lesquels une abstraction peut être définie afin de résoudre le problème de l'explosion de l'espace d'états [64].

Ainsi, le scénario renvoyé par le model checker sous la forme d'une liste d'événements, n'est pas adapté pour permettre à l'utilisateur de répondre à des questions sur la communication inter-objets. En effet, comme le montre le listing 4.1 il s'agit d'une structure plate décrivant les messages échangés ainsi que des événements internes au model checker. Ce scénario n'expose pas directement la structure du graphe de communication. Il est évident que nous ne pouvons pas en déduire facilement la participation d'un objet à un scénario et l'ensemble des messages qu'il échange avec un autre objet. Cette information de communication concrète est primordiale pour le diagnostic.

Nous proposons dans la suite de construire une visualisation du graphe de communication pour permettre d'explorer les scénarios du composant SGS.

```
<IfEvent kind="INPUT" value="u2i__complete{}">
 <by>
  <pid name="u2i__default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS"
 no="0"/>
 </by>
</IfEvent>
<IfEvent kind="INFORMAL" value="--create sub-component
```

```
TS_SGS_F012_EXECUTE_SGS_AP−−">
 <by>
  <pid name="u2i__default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS"
no="0"/>
 </by>
 </IfEvent>
<IfEvent kind="IMPORT" value="">
 <by>
  <pid name="u2i__default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS"
no="0"/>
 </by>
</IfEvent>
```

Listing 4.1 – Extrait d'un contre-exemple de SGS

Avec les interfaces de diagnostic classique, l'utilisateur peut explorer le scénario pas à pas, mais il n'a pas de vision globale sur le graphe de communication des objets. La limitation des capacités cognitives de l'utilisateur nous impose de lui fournir un moyen externe pour augmenter ces capacités (cf. section 4.1.3).

Visualiser un grand nombre d'objets avec leurs interactions peut être réalisé avec un diagramme de communication (similaire au diagramme de communication UML). Ce diagramme consiste en un ensemble de nœuds typés qui représentent les objets et un ensemble d'arcs annotés par les types des messages échangés.

Le tableau 4.1 donne une caractérisation de cette visualisation en utilisant le framework Metaviz. Ce tableau montre pour chaque étage (respectivement chaque opérateur) la description et l'implémentation. Les listings 4.2 et 4.3 montrent une partie des transformations mentionnées dans le tableau de caractérisation.

Étage	Opérateur	Description	Implémentation
des données		Métamodèle de scénaio	Scenario.ecore
	d'analyse	Crée un graphe de communication à partir des messages échangés	Scenario2ComDiag.atl
d'analyse		Graphe de communication inter-objets	ComDiag.ecore
	de visualisation	Crée un graphe typé	ComDiag2Graph.atl
de visualisation		Graphe typé	Graph.ecore
	de mapping visuel	Crée des représentations spécifiques à l'outil de visualisation (layout, couleurs, etc.)	Graph2Dot.atl, Graph2Zest.atl
de vue		Métamodèle spécifique à l'outil de visualisation	Zest.ecore, Dot.ecore

TABLE 4.1 – Caractérisation de la technique de visualisation Graphcom avec le framework Metaviz [18]

```
-- ATL module that transforms a Scenario to
-- a communication diagram
module Scenario2ComDiag;
create OUT: ComDiag from IN: Scenario;

-- override this helper to filter messages with a predicate
helper context Scenario!Message def: messagePred(): Boolean =
        true;

-- override to filter processes
helper context Scenario!Pid def: processPred(): Boolean =
        true;

-- override to filter actions
helper context Scenario!Action def: actionPred(): Boolean =
        self.kind = 'INPUT' and -- reject IF internal actions
                not self.value.startsWith('u2i_');

helper def: allActions: Set(Scenario!Action) =
        Scenario!Action.allInstances();

helper def: satisfyActions: Set(Scenario!Action) =
        thisModule.allActions -> select(a | a.actionPred());

helper def: scenario: Scenario!Scenario =
        Scenario!Scenario.allInstances() -> first();

-- Entry point rule to transform a Scenario to
-- a communication diagram
rule Scenario2ComDiag {
        from
                sS: Scenario!Scenario
        to
                tC: ComDiag!ComDiag (
                        name <- sS.name,
                        elements <- thisModule.satisfyActions
                                -> collect(a | a.message)
                                -> select(m |m.messagePred())
                                -> collect(m | thisModule.message2Link(m))
                )
}

lazy rule message2Link {
        from
                sM: Scenario!Message
```

```
        to
                tL: ComDiag!Message (
                        name <- sM.signalType,
                        source <- tSN,
                        target <- tTN,
                        graph <- thisModule.scenario
                ),
                tSN: ComDiag!Lifeline (
                        name <- sM.from.name,
                        number <- sM.from.number,
                        graph <- thisModule.scenario
                ),
                tTN: ComDiag!Lifeline (
                        name <- sM.to.name,
                        number <- sM.to.number,
                        graph <- thisModule.scenario
                )
}
```

Listing 4.2 – Opérateur d'analyse de scénarios par production d'un graphe de communication.

```
-- ATL module that transforms a graph to a
-- Dot description compatible with Graphviz tool
module Graph2Dot;
create OUT : Dot from IN : Graph;

-- Transform a graph to a dot description
-- compatible with Graphviz tool
rule Graph2Dot {
        from
                sC : Graph!Graph
        to
                tD : Dot!DiGraph(
                        ratio <- 0.05,
                        nodesep <- 0.5,
                        NodeShape <- 'box',
                        ArcColor <- 'red',
                        name <- sC.name,
                        contents <- sC.contents
                )
}
rule Node2Node {
        from
                sL : Graph!Node
```

```
        to
                tDA : Dot!Node(
                        name <− sL.name
                )
}
rule DirectedArc2DirectedArc {
        from
                sL : Graph!DirectedArc
        to
                tDA : Dot!DirectedArc(
                        name <− sL.name,
                        sourceNode <− sL.sourceNode,
                        targetNode <− sL.targetNode
                )
}
```

Listing 4.3 – Opérateur de mapping visuel vers l'outil de visualisation de graphe Graphviz.

4.2.4.4 Visualisation TraceSummary

Pour proposer à l'utilisateur une visualisation d'abstraction nous allons caractériser et implémenter une technique d'extraction automatique de résumé [80]. Le but est d'extraire une vue d'ensemble de la trace qui renseigne sur sa complexité et sur les composants et messages les plus pertinents.

Pour construire cette visualisation il suffit de personnaliser la visualisation Graphcom construite précédemment. Nous avons utilisé la technique de superimposition [156] implémentée dans le langage ATL. Cette technique permet de modifier le comportement d'une transformation sans en modifier le code, une technique similaire à la surcharge de méthode dans les langages orientés objet. Les fonctions ATL (helpers) du listing 4.4 réalisent cette personnalisation.

```
helper context Scenario!Message def: messagePred(): Boolean=
        self.from.processPred() and self.to.processPred();

helper context Scenario!Pid def: processPred(): Boolean=
        self.forwardingMetric() < thisModule.threshold;

helper context Scenario!Pid def: forwardingMetric(): Real=
        let N : Integer = thisModule.allProcessSize in
          (self.fanin()/N)*(N/(self.fanout()+1)).log()/N.log();
```

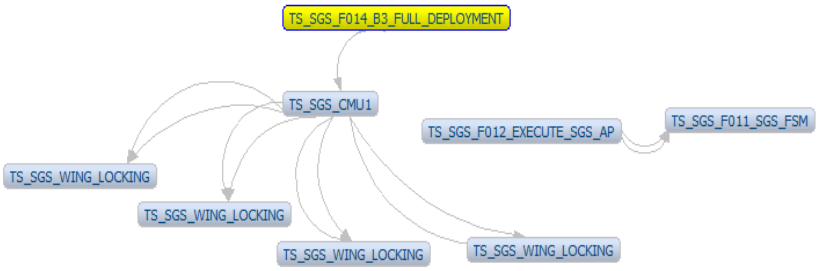


FIGURE 4.18 – Technique de résumé de trace

Listing 4.4 – Fonctions permettant d'obtenir le résumé de trace via la technique de superimposition

La première fonction *messagePred()* implémente un prédicat sur les messages échangés dans une trace. Cette fonction fait appel à *processPred()*, un autre prédicat sur les objets participant à une trace, qui à son tour fait appel à *forwardingMetric*, la fonction qui implémente une métrique de résumé [80].

Cette métrique est calculée pour chaque objet de la trace à partir du nombre d'objets qui lui envoient des messages (fonction *fanin()*) et du nombre d'objets à qui il envoie des messages (fonction *fanout()*). L'intérêt de cette métrique est de pouvoir, moyennant un seuil défini, filtrer les objets qui ne font que du transfert de messages. La figure 4.18 montre le résultat de cette personnalisation.

Le résultat montre les objets les plus pertinents qui collaborent pour exécuter les fonctionnalités principales du composant SGS.

Cette visualisation montre la réutilisation que permet le framework, il n'aura fallu que 3 fonctions ATL pour implémenter une visualisation de résumé de trace.

4.2.4.5 Visualisation Timeline

Durant le processus de validation des modèles, l'utilisateur peut être confronté à un processus de vérification automatique qui ne s'arrête pas. Ce problème est dû à l'explosion de l'espace d'états qui peut être intrinsèque au modèle (il faut donc appliquer des abstractions) ou le résultat d'une erreur de conception.

Une erreur récurrente qui mène à l'explosion de l'espace d'état est l'envoi de messages à des objets qui ne les attendent pas. Ces message s'accumulent dans la file d'attente de l'objet et créent indéfiniment de nouveaux états du système. Le principe de la visualisation *Timeline* est de permettre à l'utilisateur de voir l'évolution des

files d'attentes des objets impliqués dans une trace pour vérifier qu'il n'y a pas de file anormalement croissante.

La figure 4.19 montre le résultat de la visualisation d'une trace avec un objet (de type *Ambients-PatientMonitor* dans la figure) qui a une file de taille croissante. Après correction du modèle (comme montré dans la figure 4.20) la visualisation montre un comportement normal des files de message (cf. figure 4.21). Pour construire cette visualisation nous avons utilisé la caractérisation proposé par Metaviz et explicitée dans le tableau 4.2.

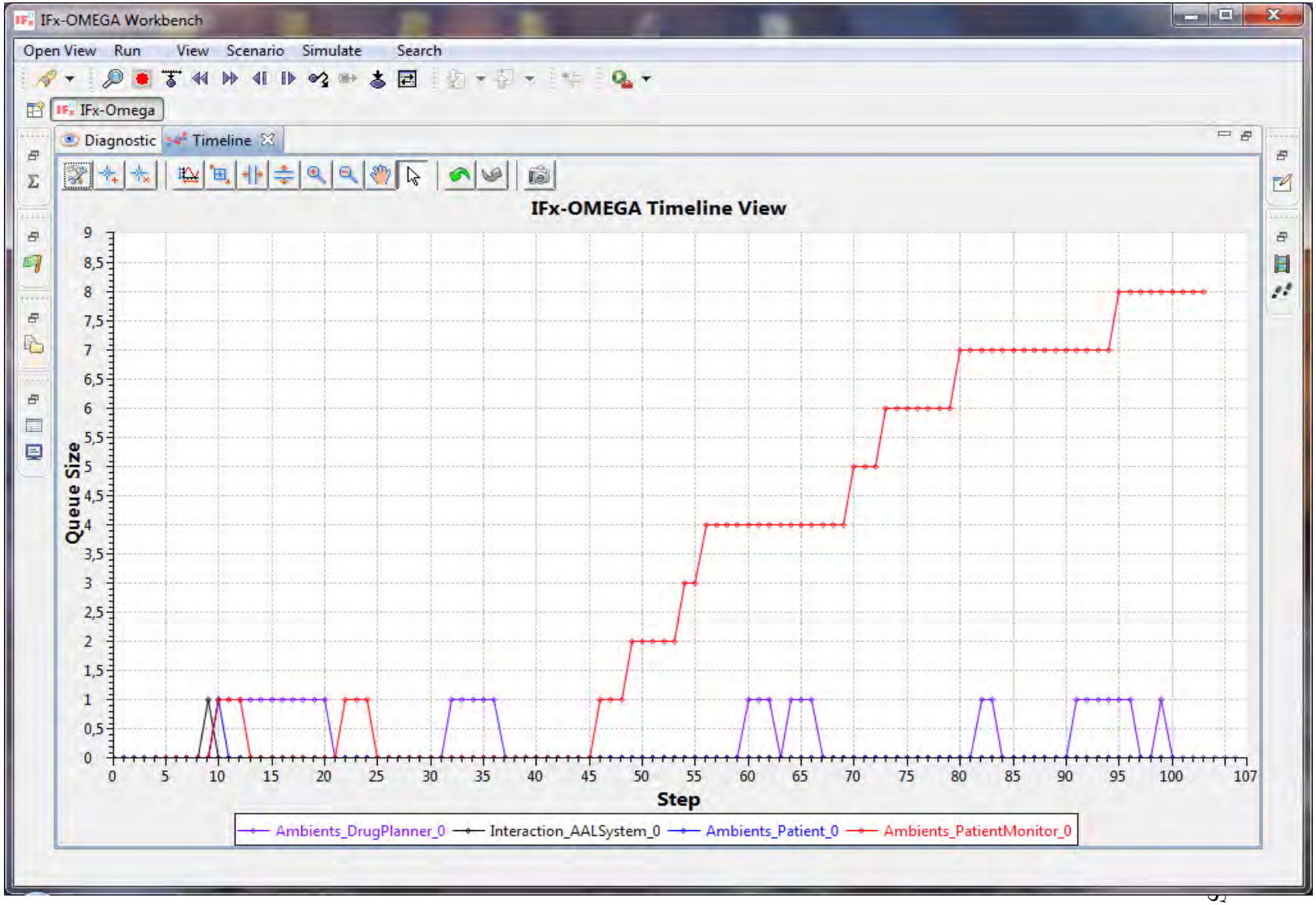


FIGURE 4.19 – Détection visuelle d'une erreur de spécification dans les traitements des messages.

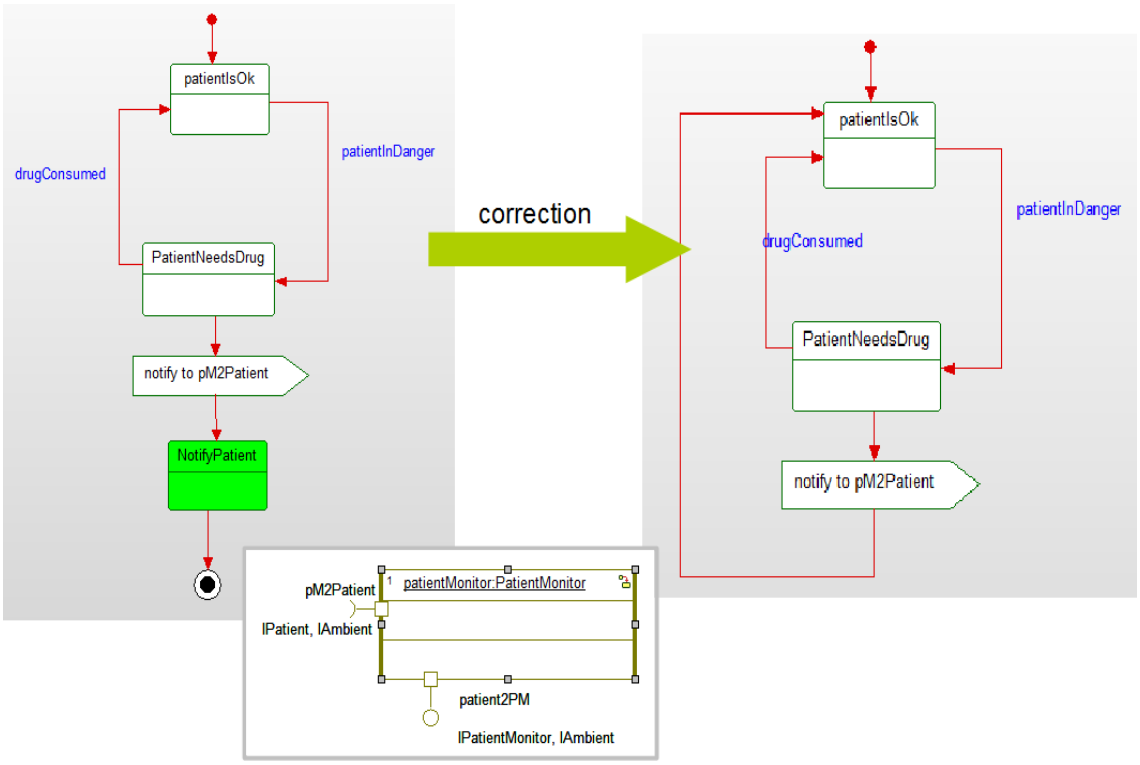


FIGURE 4.20 – Correction du modèle de départ.

