Un environnement cadre pour la gestion de modèles hétérogènes en ingénierie système

David Simon-Zayas

▶ **To cite this version:**

David Simon-Zayas. Un environnement cadre pour la gestion de modèles hétérogènes en ingénierie système. Autre. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers, 2012. Français. NNT : . tel-00740161

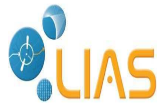
HAL Id: tel-00740161

https://theses.hal.science/tel-00740161

Submitted on 9 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

	Ecole Nationale Supérieure de Mécanique et d'Aérotechnique Laboratoire d'Informatique et d'Automatique pour les Systèmes	

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information, Mathématiques

Secteur de Recherche : INFORMATIQUE ET APPLICATION

Présentée par :

David Simon Zayas

****

A framework for the management of heterogeneous models in Systems Engineering

****

Directeur de Thèse : **Yamine AIT-AMEUR**

Co-directeur de Thèse : **Anne MONCEAUX**

****

Soutenue le 8 Juin 2012

devant la Commission d'Examen

****

JURY

Rapporteurs :

Frédéric BONIOL	**Maître de recherche, ONERA, Toulouse**
Parisa GHODOUS	**Professeur, Université Claude Bernard Lyon I, Villeurbanne**

Examinateurs :

Yamine AIT-AMEUR	**Professeur, INPT-ENSEEIHT/IRIT, Toulouse**
Ladjel BELLATRECHE	**Professeur, ENSMA, Futuroscope**
Anne MONCEAUX	**Docteur, Chef de Projet, EADS IW, Toulouse**
Mourad OUSSALAH	**Professeur, CNRS, Nantes**
Romaric REDON	**Docteur, Responsable d'Equipe, EADS IW, Toulouse**

A Aurélie, Matt i la panxeta. Us estimo molt.

Remerciements

En préambule à ce mémoire, je souhaitais adresser mes remerciements aux personnes qui m'ont apporté leur aide et qui ont ainsi contribué à l'élaboration de ces travaux.

Je tiens à remercier sincèrement mes directeurs de thèse, Yamine Ait-Ameur et Anne Monceaux, qui se sont toujours montrés à l'écoute et disponibles au cours de ces trois dernières années. Sans leurs conseils avisés, le temps qu'ils ont bien voulu me consacrer et leur soutien, cette thèse n'aurait jamais abouti.

Mes remerciements s'adressent également aux autres membres de mon comité de pilotage, Christian Benac, Romaric Redon et Christian Trinquier, pour leurs appréciations et leurs sages recommandations.

Je remercie Frédéric Boniol et Parisa Ghodous pour m'avoir fait l'honneur d'être rapporteurs de cette thèse ; Ladjel Bellatreche et Mourad Oussalah pour avoir accepté d'être membres du jury en tant qu'examinateurs.

J'exprime ma gratitude à tous mes collègues d'EADS Innovation Works qui m'ont permis d'avoir un cadre industriel de travail idéal pour ma thèse CIFRE.

J'adresse également un très grand merci aux membres du laboratoire LISI, aujourd'hui LIAS, qui m'ont accueilli et m'ont fourni tout le support scientifique indispensable pour cette thèse.

Je n'oublie pas ma famille et mes amis, mes piliers émotionnels, qui m'ont toujours soutenu et encouragé au cours de la réalisation de ces travaux.

Introduction

L'Ingénierie Système est une discipline qui a pour objectifs de concevoir et gérer des systèmes en se basant sur une approche multidisciplinaire. Elle s'applique sur le cycle de vie complet d'un système. Les grands systèmes mécatroniques tels que les avions ont un cycle de développement de plusieurs années. Ils sont développés au sein de vastes organisations généralement très complexes, qui mettent en scène de nombreuses équipes de différentes entreprises qui elles-mêmes travaillent avec leurs réseaux de fournisseurs. Lors de la modélisation, la coopération entre ces différents acteurs donne lieu à une multiplicité de problèmes locaux et de points de vue divergents.

Les facteurs les plus importants d'hétérogénéité sont liés aux connaissances et au savoir-faire des ingénieurs qui ne sont pas formalisés dans les modèles mêmes. Ainsi les ingénieurs construisent les modèles d'un système ou d'une partie de celui-ci avec un objectif précis. Bien qu'ils les développent conformément aux standards et aux bonnes pratiques, certaines connaissances nécessaires à leur interprétation restent tacites. Elles sont appelées connaissances implicites car elles sont généralement connues des seuls ingénieurs. Elles nécessitent donc d'être explicitées afin de pouvoir comprendre et valider ces modèles dans un milieu collaboratif.

Dans ce contexte, le but de notre travail est de proposer une approche permettant de gérer l'hétérogénéité des inter-modèles et de la déployer dans le cadre de l'Ingénierie Système appliquée à l'Aéronautique.

Notre premier objectif est de réduire l'hétérogénéité due aux différentes natures de modèles et de langages de modélisation. Nous proposons pour cela l'utilisation d'un langage unifié et partagé permettant de travailler dans un même environnement. Grâce à l'application des techniques de méta-modélisation, nous exportons des modèles source vers un

environnement commun où nous pouvons utiliser les modèles sans modifier leurs définitions originales.

Cela nous mène à notre second objectif qui tend à manipuler des modèles exportés en utilisant la connaissance additionnelle normalement non exprimée. Nous nous proposons d'enrichir les modèles avec cette information essentielle à leur compréhension et intégration. Cette étape est nécessaire à la validation des propriétés inter-modèles qui nécessitent également que les connaissances implicites soient explicitées.

Ainsi, notre approche soutient la gestion des modèles hétérogènes en

- décrivant, modélisant et vérifiant des contraintes et des relations entre des modèles hétérogènes existants utilisés dans un processus d'Ingénierie Système;

- explicitant, formalisant et exploitant des connaissances additionnelles normalement non exprimées par les ingénieurs pour décrire ces contraintes et relations.

D'un point de vue scientifique, les principales contributions de notre travail au problème de l'hétérogénéité de modèles interviennent de deux façons. D'une part, la formalisation des connaissances implicites des ingénieurs pour annoter des modèles fonctionnels et d'architecture. D'autre part, l'utilisation d'un environnement commun et partagé pour exporter les modèles source en les homogénéisant syntaxiquement sans modifier les éléments originaux. Ces deux principes s'associent en une méthode qui est le noyau de notre proposition et qui permet aux ingénieurs d'interconnecter des modèles de même niveau du cycle de vie et de valider des contraintes inter-modèles. D'un point de vue industriel, la méthode proposée est outillée grâce à un prototype orienté processus qui peut être utilisé comme base pour une future industrialisation de l'approche.

Chapitre I

Les systèmes complexes font intervenir de multiples domaines et doivent intégrer des sources hétérogènes tout au long du cycle de vie. Parmi ces sources, les modèles sont nécessaires pour mieux organiser le développement d'un système et la gestion de sa complexité intrinsèque. Les approches d'ingénierie actuelles telles que l'Ingénierie Système Basée sur les Modèles (MBSE en anglais) se focalisent sur cet aspect. Néanmoins, ces modèles contribuent au développement d'un système à différents moments de son cycle de

vie. Par conséquence, ils ont différents objectifs et prennent en compte une grande variété de domaines.

De nos jours, la complexité des systèmes fait naître le besoin de disciplines multi-domaines capables gérer de telles complexités. Les meilleures pratiques d'ingénierie recommandent l'utilisation de modèles pour gérer les processus de développement. Notre travail se focalise sur la phase de conception de l'architecture fonctionnelle et physique du produit final. Le cycle de vie de développement ainsi que le travail collaboratif, entre autres facteurs, entraînent un accroissement de l'hétérogénéité des modèles. Cette dernière devient un problème au moment de partager et d'intégrer ces modèles afin d'en garantir la cohérence. Bien qu'il existe différentes approches fructueuses visant à intégrer les modèles, elles ne prennent pas en compte les connaissances implicites, i.e. les connaissances des ingénieurs que ne sont pas formalisées dans les modèles mêmes mais qui sont indispensables pour les comprendre et les valider. Ainsi, notre travail se base sur les principes de méta-modélisation tout en les complétant avec l'explicitation des connaissances implicites.

Chapitre II

Les modèles sont le résultat d'un travail d'équipe entre ingénieurs et représentent un système ou partie de celui-ci sous un angle particulier. Néanmoins, afin de pouvoir analyser correctement un modèle, des connaissances additionnelles provenant des ingénieurs responsables de celui-ci sont nécessaires. Il existe de nombreux mécanismes pour formaliser ces connaissances mais nous pensons que dans le cadre de l'Ingénierie Système, les ontologies formelles sont adéquates, principalement grâce à leur nature précise et consensuelle. Nous défendons la formalisation des connaissances implicites comme moyen d'intégrer et de valider des modèles hétérogènes. En effet, en explicitant les connaissances implicites nous pouvons annoter ces modèles afin de faciliter leur intégration et de donner un support à la validation des propriétés inter-modèles.

Dans ce chapitre nous présentons différents aspects de la modélisation des connaissances. Nous discutons des efforts actuels de rapprochement des modèles et des ontologies. En effet, nous pensons qu'il existe un besoin d'approche non intrusive qui défend la formalisation de connaissances implicites pour intégrer des modèles hétérogènes. Son but est de valider les contraintes inter-modèles.

Notre objectif est de décrire, modéliser et vérifier des relations et des contraintes entre des modèles hétérogènes en explicitant, formalisant et exploitant les connaissances additionnelles des ingénieurs. Notre travail se concentre sur les relations inter-modèles au sein du processus de développement de l'Ingénierie Système appliquée à l'Aéronautique.

Chapitre III

Dans ce chapitre nous décrivons notre contexte industriel -l'aéronautique-, et nous évoquons les bénéfices attendus de notre approche. En effet, la conformité de la solution de gestion des modèles hétérogènes que nous proposons est directement liée à son contexte industriel ayant ses propres méthodes et pratiques.

La complexité de conception d'un modèle est directement liée au contexte industriel auquel il est rattaché. Dans l'industrie aéronautique, la modélisation est très complexe, tant du point de vue de l'organisation que des méthodes. Ceci est non seulement dû à la complexité du système lui-même mais aussi aux grandes organisations et aux multiples fournisseurs collaborant à la conception. Historiquement, ces difficultés ont été surmontées grâce à l'application de règles strictes de documentation et plus récemment grâce aux principes MBSE. Toutefois, le déploiement progressif de MBSE dans l'industrie aéronautique a besoin de nouvelles méthodes et d'outils de gestion des modèles. Dans ce contexte, nous présentons dans le chapitre IV une approche dont les bénéfices attendus sont une meilleure cohérence des modèles, la formalisation des relations inter-modèles, une meilleure réutilisation des modèles et une réduction du temps de conception.

Chapitre IV

L'utilisation de modèles dans le processus d'ingénierie est encouragée par le besoin de gérer les systèmes actuels complexes. De nos jours, le partage du travail et la maturité des techniques d'ingénierie collaborative exigent de mettre en relation des modèles hétérogènes afin d'atteindre les objectives globaux. Nous proposons une méthode permettant d'interopérer des modèles hétérogènes existants. Notre approche s'appuie sur les connaissances pour annoter ces modèles.

Dans ce chapitre nous décrivons cette approche basée sur une méthode visant à intégrer et valider des modèles structurels et fonctionnels de même niveau grâce à l'explicitation des connaissances implicites des ingénieurs. L'approche est axée sur un processus qui manipule ces modèles avec le support de modèles externes. Les différentes activités décrites s'appuient sur un exemple fil rouge. Ainsi, nous décrivons en premier lieu l'exportation des modèles source vers un environnement partagé qui garantit l'homogénéisation syntaxique et l'intégrité des modèles source. En deuxième lieu, nous utilisons des modèles externes de connaissances pour annoter les modèles exportés en obtenant l'homogénéisation sémantique. Cette homogénéisation permet de mettre en relation les modèles annotés et d'exprimer des contraintes inter-modèles grâce à un modèle externe d'expressions. Ces modèles d'expressions utilisent à la fois les entités annotées et les concepts de la base de

connaissances. Finalement, les contraintes sont validées grâce à l'implémentation du modèle d'expressions.

Chapitre V

Dans ce chapitre nous développons un cas d'étude afin de valider notre approche et d'illustrer ses différentes étapes. Le scénario inclut deux langages de modélisation différents, SysML et CORE, et une contrainte inter-modèle complexe. Ce cas d'étude est utilisé pour valider formellement notre proposition en utilisant EXPRESS comme langage commun et partagé.

Nous avons choisi EXPRESS pour cette validation dans le but : 1) d'exporter les modèles source en tant qu'instance des méta-modèles construits en EXPRESS ; 2) de développer des modèles de connaissances ; 3) de supporter la logique de premier ordre (FOL en anglais) comme langage d'expression de propriétés. Ainsi, la modélisation formelle en EXPRESS nous a permis de valider l'approche car nous sommes capables : d'importer des modèles SysML et CORE ; de concevoir et d'instancier des modèles de connaissances ; d'utiliser la base de connaissances pour annoter les modèles importés ; d'établir des relations inter-modèles ; d'écrire une contrainte dynamiquement ; et de valider cette contrainte.

Ces modèles formels d'EXPRESS ont été validés opérationnellement d'un point de vue scientifique en utilisant l'outil ECCO afin de les instancier et de valider la contrainte en s'appuyant sur son vérificateur d'instances.

Chapitre VI

Dans ce chapitre, nous effectuons une évaluation industrielle de notre approche, en utilisant des modèles simplifiés basés sur l'analyse de quatre cas d'étude réels. L'objectif principal est de valider l'usabilité de l'approche.

La variabilité des cas d'études en termes de nombre de modèles, de langages de modélisation et de règles de modélisation démontre que notre approche est générique. Du point de vue du cycle de vie des modèles, l'approche peut être appliquée à différents niveaux : avant le développement des modèles source pour trouver d'anciens modèles grâce aux annotations ; lors du développement des modèles source pour valider des contraintes inter-modèles ; et après le développement des modèles source pour vérifier à nouveau, suite à des modifications, des contraintes précédemment validées.

Néanmoins, les modèles et les instances EXPRESS représentant les différents cas d'étude ont été créés manuellement. De ce fait des améliorations sont nécessaires pour atteindre une

industrialisation réussie : une automatisation maximale des activités de l'approche ; des annotations a priori, i.e. lors du processus de modélisation ; un niveau intermédiaire d'abstraction de la sémantique de modélisation ; la réutilisation des contraintes et un support visuel pour les construire.

Chapitre VII

Un prototype a été développé lors du déploiement des cas d'étude industriels. L'objectif de ce prototype est de fournir aux ingénieurs un outil permettant la gestion des concepts de l'approche. Le prototype décrit dans ce chapitre nous permet d'illustrer visuellement les cas industriels et de valider ainsi l'approche avec des ingénieurs.

L'outil couvre les besoins identifiés et les conclusions d'usabilité de l'évaluation industrielle. La prochaine étape consiste à fournir une version du prototype plus adaptée à une future industrialisation et incluant des améliorations graphiques.

Chapitre VIII

L'Ingénierie Système Basée sur les Modèles est une discipline qui suscite beaucoup l'attention de l'industrie aéronautique. Par conséquent, notre approche doit prendre en considération l'état actuel de déploiement du MBSE afin de développer une solution industrielle adaptée. Cette solution doit être technologiquement robuste et intégrée dans les processus de modélisation actuels afin d'obtenir les bénéfices attendus de l'industrialisation.

Le futur déploiement de notre approche est analysé dans ce chapitre de deux points de vue. En premier lieu, des améliorations, notamment concernant la gestion en réseaux des modèles, sont nécessaires dans les processus MBSE actuels si nous aspirons à une intégration optimale de notre approche. D'autre part, ces améliorations devront être accompagnées de modifications technologiques sur la base de notre implémentation actuelle de l'approche. Ces améliorations sont le sujet de la deuxième partie de ce chapitre.

Conclusions et perspectives

Dans le contexte des méthodologies de conception de l'Ingénierie Système, les ingénieurs travaillent avec des modèles issus de différentes équipes, méthodologies et savoir-faire. Ce travail collaboratif donne lieu à différents types de modèles, de langages de modélisation et de techniques de modélisation. Ainsi, les modèles hétérogènes sont une conséquence logique de cette variabilité. Cette hétérogénéité devient un véritable problème lorsque les modèles doivent être partagés entre différentes équipes afin d'effectuer des analyses et des validations globales. Dans ce contexte, expliciter les connaissances implicites est essentiel.

Notre approche propose l'intégration de modèles hétérogènes et la modélisation et validation de contraintes inter-modèles en explicitant, en formalisant et en exploitant ces connaissances additionnelles qui sont habituellement implicites pour les concepteurs.

Nos contributions ont été développées à partir de différentes lignes directrices :

- **Méthodologie.** Notre travail réunit deux concepts : la modélisation hétérogène et l'explicitation des connaissances implicites. Nous avons défini une méthode permettant d'utiliser la connaissance et de définir des expressions à l'aide d'un langage flexible dans le but de vérifier les contraintes de relation inter-modèles.

- **Explicitation de connaissances implicites.** L'originalité de notre approche se situe dans la formalisation de l'explicitation des connaissances implicites et l'annotation de modèles d'ingénierie hétérogènes. Ces connaissances sont gérées indépendamment des modèles annotés grâce à l'utilisation de modèles externes et d'identifiants uniques. Ainsi, les annotations contiennent le lien entre des modèles exportés et des concepts de connaissance agissant comme une couche intermédiaire. Ce niveau intermédiaire, et le fait que les modèles source soient exportés, permettent l'évolution des modèles source indépendamment de l'application de l'approche.

- **Contraintes inter-modèles.** Dans nos cas d'étude, nous avons validé des contraintes pouvant s'exprimer comme expressions FOL. Afin de les implémenter, nous avons développé un modèle formel d'expressions en utilisant le langage de modélisation EXPRESS. Ce modèle est une extension du modèle d'expressions de PLIB qui n'inclut pas les expressions FOL.

- **Outillage.** Afin de guider les utilisateurs dans l'appropriation de notre méthodologie, nous avons développé un prototype. Cet outil est orienté processus et supporte chacune des activités de modélisation de notre approche.

- **Déploiement et applicabilité.** La validation formelle de la proposition et l'implémentation des différents types de cas d'étude dans le prototype démontre l'applicabilité de notre approche. Néanmoins, l'évolution du prototype est nécessaire avant le déploiement industriel de la solution.

Le travail décrit dans ce manuscrit ouvre plusieurs perspectives :

- **Evolution de modèles.** Lors du processus de modélisation, les modèles ont différents degrés de maturité. Par conséquence ils évoluent et de nouvelles versions apparaissent.

Les modèles peuvent également évoluer parce qu'ils sont réutilisés dans un nouveau programme. Notre approche doit prendre en considération cette évolution des modèles et gérer la réutilisation des annotations et des contraintes inter-modèles.

- **Abstraction du langage de modélisation.** Nous pensons que la définition des contraintes devrait s'appuyer sur une ontologie décrivant des concepts généraux d'Ingénierie Système. Cette ontologie permettrait aux ingénieurs d'écrire leurs contraintes d'une façon plus naturelle et rendrait plus facile l'éventuelle génération de contraintes à partir d'exigences formelles. Finalement, le choix d'autres logiques différentes de FOL devrait être considéré au moment de l'analyse des caractéristiques des contraintes à valider.

- **Relations inter-modèles.** Nos cas d'étude se sont focalisés sur des relations de même niveau et, en tant que perspective, les cas de relations verticales devraient être pris en compte également. Nous défendons que notre approche est applicable à des relations verticales mais que l'activité d'intégration et le méta-modèle de relations devraient être renforcés. Par conséquent, des cas incluant ce genre de relations inter-modèles devraient être étudiés afin d'amplifier notre travail.

- **Passage à l'échelle.** Le prototype nous a permis la validation formelle de notre proposition. Désormais, le passage à l'échelle de la solution peut être abordé de deux manières. D'un côté et afin de permettre son industrialisation, l'implémentation de notre approche doit être capable de gérer un grand nombre de modèles et d'entités. Par ailleurs, il serait nécessaire d'analyser des domaines d'application autres que l'Ingénierie Système (automotivité, espace et autres systèmes complexes).

Intégration MBSE et services. En ce qui concerne MBSE, l'intégration de notre méthode avec les pratiques actuelles devrait être accompagnée d'un processus de standardisation. Nous pensons que l'explicitation des connaissances implicites et la relation entre modèles hétérogènes doivent faire partie des standards MBSE afin de fournir une meilleure gestion du cycle de vie des systèmes complexes. Une perspective ambitieuse consiste à développer une plate-forme supportant les processus MBSE avec des services adéquats et une définition précise des rôles (administration des tâches, gestion des connaissances, gestion des contraintes…). Dans un tel contexte, notre approche ferait partie des services offerts par la plate-forme aux rôles indiqués.

Summary

Introduction *1*

Context... **1**

Current practices **2**

Our proposal... **3**

Structure of the document... **4**

Chapter I Heterogeneity of models in Systems Engineering domain...... *5*

I.1. Introduction **7**

I.2. System modeling **7**

I.2.1. Notion of system 7

I.2.2. Models Typology 8

I.2.3. Functional modeling 9

I.2.4. Modeling languages 10

I.3. Systems Engineering... **12**

I.3.1. Systems Engineering Life Cycle description and instantiations 14

I.3.2. Collaborative design... 15

I.4. Inter-model relations **16**

I.4.1. Typology 16

I.4.2. Relations in the process development 16

I.5. Heterogeneity **17**

I.6. Current approaches to handle heterogeneity... **18**

I.6.1. Model-Based Systems Engineering... 18

I.6.2. Integration approaches 20

I.7. Conclusion **21**

Chapter II Knowledge models to integrate and validate heterogeneous models 23

II.1. Implicit knowledge... **25**

II.2. Formalization of knowledge **25**

II.2.1. Need of ontologies... 26

II.3. Ontologies and annotation of models... **27**

II.4. Validation of inter-model properties... **28**

II.4.1. Requirements... 29

II.4.2. Property languages 29

II.5. EXPRESS modeling language... **30**

II.5.1. Meta-modeling 30
II.5.2. Expressions with EXPRESS 31
II.5.3. The choice of EXPRESS 32

II.6. Conclusion 33

Chapter III Current practices in Aircraft Systems Engineering 35

III.1. Introduction 37

III.2. Aircraft Systems Modeling 37

III.3. Current MBSE applications 39

III.4. From documents to models 40

III.5. MBSE and development process 41

III.6. Management of heterogeneous modeling in Aircraft Systems Engineering 41

III.7. Expected benefits of the proposed approach 42

III.8. Conclusion 44

Chapter IV Knowledge-based inter-model constraint verification 45

IV.1. Introduction 47

IV.2. The proposed General integrated models representation 49

IV.3. Manipulated models 51

IV.3.1. Source models 51
IV.3.2. Exported models 52
IV.3.3. Annotated models 52
IV.3.4. Integrated model 52
IV.3.5. Constraint Relational model 53

IV.4. The resources 53

IV.4.1. Source Meta-models 53
IV.4.2. Knowledge models 53
IV.4.3. Constraint Relational meta-models 53

IV.5. The modeling process activities 54

IV.5.1. Export 54
IV.5.2. Annotation 58
IV.5.3. Model integration 61
IV.5.4. General constraint definition 63

IV.6. Conclusion 67

Chapter V Approach validation 69

V.1. Introduction 71

V.2. Exportation of SysML and CORE models 71

V.3. Annotation using implicit knowledge 73

V.4. Model integration using equivalences 74

V.5. General constraint definition with First Order Logic expressions 75

V.5.1. Contribution to PLIB expressions language 75

V.5.2. Inter-model constraint verification 78

V.6. Implementation with ECCO toolkit 79

V.6.1. The common framework 81

V.6.2. Step by step implementation 81

V.7. Conclusion 86

Chapter VI Industrial evaluation 87

VI.1. Introduction 89

VI.2. Water and Waste System model 90

VI.2.1. Description 90

VI.2.2. The modeling process activities 90

VI.2.3. Conclusions 95

VI.3. Hydraulic and Engine systems models 95

VI.3.1. Description 95

VI.3.2. The modeling process activities 96

VI.3.3. Conclusions 102

VI.4. Ram Air Turbine models 103

VI.4.1. Description 103

VI.4.2. The modeling process activities 103

VI.4.3. Conclusions 110

VI.5. Conclusion 110

Chapter VII Prototyping tool 113

VII.1. A prototype to support the method 115

VII.2. Actors and use cases 116

VII.2.1. Actors 116

VII.2.2. Configuration use cases 116

VII.2.3. Operational use cases 117

VII.3. Selected technology and architecture 119

VII.4. Current HCI (Human Computer Interface) 121

VII.5. Conclusion 125

Chapter VIII Deployment in industry 127

VIII.1. Industrialization requirements 129

VIII.1.1. Model and configuration management 129

VIII.1.2. Knowledge management 129
VIII.1.3. Model integration 130
VIII.1.4. Inter-model constraint management 130
VIII.1.5. Conclusions 131

VIII.2. Needed technology enhancements 132
VIII.2.1. Technology features 132
VIII.2.2. Needed HCI enhancements 134

VIII.3. Conclusion 138

Conclusion and perspectives 139

Contributions 139

Perspectives 142

References 145

Annex A 155

Annex B 189

Annex C 213

1. SIS and CIS message models 213

2. Water and Waste System model 217

3. Hydraulic and Engine systems models 223

4. Ram Air Turbine models 233

Table of figures 243

Introduction

Context

Systems Engineering is a discipline whose objective is to manage and design systems with a multi-disciplinary approach and taking into consideration the entire life cycle of the system. For what it concerns big mechatronic systems such as aircrafts, their development life cycle is several years long. They are developed within large and complicated organizational structures which involve many teams of large enterprises and their supplier networks. It is a fact that the necessary work sharing and collaboration between multiple actors over time always result in a multiplicity of local problems and viewpoints

Based on Systems Engineering principles, Model-Based Systems Engineering tries to structure and organize the use of modeling to support the main system engineering activities: Requirement establishment, Design, Analysis and Validation and Verification (V&V). The multiplicity of local problems and viewpoints result in possible heterogeneity in models, which we want to reduce.

The Aircraft Model Based Systems Engineering (MBSE) is the application domain of our work. During the design phases, the aircraft system is represented by a number of models, each bounded to some given sub-system or viewpoint, and each valid at some given stage of the design progress. Models are developed in different working realities and managed by several teams, all of this being sources of heterogeneity in the final result of the design tasks. Nevertheless, these heterogeneous models exist in a collaboration context implying their sharing and inter-model relations. Even when models have been built following the same guidelines and applying the same methods or using the same modeling language, one can find differences between models of different teams. This heterogeneity is mainly due to the:

1) different objectives of the models;
2) variability in modeling languages and modeling techniques;
3) different applied methodologies;
4) know-how of the involved teams.

Thus, the most important factors of heterogeneity are linked to the knowledge and know-how of the engineers that are not formalized in the models themselves. Engineers build models of a system or part of a system with a particular objective. Nevertheless, even though they develop models according to standards and good practices, some of the knowledge necessary to correctly interpret them remains tacit. We call it implicit knowledge since it is usually in engineers' mind but needs to be made explicit in order to be able to fully understand and validate the models. It is important to distinguish it from implicit semantics (A. Sheth, Ramakrishnan, & Thomas, 2005), i.e. the knowledge which is intrinsic to data itself and which is not treated in our work.

Nevertheless, these heterogeneous models exist in a collaboration context implying their sharing and inter-model relations. Models are verified and validated all along the development life cycle at different stages. At some of these milestones, models that have been developed by different teams need to be integrated in order to perform inter-model validations and analysis, i.e. to check properties that involve several models. Therefore, sharing implicit knowledge becomes essential to integrate such different but related models and to validate them.

In order to achieve this objective we have analyzed current solutions for the use of implicit knowledge in verification of inter-model constraints.

Current practices

The problem of managing heterogeneous models has been addressed in different ways. The standardization of Systems Engineering methodologies and MBSE methods in particular have this objective. They establish the basis for the organization of work and modeling principles but they do not offer solutions to the use of multiple modeling languages. Nevertheless, having a unique modeling language is not a possibility in the layered design of complex systems. The development life cycle of such systems goes from mission-level requirements elicitation to low-level detailed design of equipments which implies different objectives and different modeling needs at each stage. It is a multi-modeling environment and specialized modeling languages are necessary. Therefore, current solutions are oriented to the integration of heterogeneous models by developing gateways between modeling tools based on mapping or on meta-modeling techniques. However, these solutions do not take into consideration the implicit knowledge.

In order to make explicit this implicit knowledge we need to formalize it. The formalization of knowledge is the objective of knowledge models. One of its possible implementations are domain ontologies which are formal representations of consensual

knowledge. Concerning engineering context, the formal and consensual aspects fit the needs for the representation of engineering knowledge. Thus, domain ontologies can be used to annotate engineering models, i.e. to enrich them with additional knowledge. There are some studies regarding the use of ontologies in modeling activities and processes but they do not address the heterogeneity problem of our context.

To sum up, there are multiple works tackling heterogeneity in models and knowledge explicitation in a separate way but we found a lack of a consistent and grouped solution. Thus, we developed a method to bring together these characteristics which are the core of our proposal.

Our proposal

Having this industrial problem in consideration, the goal of our work is to propose an approach to support the management of inter-models heterogeneity and to deploy it in an Aircraft Model Based Systems Engineering setting.

Our first objective is to reduce the heterogeneity regarding the different nature of models and modeling languages. We suggest the use of a shared and unified modeling language to work in the same framework. Thus, applying meta-modeling techniques we export the source models to a common framework where we can manipulate the models without modifying their original definition.

That leads us to a second objective, the manipulation of exported models from a knowledge point of view. That means making explicit the implicit knowledge to annotate the models. By annotating them we add information which is essential to understand the models and to correctly integrate them. This integration is the necessary step to allow the validation of inter-model properties which also need the use of formal knowledge to be explicitly expressed.

As a conclusion, our approach supports the management of heterogeneous models by

- describing, modeling and verifying some inter-model constraints and relationships between pre-existing heterogeneous models used in a System Engineering process;

- making explicit, formalizing and exploiting additional knowledge usually not expressed by the engineers to express these constraints and relationships.

From a scientific point of view, the main contributions of our work to the problem of models' heterogeneity take part in two ways. On the one hand the formalization of engineers'

implicit knowledge to annotate architecture and functional engineering models. On the other hand the use of a unified and common framework to export the source models aiming at syntactically homogenize them without modifying the original elements. These two ideas are combined in a method which is the core of our proposal and which allows engineers to interconnect heterogeneous models of the same life cycle level and to validate inter-model constraints over them. From an industrial point of view, the proposed method is supported by a process-oriented prototype which can be used as a basis for a future industrialization of the approach.

Structure of the document

This thesis is organized in 4 parts.

The first part introduces the state of the art and includes chapters I, II and III. Chapter I describes the problem of heterogeneity of models in Systems Engineering and analyzes current approaches aiming at solving it. Chapter II introduces the notion of implicit knowledge and discusses its formalization using ontologies as a support for the validation of inter-model constraints. Chapter III presents the industrial context of our work and the expected benefits of the proposed approach.

The second part focuses on our contribution. Our proposal is introduced in Chapter IV with the description of our knowledge-based inter-model constraint verification approach.

The third part concerns the formalization and validation of our approach. The formal validation of the proposal using the EXPRESS modeling language and the operational validation which has been carried out with the ECCO toolkit are discussed in Chapter V. Chapter VI analyzes the four case studies implemented applying our approach.

The fourth part discusses the industrial implementation of our proposal. The development of the case studies has allowed us to implement a prototype which is described in Chapter VII. Finally, in Chapter VIII we discuss the industrialization requirements and the necessary enhancements to improve the current version of the prototype.

Chapter I Heterogeneity of models in Systems Engineering domain

Summary

I.1. Introduction 7
I.2. System modeling 7
I.2.1. Notion of system 7
I.2.2. Models Typology 8
I.2.3. Functional modeling 9
I.2.4. Modeling languages 10
I.3. Systems Engineering 12
I.3.1. Systems Engineering Life Cycle description and instantiations 14
I.3.2. Collaborative design 15
I.4. Inter-model relations 16
I.4.1. Typology 16
I.4.2. Relations in the process development 16
I.5. Heterogeneity 17
I.6. Current approaches to handle heterogeneity 18
I.6.1. Model-Based Systems Engineering 18
I.6.2. Integration approaches 20
I.7. Conclusion 21

Abstract. Complex systems involve multiple domains and need the integration of heterogeneous sources all along the life cycle of a system. Amongst these sources, models are necessary to better organize the development of a system and manage its intrinsic complexity. Current engineering approaches as the Model-Based Systems Engineering (MBSE) focus on this aspect. Nevertheless, these models contribute to the development of a system at different stages of its life cycle. Therefore, they aim at different objectives and take into consideration a great variety of domains. At the same time they need to be interconnected since they are part of the overall design of the system and they have to be consistent from the point of view of general properties and constraints. Thus, models are not exempt from heterogeneity issues and proposals are needed to handle them.

I.1. Introduction

The complexity of the design of modern systems makes necessary the use of models in order to guarantee a good management and the correctness of the system to be built. Nevertheless, even though models are essential to design such complex systems, the organization of large enterprises involves work sharing and collaboration of engineering teams of different domains and different backgrounds. In such a context, each team develops a part of the system using its own models and maybe with different methods and modeling practices. Our experience shows that even with a common methodology, very often other aspects as expertise domain, modeling comprehension and particular terms lead to heterogeneous models and can make difficult the inter-model validations.

I.2. System modeling

A model of a system "is a description or specification of that system and its environment for some certain purpose" (Mukerji & Miller, 2003). Models being simplified or abstract representations of a system, or of a part of it, with a particular objective, should allow engineers to better master the design of the system. A model represents sub-system context and interfaces, internal structure and behavior. Models are developed independently on the basis of specifications and requirements, but from a validation and verification (V&V) perspective they might need to be integrated someway or at least verified or validated from an integrative perspective. Depending on the stage of the development cycle, which is directly related to the level of detail, and on the characteristic of the design, we can identify different types of models.

I.2.1. Notion of system

One of the possible definitions of a system is that it is "an integrated set of elements, subsystems, or assemblies that accomplish a defined objective" (Haskins, Forsberg, Krueger, Walden, & Hamelin, 2010).

In Systems Engineering context, we use a typology of systems in order to take in account the variety of natures of systems:

- A "*System of systems" (SoS)* is a set of different systems which collaborate to provide more functionality and performance that the sum of the individual systems, a concept called emergent behavior (Krygiel, 1999). For example in a military context (C2 constellation (Sweet, 2004)), or in air traffic management (SESAR (Eurocontrol & Commission, 2010)), etc.

- The *system* corresponds to a product, e.g. an aircraft; or, as stated by system engineering standards, EIA 632 (EIA & ANSI, 1994), that will be discussed later in the document, the system includes both the operational *end product* (the aircraft) and the so-called *enabling products* (such as development, production, test, etc…).

- The *sub-systems* compose the previously defined system, for example the landing gear, the engine, a guidance system, etc. A sub-system can be decomposed into other sub-systems.

- The *equipments* compose the previously defined sub-system for example a display, a calculator, …

I.2.2. Models Typology

A first typology of models distinguishes between physical and functional modeling.

- **Physical modeling of a system or equipment**: for understanding the system's physical properties. Physical modeling uses the digital mockup, geometrical descriptions, and model physical properties such as mechanics, thermodynamics, aerodynamics, etc… in order to understand the physical behavior of the system in some experimental conditions.

- **Functional and structural modeling of a system or equipment**: for modeling the functional behavior of the system, i.e. identification of the systems functions, their interactions and their structure (architecture). Since in our research we focus on structural and functional models of a system or equipment we detail this topic further on.

A second typology distinguishes between typical possible intents of the model in the system engineering process:

- **Specification or Descriptive models** (Gonzalez-perez & Henderson-sellers, 2007): to represent the needs and to validate architectural choices, starting from the requirements. It is believed that specification models can be the starting point for design models (see later MBSE approach in section I.6.1). These models are produced in early phases of the development of a system (sub-system or equipment).

- **Design or Prescriptive models**: they are the detailed specification of the system, including the detailed and complete definition of the interfaces and functions to be

implemented. They are often given to the subcontracted partner as an input for their development.

- **Implementation models**: for example the software that implements a function (exact representation of the function). It is the most exact representation of the system (*end product*), showing its dynamic behavior. Used to test and validate sub-system or equipment integration into the system.

We can also classify models according to their goals:

- **Models for prototyping**. They are means of validation and verification. The objective of this type of models is to master the requirements and the main architecture choices.

- **Models for early integration**. They are used to manage the interfaces and the interactions in order to minimize the time and the cost of the physical integration.

- **Models for simulation**. These models represent and simulate the logic behavior and the interactions with external systems and are means to validate a specification and to perform test measurements.

- **Models for communication**. Models are also a formal way of communication, so some models, usually high-level detail models, can be used to introduce the main characteristics of a system to a non-specialized audience.

- **Models for generating code**. They are means of development and their objective is to reduce the number of iterations between designers and code developers (code, test cards…).

I.2.3. Functional modeling

Functional analysis is an approach aiming at describing the group of functions of a system and their relations. The main activities of functional analysis make it possible to:

- identify, group and classify the functions.
- characterize the functions (criteria, performances, relations…).
- guarantee the validity of a function, i.e. whether it is necessary or not for the system.
- decompose the functions, i.e. to organize the functions into a hierarchy

Functional analysis is a method to translate the needs of the customer into useful functions but keeping the choice of solutions open. Therefore, the objective of the method is to provide a set of functions and a functional architecture (hierarchy and relations) to designers without advocating for a particular implementation.

From the beginning, functional analysis has been supported by models. Functional modeling has evolved along with the modeling approaches; from the early analytical approaches as FAST (Snodgrass & Kassi, 1986) and SADT (Marca & McGowan, L., 1987) to the most recent object-oriented approaches as UML (OMG, 2011a). Thus, functional modeling is a discipline with a long history and a high degree of maturity which has been incorporated to disciplines of wider application areas as Systems Engineering (see I.3 for details).

I.2.4. Modeling languages

In order to build models we need appropriate modeling languages. Nowadays two different approaches emerge concerning modeling languages: domain specific languages (DSL) and general-purpose languages. DSL (Abouzahra, Bézivin, Didonet, Fabro, & Jouault, 2005) are languages focused on a particular domain and have precise semantics whereas general-purpose languages are not limited to a domain. A classical example of general-purpose language is UML whose semantics ambiguity is notorious (France, Raton, Evans, Lano, & Rumpe, 1998). In the case of UML, the use of the notion of *profile* involves mechanisms to reduce these semantics problems by providing more specific views of UML, e.g. ModelicaML (Pop, Akhvlediani, & Fritzson, 2007) (Paredis, Bernard, Koning, & Friedenthal, 2010) which is used to graphically represent the Modelica (Fritzson, 2003) simulation model. In our case studies we have treated models expressed with two different modeling languages which are briefly described below.

SysML modeling language

SysML (OMG, 2008) is a modeling language specialized in Systems Engineering domain which provides several kinds of abstraction types (structure, state, processes) to model a system. SysML is not a profile of UML but an extension of UML 2 (OMG, 2009). So although the aim of SysML is to ease modeling in Systems Engineering one cannot consider it a DSL and SysML shares the ambiguity of UML. Thus, depending on the applied modeling rules one element can be represented in quite different manners. An example of modeling rules in SysML is, for instance, to represent the different types of aircraft programs as classes. In contrast, a semantically different approach could be to use one class named *Aircraft* and an attribute of it to distinguish the different programs.

SysML reuses a subset of UML 2 constructs and extends them by adding new modeling entities and two new diagram types (see Figure 1 from OMG). The diagrams provide multiple views of the same system model. The *Behavior Diagrams* describe the sequence of events and activities that the system executes. The *Requirements Diagram* allows the graphical representation of requirements. Concerning structure diagrams, *Block Definition Diagrams* (BDD) are used to illustrate the interconnections between the system and its external systems whereas *Internal Block Diagrams* (IBD) refer to the internal structure and the interconnections between parts of the system.

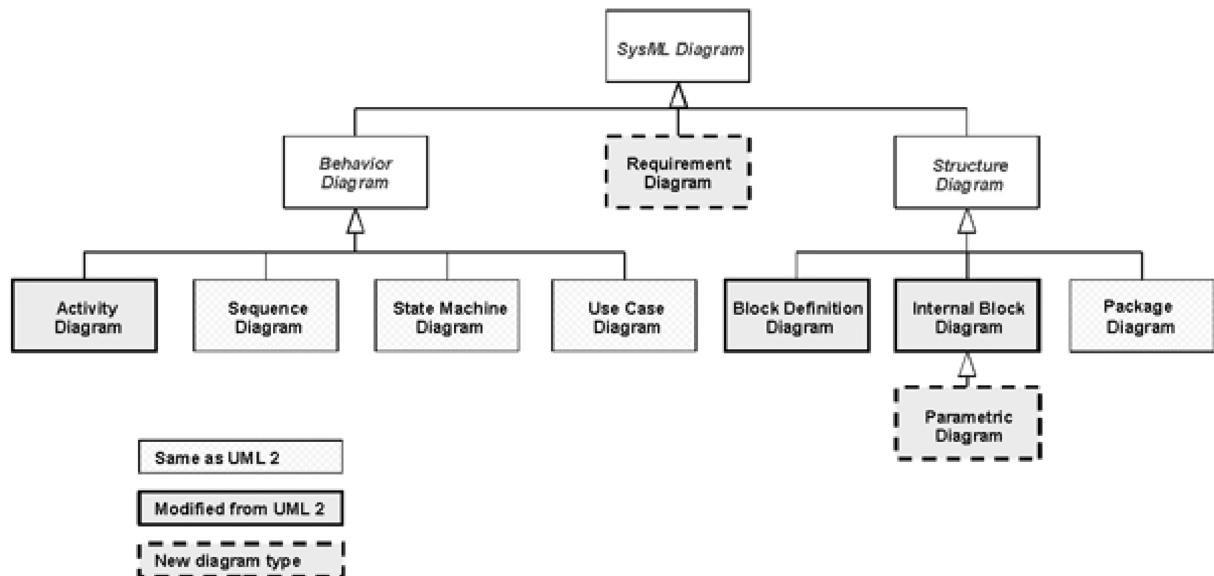


Figure 1. SysML diagrams from OMG

CORE modeling language

CORE is a function-oriented modeling language integrated in a tool developed by Vitech (Vitech Corporation, 2011) of the same name. Like SysML, the CORE application domain is Systems Engineering but, in contrast, it has precise semantics which place it in the DSL category. The tool permits the use of different Systems Engineering schemas, i.e. different meta-models depending on the chosen modeling approach. Thus, basic CORE schema consists of a set of structural entities representing the most important concepts of Systems Engineering modeling: Requirements, Functions, Components, Interfaces, Links… Whereas DoDAF schema, for example, extends it by adding entities as Mission, Operational Task, Architecture… Each schema is based on a set of primitive language concepts containing elements, relationships, attributes, attributed-relationships and system control constructs. The system control constructs are used in the graphical representations which complete the structural view. Such graphical representations aim at describing the behavior of the system, being eFFBD (Long, 2000) the central one. Moreover, eFFBDs (Figure 2 illustrates an example) allow modelers to simulate the behavior of the system which is an outstanding feature of the CORE tool.