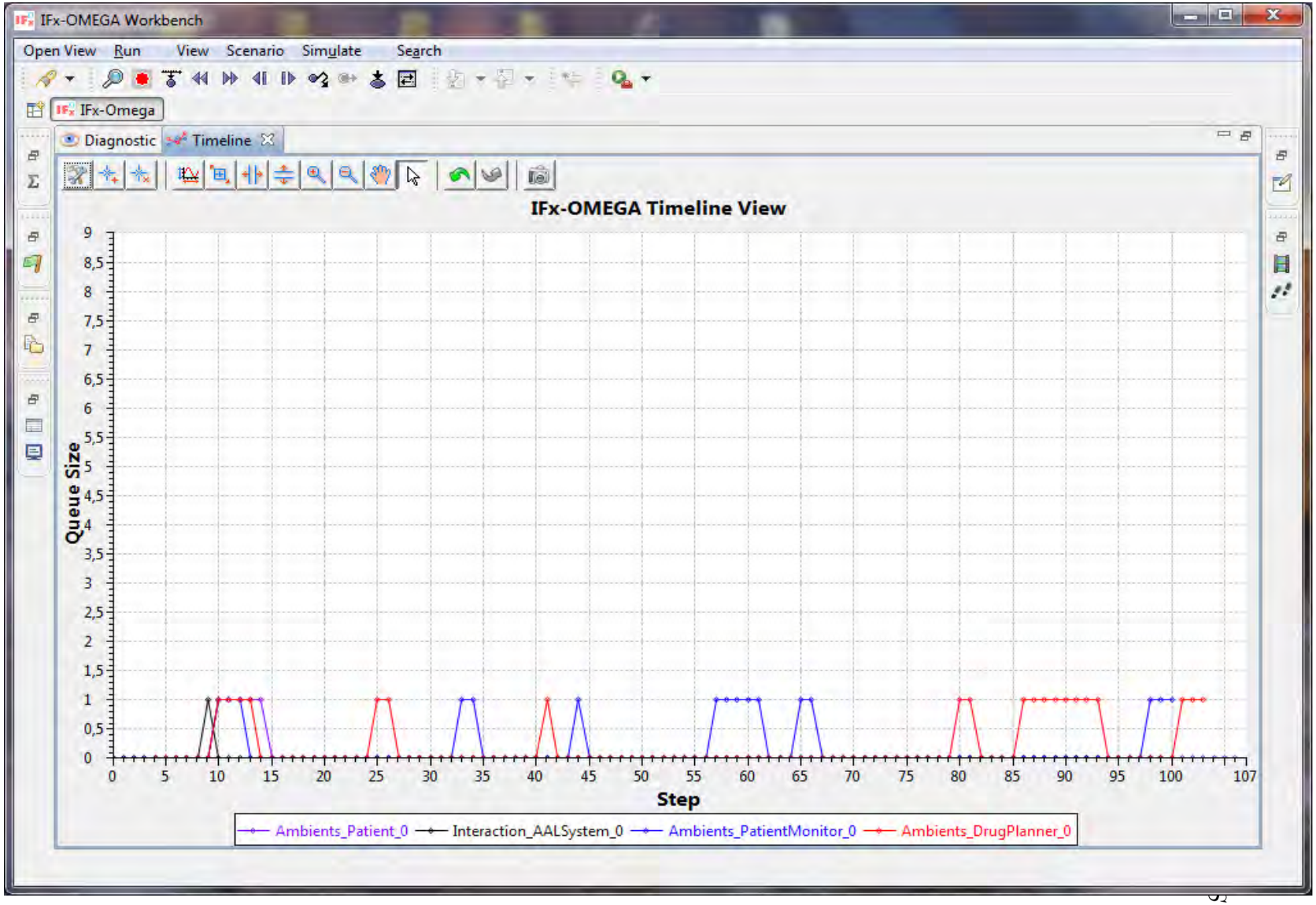


FIGURE 4.21 – Après correction, la file de message reprend un comportement normal (i.e. taille bornée).

Étage	**Opérateur**	**Description**	**Implémentation**
des données		Métamodèle de trace	Trace.ecore
	d'analyse	Crée une table d'évolution des queues des processus IF	Trace2QueueEvolution.atl
d'analyse		Graphe d'évolution	QueueEvolution.ecore
	de visualisation	d'une queue de processus Crée une ligne de temps	QueueEvolution2Timeline.atl
de visualisation		Ligne de temps	Timeline.ecore
	de mapping visuel	Crée des représentations spécifiques à l'outil de visualisation	Timeline2XYGraph.atl,
de vue		Métamodèle spécifique à l'outil de visualisation	XYGraph.ecore

TABLE 4.2 – Caractérisation de la technique de visualisation Timeline avec le framework Metaviz [18]

Pour l'étage de la vue nous avons utilisé la librairie graphique SWT XYGraph [5] qui permet une intégration directe dans notre outil de diagnostic basé sur Eclipse[6].

Chaque visualisation développée dans cette partie assiste l'utilisateur dans une tâche spécifique de diagnostic. Le tableau 4.3 montre la tâche correspondante pour chaque visualisation.

Visualisation	Tâche
Graphcom	Compréhension de trace, détection de symétrie
TraceSummary	Compréhension de trace, identification des composants
Timeline	Correction de réception erronée de messages
ObjectStory	Compréhension de trace

TABLE 4.3 – Ensemble des visualisations développées avec Metaviz pour des tâches de diagnostic.

4.2.5 Outils d'exploration des vues

Le framework Metaviz offre une approche de construction des visualisations basée sur un modèle de référence qui simplifie la conception de visualisations. Nous avons dans les sections précédentes construit 3 types de visualisations qui assistent l'utilisateur dans des tâches de diagnostic. Mais nous n'avons pas encore abordé l'interaction de l'utilisateur avec ces visualisations. Il s'agit d'offrir un ensemble d'outils de contrôle du résultat final affiché par l'utilisateur.

Ces outils de contrôle vont permettre d'explorer plus en profondeur le résultat affiché ainsi que les possibilités de personnalisation d'une visualisation existante. Ces outils seront rajoutés au niveau de l'*étage des vues*.

4.2.5.1 Explorateur de vues (OCL Dynamic Queries)

L'une des fonctionnalités d'un système de visualisation efficace est l'exploration de l'espace des données [24]. Une méthode largement utilisée consiste à utiliser un langage de requêtes. Notre approche étant basée sur les standards de l'OMG, il est naturel de s'orienter vers le langage OCL [135].

Cet explorateur de vue est constitué de trois composants principaux :

- **la vue** : une vue d'arbre qui affiche le modèle objet de la requête ainsi que le résultat de son évaluation ;

[5] `http://code.google.com/p/swt-xy-graph`
[6] `http://www.eclipse.org`

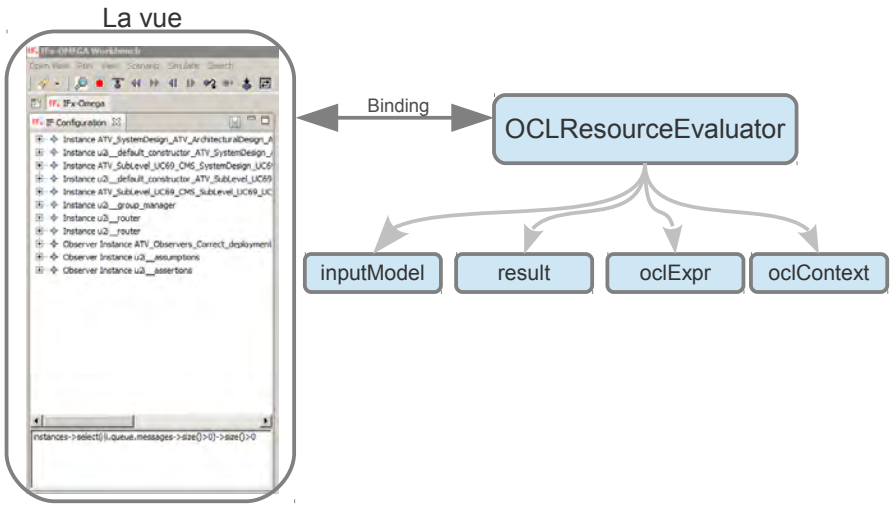


FIGURE 4.22 – Architecture de l'explorateur de vue. L'objet OCLResourceEvaluator référence l'objet inputModel (modèle à explorer), l'objet result (résultat de la requête OCL) et les objets oclExpr et oclContext représentant la requête OCL de l'utilisateur.

- **l'éditeur de requêtes** : qui permet de saisir des requêtes OCL. Une fonctionnalité d'auto-complétion permet d'assister l'utilisateur dans la formulation des requêtes ;
- **la console** : qui affiche le résultat de la requête sous forme textuelle. L'affichage textuel des résultats est utilisé par de nombreux outils (e.g. USE [76], Topcased [14]).

La figure 4.22 montre l'architecture détaillée de cet explorateur. L'objet *OCLResourceEvaluator* est chargé d'exécuter la requête sur le modèle courant et de renvoyer le résultat à la vue pour l'affichage. L'utilisateur peut aussi sélectionner le contexte de la requête OCL par simple clique sur un élément du modèle courant. L'outil de diagnostic peut contenir plusieurs vue de diagnostic, chaque vue embarquant son propre explorateur.

Une autre utilisation possible de cet explorateur en simulation interactive est d'utiliser des requêtes OCL comme des points d'arrêts multiples. La simulation s'arrêtant dès qu'une des requêtes est satisfaite. Cela permet, d'une part d'explorer de manière plus rapide les modèles affichés dans les vues, et d'autre part d'explorer plusieurs vues simultanément avec plusieurs points d'arrêts (états des objets, transitions activées,...). La figure 4.23 montre un exemple de ce cas d'utilisation.

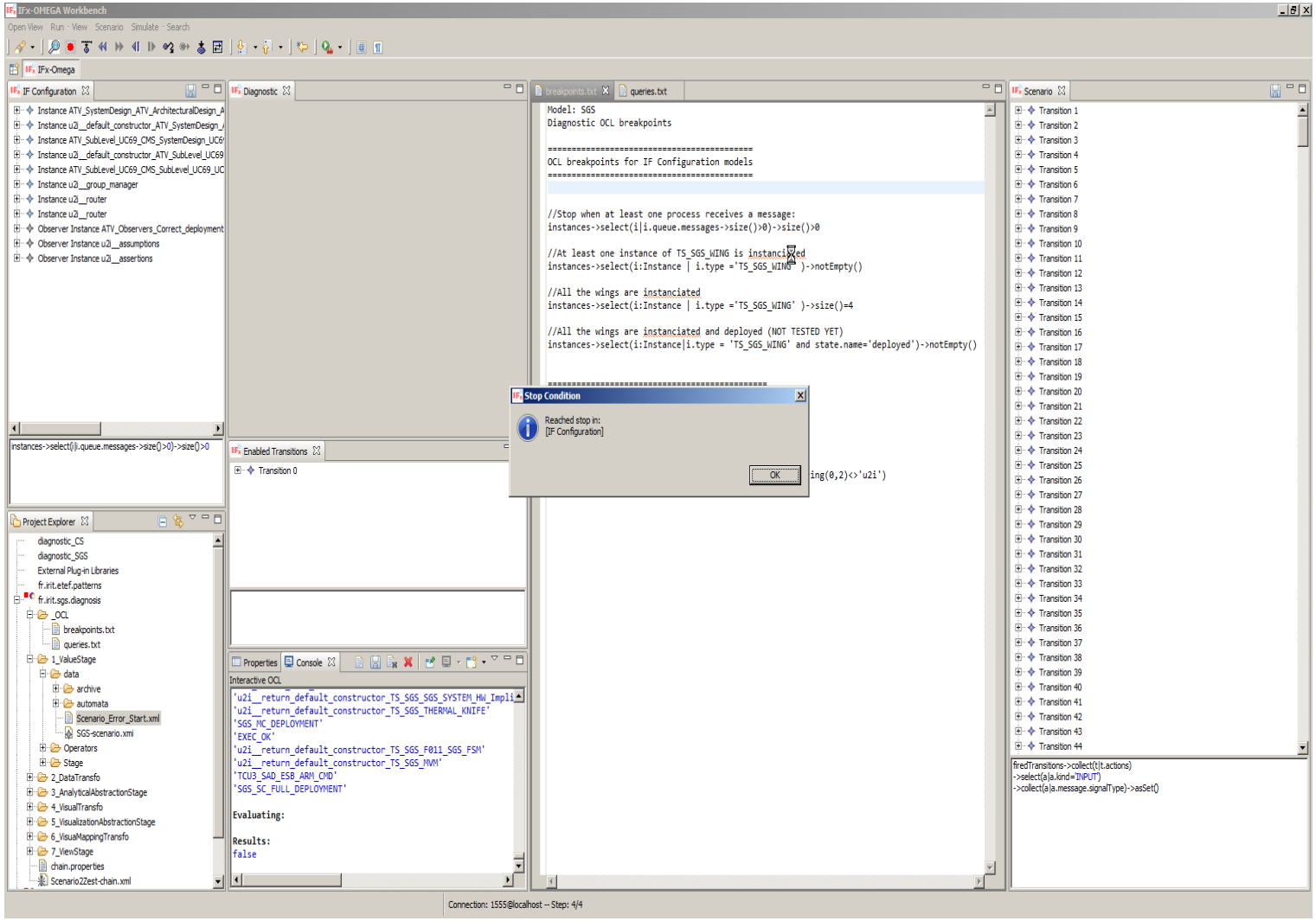


FIGURE 4.23 – Utilisation des explorateurs pour poser des points d'arrêt en simulation interactive. Le point d'arrêt OCL dans la vue de gauche *IFConfigView* permet d'arrêter la simulation automatique au premier échange de message ce qui permet par exemple de sauter toute la phase d'initialisation des objets.

4.2.5.2 Les filtres de vues

Le filtrage constitue une technique fondamentale dans la visualisation d'information [149]. Cette fonctionnalité permet de supprimer de la vue les informations que l'utilisateur ne juge pas pertinentes. Pour cela il faut offrir un mécanisme de filtrage des objets de la vue. Ce mécanisme devra aussi permettre un filtrage rapide pour servir de moyen d'exploration.

Nous avons réalisé ce mécanisme et l'avons testé sur le modèle du composant SGS. Il s'agit de construire un filtre sur la base de la visualisation du graphe de communication. Le but est de fournir un moyen simple pour filtrer les messages ou les objets qui ne sont pas pertinents au yeux de l'utilisateur.

Ce filtre peut aussi être vu comme une variation dans le comportement de la visualisation de base, et nous pouvons donc appliquer la même technique de superimposition utilisée précédemment (section 4.2.4.4) pour le résumé des traces. Le listing 4.5 montre le code du module ATL nécessaire pour filtrer les messages d'une trace. Dans ce code seuls les communications qui impliquent les 2 types de messages *SGS_AP_SET_REMOVE_SB* et *SGS_DEPLOY_WING_STATUS* seront affichées. Ce code est utilisé par l'outil pour appliquer le filtre et le résultat final est affiché sur une vue de diagnostic de l'outil *IFx-Workbench* (cf. figure 4.24).

```
helper context Scenario!Message def : messagePred(): Boolean=
   Set{'SGS_AP_SET_REMOVE_SB', 'SGS_DEPLOY_WING_STATUS'}
    ->includes(self.signalType)
;
```

Listing 4.5 – Helper ATL pour implémenter un filtre de message

Nous avons jusqu'à maintenant évalué la partie du processus générique qui caractérise les visualisations. Pour cela nous avons conçu, implémenté et personnalisé un ensemble de visualisations. Dans les prochaines sections nous proposons d'évaluer tout le processus, de l'analyse de la tâche jusqu'à la conception des visualisations. Une évaluation empirique et un protocole expérimental seront proposés plus tard dans la section 4.4.

4.3 Application du processus

Dans cette partie nous allons évaluer tout le processus de conception d'outils d'assistance au diagnostic proposé dans la section 4.2.1.

L'évaluation du processus répond essentiellement aux questions suivantes :

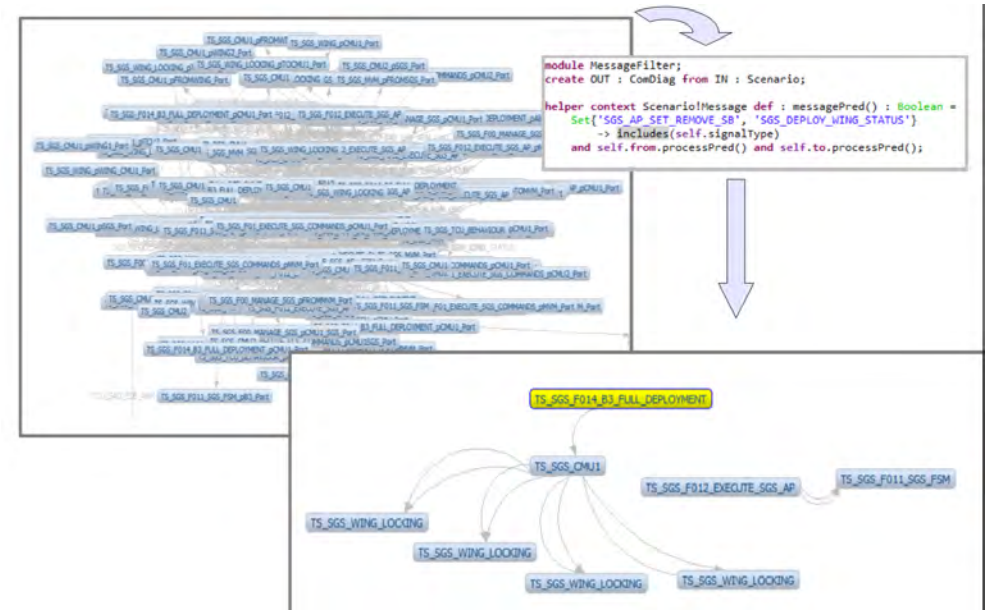


FIGURE 4.24 – Application d'un filtre basé sur les messages

- peut-on concevoir, documenter, maintenir et personnaliser des visualisations ?
- les visualisations construites à la fin du processus améliorent-t-elle l'efficacité de l'utilisateur dans une tâche précise ?

Les visualisations que nous avons construites dans les sections précédentes (GraphCom, Timeline, etc.) ainsi que celle que nous allons construire dans cette section apportent une réponse à la première question. Pour la seconde question une évaluation empirique sera proposée dans la section évaluation empirique 4.4.

4.3.1 Analyse de la tâche

4.3.1.1 Erreurs récurrentes dans la validation des modèles OMEGA

L'activité d'analyse de la tâche proposée par le processus générique permet de comprendre au sein d'une tâche utilisateur quels sont les blocages récurrents. L'utilisation de notre outil dans plusieurs études de cas, dont certaines de taille industrielle comme SGS [64], a permis de relever un certain nombre de difficultés rencontrées par les utilisateurs. Il s'agit d'erreurs récurrentes, commises par l'utilisateur au niveau des modèles OMEGA et qui mènent soit à un blocage dans le processus de

vérification automatique (i.e. explosion de l'espace d'états) soit à des modèles qui ne satisfont pas les propriétés spécifiées.

Le recensement de ces erreurs fait ressortir principalement deux types d'erreurs. La première est l'**oubli d'une transition dans la machine à états d'un objet**, menant à un comportement infini via l'accumulation de messages dans la file de cet objet. Cette erreur mène à l'explosion de l'espace d'états et à l'impossibilité de finir le processus de vérification automatique.

Le second type d'erreurs est le **traitement du même message par plusieurs machines à états parallèles**. Cette erreur mène souvent à une situation d'interblocage. En effet l'hypothèse sur la priorité d'une machine à états sur une autre a été supposée par l'utilisateur mais n'a pas été correctement spécifiée dans le modèle.

Essayons de comprendre comment l'utilisateur outrepasse ces erreurs en utilisant l'outillage basique fourni par IFx-OMEGA. Le but est de comprendre quelles sont les difficultés (d'ordre cognitif) rencontrées pendant la tâche de diagnostic et de proposer une solution en utilisant le processus générique.

4.3.1.2 Utilisabilité de l'interface existante

L'outil IFx-OMEGA est actuellement fourni avec une interface de simulation (appelée *ifsimgui*) qui a pour fonction première de permettre une exploration des scénarios et des contre-exemples générés par le model checker. L'interface *ifsimgui* offre plusieurs fonctionnalités de simulation (i.e. exploration de l'espace d'état) :

- simulation pas à pas en avant ou en arrière d'un scénario ;
- tirage de transitions activées dans l'état courant ;
- sérialisation/chargement d'un scénario au format XML ;
- retour à l'état initial du système.

L'utilisation de cette interface sur des modèles de taille industrielle montre qu'elle souffre de quelques limitations. En effet le diagnostic de scénario de simulation implique l'exploration et la vérification visuelle d'un très grand nombre d'information (le scénario, les configurations, les objets actifs et leurs propriétés, les transitions, les diagrammes UML, etc...).

D'une part, cette information est non-contiguë et de nature différente, d'autre part, les capacités humaines sont très restreintes (cf. section 2). Une restriction principalement due à la mémoire de travail. Cependant l'interface actuelle est constituée uniquement de visualisations à base de listes arborescentes (représentation des artefacts XML générés par le model checker) comme le montre la figure 4.25. Ces

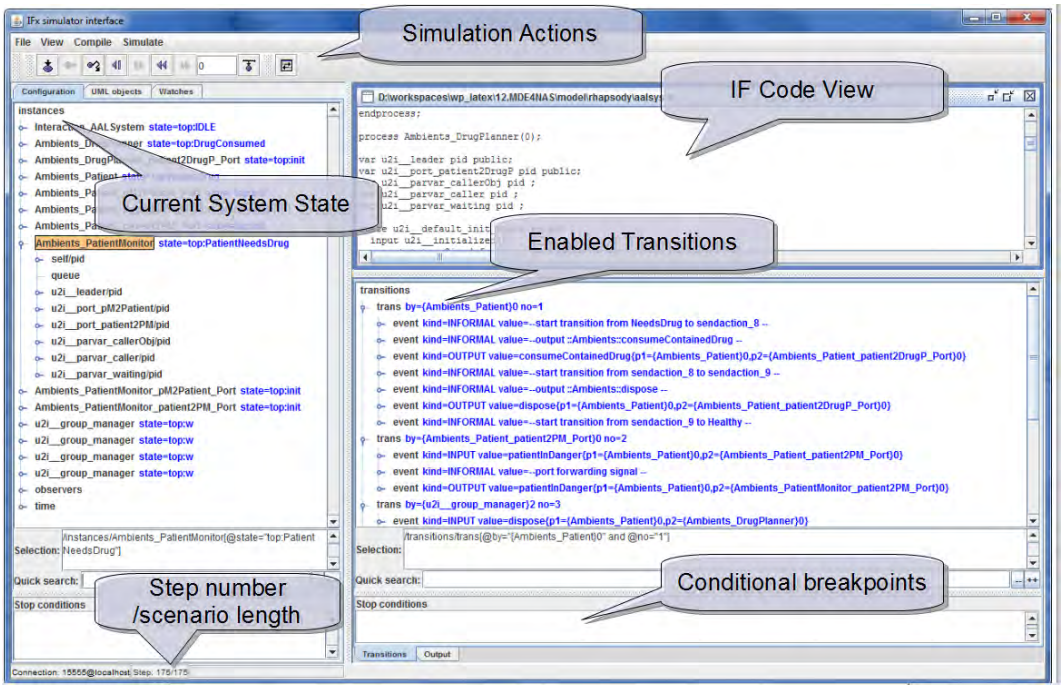


FIGURE 4.25 – Interface de simulation de la chaîne d'outils IFx-OMEGA dans sa version 2.0

limitations ont été présentés plus en détails dans la section 2.3.5 où nous avons montré qu'elle se résume à une surcharge cognitive de l'utilisateur. L'application des 5 recommandations proposées dans la section 4.1.3 permettra de réduire cette charge.

Une autre limitation est la personnalisation et l'extension de cette interface. Pour personnaliser les listes arborescentes, l'utilisateur doit charger une feuille de style XSL [11] pour transformer le fichier XML chargé dans la vue principale qui affiche les configurations. Il est donc nécessaire d'avoir une connaissance suffisante d'XSL pour effectuer la personnalisation. Cette fonctionnalité peut être suffisante pour faire du filtrage des arbres affichés par l'interface. Mais elle s'avère inutilisable pour implémenter une visualisation plus élaborée comme celle du graphe de communication. Pour ce type de visualisation il faut recourir à des implémentations ad-hoc en dehors de l'interface de simulation. Le listing 4.6 montre un exemple de commande utilisé par les utilisateurs pour extraire le graphe de communication à partir d'une trace.

```
grep −v ”””” | grep −v u2i__default | grep −v u2i__init | sed 's/^.*<<//g'
    | sed 's/!.*}{/{/g' | sed 's/>>.*//g' | sort −u | sed 's/}//g' | awk '
BEGIN { print ”digraph LTS {”; print ” node [shape = circle];”; }
```

```
END { print "}"; }
/des.*/ { next; } // { split($1,a1,"_"); split($2,a2,"_");
print "\"" a1[length(a1)] "\"" " -> " "\"" a2[length(a2)] "\"" ";" ; }
' | dot -Tepdf > net.pdf
```

Listing 4.6 – Commande Unix pour la construction d'une visualization ad-hoc

Le tableau 4.4 montre une comparaison de l'approche basée sur le framework Metaviz et de l'approche ad-hoc utilisée dans l'outillage IFx-OMEGA (v2.0).

	Implémentation ad hoc	**Implémentation Metaviz**
Besoins fonctionnels		
Personnalisation des données	Oui	Oui
Personnalisation des représentations	Oui	Oui
Contrôle utilisateur	Non	Faisable
Recommandations de conception		
Spécification efficace	Non	Oui, via le style déclaratif des transformations de modèles
Langage familier à l'utilisateur du DSML	Non	Oui
Intégration des outils	Non	Oui
Fournit des adaptateurs de modèles aux vues	Non	Oui
Utilise des librairies graphiques existantes	Oui (Graphviz)	Oui (Zest, Graphviz, etc.)
Spécification explicite	Non	Oui, via l'utilisation des règles de transformation

TABLE 4.4 – Comparaison entre une implémentation ad hoc et une implémentation avec le framework Metaviz en utilisant la taxonomie de I.Bull [41]

De cette comparaison nous déduisons quelques contraintes sur la conception de l'outil de diagnostic. En effet, il faut offrir un outillage qui permette de :

- découvrir rapidement les interactions entre les objets ;
- personnaliser rapidement les résultats en utilisant des outils familiers à l'utilisateur (OCL ou Java).

4.3.2 Définition d'une stratégie d'amélioration

Au vue des limitations du contexte d'exécution de la tâche de diagnostic exposées dans le paragraphe précédent, il est nécessaire de définir une stratégie de résolution du problème de la surcharge cognitive. Les recommandations que nous proposons dans la section 4.1.3 ont pour principal but de diminuer la charge cognitive.

Application des recommandations A partir de l'analyse de la tâche de l'utilisateur et des erreurs récurrentes auxquelles il est confronté pendant le diagnostic, il s'avère que l'information recherchée dans la trace porte sur :

- l'initialisation des objets ;
- l'exécution des transitions qui impliquent un envoi de messages : ce qui permet de vérifier qu'aucune transition n'a été omise durant la spécification et évitera le premier type d'erreur mentionné précédemment 4.3.1.1 et lié à l'explosion de l'espace d'états ;
- le changement d'états induit par l'exécution sus-mentionnée : ce qui permet de définir quel objet a reçu le message et évitera ainsi le second type d'erreur lié aux machines à états concurrentes.

Nous allons pour chacune de ces trois informations, développer une notation, en prenant en considération les recommandations. La maquette développée à cet effet est donnée dans la figure 4.26, mais pour plus de clarté l'application des recommandations sera montrée en utilisant le prototype final.

Gestion de l'attention
Dans chacune des trois notations, l'information pertinente sera coloriée en rouge permettant ainsi une focalisation rapide de l'œil. Par exemple pour l'envoi de message, une flèche rouge et une étiquette portant le type du message seront utilisées (cf. figure 4.28).

Syntaxe intuitive (ou à sémantique immédiate)
Les notations reprendront la forme de la notation du langage de modélisation OMEGA (diagramme de classe), une icône sera utilisée pour différencier les objets nouvellement instanciés (cf. figure 4.27).

Contiguïté de l'information pertinente
Les changements d'états d'un objet seront représentés de manière contiguë (i.e. à

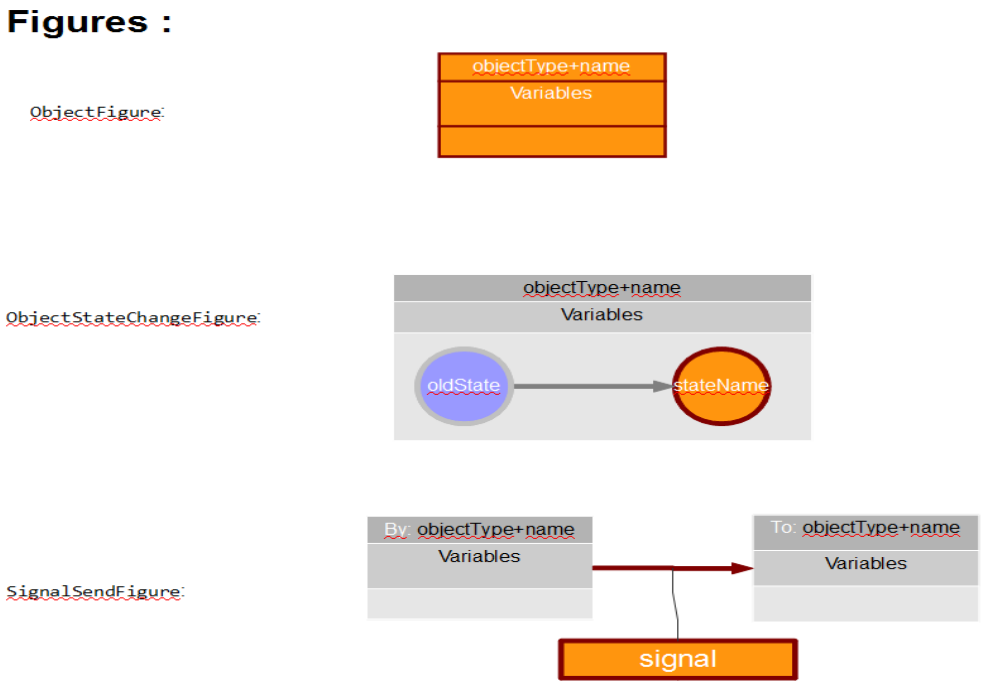


FIGURE 4.26 – Maquette de la notation ObjectStory qui montre l'application des 5 recommandations.

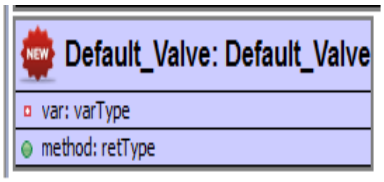


FIGURE 4.27 – Syntaxe intuitive pour l'instanciation des objets dans la trace.

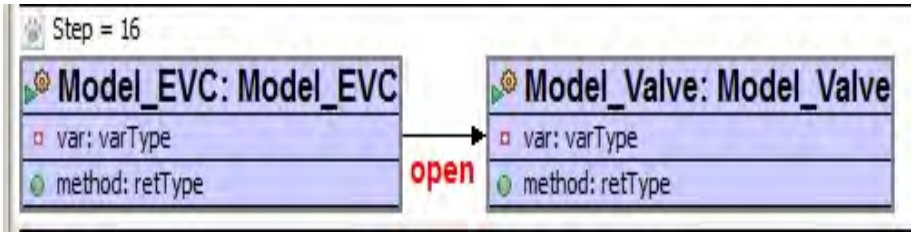


FIGURE 4.28 – Principe de contiguïté de l'information pertinente appliqué à l'échange de message entre objets.

l'intérieur de la représentation de cet objet). Deux objets qui communiquent seront représentés à proximité et le message échangé sera mis en valeur 4.28.

Persistance de l'information
L'ensemble de l'information recherchée sur la trace sera disponible à l'utilisateur sans effort de manipulation via l'utilisation d'un diagramme à défilement (cf. figure 4.29).

Réduction de l'information
L'information sera réduite via un système d'extraction des différences entre deux configurations successives dans la trace. Nous n'afficherons que les changements intéressants aux yeux de l'utilisateur. Ce qui correspond à trois événements pertinents : l'instanciation d'un objet, le changement d'état, et la réception d'un message. Les autres informations de la trace restent accessibles grâce à une autre vue de l'interface que l'utilisateur peut consulter si nécessaire (cf. figure 4.30).

4.3.3 Caractérisation d'une visualisation

Dans le processus générique de conception d'outils de diagnostic, l'activité suivante est la **sélection d'une technique de visualisation** qui réalise la stratégie proposée ci-dessus. Nous n'avons pas trouvé dans la littérature [50, 46, 65, 157] de technique que nous pouvons réutiliser, nous caractériserons donc une nouvelle visualisation en utilisant le framework Metaviz. Le tableau 4.5 propose une caractérisation de cette nouvelle visualisation. Nous l'avons appelé **Objectstoy** puisqu'elle est inspirée du mécanisme de *filmstrip* de cinéma qui permet de voir une scène en gardant les scènes passées et en rendant les scènes futures accessibles.

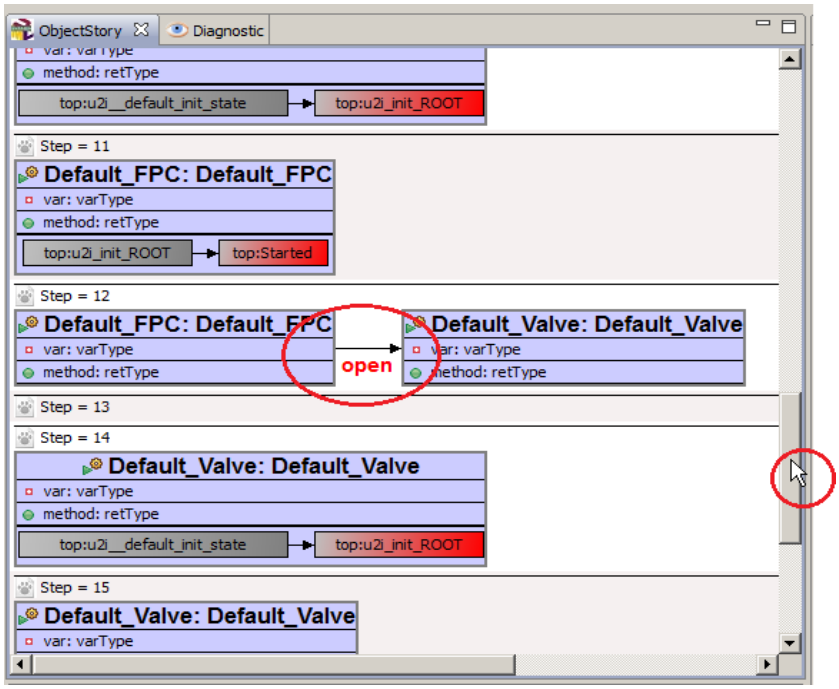


FIGURE 4.29 – Application du principe de persistance de l'information via une liste de pas déroulante

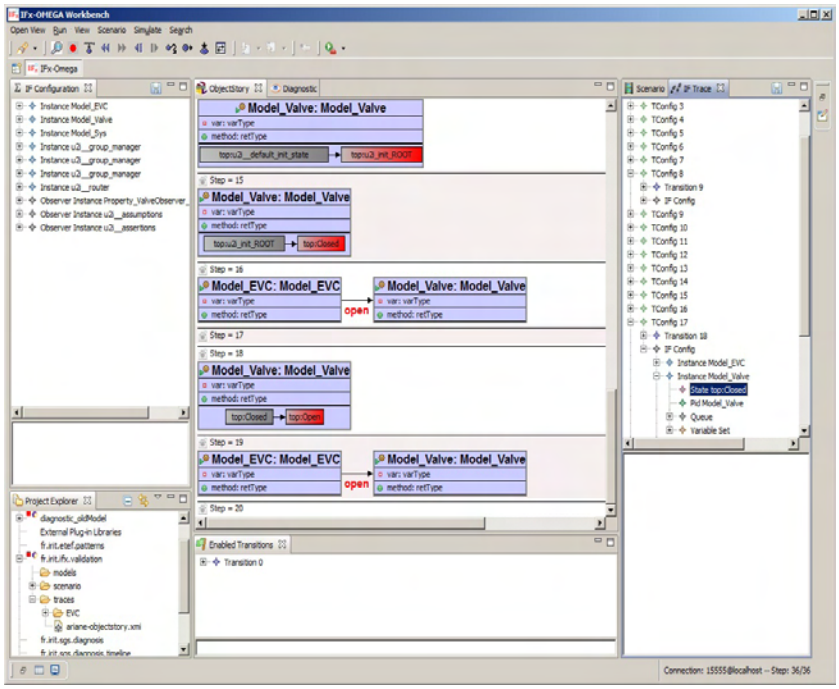


FIGURE 4.30 – Réduction de l'information à une partie de la trace

Étage	**Opérateur**	**Description**	**Implémentation**
des données		Métamodèle de trace	IFTrace.ecore
	d'analyse	Crée un modèle de trace orienté événements	IFTrace2IFTraceEB.java
		orienté événements	
d'analyse		Métamodèle de trace orienté événements	IFTraceEB.ecore
	de visualisation	Crée des événements OMEGA	IFTraceEB2OMEGATraceEB.atl
de visualisation		Métamodèle des événements OMEGA	OMEGATraceEB.ecore
	de mapping visuel	Crée des représentation spécifique à l'outil de visualisation Objectstory	OMEGATraceEB2ObjectStory.atl
de vue		Métamodèle spécifique à l'outil de visualisation	ObjectStory.ecore

TABLE 4.5 – Caractérisation de ObjectStory avec le framework Metaviz [18]

4.3.4 Construction de la visualisation

Construire la visualisation *ObjectStory* revient à implémenter les différents composants IDM exposés dans le tableau de la caractérisation (tableau 4.5). L'étage de données est évidement le métamodèle de trace exposé précédemment 4.1.1.4. Les autres étages seront décrits dans les paragraphes suivants avec les opérateurs qui les alimentent.