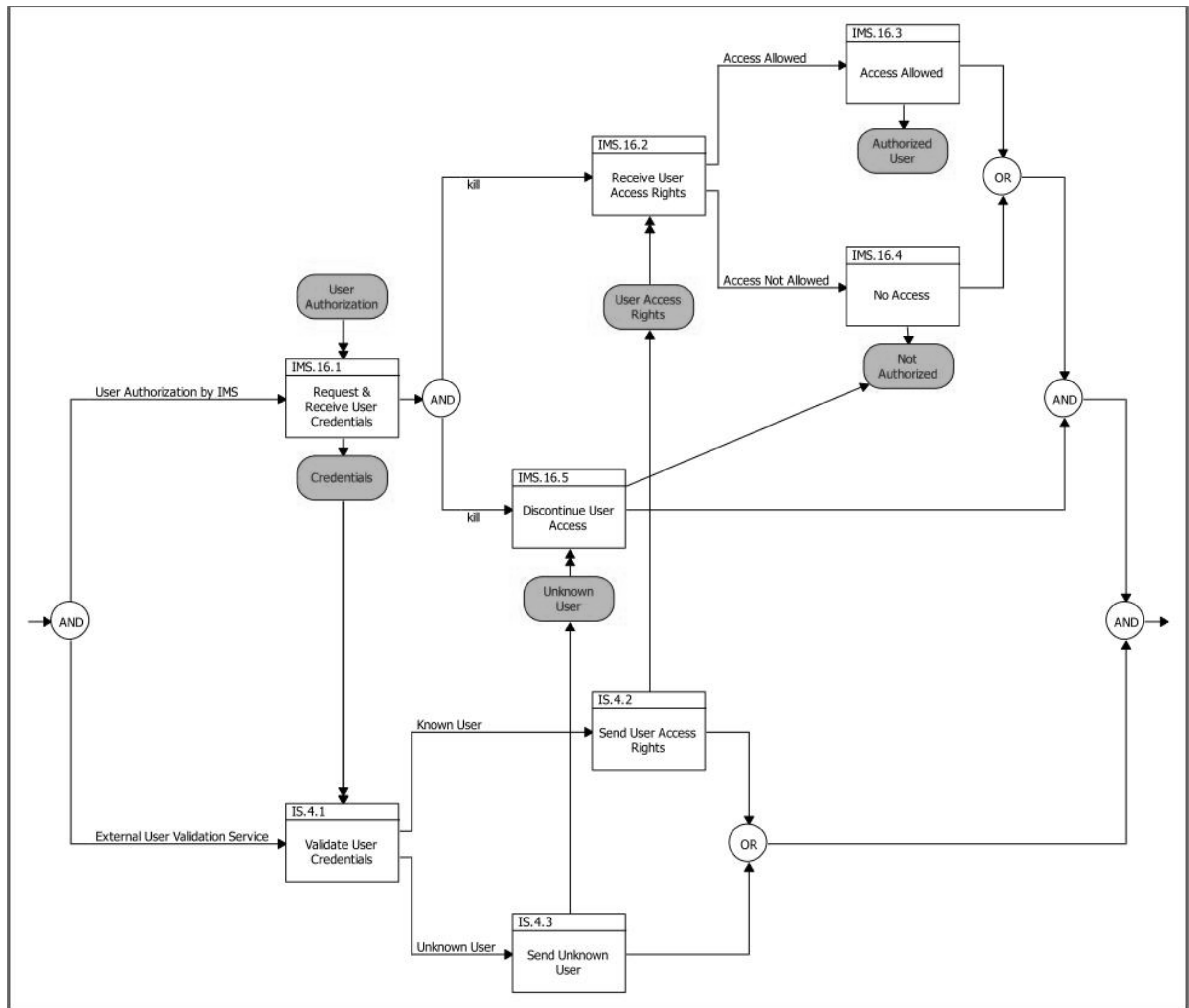


Figure 2. eFFBD diagram illustration (Vitech Corporation, 2011)

I.3. Systems Engineering

New technologies, particularly computer-based technologies, have contributed to the development of more efficient and powerful systems but also more complex. First significant attempts to manage such a complexity have a military origin in the 1960s and were the basis of a new discipline: Systems Engineering (SE). As described in (NASA, 1999), "*Systems engineering is a methodical, disciplined approach for the design, realization, technical management, operations, and retirement of a system*". we would also add that it is a multidisciplinary approach, as opposition to software engineering, and that aspect is one of its major challenges. As seen in Figure 3, several standards have been developed for Systems Engineering domain and others for Software Engineering which is a complex domain itself. Software Engineering is very close to Systems Engineering since nowadays systems cannot be understood without the participation of computing. Thus, from the eighties, standards of Software and Systems Engineering have evolved in a parallel way and some ideas have been exchanged from one discipline to the other one.

Concerning Systems Engineering standards, they were initially developed in a military context since armies (USAF, 1969) were the first ones to tackle the management of missions involving complex systems. In 1994 two civilian standards emerged (EIA & ANSI, 1994) and (IEEE, 2005). Particularly the EIA standard gained popularity and inspired other new standards as ISO 15288 (ISO, 2008). The EIA standard defines different processes grouped in several groups: technical management, acquisition & supply, system design, product realization and technical evaluation. The processes are organized around the concept of *building block*. A *building block* is made up of the system, which is the object of the requirements. This system is composed of one or more *end products*, which perform the operational functions, and of *enabling products* (test, training, development, disposal, production and support). As the development of a system is quite complex, usually an *end product* is decomposed into subsystems each of them being a *building block*. Thus, *building blocks* support a top-down development. In turn, ISO has enlarged the coverage of the EIA standard in order to take into consideration the entire life cycle of the system. Thus, the ISO standard describes agreement, enterprise, project and technical processes but considering the operation, maintenance and disposal stages as well. These various norms have been updated during their evolution and currently Software and Systems Engineering standards are fully aligned.

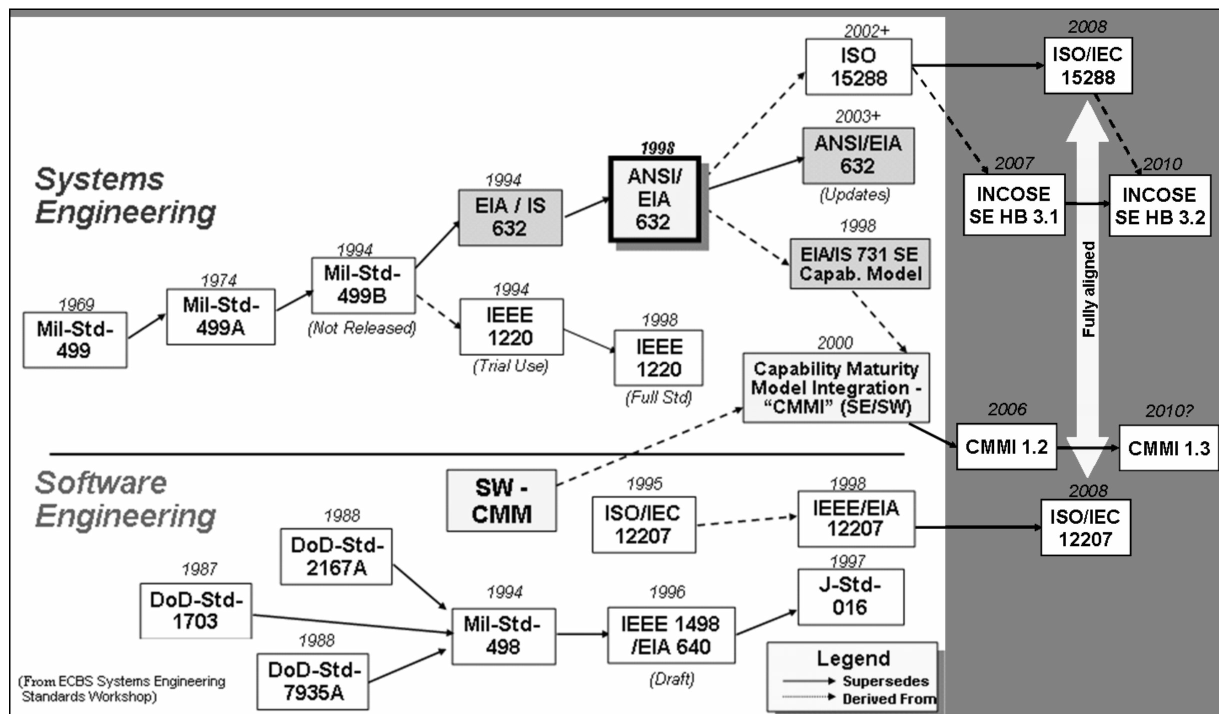


Figure 3. Systems and Software Engineering standards evolution up to 2010 (Monzón, 2010)**.**

I.3.1. Systems Engineering Life Cycle description and instantiations

A System Life Cycle is the description of the different stages of the existence of a system. This description may only refer to the development of the system, from analysis to integration (e.g., Figure 4 illustrates the process development of an aircraft), but most modern visions take into consideration a larger cycle. That means spanning from the concept of the system to the retirement or end of use of it. It is the approach of the ISO 15288 standard which is described later: "*A life cycle can be described using an abstract functional model that represents the conceptualization of the need of a system, its realization, utilization, evolution and disposal*".

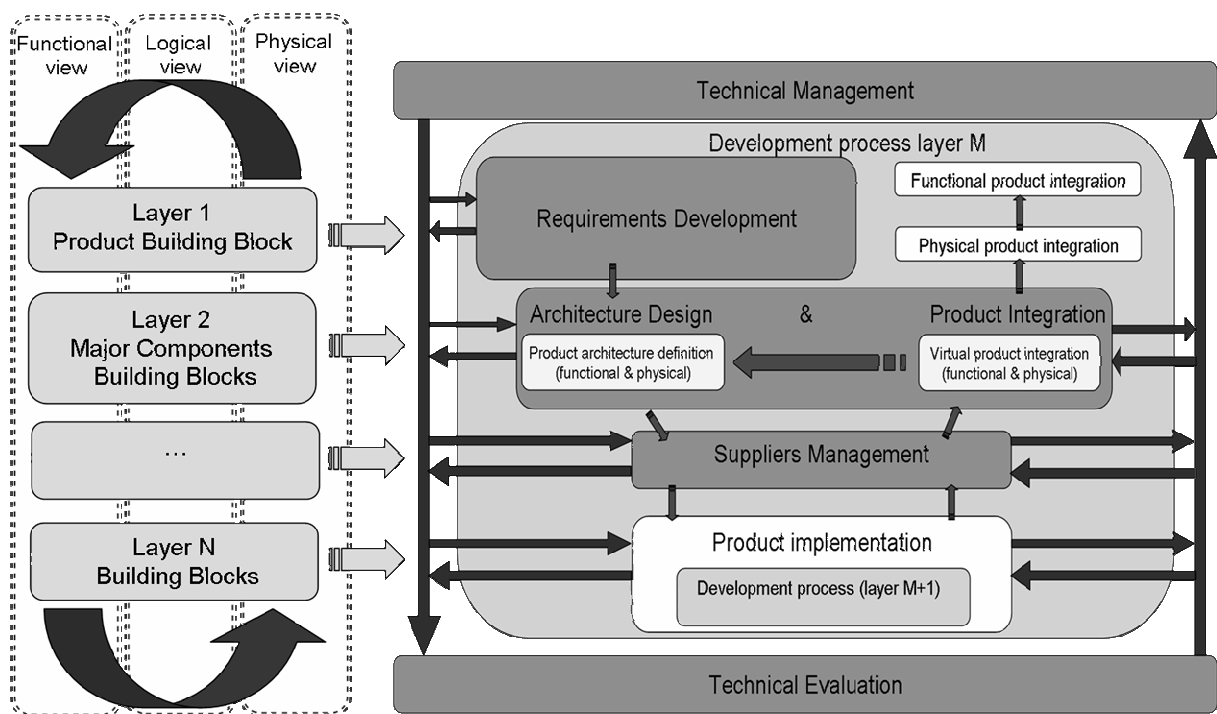


Figure 4. Aircraft Process Development

According to the SE standards introduced in the previous section, all of them recommend starting the development of a system by eliciting the requirements which specify the overall system. As requirements' management is a difficult task in itself, an entire discipline, the Requirements Engineering, has been established. Thus, Requirements Engineering is the entry point of SE and a very important area of interest. A good management of requirements is the key for the success of a project (Honour, 2004), however all the entire system life cycle must be managed in the end and that is the foundations of Systems Engineering.

Based on requirements, the development progresses through distinct stages from the system concept description to the subsystem and component detailed description. The system is refined in an iterative way to progressively reduce the level of complexity to be managed. Each refinement step requires a validation with respect to the expressed requirements. This

kind of development approach corresponds to the classical V-development cycle. In a V-like image, the cycle begins by performing the descending (first part of V) activities, which concern design and construction of the system (realization), and then the cycle lasts following the ascending (second part of V) branch, which is related to the integration and validation activities. Nevertheless, there is not one only way of implementing a V-cycle and we can find different techniques to implement it, some examples are: the *plan driven approach* (Boehm & Turner, 2003), which consists of starting with a quite stable set of requirements and of performing the development activities consecutively; the *incremental and iterative development* (Larman & Basili, 2003) which applies incremental cycles; c*ollaborative design* (described in next section) which involves the simultaneous distribution of tasks between multiple teams of a project in order to optimize the time to achieve the overall goal; the *Agile* approach (Kelly, 2008) which offers flexibility since a lot of usually hierarchical tasks are actually performed in a parallel way; and *Lean* approach (Oppenheim, 2009), one of the most successful current trends, which is based on three guidelines: the value (mission assurance in the context of Systems Engineering), the waste and the process of creating value without waste.

I.3.2. Collaborative design

Actually, nowadays the systems managed are so complex that traditional monolithic design would be neither sufficient nor efficient enough in a time-to-market perspective. Teams have to share information and work together in order to increase productivity.

In despite of its advantages, the advent of work sharing techniques such as collaborative design (Kvan, 2000) can make complex the modeling activity. Collaborations (Yoshimura, 2007) involve the simultaneous distribution of tasks between multiple teams of a project in order to optimize the time to achieve the overall goal. In the context of models, it means that several models of the same system or part of it are developed by different actors commonly having a variety of points of view. As described by (Tudorache, 2006), the collaborative design process denotes multiple teams, possibly belonging to different technical domains, who develop models in a heterogeneous way. Therefore, the heterogeneity of models becomes a problem when engineers need to share them amongst different teams. Due to the complexity of the concerned systems, both Model Based Systems Engineering (see section I.6.1) and collaborative design are necessary and complementary but a new degree of difficulty is added. Models are consequently distributed and the design teams use different modeling languages and have different approaches, objectives and vocabularies. Thus, the current challenge is the capability to manage the heterogeneity of models and to have a global view of the modeling results.

I.4. Inter-model relations

I.4.1. Typology

One can also see models as a group of entities and their relations. Relations are central in modeling activities but contribute to the complexity of models at the same time. There are different types of relations and depending on their characteristics we can classify them in three main groups:

- Intra-model relations. They are the classical relations between entities of a model: association, aggregation, inheritance, instantiation…

- Inter-model relations. They imply more than one model and are relations between entities of the models, i.e. we connect entities of a model to the ones of another model.

- High-level inter-model relations. These are relations between models without taking into consideration the content of the models. In (D. Kolovos, Paige, & Polack, 2008) there is an analysis of this kind of relations where we can find an exhaustive list containing notions as *uses*, *extends*, *refines* and so on.

However, if we consider the Systems Engineering development process we can also talk about *same level* and *top-down or bottom-up relations*.

I.4.2. Relations in the process development

Our main hypothesis is that models are used in each of the Systems Engineering development process descending layers, i.e. from "Requirements Analysis" to "Detailed implementation in equipments" of Figure 4. Inter-model relations, *same level* and *top-down or bottom-up*, are an important source of heterogeneity in such a development process. In this context, advancing from a layer to another one in the process flow, i.e. overcoming a milestone, implies some kinds of refinement of the models of the upper layer (top-down) or composition of models of the lower layer (bottom-up). Defining refinement or composition rules and keeping traceability between models is then important for design rationale purpose. We call that type of inter-model relation a *top-down or bottom-up relation*. Given two models M1 and M2 released at successive steps of the process, M2 refines M1 if it adds details to it. For example an UML class *Airframe* in model M1 is refined in model M2 as numerous classes: *Wing*, *Control surfaces*, *Fuselage*, *Drop tank* and *Vertical* s*tabilizer*, linked by association, inheritance, etc. Nevertheless, before achieving a milestone, several domain specific models usually exist. The models of the same stage should be validated after establishing another type of inter-model relations called *same level relations*. In this case,

defining relations and constraints is important for ensuring that no conflict exists between models: no inter-model constraint is violated. Actually, a *same level relation* between two models is induced by the fact that one (or several) field regards them having a particular objective (for instance, to compare or to validate a feature of the two models). If more than one field is implied, usually they are examining the models at the same development step in the general Model-Based Systems Engineering processes. The observers indeed are interested in the same kind of information. They share a comprehension level, but usually they have different points of view (Auzelle, Garnier, & Pourcel, 2009) on the models according to their objectives. The modeling point of view is the consideration angle from which observers are projected over a modeling language.

I.5. Heterogeneity

In a collaboration frame of work, heterogeneity is a logical consequence of the involvement of different engineering groups, models, domains, modeling languages and paradigms. Actually, one could consider heterogeneity as necessary in such a context since a great variety of points of view and proposals improves the design and promotes the innovation. Nevertheless, approaches aiming at reducing this heterogeneity are demanded in the case of heterogeneous models which need to be shared.

Klein (M. Klein, 2001) identifies four types of heterogeneity or mismatches: conceptualization (what one wants to model), explication (how one specifies a concept), terminological (concerning the used words) and encoding (data formats). In our context, heterogeneity in design models has diverse origins.

• Data exchanged between applications (encoding): for instance, one application uses an identifier of 9 digits for an engineering data and another application uses an identifier of 10 digits.

• Objectives of the models (conceptualization): e.g., one model is used to discuss with the customers whereas another model is focused on simulation tests.

• Models structures (terminological): e.g., one model describes the *Power Plant* which is called *Engine Unit* in a different model although both refer to a same concept.

• Modeling languages (explication): e.g., one given design model can be formalized within different modeling languages, like SysML or CORE modeling languages.

• Modeling paradigms (explication): e.g., SysML is an object-oriented language whereas CORE uses a function-oriented language approach.

In order to go further in this topic, (Silva, 2007) performs a detailed analysis of heterogeneity in systems interoperability context.

I.6. Current approaches to handle heterogeneity

Several approaches deal with the problem of heterogeneity in the modeling context. One of them is to establish a common modeling language (INCOSE, 2007) in Systems Engineering such as SysML. In models of computation area, (Eker & Janneck, 2003) suggest an approach which groups locally homogeneous models together using an actor-oriented architecture. Another example for modeling in a heterogeneous domain environment is Rosetta (Alexander et al., 2003). It uses facets to allow the re-use of components in different domains. Some studies are more focused on data heterogeneity by enabling the exchange of product model data between different systems, as STEP (Pratt, 2001) which intends to cover the data of the product entire life cycle based on implementable models known as Application Protocols. Even though this may solve problems at a data level, this is not the only layer source of heterogeneity in a Model-Based Systems Engineering process. In addition, although that kind of solutions may assist in obtaining a consensus in new models, they cannot bear with the heterogeneity of existing models.

I.6.1. Model-Based Systems Engineering

A means for reducing the heterogeneity while designing a system is the application of well-defined methods. This approach is the main concept of the Model-Based Systems Engineering (MBSE), also known as Model-Driven System Design (MDSD) (Estefan, 2008).

MBSE methods

Currently several methods implementing the MBSE principles are available. The main ones in our industrial context are briefly described below.

INCOSE Object-Oriented Systems Engineering Method

OOSEM (Lykins & Ave, 1999) is a method originally conceived by the Software Productivity Consortium in collaboration with Lockheed Martin Corporation. The first versions of the method were UML focused until the participation of INCOSE. INCOSE[1], the main Systems Engineering association in the world, reoriented it to use the SysML language. That kind of languages is view/diagram driven where a model consists of a group of diagrams, corresponding to the notion of modeling points of view, and elements that are usually accessible from them.

[1] http://www.incose.org/

CORE System Definition Guide

CORE System Definition Guide (SDG) (Vitech Corporation, 2007a) is a guideline to perform Systems Engineering activities using the CORE language developed by Vitech Corporation. CORE is driven from a single integrated model, it is driven by classes and elements rather than individual diagrams as in SysML or UML. A variant or extension of this guideline is the Architecture Definition Guide (ADG) (Vitech Corporation, 2007b), which provides a structured approach for populating a CORE project with architectural definition information using the Department of Defense Architecture Framework (DoDAF) schema (Department Of Defense, 2007). It complements the CORE SDG by taking into consideration the DoDAF standard, mainly to add the operational dimension. The main objective is to obtain an Operational Architecture (Figure 5) in parallel to the System Architecture (Functional and Physical) output of the structured approach of SDG. As a consequence, the ADG is consistent with this approach and it has to be applied jointly with the SDG to obtain the Architecture.

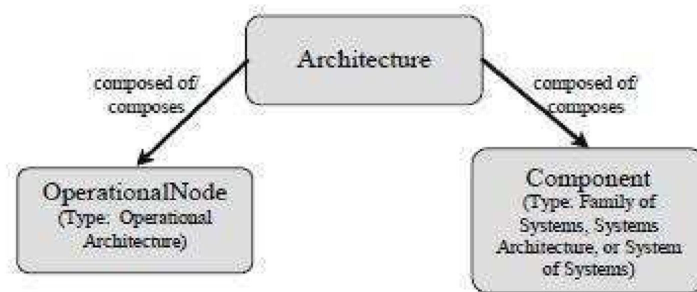


Figure 5. The Architecture consists of Operational –a group of Operational Nodes- and System Architecture –a hierarchy of Components-

IBM Rational Harmony for Systems Engineering

HARMONY (IBM, 2011a) is a model-based Systems Engineering methodology suggested by IBM which is supported by IBM products. The process is built on UML and SysML. It consists of a tool chain which follows a flow of efficient design iterations covering:

- System specification
- Requirements analysis
- System architectural design
- Interface definition
- Validation

Modeling languages issue

All the aforementioned approaches are suitable for the development of products based on models. Nevertheless they do not face the problem of combining several modeling languages which is very common in large enterprises. We need multiple modeling languages since complex systems involve multiple domains or logics (Mossakowski, 2004). Next section analyses several approaches concerning multi-modeling.

I.6.2. Integration approaches

The problem of distributed autonomous and heterogeneous sources has been addressed in several domains such as system interoperability (Vinoski, 1997)(Curbera et al., 2002)(Cerami, 2002) data integration and model integration.

Distribution, autonomy and heterogeneity are the main needs which originated data models like multidatabases and federated databases (A. P. Sheth & Larson, 1990)(Pierra, 1992)(Hose, Roth, Zeitz, Sattler, & Naumann, 2008). Different mapping approaches can be used to implement these data integration systems (A. Halevy & Ordille, 2006) (Bakhtouchi, Chakroun, Bellatreche, & Aït-Ameur, 2011). Schema mappings like global-as-view (GaV), local-as-view (LaV) (Seng & Kong, 2009)(Lenzerini, 2002), GLAV (Generalized local-as-view) (Friedman, Levy, & Millstein, 1999)(A. Y. Halevy, Arbor, & Yu, 2007) or data exchange systems (Kolaitis, 2005) specify the relationships between a source schema and a target schema. In the model integration research area (Caplat, Sourrouille, & Pascal, 2003), a mapping is a morphism (Antonio, Missikoff, Bottoni, & Hahn, 2006) consisting of a set of functions which transforms a model *M1* to a model *M2* and of the set of relations enabling the traceability (1:1, 1:n, m:1, m:n) between corresponding entities of both models. When models are expressed with the same modeling language we call it an endogenous mapping, otherwise it is an exogenous mapping. Some researches on mapping two particular different modeling languages have already been done, e.g. mapping DFD (Data flow diagram) into UML activity models (Tran, Khan, & Lan, 2004). The main drawback of mapping techniques is that such solutions are very hard to maintain (Ruzzi, 2004).

This leads us to another type of morphism: the transformation of models. The transformation of models is central in the MDA (Model-driven architecture) (Mukerji & Miller, 2003) approach. In this approach the correspondence between models is established by applying transformation rules to their meta-models (see Figure 6). Thus, the equivalence is done at meta-model level as in our proposal. Applying this technique (Boronat, Knapp, Meseguer, & Wirsing, 2008) suggests a unique multi-model language compliant with several modeling languages in order to guarantee the consistency of models from the syntactical point of view. Some other researches have suggested the use of common meta-models (Hardebolle & Boulanger, 2008). In weaving modeling (Jean Bézivin, Didonet Del Fabro, Jouault, &

Valduriez, 2005) a third meta-model, the weaving meta-model, is used to represent the combination of models although it is not suitable for evolution scenarios (Hessellund, 2009).

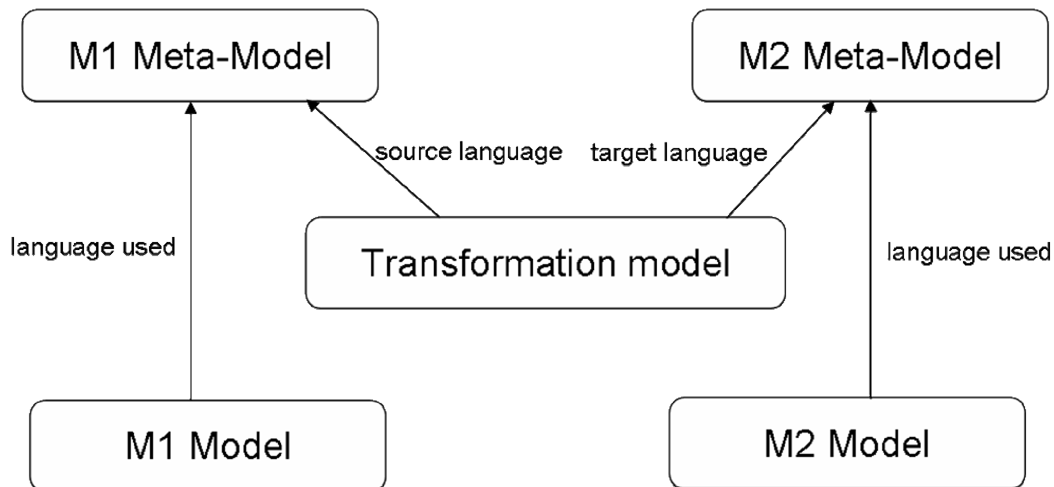


Figure 6. Transformation of models via meta-models

Mapping and meta-modeling are techniques currently applied to the problem of integration of models.

I.7. Conclusion

Nowadays, the complexity of systems entails the need of multi-domain disciplines such as Systems Engineering in order to manage such a complexity. Engineering best practices recommend the use of models during the engineering process. The core of our work is focused on the design stages of the functional and physical architecture of the *end product*. The development life cycle and the collaborative work, amongst other factors, lead to an increase of the heterogeneity of these models. This heterogeneity becomes an issue when models need to be shared or integrated in order to guarantee their consistency. Even though there are multiple and successful approaches for the integration of models, there is still a gap to take into consideration implicit knowledge, i.e. the knowledge of engineers that is not in the models themselves but that is essential for understanding and validating them. We base our work on meta-modeling principles, however, our proposal complete them with the use of implicit knowledge made explicit as described in next section.

Chapter II Knowledge models to integrate and validate heterogeneous models

Summary

II.1. Implicit knowledge 25

II.2. Formalization of knowledge 25

II.2.1. Need of ontologies 26

II.3. Ontologies and annotation of models 27

II.4. Validation of inter-model properties 28

II.4.1. Requirements 29

II.4.2. Property languages 29

II.5. EXPRESS modeling language 30

II.5.1. Meta-modeling 30

II.5.2. Expressions with EXPRESS 31

II.5.3. The choice of EXPRESS 32

II.6. Conclusion 33

Abstract. Models are the result of the work of engineering teams which represent a system or a part of a system from a particular point of view. Nevertheless, to correctly interpret a model, additional knowledge issued from the engineers in charge of such a model is necessary. It is the implicit knowledge which is kept in engineers' minds. There are multiple ways of formalizing that knowledge but we think that in the Systems Engineering context formal ontologies are suitable, mainly due to their precise and consensual nature. We defend the formalization of implicit knowledge as a means to integrate and validate heterogeneous models. Thus, by making explicit the implicit knowledge we can annotate such models to ease their integration and to support the validation of inter-model properties.

II.1. Implicit knowledge

The problem of heterogeneity in the context of MBSE is increasingly drawing the attention of researchers, and several approaches deal with it. We think that one factor for heterogeneity may, paradoxically, be used at the same time to reduce it. It concerns the knowledge of engineers which is common in their context but that is usually not included in models. We call it implicit knowledge and its formalization is a key factor to understand the models when they are shared or analyzed as a whole. An example of this kind of implicit knowledge can be found in a scenario such as one aircraft engineering team designs the avionics system, i.e. critical domain systems from the security point of view, whereas a different team models non-critical aircraft software. None of the teams indicates the domain of their models since it is obvious for them; nevertheless this knowledge is crucial when both models are checked jointly, since different constraints apply to them. Another case arises when engineers use different concepts or names for the same element using different words in their respective models. This can be due to incomplete specifications or to the evolution of the models themselves -e.g. an interface has changed name and only one team is aware of this amendment whereas another team still uses the old name-.

Engineers have their own knowledge concerning the models they are working with. Nevertheless, this knowledge is not always made explicit in the content of the models. Therefore we think that using the content of the models is not enough. Some studies (Vajna, 2002) (Damjanović, Behrendt, Plössnig, & Holzapfel, 2007) have shown that design and particularly models need additional data and knowledge to be completed and understood. Consequently, knowledge should be taken into consideration in order to find or establish links between models. It is an engineering issue and as (R. Klein, 2000) demonstrates in his description of the MOKA framework, design knowledge should be processed in a specific way.

II.2. Formalization of knowledge

In computer science, the formalization of knowledge is an old topic which has been tackled with different approaches. In our domain, we can have several models representing different types and levels of knowledge. (Chen & Chu, 2007) proposes a classification of engineering knowledge involved in product design. Some other examples of types of knowledge can be found below:

- Modeling semantics, e.g. a *Block* represents a component in SysML.
- Terminology, e.g. the same interface named differently in two models.

- Modeling process semantics, e.g. the checkpoints to ensure consistency of external interfaces.
- Domain semantics, e.g. the phases of the aircraft flight.

Translating the concepts that are in the mind of engineers is not an easy task but it is a key activity in domain modeling, in requirement analysis and in computer design. One classical implementation of knowledge in design is the elaboration of conceptual models in order to obtain a consensus in the main concepts one wants to work with. They are relatively simple models, especially if we compare them with the needs of requirements knowledge representation. The main objective of this discipline is to formalize the concepts non ambiguous, so therefore the formalization of knowledge is more complex, including models, instances of these models known as knowledge base and rules or mechanisms to reason and to produce new knowledge. Therefore, depending on the complexity of knowledge to be treated, different techniques or tools can be applied. Amongst them ontologies play a key role.

II.2.1. Need of ontologies

Ontology (Chandrasekaran, Josephson, & Benjamins, 1999) is a word borrowed from philosophy meaning “the study of the kinds of things that exist”, nonetheless the most well-known actual definition is “an explicit specification of a conceptualization” (Gruber, 1995).

In (Jean, Pierra, & Ait-Ameur, 2007) the authors suggest that a domain ontology is a “formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them”. They defend that a domain ontology needs to be formal, consensual and to have the capability to be referenced. These three characteristics are relevant for our proposal of employing engineering explicit knowledge to reduce the heterogeneity of models. The formal aspect is important to avoid ambiguity and to allow reasoning capabilities in such a computerized environment. Consensual property is necessary in the multi-domain context of Systems Engineering. Finally a generic identifier is essential to allow a correct knowledge management. Referencing uniquely is the objective of the Uniform Resource Identifier (URI). URI is a standard (Berners-Lee, Fielding, Irvine, & Masinter, 1998) identification widely used in ontologies, and in our proposal, to precisely identify concepts (W3C, 2008). We adopt this definition for our work.

To distinguish ontologies from other mechanisms aiming at modeling concepts, (Oberle, 2006) suggests that:

1. Primary goal of ontologies is to enable agreement on the meaning of vocabulary terms to ease information integration.

2. Ontologies are formalized in logic-based languages and have unambiguous semantics.

3. Ontology languages have executable calculi enabling query and reasoning services.

Ontologies can be used in different domains but following (Pierra, 2008) we classify them into two big categories:

- Linguistic ontologies. They are focused on words, i.e. how concepts are reflected in a particular natural language, we go from words to concepts. Semantic web (Uschold, 2002) is the natural domain of this group and it is the origin of several formalisms, in particular: RDF (W3C, 2004), DAML+OIL (Connolly et al., 2001), OWL (Bechhofer et al., 2004), SWRL (Ian Horrocks et al., 2004).

- Concept ontologies. In this case the areas of interest are the concepts and the properties that are used to represent some part of the world which is described using natural language, i.e. we go from concepts to words. This type of ontologies fits engineering conceptualization. Engineering concepts is the core of the STEP (STandard for the Exchange of Product model and data) project, where each engineering domain is represented as a STEP Application Protocol (AP), for example AP233 (Gert & Eckert, 2000) is the AP for Systems Engineering and has a semantic extension specified in OWL (Spiby, 2007). In close relation with STEP we find PLIB (ISO, 1997a), an ontology model defined in EXPRESS and used for component libraries of industrial technical data. UFO ontology model (Guizzardi, 2008) is another example of employ of ontologies in an engineering domain.

II.3. Ontologies and annotation of models

We formalize explicit knowledge because we want to manage it. One of the most common applications is to use formal knowledge to complete or enrich existing elements like documents and models. The mechanism allowing the association of such elements with concepts of ontologies is called annotation. For document centric annotations readers can refer to the article (Uren, Hall, & Keynes, 2006) and the Edelweiss team work (Mokhtari & Corby, 2009). Below we analyze related work concerning ontologies and engineering models.

In (K. Oliveira, Breitman, & Oliveira, 2009) authors suggest the use of ontologies to compare models. In the petrology and geological modeling context, (Mastella, Abel, Ros, Perrin, & Rainaud, 2007) introduce an ontology of events. Tudorache (Tudorache, 2006)

suggests the transformation of heterogeneous design models into an ontology (enrichment) in order to establish matches at the ontology level. Nevertheless, our objective is to connect current models keeping their own nature; we aim at the collaboration not at the final integration. The (Bräuer, 2007) software-oriented approach consists of mapping each meta-model -automatically- to an ontology and to link all the ontologies to an upper-level ontology (USMO). Concerning inter-model relations, (An & Song, 2008) describes a technique for discovering meaningful associations between design models using complex ontology mappings. As a conclusion, the Tudorache's approach is very close to ours, albeit we keep a clean distinction between the models, and the ontologies used to annotate them.

Regarding annotation, some works have treated the problem of annotating models mainly in the enterprise modeling area. Some of the proposals can be applied to our approach. In (Zouggar, Vallespir, & Chen, 2008) authors suggest a method for linking elements of the models to concepts of an ontology whereas (Boudjlida & Panetto, 2008) describes a more complex framework with different ways and types of annotations depending on the kind of interoperability issue. Therefore, these articles analyze very useful properties to be provided in the characterization of an annotation predominantly concerning categories of annotations according to different points of view (informal, formal, structural, behavior…) and the accuracy of the annotation itself (exact, partial…), since sometimes the engineer is not able to find the exact knowledge concept corresponding to its modeling entity but a similar or a possible one. (Lin, 2004) presents a proposal to use requirement engineering techniques to annotate models. Other approaches (Mandutianu, 2009) recommend the annotation of each of the elements, point-to-point, of the models with an integrating point of view. Concerning current model annotation frameworks, in general annotations are written by domain experts, but some systems like A* (Athena Project, 2006) intends to provide some semi-automatic annotations.

As explained, ontologies have all the needed characteristics to represent engineering implicit knowledge. The annotation of models using ontological concepts allows engineers to enrich their heterogeneous models in order to interconnect them. This integration based on explicit knowledge permits the analysis and validation of inter-model properties.

II.4. Validation of inter-model properties

So far we have described the problem of heterogeneity of models and the lack of representation of the implicit knowledge coming from engineers. These heterogeneous but related models are an issue for engineers because some properties to be validated involve more than one model, as a consequence of collaborating engineering. Thus, once the implicit knowledge is added to the models we need to be able to use it to validate inter-model constraints.

II.4.1. Requirements

The main source of inter-model properties are the requirements. Requirements are the departing point of Systems Engineering processes, they describe the specification of the system and they guide its validation. There are different categories of requirements but quite a common categorization divide them between functional and non-functional requirements.

- Functional requirements describe the functionality of the system.
- Non-functional requirements refer to the characteristics of the system that the user cannot affect. Nevertheless the distinction between functional and non-functional requirements is not always clear and depends on the context. Non-functional requirements are also known as "*ilities*" (security, portability, quality, reliability…). Constraints are commonly included in this category. A constraint describes limits that the system must respect independently of the final solution, e.g. "*the aircraft systems shall reduce interferences according to EMC (electromagnetic compatibility) directives of European Union*".

This general classification of requirements denotes that we can find a large variety of properties to be validated besides the heterogeneity of the models themselves. Thus, the language or formalism that we choose to check a particular requirement must fit the right typology. Moreover, in our proposal such a property language must be compatible with ontologies in order to use the formalized implicit knowledge.

II.4.2. Property languages

In literature we find several types of languages allowing the validation of properties. Most of these languages are optimized for a particular domain or modeling language.

Some property languages envisage a more general use. This is the case of OCL (Warner & Kleppe, 1998) which is a contribution to express constraints over UML models. Nevertheless, the fact that it is a language quite different from UML increases the learning curve for modelers. On the other hand, OCL is considered not convenient for more than one model (D. S. Kolovos, Paige, & Polack, 2006).

As a conclusion, to validate inter-model properties we need a language adequate to the typology of the checked property and able to express properties over more than one model and using ontologies. For the validation of our approach, we decide to set up our ad-hoc constraint language, based on the procedural knowledge model of PLIB (ISO, 1997b) and implemented in EXPRESS modeling language, which fits the needs of our case studies.

II.5. EXPRESS modeling language

For the formalization of our approach, we have chosen the EXPRESS (ISO, 1994) modeling language. EXPRESS is a normalized language defined in the context of the STEP project. It was originally defined to represent product data models in the engineering area and it is now widely used for solving several data modeling problems. The major advantage of this language is the integration of the structural, descriptive and procedural concepts in a common formalism and a common semantics. Semantics of the EXPRESS language is clear and it has allowed a time-efficient implementation of the approach. Furthermore, EXPRESS eases the modularization of the models and the associated code applying the notion of schemas. A schema contains a group of entities, attributes and constraints strongly intra-related. In practice a schema corresponds to a model. As described further on, the notion of meta-model, which does not exist in EXPRESS, has been added.

II.5.1. Meta-modeling

EXPRESS is type oriented: entity types are defined at compile time and there is no concept of meta-class. Each entity is described by a set of characteristics or properties called attributes (see Figure 7).

```
SCHEMA Example;
ENTITY A;                     ENTITY B;
 att_A: INTEGER;                      att_1: REAL;
INVERSE                               att_2: LIST [0:?] OF STRING;
 att_I: B FOR att_3;                  att_3: A;
END_ENTITY;                   END_ENTITY;
END_SCHEMA;
```

Figure 7. Entity and properties in EXPRESS

It is also possible to describe derived attributes in the entity definitions. In Figure 8 a derived attribute att_3 is calculated as the addition of att_1 and att_2.

```
ENTITY B2;
   att_1: REAL;
   att_2: REAL;
DERIVE
   att_3: REAL := (SELF.att_1 +SELF.att_2);
END_ENTITY;
```

Figure 8. Example of a derived attribute in EXPRESS

One of the advantages of using EXPRESS is that the same language supports the expressions of entities constraints and the implementation of functions and procedures. Constraints are introduced thanks to the *WHERE* clause of EXPRESS that provides for instance invariant, and thanks to the global *RULE* clause that provides for model invariant. In Figure 9 the value of attribute *att_1* of entity *A* must be greater than 5 (*WHERE* clause) for each instance, whereas the addition of attribute *att_1* values of the totality of entity *A* instances has to be less than 1000 (*RULE* clause). *QUERY* is a built-in instance iterator function and *PLUS_FUNCTION* is an implemented function.

```
ENTITY A;
   att_1: INTEGER;
WHERE
 SELF.att_1 > 5
END_ENTITY;
```

```
RULE Control FOR A;
WHERE
PLUS_FUNCTION(QUERY(inst<*
                A)) < 1000;
END_RULE;
```

Figure 9. Constraints in EXPRESS

As the meta-class concept does not exist in EXPRESS we use a meta-programming (see (Y Ait-Ameur, Pierra, & Sardet, 1995) (Y Ait-Ameur, Besnard, Girard, Pierra, & Potier, 1995) for details) technique. It is the process that allows us to represent data and/or programs by data in a meta-model. In our proposal this technique has been used to represent procedural knowledge (expressions).

II.5.2. Expressions with EXPRESS

In our approach the problem of representing procedural knowledge is solved by considering programs or procedures as data. Thus, we can represent expressions like in functional languages. An expression is modeled to be either a constant (literal), a variable, an unary, a binary or a multiple arity expression as illustrated in Figure 10.

```
SCHEMA generic_expressions_schema;
ENTITY generic_expression
  ABSTRACT SUPERTYPE OF(ONEOF(
   generic_literal,
   generic_variable,
   unary_generic_expression,
   binary_generic_expression,
   multiple_arity_generic_expression));
END_ENTITY;
```

Figure 10. Expressions top entity

In our work we have extended and interpreted these expressions for allowing the validation of inter-model constraints (see details in section V.5). We use EXPRESS local rules to trigger the validation. This cause the interpretation of expressions via particular derived attributes belonging to the different involved entities (an expression is encoded by a tree of entities). To illustrate with an example, in Figure 11 the comparison between two elements is triggered by the local rule *WR1* which implies the calculation of the derived attribute *THE_VALUE*. This derived attribute calls the function *COMPARISON_GREATER_FCT* which fulfils the rule by returning *true* whether the comparison is correct.

```
ENTITY COMPARISON_GREATER
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
   SELF\BOOLEAN_EXPRESSION.THE_VALUE:BOOLEAN:=
                                 COMPARISON_GREATER_FCT(SELF);
WHERE
    WR1 : SELF\BOOLEAN_EXPRESSION.THE_VALUE = TRUE ;
END_ENTITY;
```

Figure 11. Interpretation of an expression using EXPRESS

II.5.3. The choice of EXPRESS

We have chosen EXPRESS modeling language for the validation of our approach mainly because:

- it is a language allowing both the construction of models and the validation of constraints in a homogeneous formalism
- its object-oriented philosophy and the multiple inheritance capabilities fitted the nature of the meta-models we have built
- from the perspective of knowledge, the formal semantics of EXPRESS language permit the implementation of simple knowledge models (i.e. focused on classes and without reasoning)
- it exists tools providing environments to validate constraints over instances of EXPRESS models

Thus, these characteristics allowed us to perform a rapid validation of the different concepts of the approach.

II.6. Conclusion

In Chapter I we have treated the topic of heterogeneous models and integration issues. In the current chapter we have focused on the necessity of make explicit the implicit knowledge is necessary to correctly integrate and validate engineering models. Therefore we have presented different aspects of the knowledge modeling denoting the need of ontologies in our Systems Engineering context. We have discussed the current efforts for bringing together models and ontologies. We think there is a need of research concerning non-intrusive approaches which defend the formalization of implicit knowledge to integrate engineering heterogeneous models in order to validate inter-model constraints.