4.3.5 Étage d'analyse : extraction d'événements de la trace

Cet étage est le résultat de l'application de l'opérateur d'analyse des données sur une trace IF, instance du métamodèle *IFTrace*. L'idée générale est de générer une série d'événements d'intérêt que l'utilisateur aura choisi d'observer sur la trace au lieu d'afficher toute la trace.

En fait **nous allons passer d'un modèle de trace orienté donnée vers un modèle de trace orienté événements**. L'opérateur d'analyse sera donc chargé d'extraire un ensemble d'événements. D'abord nous spécifions les événements pertinents pour l'utilisateur (i.e. l'instanciation d'objets, la réception de messages et le changement d'états) puis grâce un algorithme de calcul de différences (cf. annexe D) entre les configurations nous extrayons ces événements.

Le résultat final sera alors un ensemble ordonné d'événements *gros grain* présentés à l'utilisateur (cf. principe de réduction de l'information). La figure 4.32 montre une comparaison entre un modèle de trace orienté donnée (instance de *IFTrace*) et un modèle de trace orienté événements (instance de *IFTraceEB*). Cet étage est donc représenté par le métamodèle *IFTraceEB* 4.31.

Pour l'implémentation de l'algorithme d'extraction des événements pertinents nous n'avons pas utilisé de langage de transformation de modèle. En effet, les langages de type QVT [134] ne permettent pas d'implémenter de manière simple l'algorithme d'extraction d'événements. Des langages de transformation dits orientés changement sont plus adaptés à cet effet [140, 33] mais la non compatibilité de l'outillage support [7] avec les standards de l'IDM (MOF, QVT, etc...) empêche son utilisation dans notre outil basé essentiellement sur ces standards. Par conséquent, l'implémentation a été réalisée en utilisant l'API Java d'EMF [3]. L'architecture générale de l'implémentation est décrite par la figure 4.33.

Opérateur d'analyse : Il réalise des fonctions d'analyse sur les données de départ. Dans cet exemple nous cherchons à extraire un ensemble d'événements pertinents à partir de la trace.

[7] `http://www.eclipse.org/viatra2/`

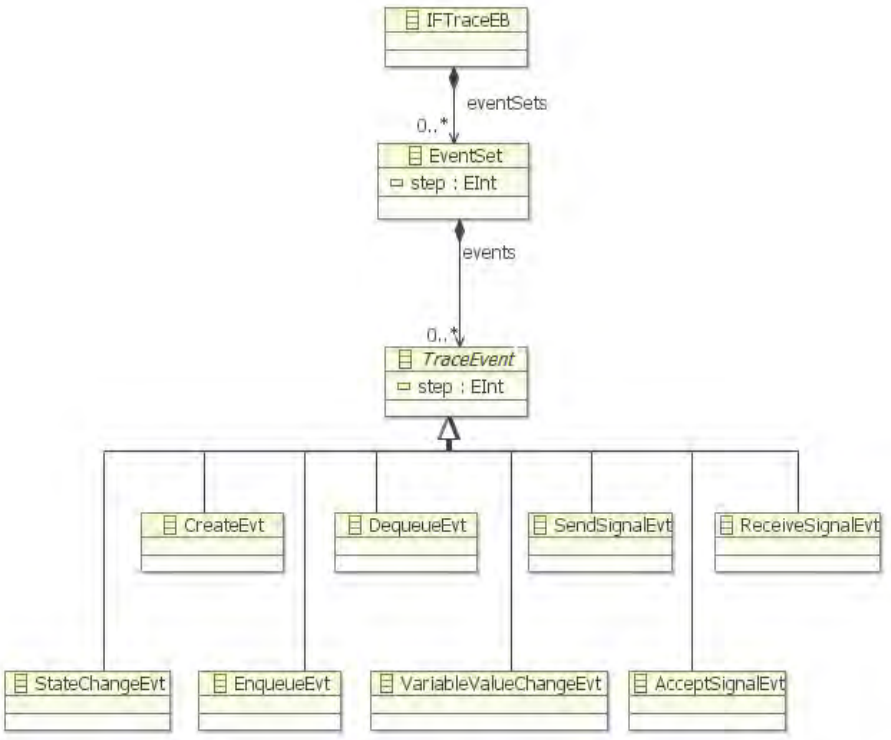


FIGURE 4.31 – Extrait du métamodèle de trace orienté événements IFTraceEB.

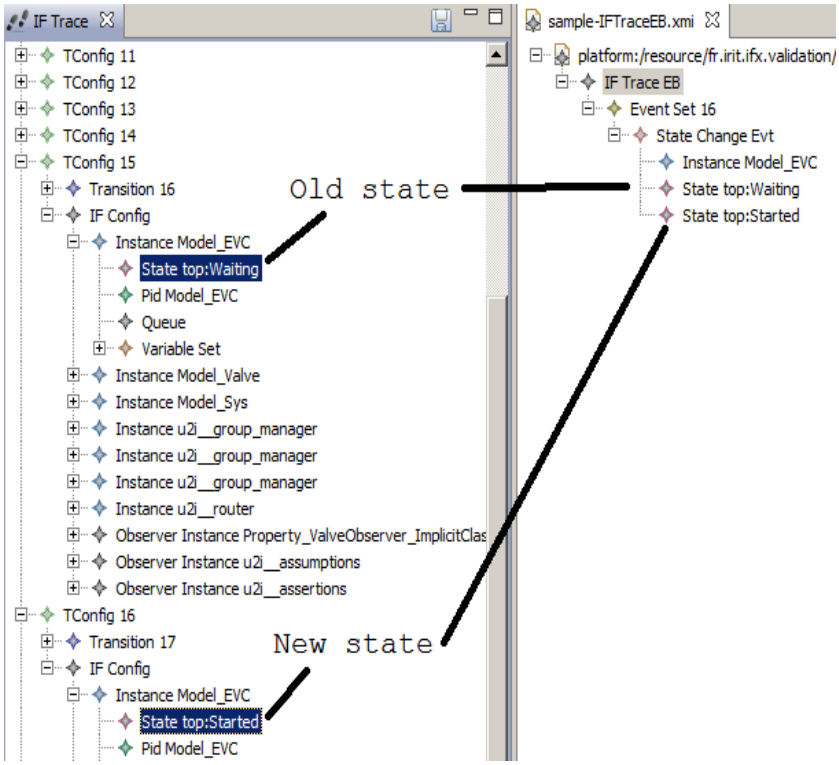


FIGURE 4.32 – Comparaison entre un modèle de trace orienté données (à gauche) et un modèle de trace orienté événements (à droite). L'utilisation d'événements permet de diminuer la complexité des représentations et donc la charge cognitive.

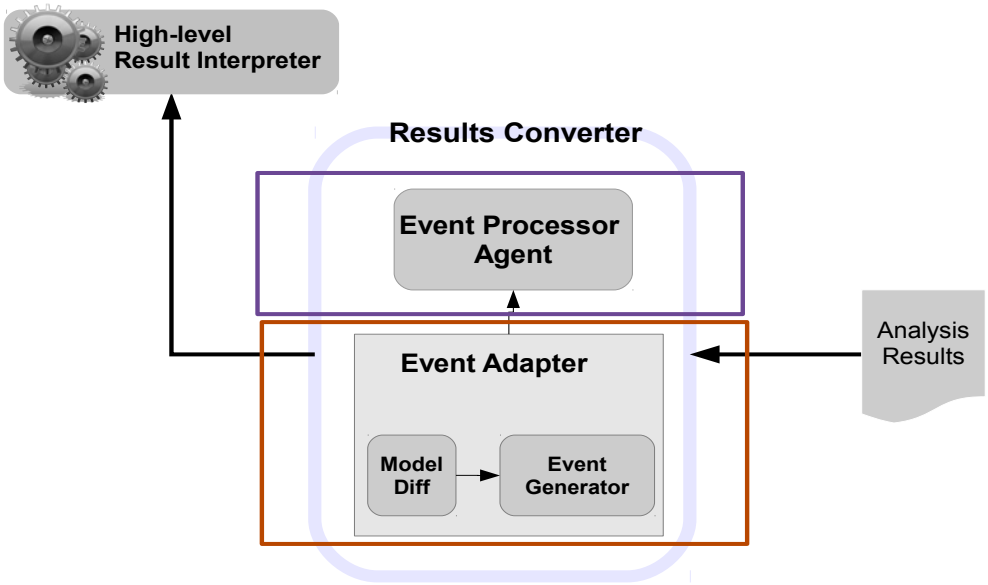


FIGURE 4.33 – Architecture générique pour l'extraction d'événements de haut niveau à partir des résultats d'analyse via le composant *Result Converter*. Ce dernier est composé d'un adaptateur d'événements *Event Adapter* chargé de générer les événements pertinents à partir d'une sémantique opérationnelle de bas niveau, le processeur d'événement *Event Processor Agent* se chargera de la traduction de ces événements vers des événements de haut niveau.

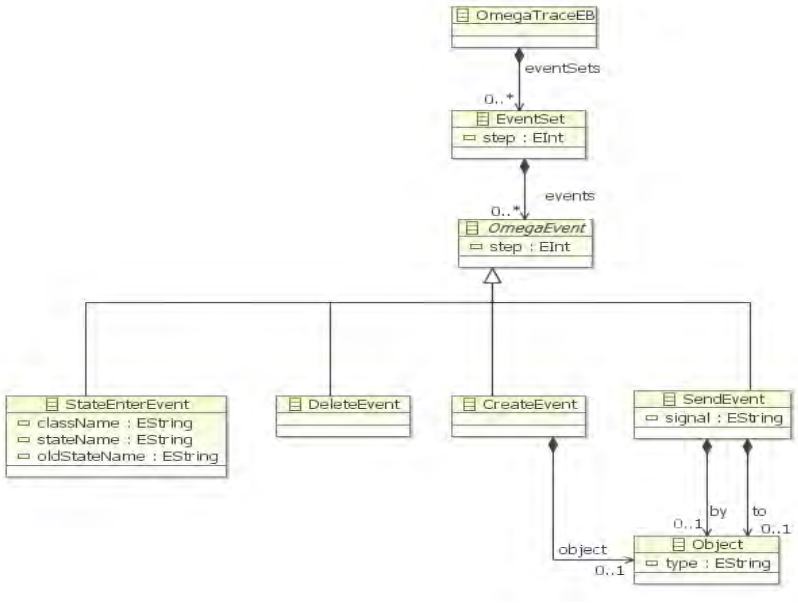


FIGURE 4.34 – Métamodèle OMEGATraceEB qui décrit l'ensemble des événements pertinents dans une sémantique de haut niveau.

4.3.6 Étage de visualisation : événements OMEGA

La prochaine étape dans le pipeline de visualisation est de produire des données visualisables par l'utilisateur. C'est-à-dire des informations dans la sémantique de haut niveau. Nous allons produire à partir des événements extraits de la trace, des événements dans la sémantique du langage OMEGA.

Cet étage est implémenté avec le métamodèle *OmegaTraceEB* (cf. figure 4.34). Ce métamodèle est largement inspiré du modèle d'événements proposé dans le profil OMEGA [77]. Il est composé d'une metaclasse *EventSet* qui représente une collection d'événements de type *OmegaEvent*. Cette dernière metaclasse est implémentée par des metaclasses concrètes représentant chacune un événement pertinent. Par exemple *StateEnterEvent* correspond à l'événement d'entrée dans un nouvel état pour un objet donné. Les attributs de cette metaclasse détaillent le type de l'objet (*className*), le nom de l'état courant (*stateName*) et celui de l'ancien (*oldStateName*).

Opérateurs de visualisation : A partir des modèles de l'étage d'analyse (instance de *IFTraceEB*) nous allons produire des instances du métamodèle *OMEGATraceEB*. Pour cela nous avons implémenté une transformation de modèle en ATL [1]. La transformation est en partie détaillée dans le code source du listing 4.7. La règle

IFEventSet2OmegaEventSet est chargée de transformer les collections d'événements de la trace IF vers ceux de la trace OMEGA en utilisant les règles spécifiques à chaque événement (e.g. *CreateEvt2CreateEvent*, *StateChangeEvt2StateEnterEvent*, etc.) via le polymorphisme et l'héritage de règles.

```
module IFTraceEB_2_OmegaTraceEB;
create OUT: OmegaTraceEB from IN: IFTraceEB;

...

rule IFTraceEB2OmegaTraceEB {
        from
                iFTrace: IFTraceEB!IFTraceEB
        to
                omegaTrace: OmegaTraceEB!OmegaTraceEB (
                        name <- iFTrace.name,
                        eventSets <- iFTrace.eventSets
                )
}

rule IFEventSet2OmegaEventSet {
        from
                sSet: IFTraceEB!EventSet
        to
                tSet: OmegaTraceEB!EventSet (
                        step <- sSet.step,
                        events <- thisModule.getRelevantEvents(sSet.events)
                                -> collect(evt | thisModule.TraceEvent2OmegaEvent(evt))
                )
}

lazy abstract rule TraceEvent2OmegaEvent {
        from
                sEvt: IFTraceEB!TraceEvent
        to
                tEvt: OmegaTraceEB!OmegaEvent (
                        name <- sEvt.name,
                        step <- sEvt.step
                )
}

lazy rule CreateEvt2CreateEvent
                        extends TraceEvent2OmegaEvent {
        from
                sEvt: IFTraceEB!CreateEvt
```

```
        to
                tEvt: OmegaTraceEB!CreateEvent (
                        object <− thisModule.instance2Object(sEvt.instance)
                )
}

lazy rule StateChangeEvt2StateEnterEvent
                                extends TraceEvent2OmegaEvent {
        from
                sEvt: IFTraceEB!StateChangeEvt
        to
                tEvt: OmegaTraceEB!StateEnterEvent (
                        className <− sEvt.instance.type,
                        name <− sEvt.instance.pid.name,
                        stateName <− sEvt.newState.name,
                        oldStateName <− sEvt.oldState.name
                )
}
...
```

Listing 4.7 – Transformation implémentant l'opérateur de visualisation

4.3.7 Étage de la vue : ObjectStory

Pour visualiser les événements OMEGA extraits de la trace, nous avons développé un métamodèle qui implémente la notation graphique conçue précédemment (cf. section 4.3.2). Il s'agit d'un vocabulaire graphique à trois mots, chacun dénotant un des trois événements pertinents. La figure 4.35 montre les différentes classes du métamodèle *ObjectStory*. Un modèle *ObjectStory* est composé d'une séquence ordonnée de *Shots*.

Un *Shot* est un pas de la trace dans lequel un ensemble d'événements se sont déroulés. Pour tout événement pertinent dans cet ensemble une instance concrète de *VisualEvent* sera créée. Trois classes concrètes implémentent la classe *VisualEvent* : *NewObject*, *ObjectStateChange* et *SignalSend*. Ces classes représentent respectivement la création d'une nouvelle instance d'objet OMEGA, le changement d'état d'un objet, et l'envoie d'un message.

Opérateur de mapping visuel : il permet de rajouter les informations spécifiques à une librairie graphique donnée. Cet opérateur est implémenté par la transformation listée dans 4.8. La figure 4.36 montre le résultat de l'exécution d'une des règles de cette transformation.
Cet étage est interprété par une vue graphique de l'outil IFx-Workbench pour donner

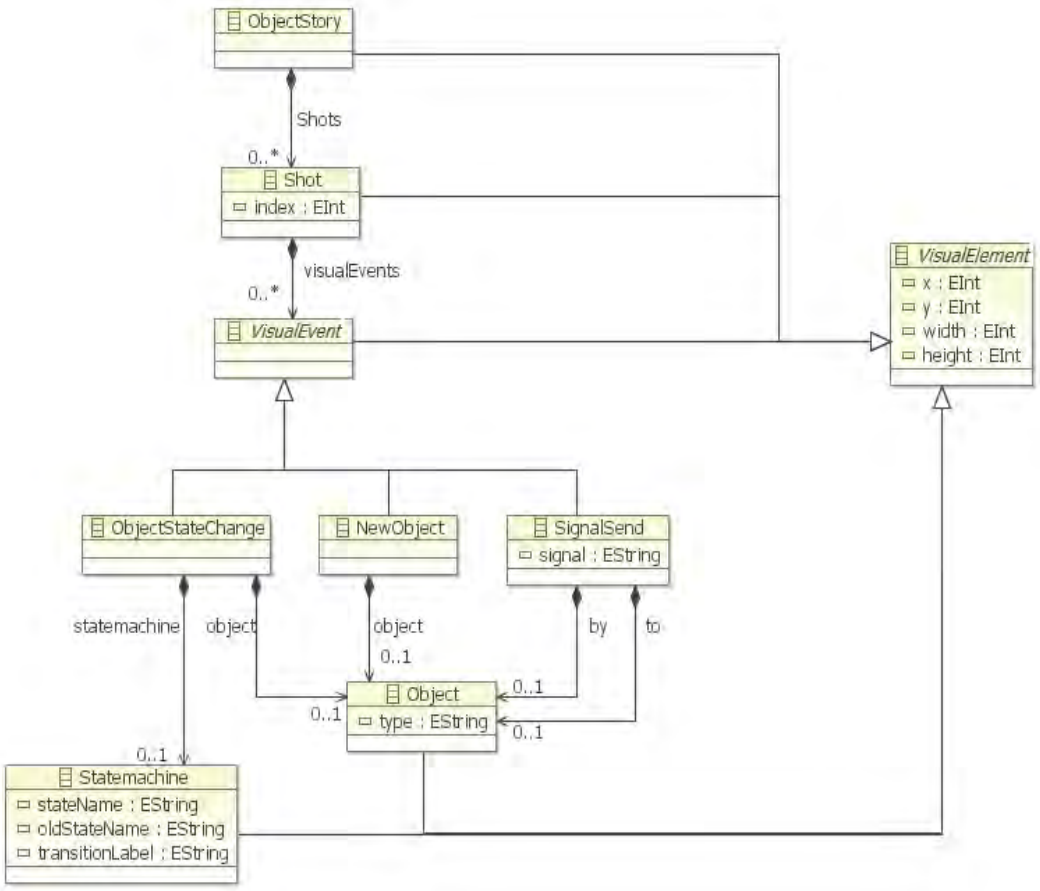


FIGURE 4.35 – Le métamodèle ObjectStory qui décrit l'ensemble des éléments visuels réalisant l'étage de la vue.

le résultat de la figure 4.37.

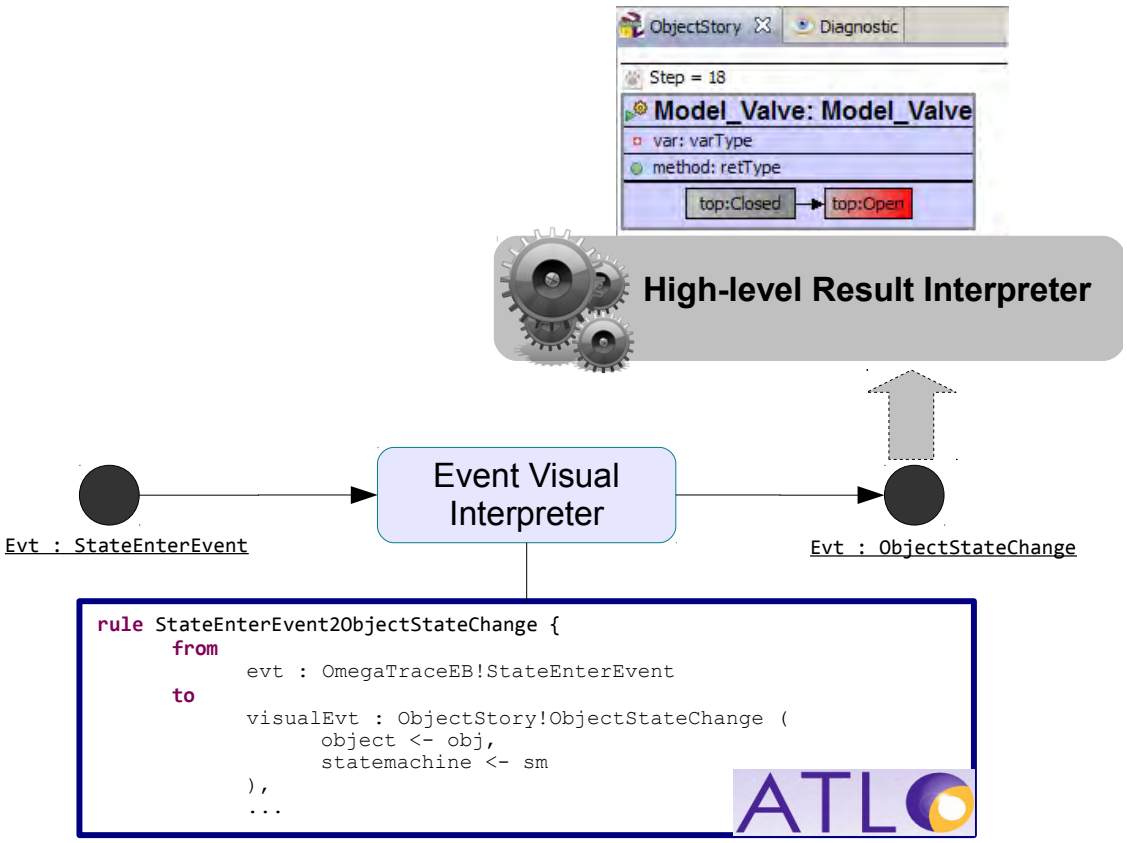


FIGURE 4.36 – Affichage graphique des événements de haut niveau. L'interprète des événements visuels (*Event Visual Interpreter*) exécute une transformation de modèle pour générer le modèle d'entrée vers le visuel rendu.

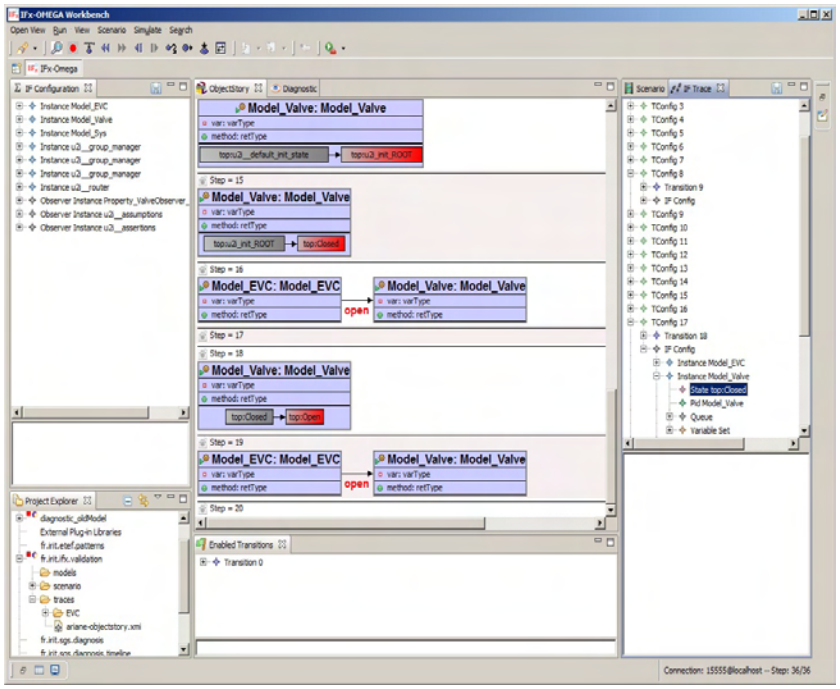


FIGURE 4.37 – Résultat de la visualisation de trace ObjectStrory

```
module OmegaTraceEB_2_ObjectStory;
create OUT: ObjectStory from IN: OmegaTraceEB;

rule OmegaTraceEB2ObjectStory {
        from
                oTrace: OmegaTraceEB!OmegaTraceEB
        to
                omegaTrace: ObjectStory!ObjectStory (
                        Shots <- oTrace.eventSets
                )
}
rule EventSet2Shot {
        from
                eSet : OmegaTraceEB!EventSet
        to
                shot : ObjectStory!Shot (
                        index <- eSet.step,
                        visualEvents <- eSet.events
                )
```

```
}

rule CreateEvent2NewObject {
        from
                evt : OmegaTraceEB!CreateEvent
        to
                visualEvt : ObjectStory!NewObject (
                        object <- evt.object
                )
}
rule SendEvent2SignalSend {
        from
                evt : OmegaTraceEB!SendEvent
        to
                visualEvt : ObjectStory!SignalSend (
                        signal <- evt.signal,
                        by <- evt.by,
                        to <- evt.to
                )
}
rule StateEnterEvent2ObjectStateChange {
        from
                evt : OmegaTraceEB!StateEnterEvent
        to
                visualEvt : ObjectStory!ObjectStateChange (
                        object <- obj,
                        statemachine <- sm
                ),
                obj : ObjectStory!Object (
                        type <- evt.className,
                        name <- evt.name
                ),
                sm : ObjectStory!Statemachine (
                        oldStateName <- evt.oldStateName,
                        stateName <- evt.stateName
                )
}
...
```

Listing 4.8 – Transformation implémentant l'opérateur de mapping visuel

4.4 Évaluation empirique

Dans cette section nous allons proposer un protocole d'évaluation du résultat de l'exécution du processus générique sur un cas réel, à savoir le diagnostic d'un modèle de contrôleur électronique de valve.

4.4.1 Techniques d'évaluation des visualisations

Afin d'évaluer les avantages et les limitations d'une technique de visualisation, nous devons apprécier son utilisation. Pour cela plusieurs techniques peuvent êtres utilisées. Certaines d'entres-elles sont subjectives, alors que d'autres utilisent des approches quantitatives [69]. Par ailleurs, les techniques d'évaluations peuvent être catégorisées en fonction des profils des participants et des paramètres observés.

D'une part, il y a les techniques telles que l'*inspection cognitive* et l'*évaluation heuristique* qui impliquent des spécialistes en ergonomie et des facteurs humains. D'autre part, il y a les techniques d'observations ou d'expérimentations basées sur une participation d'un échantillon d'utilisateurs.

Le premier objectif de notre étude est de rendre notre outil d'analyse des modèles plus accessible à un large public. A ce jour, l'écrasante majorité des modélisateurs ne sont pas familiers avec les techniques d'analyses telles que le model checking. Ainsi, pour déterminer si cet objectif a été atteint, nous devons évaluer l'approche sur un échantillon d'utilisateurs conformes à ce profil. De plus, en raison de notre orientation vers une approche quantitative, une *expérience contrôlée* impliquant la participation d'utilisateurs sera effectuée. Cependant, puisque cette technique ne permet pas une analyse détaillée de l'impression des utilisateurs en termes de satisfaction, notre validation sera complétée par une approche subjective. Pour cela, nous utiliserons un système d'évaluation éprouvé et largement utilisé, à savoir le System Usability Scale (SUS [39]). Ce système se présente sous forme d'un questionnaire dont les réponses sont comparées à une échelle de référence.

Dans les deux sections 4.2.4 et 4.3, plusieurs visualisations ont été construites pour montrer l'efficacité du framework Metaviz ainsi que du processus générique. Dans cette section nous proposons un protocole d'évaluation empirique appliqué à une de ces visualisations, à savoir ObjectStory.

4.4.2 Évaluation empirique

La question centrale à laquelle répond l'évaluation est la suivante : **Les visualisations conçues avec l'outillage proposé et respectant les recommandations**

émises dans la section 4.1.2, permettent-elles une meilleure efficacité de l'utilisateur dans la tâche de diagnostic ?

4.4.2.1 L'utilisabilité dans le contexte des méthodes formelles

Avant d'énoncer les hypothèses et de concevoir une expérience, nous devons comprendre la notion d'utilisabilité dans le cadre des techniques d'analyse formelles. La norme ISO [90] définit l'utilisabilité comme : **La mesure dans laquelle un système, un produit ou un service peut être utilisé par des utilisateurs spécifiés dans un contexte donné d'utilisation, pour atteindre des buts définis avec efficacité, efficience et satisfaction.** Contextualiser la définition de l'utilisabilité définit les trois caractéristiques d'efficacité, d'efficience et de satisfaction dans le contexte des méthodes formelles :

- l'efficacité est la capacité des utilisateurs à comprendre, à localiser et à corriger les erreurs signalées par l'outil d'analyse ;
- l'efficience est le temps et l'effort cognitif nécessaires pour effectuer chacune des trois tâches précitées ;
- la satisfaction est l'impression subjective des utilisateurs après l'utilisation de la suite d'outils qui prend en charge les trois tâches.

4.4.2.2 Formulation des hypothèses

Comme mentionné plus haut, notre objectif est d'augmenter l'utilisabilité de l'outil d'analyse. Pour ce faire, nous proposons de les étendre avec des fonctionnalités de visualisation. Ainsi, nous pouvons formuler l'hypothèse suivante :

hypothèse H. **Aider efficacement l'utilisateur à comprendre les résultats d'analyses des modèles, facilite l'utilisation de l'outil d'analyse.** La notion d'utilisabilité de l'outil d'analyse a été définie dans la section précédente. Sur cette base et au vue de la définition de la notion d'utilisabilité, notre hypothèse H peut être affinée :

- *H1.* Les participants utilisant la nouvelle extension de l'outil mettront moins de temps à comprendre la trace ;
- *H2.* Les participants utilisant la nouvelle extension de l'outil auront une meilleure compréhension de la trace ;

- *H3*. Les participants utilisant la nouvelle extension de l'outil passeront moins de temps à localiser les erreurs dans la trace ;
- *H4*. Les participants utilisant la nouvelle extension de l'outil localiseront l'erreur dans la trace avec plus de précision ;
- *H5*. Les participants utilisant la nouvelle extension de l'outil mettront moins de temps à comprendre la (ou les) cause(s) d'erreur(s) ;
- *H6*. Les participants utilisant la nouvelle extension de l'outil auront une meilleure compréhension de la (ou des) cause(s) d'erreur(s).

Ces hypothèses affinées permettront d'analyser avec plus de précision les résultats de l'expérimentation.

4.4.2.3 Définition d'un protocole de validation

Dans cette section, des détails précis sur la conception de l'expérience seront donnés en définissant plusieurs de ses caractéristiques. La terminologie utilisée dans cette section est empruntée à [101].

4.4.2.4 Les participants

Les participants ont été choisis parmi les étudiants de Master et Doctorat de l'Université de Toulouse. L'expérience a été menée avec 10 sujets répartis en deux groupes. Le groupe expérimental et le groupe témoin, respectivement appelé groupe A et groupe B, dans les sections suivantes. Tous les participants étaient déjà familiarisés avec UML, deux d'entre eux ont déjà utilisé un outil d'analyse formelle, mais aucun d'entre eux n'a utilisé le nôtre. Afin d'évaluer leur adéquation avec la population des utilisateurs, les participants ont été invités à remplir un questionnaire. Le questionnaire comporte 7 parties et 12 questions portant sur le niveau éducatif, l'expérience en modélisation et dans les techniques de vérification de modèles.

Dans le cadre de l'évaluation préliminaire de l'homogénéité des groupes A et B, nous avons obtenu les résultats suivants :

- Groupe A : médiane = 4 ; moyenne = 4,4 ; écart-type = 1,67 ;
- Groupe B : médiane = 4 ; moyenne = 4,6 ; écart-type = 1,52

Ces résultats montrent que les deux groupes sont composés d'une population homogène, nous permettant une exploitation des résultats obtenus sur l'utilisation de l'outil d'analyse [35].

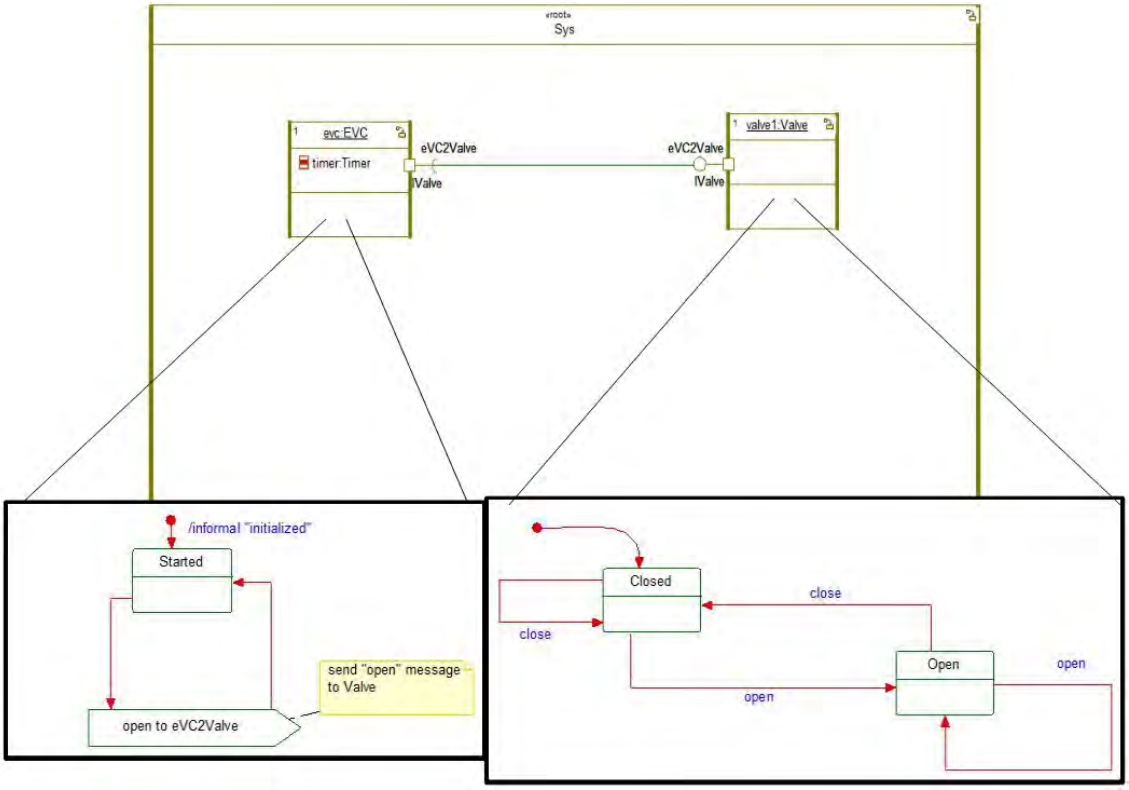


FIGURE 4.38 – Modèle utilisé pour l'expérimentation. Il s'agit un contrôleur électronique de valve.