Our objective is to describe, model and verify inter-model constraints and relationships between existing heterogeneous models by making explicit, formalizing and exploiting additional knowledge usually not expressed by the engineers to express these constraints and relationships. Our work focuses on same level inter-model relations in the Aircraft Systems Engineering development process (Simon Zayas, Monceaux, & Ait-Ameur, 2011). Thus, we have to take into consideration our aeronautical industrial context, which is described in next chapter.

Chapter III Current practices in Aircraft Systems Engineering

Summary

III.1. Introduction 37
III.2. Aircraft Systems Modeling 37
III.3. Current MBSE applications 39
III.4. From documents to models 40
III.5. MBSE and development process 41
III.6. Management of heterogeneous modeling in Aircraft Systems Engineering 41
III.7. Expected benefits of the proposed approach 42
III.8. Conclusion 44

Abstract. The adaptation of a solution proposal for the management of heterogeneous models relies on its industrial context. Therefore, each industry has its own applied methods and practices. In this chapter we describe our aeronautical industrial context and we discuss the expected benefits of our approach.

III.1. Introduction

The particularity of aeronautics domain concerning the heterogeneity is due to the complexity of the system itself, the aircraft, but also to the complex organization. On the one hand, there is a great number of internal departments and teams involved in the design of an aircraft. On the other hand more and more suppliers are collaborating in such design. Thus, the applied approaches and the different ways of work increase the collaboration issues. Moreover, this collaboration is necessary all along the lifecycle of the aircraft which is very long. These factors result in interoperability problems (Figay, 2009) and in modeling variability and heterogeneity. Consequently, generic and multi-view methods are needed, e.g. (Tenorio, Mavris, Garcia, & Armstrong, 2008).

Even though models have been used for long in aeronautics domain, their complete integration in a Model-Based Systems Engineering (MBSE) approach is not yet fully accomplished. In detailed design, i.e. the design closer to the physics of the aircraft where the semantics of the modeling languages are clear and specific, models are historically well managed. Nevertheless the best practices for using models in higher levels, operational or functional, are still an open discussion and MBSE is seen as the perfect framework in order to find an overall solution.

III.2. Aircraft Systems Modeling

Common definitions of a system involve end and enabling products (see I.3), processes and people as main elements. According to this definition one can consider an aircraft as a system. Nevertheless, in aircraft engineering the term *system* does not fit completely this description. Historically, an aircraft was built from a set of systems (one of the domains of the product breakdown) called embedded systems each corresponding to almost exclusively one dedicated calculator usually in charge of one function. This is the reason why quite often the notions of function and system were confused.

Progressively, the architectures have evolved to commercial microprocessor-based architectures, that meaning more powerful processors able to perform several functions but also a stronger dependence of the market products. As a consequence, the separation between functional architecture and physical architecture has become essential. Therefore, one function is provided by an application, which is composed of one or more software programs that are loaded, with other programs, on cards themselves installed on various equipments on-board, but also on-ground for some applications.

Aircraft domain has advanced from describing the aircraft as a mechatronic system, i.e. a set of mechanical, electronical and computer components interacting, to considering it more and more as an internal element of the information system. The information system is the group of domain objects, messages (information), data and business rules used or implemented in order the aircraft to be operated during its life cycle by actors involved in different specialties. The information system widely exceeds the aircraft system since it includes fleet management and the stakeholders implied in the external interactions of an aircraft. Amongst the sub-systems of the information system, the computer system is composed of the electronical and computer means and the telecommunication elements allowing automating and supporting the operations. Therefore, the computer system is the structured collection of software and hardware components and data enabling the almost total automation of the information system. It includes both on-board and on-ground elements as well as the communication means.

The progression of aircrafts from mechatronic systems to information systems entails a more complex management of requirements as well. Historically requirements have been managed as a set of documents hierarchically organized. Hence, in order to keep traceability, documents were used not only for eliciting requirements but also to carry out the design. In this way, the cycle beginning with the requirements and finishing with the construction of the aircraft was document-based. Unfortunately, documents are very difficult to handle for such complex systems because of their textual nature, the different interpretations of the content and the exchanging information problems. In this context, models, already widely used in computer science, seemed the logical evolution, particularly taking into consideration the growing impact of the information system in the aircraft. Nevertheless, the use of models is not new in aircraft domain since they were already necessary to perform simulations of physical laws or to represent 3D data at lowest levels (CAD tools). The actual need was to fill the gap between the requirements and these detailed models and to set-up processes which formalize the use of models at different levels of the design and all along the aircraft lifecycle, the aim of the Model-Based Systems Engineering techniques (seen in I.6.1).

In aircraft domain, models are used to build a virtual representation of the aircraft starting with a description of the operations linked to the top level requirements. The operations are then supported by functions that are also described in functional models. When a function is really complex, e.g. to perform maintenance, an entire model can be reserved to represent it but it is not necessarily limited to a unique embedded system. Actually, a model describes often the interactions between various systems. Functional models are supported by one physical architecture, after performing trade-off comparisons between other candidate architectures. This physical architecture is also described by one or several models until arriving to a level where traditional engineering models (CAD and logical) take part. Obviously, model-based design it is not a single execution of the chain operation-function-

architecture but an iterative activity which details the design progressively in a top-down perspective, in order to use the different models to build the aircraft in a bottom-up approach.

III.3. Current MBSE applications

The industrialization of MBSE principles is a difficult task which has to be carried out in progressive steps. Therefore, in our aeronautics context the MBSE approach has been addressed in different ways throughout the time.

Requirements engineering. As a previous but essential step to a correct MBSE implementation the management of requirements is considered a focal point. Currently a Requirements-based Systems Engineering method exists and is industrialized. Rules and techniques are defined to write requirements and to handle their traceability. Thus, aeronautics industry defends Requirements-Based Systems Engineering as the starting point of large scale SE applications.

Models with a local perspective. In order to improve the modeling skills of engineers it is a good practice to introduce modeling techniques in a progressive way. Hence, in our industrial context some models have been developed aiming at very precise objectives. They are models that follow MBSE recommendations but there are not developed within a process and evolution perspective. The aim of these modeling activities is two-fold: on the one hand models are used to verify some particular properties (specification validation, executable specifications, impact analysis…); on the other hand it allows engineers to learn the foundations and benefits of applying MBSE approaches.

MBSE as a process. Next logical step in a MBSE deployment tactic is to evolve from current modeling practices, which are varied and sometimes ad-hoc, into more consistent MBSE methods. This consistency means that:

- modeling development process has to be organized and clearly described
- most suitable models and modeling languages need to be recommended for each of the life cycle stages
- best practices and recommendations for each modeling language must be available
- relationships between models have to be structured.

As the development of an aircraft is a sensitive activity, these methods have firstly been applied to a research context in order to reduce risks and to get a satisfying maturity status. As a result, current MBSE methods have demonstrated efficiency enough to exceed the research boundaries and be implemented in new programs.

These different but complementary experiences give now the necessary background to contemplate the possibility of applying the MBSE approach to the entire development cycle

of new aircrafts. The current challenge is to study actual development processes in order to correctly incorporate the MBSE.

III.4. From documents to models

Traditionally modeling was an activity involving a piece of paper to illustrate the design diagrams and a huge quantity of documents to describe the system. Progressively computers have facilitated the design tasks by digitalizing documents and by providing engineers with computer-aided design tools. Nevertheless, documents have been the historical central point in design.

Before the use of digital models, i.e. models developed by computer, design documents contained text and formal descriptions. This content, due to its nature, is error-prone, ambiguous and difficult to be re-used. Models try to overcome all these problems by granting a more consistent design. As described previously, Requirements-Based Engineering (RBE) has been the first SE domain to be addressed from a methodical point of view. So, as shown in Figure 12, traditional documents are organized and structured in order to correctly manage requirements.

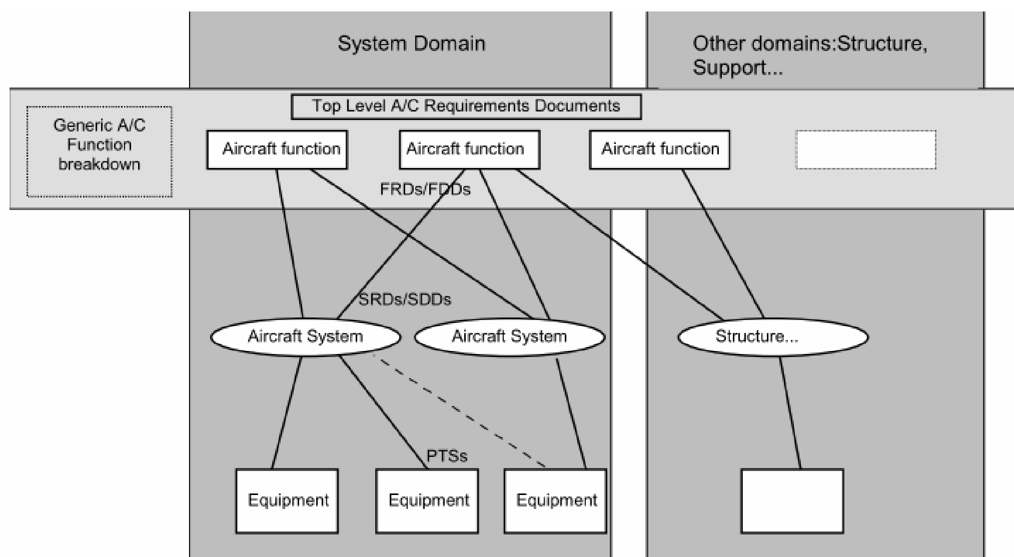


Figure 12. Airbus RBE process

In order to evolve RBE methods towards integration with models, documents incorporate the modeling diagrams. This may be a manual process or an automatic document generation task depending on the modeling tool solution. Nevertheless in most cases these documents need additional information to complete and to understand the design. For instance, as SysML modeling language has not a unique way of building a functional architecture, engineers are required to select the most representative diagrams to provide with this architecture point of view. This kind of tasks are costly since most of this information is already described in the

models but needs to be reworked in order to have a consistent design. Thus, MBSE has a lot of interest for the design documentation and for a complete profit of models in order not to rerun design activities inefficiently.

III.5. MBSE and development process

In order to have a successful establishment in the development of new aircrafts, MBSE principles need to take into consideration the current development process. That is, a process including different levels of design and relationships between models. As described in section I.4.2, two categories of inter-model relationships are identified in modeling process, each one presenting different difficulties from the MBSE point of view.

Same level relationship. In this case models to be managed have the same level of detail. From the MBSE perspective, boundaries and overlaps of models are the main issues. Furthermore, the links between models are not clear since even though MBSE recommends using interfaces as model joint points, it is not always possible due to the way work is sometimes shared in the collaboration engineering frame. Therefore, in some situations big models are divided and distributed amongst several engineering teams following criteria different from interfaces approach, e.g. chief engineer may decide to decompose a model according to domains of interest (maintenance, flight…).

Top-down or bottom-up relationship. In top-down or bottom-up relationships the connection between models of one level N to the ones of level N+1 is better formalized since they follow the logical design evolution, i.e. models of level N+1 detail those of level N. Nevertheless, in such cases the difficulty often arises from the different modeling languages used and from not having a homogeneous way of declaring these top-down or bottom-up relationships. For instance, traceability between SysML high-level models and Scade (E. Technologies, 2011a) detailed models (in the context of cockpit display code generation) is not currently implemented although solutions as Scade System Designer (E. Technologies, 2011b) try to address the problem.

III.6. Management of heterogeneous modeling in Aircraft Systems Engineering

Currently, there is no formal approach to tackle heterogeneity in models in our industrial context. Nevertheless, some efforts have been carried out to face this problem.

Concerning **same level relationships**, meetings between engineering teams are the most common way of work. Engineering teams put in common their models and, manually, they identify common elements and denote the inconsistencies that must be managed. The results

of these meetings are documents (usually Microsoft Excel files) with a list of inconsistencies solved after a lot of research and reanalyze work. In order to improve this kind of meetings some additional resources are added to the modeling activity, mainly pre-formatted documents containing information that allows engineers to have a more homogeneous understanding of the models (model architecture, internal and external interfaces and so on). Nonetheless, as described in section III.4, these documents imply additional costs as well.

In the **top-down or bottom-up relationships** context, the traceability of requirements is quite well mastered. Currently, requirements management tools as DOORS (IBM, 2011b) are correctly connected to the modeling tools in order to get a good traceability. This is a solution for the first stages of the development cycle which focus on top level requirements; however there is not such a consensus for the implementation of top-down or bottom-up relationships in more detailed design. An industrial research axis for solving this aspect is related to the definition of common meta-models. Such meta-models aim at being shared by the different modeling tools and being the central point to handle with the inter-model relationships. Nevertheless, taking into account the variety and quantity of modeling languages, this sort of solutions are basically applied in limited scenarios and not for the entire development cycle.

The whole development cycle is actually the core of MBSE methods. Aeronautical industry has used and developed different MBSE approaches, most of them closely related to particular modeling languages. That is the case of IBM Harmony (IBM, 2011a) and OOSEM (Lykins & Ave, 1999) for SysML or CORE System Definition Guide for CORE language. These are solutions that try to cover the entire development cycle but industry experiences have shown that the use of a unique modeling language is not a realistic approach. Thus, industrials have improved this point of view by introducing different solutions, sometimes in the form of ad-hoc proposals for a particular context but also by proposing more formal methods as AMISA (AIRBUS, 2008) which are applied to various modeling languages. Such methods solve part of the problem but still there are some lacks concerning: 1) the heterogeneity management of existing models; 2) the simultaneous use of different modeling languages; 3) the management of implicit knowledge. These missing areas are important assets for the future deployment of our approach.

III.7. Expected benefits of the proposed approach

In next chapter we describe a method which allows the expression and validation of constraints over inter-model relations. Our idea (Simon Zayas, Monceaux, & Ait-Ameur, 2010) is based on the use of knowledge models to make explicit the engineers' implicit knowledge and on the preservation of the original models by means of meta-modeling techniques. Concerning the latter, the key issue is to work in a shared framework where

source models are exported in order to be aligned in the same universe. Below the expected benefits of the proposed approach in our aeronautical context are described.

Model consistency. Consistency of models is improved thanks to the formalization of explicit knowledge which can be in that way shared and managed. Formalization of knowledge is a support not only to get agreements concerning concepts of the domain, i.e. aeronautics, but also concerning Systems Engineering modeling concepts. Due to the native heterogeneity of modeling languages there are different possibilities of representation of equivalent modeling entities. For example, *function*, a key concept in Systems Engineering, is represented in CORE by an entity called *Function* whereas in languages with open semantics as SysML a function can be a *Block*, an *Operation* or a *State* depending on the specific domain modeling rules applied. Thus, amongst the knowledge models that can be used with our approach, one describing such modeling concepts will allow engineers to improve their modeling capabilities.

Model relationships. We have analyzed previously the difficulties for establishing both same level and top-down or bottom-up relationships. Our approach includes a relation meta-model. Such a model is an advantage of the industrialization of our method since it will provide engineers with a formal representation of inter-model relationships. Thus, a relation meta-model will be enriched in order to include complex relationships as redundancy for same level cases or composition for top-down or bottom-up scenarios.

Model reuse. The black-box annotation, i.e. the annotation of the models without analyzing their content from the user point of view, is considered in our approach. This feature in combination with a repository of models will allow engineers to perform requests over previous models in order to ease their reuse when developing a new aircraft program. Naturally, the black-box annotations have to be completed by the inner-model annotations and knowledge concerning modeling concepts to give the necessary background in order to guarantee the correct reuse of models.

Non-modeling tasks. Currently meetings are organized to validate the consistency between models developed by different teams. Even though it will not definitely prevent those meetings, the use of explicit knowledge made by our approach will help to reduce the number of issues to treat, e.g. questions found in actual documents such as "*Which bypass valves are we referring to? (Cockpit or Humidifiers)*" will be answered by ontologies and not considered an issue anymore. At the same time, annotation of models will add information that is currently contained in textual documents (e.g. description of model properties as objective of the model, simulation type, author…) and it will help to generate technical documents. Therefore, the number of documents will be shortened and, consequently, the global time devoted to the creation of documents.

To sum up, optimization of the modeling activities, increase of the quality of the design and improvement in the communication between engineering teams are expected as the main benefits of industrializing our approach.

III.8. Conclusion

Modeling in aircraft industry is very complex from a point of view of organization and methods due to the complexity of the system but also to large structures and multiple suppliers. These difficulties have historically been addressed by using strict documentation rules and, more recently, by starting the application of MBSE principles. Since MBSE in aircraft industry is still evolving, new methods and tools are necessary to manage models. In such context, next chapter presents an approach which focuses on the formalization of implicit knowledge to integrate heterogeneous models and to perform inter-model validations over them. The main expected benefits of the proposed approach enclose improvements in model consistency, in formalization of inter-model relationships, in model reuse and in design time efficiency.

Chapter IV Knowledge-based inter-model constraint verification

Summary

IV.1. Introduction 47
IV.2. The proposed General integrated models representation 49
IV.3. Manipulated models 51
IV.3.1. Source models 51
IV.3.2. Exported models 52
IV.3.3. Annotated models 52
IV.3.4. Integrated model 52
IV.3.5. Constraint Relational model 53
IV.4. The resources 53
IV.4.1. Source Meta-models 53
IV.4.2. Knowledge models 53
IV.4.3. Constraint Relational meta-models 53
IV.5. The modeling process activities 54
IV.5.1. Export 54
IV.5.2. Annotation 58
IV.5.3. Model integration 61
IV.5.4. General constraint definition 63
IV.6. Conclusion 67

Abstract. The need to manage the complexity of current systems encourages the use of abstract models in Engineering processes. Nowadays, dividing the work and the maturity of collaborative engineering techniques require combination of heterogeneous models in order to achieve the overall engineering process. In such a context, we propose a method making possible to interoperate existing heterogeneous functional and structural models. Our approach is knowledge-based in order to annotate and make the models interoperate.

IV.1. Introduction

Engineers have a very clear understanding of the internal structures of the models they develop. Nevertheless, in the case their activities involve establishing and formalizing links between elements of several models (classes or data), they require assistance for handling inter-model relationships. We present an approach giving such kind of support when establishing *same level* inter-model relations in order to check constraints over heterogeneous functional and structural models. Our idea is based on two main ideas: 1) the preservation of the original models by means of meta-modeling techniques and 2) the use of explicit knowledge by means of ontologies.

Nowadays we know how to write constraints for one single model since, basically, either they are part of the modeling language itself, i.e. semantics of language, or because an additional language is provided to add more specific rules, e.g. OCL in UML. Nevertheless, the context of our research involves more than one model usually expressed in different modeling languages and the expression of inter-model constraints in such circumstances needs a different approach.

In next sections we develop our approach using an example which involves two structural and functional models, the Cockpit Information System (CIS) and the Shared Information System (SIS). These two models use different modeling languages: *CIS* is a model expressed in SysML representing the management of cockpit messages; *SIS* is a model designed using the CORE modeling language and whose objective is to describe the treatment of maintenance messages received from *CIS*. At the end, *CIS* belongs to high-level security domain (*ClosedWorld*) whereas *SIS* is a medium-level security system (*OpenWorld*).

On one side a *Physical Block Diagram*, see Figure 13, describes the internal component (*Maintenance application*) which transfers maintenance messages (*Items*) through a *Link* (*extcomm*). On the other side a *BDD* diagram shown in Figure 14 represents the subsystem (*CIS*) generating maintenance messages that are sent to SIS (*NCSystem*). This communication is performed through a link (*ExtPort*) according to an interface (*ExternalCommunication*).

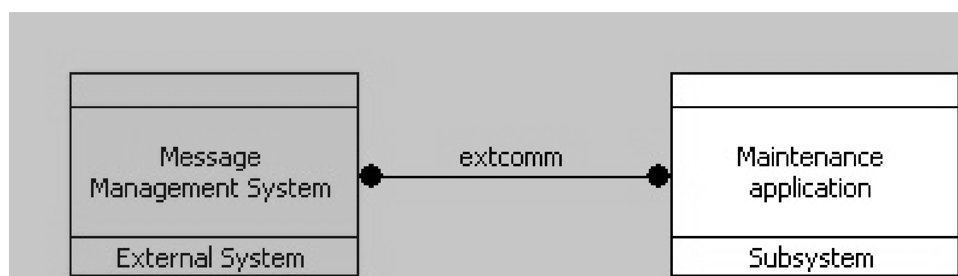


Figure 13. Physical Block Diagram representing the communications from a subsystem to an external system.

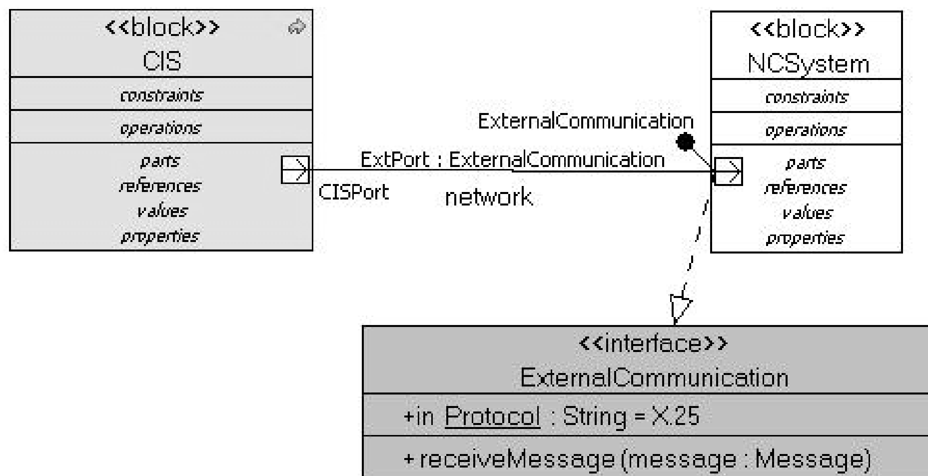


Figure 14. Block Definition Diagram showing external interfaces of CIS model.

When composing these two models together the constraint below must be checked:

"All messages from ClosedWorld to OpenWorld shall use a secure communication protocol"

As we have mentioned previously, checking such a constraint requires to address two problems.

1) Expressing a constraint over two models

This is the first problem we need to tackle. Models are described using different model practices and semantics. Moreover, they are also based on different modeling languages, in our case SysML and CORE. The proposal to overcome this difficulty is to export both models in a unique and shared modeling language. As a consequence it becomes possible to apply MDE techniques when exporting the original models into a common framework. For instance, SysML and CORE meta-models are written in such a unified language and then *SIS* and *CIS* models are expressed as instances of these meta-models in the shared framework according to the MDE principles. Thus, once the models are described in the common framework, the constraint can be expressed by referencing elements of both models since these models share the same modeling language. The heterogeneity due to the nature of models and modeling languages is reduced. Nevertheless, this action is not sufficient to allow the designer to express this constraint.

2) Using implicit knowledge

The second problem concerns the semantics carried by the concepts. Each model is developed in a particular technical domain with a particular team of engineers. In this context,

hidden knowledge shared by the team and is kept in engineers' mind, i.e. it is not made explicit during the modeling process. Therefore, models are understandable by the team in charge of the design only. So, some lacks of comprehension may arise when the model is shared or combined with other ones. In the constraint of our example we find some concepts belonging to this hidden knowledge: 1) the concepts of *OpenWorld* and *ClosedWorld* that should be added to each model; 2) the concept of *message*, not represented in the same way in both models; 3) the concept of *security* of *protocol,* an information that must enrich the *protocols* described in the models. Thus, whether we want to validate this inter-model constraint we have to make explicit such a hidden knowledge. Consequently, we suggest formalizing this knowledge by the means of explicit aside knowledge models: ontologies. The knowledge models and their instances represent the concepts that we need to make explicit, e.g. domain, messages and communication protocols. Finally, these instances that form the domain knowledge base, enable the annotation of the original models expressed in the common framework and ergo, the complete representation of the inter-model constraint.

Next sections take these two main ideas and develop our approach in details.

IV.2. The proposed General integrated models representation

Our approach is a four steps method that lies on the definition of models in a shared modeling language and on ontologies for encoding explicit knowledge bases. The idea of the method is to export the elements of the models we want to work with to a unified and shared modeling language, which handles the meta-models of the different original modeling language elements. Secondly, the exported elements are annotated by explicit knowledge concepts borrowed from the explicit knowledge base, i.e. domain ontologies. Then, all this information is taken into consideration to set up the inter-model relations. Finally, inter-model constraints are formalized and checked over the annotated models.

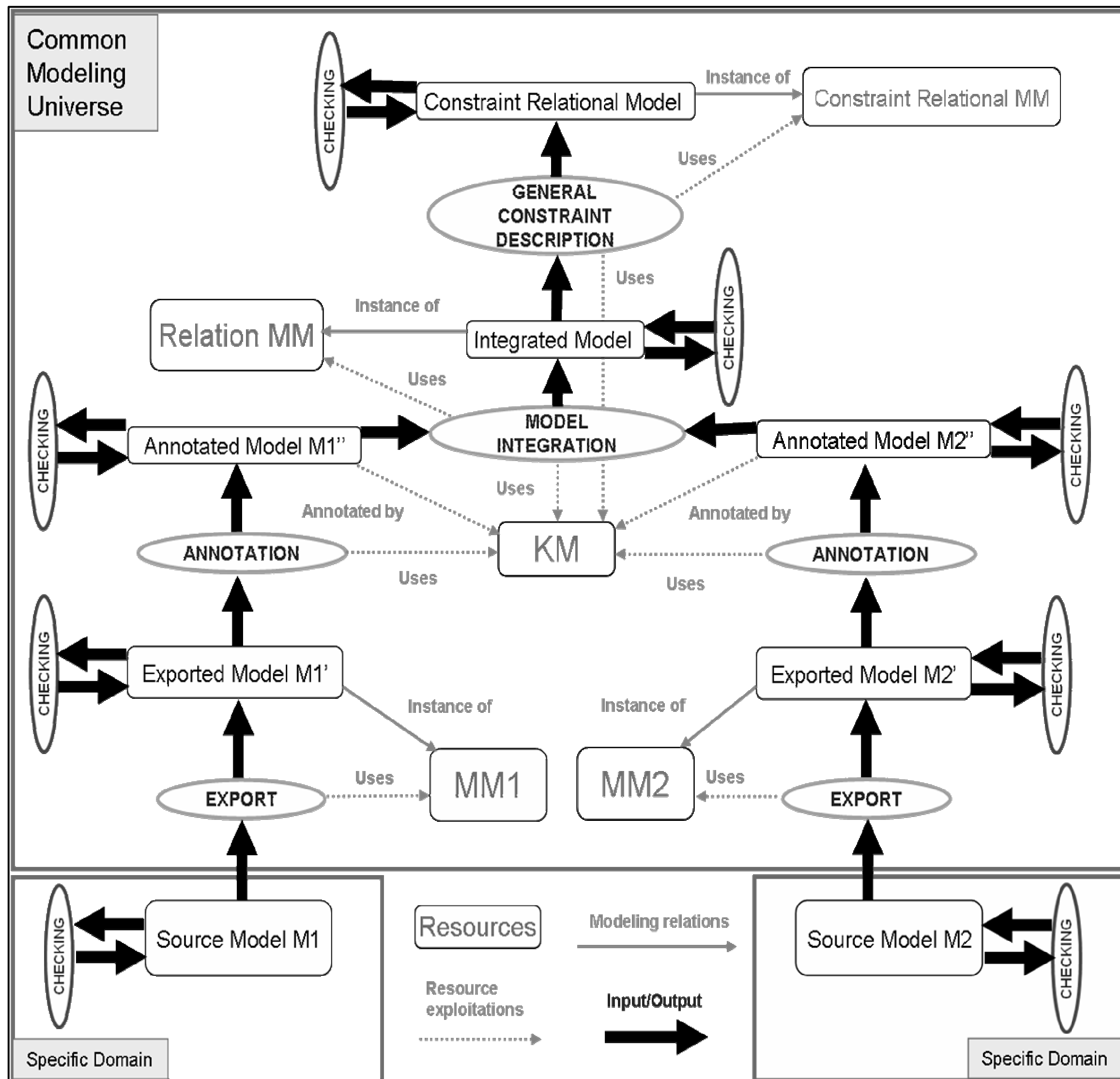


Figure 15. Method to validate inter-model constraints based on knowledge models.

Our goal is to apply this approach to support a Systems Engineering methodology, specifically when the engineer designing a system needs to guarantee the correctness of models before switching from a development step to the next one, e.g. from "*Requirement Analysis*" step to "*Detailed System Architecture*" step. This approach aims at strengthening the cross model verification and validation activities. During these checkpoints, also known as maturity gates, the different models, resulting of the concurrent engineering activities, should be put together to verify the consistency of the design before continuing the modeling process.

The approach consists of a top-down activity to analyze and describe the inter-model constraint that needs to be checked and of a bottom-up process to check the analyzed constraint based on the proposed model illustrated in Figure 15. Next sections develop the elements and the activities contributing to this model.

IV.3. Manipulated models

According to the Figure 15 a set of models or evolutions of models are manipulated throughout the development of the approach. From the source models to the constrained integrated model, the identified methodological steps of the process are followed in order to be able to check inter-model constraints.

Even though our method is presented with an example involving only two models, the approach is multi-model, i.e. its principles are valuable for one, two or more source models.

IV.3.1. Source models

Systems Engineering models are used all along the development process of a given system: an aircraft for example. These models are managed by different engineering teams and are constructed using several modeling languages and tools. They are the input of our method, bottom of Figure 15, and therefore we call them source models, i.e. the models developed by engineers applying their own methodologies and best practices.

We focus on descriptive models according to the first stages of the V-cycle development process. During this development process different levels of details and various engineering domains are involved as shown in Figure 16. In the context of aircraft design, in the architecture stage where requirements, operations and functions are described at a high level of detail, we find modeling languages like SysML and CORE. Nevertheless, when a more detailed definition is necessary, i.e. at the subsystems level, other languages like MATLAB/Simulink, or Scade are more appropriated. Moreover, subsystems usually imply the collaboration between different technical domains with the corresponding modeling practices. These practices depend on the domain background and heterogeneity will arise even when design models are provided in the same modeling language.

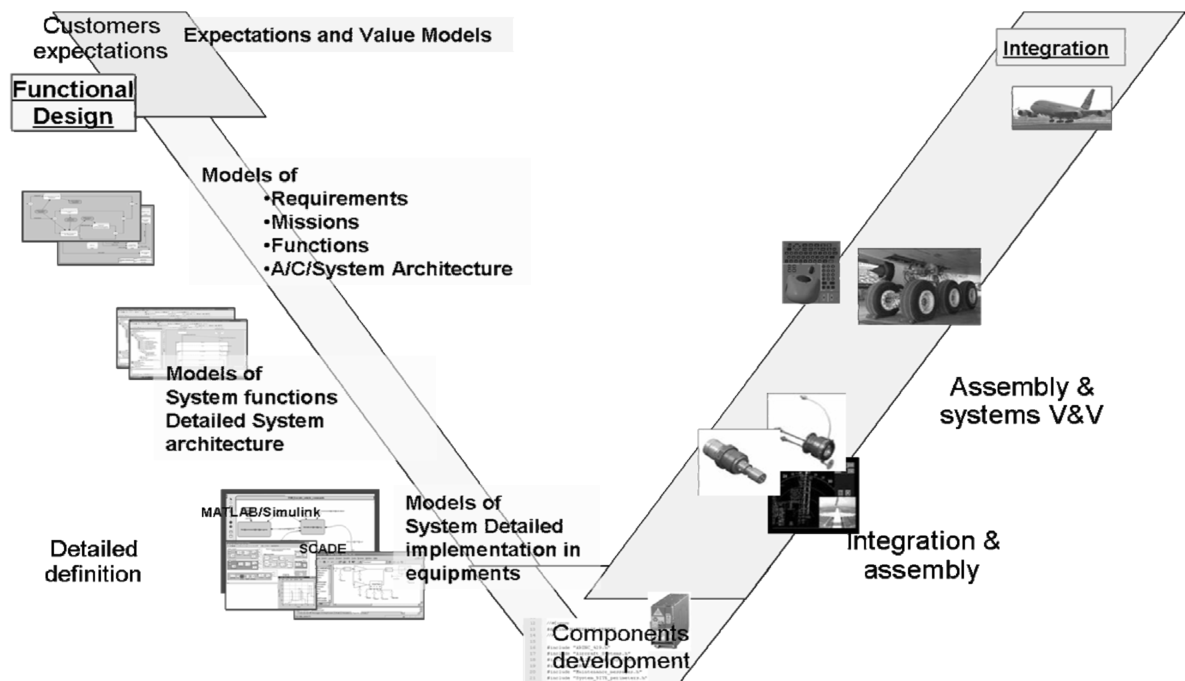


Figure 16. Models in the Aircraft Development V-Cycle.

IV.3.2. Exported models

The exported models (*M1', M2'*), see Figure 15, are the result of exporting the source models (*M1, M2*) into a shared and common modeling language. Thus, the meta-models (see IV.4.1) of the source models are written using the same modeling language and the source models can be exported, as instances of these meta-models, to a common framework. The exported models can be either the whole source models or part of them; a projection. Actually for some inter-model constraint verifications it is not necessary to take into consideration the entire source models but only some particular parts of them, e.g. if we need to check the consistency of messages using a particular interface we do not need to export the whole interfaces.

IV.3.3. Annotated models

The exported models are enriched or constrained by explicit knowledge concepts borrowed from domain ontologies. This process of linking exported models to concepts of the knowledge models (see IV.4.2) is called annotation. Thus, knowledge models are used to provide the implicit knowledge and the output of the annotation process define the annotated models (*M1'', M2''*) of Figure 15.

IV.3.4. Integrated model

After annotating the exported models, the annotated versions (*M1'', M2''*) are integrated into a new model whose objective is to formalize inter-model connections. This is the role of the integrated model in our approach (see Figure 15). The integrated model is an instance of

the relation meta-model (*Relation MM* in Figure 15) which describes different types of inter-model relations. These relations are used to build the integrated model in order to connect *M1''* et *M2''*.

IV.3.5. Constraint Relational model

The expression of inter-model constraints is carried out by the constraint relational model (see Figure 15). This model is an instance of the constraint relational meta-model (*Constraint Relational MM* in Figure 15) which represents the inter-model properties which need to be checked by exploiting the inter-model relations of the integrated model and the knowledge described by instances of the explicit knowledge models.

IV.4. The resources

The approach is also supported by a group of resources that are used at the different steps.

IV.4.1. Source Meta-models

These resources consist of the meta-models (*MM1* and *MM2* in Figure 15) of the different source models expressed in a common and shared modeling language. Thus, our approach does not consist of a common meta-model in order to map the source models issued from different modeling languages, as in (Tolvanen & Kelly, 2008) for instance, but to translate the original meta-models in a common and shared modeling language in order to work in a shared framework. The translation of a meta-model is a one-shot action, once a meta-model is incorporated to the framework we can export any model conforming to it.

IV.4.2. Knowledge models

A knowledge model (*KM* in Figure 15) illustrates the concepts of explicit domain knowledge necessary to understand and to complete or to constraint the source models in an inter-model relation perspective. Knowledge models are the central point of our proposal since they are used during the annotation of the exported models; as a support to integrate the annotated models; and furthermore to build richer inter-model constraints. Thus, inter-model constraints in our approach support the combination of both concepts coming from the knowledge base and elements of the annotated models.

IV.4.3. Constraint Relational meta-models

The constraint relational meta-model (*Constraint Relational MM* in Figure 15) is a general model for expressing constraints. It has to provide all the entities which are necessary to construct the constraints depending on the context of the problem. Thus, in the case studies that we have analyzed the constraint meta-model allows us to represent First Order Logic expressions.

IV.5. The modeling process activities

Manipulated models and resources are used all along the four steps of our approach, i.e. the modeling process activities. These activities are performed sequentially in order to enable the evaluation of inter-model constraints after a progressive integration as shown in Figure 15. Firstly source models are **exported** into a common framework; secondly the exported models are **annotated**; thirdly the annotated models are **integrated**; and finally the inter-model constraint is **described** over the integrated model. Export is an activity focused on modeling semantics whereas the remaining activities are focused on domain semantics (see section II.2). Next sections detail these activities.

IV.5.1. Export

Definition

In order to handle different models, the first difficulty is the variety of modeling languages we consider. Our recommendation is to work in a same modeling universe if we want to add knowledge and to connect heterogeneous models. We need a syntactical homogenization. Thus, taking into consideration the different origins of the source models we suggest the definition of a unified representation in order to work in the same modeling universe. Therefore, the source models can be exported (or imported from the point of view of the common framework) into a same universe when corresponding meta-models are formalized in the unified and shared modeling language.

In this case, the exportation process shall preserve the original modeling semantics of the source models in the shared modeling language. This process, not addressed here, is performed when designing the exportation procedure.

Method

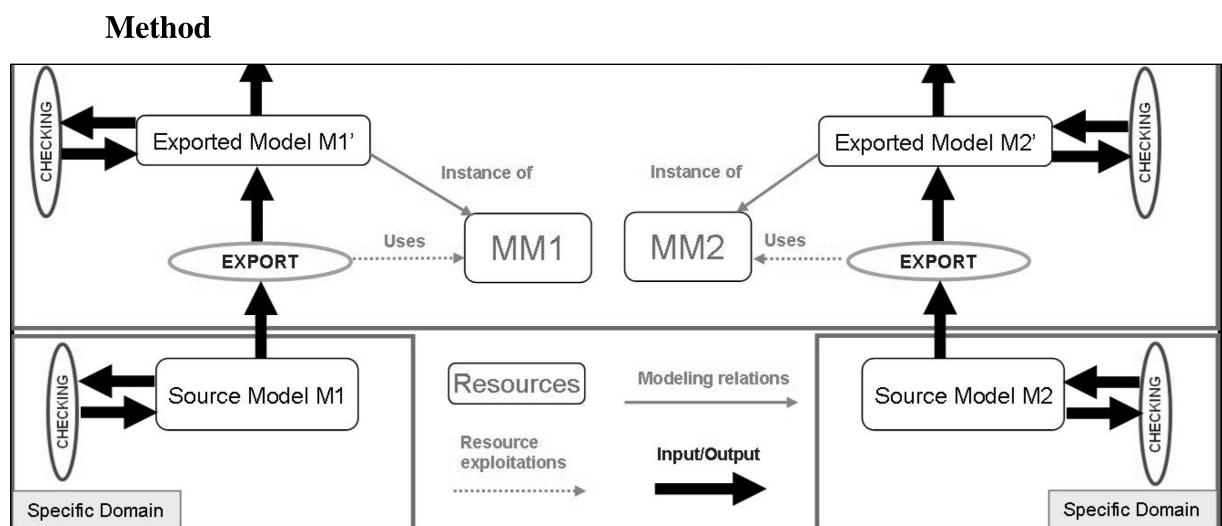


Figure 17. Focus on Export activity

Considering Figure 17 as a reference, the meta-models (*MMi*) of the different source models are written in the unified and shared modeling language. Each model *Mi* is exported as a model *Mi'*, instance of *MMi* in this modeling universe. Nevertheless, in order to allow the definition of the different meta-models we have previously defined a meta-meta-model. This approach fits with the concepts of OMG's MOF [OMG03] standard where 4 modeling levels can be distinguished.

- The information layer (MO), contains the data that one wishes to model. In our case this layer corresponds to the source models (*Mi*).

- In the model layer (M1), one adapts the meta-model to describe the data. It is the role of meta-models (*MMi*) in our approach.

- The meta-model layer (M2) defines the structure and constraints of the language used to describe the elements of the model: e.g. in UML we have *Classes* and *Attributes*. The meta-meta-models used in our approach to build the meta-models belong to this layer. Figure 18 illustrates this layer. *EntityClass* represents the basic element of models. One *EntityClass* can have multiple attributes represented by the *AttributeClass*. We can extend *AttributeClass* with the types considered as necessary. Figure 18 shows the most basic ones. One particular kind of attribute is the *EntityAttributeClass*, used to model the association relationship amongst *EntityClass*.

- Finally, the meta-meta-model layer (M3) contains the basic elements which handle the description of the modeling language. The set of these basic components represents the root modeling language, i.e. the shared modeling language in our method.

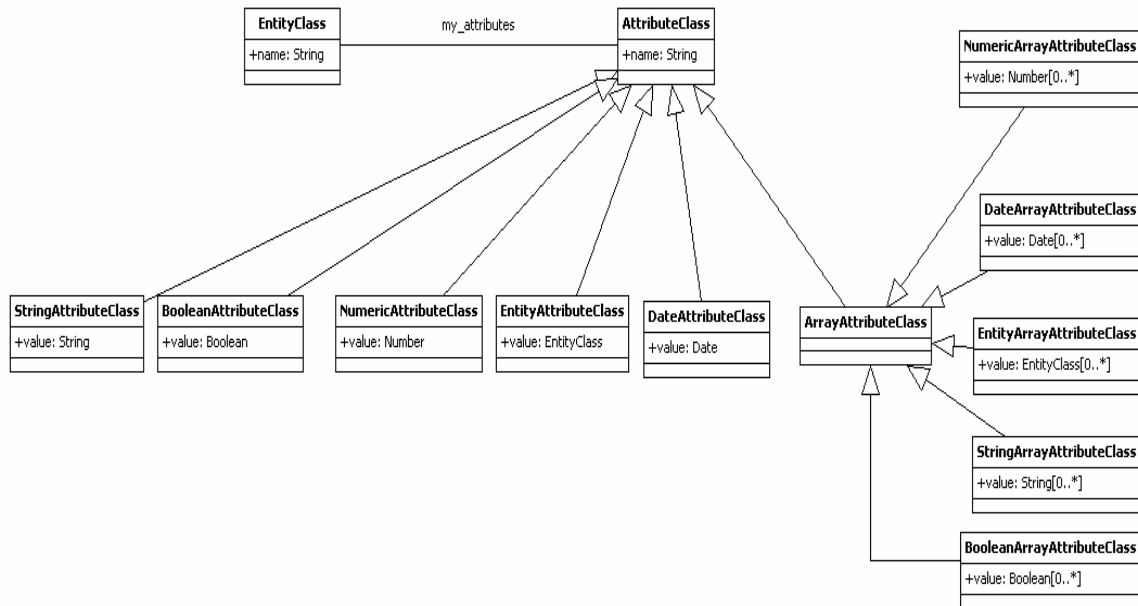


Figure 18. UML diagram of the meta-model layer of our approach

Checking of source models

Our hypothesis is that source models are locally correct, i.e. they are validated by the appropriated modeling tools in their *Specific Domain* environments. Nevertheless, during the exportation activity we check that source models can be exported, i.e. whether they respect the abstraction defined by the meta-models defined using the shared and common modeling language.

Example

In order to give an example of the exportation step and according to the models described in section IV.1, let us take the MOF framework again. We describe the actions implied in the exportation of the source models following the order of execution.

M3 layer

It contains the shared modeling language, necessary to describe the meta-meta-model classes, for instance UML.

M2 layer

In this layer, the shared modeling language is used to implement the classes shown in Figure 18 for the definition of the meta-models of source models: *EntityClass*, *AttributeClass*…

M1 layer

Considering our example, at this stage we need to build two meta-models using the elements of M2 layer: the SysML meta-model and the CORE meta-model. Therefore, in the CORE meta-model *Component*, *Link* and *Item* classes instantiate from *EntityClass* to form *ComponentClass*, *LinkClass* and *ItemClass* respectively. Their attributes instantiate *AttributeClass* according to their type, for instance the attribute type of the *ItemClass* is a *StringAttributeClass* called *ItemTypeAttributeClass* as you can see in Figure 19. On the other hand and similarly, for the SysML meta-model *EntityClass* is instantiated to build up the entities shown in Figure 14: *Block*, *Port*, *Interface* and their attributes are instances of *AttributeClass* as well. Thus, even though we have written two different meta-models, CORE and SysML, they have shared elements thanks to this layered approach.

M0 layer

Once all the previous models are completed the exportation can be carried out. Therefore, the content of the *SIS* model is exported as instances of the CORE meta-model, e.g. *extcomm* in Figure 13 is an instance of *LinkClass*, and the elements of the *CIS* model are exported as instances of the SysML meta-model, for instance *NCSystem* in Figure 14 would be an instance of *BlockClass*.

.

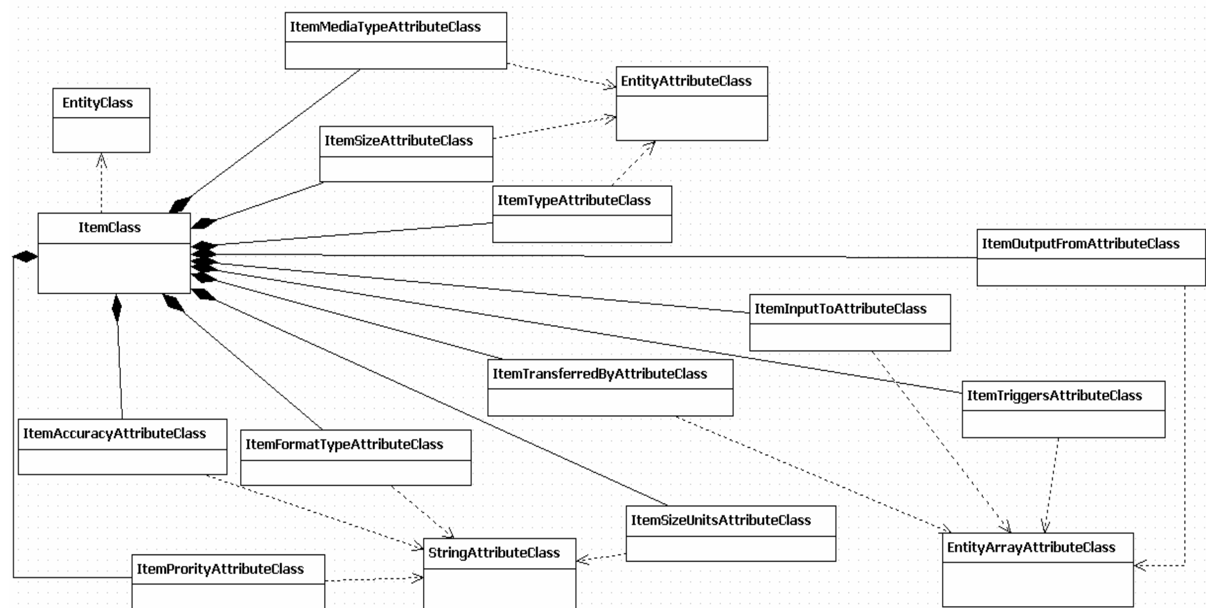


Figure 19. An excerpt of the CORE meta-model, focus on ItemClass.

An important point at this stage of the method is to note that source models are kept in their original design and that the rest of the process is performed over the exported version.

IV.5.2. Annotation

Definition

In this phase, the imported models are annotated, i.e. they are enriched or clarified thanks to the use of explicit knowledge introducing more domain semantics. As mentioned previously, models do not always contain all the knowledge of engineers. Our approach suggests enriching the descriptive models by explicit knowledge borrowed from aside knowledge models like ontologies. This enrichment is performed by annotation. The knowledge models formalize the missing information crucial to perform such inter-model relations and checking. Indeed, the use of such knowledge models offers a common reference mechanism to overcome terminology and modeling approaches differences originated from the source models.

Method

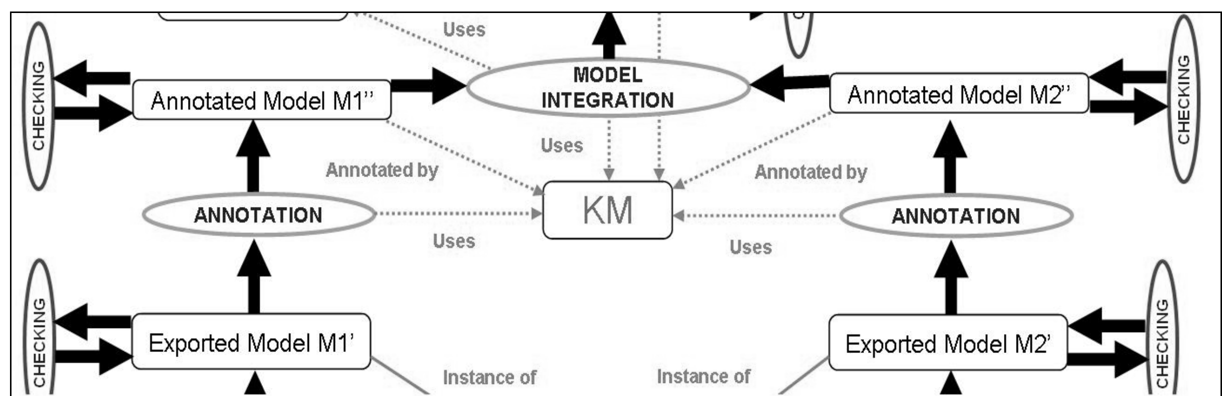


Figure 20. Focus on Annotation activity

At this stage of our method, see Figure 20, we put in relation (annotate) the exported models (*Mi*) with the knowledge models (*KM*) in order to harmonize the different modeling aspects. As a result we obtain the annotated models (*Mi''*).

In the annotation step there are two important components: the knowledge models and the annotation meta-meta-model. The former formalizes concepts of common domain knowledge agreed by engineers and can be developed outside the unified and shared modeling language. The latter is part of the models belonging to the approach and it is written using the same language of the meta-models. The only condition is that instances from the knowledge models may be uniquely identified. Thus, knowledge instances, i.e. the knowledge base, must be precisely distinguished by Uniform Resource Identifiers (URIs) in order to use them when annotating the imported elements. These URIs, shown in Figure 21, connect elements of the imported models to knowledge concepts via the annotation class. Therefore, *EntityClass* is connected to one or more pieces of knowledge by means of *AnnotationClass* whereas one

AnnotationClass links one or more URIs which is/are also modeled by a meta-class. The inverse relationship is of course also possible, i.e. one *KnowledgeClass* can be related to more than one *EntityClass*. This relatively simple annotation representation can be completed with some other attributes and properties (like Dublin Core (Hillmann, 2005) attributes for example) as suggested by some other work that addressed the problem of model annotations. Reference [ZOU08] suggests linking elements of the models to concepts of an ontology whereas [BOU08] describes a more complex framework with different ways and types of annotations depending on the kind of interoperability issue. These contributions analyze useful properties to be provided in the characterization of an annotation. Some of these properties are used to categorize the annotations according to different criteria (informal, formal, structural, behavioral…). Other properties give more precise information about the annotation, e.g. the accuracy of the annotation itself (exact, partial…) in order to report those cases when engineers are not able to find the exact knowledge concept to annotate a modeling entity and, instead, apply a similar or a possible knowledge concept.

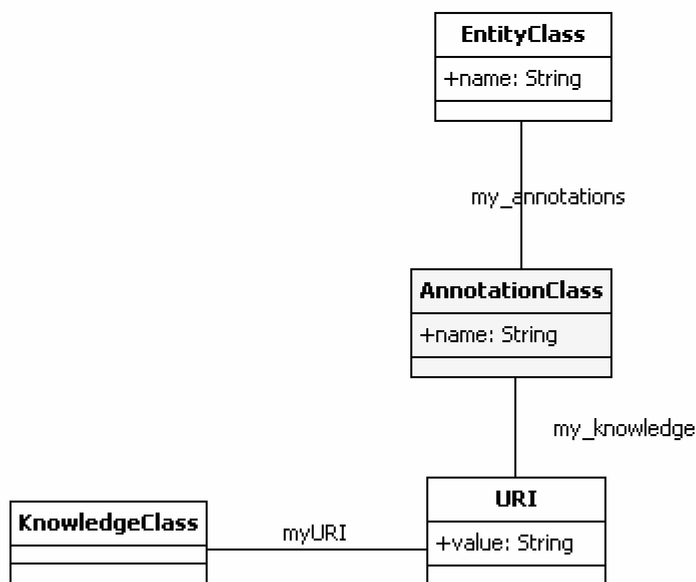


Figure 21. Annotation class.

Checking of annotated model

At this stage we can perform verification of the characteristics related to the added knowledge from the point of view of each individual annotated model (*Mi''*). For instance one can check that the communication protocols represented in a model belong to an agreed set of protocols.

Example

Continuing with the illustration, necessary knowledge not available in the sketched models is: 1) domain of the components, i.e. whether they are in a critical or a non-critical

domain; 2) the concept of *message* depending on the used modeling language; and 3) the name of the authorized *communication protocols* which are "*X.25*" and "*Encrypted Ethernet*". Part of this knowledge is summarized in the model of Figure 22.

Concerning the annotation, a short example illustrates the use of the annotation meta-meta-model. Figure 23 shows a graphical representation of the knowledge base, i.e. of the instances of the knowledge model. In this figure, one type of *message* is available, the "*Maintenance Message*", and 3 kinds of *communication protocols*: two secure protocols ("*EX25*" and "*Encrypted Ethernet*") and one non-secure (X25). During the annotation of the *Link* of the *SIS* model and in order to make explicit the type of protocol, *LinkClass* is connected to one *AnnotationClass* instance which points to one *communication protocol* ("*EX25*") of the knowledge base; we can see a detail of the related instances in Figure 24. This is a one-to-one annotation example, but in some other cases several entities might be annotated with one *AnnotationClass* instance. For example, if several *Items* in a CORE model compose altogether a message; whereas in a SysML model (see Figure 14), solely the *Parameter* of the *receiveMessage* operation of the *ExternalCommunication Interface* corresponds to this message. In such a case, the different *Item* instances will be connected to one *AnnotationClass* instance pointing to the message concept, to which the *Parameter* instance is also related.

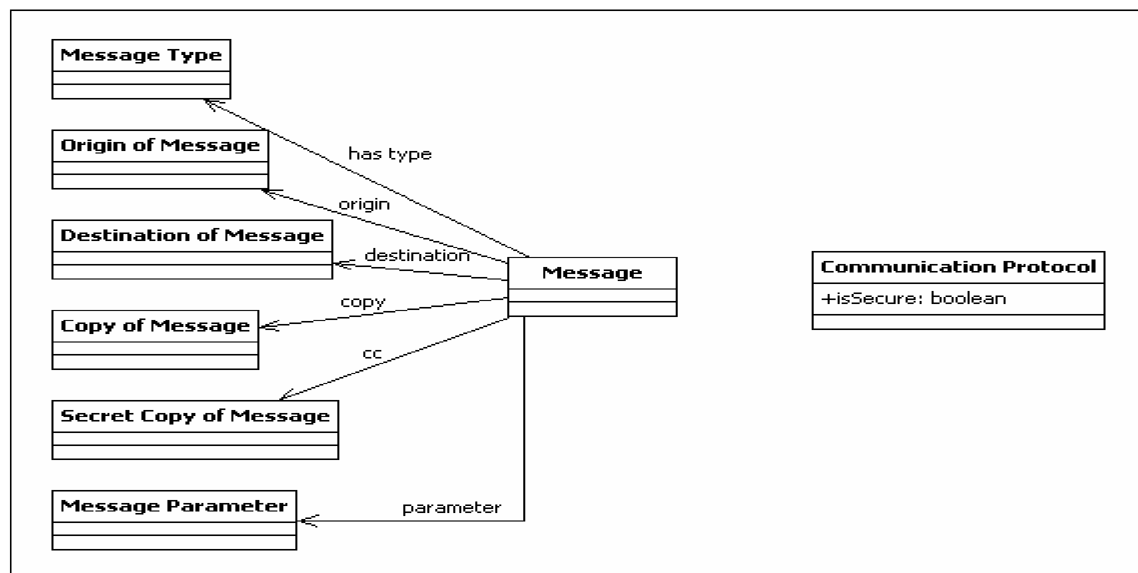


Figure 22. Knowledge model of messages and communication protocols

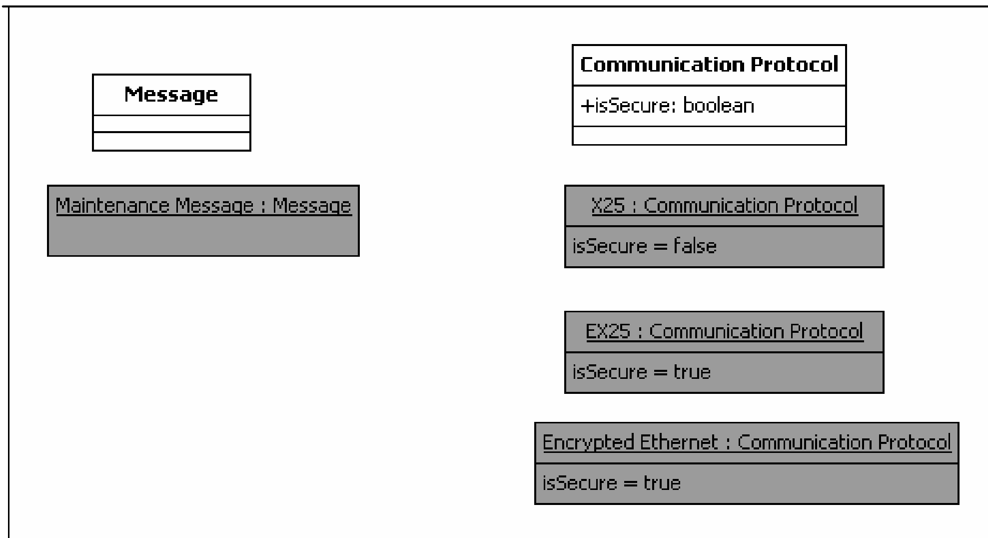


Figure 23. Knowledge base, instances of messages a communication protocols.

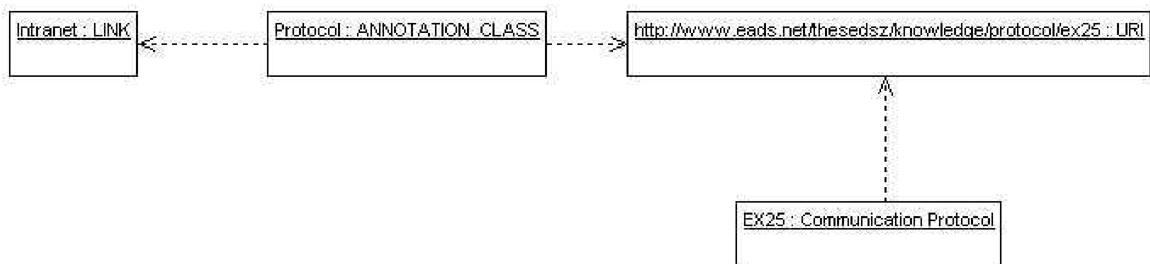


Figure 24. Instances of an annotation

IV.5.3. Model integration

Definition

Having the imported models annotated, we obtain the resources to describe relations between the models. These relations are necessary to correctly formulate the inter-model constraint and therefore validate it. Inter-model relations are the bridge between models and can involve several entities of the design materializing these links.

Method

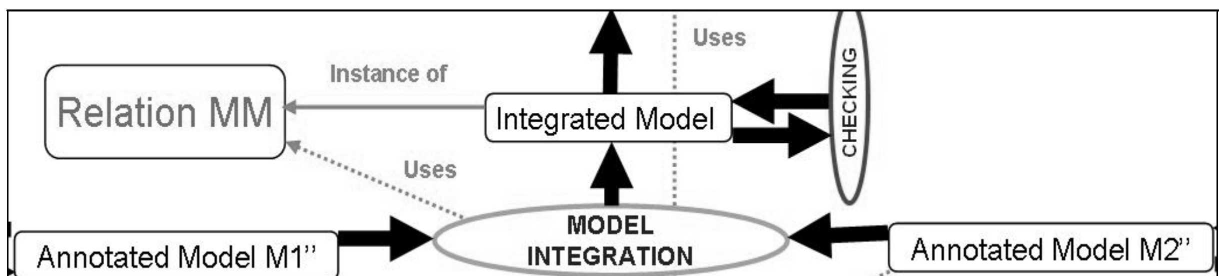


Figure 25. Focus on Model Integration activity

As illustrated in Figure 25, the inter-model connection activity takes the instances of the annotated models (*M1'', M2''*) as input and produces instances of the *Relation Model* as output.

At this step we are able to define our inter-model relations by instantiating a model of relations and by using the annotated imported elements. We have developed a first version of model of relations. This model will contain and formalize the different types of relations concerning elements of the design models: composition, equivalence, interface, trigger, etc. Figure 26 shows the inter-model relation UML class which models a relation and some possible specializations but not all of them.

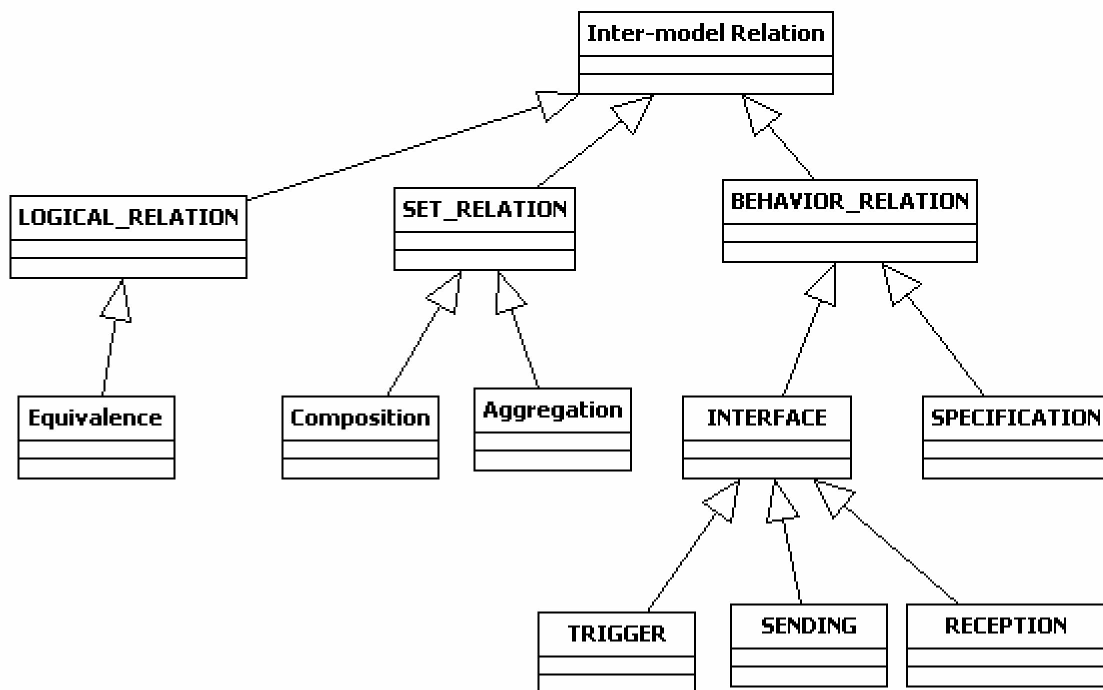


Figure 26. Inter-model relations diagram

Checking of integrated model

During this activity the verification process involves more than one model, i.e. the integrated model. Guaranteeing that the same types of messages are used in the integrated model is an example of this kind of checking.

Example

Concerning *CIS* and *SIS* models, there is one inter-model relation which is an instance of the *Equivalence* class of the relation model (Figure 26). It is the relation between *ExternalCommunication* CORE Interface in Figure 13 and *EComm* SysML Interface, which *extcomm Link* in Figure 14 belongs to. They are the same concept but defined differently in

both models. Actually it is the interface between the systems described in each model, their joint point.

IV.5.4. General constraint definition

Definition

A constraint is modeled as a property of a system that must be satisfied. Commonly a constraint is generated or derived from system requirements, i.e. it is part of its specification (e.g. "*The maximum duration of an upload/download of the flight ops daily data information shall be limited to 5 minutes whatever the media, wire or wireless*"). In our approach, we can set up constraints implying elements of different models issued from different points of view thanks to the annotations that carry out these points of view depending on the sued domain ontology. The constraints are formalized using the terms and concepts of the knowledge model. For instance, we can assert that communications between two models shall always be from components belonging to high-level security sectors to components of lower-level security sectors.

Method

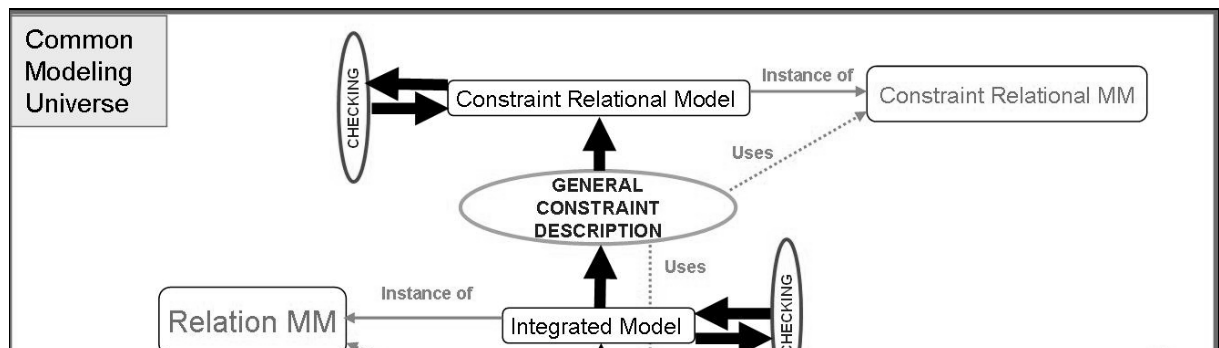


Figure 27. Focus on General Constraint Description activity

At this level of the approach, we need to express constraints that involve both model entities and knowledge. This capacity must be flexible enough since different kinds of models and constraints may occur. A model of expressions (*Constraint Relational Model* in Figure 27) encoding the properties to be validated needs to be defined. It has to support the formalization of the properties expressed using both annotated models and inner models. Once the property is described by instantiating the defined expression model referring to the annotated elements and to the inter-model relations, this property over the models can be finally checked.

The expression model we have adopted in our examples and case studies is based on First Order Logic (FOL) expressions. On one side, this model contains elements representing implicit semantics (FOL part shown in Figure 28). As we can see, FOL expressions have been

defined as a new type of *BOOLEAN_EXPRESSION*. Thus, one FOL expression consists of a set of quantified variables over a *BOOLEAN_EXPRESSION* (the predicate).

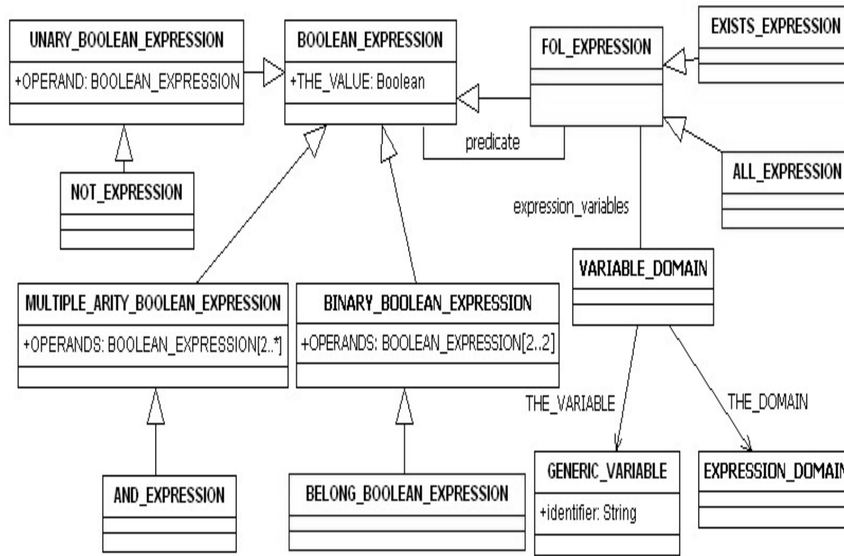


Figure 28. Excerpt of expressions structure in a UML diagram.

On the other side we need to cover explicit semantics. This is the role of variables since their meaning depends on the entity they refer to. As illustrated on Figure 29, variables may be typed by basic types like *String* (*STRING_VARIABLE*), *Boolean* (*BOOLEAN_VARIABLE*) and *Numeric* (*NUMERIC_VARIABLE*), or by complex types. For instance, in the FOL model of Figure 29 we define complex variables representing a path to entities and to attributes of models.

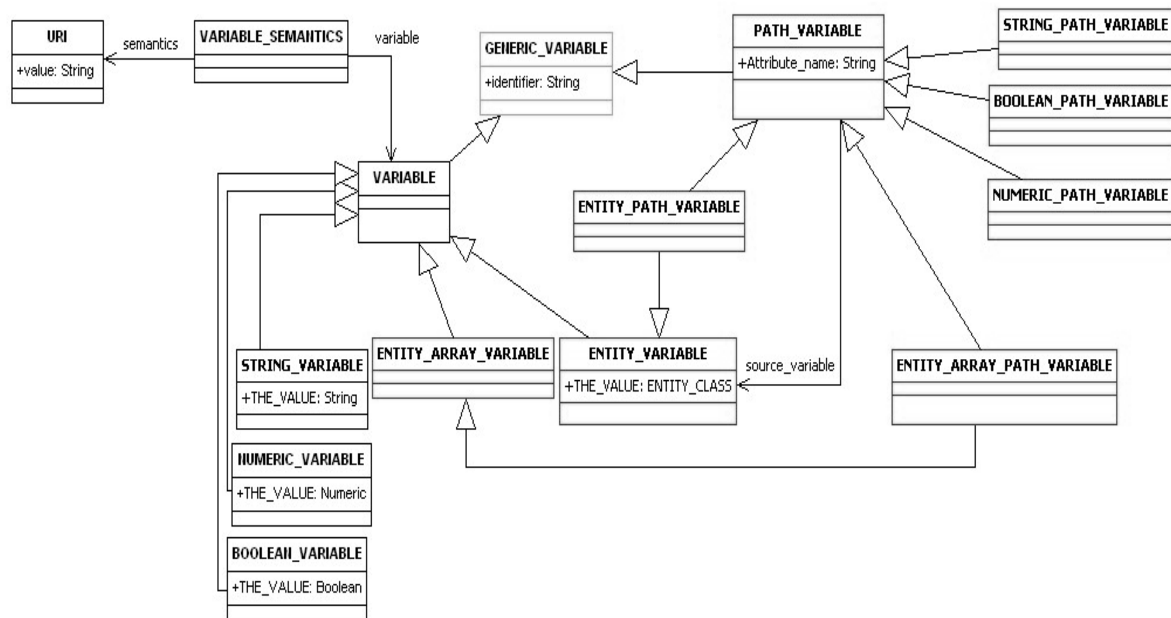


Figure 29. View of variables model in UML.

Checking of constraint

Once the constraint to be validated is expressed in terms of instances of an expression model, taking into consideration both imported elements and annotations, the final activity of the approach consists of the validation of the expression and the analysis of the results. For the validation aspect, the expressions must be evaluated, i.e. the framework supporting the approach has to provide the capacity of interpreting the instances of the expression models. With regard to the analysis of the results, we need to be able to keep the traceability of the constraint evaluation in the source models. This traceability is guaranteed by the importation process and the shared and unified concepts and object identifiers. Thus, observing the output of the validation of a constraint an engineer can identify the erroneous elements of the source models and perform the required actions to fulfill the broken requirement.

Example

In order to describe the use of expressions, we complete the message communication case. As we have mentioned, we need to check that *every communication from a critical domain system to a non-critical domain one shall implement a protocol considered as secure*. From a logical point of view, we can express this constraint for the elements of the case as a First Order Logic expression:

$$\forall l : Link \left| \begin{pmatrix} \begin{pmatrix} (l \in Y.comprised_of) \wedge \left(\exists i : Item \left| \begin{pmatrix} (i \in l.transfer) \\ \wedge \left(i\{represents\} = Z\right) \end{pmatrix} \right. \right) \\ \wedge\ \exists o : Operation_UML \left| \begin{pmatrix} (o \in X.owned_operation) \wedge \\ \exists p : Parameter_UML \left| \begin{pmatrix} (p \in o.owned_parameter) \\ \wedge \left(p\{represents\} = Z\right) \end{pmatrix} \right. \end{pmatrix} \right. \end{pmatrix} \\ \Rightarrow \\ \left(\exists cp : Communication_Protocol \left| \begin{pmatrix} cp.isSecure = TRUE \\ \wedge\ l\{protocol\} = cp \wedge X\{protocol\} = cp \end{pmatrix} \right. \right) \end{pmatrix} \right.$$

Figure 30. First Order Logic expression.

Figure 30 expresses the constraint to be checked. Y is a variable referring to CORE *Interface*; X is the variable referring to the related (equivalence inter-model relation) SysML *Interface* and Z is the variable linked to the knowledge concept representing a message. Concerning the variables:

- l is a variable containing the instances of the CORE *Links* belonging to Y.

- i is a variable denoting the instances of the CORE *Items* transferred by l .
- o is a variable defining the instances of the SysML *Operations* owned by *X*.
- p is a variable referring to the SysML *Parameters* of *o*.
- cp is a variable containing the instances of the communication protocols defined in our knowledge base.

Below, the different parts of the expression are detailed.

$$(l \in Y.comprised_of) \wedge \left(\exists i : Item \left| \begin{pmatrix} (i \in l.transfer) \\ \wedge \left(i\{represents\} = Z\right) \end{pmatrix} \right. \right)$$

Figure 31. Messages in the CORE model.

The expression of Figure 31 characterizes all the instances of *Item* CORE class that are transferring the message with URI *Z* via the interface *Y*. Here the *i{represents}* notation defines the annotation named *represents* connected to the entity *Item* represented by the variable *i*.

$$\exists o : Operation_UML \left| \begin{pmatrix} (o \in X.owned_operation) \wedge \\ \exists p : Parameter_UML \left| \begin{pmatrix} (p \in o.owned_parameter) \\ \wedge \left(p\{represents\} = Z\right) \end{pmatrix} \right. \end{pmatrix} \right.$$

Figure 32. Messages in the SysML model

The expression of Figure 32 defines the fact that some *Parameter_UML* instances of the SysML model are annotated as the message of URI *Z* and belong to an *Operation* of the interface *X*.

$$\exists cp : Communication_Protocol \left| \begin{pmatrix} cp.\mathrm{sec}urised = TRUE \\ \wedge l\{protocol\} = cp \wedge X\{protocol\} = cp \end{pmatrix} \right.$$

Figure 33. Communication protocol must be secure

Finally, Figure 33 means that the same protocol is used in both models by comparing the annotation *{protocol}* of the *Link* instance in CORE and the annotation *{protocol}* of the *Interface X* in SysML. It also asserts that it is a secure protocol (attribute *isSecure* has value *"true"* in the knowledge base).

As shown in the first part of the expression of Figure 30, validation is performed for all the *Link* instances of the CORE model belonging to *Interface Y*. Thus, in order to permit its automatic and dynamical evaluation, this logical expression is implemented as instances of the expression model. As an illustration of the results of such an evaluation, below two different instances of CORE *Link* class are defined to identify two checking situations.

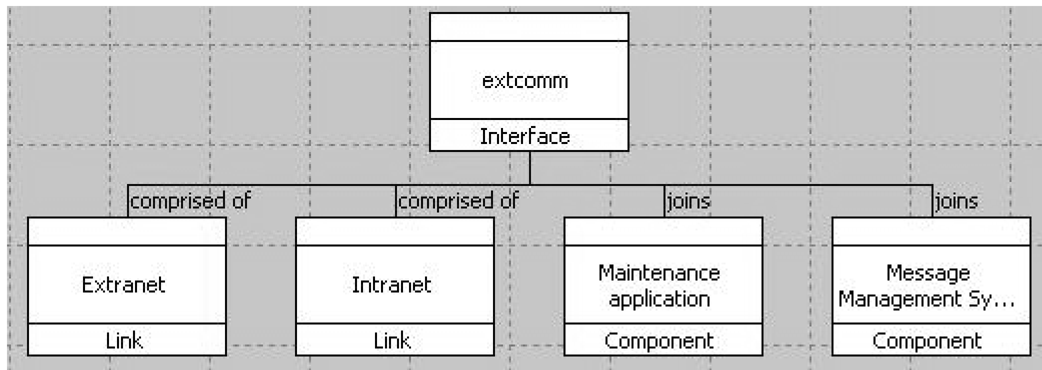


Figure 34. Instances of CORE Link class in SIS model

1) **Fulfilled constraint**. In this scenario, a CORE *Link* (*Intranet* in Figure 34) has been annotated to indicate that its protocol is "*EX25*", the same of the SysML *ExternalCommunication Interface* of Figure 14. According to the knowledge base (see Figure 23) "*EX25*" is a secure protocol, i.e. *Boolean* attribute *isSecure* value is "*true*". Therefore evaluation of the constraint returns *true,* i.e. *every communication from Intranet to ExternalCommunication* ***implements*** *a protocol considered as secure*.

2) **Incorrect relation**. Another scenario involves a different CORE *Link* (*Extranet* in Figure 34) pointing to the "*X25*" communication protocol, which is not secure and is different to the "*EX25*" protocol of the SysML *ExternalCommunication Interface*. As a result, the verification of the constraint fails in this case, i.e. *communication from Extranet to ExternalCommunication* ***does not implement*** *a protocol considered as secure*. In this situation, the engineer in charge of the design can exploit the links from the annotated models to the exported models to track the erroneous elements, i.e. the *Extranet Link*, of the source models in order to correct them.

IV.6. Conclusion

In this chapter we have described our approach which is a method to integrate and validate *same level* structural and functional models based on making explicit the implicit knowledge of engineers. The approach consists of a process which manipulates models with the support of some aside models as resources. The different activities of the method are presented around an example. Thus, we firstly describe the export of the source models to a shared framework which guarantees both the syntactical homogenization and the integrity of original models. Secondly, we use aside knowledge models to annotate the exported models obtaining the semantics homogenization. Next, this homogenization allows us to relate the annotated models and to express inter-model constraints thanks to an aside expression model. The

expression model permits references to both annotated entities and concepts of the knowledge base. Finally, such constraints are validated via the implementation of the expression model.

Chapter V

Approach validation

Summary

V.1. Introduction 71

V.2. Exportation of SysML and CORE models 71

V.3. Annotation using implicit knowledge 73

V.4. Model integration using equivalences 74

V.5. General constraint definition with First Order Logic expressions 75

V.5.1. Contribution to PLIB expressions language 75

V.5.2. Inter-model constraint verification 78

V.6. Implementation with ECCO toolkit 79

V.6.1. The common framework 81

V.6.2. Step by step implementation 81

V.7. Conclusion 86

Abstract. In this chapter we develop a case study in order to validate our approach and to illustrate their different steps. The scenario involves two different modeling languages, SysML and CORE, and a complex inter-model constraint. This case study is used to formally validate our proposal using EXPRESS as the shared and common modeling language. In the end, we describe the implementation of models, resources and activities of our approach in the ECCO toolkit framework in order to perform the operational validation from a scientific point of view.

V.1. Introduction

In this chapter we develop the formal modeling and implementation of the approach introduced in Chapter IV. The objective of this modeling is to formally validate the approach in a priori process.

We implement the case introduced in section IV.1. This case involves models of two systems that belong to different domains of the digital systems architecture determined by distinct security, integrity, and availability requirements. We call those domains the *Closed World*, and the *Open world* respectively. One SysML model illustrates the *Closed World*, Cockpit Information System (CIS), and one CORE model describes the *Open World*, Shared Information System (SIS). Amongst the functions performed by SIS we find maintenance supporting functions. Therefore, in these models an interface is set to represent the communications between the cockpit and the maintenance subsystem which is part of the *Open World*. This interface is used to send messages from the *Closed World* to the *Open World*. The objective of this case study is to verify an overall requirement, i.e. a requirement implying both models: "*All messages from Closed World to Open World shall use a secure communication protocol*".

Next sections depict the implementation of the case study along with the manipulated models and the resources and in accordance with the modeling process activities described in section IV.5.

V.2. Exportation of SysML and CORE models

In order to permit the exportation of the CIS and SIS models we build the SysML and CORE meta-models respectively. Each meta-model is implemented once as a distinct EXPRESS schema enclosing the entities, attributes and constraints of each particular modeling language. As an example, Figure 35 shows LINK entity of the CORE meta-model in EXPRESS format.

```
SCHEMA CORE_SCHEMA;
...
--THIS ENTITY REPRESENTS THE LINKS OF CORE META-MODEL
ENTITY LINK
      SUBTYPE OF (ENTITY_CORE);
      CAPACITY: OPTIONAL CALCULATION_KIND;
      CAPACITY_UNITS: OPTIONAL STRING;
      DELAI: OPTIONAL CALCULATION_KIND;
      DELAI_UNITS: OPTIONAL STRING;
      PROTOCOL: OPTIONAL STRING;
      SPECIFIED_BY: SET[0:?] OF REQUIREMENT;
      TRANSFERS:  SET[0:?] OF ITEM_CORE;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'LINK';
INVERSE
      CONNECTS_THROUGH: SET[0:?] OF COMPONENT FOR
CONNECTED_THROUGH;
      CONNECTS_TO: SET[0:?] OF COMPONENT FOR CONNECTED_TO;
      SERVICED_BY: SET[0:?] OF FUNCTION_CORE FOR SERVICES;
      COMPRISES: SET[0:?] OF INTERFACE_CORE FOR COMPRISED_OF;
END_ENTITY;
...
END_SCHEMA ;
```

Figure 35. Meta-model of CORE language implemented in EXPRESS

The original modeling tools allow modelers to export the SysML and CORE models into XMI-compliant format (OMG, 2011b). The exportation of the source models is done by interpreting the content of the XMI files in terms of the EXPRESS meta-models, i.e. they are converted into instances of the meta-models and imported in this way into the framework. Thus, the result of the exportation activity is a set of instances in EXPRESS format according to CORE (see Figure 36 for an example of instances) and SysML meta-models.

```
DATA ('CORE_SCHEMA', ('CORE_SCHEMA'));
/********** Creation stamp date ****************/
#1=T_DATE(2, 2010, 11, 13, 0, 0);
/********** Items representing a message ****************/
#13=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0001', $, $, $, $, $,
$, $, $, (), (), (), ());
#26=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0002', $, $, $, $, $,
$, $, $, (), (), (), ());
#39=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0003', $, $, $, $, $,
$, $, $, (), (), (), ());
/********** Link transferring the items ****************/
#50=LINK(*, $, $, #1, 'dsz', $, #1, 'ExtComm', $, $, $, $, $, (),
(#39,#26,#13));
/********** Interface comprising the link ****************/
#247=INTERFACE_CORE(*, $, $, #1, 'dsz', $, #1, 'extcomm', (#50),
());
ENDSEC;
```

Figure 36. Instances of CORE in ISO-10303-21 format

V.3. Annotation using implicit knowledge

The property to be validated ("*All messages from Closed World to Open World shall use a secure communication protocol"*) involves several concepts that are not explicitly formalized in the models: the concept of *message*, the security *domain* notion (*Closed World/Open World*) and the list of encrypted protocols. These concepts are represented by EXPRESS classes and their instances form the knowledge base. Figure 37 shows the *message* and *communication protocol* concepts. A *message* is composed of attributes indicating the origin of the message (*person_from*), its addresses (*person_to, person_cc, person_cco*) and its content (*message_parameter*). A *communication_protocol* is composed of a name (*protocol_name*) and an attribute (*is_secure*) indicating whether it is a secure communication protocol or not. Each instance is identified by an URI. For example, in Figure 38 the *communication protocol* named *EX25* is uniquely identified by "*http://www.eads.net/thesedsz/knowledge/protocol/ex25*".

```
--This entity represents a Message
ENTITY MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
   PERSON_FROM: SET [0:?] OF ORIGIN_OF_MESSAGE;
   PERSON_TO: SET [0:?] OF DESTINATION_OF_MESSAGE;
   PERSON_CC: SET [0:?] OF COPY_OF_MESSAGE;
   PERSON_CCO: SET [0:?] OF SECRET_COPY_OF_MESSAGE;
   MESSAGE_PARAMETER: SET [1:?] OF STRING;
END_ENTITY;

--This entity represents the communication protocols
ENTITY COMMUNICATION_PROTOCOL
SUBTYPE OF (KNOWLEDGE_CLASS);
   PROTOCOL_NAME: STRING;
   IS_SECURE: BOOLEAN;
END_ENTITY;
```

Figure 37. Knowledge model implemented in EXPRESS

```
#115=URI('http://www.eads.net/thesedsz/knowledge/protocol/ex25');
#116=URI('http://www.eads.net/thesedsz/knowledge/protocol/ethernet');
#117=URI('http://www.eads.net/thesedsz/knowledge/protocol/x25');
#112=COMMUNICATION_PROTOCOL(*, *, $, #115, 'EX25', .T.);
#113=COMMUNICATION_PROTOCOL(*, *, $, #116, 'ETHERNET', .T.);
#114=COMMUNICATION_PROTOCOL(*, *, $, #117, 'X25', .F.);
```

Figure 38. EXPRESS instances representing part of the knowledge base

The instances of the knowledge model are used to annotate the exported model, i.e. to link entities of the exported models to instances of the knowledge base using the *ANNOTATION_CLASS* entity of Figure 39.

```
ENTITY ANNOTATION_CLASS;
   NAME: T_DOMAINE;
   MY_KNOWLEDGE: LIST OF URI;
   MY_ENTITIES: LIST OF ENTITY_CLASS;
END_ENTITY;
```

Figure 39. Annotation class implemented in EXPRESS

Figure 40 illustrates several possibilities of annotation:

- **A concept represented differently depending on the modeling language**. The annotation *#118* indicates that the URI identified by *#115* (*EX25* communication protocol) is the *protocol* used in entity *#50*, which is a *Link* of the SIS model (CORE). The annotation *#119* indicates that the same URI is also the *protocol* applied in entity *#78*, which is an *Interface* of the CIS model (SysML).

- **A concept represented by a group of entities in the exported models**. The annotation *#59* says that a *message*, URI identified by *#60*, is *represented* by the CORE *Items #13*, *#26* and *#39*.

```
#105=ANNOTATION_CLASS ('represents', (#60), (#100));
#59=ANNOTATION_CLASS ('represents', (#60), (#13,#26,#39));
#118=ANNOTATION_CLASS ('protocol', (#115), (#50));
#119=ANNOTATION_CLASS ('protocol', (#115), (#78));
```

Figure 40. EXPRESS instances representing the annotated model

V.4. Model integration using equivalences

The integrated model in this case study is simple and includes only one instance of the *Equivalence* class (see Figure 41) from the relation meta-model. This instance, shown in Figure 42, indicates that the entity *#78* of the CIS model (a SysML *Interface*) and the entity *#247* of the SIS model (a CORE *Interface*) are equivalent. Therefore, this equivalence can be used to build the constraint about the communication protocol between the connected entities.

```
--This entity represents relations of type Equivalence
ENTITY EQUIVALENCE
   SUBTYPE OF(LOGICAL_RELATION);
END_ENTITY;
```

Figure 41. Equivalence class implemented in EXPRESS

```
#300=EQUIVALENCE('Interface Equivalence', (#78), (#247), $);
```

Figure 42. EXPRESS instance of an equivalence relation

V.5. General constraint definition with First Order Logic expressions

The constraint "*All messages from Closed World to Open World shall use a secure communication protocol*" can be translated into a logical expression, in First Order Logic. Therefore, for this case study we need an expression model allowing the instantiation of such expressions. Next sections detail its characteristics.