4.4.2.5 Unité expérimentale

Les participants ont été invités à visualiser la trace d'exécution d'un petit modèle OMEGA représentant un contrôleur de valve électronique (cf. figure 4.38). Une propriété temporelle a été spécifiée sur le modèle. Cette propriété stipule que les directives d'ouverture (message *open*) envoyées par le contrôleur électronique de valve à la valve doivent être espacées d'au moins 5s (étapes dans la trace). Le modèle OMEGA est alors utilisé comme entrée pour le model checker IF.

Comme le modèle ne satisfait pas la propriété spécifiée, le model checker génère un contre-exemple. C'est ce contre-exemple qui sera exploité par les 2 groupes pour l'expérimentation.

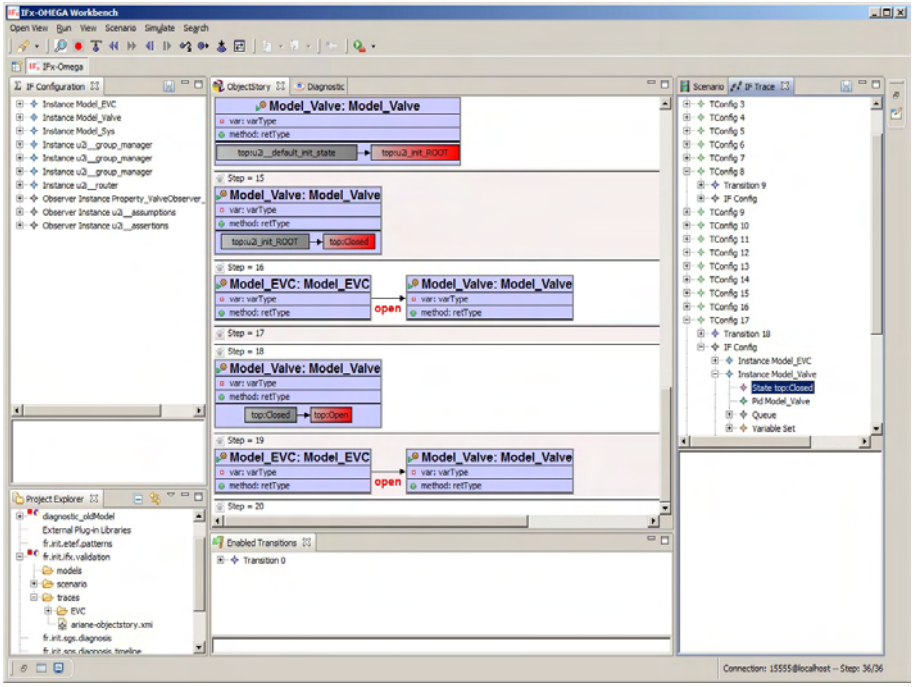


FIGURE 4.39 – Support utilisateur pour l'analyse de traces dans IFx-Workbench

4.4.2.6 Variables expérimentales

La variable expérimentale correspond au logiciel utilisé par le participant pour explorer les résultats de l'analyse. Nous considérons l'outil d'analyse comme étant cette variable. Cet outil offre un support pour l'exploration des traces IF aux côtés des visualisations complexes. La figure 4.39 montre une capture d'écran du support pour l'analyse de traces. Le panneau de droite contient le navigateur de trace d'erreur, qui est similaire à ce qui est disponible dans d'autres outils d'analyses. Le panneau du milieu contient la vue ObjectStory des traces.

4.4.2.7 Facteurs

Les facteurs (ou variables indépendantes) sont les variables que nous allons manipuler dans l'environnement des participants pour voir comment d'autres variables, dites variables de réponse sont affectées. Notre expérience a pour but d'analyser si un support avancé pour les utilisateurs aurait une incidence sur leur compréhension du résultat de l'outil d'analyse. Ainsi, nous effectuons l'expérience avec un facteur à deux niveaux : l'outil d'analyse avec un support élaboré de visualisation de la trace

activé (niveau 1) et désactivé (niveau 2). Par conséquent, les 2 niveaux de la variable indépendante sont :

- niveau 1 : seuls les vues de base sont activés ;
- niveau 2 : la visualisation avancée est activée

4.4.2.8 Variables de réponse

Les variables de réponse correspondent aux aspects expérimentaux impactés. Pour mesurer leurs valeurs quantitatives, nous avons conçu un ensemble de tâches pour l'utilisateur, distribuées en trois catégories. Chaque catégorie de tâches correspond à une variable de réponses. Les catégories sont liées aux tâches cognitives suivantes :

- comprendre la trace ;
- localiser l'erreur ;
- comprendre la cause de l'erreur et corriger le modèle de départ.

Pour chaque catégorie, nous évaluons deux caractéristiques : la vitesse d'exécution et la qualité des réalisations.

4.4.2.9 Définition des tâches utilisateur

L'ensemble des tâches que chaque participant doit effectuer, a été conçu pour couvrir la compréhension de la trace, la compréhension de l'erreur et puis de sa cause. Chaque participant dispose d'un ensemble de sept tâches à effectuer à l'aide de l'outil d'analyse des traces. De plus, le temps passé par chaque participant pour exécuter une tâche, est compté par un chronomètre. Nous ne prenons pas en compte le temps passé par le participant à noter ses réponses. Le tableau 4.6 donne un aperçu des tâches.

4.4.2.10 Résultats

Le but de notre expérience est de voir si l'extension d'un outil d'analyse avec le mécanisme de visualisation *ObjectStory* précédemment décrit en section 4.3.4, améliore l'exploitation des résultats d'analyses. Les figures 4.40 et 4.41 présentent les résultats pour chaque tâche, en termes de temps et de qualité des réponses. Les participants qui utilisent l'extension *ObjectStory* appartiennent au groupe A (groupe expérimental), tandis que le groupe B (groupe témoin) correspond à des participants utilisant la version basique de l'outil.

Type cognitif	Tâche
Compréhension de la trace	1. Quelles sont les instances créées dans cette trace et à quelles étapes
	2. Quels sont les messages échangés, à quelles étapes et par quelles instances
	3. A quelle étape de la trace, l'instance Model_Valve passe dans l'état Closed
Localisation de l'erreur	4. Citer l'étape où la propriété à vérifier est violée
Compréhension de la cause	5.Quel diagramme devrait être modifié pour éviter cette erreur
	6. Expliquer de manière informelle la modification que vous voulez apporter
	7. Modifier le diagramme pour corriger l'erreur de vérification (la syntaxe est libre)

TABLE 4.6 – Tâches par catégorie cognitive. Les participants ont été invités à effectuer un ensemble de tâches qui s'étend sur trois types de tâches cognitives.

Temps consacré à chaque tâche : Les résultats obtenus montrent que les participants du groupe A ont effectué les tâches T1 à T3, 5 fois plus rapidement que les participants du groupe B. La tâche T4 a été accomplie par le groupe A, 8 fois plus rapidement que le groupe B, tandis que pour la dernière série de tâches allant de T5 à T7, le groupe A a été 2 fois plus rapide.

Le taux global d'amélioration est donc de 400%. Cette augmentation importante dans la vitesse d'exécution des tâches est due à la nature cognitive de chaque catégorie. Par exemple, on peut remarquer que le groupe expérimental (groupe A) a été 8 fois plus rapide dans l'exécution de la tâche T3. Cette tâche est la plus exigeante en charge cognitive pour les participants. En effet, il y était demandé aux participants de trouver l'étape dans la trace où une certaine instance entre dans un état particulier. Le participant devait se rappeler des instances et des noms des états accédés tout en parcourant étape par étape l'ensemble de la trace. C'est là que nous pouvons voir l'efficacité des visualisations qui ne présentent que ce qui a changé dans la trace.

La visualisation *ObjectStory* se concentre sur le changement d'état des instances pertinentes et filtre une grande partie des informations. La figure 4.39 montre la visualisation de base aux côtés d'une visualisation plus complexe. Le vocabulaire visuel utilisé est très intuitif (cf. section 4.2.1) ce qui permet au groupe de contrôle

d'effectuer la tâche T3, 8 fois plus rapidement.

Qualité des réponses : En ce qui concerne la qualité, les résultats ont révélé une augmentation nette de la qualité des réponses des participants. Les résultats pour la compréhension de la trace (i.e. tâches T1 à T3) montrent une amélioration de 25% dans la compréhension des interactions entre les instances, comme l'envoi de message. Pour la localisation l'erreur (tâche T4), l'amélioration est d'environ 9%. Pour la compréhension de la cause de l'erreur, l'amélioration est de 40%. La figure 4.41 montre la qualité des réalisations pour chaque tâche.

Pour compléter notre analyse, nous avons effectué une évaluation de l'impression des participants à l'aide d'un questionnaire. Cette évaluation a porté sur la satisfaction des utilisateurs. A cet effet, nous nous sommes appuyés sur le test SUS [39]. Nous avons choisi cette méthode d'analyse car elle est simple, rapide et fiable [152]. Les résultats de satisfaction des utilisateurs sont légèrement plus élevés pour le groupe expérimental (groupe A) avec 68% contre 61% pour le groupe témoin (Groupe B). Les résultats présentés dans le tableau 4.7 montrent ainsi une augmentation d'environ 11% de la satisfaction des utilisateurs.

Test des hypothèses : Les déviations standards calculées pour les 2 groupes (cf. 4.4.2.4) étant quasi identique, nous pouvons tester l'hypothèse nulle en appliquant le test de *Student*. Les résultats sont présentés dans le tableau 4.8. L'expérimentation a impliqué 5 participants par groupe, ce qui donne 8 degrés de liberté et une valeur de $t_{0,99}$ à 3,355. Toutes les valeurs calculés sont en dessous de $t_{0,99}$, nous pouvons donc rejeter l'hypothèse nulle. Le résultat pour la satisfaction des utilisateur (cf. tableau 4.7) invalide l'hypothèse nulle sur l'amélioration de la satisfaction. Nous pouvons donc accepté les hypothèses initialement formulées. Par conséquent, aider efficacement l'utilisateur à comprendre les résultats d'analyses des modèles, facilite l'utilisation de l'outil d'analyse.

	System Usability Scale
Group A	67,7
Group B	61
t	**0,71**

TABLE 4.7 – Tests d'utilisabilité selon l'échelle SUS [39]. Les résultats montrent une augmentation de 11% de l'utilisabilité pour le groupe expérimental (groupe A). L'application du test statistique de *Student* montre la validité des résultats.

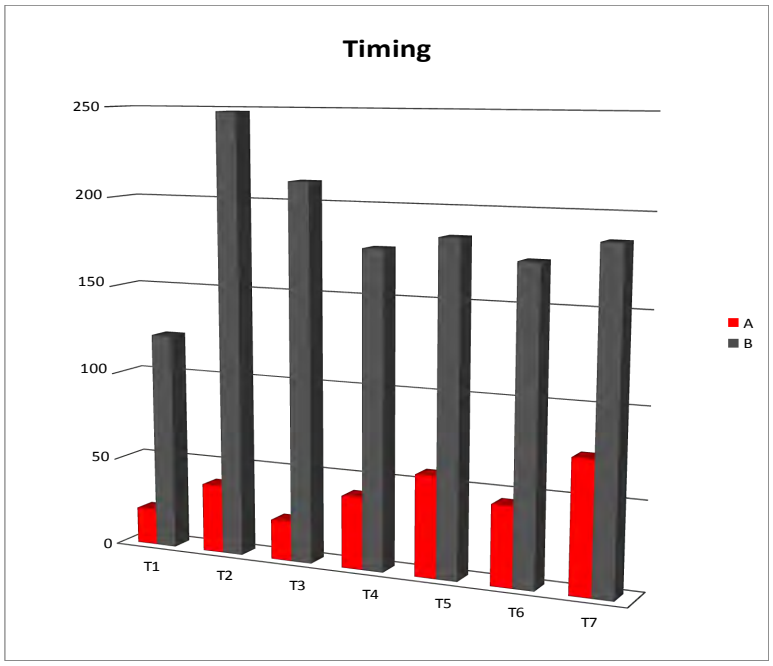


FIGURE 4.40 – Temps (en secondes) pour effectuer chaque tâche. Les participants qui utilisent une visualisation élaborée effectuent les tâches 4 fois plus rapidement.
A : groupe expérimental, utilise une visualisation élaborée
B : groupe contrôle, utilise une visualisation de base

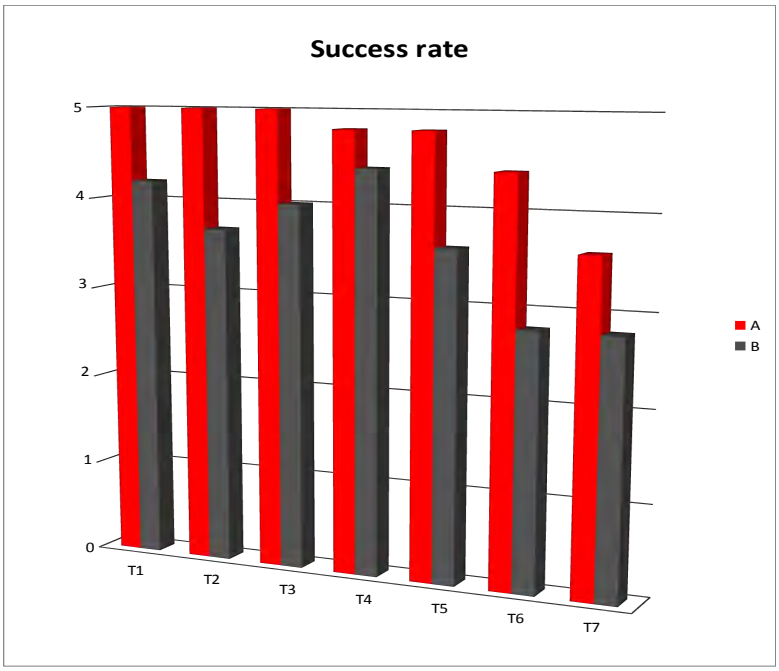


FIGURE 4.41 – Taux de réussite pour chaque tâche. Les Participants qui utilisent la procédure de visualisation élaborée produisent des réponses de meilleure qualité.
A : groupe expérimental, utilise une visualisation élaborée
B : groupe contrôle, utilise une visualisation de base

	Compréhension de la trace (T1 to T3)		**Compréhension de l'erreur (T4)**		**Compréhension de la cause (T5 to T7)**		
	Qualité	**Temps**	**Qualité**	**Temps**	**Qualité**	**Temps**	
Group A	5	44,33	5	18	5	35	
	5	27,33	5	21	4,33	55,67	
	5	27	5	59	5	30,67	
	5	8,33	5	51	3,33	62,67	
	5	29,67	4	55	3,67	110	
Group B	5	103	5	20	3	100	
	3,17	325	5	540	0	600	
	5	183,33	5	27	5	90,33	
	5	120,67	5	32	5	23,33	
	1,67	238	2	267	2,33	94,33	
t	**0,16**	**0,004**	**0,54**	**0,22**	**0,26**	**0,28**	

TABLE 4.8 – Résultats statistiques par catégorie cognitive. Les résultats montrent les moyennes et le test *Student* sur les hypothèses de chaque catégorie de tâches cognitives.

4.4.2.11 Validité

Pour tester la validité de nos conclusions, nous avons utilisé des tests statistiques adaptés, à savoir le test *Student*. Nos échantillons ont le même nombre de personnes et des écarts similaires ce qui oriente le choix de la méthode de test vers le test *Student* [35].

Le rôle principal des visualisations de traces est de réduire la quantité de données à un ensemble pertinent d'information pour une tâche spécifique de l'utilisateur, évitant ainsi la surcharge cognitive. La visualisation joue aussi un autre rôle. Elle apporte aux utilisateurs des informations orientées de domaine, elle utilise pour cela des concepts déjà connus par l'utilisateur. Aucun effort supplémentaire n'est nécessaire de la part de l'utilisateur pour comprendre les notations et la sémantique des constructions de visualisation (surcharge de la perception). Ce sont ces avantages de la visualisation de trace qui expliquent les résultats obtenus par le groupe contrôle.

Validité interne : Les statistiques présentées dans la section 4.4.2.4 assurent qu'il n'y a pas de différences significatives dans le niveau technique des participants.

Validité externe : même si l'expérimentation a porté sur un modèle de petite taille, la technique est susceptible d'accélérer l'exploration et la compréhension de trace pour des modèles plus grands. Cependant, nous pensons que pour des experts, les améliorations pourraient ne pas être aussi importantes. En effet nous devrions rester prudents concernant cette généralisation. L'effet d'inversion liée à l'expertise découvert par Sweller [154] pourrait survenir dans notre contexte. Dans la section problématique 2 nous expliquons que le profil utilisateur impliqué dans notre contexte est celui de l'utilisateur non initié à la syntaxe de bas niveau. Des expériences supplémentaires devront être menées pour évaluer les résultats avec des utilisateurs experts

Chapitre 5

Conclusions et perspectives

5.1 Rappel de la problématique

Les langages de modélisation ainsi que les formalismes de validation, par leur degré d'abstraction, apportent une solution de gestion de la complexité et fournissent de puissantes techniques d'analyses. Cependant le processus et l'outillage de diagnostic, permettant de passer des résultats d'analyse à la correction des modèles de départ, reste limité.

L'utilisation des langages comme UML, SysML ou SDL pour la modélisation des systèmes embarqués critiques permet d'exploiter les outils d'analyse existants et de vérifier les modèles produits. C'est la compréhension de ces résultats d'analyse qui reste aujourd'hui une tâche difficile.

L'adoption d'une approche IDM dans l'ingénierie système, est conditionné par l'efficacité du support utilisateur durant les phase de diagnostic. Ce support devra s'intégrer de manière simple dans les chaînes d'outils existantes. Les outils actuelles de validation de modèles sont largement basés sur l'approche de sémantique par traduction. Dans cette optique, les modèles utilisateur sont alors traduits vers une sémantique formelle de bas niveau, nécessaire aux activités d'analyse. Les résultats sont souvent rapportés à l'utilisateur dans la sémantique de bas niveau induisant une **surcharge cognitive** de l'utilisateur due à la quantité d'information générée. Certains outils proposent une traduction inverse des résultats vers le formalisme de haut niveau. Cette dernière solution est en effet très prometteuse pour autant que l'on considère les spécificités cognitives de l'utilisateur. En effet, **la perception humaine est un puissant levier encore très peu exploité dans les outils de spécification et vérification formelles**.

5.2 Résumé des contributions

Nous proposons dans cette thèse une approche systématique pour la conception et l'implémentation de mécanismes de diagnostic dans le cadre des approches d'analyse par traduction.

L'approche bénéficie d'une grande flexibilité liée à l'utilisation des techniques IDM, l'extension d'outils existant est alors aisée [18].

Nous avons appliqué ces techniques à l'outil IFx-OMEGA de validation de modèles UML/SysML en développant une extension pour l'analyse de traces de simulation et de contre-exemples de model checking. L'extension, sous forme de visualisations de modèles de trace orienté événements a fait l'objet d'une évaluation empirique avec la participation d'utilisateurs. Cette évaluation a montré une nette amélioration de l'efficacité de la phase de diagnostic [19].

Notre approche systématique se résume à :

- **un ensemble de recommandations** [19] pour la conception d'outils de diagnostic efficace. Ces recommandations permettent de prendre en compte des spécificités du système cognitif et perceptif humain pour éviter la surcharge cognitive de l'utilisateur.

- **un processus et un framework** [18] pour concevoir et développer des visualisations de diagnostic, adaptées à la tâche de diagnostic.

- **un protocole de validation expérimentale** des outils de diagnostic [19] qui prend en compte les aspects d'efficacité de l'utilisateur mais aussi de satisfaction.

L'outillage proposé est extensible. En effet, dès qu'une nouvelle erreur récurrente est détectée, le processus permet de savoir quel type d'interface développer pour diagnostiquer cette erreur ainsi que les artefacts nécessaires à l'implémentation de cette interface (transfos, metamodèles, vues). Nous nous sommes basés sur les travaux de psychologie cognitive, de visualisation d'information et d'ergonomie d'interfaces pour extraire les recommandations de conception pour des visualisations de diagnostic efficaces. La partie support outillé est définie dans une démarche IDM et repose sur le framework Metaviz. Les outils de diagnostic proposés ne seront pas forcément en adéquation cognitive avec les experts. L'effet de renversement dû à l'expertise [103] suggère même qu'une optimisation des outils pour le novice pourrait altérer les performances des experts.

5.3 Perspectives

Les résultats obtenus sur l'efficacité de l'approche proposée sont valables pour un profil d'utilisateur précis. Nous avons pris comme référence un utilisateur novice dans la sémantique de bas niveau (i.e. la sémantique cible de la traduction). La validation expérimentale conduite porte sur un type d'erreur précis avec un modèle de taille restreinte. D'autres expérimentations doivent être réalisées pour permettre l'extension des résultats. L'évaluation avec des utilisateurs experts doit aussi être réalisée pour étendre les résultats et prouver l'efficacité de l'approche pour des utilisateurs experts.

Le processus d'utilisation de l'outillage proposé dépend des erreurs à diagnostiquer et du profil de l'utilisateur. Nous pourrions donc imaginer un processus qui s'adapte automatiquement à différents contextes. Une possibilité que nous avons commencé à étudier est celle de l'utilisation de *systèmes de recommandations*[26, 20] pour assister l'utilisateur dans le processus de vérification et de diagnostic en fonction de son profile et du types d'erreurs.

Concernant les visualisations, l'ajout de nouvelles visualisations qui répondent à d'autres types d'erreur serait pertinent. D'autre part, l'ajout d'un mécanisme permettant de répertorier et de mettre en correspondance les types d'erreur avec les outils à utiliser pour les diagnostiquer et les corriger améliorerait grandement l'utilisabilité.

Une autre amélioration de l'assistance à la correction des modèles de départ serait de suggérer des corrections automatiques à l'utilisateur, à l'image de ce qui est proposé dans les environnement de développement des langages de programmation (e.g. les *Quick fixes* dans Eclipse).

Pour améliorer la compréhension des résultats d'analyse, une méthode de diagnostic par extraction de liens de causalité pourrait être envisagée [115, 72]. Elle permettrait de spécifier les liens de causalité entre événements. Par exemple la réception d'un message à pour cause l'envoi de ce même message, deux événements qui ne sont pas forcément contigus dans les contre-exemples générés par les model checker. Cette approche pourrait aussi être automatique, ainsi des liens de causalité par défauts serait injectés dans les modèles de traces permettant à l'utilisateur de remonter plus rapidement dans les chaînes causales vers la cause de l'erreur.

Annexes

Annexe A

Questionnaire pour la définition du profil utilisateur

Dans la section 4.4 nous avons soumis les interfaces de diagnostic conçues à des utilisateurs. Pour définir leur profiles nous leur avons soumis un ensemble de questions, détaillées ci-dessous.

11/02/2013 IRIT salle des usages
IFx-Workbench V030
Validation d'IHM
Profil Utilisateur

Veuillez répondre aux questions suivantes :

1. **Sexe : F M**
2. **Age : .....**
3. **Niveau d'étude: Master / Thésard / autres :.....**

4. **Quels sont vos languages de programmation préférés ?**
   .....
5. **Durant votre cursus universitaire, avez-vous eu des cours sur UML ?**
   Oui Non
6. **Avez vous déjà utiliser UML ?** Oui Non
   1. **Quels types de diagrammes avez-vous utilisez ?**
      Classe / Statecharts / Activité / autres : .....
   2. **Pour quel domaine d'application ?**
      1. Informatique de gestion
      2. Applications mobiles
      3. Systèmes embarqués
      4. Autre : .....
   3. **Dans quel contexte ?** En entreprise / à l'université / autres : .....
   4. **Comment jugez vous votre niveau de connaissance en UML ?**
      Excellent / Bon / Moyen / Débutant
   5. **Pour de la génération de code ?**
      1. Avec quel outil ? .....
      2. Vers quel language ? .....

7. **Avez vous déjà utiliser des techniques de vérification de modèles ?**
   Oui Non
   1. **Lesquelles ?**
      1. Model checking
      2. OCL
      3. Autres .....
   2. **Avec quels outils ?**
      1. Uppaal
      2. SPIN
      3. NuSMV
      4. IFx
      5. Autres .....

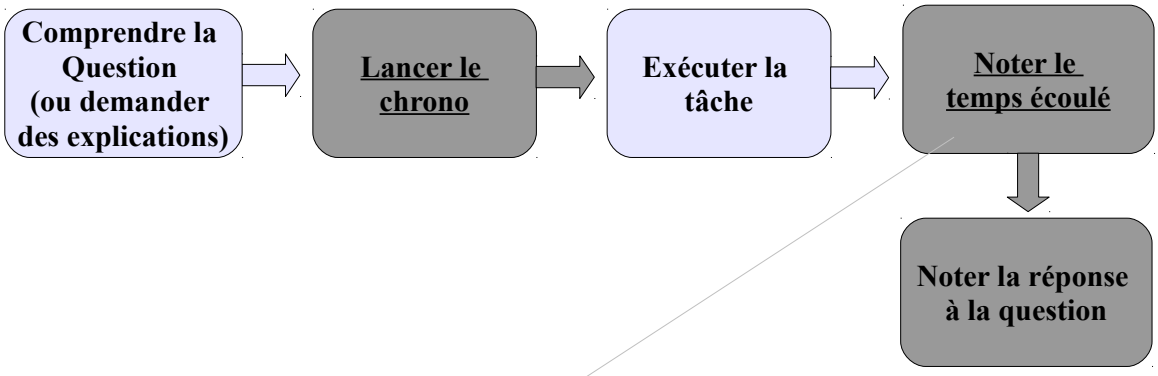
Annexe B

Ensemble des tâches utilisateur soumises pour la validation expérimentale

L'ensemble des tâches que chaque participant a effectuées couvre la compréhension de la trace, la compréhension de l'erreur ainsi que la compréhension de la cause de l'erreur. Le questionnaire conçu à cet effet est décrit ci-dessous.

11/02/2013 IRIT salle des usages
IFx-Workbench V030
Validation d'IHM
Tâches

N'oubliez pas de chronométrer chaque tâche. Vous devez suivre le processus suivant pour chaque tâche.



Compréhension de la trace :

1. Quelles sont les instances créées dans cette traces et à quelle étape (seuls les instances commençant par *Model_* sont à prendre en compte) ?
   Temps : ….....
2. quels sont les messages échangés, à quelle étape et par quelles instances ?
   ex: l'instance I1 envoie le message "m" à l'instance I2 à l'étape 13
   remarque : ne pas considérer les messages commençant par *u2i_*
   Temps : ….....
3. A quelle étape de la trace, l'instance *Model_Valve* passe dans l'état *Closed ?*
   Temps : ….....

Compréhension de l'erreur :

1. La trace que vous visualisez ne satisfait pas la propriété à vérifier (rappel de la propriété : pas de messages « *open* » espacés de moins de 5 étapes). Citez l'étape ou vous voyer l'erreur.
   Temps : ….....

Correction du modèle :

1. Quel diagramme devrait être modifié pour éviter cette erreur ?
   Temps : ….....
2. Expliquez de manière informelle la modification que vous voulez apporter.
   Temps : ….....
3. Modifiez le diagramme pour corriger l'erreur de vérification (la syntaxe est libre).
   Temps : ….....

Annexe C

Questionnaire System Usability Scale

Pour mesurer la satisfaction des utilisateurs concernant l'outillage proposé pendant la validation expériementale, nous avons traduit le questionnaire System Usability Scale de Brooke [39]

System Usability Scale *© Digital Equipment Corporation, 1986.*

IMPORTANT:

- Répondre spontanément, ne prenez pas le temps de réfléchir, notez votre première impression.
- Répondre à toutes les questions.
- Si vous n'arrivez pas à vous décider alors cocher la réponse du milieu

	Pas du tout d'accord				Tout à fait d'accord
1. Je pense que je vais utiliser cet outil fréquemment.	1	2	3	4	5
2. Je trouve cet outil inutilement complexe.	1	2	3	4	5
3. Je pense que cet outil est facile à utiliser.	1	2	3	4	5
4.Je pense que j'aurai besoin de l'aide d'un expert pour être capable d'utiliser cet outil.	1	2	3	4	5
5.J'ai trouvé que les différentes fonctions de cet outil ont été bien intégrées.	1	2	3	4	5
6. Je pense qu'il y a trop d'incohérences dans cet outil.	1	2	3	4	5
7. J'imagine que la plupart des gens seraient capable d'apprendre à utiliser cet outil très rapidement.	1	2	3	4	5
8 J'ai trouvé cet outil très lourd à utiliser.	1	2	3	4	5
9. Je me sentais très en confiance en utilisant cet outil.	1	2	3	4	5
10. J'ai besoin d'apprendre beaucoup de choses avant de pouvoir utiliser cet outil.	1	2	3	4	5

Annexe D

Implémentation de l'opérateur d'analyse pour la visualisation Objectstory

L'opérateur d'anaylse de la visualisation Objectstory extraie un ensemble d'évènements pertinents à partir de la trace.

```
public static void fromIFTrace2IFTraceEB(IFTrace trace, IFTraceEB ifTraceEb) {
        int nbConfigs = trace.getTConfigs().size();
        //boucle de parcours de l'ensemble des paires de configurations successives
        for (int i = 0; i < nbConfigs − 1; i++) {

                //extraction d'une paire de configurations successives dans la trace
                IFConfig oldConf = trace.getTConfigs().get(i).getTargetConfig();
                IFConfig newConf = trace.getTConfigs().get(i + 1).getTargetConfig();

                //extraction des événements de création d'instances d'objets
                EList<TraceEvent> instanceChangeEvents = computeInstanceChangeEvt(
                                newConf, oldConf);
                //extraction des événements de changements d'états
                EList<TraceEvent> stateChangeEvents = computeInstanceStateChangeEvt(
                                newConf, oldConf);
                //extraction des événements de réception de messages
                EList<TraceEvent> enqueueEvents = computeEnqueueEvt(newConf,
                                oldConf);

                //population du modèle IFTraceEB avec les 3 collections d'événements crées
                ci−dessus EventSet eventSet = IFTraceEBFactory.eINSTANCE.createEventSet();
                eventSet.setStep(i + 1);
```

```
                if (instanceChangeEvents != null)
                        eventSet.getEvents().addAll(instanceChangeEvents);
                if (stateChangeEvents != null)
                        eventSet.getEvents().addAll(stateChangeEvents);
                if (enqueueEvents != null)
                        eventSet.getEvents().addAll(enqueueEvents);

                ifTraceEb.getEventSets().add(eventSet);
        }

}
```

Listing D.1 – Algorithme d'extraction des évènements pertinents implémenté en Java

Liste des abréviations

BPEL Business Process Execution Language

CIM Computation Independent Model

CMMI Common Maturity Model Integration

CTL Computation Tree Logic

DRM Data Reference Model

DSML Domain Specific Modeling Language

DSRM Data State Reference Model

EMF Eclipse Modeling Framework

IDM Ingénierie Dirigée par les Modèles

LTL Linear Temporal Logic

MARTE Automated Transfer Vehicle

MARTE Modeling and Analysis of Real-Time and Embedded Systems

MDA Model Driven Architecture

MOF MetaObject Facility

MSC Message Sequence Chart

OCL Object Constraint Language

OMG Object Management Group

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query/View/Transformation

SDL Specification and Description Language

SGS Solar Generation Subsystem

SUS System Usability Scale

SysML Systems Modeling Language

TCTL Timed Computation Tree Logic

TOCL Timed Object Constraint Language

UML Unified Modeling Language

XSL eXtensible Stylesheet Language

Bibliographie

[1] Atlas Transformation Language : http://www.eclipse.org/atl.

[2] Construction and Analysis of Distributed Processes (CADP) - Software Tools for Designing Reliable Protocols and Systems : http://www.inrialpes.fr/vasy/cadp/.

[3] Eclipse Modeling Framework : http://www.eclipse.org/modeling/emf.

[4] Galaxy ANR Project : http://galaxy.lip6.fr/.

[5] Graphviz : http://www.graphviz.org/.

[6] NuSMV Home Page : http://nusmv.fbk.eu/.

[7] Official Reference MARTE Tutorial : http://www.omg.org/omgmarte/Tutorial.htm.

[8] Projet RT-SIMEX : www.rtsimex.org.

[9] Rigorous Design of Component-Based Systems - The BIP Component Framework : http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.

[10] SPIN Model Checker : http://spinroot.com/.

[11] The Extensible Stylesheet Language Family (XSL) : http://www.w3.org/Style/XSL/.

[12] The Scala Programming Language : http://www.scala-lang.org/.

[13] TIme petri Net Analyzer (TINA) : http://projects.laas.fr/tina/.

[14] Topcased Project : http://www.topcased.org/.

[15] UML Profile for MARTE : `http://www.omgmarte.org/`.

[16] UPPAAL web site : `http://www.uppaal.org/`.

[17] *Proceedings ESA Board for Software Standardisation and Control (BSSC) Workshop*, volume 114 of *EPTCS*, 2005.

[18] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. Seeing Errors : Model Driven Simulation Trace Visualization. In *MODELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 480–496. Springer, 2012.

[19] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. Significantly Increasing the Usability of Model Analysis Tools Through Visual Feedback. In *SDL 2013 - Model Driven Dependability Engineering*, LNCS. Springer-Verlag, June 2013.

[20] Gediminas Adomavicius and Alexander Tuzhilin. Toward the Next Generation of Recommender Systems : A Survey of the State-of-the-Art and Possible Extensions. *IEEE Trans. Knowl. Data Eng.*, 17(6) :734–749, 2005.

[21] Kawtar Benghazi Akhlaki, María Visitación Hurtado, María Luisa Rodríguez, and Manuel Noguera. Applying Formal Verification Techniques to Ambient Assisted Living Systems. In *OTM Workshops*, pages 381–390, 2009.

[22] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *LICS*, pages 414–425, 1990.

[23] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.

[24] Natalia Andrienko and Gennady Andrienko. *Exploratory analysis of spatial and temporal data : a systematic approach.* Springer, 2006.

[25] ANSI/EIA. Processes for Engineering a System, ANSI/EIA-632-1999 (R2003). Technical report, American National Standards Institute / Electronics Industry Association, 1999.

[26] Bruno Antunes, Joel Cordeiro, and Paulo Gomes. An approach to context-based recommendation in software development. In *RecSys*, pages 171–178, 2012.

[27] Gilles Audemard and Belaid Benhamou. *Symétries*, chapter 6. Hermes, 2008.

[28] K. Suzanne Barber, Thomas Graser, and Jim Holt. Providing early feedback in the development cycle through automated application of model checking to software architectures. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 341–345, Nov.

[29] Howard Barringer and Klaus Havelund. Checking flight rules with TraceContract : Application of a Scala DSL for trace analysis. In *Proceedings of Scala Days 2011*, 2011.

[30] Howard Barringer and Klaus Havelund. TraceContract : A Scala DSL for Trace Analysis. In *FM*, pages 57–72, 2011.

[31] Gerd Behrmann, Alexandre David, and KimG. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[32] Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer, 2008.

[33] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations - Change (in) the rule to rule the change. *Software and System Modeling*, 11(3) :431–461, 2012.