V.5.1. Contribution to PLIB expressions language

First of all, we build a grammar in order to clearly define the variety of formal expressions to be verified. The grammar, expressed in the Backus Naur Form (BNF (Naur, Backus, Bauer, & Green, 1963)) is inspired on the PLIB (ISO, 1997b) expressions language proposal. Even though the problem we deal with is not the components and parts library modeling which is the core of PLIB (see II.2.1), its approach to build expressions in a structured and easily extendible way fits our expressivity needs. Thus, we extend the original PLIB proposal with First Order Logic concepts in order to fulfill our logical constraint expressions needs.

For clarity purpose, the structure of the model is organized into several basic concepts. This organization is described below.

Type

Some elements of the model are used to distinguish the type of an expression. The main types are: *String*, *Boolean* and *Numeric*. They are combined with other elements of expressions, as cardinality which is explained in the next paragraph.

Cardinality

According to the number of operands, an expression is *unary*, *binary* or *multiple*. The model reflects these cardinalities and the combination with other structures:

- NOT true; is an example of a *unary Boolean* expression.
- 4 DIV 2; is a sample of a *binary Numeric* expression.
- 4 + 3 + 1; shows a *multiple Numeric* expression.

Functions

Functions are elements used to represent the processing of input providing an output as a result. They are built-in functions or defined (ad-hoc) functions. The *length* function is an example of a built-in function: given a string expression it returns the length of the string.

Variables

The model allows the inclusion of variables in expressions. They are replaced by a value during the evaluation of expressions. Variables are used to manipulate entities of the models.

Literals

Literals are the basic elements to build expressions. In a tree-modeling perspective they are the leaves. We have *String*, *Numeric* and *Boolean* literals as: "*X.25*", *5* and *false*, respectively.

Expressions

Finally, all the aforementioned principles are composed to build the expressions, for example *ODD(" <NUMERIC_EXPRESSION>")* is a *binary Numeric* expression whereas *BELONG_BOOLEAN_EXPRESSION("<OPERANDS>")* is a *Boolean* function expression.

From grammar to model

We developed the FOL model in EXPRESS. The first step consists in describing a FOL meta-model in terms of entities and attributes. To illustrate this idea let's take FOL *EXISTS* expression.

Grammar definition

In the grammar the *EXISTS* expression is represented as:

```
   <EXISTS_EXPRESSION> ::= " EXISTS " {<GENERIC_VARIABLE>} "| ("
<BOOLEAN_EXPRESSION> ")"
```

Figure 43. Exist expression in BNF form

EXISTS is a type of *FOL expression* which is a derivation of *Boolean expression*. It means that the result of the expression is a *Boolean*. The two operands are a set of variables containing the elements to be validated and a *Boolean expression* representing the property, *predicate*, that at least one of the values of the variable must fulfill.

Model translation

The previous grammar elements are represented in our model with two corresponding entities:

```
--THIS ENTITY REPRESENTS THE FOL (FIRST ORDER LOGIC) EXPRESSIONS
ENTITY FOL_EXPRESSION
SUBTYPE OF (BOOLEAN_EXPRESSION);
      CONTEXT_VARIABLES: OPTIONAL SET OF GENERIC_VARIABLE; --CONTEXT
      EXPRESSION_VARIABLES: SET OF VARIABLE_DOMAIN; --VARIABLES
      PREDICATE: BOOLEAN_EXPRESSION;--THE PREDICATE TO BE EVALUATED
END_ENTITY;


--THIS ENTITY REPRESENTS THE EXISTS FOL ASSERTION
ENTITY EXISTS_EXPRESSION
SUBTYPE OF (FOL_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN:=EXISTS_FUNCTION
(SELF);
END_ENTITY;
```

Figure 44. Exist expression in the EXPRESS model

As shown in the previous figure each node of the grammar is translated into an entity. For modeling reasons we need to add *EXPRESSION_VARIABLES* attribute to store the values to be replaced in the variables. The semantics of the expression are carried and calculated by the *EXISTS_FUNCTION* function as we explain in the next sub-section.

Modeling the semantics

To complete our model we add semantics with a procedural approach, i.e. implementing the semantics using EXPRESS functions. For instance, the *exists_function* of the example interprets the attributes of the expression and replaces the variable by the different values until one of them satisfies the *Boolean* expression (*predicate*).

```
   exists_function(arg:EXISTS_EXPRESSION): BOOLEAN;
```

Figure 45. EXPRESS function implementing the interpretation of the expression

V.5.2. Inter-model constraint verification

Once the FOL model is built according to the rules described in previous section, next step translates the inter-model constraint ("*All messages from ClosedWorld to OpenWorld shall use a secure communication protocol*") into a set of instances forming a FOL expression (evoked in Figure 30), i.e. a group of sub-expressions and variables as shown in Figure 46.

```
...
#157=ENTITY_DOMAIN((#50));
#362=ENTITY_VARIABLE(*, 'l');
#158=VARIABLE_DOMAIN(#157, #362);
#159=ALL_EXPRESSION(*, (), (#158), #352);
#351=NOT_EXPRESSION(*, #356);
#352=OR_EXPRESSION(*, (#351,#348));
#348=EXISTS_EXPRESSION(*, (), (#350), #347);
#350=VARIABLE_DOMAIN(#349, #339);
#347=AND_EXPRESSION(*, (#344,#345,#346));
#344=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#340,#343));
#345=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#341,#339));
#346=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#342,#339));
#340=BOOLEAN_ARRAY_PATH_VARIABLE($, 'cp.securised', #339,
'IS_SECURE', .F.);
#341=ENTITY_PATH_VARIABLE($, 'l\\{protocol\\}', #362, 'protocol',
.T.);
#342=ENTITY_PATH_VARIABLE($, 'X\\{protocol\\}', #331, 'protocol',
.T.);
...
```

Figure 46. Excerpt of the instances implementing the inter-model constraint

Finally, we are ready to effectively perform the inter-model constraint verification. For that, two EXPRESS features are exploited in order to evaluate of the constraint: the *derived attributes* and the *local rules*. A *derived attribute* is a kind of attribute in EXPRESS whose value is calculated whenever it is used. In our operational validation, all FOL classes that may build up an expression have a *derived attribute* called *THE_VALUE*. A *local rule* is a property which must be *true* for all the instances of a class, e.g. "*diameter>5*". Hence, taking advantage of the instances checking engine, the evaluation of a constraint starts by defining a *local rule* at the higher level, i.e. the root of a FOL expression (which is instance *#159, ALL_EXPRESSION,* in our case study) saying that *THE_VALUE* attribute must be *true*. Since this attribute is a derived one, the evaluation of the rule triggers a concatenation of calculation of *THE_VALUE* attributes following the tree structure of a FOL expression, e.g. in order to calculate *#159* the instance *#352* (an *OR_EXPRESSION*) must be evaluated previously (i.e.,

code in Figure 47 is executed to calculate the derived attribute value) and so on. In the end, the constraint is completely analyzed and the EXPRESS instances checker provides the result.

```
--This function implements OR_EXPRESSION
FUNCTION or_fct (ARG: BOOLEAN_EXPRESSION): BOOLEAN;
--Local variables
LOCAL
  I: INTEGER;
END_LOCAL;
--We treat each operand
REPEAT I:=1 TO SIZEOF (ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS);
  --The operand must be of type BOOLEAN_EXPRESSION
  IF ('TOP_SCHEMA.BOOLEAN_EXPRESSION' IN TYPEOF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I])) THEN
    --We read the value of the operand.
    --This action triggers the calculation of derived attribute the_value
    IF (ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I].the_value) THEN
      --When one of the operands is TRUE we finish and return TRUE
      RETURN(TRUE);
    END_IF;
  ELSE
    --Otherwise we return FALSE (operand is not a BOOLEAN_EXPRESSION)
    RETURN(FALSE);
  END_IF;
END_REPEAT;
--Otherwise we return FALSE (none of the operands is TRUE)
RETURN (FALSE);
END_FUNCTION; -- OR_FUNCTION
```

Figure 47. Derivation of the value of attribute "the_value" for OR_EXPRESSION entity

V.6. Implementation with ECCO toolkit

Previous sections have introduced the models of a first case study formalized in EXPRESS modeling language. These models have been implemented in a tool which covers the main functions of our approach. Concerning the functions needed for our process we identify:

- **Exportation**. To instantiate the source models by the use of meta-models in the common framework. The *Exportation* module (shown in the functional architecture of Figure 48) manages the exportation function which exports the source models to the common framework

- **Annotation**. To provide the mechanism to link the imported models with the knowledge models. According to the functional architecture of Figure 48, the *Knowledge* module provides the interface to the knowledge models that are used

by the annotation module whereas the *Annotation* module supports the annotation function and puts in relation the knowledge models with the exported models to obtain the annotated models.

- **Integration**. To enable the connection between elements of the annotated models. The *Integration* module (Figure 48) corresponds to the integration function and it allows the construction of the integrated model.

- **Constraint expression**. To provide a flexible way for building expressions in order to declare the properties to be checked over the integrated model. The *Expression* module (Figure 48) is in charge of the definition of dynamical expressions which formalize the inter-model constraints.

- **Constraint validation**. To validate the constraints by interpreting the instances of expressions. The "*Expression validation*" module (Figure 48) contains the code that interprets and executes the expressions over the integrated model in order to validate the inter-model constraints.

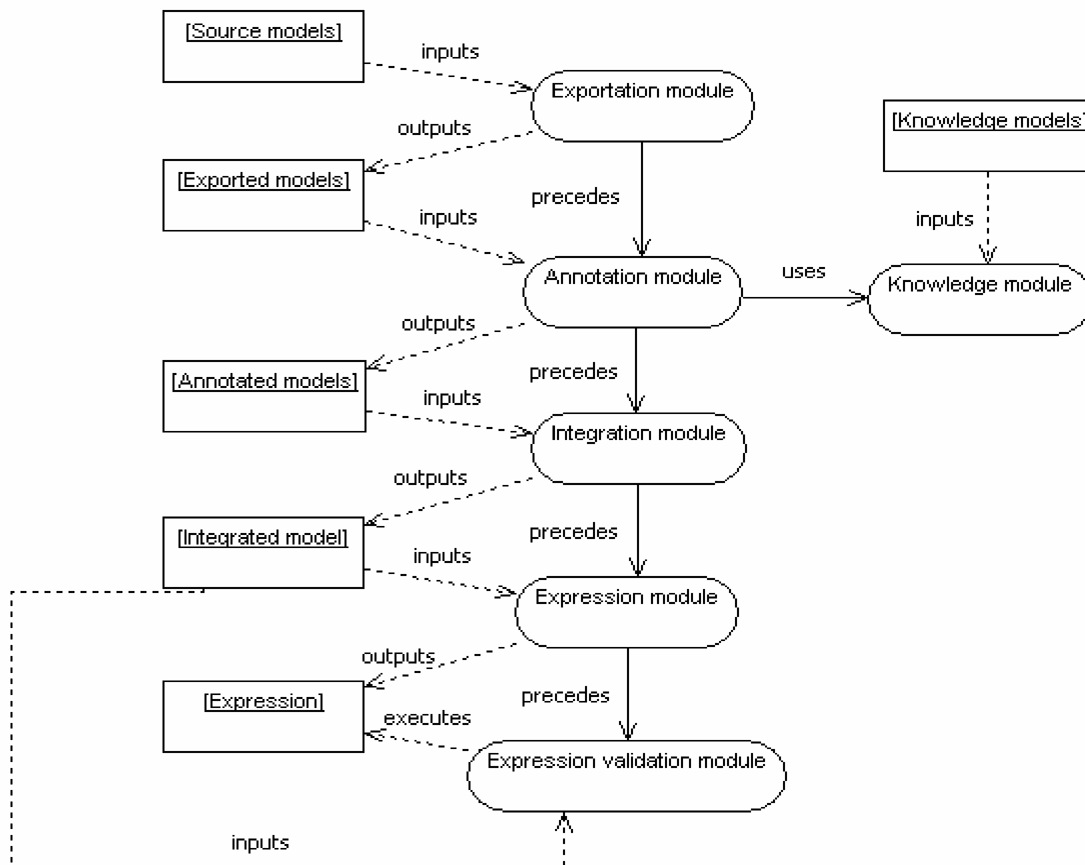


Figure 48. Functional architecture of the operational validation

V.6.1. The common framework

The previous analysis leads us to take some decisions about an adequate implementation in order to perform a consistent operational validation from a scientific point of view.

One of the most robust EXPRESS environments is the ECCO toolkit (PDTec GmbH, 1998). It offers a set of tools which provides a common user interface allowing mainly: the construction of EXPRESS models; the management of EXPRESS instances; and the evaluation of constraints. Thus, in this common framework we can operate the different models of our approach and use the instances checking engine to verify the inter-model constraints.

The models are organized in schemas. Figure 49 shows the architecture regarding the schemas used in our operational validation. *TOP_SCHEMA* includes some common entities and types, e.g. the type DATE; the objectives of the rest of schemas are detailed in the following sections along with the implementation in ECCO of the modeling process activities.

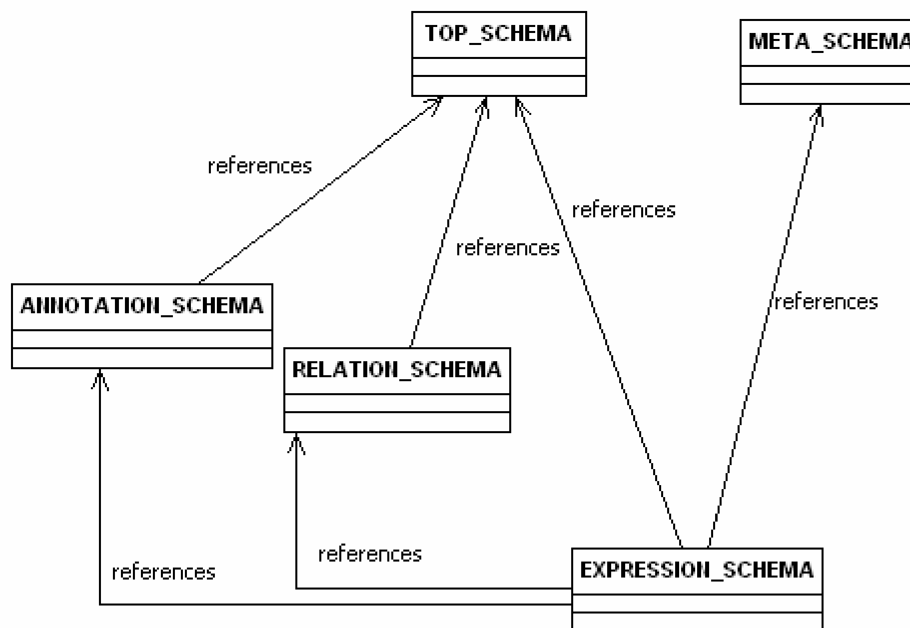


Figure 49. Schemas in EXPRESS

V.6.2. Step by step implementation

This section describes the implementation of the activities of the approach in the ECCO toolkit.

Prior to the execution of the activities, we prepare the environment according to the architecture defined in V.6.1, by creating a project including all the needed schemas as seen in Figure 50.

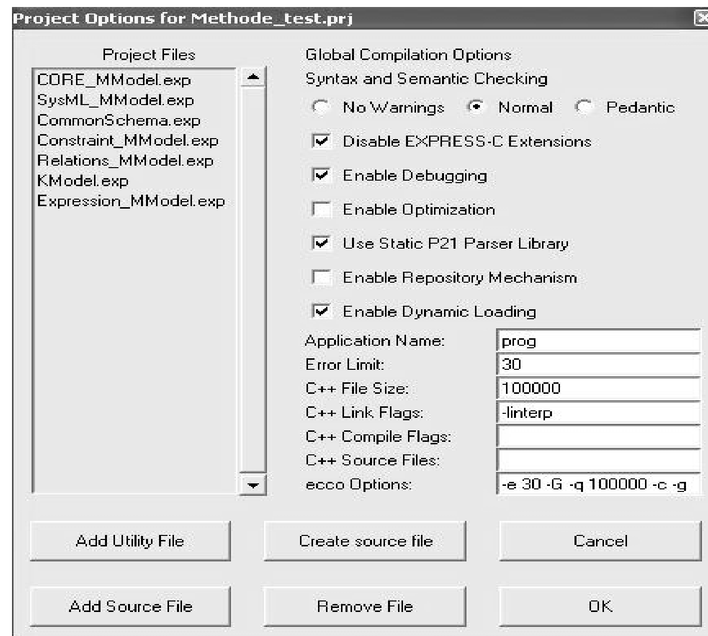


Figure 50. Creation of a project with ECCO toolkit

Building up the framework continues with the creation of entities, attributes and constraints of the different schemas using the model edition properties of the tool. Figure 51 shows an example with *META_SCHEMA*.

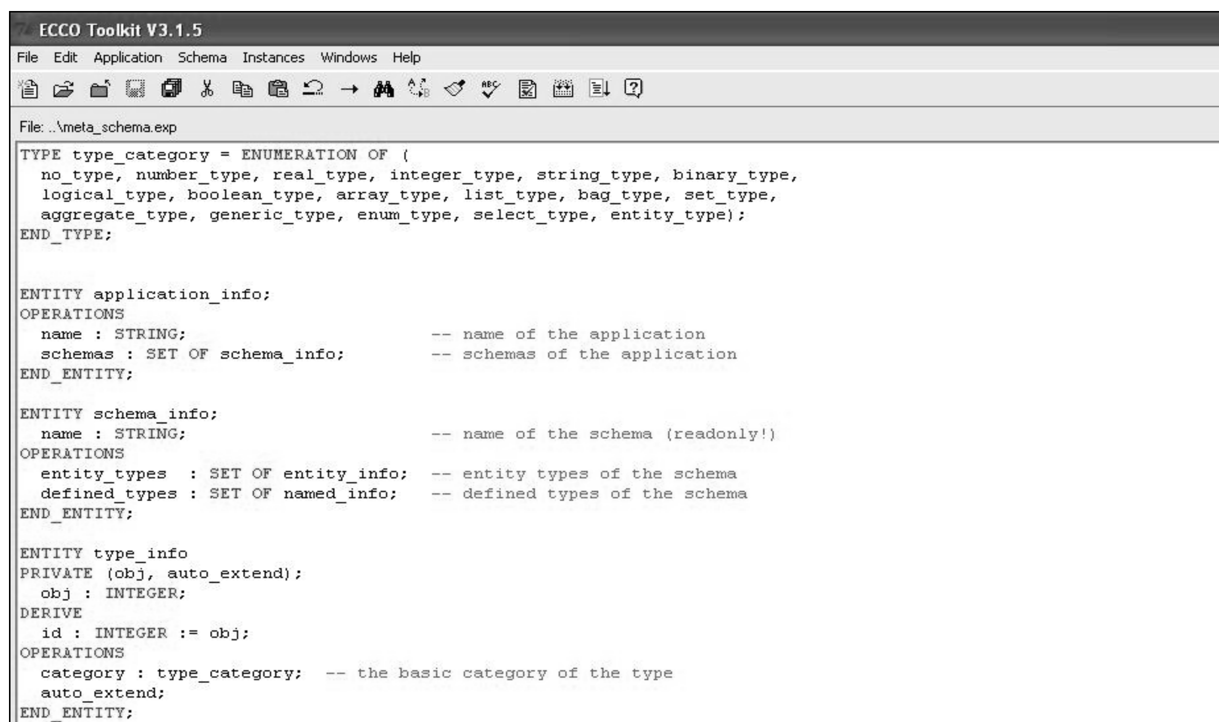


Figure 51. Edition of schema using ECCO toolkit

Taking into consideration the involved schemas we can start the inter-model constraints verification process.

Export

Source Meta-models

The meta-models are the means to export the source models into ECCO toolkit. Therefore, for each meta-model we implement a schema referencing the *TOP_SCHEMA* for the common types and entities. Furthermore, in order to manipulate meta-models, i.e. to work with classes and attributes without knowing the content of the meta-model below, ECCO dispenses the schema *META_SCHEMA*. This module makes it possible to apply meta-meta-modeling techniques, thus it is a layer over the implemented meta-models.

Source and exported models

The source models are exported into the ECCO toolkit as instances of the appropriate meta-model schema. A mapping is done between the meta-model in its exchange format, e.g. XMI, and the meta-model defined in EXPRESS. Then, an instance file compatible with one of the EXPRESS' standard (e.g. ISO-10303-21) is generated applying meta-modeling techniques. Thus, the source model is now expressed in terms of instances of the meta-model written in the EXPRESS language.

ECCO toolkit execution

The instances of the meta-models are uploaded in the tool through the "*Read Instances*" action. This action allows the user to look for the file containing the instances and to load it into the framework. By performing this simple action our source models are now loaded in our shared and common framework.

Annotation

Knowledge models

Even though our approach admits the use of different formalisms for the implementation of knowledge models, we have implemented the knowledge models in the EXPRESS modeling language in order to accelerate the operational validation. These knowledge models have only, for reuse reasons, a reference to the *TOP_SCHEMA* since they are independent from the rest of modules.

Annotated models

Once the source models are exported as instances in our framework, we need to complete them with annotations. As we are in the universe of EXPRESS instances, the annotation consists of adding new instances to form the annotated model. They are instances of the

ANNOTATION_SCHEMA of Figure 49. This annotation puts in relation the exported models with the instances of the knowledge schema, i.e. the knowledge base.

ECCO toolkit execution

As described before, we have decided to implement our knowledge models in the same environment; therefore before starting the annotation process we read the instances corresponding to our knowledge models in order to load them in the tool. Next, we execute the "*Open Entity Types*" action of the *Instances* menu and we add instances of the *ANNOTATION_SCHEMA* in order to connect the instances loaded during the *Export* activity and those forming our knowledge base. Figure 52 shows an illustration of the creation of instances via the ECCO toolkit interface.

```
--THIS ENTITY REPRESENTS AN UNIQUE IDENTIFIER (URI)
ENTITY URI;
        URI_VALUE: STRING;
INVERSE
        THE_CLASS: KNOWLEDGE_CLASS FOR MY_URI;
UNIQUE
        URI: URI_VALUE;
END_ENTITY;

-THIS ENTITY REPRESENTS AN ANNOTATION
ENTITY ANNOTATION_CLASS;
        NAME: T_DOMAINE;
        MY_KNOWLEDGE: LIST OF URI;
        MY_ENTITIES: LIST OF ENTITY_CLASS;
END_ENTITY;

-THIS ENTITY IS THE SUPER CLASS OF KNOWLEDGE CONCEPTS
ENTITY KNOWLEDGE_CLASS
        ABSTRACT SUPERTYPE
        SUBTYPE OF (ENTITY_CLASS);
        MY_URI: URI;
DERIVE
        SELF\ENTITY_CLASS.NAME:STRING := 'KNOWLEDGE_CLASS';
END_ENTITY;
```

Instances of Type ANNOTATION_CLASS

File Instance Options Attributes

OID	name	my_knowledge	my_entities
#118	'protocol'	[#115]	[#50]
#119	'protocol'	[#115]	[#78]
#248	'protocol	?	?

Figure 52. Creation of instances using ECCO toolkit

Model integration

Integrated model

The integrated model consists of instances of the *RELATION_SCHEMA* which allow interconnecting entities of the annotated models. The *RELATION_SCHEMA* contains the definition of the relation concept, i.e. an entity which has attributes as *origin* and *destination* to point to other entities, amongst other attributes. *RELATION_SCHEMA* needs the *TOP_SCHEMA* to process some general concepts.

ECCO toolkit execution

In this case we use the "*Open Entity Types*" action to create the instances of the *RELATION_SCHEMA*, i.e. we interconnect the annotated instances.

General constraint description and validation

Constraint relational meta-model

The *EXPRESSION_SCHEMA* contains the entities and attributes used for the construction of expressions. These expressions translate the inter-model constraints to an executable form for validation over the annotated models. Since it has to manage both concepts from knowledge and entities from the exported models, *EXPRESSION_SCHEMA* refers to the *ANNOTATION_SCHEMA* and to the *META_SCHEMA* respectively, besides to the *TOP_SCHEMA* for the general elements.

Constraint Relational model

We instantiate the *EXPRESSION_SCHEMA* in order to describe the expressions that implement the inter-model constraints that we verify. Therefore, we complete the EXPRESS instances of the integrated model with the instances that build up the expression. From this point, the instances cover all the necessary information to validate the inter-model constraints.

ECCO toolkit execution

First of all, the instances needed to form the expressions that translate the inter-model constraints are created through the "*Open Entity Types*" action onto the *EXPRESSION_SCHEMA*. Next and final step is to check the expressions. This checking is performed by the instances checker of the ECCO toolkit. This checker is called from the "*Check*" action and analyzes each instance in order to verify that constraints are observed. The result of the checking is a list of errors that can be browsed as seen in Figure 53. The details of the implementation of the instances checker in our inter-model expressions are explained in section V.5.2 with a concrete example.

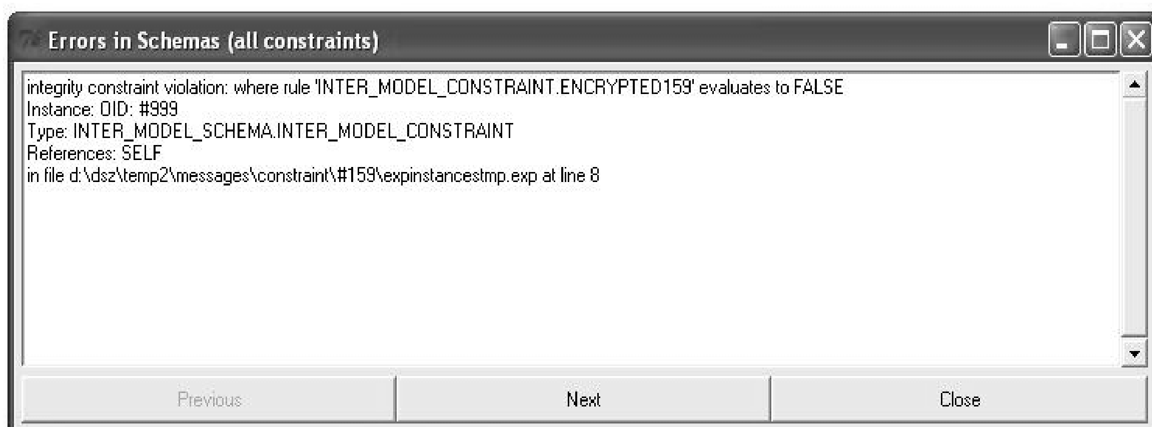


Figure 53. Check of instances with ECCO toolkit

V.7. Conclusion

In order to validate our approach we have chosen EXPRESS as the shared and common modeling language. In this case study we have used EXPRESS mainly to: 1) export the source models as instances of meta-models in an EXPRESS-compliant format; 2) develop knowledge models and 3) support FOL as property language. The formal modeling using EXPRESS modeling language has enabled the formal validation of the approach since verifies that we are able: to import SysML and CORE models; to design and populate knowledge models; to use the knowledge base to annotate imported models; to establish inter-model relations; to dynamically write a constraint; and to check such a constraint.

These formal models have been operationally validated using the ECCO toolkit as the framework of implementation. EXPRESS standard structures the models into different schemas. We have used this feature to better organize the different modules. Thus, a common schema describes basic elements of our method as models, classes and knowledge items. This schema is extended by other schemas in order to support each of the activities of our approach. A schema is written for each meta-model. Models are imported as instances of the classes of the meta-model schemas. Furthermore, the inter-model constraints are verified thanks to the instance checker implemented in the ECCO toolkit. However, the proposed method chain is generic enough, it could have been developed with other tools as JSDAI (GmbH, 2012) or Topcased (Topcased.org, 2011). Moreover, we have developed a prototype in order to provide users with an ad-hoc HCI and according to the activities of our method. The industrial evaluation described in Chapter VI is the guideline to define the main characteristics of the prototype.

Chapter VI Industrial evaluation

Summary

VI.1. Introduction ... 89

VI.2. Water and Waste System model ... 90

VI.2.1. Description ... 90

VI.2.2. The modeling process activities ... 90

VI.2.3. Conclusions ... 95

VI.3. Hydraulic and Engine systems models ... 95

VI.3.1. Description ... 95

VI.3.2. The modeling process activities ... 96

VI.3.3. Conclusions ... 102

VI.4. Ram Air Turbine models ... 103

VI.4.1. Description ... 103

VI.4.2. The modeling process activities ... 103

VI.4.3. Conclusions ... 110

VI.5. Conclusion ... 110

Abstract. In this chapter we carry out a validation of our proposal, using simplified models based on the analysis of four real industrial cases. The main objective of these cases is to validate the usability of the considered approach. The conclusions of this validation are used for the specification of the pre-industrial prototype described further on.

VI.1. Introduction

In a first stage we have carried out an exploration through a case study involving two modeling languages and a complex constraint expression. The goal is to have a proof of concept of the entire process. In order to increase the coverage and validate the industrial usability of the proposed approach we analyze several other cases. In each case, the complexity depends on the number of models, the number of modeling languages and the variations in modeling principles applied by the engineers. The particularities of each case are synthesized in order to give an idea of the main modeling scenarios that we want to implement in our approach. These scenarios are summarized in Figure 54 and listed below.

Number of models	Number of modeling standards	Focus
1 model	1 standard	Knowledge and annotations
2 models	1 standard	Same modeling rules
2 models	1 standard	Different modeling rules
2 models	2 standards	Heterogeneity

Figure 54. Industrial validation strategy

- Firstly, in section VI.2 we consider a case implying only one model in order to focus on the use of knowledge and annotations.

- In the second case described in section VI.3, a scenario implying two models is studied where the modeling standard is the same (e.g. SysML) and the models have been developed using the same modeling approach, i.e. the same modeling rules.

- Precisely, the third case (detailed in section VI.3.2) uses two models built with the same modeling language but using different modeling rules.

- Finally, the last case (section VI.4) has the largest variety in terms of heterogeneity since we worked with two models expressed in different standards, SysML and CORE, and with a rich knowledge model.

VI.2. Water and Waste System model

VI.2.1. Description

One first scenario concerns the retrieval and initial assessment of an existing Water and Waste System (WWS) architecture model that an engineer intends to reuse for the design of the Water and Waste System of a new aircraft. This scenario is based on a real situation where a team of A380 engineers checked the relevance of previous existing WWS models for their current program during the functional design stage. WWS is the system in charge of:

- supplying potable water to the lavatories and galleys
- draining of waste water from the lavatory washbasins overboard through drain mast
- vacuum of waste from the toilet and galley waste disposal
- draining of the waste tanks in waste vehicle on ground.

General Metadata

Extrapolating on this real situation, we make the assumption that system models from all previous aircraft programs could be managed, described with some annotations stored in some repositories and then searched for reuse.

General characteristics may help the engineer in retrieving models relevant to a new context. For instance, one would search an *Architecture* model, preferably modeled using SysML language, of the Water and Waste System and belonging to an *Aircraft Program* with more than one deck. Therefore, we introduce in the specification of our prototype the possibility to edit and manage such model annotations applied to models considered as black boxes. Amongst other possible scenarios, it is realistic enough to think that systems' models can be annotated with this kind of general information at the time they are built. This general information can easily be represented in the form of knowledge models. We use these knowledge models to annotate the exported WWS model in order to indicate its scope and applicable *Aircraft Program*. The advantage of having knowledge models is that, for instance, we can obtain the number of decks directly from the properties of the *Aircraft Program* instead of repeating this information as a specific annotation. Thus, knowledge models provide general metadata with more consistency.

VI.2.2. The modeling process activities

Next sections develop the implementation of this case study according to the models and activities represented in Figure 55.

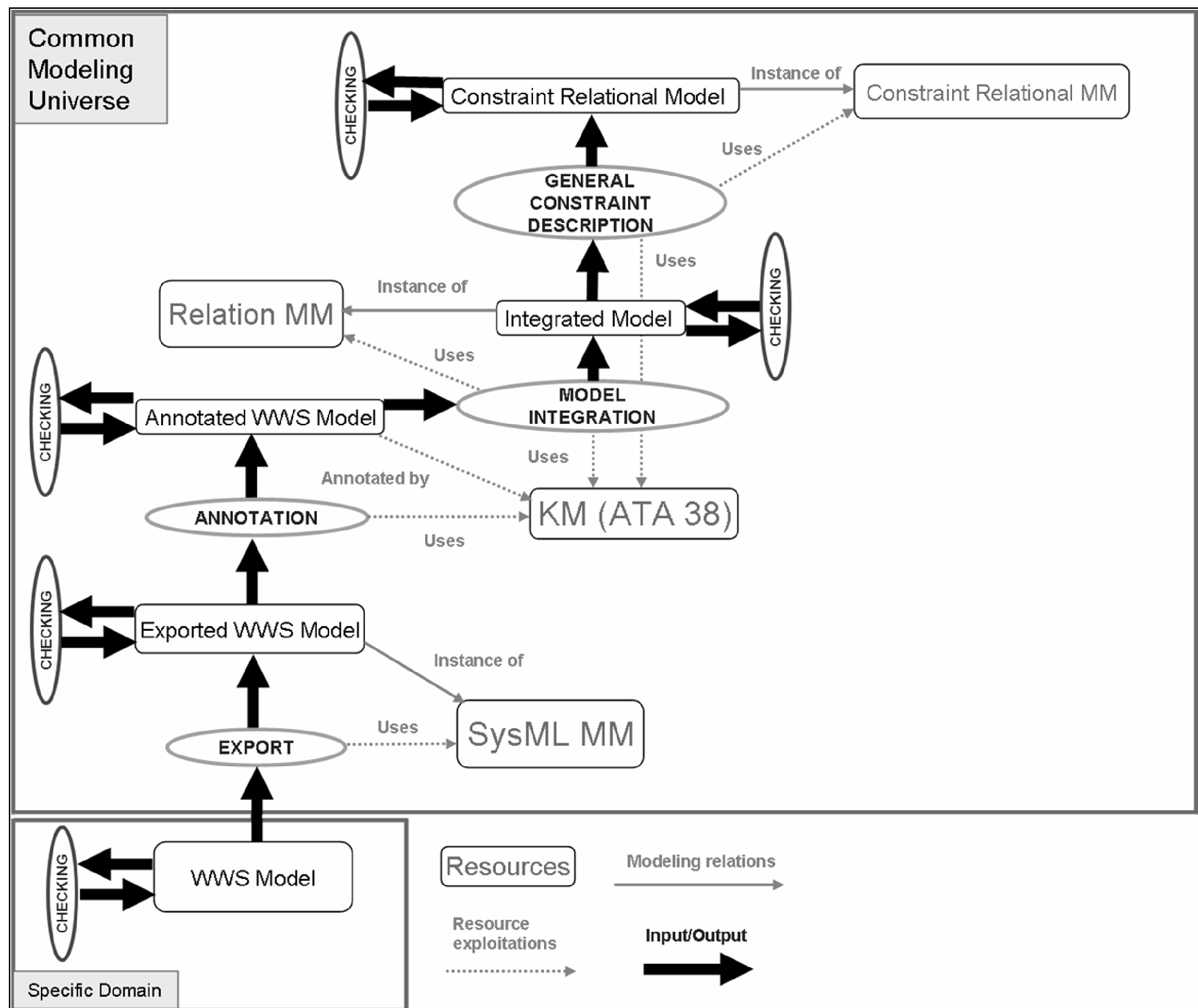


Figure 55. Approach applied to the WWS case study

Export

In this context we consider a SysML model of the Water and Waste System. Figure 56 shows an *Internal Block Definition (IBD)* representing a subpart of the WWS system comprised of four *Toilet Units* which are SysML *Properties* and are connected to a *Flush Control Unit* (*FCU*) represented by a SysML *Block*. The *FCU* is an element of the WWS managing the synchronization of the different water waste flushes in order to avoid flushes going from a *Toilet Unit* to another one instead of being ejected through the *Wasteline*. This characteristic is very important in aircraft having more than one deck as we will see in the constraints section.

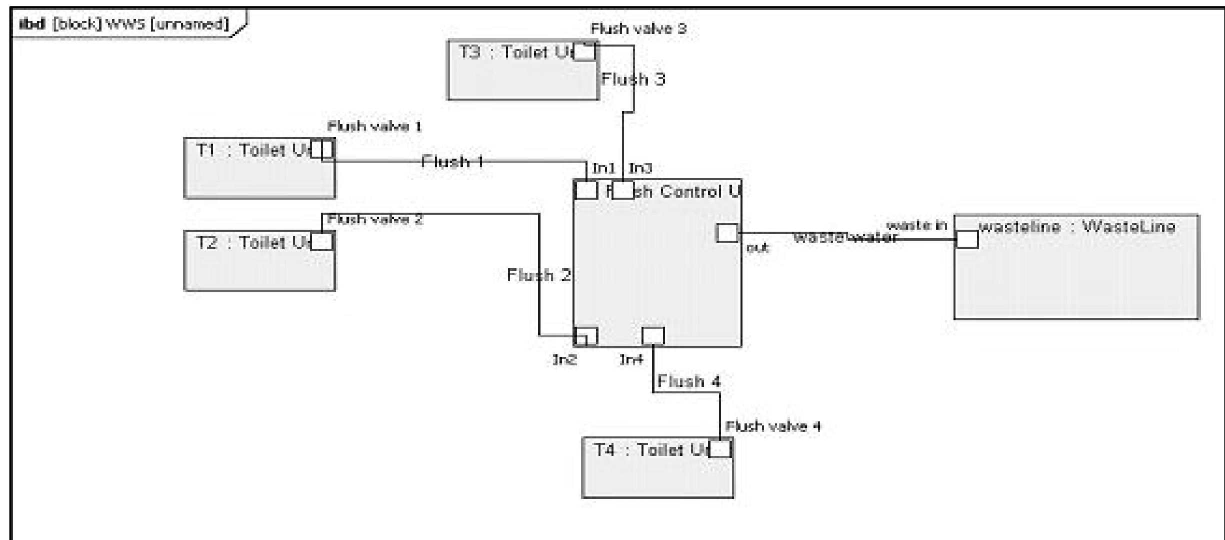


Figure 56. Internal Block Diagram of the WWS SysML model

This model is exported into the common framework by instantiating the SysML meta-model. Part of the instances of this meta-model in EXPRESS are shown in Figure 57.

```
...
#47=PROPERTY_UML(*, $, $, (), $, 'T3', $, .PRIVATE., ...);
#48=PROPERTY_UML(*, $, $, (), $, 'T4', $, .PRIVATE., ...);
#49=PROPERTY_UML(*, $, $, (), $, 'fcu', $, .PRIVATE., ...);
#50=BLOCK(*, $, $, (), $, 'Toilet Unit', $, .PRIVATE., ...,
(#92,#91,#94,#93,#96,#95), ...);
...
```

Figure 57. Instances representing the WWS model in EXPRESS modeling language

Annotation

In our context, knowledge is either technical or general knowledge. Technical knowledge is often formalized by standards. The ATA chapters (ATA, 2011) [2] are an example of common and shared knowledge in Aircraft Systems Engineering domain. This is the reason for us to choose them as a basis to build ontologies in the frame of some of our case studies.

[2] The Air Transport Association (ATA) is an American airline trade association, founded in 1936, whose fundamental purpose is to improve the safety of air transportation. Pursuing this objective ATA has organized an aircraft into a series of systems with general characteristics trough what aircraft engineers call the ATA chapters, some examples: ATA 09 references to "*Towing and Taxing*" in Aircraft General domain; ATA 29 describes the "*Hydraulic Power System*"; ATA 52 discusses about the doors belonging to the structure of the aircraft; and concerning the "*Power Plant*" ATA 79 gives a description of *Oil*.

Concerning the WWS case, we use the description of a *Water and Waste System* given by ATA 38 standard and introduced by the knowledge model (a UML class diagram) of Figure 58.

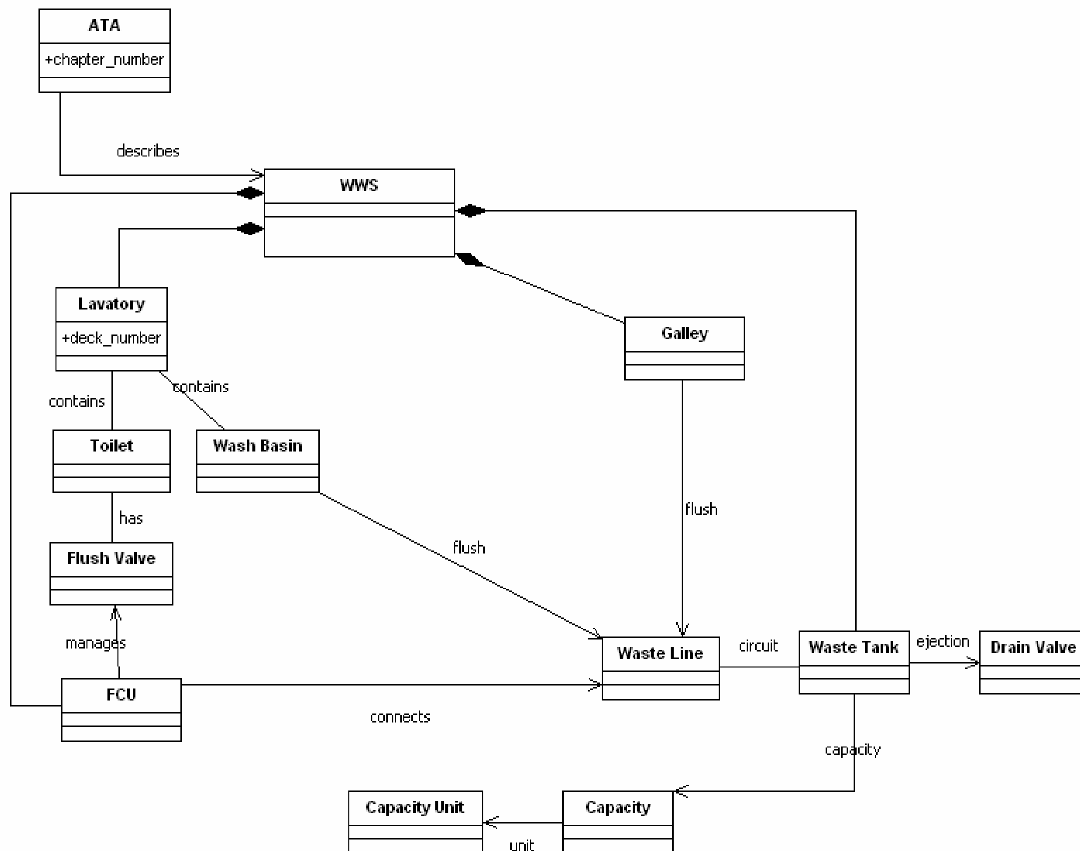


Figure 58. Knowledge model according to ATA 38 architecture

According to this knowledge model, *WWS* consists of some *Lavatories*, some *Galleys*, a *Flush Control Unit* and a *Waste Tank*. Each *Lavatory* is installed in a specific deck and contains a *Toilet* and a *Wash Basin*. The *Wash Basin* is directly connected to the *Waste Line*, whereas a toilet has a *Flush Valve* that is managed by the *FCU* in synchronization with the rest of *Flush Valves*. The *Waste Line* drives wasted water to a *Waste Tank* which has a particular *Capacity*. In the end the wasted flush is ejected from the *Waste Tank* via the *Drain Valve*.

ATA 38 provides a general overview of the architecture of WWS system which is formalized in the above figure. That kind of knowledge model, architecture-oriented, eases the communication with engineers in order to clarify the comprehension of their models. Yet, to construct the constraints introduced in the previous section only some concepts and relations, the ones related to the *Toilets*, are required. Thus, depending on the context of use the knowledge model may be more or less complete; e.g. if the knowledge model is not going to be shared by other design teams a model describing the concepts without the architecture aspects could be enough.

In our implementation, the instances of the ATA 38 knowledge model (see Figure 59) are used to identify the *Toilets* and the *Flush Control Unit* of the *IBD* in Figure 56. These more fine-grained annotations (see Figure 60) enable the validation of the assessment questions regarding flush control.

```
...
#115=URI('http://www.eads.net/systems/wws/wastetank1');
#116=URI('http://www.eads.net/systems/wws/fcu');
#117=URI('http://www.eads.net/systems/wws/flushvalve1');
#119=FCU(*, $, $, #116, #264, (#111,#146,#136,#156));
#121=WASTE_TANK(*, $, $, #115, $, $);
#123=CAPACITY_CLASS(*, $, $, 50., .LITER.);
...
```

Figure 59. Instances of ATA 38 knowledge model in EXPRESS modeling language

```
...
#163=ANNOTATION_CLASS('is', (#128),(#46));
#164=ANNOTATION_CLASS('is', (#149),(#48));
#165=ANNOTATION_CLASS('is', (#139),(#47));
#166=ANNOTATION_CLASS('is', (#116),(#98));
...
```

Figure 60. Instances of annotations using ATA 38 knowledge model

Model Integration

For this case study the model integration activity is not necessary since only one model is involved. Nevertheless, our approach is still viable since, as seen in the previous passage, the model is annotated and we use these annotations to implement the constraints described in next section.

General Constraint Definition

Once the engineer has retrieved one or several candidate models through the global search, he or she still has more detailed questions to assess how much this model fits with the expressed needs. It is a first assessment of an existing model. Concerning the WWS case, one needs to verify properties such as: Are all *Toilet Units* connected to a *Flush Control Unit (FCU)*? At this point, our approach allows performing some more flexible, ad-hoc, annotation of the model internal entities in order to check model properties, which requires that the model is no more a black box and that the modeling language question is considered. Concerning the FCU property we annotate each *Toilet Unit* with its corresponding instance of

Toilet of the knowledge model and the *Flush Control Unit Block* with the instance of *FCU*. Then, a constraint expresses that all the *Toilet Units* (i.e., all the *Properties* of the model annotated by the instance *Toilet* in our knowledge model) are connected to the same element which is a *Block* annotated by *FCU* of our knowledge model. This expression combines elements issued from the SysML meta-model (*Properties, Block*) and from the knowledge base (*Toilet, FCU*), which means that a good understanding of both is necessary. In particular, the SysML meta-model must be explored (our prototype allows it, as it will be presented later on) in order to find and annotate the right elements (*Properties, Blocks*). Figure 61 shows some instances of the *Constraint Relational Model* which implement such constraint.

```
...
/*** Are all Toilet Units connected to a Flush Control Unit?  ***/
#176=ALL_EXPRESSION(*, (#167), (#171), #177);
#177=OR_EXPRESSION(*, (#178,#179));
#178=NOT_EXPRESSION(*, #168);
#179=EXISTS_EXPRESSION(*, (#180), (#182), #184);
...
```

Figure 61. Instances of constraints in EXPRESS modeling language

VI.2.3. Conclusions

In this case study, in first place we enrich the model from a black-box point of view. Therefore, we annotate the model linking it to the particular *Aircraft Program* involved. Amongst the properties of the *Aircraft Program* we find the number of decks, which is the first verification to be carried out; we look for a model describing the Water and Waste System of an aircraft having more than one deck. Once this first filter is applied, we check the most fine-grained properties analyzing the content of the chosen model.

As the constraints of this case study involve only one model they could also have been verified using the features of the source modeling tools. However, this is not an easy task. For instance, in the case of SysML modeling with the Rhapsody tool (IBM, 2012), to express this kind of constraints implies coding Visual Basic for Applications (IBM, 2009) macros. Nevertheless, we mainly implement this case study applying our approach in order to explore the different annotation aspects, i.e. black-box and white-box.

VI.3. Hydraulic and Engine systems models

VI.3.1. Description

Our second case is built upon a real detailed design scenario where two models, respectively, of the Hydraulic system and the Engine system shall be coupled and co-

simulated. The Hydraulic Power system has to produce and carry any type of hydraulic energy up to its consumers. The Engine system is in charge of the mechanical power of an aircraft but it generates the hydraulic flow as well. The hydraulic flow is produced by the means of pumps called *Engine Driven Pumps* (*EDP*).

From this starting point we could identify, based on interviews, the following properties that shall be verified prior to co-simulation integration tasks: 1) to identify in the models the points of connection (interfaces) between the systems; 2) the consistency of the units of measurement of the hydraulic flow involving both models.

VI.3.2. The modeling process activities

Next sections describe the implementation of this case study according to the approach in Figure 62.

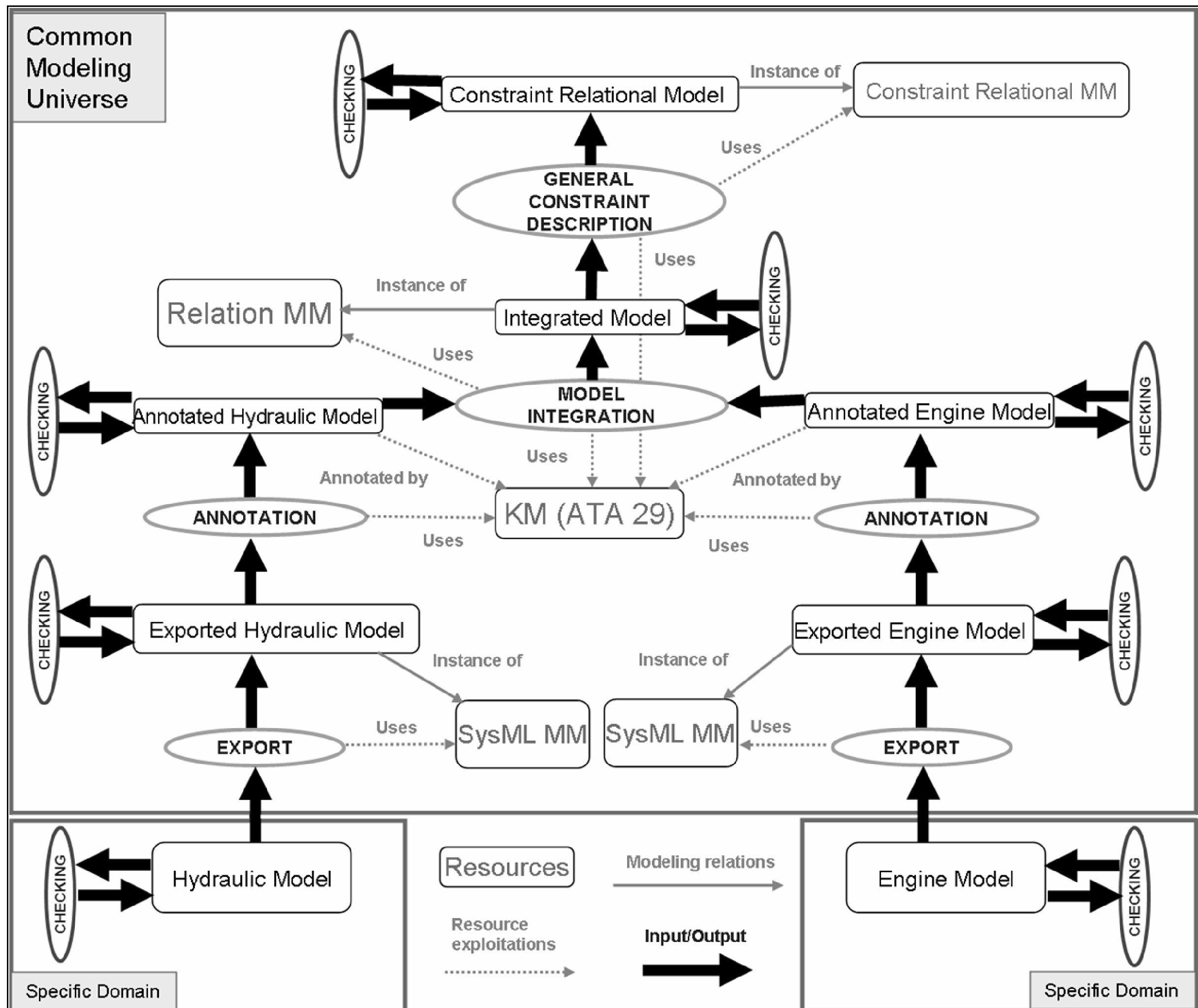


Figure 62. Approach applied to the Hydraulic and Engine case studies

Export

Homogeneous modeling choices

Both models have been re-designed in SysML (one of the available meta-models in our current framework), while the original models were designed using the Simulink modeling language (for which meta-models are not yet available).

We built a first sub-case where the applied modeling principles are identical for both models, which means that subtypes of *Blocks* applying the parts concept are used in both cases to represent subsystems and that SysML *Ports* are consistently employed.

Concerning the SysML models, the engineer co-simulates them by previously connecting points of the Engine to points of the Block which represents the exchange with the Hydraulic system; the Distribution sub-system. Then engineer performs a high level simulation in order to check that the exchanged data is correct. In the models shown in Figure 63 and Figure 64, the connection points are represented by SysML *Ports* in both cases, therefore *in_edp1* has to be connected to *out_pump1* and so on.

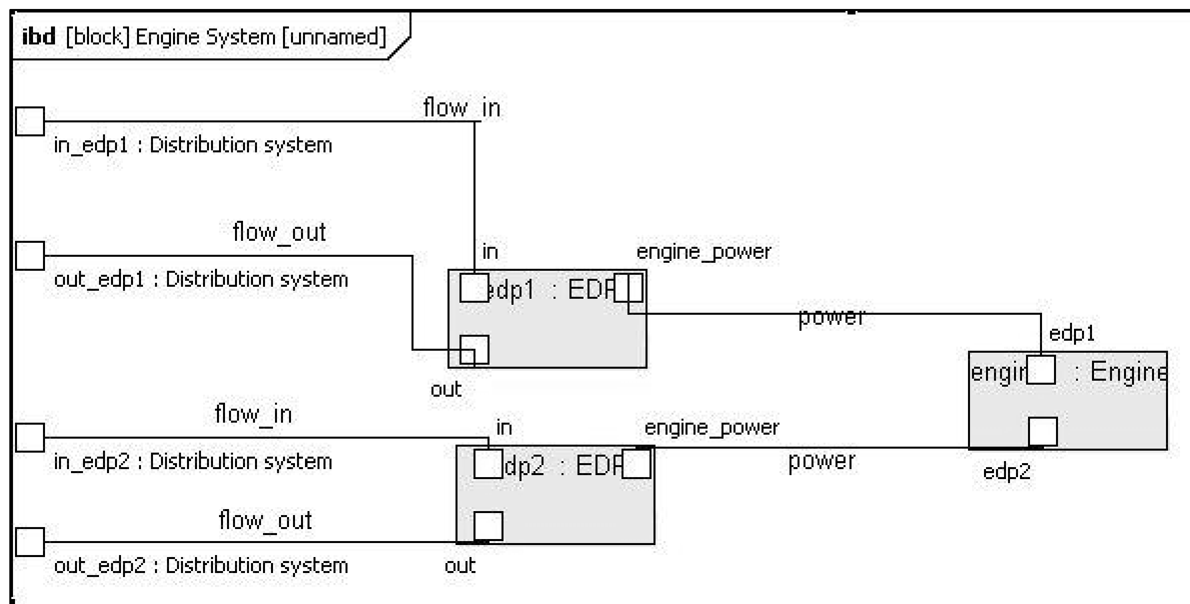


Figure 63. Engine model in SysML

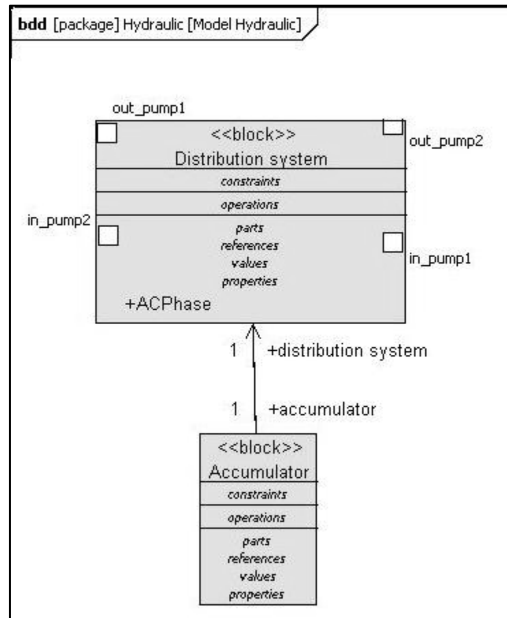


Figure 64. Hydraulic model in SysML

These models are exported into the common framework as instances of the SysML meta-model which are illustrated in Figure 65

```
/** Hydraulic model ***/
...
#142=BLOCK(*, $, $, (), $, 'Distribution system', ...);
#145=PORT(*, $, $, (), $, 'out_pump1', $, .PRIVATE., ...);
#146=PORT(*, $, $, (), $, 'in_pump1', $, .PRIVATE...);
...
/** Engine model ***/
...
#58=PROPERTY_UML(*, $, $, (), $, 'engine1', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, #158, .F., $, .COMPOSITE., ...);
#60=PORT(*, $, $, (), $, 'Port1', $, .PRIVATE., ...);
#62=PORT(*, $, $, (), $, 'in_edp1', $, .PRIVATE., ...);
...
```

Figure 65. Instances representing Hydraulic and Engine models in EXPRESS modeling language

Heterogeneous modeling choices

Keeping the same context and for demonstration purpose, we built a second sub-case where different modeling rules are applied for the two models. As we already outlined, such situations actually arise in real practices. Keeping the Hydraulic System model unchanged, we introduce an alternative representation of the Engine model as shown in Figure 66. The connection points are designed as *Blocks*: *VALVE_IN* and *VALVE_OUT*. These valves belong to one of the two *EDPs* of each *Engine*, represented as *edp1* and *edp2* connections.

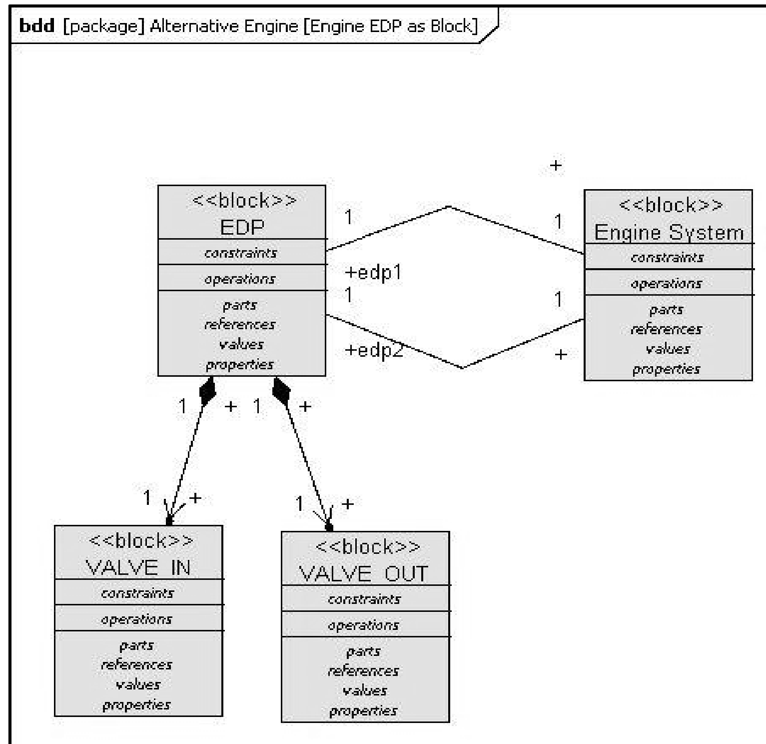


Figure 66. Alternative SysML Engine model

Thus, some modeling heterogeneity arises from the fact that on one hand the Hydraulic model describes the connection points as SysML *Ports* and on the other hand the alternative Engine model represents them in terms of SysML *Blocks*. Figure 67 shows part of the instances of this alternative SysML Engine model.

```
...
#16=BLOCK(*, $, $, $, $, 'EDP', ...);
#17=BLOCK(*, $, $, $, $, 'VALVE_IN', ...);
#18=BLOCK(*, $, $, $, $, 'VALVE_OUT', ...);
#19=BLOCK(*, $, $, $, $, 'Engine System', ...);
...
```

Figure 67. Instances of the alternative Engine model in EXPRESS modeling language

Annotation

For the need of verifying the constraints, an ontology describing the kind of ports (in/out), and another one for the related pump and its hydraulic flow (value, units, pressure…) are necessary. Main elements of the ontology are extracted from the ATA 29[3] documentation. Figure 68 represents the knowledge model of the main concepts of this chapter.

[3] ATA 29 is the chapter which describes the Hydralic Power System, i.e. the system that have to produce and carry any type of energy up to its consumers using several means such as: mechanical mediums, electrical mediums and fluid mediums. Aeronautic industries principally use hydraulic fluid under pressure to provide energy from a power source to consumers.

For the interest of our cases the important concepts are the *Engine Driven Pumps* (*EDP*) located in the *Engine*. Each *Pump* is connected to the Hydraulic System via two valves: "*Valve IN*" and "*Valve OUT*". One *Pump* provides the Hydraulic System with a *Flow* produced in some particular conditions of *Pressure* and *Frequency*.

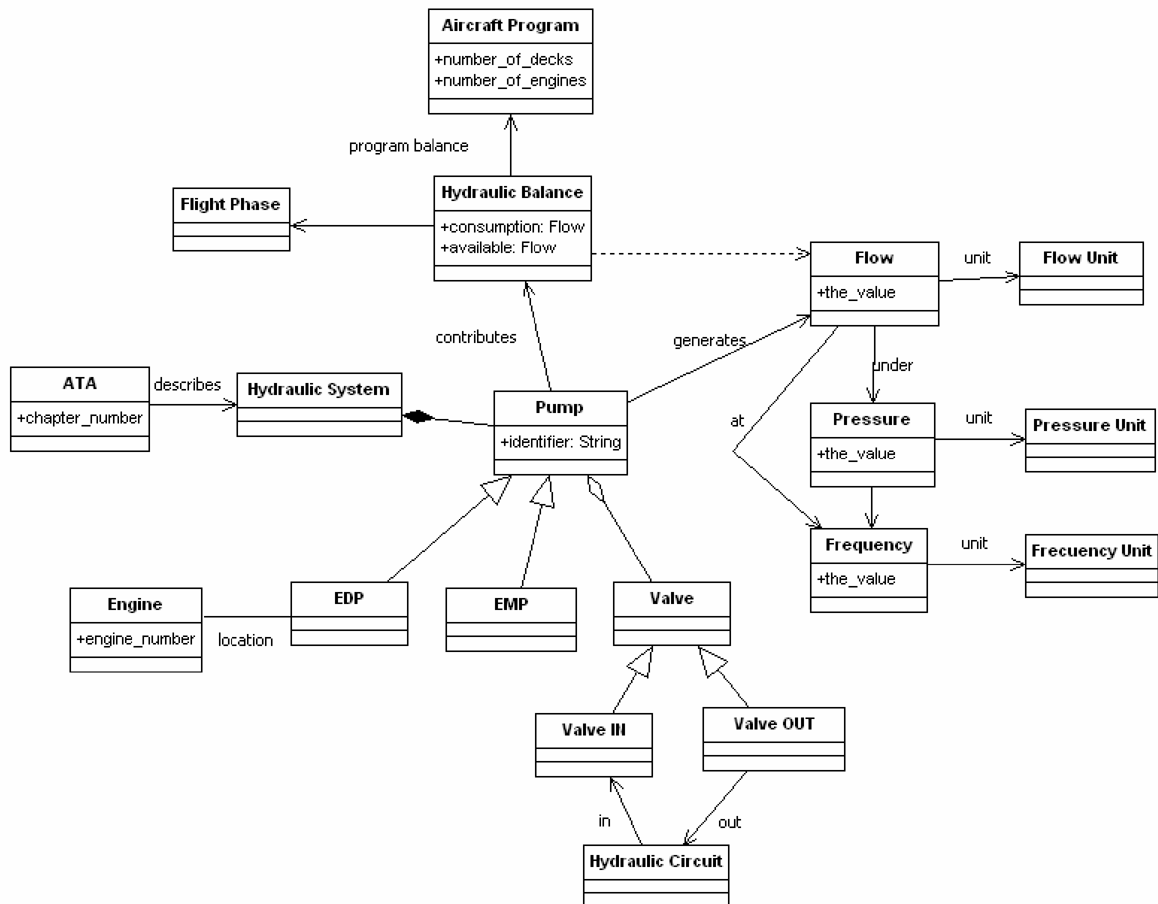


Figure 68. Knowledge model according to ATA 29

The instances (see part of them in Figure 69) of this knowledge model are used as follows:

- Hydraulic model. Each P*ort* is linked to an instance of *Valve* depending whether it is an input or an output.
- Engine model with same modeling semantics. Every *Port* having *flow_in* or *flow_out* connections corresponds to a kind of *Valve*.
- Engine model with different modeling rules. In this case the link is carried out between *VALVE_IN* and *VALVE_OUT Blocks* and, respectively, the concepts "*Valve IN*" and "*Valve OUT*" of the knowledge base. This annotation takes into consideration the instance of *EDP* owner of the valves.

```
...
#372=URI('http://www.eads.net/systems/hydraulic/valve11');
#373=URI('http://www.eads.net/systems/hydraulic/valve12');
#374=VALVE_IN(*, $, $, #372);
#375=VALVE_OUT(*, $, $, #373);
#376=EDP(*, $, $, #382, #328, (#375,#374), 'edp1', $);
...
```

Figure 69. Instances of the ATA 29 knowledge model in EXPRESS modeling language

Thus, the annotations shown in Figure 70 support the identification of equivalences between elements of the models, e.g. *Port* in_edp1 (or the instance of *Block VALVE_IN* using *edp1* connection in the alternative model) in the Engine model and *Port* in_Pump1 in the Hydraulic model are equivalent because they are annotated by the same instance of *VALVE_IN*. Furthermore, in the knowledge model a *Pump* generates a *Flow* with specific properties and units of measurement that we use to guarantee that flow data units are consistent with the content of the models.

```
...
/* in1 */
#370=ANNOTATION_CLASS('valve', (#372), (#146));
#371=ANNOTATION_CLASS('valve', (#372), (#17,#4));
/* out1 */
#390=ANNOTATION_CLASS('valve', (#373), (#18,#4));
#391=ANNOTATION_CLASS('valve', (#373), (#145));
...
```

Figure 70. Instances of the annotations using the ATA 29 knowledge model

Model Integration

The integration in this case study is guaranteed by the annotation of the connection points, as explained in next section. Thus, we identify two ports as equivalent when they are linked to the same knowledge concept.

General Constraint Definition

The goal of the verifications in this scenario is twofold:

- To identify the connection points in order to correctly connect them. This identification is done by means of the annotations.
- To guarantee that the linked ports are compatible: same flow unit and same conditions of flow production, i.e. frequency and pressure units. This verification is implemented as a constraint whose instances are shown in Figure 71.

```
...
/*  linked  elements  are  compatible:  same  flow  unit  and  same
conditions of flow production */
/* all ports from hydraulic */
#400=ENTITY_VARIABLE(#146, 'p');
#401=VARIABLE_DOMAIN(#402, #400);
#402=ENTITY_DOMAIN((#145,#146,#147,#148));
#403=ALL_EXPRESSION(*, (#400), (#401), #407);
...
```

Figure 71. Instances implementing the constraints in EXPRESS modeling language

VI.3.3. Conclusions

This combination of cases allows us to demonstrate that the applied knowledge model depends on the constraint to be validated even whether the models representing the same system, i.e. Engine, are expressed using different modeling semantics, i.e. *Ports* versus *Blocks*. Actually, the application of knowledge models removes the heterogeneity related to the chosen modeling rules.

In this case study, we identify the connection points between the systems by annotating elements of the models with the common *Valve* concept of the knowledge base. In this way, departing from the concept of the knowledge base, we can found which elements of the exported models are annotated by the same piece of knowledge, i.e. they are equivalent. This feature originates a new requirement for our prototype: users must be able to navigate the knowledge model and its annotations.

At the same time, we must check that the flow between these equivalent interfaces is using the same units of measurement. Our knowledge base contains information of the attended units of measurement for the *Flow* associated to a particular *Pump* and we need to guarantee the consistency regarding the models to be connected during the co-simulation. Thus we annotate the *Distribution System Block* of the Hydraulic model and the *edp1* and *edp2 Ports* of the Engine model (*EDP Block* for the alternative one) with their corresponding flow characteristics (units, pressure and frequency). Finally, our constraint solver checks that this information is consistent, i.e. that the units of flow, pressure and frequency are the same than the ones declared in the knowledge base for the *Pumps* corresponding to *edp1* and *edp2* (*EDP* in the alternative model).

VI.4. Ram Air Turbine models

VI.4.1. Description

The Ram Air Turbine (RAT) system provides power to other systems in case of emergency. The RAT is an aircraft electrical generation system which powers the essential bus bar when there is a total loss of hydraulic and electric power or a total loss of electric power in flight. Therefore, the RAT has to provide electricity to a minimum set of systems that are absolutely necessary to land the aircraft, mainly: Slats, Fuel Pumps, Windshield Anti-Ice and Probes Anti-Ice.

These consumer systems have different power consumption needs depending on the flight characteristics, mainly related to the speed of the aircraft. Thus, different configuration scenarios are needed to estimate the power needs of each system and the requirement for the RAT. For instance, when aircraft speed is less than 140kts, the sum of systems' consumption is 29kVA, that means that RAT must provide at least this value.

Native heterogeneity of models

The RAT and the aforementioned related systems have their own functional design models, each one describing the emergency scenarios and the power generation (in the case of the RAT) or the power consumption (for the systems) in every situation. However, some models explicitly refer to the speed of the aircraft whereas other models refer to flight phases (which are implicitly characterized by speed values amongst other properties).

Starting from these real modeling circumstances, we focus on two models describing the RAT and the Slats systems respectively, in order to check that for any configuration scenario the power provided by the RAT is greater than the energy value demanded by the Slats. Based on their specifications we develop two simplified versions of these systems' models to be imported into our platform using two different modeling languages. To reflect the heterogeneity of models, the case study RAT model is developed using CORE language and the Slats model uses SysML.

VI.4.2. The modeling process activities

Next sections develop the implementation of this case study according to the models and activities shown in Figure 72.

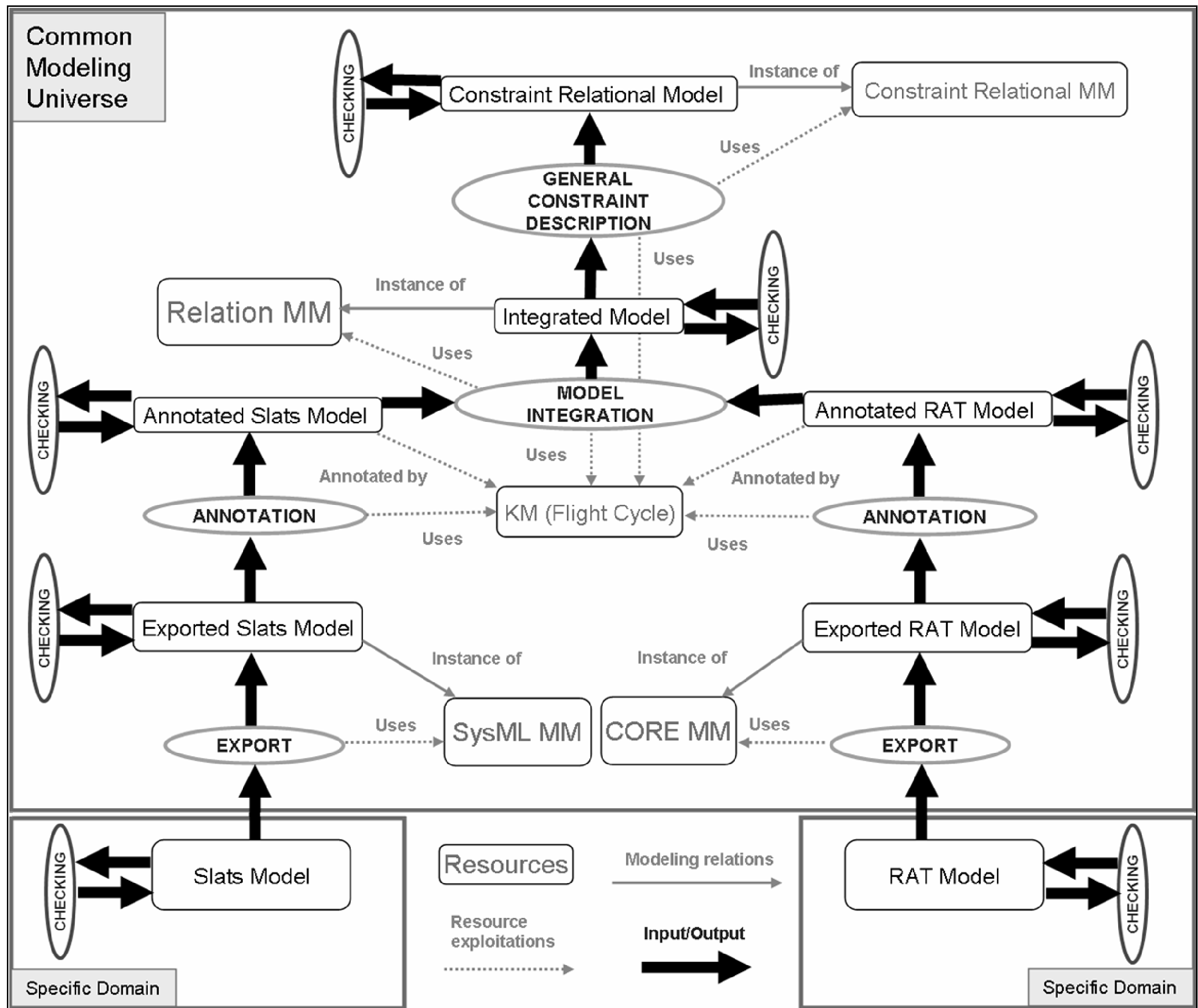


Figure 72. Approach applied to the RAT case study

Export

Figure 73 shows an *eFFBD* of the CORE model of the RAT. It describes the "*RAT extension*" operation, by means of four *Functions*, each one representing an alternative flight scenario and, in consequence, a different power generation. The value of the provided power is modeled as a *Resource* called "*Load Capability*" to which each *Function* gives a different amount, e.g. "*Landing Power Generation*" function produces the resource with a value of 9,5kVA as shown in Figure 74. An extract of the instances of the CORE meta-model, result of the export of the model in the common framework, is shown in Figure 75.

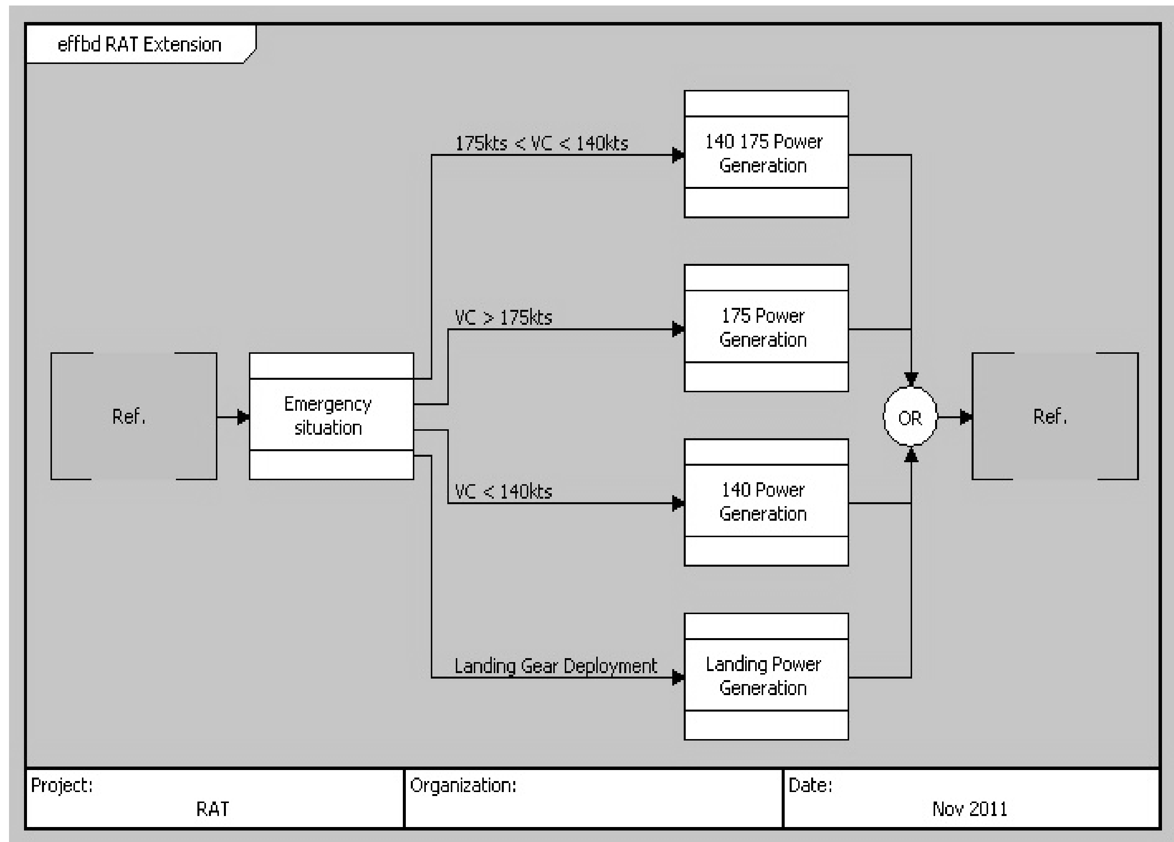


Figure 73. eFFBD diagram of RAT power generation functions

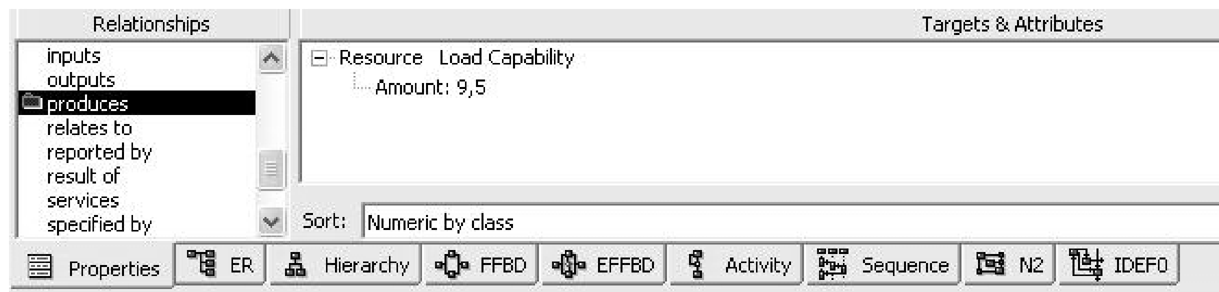


Figure 74. Relationship with the Resource and its value

```
...
#44=FUNCTION_CORE(*,$,$, #70, 'MS', 'Landing Power Generation'...);
#45=FUNCTION_CORE(*,$,$, #70, 'MS', '140 Power Generation'...);
#46=FUNCTION_CORE(*,$,$, #70, 'MS', '175 Power Generation'...);
#47=PRODUCES_RELATION($, $, $, #41, CONSTANT_CORE(42.));
...
```

Figure 75. Instances representing the RAT model in EXPRESS modeling language

One of the impacted systems during the RAT extension is the Slats system. Figure 76 shows a SysML *State Machine* describing the load needed by Slats in different flight phases. In each state an *Activity* called “*power consumption*” is performed. This *Activity* has an

attribute which is a *Flow Property* containing the related load quantity, e.g. during Climb subphase the load is 33kVA. Figure 77 shows an excerpt of the instances of the SysML meta-model as a result of the export activity.

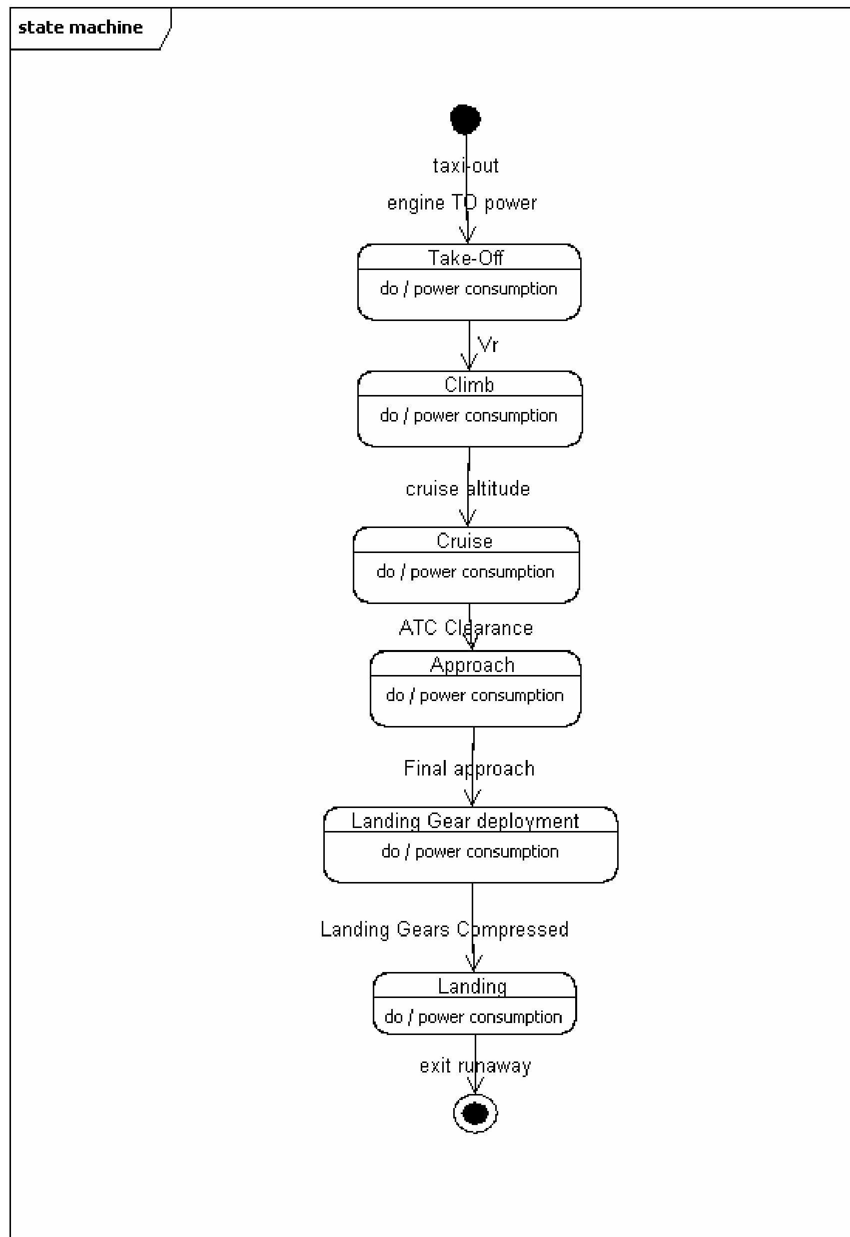


Figure 76. State machine of Slats consumption during flight

```
...
#17=ACTIVITY(*, $, $, $, $, 'power consumption',...);
#18=ACTIVITY(*, $, $, $, $, 'power consumption',...);
#19=STATE(*, $, $, $, $, 'Take-Off', ...);
#20=STATE(*, $, $, $, $, 'Climb', ...);
...
```

Figure 77. Instances representing the Slats model in EXPRESS modeling language

Annotation

To validate these models, we must be able to correctly identify the flight scenarios in each model. That is the objective of the knowledge model described below.

Flight phases include in fact both *Flight* and *Ground phases*. Even though anybody could say that an aircraft takes-off, flies and lands, it is actually not easy to find a common and agreed definition of the different phases. Clearly, every discipline involved in the design of an aircraft makes use of information related to *Flight phase:* for example the Hydraulic System regulates the flow according to the phase; the granted communications are not the same when the aircraft is on ground or in flight, etc. Not surprisingly, we find different ways to represent this information into models. In some cases, a phase is represented as a combination of speed, altitude and some other parameters, whereas in other cases it is represented by a simple code identifying it. Thus, this is a source of heterogeneity and difficulties for sharing the models. An ontology giving an agreed understanding of the *Flight phase* concepts is then necessary.

Figure 78 below, describes general knowledge about *Flight phases*. It is inspired from an internal document of the aircraft manufacturer which is approved to be shared. The aim of this knowledge model is to homogenize the different ways of describing the emergency scenarios in our source models.