[34] Jacques Bertin. *Sémiologie graphique. : Les diagrammes, les réseaux, les cartes*. Les Réimpressions des Éditions de l'École des hautes études en sciences sociales. École des Hautes Études en Sciences Sociales, 1999.

[35] Gilles Boisclair and Jocelyne Pagé. *Guide des sciences expérimentales : Observations, mesures, rédaction du rapport de laboratoire*. Éd. du Renouveau pédagogique, 2004.

[36] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0 : A Validation Environment for Component-Based Real-Time Systems. In *CAV*, pages 343–348, 2002.

[37] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In *SFM*, pages 237–267, 2004.

[38] Carol Britton and Sara Jones. The untrained eye : how languages for software specification support understanding in untrained users. *Human-Computuer Interaction*, 14(1) :191–244, March 1999.

[39] John Brooke. SUS : A quick and dirty usability scale, 1996.

[40] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8) :677–691, 1986.

[41] R. Ian Bull. *Model Driven Visualization : Towards A Model Driven Engineering Approach For Information Visualization*. PhD thesis, University of Victoria, BC, Canada, 2008.

[42] Alan Burns and Andrew J. Wellings. *Real Time Systems and Their Programming Languages : Ada 95, Real-time Java and Real-time POSIX*. International computer science series. Addison-Wesley, 2001.

[43] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4 :171–188, 2005.

[44] Jean Bézivin, Mireille Blay, Mokrane Bouzhegoub, Jacky Estublier, Jean-Marie Favre, Sébastien Gérard, and Jean-Marc Jézéquel. Rapport de synthèse de l'AS CNRS sur le MDA. CNRS, November 2004.

[45] Stuard K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human Computer Interaction*. Lawrence Erlbaum Associates, Incorporated, 1983.

[46] Stuart K. Card, Jock D. MacKinlay, and Ben Schneiderman. *Readings in information visualization : using vision to think*. Interactive Technologies Series. Morgan Kaufmann Incorporated, 1999.

[47] Paul Chandler and John Sweller. The Split-Attention Effect as a Factor in the Design of Instruction. *British Journal of Educational Psychology*, 62(2) :233–246, 1992.

[48] Paul Chandler, John Sweller, Paul Tierney, and Martin Cooper. Cognitive load as a factor in the structuring of technical material. *Journal of Experimental Psychology : General*, 119 :176–192, 1990.

[49] William G. Chase and H. Alexander Simon. *Perception in Chess*. CIP 182 ; Report 71-16. Department of Psychology, Carnegie-Mellon University, 1971.

[50] Sophie Chauvin. *Information et Visualisation : Enjeux, Recherches et Applications*. Editions Cépaduès, 2008.

[51] Ed Huai-hsin Chi. A Taxonomy of Visualization Techniques Using the Data State Reference Model. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, INFOVIS '00, Washington, DC, USA, 2000. IEEE Computer Society.

[52] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2 : an OpenSource Tool for Symbolic Model Checking. In *CAV*, pages 359–364, 2002.

[53] Tony Clark, Paul Sammut, and James Willans. *Applied metamodelling : a foundation for language driven development*, volume 2005. Ceteva, 2008.

[54] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5) :752–794, September 2003.

[55] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[56] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking : algorithmic verification and debugging. *Commun. ACM*, 52(11) :74–84, 2009.

[57] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2) :217–232, 1995.

[58] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5) :1512–1542, 1994.

[59] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[60] Benoit Combemale. *Simulation et vérification de modèle par métamodélisation executable*. éditions Universitaires Européennes, 2010.

[61] Benoit Combemale, Xavier Crégut, Jean-Patrice Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *4th European Congress Embedded Real Time Software (ERTS)*, Toulouse, France, January 2008. SIA & SEE.

[62] Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *APSEC*, pages 282–287, 2012.

[63] Benoît Combemale, Laure Gonnord, and Vlad Rusu. A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 35–51. Springer Berlin Heidelberg, 2011.

[64] Eric Conquet, François-Xavier Dormoy, Iulia Dragomir, Susanne Graf, David Lesens, Piotr Nienaltowski, and Iulian Ober. Formal Model Driven Engineering for Space Onboard Software. In *International Conference on Embedded Real Time Software and Systems (ERTS2)*. Society of Automobile Engineers (SAE), janvier 2012.

[65] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Software Eng.*, 35(5) :684–702, 2009.

[66] Dennis Dams, Orna Grumberg, and Rob Gerth. Generation of Reduced Models for Checking Fragments of CTL. In *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 1993.

[67] Julien DeAntoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. RT-SIMEX : Retro-analysis of execution traces. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 377–378, New York, NY, USA, 2010. ACM.

[68] Edsger W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inf.*, 1 :115–138, 1971.

[69] Alan Dix. *Human-Computer Interaction*. Pearson/Prentice-Hall, 2004.

[70] E. Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 995–1072. Elsevier, 1990.

[71] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio L. Medina Pasaje, Dorina C. Petriu, and C. Murray Woodside. Annotating UML Models with

Non-functional Properties for Quantitative Analysis. In *MoDELS Satellite Events*, pages 79–90, 2005.

[72] Opher Etzion, Peter Niblett, and David Luckham. *Event Processing in Action*. Manning Pubs Co Series. Manning, 2010.

[73] Jean-Marie Favre. *L'ingénierie dirigée par les modèles : au-delà du MDA*. IC2 : Série Informatique et systèmes d'information. Hermes Science Publications, 2006.

[74] Association française de normalisation. *Ergonomie de l'informatique : Aspects logiciels, matériels et environnementaux*. Recueil normes. AFNOR, 2003.

[75] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[76] Martin Gogolla, Fabian Büttner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3) :27–34, 2007.

[77] Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *STTT*, 8(2) :113–127, 2006.

[78] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3) :229–247, 2006.

[79] Jungpil Hahn and Jinwoo Kim. Why are some diagrams easier to work with ? Effects of diagrammatic representation on the cognitive intergration process of systems analysis and design. *ACM Transaction on Computer-Human Interaction*, 6(3) :181–213, September 1999.

[80] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *14th IEEE International Conference on Program Comprehension. ICPC 2006.*, pages 181 –190, 2006.

[81] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol : an industrial case study using UPPAAL. In *RTSS*, pages 2–13, 1997.

[82] Ábel Hegedüs, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of Simulation Traces with Change-Driven Model Transformations. In *SEFM*, pages 145–155, 2010.

[83] C. A. Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[84] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In *Proc. 15 th Spin Workshop*, pages 134–143, 2008.

[85] Gerard J. Holzmann, American Telephone, and Telegraph Company. *Design and Validation of Computer Protocols*. Prentice-Hall software series. Prentice Hall, 1991.

[86] Edwin Hutchins. How a Cockpit Remembers Its Speeds. *Cognitive Science*, 19(3) :265–288, 1995.

[87] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1(4) :311–338, December 1985.

[88] IEEE. Standard for Application and Management of the Systems Engineering Process, IEEE Std 1220-1998. Technical report, IEEE, 1998.

[89] Noah Iliinsky and Julie Steele. *Designing Data Visualizations*. O'Reilly Media, Inc, 2011.

[90] ISO. ISO 9241-11 :1998 Ergonomic requirements for office work with visual display terminals (VDTs) Part 11 : Guidance on usability. Technical report, ISO - International Organization for Standardization, 1998.

[91] ISO. ISO 9241-110 :2006 Ergonomics of human-system interaction – Part 110 : Dialogue principles. Technical report, ISO - International Organization for Standardization, 2006.

[92] ISO. ISO 9241-210 :2010 Ergonomics of human-system interaction Part 210 : Human-centred design for interactive systems. Technical report, ISO - International Organization for Standardization, 2010.

[93] ISO/IEC. ISO/IEC 15288 :2008 Ingénierie des systèmes et du logiciel – Processus du cycle de vie du système. Technical report, ISO - International Organization for Standardization / International Electrotechnical Commission, 2008.

[94] ISO/IEC. ISO/IEC 19505-1 :2012 Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1 : Infrastructure. Technical report, ISO - International Organization for Standardization, 2012.

[95] ISO/IEC/IEEE. *Systems and software engineering Vocabulary*. ANSI/IEEE. Institute of Electrical and Electronics Engineers, 17 mars 2011.

[96] ITU-T. ITU-T Recommandation Z.120 : Message Sequence Chart (msc) (04/2004).

[97] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[98] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[99] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL : a model transformation tool. *Science of Computer Programming, Volume 69*, 72(1-2) :31–39, 2008.

[100] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS : a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 249–254, New York, NY, USA, 2006. ACM.

[101] Natalia Juristo Juzgado and Ana María Moreno. *Basics of software engineering experimentation*. Kluwer, 2001.

[102] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique*. Références sciences. Ellipses, February 2012.

[103] Sslava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. The Expertise Reversal Effect. *Educational Psychologist*, 38(1) :23–31, 2003.

[104] Kenneth R. Koedinger and John R. Anderson. Abstract planning and perceptual chunks : Elements of expertise in geometry. *Cognitive Science*, 14(4) :511 – 550, 1990.

[105] Kurt Koffka. *Principes of Gestalt Psychology*. Routledge & Kegan Paul, 1955.

[106] Stephen Michael Kosslyn. *Elements of graph design*. W.H. Freeman and Company, 1994.

[107] Tim Kovse, Bostjan Vlaovic, Aleksander Vreze, and Zmago Brezocnik. Eclipse Plug-in for Spin and st2msc Tools-Tool Presentation. In *SPIN*, pages 143–147, 2009.

[108] Tim Kovse, Bostjan Vlaovic, Aleksander Vreze, and Zmago Brezocnik. Spin trail to message sequence chart conversion tool. In *The 10th International Conference on Telecomunications*, Zagreb, Croatia, 2009.

[109] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.*, 3(2) :125–143, 1977.

[110] Jill Larkin and Herbert Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 1987.

[111] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Hybrid Systems*, pages 575–586, 1995.

[112] KimG. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer Berlin Heidelberg, 1995.

[113] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *POPL*, pages 97–107, 1985.

[114] Johan Lilius and Ivan Paltor. vUML : A tool for verifying uml models. In *ASE*, pages 255–258, 1999.

[115] David C. Luckham. *The Power of Events : An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[116] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A Task Oriented View of Software Visualization. In *In Proc. 1st international workshop on visualizing software for understanding and analysis (Vissoft)*, pages 32–40. IEEE, 2002.

[117] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[118] Richard E. Mayer. *Multimédia Learning*. Cambridge University Press, 2001.

[119] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[120] David E. Meyer and David E. Kieras. A Computational Theory of Executive Cognitive Processes and Multiple-Task Performance : Part 2. *Pychological Review*, 104(4) :749–791, March 1997.

[121] George Miller. The Magical Number Seven, Plus or Minus Two : Some Limits on Our Capacity for Processing Information, 1956.

[122] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. omg/2003-06-01. Technical report, OMG, 2003.

[123] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[124] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

[125] Marvin L. Minsky. *Semantic Information Processing*. The MIT Press, 1969.

[126] Daniel Laurence Moody. The "physics" of notations : Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6) :756–779, 2009.

[127] Manuel Musial, Fabienne Pradère, and André Tricot. *Comment concevoir un enseignement ?* Guides pratiques : former & se former. De Boeck Supérieur, 2012.

[128] Joan C. Nordbotten and Martha E. Crosby. The effect of graphic style on data model interpretation. *Inf. Syst. J.*, 9(2) :139–156, 1999.

[129] OASIS. Web services business process execution language version 2.0 public review draft, 23th august, 2006.

[130] I. Ober and I. Dragomir. OMEGA2 : A New Version of the Profile and the Tools. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 373 –378, march 2010.

[131] Ileana Ober, Iulian Ober, Iulia Dragomir, and El Arbi Aboussoror. UML/-SysML semantic tunings. *Innovations in Systems and Software Engineering*, 7(4) :257–264, 2011.

[132] Iulian Ober, Susanne Graf, and David Lesens. Modeling and Validation of a Software Architecture for the Ariane-5 Launcher. In *FMOODS*, pages 48–62, 2006.

[133] Iulian Ober, Susanne Graf, Yuri Yushtein, and Ileana Ober. Timing analysis and validation with UML : the case of the embedded MARS bus manager. *Journal on Innovations in Systems and Software Engineering*, 4(3) :301–308, 2008. Springer.

[134] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. Technical report, OMG, January 2012.

[135] OMG. Object Constraint Language, v2.3.1. Technical report, OMG, January 2012.

[136] Allan Paivio. *Mental representations*. Oxford University Press, Incorporated, 1990.

[137] Ivan Paltor and Johan Lilius. Formalising UML State Machines for Model Checking. In *UML*, pages 430–445, 1999.

[138] Rolf-Helge Pfeiffer and Andrzej Wasowski. Cross-Language Support Mechanisms Significantly Aid Software Development. In *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 168–184. Springer Berlin Heidelberg, 2012.

[139] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[140] István Ráth, Gergely Varró, and Dániel Varró. Change-Driven Model Transformations. In *MODELS*, pages 342–356, 2009.

[141] Anders P. Ravn, Jirí Srba, and Saleem Vighio. A Formal Analysis of the Web Services Atomic Transaction Protocol with UPPAAL. In *4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, pages 579–593, 2010.

[142] Wolfgang Reisig. *Petri Nets : An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[143] OMG Staff Strategy Group Richard Soley. Model Driven Architecture White Paper Draft 3.2. Technical report, Object Management Group, 2000.

[144] Theo C. Ruys and Rom Langerak. Validation of Bosch' Mobile Communication Network Architecture with SPIN. In *Proceedings of SPIN97, the Third International Workshop on SPIN*. unpublished, April 1997.

[145] Haruhiko Sato, Shoichi Yokoyama, and Masahito Kurihara. User-friendly GUI in software model checking. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 468 –473, oct. 2009.

[146] Philippe Schnoebelen. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert informatique. Vuibert, 1999.

[147] SEI. CMMI for Development, Version 1.3. Carnegie Mellon University Software Engineering Institute. Technical report, Carnegie Mellon University Software Engineering Institute, 2011.

[148] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19–25, 2003.

[149] Ben Shneiderman. The eyes have it : a task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336 –343, sep 1996.

[150] Joseph Sifakis. A vision for computer science - the system perspective. *Central Europ. J. Computer Science*, 1(1) :108–116, 2011.

[151] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3) :243–247, 2012.

[152] Neville Stanton. *Human Factors Methods : A Practical Guide for Engineering And Design*. Ashgate, 2005.

[153] John Sweller. Evolution of human cognitive architecture. *Psychology of Learning and Motivation*, 43 :215–266, 2003.

[154] John Sweller. *Evolution of human cognitive architecture*, volume 43 of *Psychology of Learning and Motivation*, pages 215–266. Elsevier, 2003.

[155] Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, 1983.

[156] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition : a composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 9 :285–309, 2010.

[157] Colin Ware. *Information visualization : perception for design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[158] William Winn. An account of how readers search for information in diagrams. *Contemporary Educational Psychology*, 18(2) :162 – 185, 1993.

[159] Sharon Wood, Richard Cox, and Peter Cheng. Attention design : Eight issues to consider. *Computers in Human Behavior*, 22 :588–602, 2006.

[160] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging Formal Verification Tools for DSML Users : A Process Modeling Case Study. In *ISoLA (2)*, pages 329–343, 2012.

[161] Jiajie Zhang. The nature of external representations in problem solving. *Cognitive Science*, pages 179–217, 1997.

Résumé

Malgré l'existence d'un nombre important d'approches et outils de vérification à base de modèles, leur utilisation dans l'industrie reste très limitée. Parmi les raisons qui expliquent ce décalage il y a l'exploitation, aujourd'hui difficile, des résultats du processus de vérification. Dans cette thèse, nous étudions l'utilisation des outils de vérification dans les processus actuels de modélisation de systèmes qui utilisent intensivement la validation à base de modèles. Nous établissons ensuite les limites des approches existantes, surtout en termes d'utilisabilité. A partir de cette étude, nous analysons les causes de l'état actuel des pratiques. Nous proposons une approche complète et outillée d'aide au diagnostic d'erreur qui améliore l'exploitation des résultats de vérification, en introduisant des techniques mettant à profit la visualisation d'information et l'ergonomie cognitive. En particulier, nous proposons un ensemble de recommandations pour la conception d'outils de diagnostic, un processus générique adaptable aux processus de validation intégrant une activité de diagnostic, ainsi qu'un framework basé sur les techniques de l'Ingénierie Dirigée par les Modèles (IDM) permettant une implémentation et une personnalisation rapide de visualisations.

Notre approche a été appliquée à une chaîne d'outils existante, qui intègre la validation de modèles UML et SysML de systèmes temps réel critiques. Une validation empirique des résultats a démontré une amélioration significative de l'utilisabilité de l'outil de diagnostic, après la prise en compte de nos préconisations.

Mots-clés : Vérification, SysML, UML, Ingénierie Dirigée par les Modèles, visualisation d'information

Abstract

A plethora of theoretical results are available which make possible the use of dynamic analysis and model-checking for software and system models expressed in high-level modeling languages like UML, SDL or AADL. Their usage is hindered by the complexity of information processing demanded from the modeller in order to apply them and to effectively exploit their results. Our thesis is that by improving the visual presentation of the analysis results, their exploitation can be highly improved. To support this thesis, we define a diagnostic trace analysis approach based on information visualisation and human factors techniques. This approach offers the basis for new types of scenario visualizations, improving diagnostic trace understanding.

Our contribution was implemented in an existing UML/SysML analyzer and was validated in a controlled experiment that shows a significant increase in the usability of our tool, both in terms of task performance speed and in terms of user satisfaction.

The pertinence of our approach is assessed through an evaluation, based on well-established evaluation mechanisms. In order to perform such an evaluation, we needed to adapt the notion of usability to the context of formal methods usability, and to adapt the evaluation process to our setting. The goal of this experiment was to see whether extending analysis tools with a well-designed event-based visualization would significantly improve analysis results exploitation and the results are meeting our expectations.

Keywords: Verification, SysML, UML, Model Driven Engineering, Information visualisation