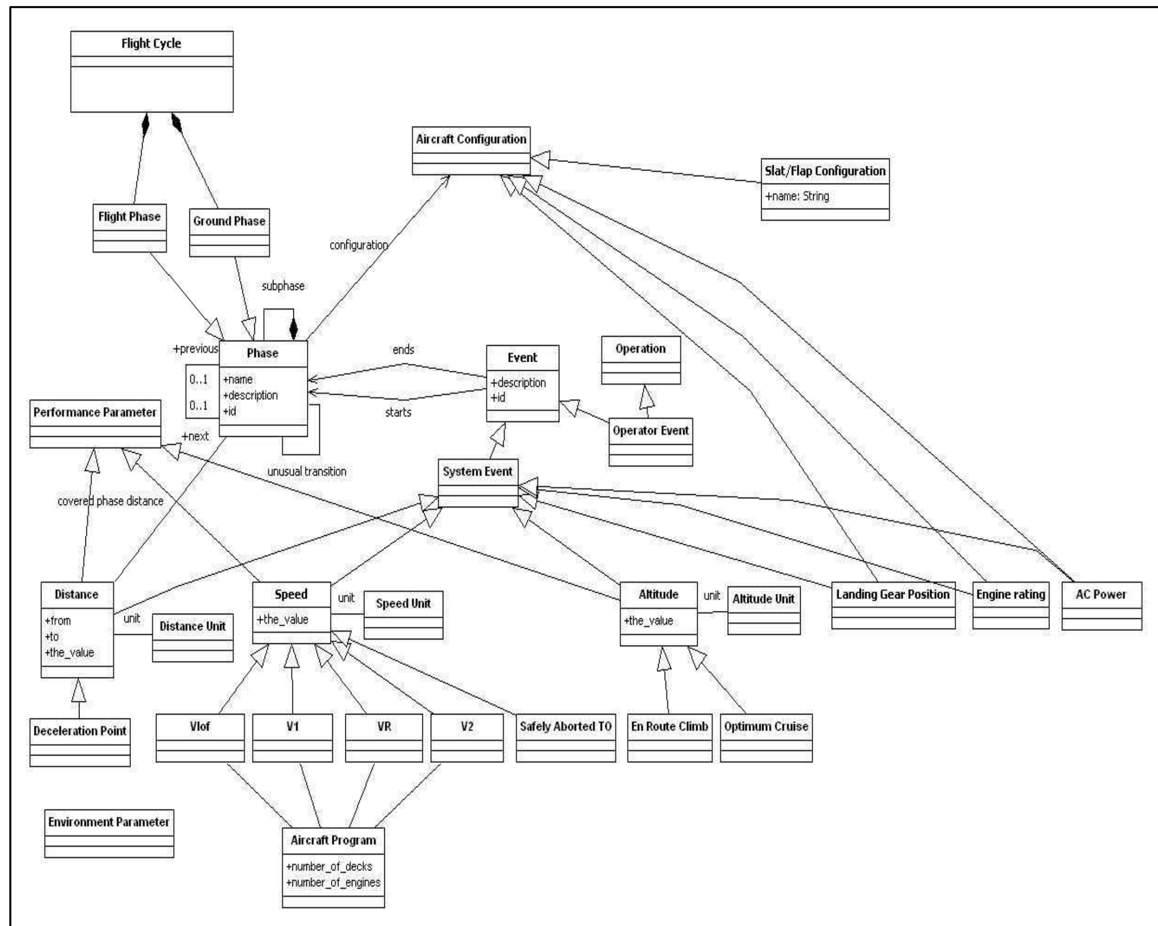


Figure 78. Knowledge model of the Flight Cycle

A *Flight Cycle* is composed of *Phases*, *Ground* or *Flight*, which start and end at the triggering of some events. These events can be an *Operator Event*, i.e. with human (operator) participation, or a *System Event*. *System Events* are related to *Performance Parameters*, *Aircraft Configuration* and *Environment Parameters*. Actually, some of these system properties, as for example the *Landing Gear Position*, complete the specification of a *Phase* as well. Moreover, most of the *Performance Parameters* values depend on the aircraft program, e.g. the categories of speed. Some instances of this knowledge model are represented in Figure 79.

```
...
#312=URI('http://www.eads.net/A350/FlightCycle/Derot');
#313=URI('http://www.eads.net/A350/FlightCycle/Brake');
#334=FLIGHT_PHASE(*, $, $, #312, 'derot', 'derot subphase', ...);
#335=FLIGHT_PHASE(*, $, $, #313, 'brake', 'brake subphase', ...);
...
```

Figure 79. Instances of the Flight Cycle knowledge model in EXPRESS modeling language

Using the instances of this knowledge model, the annotation procedure (see annotation instances in Figure 80) varies for each model:

- RAT model. By interpreting the information contained in the conditional exit branches of the *eFFBD,* a mapping between the CORE Function and the corresponding *Phase* of the knowledge base is defined. For instance, for the conditional exit branch named "*VC > 175kts*" the function "*175 Power Generation*" is assigned to the instance of *Phase* called *Cruise*.

- Slats model. In this case the annotation is more obvious since the *States* of the SysML *StateChart* in Figure 76 represent phases. Nevertheless the name and the granularity of such *States* are slightly different to the representation of phases in our knowledge base. Thus, for example, "*Landing*" *State* is assigned to the "*Final Approach Phase"* knowledge instance.

```
...
/* taxi-out */
#370=ANNOTATION_CLASS('phase', (#314), (#45,#25));
/* take-off */
#371=ANNOTATION_CLASS('phase', (#305), (#45,#19));
/* Initial Climb */
#372=ANNOTATION_CLASS('phase', (#303), (#43,#20));
/* En route climb */
#373=ANNOTATION_CLASS('phase', (#304), (#43,#20));
...
```

Figure 80. Instances of annotations using the Flight Cycle knowledge model

In this way both models refer to the same phases, i.e. instances of the same *Flight Cycle* knowledge model. The validation of the consumption constraint can then be carried out.

Model Integration

In this case study, the equivalent flight phases are annotated with the same knowledge concepts. This permits the integration of both models which is necessary to implement the constraint described in next section

General Constraint Definition

Using the above described models of RAT and other systems, we want to verify that "*RAT load capability > (Slats power consumption + other systems consumption)*" for each significant flight configuration. Some of the instances implementing such a constraint are shown in Figure 81. As explained in V.5, our expressions model is an extension of the PLIB expressions language proposal incorporating FOL expressions. That means that besides the FOL expressions and other boolean expressions, numeric and string expressions are also available. For example, in the case of the RAT constraint numeric expressions are used to calculate the "*(Slats power consumption + other systems consumption)"* part.

```
...
/* "RAT load capability > (Slats power consumption + other systems consumption)" for each significant flight configuration */
/*for all functions*/
#500=ENTITY_VARIABLE($, 'f');
#501=VARIABLE_DOMAIN(#502, #500);
#502=ENTITY_DOMAIN((#42,#43,#44,#45,#46));
#503=ALL_EXPRESSION(*, (#500), (#501), #507);
/*for all states*/
#504=ENTITY_VARIABLE($, 's');
#505=VARIABLE_DOMAIN(#506, #504);
#506=ENTITY_DOMAIN((#19,#20,#21,#22,#23,#24,#25,#26));
#507=ALL_EXPRESSION(*, (#504), (#505), #508);
...
```

Figure 81. Instances implementing the constraint in EXPRESS modeling language

VI.4.3. Conclusions

The native heterogeneity is surpassed thanks to the fact that SysML and CORE meta-models are written in the same shared modeling language. This allows us to use the meta-meta-models features to browse the different meta-models in a common framework and to annotate them independently of their heterogeneous nature.

The annotations are done based on a more complex knowledge model. By linking the phases to the *Functions* in the RAT model and the *States* in the Slats model we homogenize the description of scenarios to be taken into account. The constraint is represented by an expression the exactness of which can be verified; stating that for each *Phase* of the knowledge model, the power provided by the corresponding *Function* in CORE is greater than the energy demanded by the equivalent *State* in SysML. Once again the content of models is analyzed in detail before annotating; nonetheless, such a rich knowledge model as the one described here may be completed with domain rules in order to support and ease the annotation process, e.g. a domain rule that giving a range of speeds ([175kts, 140kts]) suggests the appropriated *Flight Phase* to be used in the annotation.

VI.5. Conclusion

In this chapter we have described the application of our approach with different case studies. The variability of these case studies in terms of number of models, modeling languages and modeling rules demonstrates that our approach is generic. From a modeling life cycle point of view, the proposed approach can be used at different stages:

- Before the development of source models. For example, using the annotations to search for previous models when starting a new aircraft program as in the case of WWS black-box perspective.
- During the development of source models. For example, to check inter-model constraints in order to correct inconsistency issues improving models quality and contributing to their maturity. That is the scenario of Hydraulic and Engine and RAT case studies.
- After the development of source models. For example, when an existing program is modified and the inter-model constraints need to be re-verified. This scenario is connected to the needed improvements in the management of inter-model constraints (reuse) listed further on.

Nevertheless, EXPRESS models and instances representing the different case studies have been built manually, which is time-demanding. Currently, the time needed to validate a constraint with the approach is greater than the time employed in actual consistency checking meetings. On the one hand, annotations are done a posteriori, i.e. after the construction of the source models. Together with the fact that the building of ontologies is a difficult task since a lot of information needs to be gathered and a consensus about the formalization of the knowledge is necessary but complicated. On the other hand the implemented constraints refer to both domain and modeling semantics, which means that a good knowledge of the involved source modeling languages is needed to be able to express a constraint.

Thus, the approach has demonstrated its value as a formal method to:

1) make explicit the implicit knowledge in order to annotate the source models
2) use this explicit knowledge to integrate heterogeneous models and to validate inter-model constraints
3) maintain the independence of source models by exporting them using EXPRESS as an unified and shared modeling language

Nevertheless, to reach a successful industrialization some improvements are needed

1) Maximal automation of the approach activities (export, integration…)
2) Annotations coordinated with the modeling process (a priori)
3) Intermediate abstraction level for modeling semantics in order to get focus on domain semantics for the expression of constraints
4) Reuse and visual supporting to efficiently build constraints

These improvements are discussed in Chapter VIII.

Chapter VII — Prototyping tool

Summary

VII.1. A prototype to support the method 115

VII.2. Actors and use cases 116

VII.2.1. Actors 116

VII.2.2. Configuration use cases 116

VII.2.3. Operational use cases 117

VII.3. Selected technology and architecture 119

VII.4. Current HCI (Human Computer Interface) 121

VII.5. Conclusion 125

Abstract. A prototyping tool has been developed during the deployment of the industrial case studies. The objective of this prototype is to provide engineers with a tool supporting the management of the concepts of the approach from a process point of view. The tool covers the identified needs and the usability conclusions of the industrial evaluation. Next step is to lead this beta version of the prototype to a more robust version adapted to industrialization and including some improvements in the graphical user interface.

VII.1. A prototype to support the method

During the industrial evaluation of the approach, we have developed a prototype. The prototype takes into consideration the engineers' point of view, supports the proposed process and is in line with the conclusions issued from the different industrial cases. The users' needs for this tool are summarized below.

- **The prototype must assist the user in the application of the approach**. This is the main objective of building such a prototype. A user must be able: to create a meta-model; to import a model; to define and populate a knowledge model; to annotate entities of the models with concepts of the knowledge models; to build constraints using expressions; and to check the constraints and get their results.

- **The prototype must be user friendly**. The user must be guided during the manipulation of the prototype which must be intuitive and easy-to-use with a graphical environment and support. It shall also include the construction of the expression associated to a constraint.

- **The prototype must allow users to navigate through the entities and concepts of the different involved models**. The content of meta-models, the instances of these meta-models, the knowledge models and the annotations shall be accessible and navigable.

- **The prototype must allow users to easily analyze the results of a constraint evaluation**. Traceability of the execution performed for the verification of a constraint is required in order to enable the identification of the entities which do not fulfill a particular constraint.

- **The prototype must be modular and extendible**. The architecture of the prototype must anticipate the possibility of adding new functions or replacing some of the modules, e.g. the case of requirements to be verified which need a specific expression model and constraint solver, the expression module will be replaced.

- **The prototype must be a light application**. The tool has to be powerful enough in terms of computation, besides of being easily installable in computers of users.

- **A trace of the checked constraints and the results of their evaluations must be recorded**. Each execution of a requirement checking needs to be stored in order to

be able to compare results and to provide an overview of the evolution of the models. Concerning this latest point, storing the results allows the users to know either that a previous checking error is connected or that a formerly correct property is no longer fulfilled (regression problems).

- **Request over meta-data or data of the models should be supported**. The prototype must allow users to provide relevant meta-data (see next section) about the imported models, the knowledge models, the annotations and the constraints in order to enable requests over it. These requests are another aspect of the validation since they give additional information and define a first filter of validations before writing a complex constraint.

VII.2. Actors and use cases

The user requirements analysis derives in a series of use cases and actors which are described in next sections.

VII.2.1. Actors

Several user profiles are defined taking into consideration the skills needed to perform the different use cases.

- **The tool expert**. It is the user expert in technical details of the prototype who gives support to other users and manages the prototype configuration.

- **The engineer in charge of design**. This user manages the modeling process and is the person in charge of ensuring the constraints satisfaction involving various models. He is the main user of the prototype.

- **The knowledge engineer**. The actor providing the knowledge models or formalizing them. He is the person in charge of knowledge management and of the establishment of the rules to create new knowledge concepts.

- **The engineer in charge of a model**. He or she is the expert of one particular model, imported into the tool. He provides the annotation of this model or assists the engineer in charge of design to perform this annotation.

VII.2.2. Configuration use cases

Concerning the use cases, firstly, the tool must allow some configuration tasks.

- Construction of meta-models
- Load of models
- Construction of knowledge models
- Definition of relation meta-model
- Formalization of the expression to validate the constraint

Figure 82 shows these configuration use cases and their associations with actors.

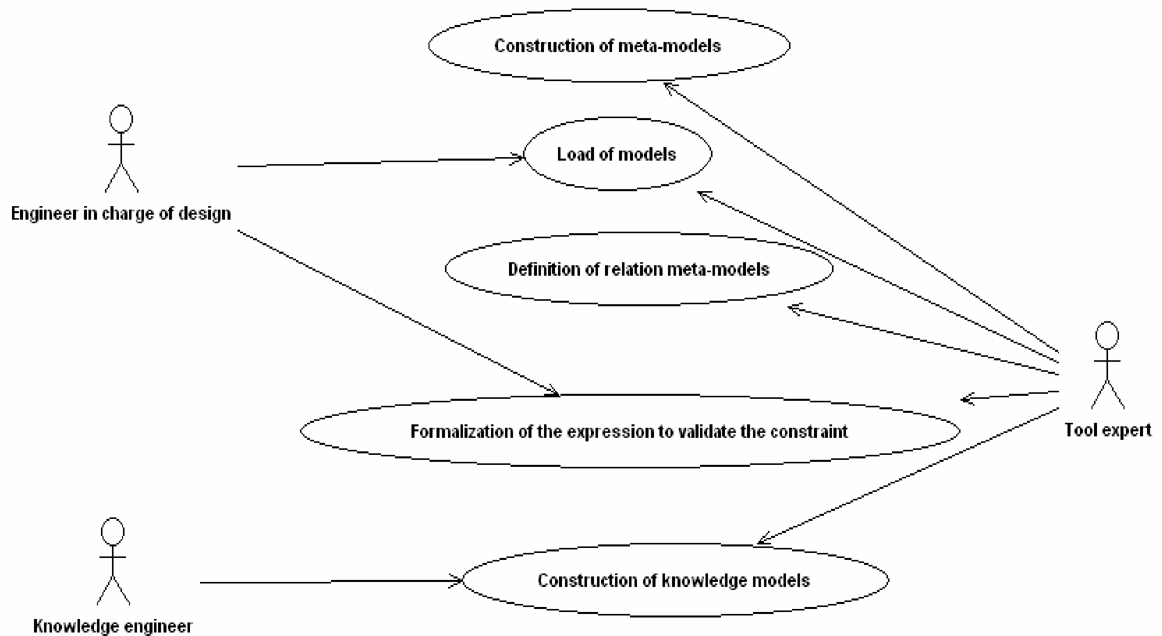


Figure 82. Configuration use cases

VII.2.3. Operational use cases

Once all the configuration pieces are set, users (see Figure 83) can complete the operational tasks.

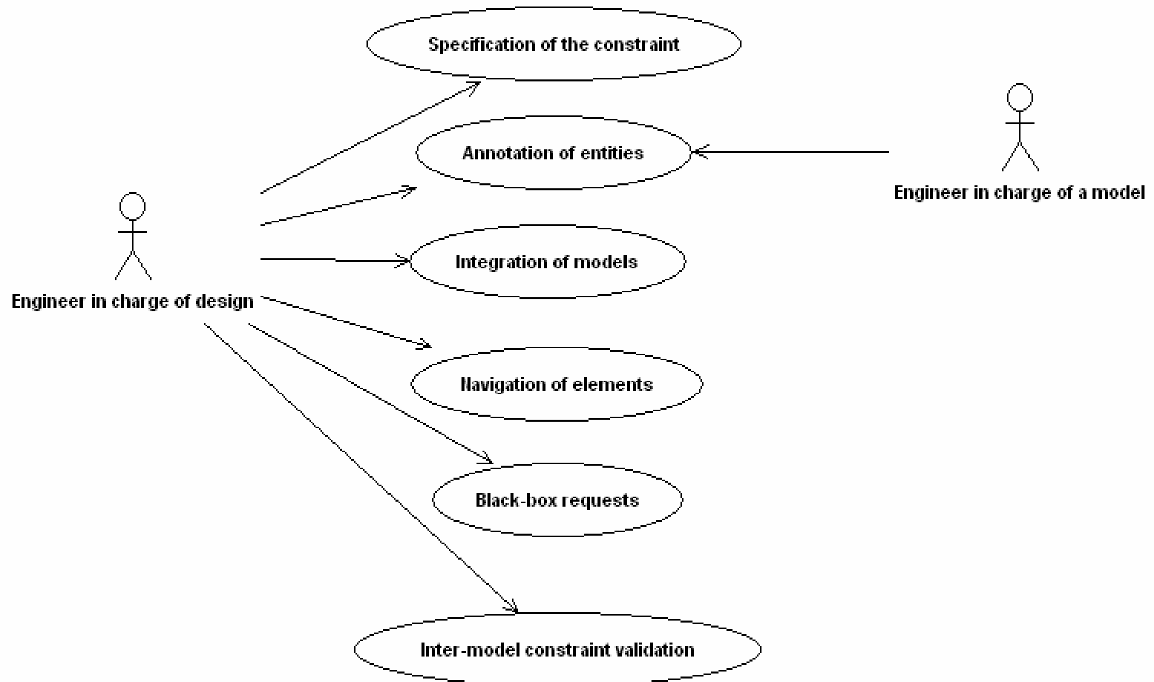


Figure 83. Operational use cases

Specification of the constraints

The specification of the property to be validated is the starting point of the process. This specification contains some of the meta-data described previously (origin, category, model level and property language) which is used to guide the designer during the properties verification process.

Annotation of entities

The **engineer in charge of design**, with the assistance of the **engineers in charge of models**, enriches the exported models using the knowledge explicitly described in the knowledge models.

Integration of models

The **engineer in charge of design** uses the relation meta-model to interconnect the annotated models.

Navigation of elements

The fact of working in the same universe enables the navigability of models elements, annotations and knowledge concepts.

Inter-model constraint validation

The main operational task is to allow **the engineer in charge of design** to validate an inter-model property over the instances of the models and to show the results of this validation in order to be analyzed.

Black-box requests

Furthermore, other complementary tasks can be performed thanks to all the information gathered by our method: to perform requests obtaining all the constraints of a category; to get the history of validated constraints; to show the traceability of inter-model constraints to the source models and requirements and so on.

VII.3. Selected technology and architecture

In the context of a preliminary design, a prototype is developed in order to implement the use cases that fulfill the user needs. The framework of development is Eclipse IDE (Eclipse.org, 2011a) since its architecture is modular. It supports the development of light applications and it is open source.

Two user interface technologies were evaluated before beginning the construction of the prototype. Firstly, we considered the development of the prototype as an Eclipse plug-in that uses the internal graphical elements of the Eclipse tool. The main drawbacks of this solution are the lack of graphical support for the construction of the screens and, mainly, the heavy load of dependencies. Actually, a plug-in Eclipse needs a lot of additional libraries which are useless in our context. Considering this situation, an alternative graphical-oriented solution was evaluated and finally accepted as the basis of the development. We chose the SWT toolkit (Eclipse.org, 2011b) in order to implement the different screens of the prototype. SWT is a complete graphical library of Java and includes an Eclipse plug-in allowing the management of the different graphical widgets. It is easy to use and it enables a quick development of the screens.

From our point of view, the right selection of the elements to build the user interface is the first step to carry out when developing a prototype. The main purpose of such a tool is to validate the approach with the users and they need to be guided in a user-friendly way. The rest of technological bricks come from the operational validation described in V.6 completed with some integration tests with Eclipse. In order to better manage all the architecture choices and according to the specified needs, the prototype is built around a modular architecture with the following components.

- **User interface module**. This module contains the packages controlling the user interface. As described previously, this module is centered in the SWT toolkit and the screens developed using this technology.

- **Meta-model module**. The aim of this module is to provide the functionality concerning the meta-models. The packages of this module load and manage the meta-models necessary for the approach.

- **Model module**. The main goal of this group of packages is to enable the importation of the models according to their meta-models and to browse of their entities, their attributes and their relations.

- **Knowledge module**. This module is the interface to the knowledge management feature. It targets both manipulating the knowledge models and instances and performing the reasoning (not implemented in the current version of the prototype but conceivable in cases of complex knowledge rules).

- **Annotation module**. As the annotation is the relation between the knowledge concepts and the entities of the imported elements, we have decided to promote this link as an independent module. Actually, treating the annotation as an independent part is consistent with the idea of keeping the traceability to the source models and we wanted to reflect this separation.

- **Integration module**. This module is in charge of the functions related to the integration of the annotated models. It manages the integrated model.

- **Constraint module**. This module manages the expression models (FOL is the implementation in the version of the prototype) and contains the engine for validating the constraints. Following the results of the operational validation stage and in order to rapidly obtain a beta version of the prototype, we use EXPRESS as a constraint solver. Nevertheless, in the case of Eclipse a new plug-in called JSDAI (GmbH, 2012) seemed to be a good candidate to be included in this module. JSDAI is an Eclipse plug-in which supports EXPRESS models and validation of their instances (as the ECCO Toolkit). The main advantage of JSDAI is that it is already integrated in Eclipse due to its nature. Unfortunately, the maturity of the plug-in did not entirely satisfy our needs. There was an important gap in the area of integrated validation of constraints which is one of our main reasons for using EXPRESS. Actually, at the moment of the evaluation, there was not any mean of obtaining the result of the checking of a constraint. Our prototype would have been useless. Therefore, the final solution to this problem is to develop an API to integrate Eclipse with the ECCO toolkit in a flexible way so that a future more evolved version of JSDAI can replace it.

VII.4. Current HCI (Human Computer Interface)

The prototype is process-oriented, i.e. it guides the user in the implementation of the different steps of our approach. Moreover, the user completes these stages by adding relevant characteristics (meta-data) that may be valued for defining a property issued from the requirements analysis or from general consistency verifications not directly related to requirements (e.g. consistency of external interface inputs with internal items). We formalize these characteristics (Simon Zayas, Monceaux, & Ait-ameur, 2011) to guide the transition from the requirements or general verifications of a classical specification to the verification of a constraint in our approach. Next paragraphs give an explanation with some examples for each characteristic.

- **Description and origin of the property**. Requirements guide the specification, the design and the management of the development process. In our method requirements are the starting point for formalizing most of the properties (constraints) to be checked. Therefore, the property must be described and its origin identified.

- **Source models**. Concerning the formalization of the source models used to check the expressed property, i.e. that contain the resources and concepts involved in the property definition, they are given a unique identifier and a brief description. A simple schema is proposed for models metadata. For example, since different modeling languages can be used modeling language is a metadata. The identification of the modeling languages enables to select the meta-models to be used during the exportation. On the other hand, models can be seen as entities themselves and some requests can be performed over them (black-box perspective) whether the adequate data is available. Amongst the information to be provided for that kind of requests, we suggest the level of detail of the models, which can be related to the aircraft, to a system of the aircraft, to a sub-system or to a component, and program applicability. In addition to these general characteristics, some more specific domain ontology may be used to describe some other metadata and to request source models, for instance, providing some available properties of the aircraft program -number of engines, doors or other engineering parameters.

- **Model or meta-model level**. Our approach can be used to check properties involving different modeling levels. For instance, if we want to check a modeling property such as that the interfaces defined in different models are consistent (same operations, same types, same parameters and same implementation) to guarantee the compatibility of the subsystems to be developed, the corresponding constraint can be formalized by referring only to meta-model elements. That is possible because only non-specific domain knowledge is needed. On the other

side, a requirement like "*the communication between system S1 and system S2 must use the network A or the network B*" needs to be worked at a model level since there are entities of the domain in the final expression.

- **Category**. To better manage the properties, they should be grouped by categories. We use a current requirement categorization borrowed from (Verries, 2010) which consists of the next categories: functional, performance, operational, architectural, qualitative safety, quantitative safety, maintainability, interchangeability, environment, weight, evolution and behavior. The category of a requirement is an important piece of information which may be used to recommend a suitable property language (see next) to check the related property.

- **Property languages**. Depending on the type of property to be validated we need different languages to formalize the property expression. The appropriate property languages must be identified. In our case studies we have based our property expressions on first order logic (FOL). This kind of property language makes it possible to express a large variety of properties but not always in the most efficient way. For instance, if we need to express a constraint related to some time sequence events, a language to describe scenarios could be more opportune. Therefore, the recommended property language must be indicated (or derived from the category) to complete the specification. The characterization of the property formalizes the expression which validates the inter-model constraint.

In order to better understand the handling of the prototype and the relevant characteristics, let us take a classical application scenario of the approach consisting of a process where:

1) a **tool expert** prepares the meta-models needed to export the source models. Since these meta-models are limited to modeling standards, e.g. CORE (Vitech Corporation, 2007a), this action has only to be done the first time we need a particular meta-model. This meta-model is charged in the tool as illustrated in Figure 84. Area 1 shows the CORE meta-model in the tree-view browser and area 2 illustrates the related meta-data.

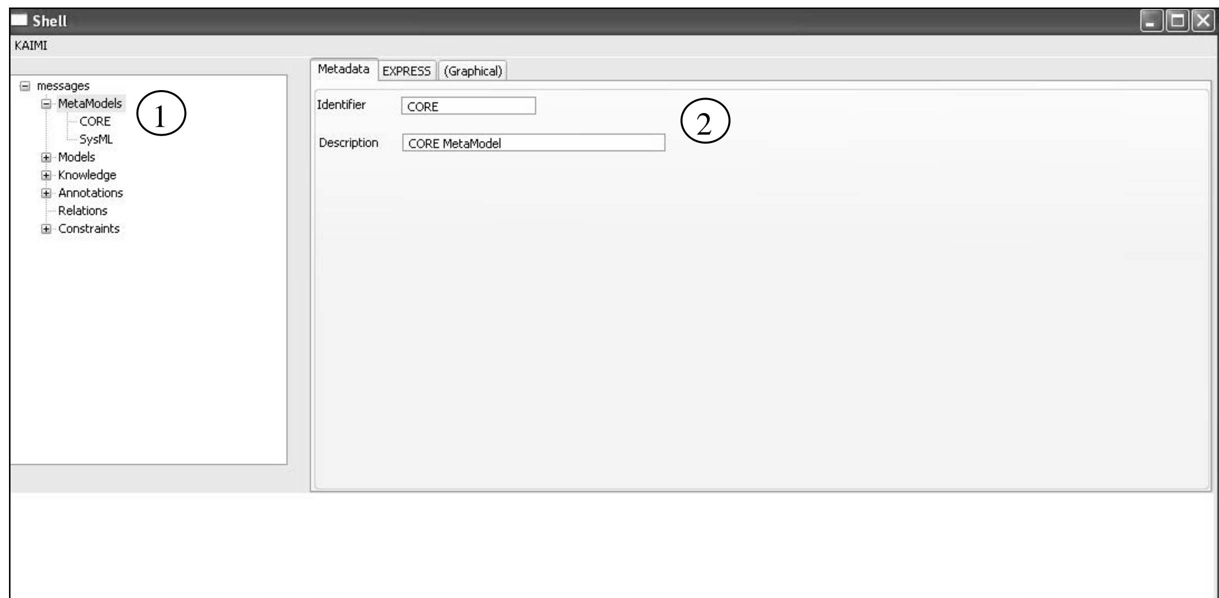


Figure 84. Load of meta-model screen

2) the **engineer in charge of design** creates a project of validation and loads the source models. After selecting the appropriate meta-model, these models are imported. Then, as shown in Figure 85, area 1, the meta-data describing the models are completed.

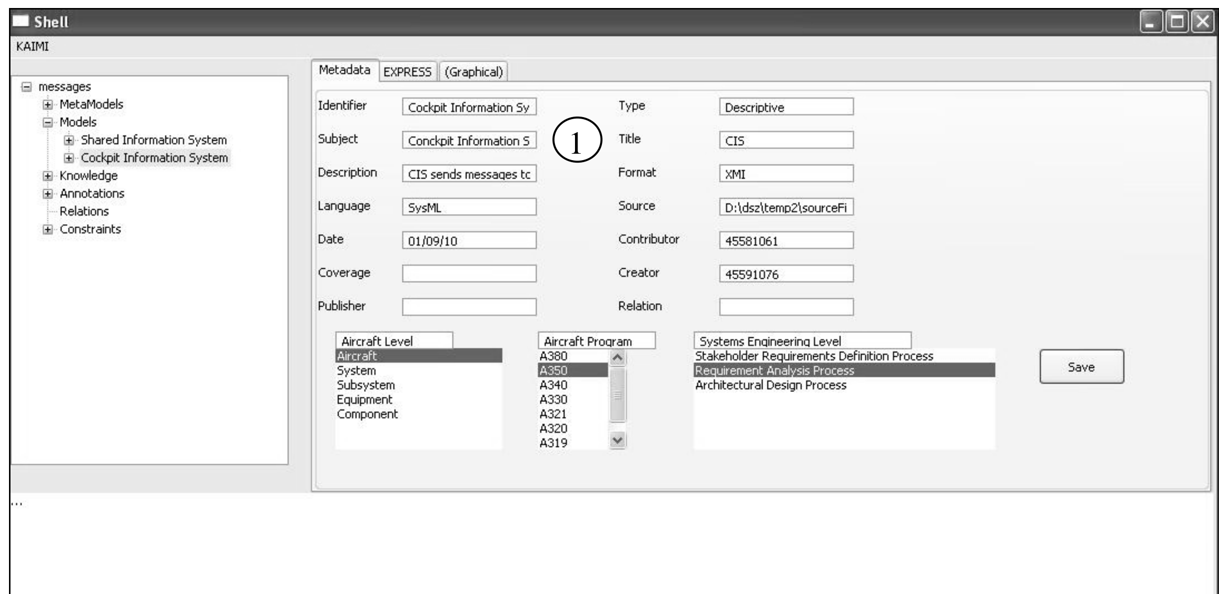


Figure 85. Model meta-data screen

3) a **knowledge engineer** creates the knowledge models which are loaded into the tool. The concepts of the knowledge models can then be browsed as shown in the area 1 of Figure 86.

Figure 86. Knowledge browsing feature

4) the **engineer in charge of design** uses the knowledge models to annotate the imported models. One model entity can be annotated by several concepts of the ontologies and on the other way round one concept of the ontology can be linked to several modeling entities. Therefore, the annotation stores both references to models (area 1 of Figure 87) and ontologies (area 2 of Figure 87) in order to ease traceability. One annotation is an important element of information by itself which also needs to be characterized as shown in area 3 of Figure 87.

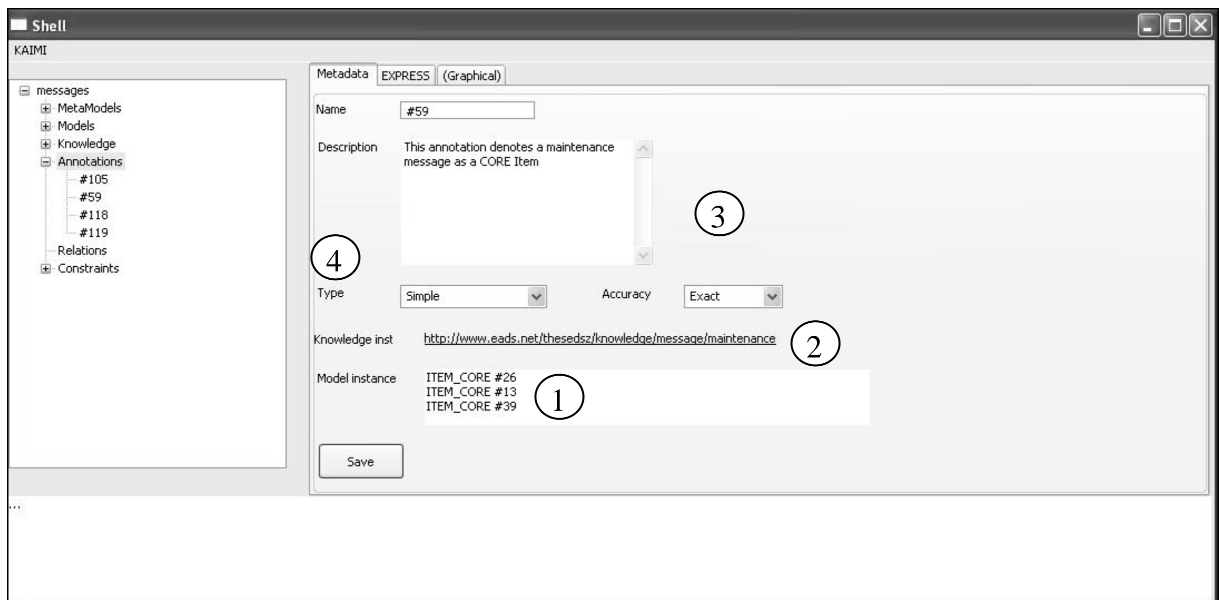


Figure 87. Annotation description screen

5) the **engineer in charge of design** relates the elements of the models allowing the cross-model checking validations. In order to set these links, our current

implementation includes equivalence relations amongst the type of annotations since our relations do not need to be directly exploited in the actual context. These equivalences are considered as a type of annotation (field 4 –*type*- in Figure 87).

6) In the end, the **engineer in charge of design** can build the expression to be checked taking into consideration the characteristics of the constraint to be verified. These characteristics, provided previously by the engineer, are displayed in Figure 88 (area 1), a screenshot of the beta prototype.

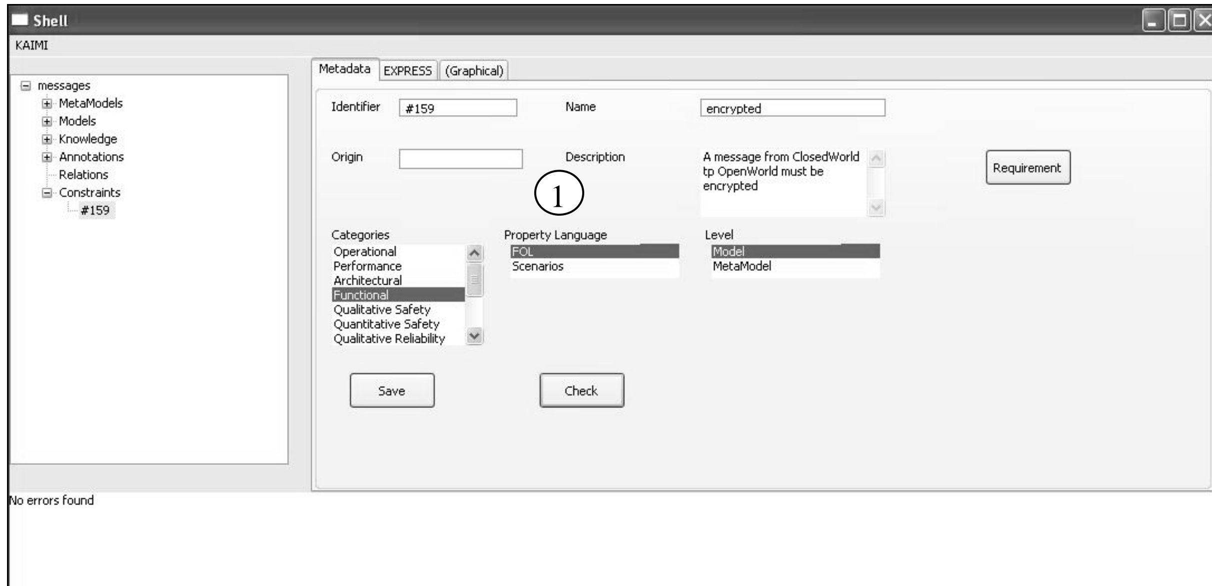


Figure 88. Constraint meta-data screen

VII.5. Conclusion

The beta prototype described in this chapter allows us to visually illustrate the industrial cases and, in this way, to validate the approach with engineers from a process point of view. The table of Figure 89 sums up the tool coverage of the users’ needs.

Need description	Coverage
The prototype must assist the user in the application of the approach	Implemented
The prototype must be user friendly	Graphical visualization and manipulation of models are not currently supported
The prototype must allow users to navigate through the entities and concepts of the different involved models	Implemented as a tree-view navigation
The prototype must allow users to easily inspect the results of a constraint evaluation	To be implemented along with the graphical enhancements
The prototype must be modular and	Covered by the proposed architecture and the

extendible	chosen platform (Eclipse)
The prototype must be a light application	Covered by the chosen platform (Eclipse) and graphical technology (SWT)
A trace of the checked constraints and the results of their evaluations must be recorded	Implemented
Request over meta-data and data of the prototype must be supported	Meta-data is managed

Figure 89. Coverage of needs in beta prototype

As seen in this figure, some important enhancements in the HCI have to be applied to provide engineers with a final platform. Nevertheless, they are not the only changes needed in order to enable an industrial deployment. Therefore, next chapter analyses current industrial deployment scenarios and describes technological and HCI improvements required in such an industrialization path.

Chapter VIII

Deployment in industry

Summary

VIII.1. Industrialization requirements 129

VIII.1.1. Model and configuration management 129

VIII.1.2. Knowledge management 129

VIII.1.3. Model integration 130

VIII.1.4. Inter-model constraint management 130

VIII.1.5. Conclusions 131

VIII.2. Needed technology enhancements 132

VIII.2.1. Technology features 132

VIII.2.2. Needed HCI enhancements 134

VIII.3. Conclusion 138

Abstract. Model-Based Systems Engineering (MBSE) is a discipline which is strongly drawing aeronautical industry's attention. Thus, our approach has to take into consideration the actual efforts related to MBSE in order to appropriately deploy an industrial solution. Such a solution needs to provide robustness from the technology point of view but also to be integrated in current modeling processes in order to get the expected benefits of the industrialization. A successful deployment will lead to enhancements concerning the consistency of models, modeling reuse capabilities and the shared knowledge formalization in order to improve the global collaborative engineering experience.

VIII.1. Industrialization requirements

In order to efficiently obtain the expected benefits of the industrialization of the approach as described in section III.7, some changes are required in the current MBSE process. They are analyzed below.

VIII.1.1. Model and configuration management

The amount of models used in the development of an aircraft is huge enough to demand some means to store and find them. Engineers need distributed means of managing such models and this is the aim of the model repositories. The notion of model repositories is a current research topic. Recent European projects like Cesar (EADS, 2010a) or Crescendo (Coleman, 2011) demonstrate the importance of that kind of solution for nowadays industry. Concerning our approach, model repositories should be completed with black box annotations in order to support queries on models involving concepts from the domain knowledge models, e.g. to find all models regarding a certain life cycle level for an aircraft program.

Besides the model repositories, a correct management of models should include the evolution aspects. Configuration management is a transverse discipline which should be expanded in order to master the different versions of models released all along the aircraft life cycle. Thus, our approach needs to be enclosed by the configuration management processes not only concerning the source models but also the manipulated models and the resources necessary for the implementation of the approach (see section IV.2).

VIII.1.2. Knowledge management

Concerning the knowledge management two possibilities have emerged from discussions with engineers and experts.

Soft knowledge

In this case the knowledge model is built in an ad-hoc manner to check a particular property involving several models. Annotations are described outside the MBSE process. From the knowledge management point of view such kind of models do not need to be managed due to their deciduous nature. Nevertheless, building knowledge models, even simple ones, is a costly task. For this reason, ad-hoc solutions should be only considered when non appropriate general knowledge is available and when the checking of the constraint is too complex to be performed manually.

Shared knowledge

Ontologies are the result of consensus between domain experts in order to formalize their common knowledge. It is difficult to get a consensus but once it is reached, ontologies could be incorporated to the modeling process. This implies that such a kind of implementation of knowledge models are available during the MBSE process and that appropriate guidelines are added. These guidelines could recommend the type of annotations expected in each development stage. Such guidelines can be built, for instance, from existing documents such as, for example, current nomenclature rule documents. At the same time knowledge models can be built from existing documentation both for white-box annotations, e.g. ATA chapters (see section VI.2.2), and for black-box annotations, for instance current *Model Specification* documents demand properties such as "*model development tool name*", "*modeling language*" and so on.

Besides the consensus needed to build an ontology, another difficulty in the aircraft domain context is the evolution of processes and domain knowledge. Thus, configuration management needs also to be applied to ontologies and annotations in order to manage their progress. The correct storage and identification of versions of models and of their contents along with their annotations will allow the reuse of these annotations in an evolutionary spirit of mind and, consequently, an improvement of the MBSE process.

VIII.1.3. Model integration

Amongst the models available via a repository we find the relation meta-model. The distributed access to this meta-model would allow modelers to formalize these kinds of relations during the modeling process. This should be accompanied by new modeling rules concerning inter-model relations. For instance, when a modeler is refining a component of an upper-level model in a new model decomposing the component into subcomponents he or she will denote a composition relation, instance of the meta-model relation, between the component and its subcomponents. In this way, the interconnection of models would be integrated in the MBSE process and the traceability between models improved. Moreover, having in mind a homogenizing perspective, the relation meta-model should be represented in a format usable by the different modeling tools.

VIII.1.4. Inter-model constraint management

Along with the annotation activity, the general constraint definition is the task requiring more human participation and, consequently, time. Therefore, knowledge models should also be used during the requirements definition phases and such requirements should be formal enough to, at least partially, generate the constraints to be validated or some templates to be easily completed by the engineers. CESAR (EADS, 2010b) is a European project that focuses

on the concept of requirements formalization and whose results may complement our own work in a constraint generation perspective. Furthermore, during the activity of definition of a general constraint, an industrial solution should allow engineers to re-use previously declared expressions or sub-expressions.

Thus, the management of inter-model constraints has two main objectives. On the one hand, constraints should be stored in order to enable the reuse of the expression or a part of it, necessary for an efficient MBSE process. Their storage would allow the research of constraints according to their characteristics, e.g. the category of the property to be verified (see section VII.4). On the other hand the different executions of a constraint validation should be accessible and integrated in the configuration management process since constraints should be re-verified with new versions of the models to avoid regression problems.

VIII.1.5. Conclusions

The centralized management of models is one of the future MBSE challenges. Currently, this aspect is not covered by the MBSE process in our industrial context. Nevertheless we defend that it should be implemented in future aircraft programs, along with all the requirements presented in above sections, in order to improve the actual MBSE process and to allow the industrialization of our approach.

During the implementation of our approach, we have detected that some tasks can be time-demanding, mainly the annotation phase. For this reason a successful deployment of a solution would depend on its capacity of integration with the industrial MBSE process.

VIII.2. Needed technology enhancements

The industrial deployment of the approach has technological and methodological outlooks. The implementation of an industrial solution should be able to support the manipulation of big models but the success of such a solution depends on the capacity of integration with MBSE methods. Next subsections describe the proposed prototype enhancements. These enhancements are issued from the conclusions of the different meetings that introduced the current version of the prototype to experts and engineers.

VIII.2.1. Technology features

For the implementation of the prototype, we have chosen EXPRESS modeling language for validation reasons since EXPRESS has all the properties we needed to develop our approach efficiently and consistently. Nevertheless, this solution is not suitable for an industrial environment for several reasons, mainly: 1) current EXPRESS implementations do not allow the management of big models due to central memory limitations; 2) there is not an associated performing request language; 3) EXPRESS has a very low success as a modeling language in industry. Thus, the different packages of the current architecture could be adapted to the industrial context as detailed next.

User interface module. The enhancements of this module are described in detail in section VIII.2.2.

Meta-model module. The ECCO toolkit has been used in a validation context to implement the common framework where the models are exported into. Nevertheless, ECCO is a proprietary solution working with central memory. A central memory implementation has limits regarding the number of instances that can be stored and the reuse features. Therefore, an industrial solution has to be provided with a database in order to store and manage the models. As the context of our work is collaborative engineering this database has to be remotely accessible to act as a repository of models as well. This database is the base pillar of a large-scale solution. For this reason an alternative which can be easily integrated in a database is necessary. The Eclipse Modeling Framework (EMF) (Eclipse.org, 2011c) which includes a meta-model called Ecore is a good candidate considering its maturity and the multiple existing tools around Eclipse. In the context of our approach Ecore would be the meta-meta-model used to define the different meta-models needed for the export activity. Although the development of some meta-models would be still necessary, the open source nature of Ecore eases the possibility of reusing existing meta-models, e.g. the SysML meta-model implemented with Ecore in Topcased (Topcased.org, 2011).

Model module. Once the meta-models are created or imported the exportation mechanism can be automated by using Model-Driven Engineering tools able to handle Ecore semantics, e.g. ATL (Eclipse Foundation, 2011). Moreover, these meta-models should include some intra-model constraints in order to guarantee the consistency of the export from the point of view of the source modeling language, e.g. in SysML "a *FlowProperty* is typed by a *ValueType*, *DataType*, *Block* or *Signal"* (OMG, 2008). These constraints should be already validated in the source modeling tools but we need to replicate them (along with the performance of some more general validations like the number of instances) in order to verify the result of the export. Even though we have introduced pure exportation cases, i.e. the source models are exported without modifying them, in our case studies, the exportation step can also be seen as a transformation. These transformations span from the simple need of the export of a subgroup of entities of a model to the complex need of targeting a meta-model different from the original one. Actually, the first type of transformation has been implicitly used in our case studies since we have worked only with the entities that were strictly necessary to validate the inter-model constraints. The goal of this simplification is to improve the efficiency of the approach. Concerning the second type, an example of a different target meta-model are SysML entities that are converted into MARTE (Gerard, 2009) elements in order to add real-time features needed for the validation of certain inter-model constraints. The industrial tool should consider both types by allowing the plugging of filters to manage such transformations in a transparent way. These filters would be plugged in different moments of the export activity.

- Before the export of the source model as instances of the target meta-model. The aforementioned choice of a subgroup of instances illustrates that kind of filter.

- During the export for replacing the default export implementation, which reads the source meta-class and instantiates the same meta-class in the common framework. It is the kind of filter necessary when targeting a meta-model different to the source meta-model.

- After the export to add information to the exported entities. The added information could consist of some computations or even some reused or automated annotations.

Knowledge module. During the implementation of the case studies we have developed knowledge models in the EXPRESS modeling language. This language allows modelers to build simple models representing the implicit knowledge. Nevertheless, as the industrial environment will logically lead to an increase of knowledge models' complexity and to the need of some reasoning features (e.g. giving an aircraft speed to get the related flight phases

thanks to ontological relationships), some specialized ontological languages could be used in combination with reasoning engines, e.g. OWL with Pellet (Sirin & Parsia, 2004) or framework solutions as Jena (Team, 2011).

Annotation module. The point of the implementation is that, independently of the ontological language applied, during the annotation activity the reference to the knowledge concepts will be done via URIs. In an industrial environment engineers would be able to choose the most adequate language to make their implicit knowledge explicit. Thus, in relation with the knowledge scenarios described in section VIII.1.2, the soft knowledge models could be represented using general modeling languages as UML. These modeling languages are well-known by engineers and allow them to easily collaborate in the construction of simple ad-hoc models. On the other hand, ontological languages as OWL could fit better the consensual and complex nature of the shared knowledge models.

Integration module. In our prototype we have integrated models by using equivalence relationships but, as we have described in previous sections, an industrial deployment would imply other inter-model relationships. This includes the top-down or bottom-up scenarios that have not been treated in our work but that have to be supported in a final solution. Thus, integration model should incorporate ontologies in order to describe complex relations, e.g. composition. Furthermore, in the purpose of leveraging the workload of our approach, rules and reasoning characteristics could be added to the integration model in order to identify and suggest inter-model relationships in a more automatic way. For example, if in the modeling process an element of an upper level is decomposed into several entities of the lower level, the reasoner could recommend interconnecting them during the integration activity.

Constraint module. Depending on the nature of the inter-model constraints to be verified, the expression model to use will vary. Thus, an industrial solution should provide engineers with recommendations about the most adequate expression languages according to the characterization of the constraint. In our case studies FOL expressions implemented in EXPRESS are well fitted but in other cases different languages could be recommended, e.g. PSL (Accellera, 2004) for hardware-related constraints. Finally, the expression language has to be interpreted by an engine enabling the evaluation of constraints, as in the case of the instance checking in ECCO, and providing traceability features in order to correctly identify the cause of a failure in the validation of a constraint.

VIII.2.2. Needed HCI enhancements

The graphical user interface of the prototype would have to be improved in order to be usable by the engineers. Currently the different models of the approach can be graphically browsed but such a graphical support is missing in the creation or modification of models

themselves and it implies a lot of costly work which would have to be reduced in an industrialized version. Therefore, in general terms the future HCI should provide users with graphical model features such as boxes to manipulate entities and attributes; lines to represent relationships and *drag and drop* facilities to interconnect elements. These needed enhancements are detailed below for each of the approach activities.

Exportation. Meta-models would have to be developed using boxes to represent the entities and their attributes and lines to illustrate the different possibilities of relationships (e.g. an arrow is used to denote a *generalization* relationship in area 1 of Figure 90). This graphical solution should also be applied for the visualization of the exported models without allowing their modification since we consider that the source models have to be modified in their original modeling tools and re-imported.

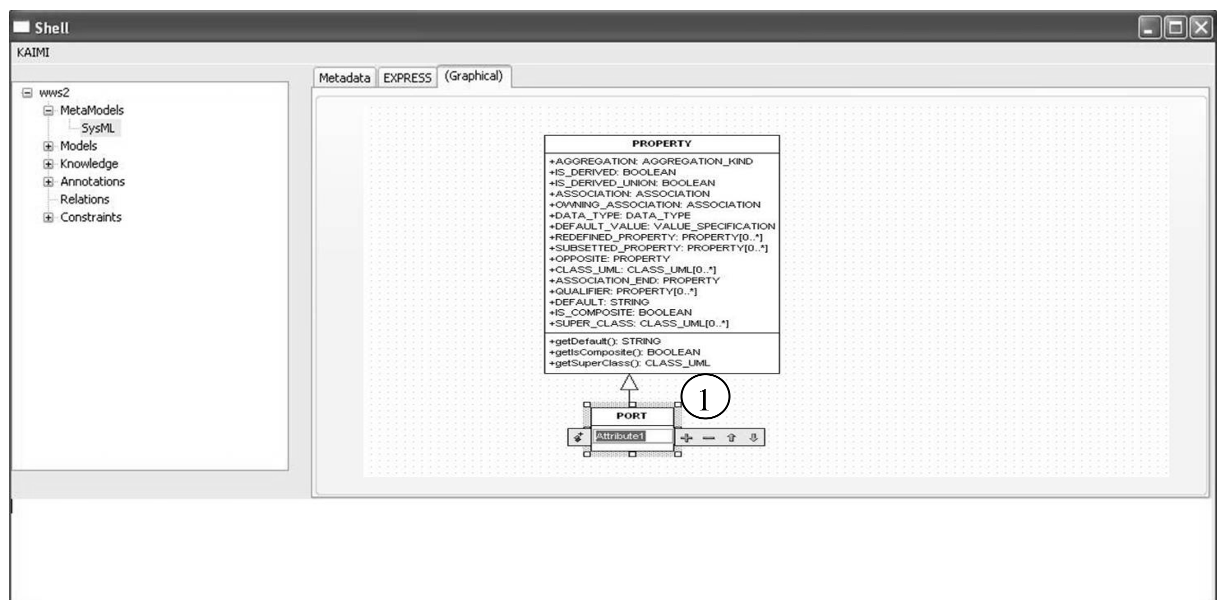


Figure 90. Future meta-model HCI with a SysML example

Annotation. As in the case of source models, knowledge models should be managed in the corresponding knowledge modeling tools. The content of the knowledge models could be presented using the same box/lines interface than the other models. This would allow the tool to have a homogeneous way of manipulating entities and concepts which is important to reduce the learning time needed by the users in order to master the tool. Concerning the annotation, an industrial version would have to provide users with *drag and drop* features which, as illustrated in Figure 91, shall allow users to: 1) drag a concept from the tree-view representation of a knowledge model and to drop it in the annotation area in order to create a new annotation; 2) to drag entities from the tree-view of models' content (area 1 of Figure 91) and to drop it in the new annotation area (trajectory illustrated by the dotted line 2 of Figure 91) to put these entities in relation with the knowledge concept, i.e. to annotate it.

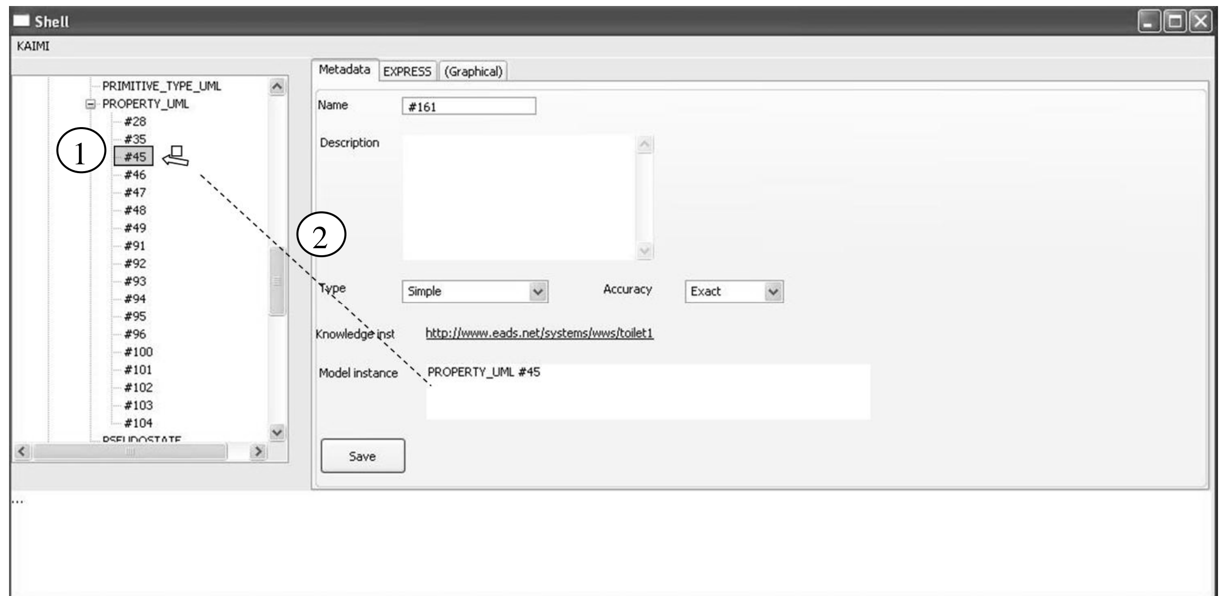


Figure 91. Drag and drop of an instance to annotate it

Integration. The future HCI would enable to *drag and drop* entities from the different exported models tree-views to the relation area in order to interconnect them. Furthermore the adequate entity denoting the kind of relationship (equivalence, composition …) could be selected from a list containing the entities of the relation meta-model as illustrated in area 1 of Figure 92.

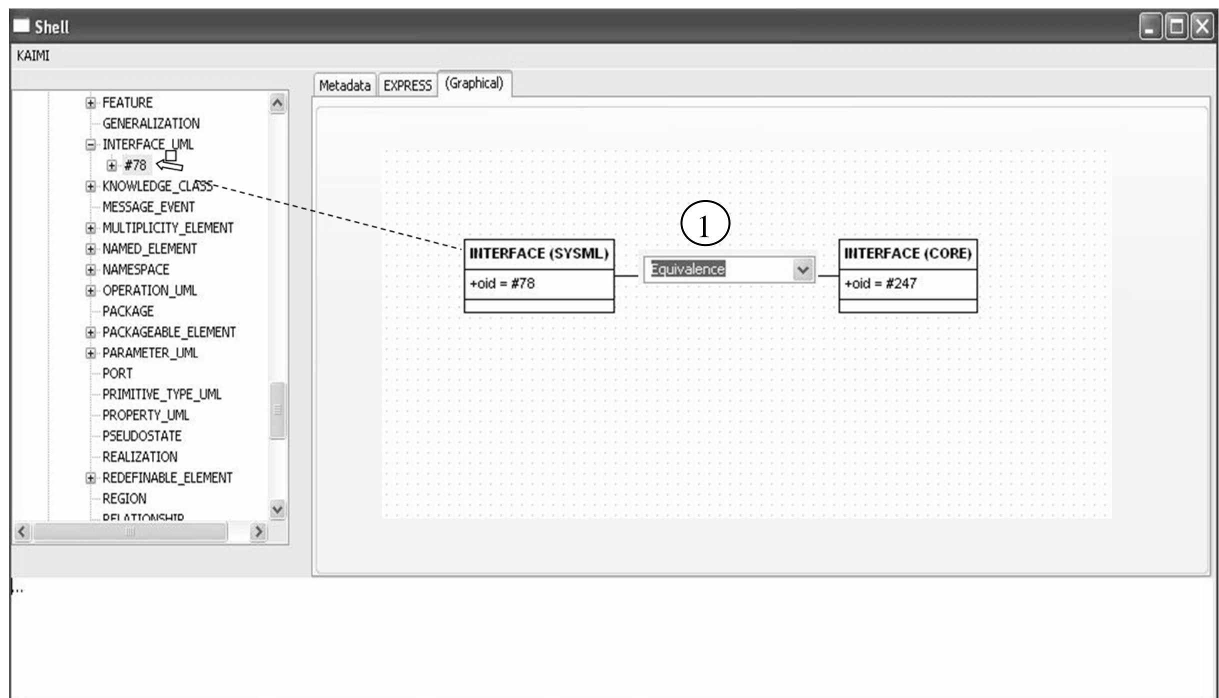


Figure 92. Equivalence relationship between instances from CORE and SysML models

Constraint description. The expression representing the inter-model constraint to be checked would better be supported in a graphical way. For instance, Figure 93 illustrates a calculator-style support (area 1) for the FOL expressions. Finally, the different executions of a constraint checking would be listed in the form of a navigable table as shown in area 1 of Figure 94.

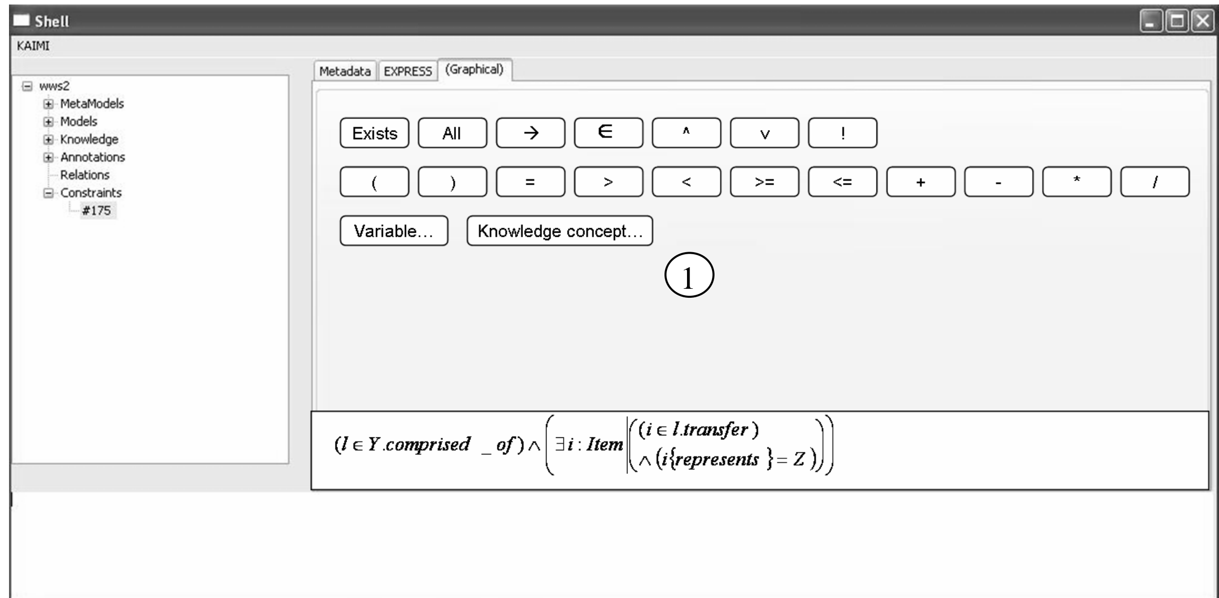


Figure 93. Graphical construction of a FOL expression

Shell
KAIMI

wws2
MetaModels
Models
Knowledge
Annotations
Relations
Constraints
#175

Metadata | EXPRESS | (Graphical)

Execution date	Result	Observations
16/09/2011	Fail	Model analysis
13/11/2011	Ok	

1

Figure 94. Traceability of the executions of a constraint validation

VIII.3. Conclusion

The expected deployment of our approach is analyzed in this chapter from two points of view. First of all, some improvements, mainly concerning the centralized management of models, are needed in current MBSE processes in order to permit an optimal integration of our approach. Moreover, these improvements should be accompanied by some technological and graphical enhancements according to the current status of the approach implementation. These enhancements are the topic of the second part of the chapter.

Conclusion and perspectives

In the context of Systems Engineering design methodologies, engineers need to work with models from different teams, methodologies and know-how. This collaborative work results in different types of models, modeling languages and modeling techniques. Thus, heterogeneous models are a logical consequence of such variability. This heterogeneity becomes a problem when models need to be shared by those different teams in order to perform overall analysis and validations. In such cases making explicit the implicit knowledge is essential.

Our approach proposes to integrate heterogeneous models and to model and check inter-model constraints validation by making explicit, formalizing and exploiting such additional knowledge that is usually implicitly assumed by designers.

Contributions

Our contributions have been developed according to different directives.

Methodology

Our work puts together two concepts: heterogeneous modeling and explicitation of implicit knowledge.

We have defined a layered method to use knowledge and to define expressions with a flexible language in order to check constraints over inter-model relations. We suggest

applying our approach as part of the validations to perform inside Systems Engineering methodologies. Particularly, our work has demonstrated that the method is suitable in a collaborative context when we put together and validate some of the models of the current design stage to guarantee the quality of the input for the next design stage.

Meta-models are built using the shared and common modeling language concept of our proposal. Thus, for each exported model its meta-model is written in such a language. This makes it possible to export of the source model as an instance of the meta-model written in the common language. Our approach does not pretend to find a unique meta-model able to replace the source meta-models but to keep such meta-models expressed in a common environment. Therefore, we can work in a shared modeling semantics with different meta-classes allowing the interconnection of entities of different modeling languages since the language to describe such meta-classes is common and shared.

The exported models are annotated and integrated in this framework in order to support the validation of inter-model constraints using implicit knowledge. The particularity of this solution is that source models are kept outside the loop and are not modified since our approach is non-intrusive. In this way we can trace back the origin of a non-fulfilled inter-model constraint to identify the original source entities implied in the fail. Next, engineers in charge of the source models perform the actions necessary to correct the problem.

This method has been validated with different case studies. The used examples permit to have a large variability of modeling cases and situations: one modeling language, one modeling language with different modeling rules, two modeling languages. Along with the fact that the presented case studies represent different domains and types of implicit knowledge.

Explicitation of implicit knowledge

The originality of our approach is the formalization of the explicitation of implicit knowledge and the annotation of heterogeneous engineering models. This knowledge is managed independently of the annotated models thanks to the use of aside models and unique identifiers. Thus, annotations contain the link between exported models and knowledge concepts acting as an intermediate layer. This intermediate layer and the fact that source models are exported permit the evolution of source models independently of the application of the approach.

Ontologies are recommended for the explicit modeling of the implicit knowledge since the engineering context fits their formal and consensual vocation besides the fact of providing precise identification concepts like URIs.

Inter-model constraints

During the design of a system several properties must be verified. Amongst them, the inter-model constraints verification involves multiple models simultaneously. Thus, our approach allows engineers to integrate models in order to validate such inter-model constraints over them. These constraints are expressed by referencing both entities of the exported models and concepts of the implicit knowledge made explicit. The definition of the constraints is based on expression models.

In our case studies we have validated constraints that can be expressed by FOL expressions. In order to implement them we have developed a formal model of expressions using the EXPRESS modeling language. This model is an extension to the PLIB expression model which does not include FOL expressions.

Tool support

In order to guide users in the use of our methodology, we have developed a prototyping tool. This prototype is process-oriented and supports each of the modeling activities of our approach.

- **Export**. It provides access to the meta-models and it permits the loading of source models by instantiating these meta-models.
- **Annotation**. It allows users to manage knowledge models and to use the concepts of such models to annotate the exported models.
- **Integration**. It permits to set up relations between entities of the annotated models.
- **General constraint definition and verification.** The prototype enables the construction of the inter-model constraint to be verified and their operational verification.

This prototyping tool has allowed us to demonstrate the usability of the approach from a user and process perspective.

Deployment and applicability

The formal validation of the proposal and the implementation of different types of case studies in the prototype demonstrate the applicability of our approach. Nevertheless, an industrial deployment requires the evolution of the prototype. These aspects are discussed in our perspectives.

Perspectives

The work described in this thesis opens several perspectives. Some of them are described below.

Scientific perspectives

Evolution of models

In the course of the modeling process, models have different degrees of maturity. As a consequence they evolve and new versions appear. Models can also evolve because they are reused in a new program. In any case, evolution of models should be part of the configuration management activities. Logically, our approach has to take into consideration this evolution of models and to handle reuse of annotations (Luong & Dieng-Kuntz, 2007) and of inter-model constraints.

Abstraction of modeling language

In the current description of inter-model constraints, entities from the annotated models and concepts of the knowledge models are used. Nevertheless, the access to entities and attributes needs a quite well comprehension of the corresponding meta-models. That is a problem for the use of multiple modeling languages since the learning curve can become too big. Therefore, we think that the definition of constraints should rely on a modeling ontology describing general Systems Engineering concepts. Such ontology would allow engineers to write their constraints in a more natural way and would ease the eventual generation of constraints from formal requirements. Finally, the choice of other logics different from FOL should be taken into consideration when analyzing the characteristics of a constraint to be verified.

Inter-model relationships

In the thesis we have described two categories of inter-model relations from a process point of view: same level and top-down or bottom-up. Our case studies have focused on same level relations and, as a perspective, top-down or bottom-up relation cases should be also taken into consideration. We defend that our approach is also applicable to bottom-up or top-down relations but that the integration activity and the relation meta-model would need to be empowered. This integration step is still valid but top-down or bottom-up relations contribute with new scenarios, i.e. design refinement, models composition and model abstraction respectively. Thus, cases including such kind of inter-model relations should be studied in

order to extend our work. On the other hand, from the relation meta-model point of view, the impact of the integration of more than two models should be studied. In this context the work described in (Delmas, 2004) can be a good support.

Industrial perspectives

Scalability

The prototype has allowed us to functionally validate the proposal. Now the scalability of the solution has to be addressed in 2 ways. On the one hand and in order to be industrialized, the implementation of our proposal has to be able to manage a great number of models and entities. This implies that an industrial version should be built upon model repositories and remote databases in order to be able to work in a collaborative engineering context. On the other hand, other Systems Engineering domains (automotive, space and other complex systems) should be considered from a domain scalability point of view. Moreover, the different modeling activities described in our approach (exportation, annotation, integration and constraint definition) should be automated as much as possible to ease its integration with current Systems Engineering methods. Concerning the automation of annotations some work already exists in the semantic web domain which can be a good point of depart (Handschuh, Staab, & Ciravegna, 2005)(Dill et al., 2003)(Hammond, Sheth, & Kochut, 2002).

MBSE integration and services

Concerning MBSE, the integration of our method with current practices should be accompanied by a procedure of standardization. We think that the explicitation of implicit knowledge and the relation between heterogeneous models need to form part of current MBSE standards in order to provide a better management of complex systems life cycle. One challenging perspective is to develop a business process platform supporting MBSE processes with adequate services and a well definition of roles (administration tasks, knowledge management, constraint management,…). In such context, our approach would be part of the services offered by the platform to the specified roles. Amongst these services, request services based on the annotations should be also considered in the perspectives.

References

AIRBUS. (2008). *AMISA Method*. Toulouse, France.

ATA. (2011). Air Transport Association of America. *Association Web Page*. Retrieved from http://www.airlines.org/About/AboutATA/Pages_Admin/AboutATA.aspx

Abouzahra, A., Bézivin, J., Didonet, M., Fabro, D., & Jouault, F. (2005). A Practical Approach to Bridging Domain Specific Languages with UML profiles. *Workshop on Best Practices for Model Driven Development, OOPSLA*. San Diego.

Accellera. (2004). *Property Specification Language Reference Manual. Language* (pp. 1-123). Napa, CA. Retrieved from http://www.eda.org/vfv

Ait-Ameur, Y, Besnard, F., Girard, P., Pierra, G., & Potier, J. C. (1995). Formal specification and metaprogramming in the EXPRESS language. *Conference on Software Engineering and Knowledge Engineering* (pp. 181-189).

Ait-Ameur, Y, Pierra, G., & Sardet, E. (1995). Using the EXPRESS language for metaprogramming. *Proceedings of the 3rd International Conference of EXPRESS User Group EUG'95*. Grenoble.

Alexander, P., Kong, C., Ashenden, P., Systems, A., Barton, D., & Menon, C. (2003). *Rosetta Strawman Version 0.3. Distribution* (pp. 1-150). Kansas.

An, Y., & Song, I.-Y. (2008). Discovering Semantically Similar Associations (SeSA) for Complex Mappings between Conceptual Models. *Conceptual Modeling - ER 2008: 27th International Conference on Conceptual Modelling* (pp. 369-382). Springer. Retrieved from http://books.google.com/books?id=Y17gZ6rdid4C&pg=PA369&lpg=PA369&dq=semantically+similar+associations+an+song&source=bl&ots=xaba0GLboa&sig=tbAC0UlSEzm5IR3eeNJvr_aWpHg&hl=fr&ei=OsyoS_foIMKSjAf224DoAQ&sa=X&oi=book_result&ct=result&resnum=1&ved=0CAsQ6AEwAA#v=onepage&q=semantically similar associations an song&f=false

Antonio, F. D., Missikoff, M., Bottoni, P., & Hahn, A. (2006). An ontology for describing model mapping / transformation tools and methodologies: the MoMo ontology. *Proceedings of EMOI-INTEROP*. CEUR.

Athena Project. (2006). *Semantic Annotation language and tool for Information and Business Processes Apendix F: User Manual* (pp. 1-26).

Auzelle, J.-P., Garnier, J.-L., & Pourcel, C. (2009). Architecture et Ingenierie des Systeme de Systemes. AFIS.

Bakhtouchi, A., Chakroun, C., Bellatreche, L., & Aït-Ameur, Y. (2011). Mediated Data Integration Systems using Functional Dependencies Embedded in Ontologies. *Recent Trends in Information Reuse and*

Integration (pp. 227-256). Springer. doi:10.1007/978-3-7091-0738-6_11

Bechhofer, S., Harmelen, F. van, Hendler, J., I.Horrocks, McGuinness, D., Patel-Schneider, P., & Stein, L. (2004). OWL Web Ontology Language Reference. *W3C*. Retrieved from http://www.w3.org/TR/owl-ref/

Berners-Lee, T., Fielding, R., Irvine, U., & Masinter, L. (1998). Uniform Resource Identifier (URI): Generic Syntax. *Request for Comments 2396*. Retrieved from http://tools.ietf.org/html/rfc2396

Boehm, B., & Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. (P. Education, Ed.) (pp. 1-272). Crawfordsville, Indiana: Addison-Wesley.

Boronat, A., Knapp, A., Meseguer, J., & Wirsing, M. (2008). What is a Multi-Modeling Language? In U. M. Andrea Corradini (Ed.), *WADT* (pp. 71-87). Pisa, Italy: Springer.

Boudjlida, N., & Panetto, H. (2008). Annotation of Enterprise Models for Interoperability Purposes. In IEEE (Ed.), *IWAISE'2008*. IEEE.

Bräuer, M. (2007). *Design of a Semantic Connector Model for Composition of Metamodels in the Context of Software Variability*. Technisque Universitat Dresden.

Caplat, G., Sourrouille, J. L., & Pascal, B. B. (2003). Considerations about Model Mapping. *INSA*. Lyon, France.

Cerami, B. E. (2002). *Web Services Essentials* (p. 304). O'Reilly. doi:0-596-00224-6

Chandrasekaran, B., Josephson, J. R., & Benjamins, V. R. (1999). What are ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, *1*(January/February 1999), 20-26. doi:1094-7167/99

Chen, Y., & Chu, H. (2007). Enabling collaborative product design through distributed engineering knowledge management. *Computers in Industry*, *59*, 395-409. doi:10.1016/j.compind.2007.10.001

Coleman, P. (2011). *Developing the Behavioural Digital Aircraft* (pp. 1-16). Madrid, Spain. Retrieved from http://www.cdti.es/recursos/doc/eventosCDTI/Aerodays2011/3D1.pdf

Connolly, D., Harmelen, F. van, Horrocks, I., McGuinness, D. L., Patel-Schneider P. F., & L.Stein. (2001). DAML+OIL Reference Description. *W3C Note*. Retrieved from http://www.w3.org/TR/daml+oil-reference

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web Services Web. An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, *6*(2), 86-93. doi:10.1109/4236.991449

Damjanović, V., Behrendt, W., Plössnig, M., & Holzapfel, M. (2007). Developing Ontologies for Collaborative Engineering in Mechatronics. Salzburg, Austria.

Delmas, R. (2004). *Un Cadre Formel pour la Modélisation Hétérogène et la Vérification Compositionnelle des Systèmes Avioniques Modulaires Intégrés*. ENSAE-SUPAERO.

Department Of Defense. (2007). *DoD Architecture Framework Volume I:*

Definitions and Guidelines. Retrieved from http://dodcio.defense.gov/docs/dodaf_volume_i.pdf

Dill, S., Eiron, N., Gibson, D., Gruhl, D., Guha, R., Jhingran, A., Kanungo, T., et al. (2003). SemTag and Seeker : Bootstrapping the semantic web via automated semantic annotation. *12th International Conference on World Wide Web* (pp. 178-186). ACM Press.

EADS. (2010a). Overview CESAR 2010 Cost-efficient methods and processes for safety relevant embedded systems.

EADS. (2010b). *CESAR: Definition and exemplification of RSL and RMM. Distribution.*

EIA, & ANSI. (1994). EIA 632, Standard – Processes for Engineering a System.

Eclipse Foundation. (2011). ATL - a model transformation technology. Retrieved from http://eclipse.org/atl/documentation

Eclipse.org. (2011a). Eclipse IDE. Retrieved a from http://www.eclipse.org/org/

Eclipse.org. (2011b). The Standard Widget Toolkit. Retrieved b from http://eclipse.org/swt/

Eclipse.org. (2011c). Eclipse Modeling. Retrieved c from http://eclipse.org/emf/

Eker, J., & Janneck, J. W. (2003). Taming Heterogeneity - the Ptolemy Approach. *IEEE* (pp. 127-144). IEEE. doi:10.1109/JPROC.2002.805829

Estefan, J. A. (2008). *Survey of Model-Based Systems Engineering (MBSE) Methodologies*. INCOSE MBSE. Retrieved from http://www.omgsys.org/MBSE_Methodology_Survey_RevB.pdf

Eurocontrol, & Commission, E. (2010). SESAR The future of flying. European Commission. Retrieved from http://ec.europa.eu/transport/air/sesar/doc/2010_the_future_of_flying_en.pdf

Figay, N. (2009). *Interopérabilité des applications d'entreprises dans le domaine technique "Interoperability of technical enterprise applications." Genesis*. UNIVERSITE CLAUDE BERNARD - LYON 1.

France, R., Raton, B., Evans, A., Lano, K., & Rumpe, B. (1998). The UML as a Formal Modeling Notation. *Computer Standards Interfaces* (19th ed., pp. 325-334). Springer.

Friedman, M., Levy, A., & Millstein, T. (1999). Navigational Plans For Data Integration. *AAAI-99*. AAAI.

Fritzson, P. (2003). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Engineering* (p. 939). Wiley-IEEE Press.

Gerard, S. (2009). MARTE: Outlines and added values for SysML 1. Toulouse: CEA.

Gert, J., & Eckert, R. (2000). Experiences from the use and development of ISO 10303-AP 233 Interfaces in the Systems Engineering domain. Retrieved from http://www.ap233.org/ap233-public-information/reference/PAPER_eckert_johansson-SEDRES-Lessons-Learned.pdf

GmbH, Lks. (2012). JSDAI. Retrieved from http://www.jsdai.net/

Gonzalez-perez, C., & Henderson-sellers, B. (2007). Modelling software development methodologies: A conceptual foundation. *The Journal of Systems and Software*, *80*, 1778-1796. doi:10.1016/j.jss.2007.02.048

Gruber, T. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *Computer Studies* (43rd ed.).

Guizzardi, G. (2008). *Ontology-Driven Conceptual Modeling with Applications*. Gramado, Brazil. Retrieved from http://www.inf.ufes.br/~gguizzardi/SBSI2008CR.pdf

Halevy, A. Y., Arbor, A., & Yu, C. (2007). Data Integration with Uncertainty. *VLDB '07 Porceeedings of the 33rd international conference on Very Large Data Bases* (pp. 687-698). VLDB Endowment.

Halevy, A., & Ordille, J. (2006). Data Integration : The Teenage Years. *VLDB '06 Porceeedings of the 32nd international conference on Very Large Data Bases* (pp. 9-16). VLDB Endowment.

Hammond, B., Sheth, A., & Kochut, K. (2002). Semantic Enhancement Engine: A Modular Document Enhancement Platform for Semantic Applications over Heterogeneous Content. In &V. K. & L. Shklar (Eds.), *Real-world Semantic Web applications* (pp. 29-49). IOS Press.

Handschuh, S., Staab, S., & Ciravegna, F. (2005). S-CREAM — Semi-automatic CREAtion of Metadata. *Int'l Journal on Semantic Web & Information Systems*, *1*(1), 1-18.

Hardebolle, C., & Boulanger, F. (2008). ModHel ' X: A Component-Oriented Approach to Multi-Formalism Modeling. In H. Giese (Ed.), *Models in Software Engineering* (pp. 247-258). Springer Berlin / Heidelberg. doi:10.1007/978-3-540-69073-3_26

Haskins, C., Forsberg, K., Krueger, M., Walden, D., & Hamelin, R. D. (2010). *Systems Engineering Handbook*. INCOSE.

Hessellund, A. (2009). *Domain-Specific Multimodeling*. IT University of Copenhagen, Denmark.

Hillmann, D. (2005). Using Dublin Core. Retrieved from http://dublincore.org/documents/2005/11/07/usageguide/

Honour, E. C. (2004). Understanding the Value of Systems Engineering. *Proceedings of the 14th Annual INCOSE International Symposium*.

Horrocks, Ian, Inference, N., Patel-schneider, P. F., Technologies, L., Boley, H., Tabet, S., Grosof, B., et al. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member Submission*. Retrieved from http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/

Hose, K., Roth, A., Zeitz, A., Sattler, K.-uwe, & Naumann, F. (2008). A Research Agenda for Query Processing in Large-Scale Peer Data Management Systems. *Information Systems*, *33*(7-8), 597-610. doi:10.1016/j.is.2008.01.012

IBM. (2009). *Rational Rhpasody API Reference Manual*. Retrieved from publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/rhapsody_api.pdf

IBM. (2011a). The Harmony Process. *Rational Harmony*. Retrieved a from http://www-01.ibm.com/software/rational/services/harmony/

IBM. (2011b). DOORS family. Retrieved b from www-01.ibm.com/software/awdtools/doors/productline/

IBM. (2012). Rational Rhapsody Designer for Systems Engineers. Retrieved from www-01.ibm.com/software/rational/products/rhapsody/designer/

IEEE. (2005). IEEE Std 1220-2005 IEEE Standard for Application and Management of the Systems Engineering Process.

INCOSE. (2007). *SYSTEMS ENGINEERING VISION 2020*. Retrieved from http://www.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf

ISO. (1994). ISO 10303-11 Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods. International Organization for Standardization.

ISO. (1997a). ISO 13584 Industrial automation systems and integration - Parts library PLIB. International Organization for Standardization.

ISO. (1997b). ISO 13584-20 Industrial automation systems and integration - Parts library PLIB - Logical model of expressions. International Organization for Standardization.

ISO. (2008). ISO/IEC 15288:2008 Systems and Software Engineering. International Organization for Standardization.

Jean Bézivin, Didonet Del Fabro, M., Jouault, F., & Valduriez, P. (2005). Combining Preoccupations with Models. *Proceedings of the First Workshop on Models and Aspects - Handling Crosscutting Concerns in Model-Driven Software Development - MDSD*. Glasgow.

Jean, S., Pierra, G., & Ait-Ameur, Y. (2007). Domain Ontologies: A Database-Oriented Analysis. *Web Information Systems and Technologies. Lecture Notes in Business Information Processing* (pp. 238-254). Springer. doi:10.1007/978-3-540-74063-6_19

Kelly, A. (2008). *Changing Software Development: Learning to Become Agile* (p. 258). Wiley.

Klein, M. (2001). Combining and relating ontologies: an analysis of problems and solutions. In Gomez- & M. Perez, A., Gruninger, M., Stuckenschmidt, H., et Uschold (Eds.), *Workshop on Ontologies and Information Sharing, IJCAI'01*. Seattle.

Klein, R. (2000). Knowledge Modeling in Design - The MOKA framework. *Artificial Intelligence in Design - The MOKA framework* (pp. 77-102). Kluwer Academic. Retrieved from http://books.google.com/books?hl=fr&lr=&id=qtxKl-jaXg8C&oi=fnd&pg=PA77&dq=Knowledge+Modeling+in+Design+?+the+MOKA+framework&ots=zns5vSjCE_&sig=H-gD_FmNGW1ViPYRu9lX-h05KKc#v=onepage&q=Knowledge Modeling in Design ? the MOKA framework&f=false

Kolaitis, P. G. (2005). Schema Mappings , Data Exchange , and Metadata Management. *PODS '05 Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of databse systems*. ACM. doi:10.1145/1065167.1065176

Kolovos, D. S., Paige, R. F., & Polack, F. A. C. (2006). The Epsilon Object Language (EOL). *Lecture Notes in Computer Science* (pp. 128-142). Springer. doi:10.1007/11787044_11

Kolovos, D., Paige, R., & Polack, F. (2008). Detecting and Repairing Inconsistencies across Heterogeneous Models. *2008 International Conference on Software Testing, Verification, and Validation*, 356-364. Ieee. doi:10.1109/ICST.2008.23

Krygiel, A. (1999). *Behind the Wizard 's Curtain. An Integration Environment for a System of Systems*. Retrieved from http://www.dodccrp.org/files/Krygiel_Wizards.pdf

Kvan, T. (2000). Collaborative design: what is it? *Automation in Construction*, *9*(4), 409-415. doi:10.1016/S0926-5805(99)00025-4

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: a brief history. *Computer*, *36*(6), 47-56. doi:10.1109/MC.2003.1204375

Lenzerini, M. (2002). Data Integration : A Theoretical Perspective. *Proceeding of 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of databse systems*. NY: ACM. doi:10.1145/543613.543644

Lin, Y. (2004). Model annotations using requirements engineering techniques for model reuse and model integration. Trondheim, Norway. Retrieved from http://caise04dc.idi.ntnu.no/CRC_CaiseDC/YunLin.pdf

Long, J. (2000). *Relationships between Common Graphical Representations in System Engineering*. Retrieved from http://www.ap233.org/ap233-public-information/reference/PAPER_J-Long-CommonGraphical-Notation-in-SE-2002.pdf

Luong, P.-H., & Dieng-Kuntz, R. (2007). A Rule-Based Approach for Semantic Evolution. *Computational Intelligence*, *23*(3), 320-338. doi:10.1111/j.1467-8640.2007.00308.x

Lykins, H., & Ave, N. F. (1999). Adapting UML for an Object Oriented Systems Engineering Method (OOSEM). *Proceedings of the 10'th International INCOSE Symposium*, (Dockerill).

Mandutianu, S. (2009). Modeling Pilot for Early Design Space Missions. *7th Annual Conference on Systems Engineering Research (CSER 2009).*

Marca, D. A., & McGowan, L., C. (1987). *SADT: structured analysis and design technique*. McGraw-Hill.

Mastella, L. S., Abel, M., Ros, L. F. D., Perrin, M., & Rainaud, J.-françois. (2007). Event Ordering Reasoning Ontology applied to Petrology and Geological Modelling. *Advances in Soft Computing*, *42*(Theoretical Advances and Applications of Fuzzy Logic and Soft Computing), 465-475. doi:10.1007/978-3-540-72434-6_46

Mokhtari, N., & Corby, O. (2009). Contextual Semantic Annotations : Modelling and Automatic Extraction. *K-CAP'09*. Redondo Beach: ACM.

Monzón, A. (2010). Bi-directional Mapping between CMMI and INCOSE SE Handbook. *ERTS*[2]. Toulouse.

Mossakowski, T. (2004). Heterogeneous Specification and the Heterogeneous Tool Set. In W. Carnielli, F. M. Dionisio, & P. Mateus (Eds.), *Proceedings of CombLog'04 Workshop on Combination of Logics: Theory and Applications* (pp. 129-140). Departamento de Matematica - Instituto Superior Tecnico.

Mukerji, J., & Miller, J. (2003). MDA Guide Version 1.0.1. Retrieved from http://www.enterprise-architecture.info/Images/MDA/MDA Guide v1-0-1.pdf

NASA. (1999). NASA Systems Engineering Handbook. NASA. Retrieved from http://www.ap233.org/ap233-public-information/reference/20080008301_2008008500.pdf

Naur, P., Backus, J. W., Bauer, F. L., & Green, J. (1963). *Report on the algorithmic language ALGOL 60.*

OMG. (2008). OMG Systems Modeling Language (OMG SysML ™) 1.1. *Source*. OMG. Retrieved from http://www.omg.org/spec/SysML/1.1

OMG. (2009). OMG Unified Modeling Language TM (OMG UML), Superstructure. OMG. Retrieved from http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

OMG. (2011a). OMG Unified Modeling Language (OMG UML), Infrastructure. Retrieved from http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/

OMG. (2011b). MOF 2.0/XMI Mapping Specification. OMG. Retrieved from http://www.omg.org/spec/XMI/2.4.1/

Oberle, D. (2006). Semantic Management of Middleware. *Semantic Web and Beyond: Computing for Human Experience* (1st ed.).

Oliveira, K., Breitman, K., & Oliveira, T. (2009). Ontology Aided Model Comparison. *14th IEEE International Conference on Engineering of Complex Computer Systems*.

Oppenheim, B. W. (2009). Lean enablers for systems engineering. *Crosstalk Defense Journal*, (July/August 2009), n/a-n/a. doi:10.1002/sys.20161

PDTec GmbH. (1998, March). ECCO Toolkit. *Strategic Analysis*. Karlsruhe: PDTec GmbH. doi:10.1080/09700168109428631

Paredis, C. J. J., Bernard, Y., Koning, R. M. B. H.-peter D., & Friedenthal, S. (2010). An Overview of the SysML-Modelica Transformation Specification. *Jet Propulsion*.

Pierra, Guy. (1992). Modelling Classes of Pre-existing Components in a CIM Perspective: the ISO 13584 Approach. *Revue International de CFAO et d'Infographie*, *9*(3), 435-454.

Pierra, Guy. (2008). Context Representation in Domain Ontologies. *Journal on Data Semantics X*, *1*(4900), 174-211.

Pop, A., Akhvlediani, D., & Fritzson, P. (2007). Towards Unified System Modeling with the ModelicaML UML Profile. *In Proceedings of the 1st International Workshop on EquationBased ObjectOriented Languages and Tools EOOLT'07*, 13-

24. Retrieved from http://www.ep.liu.se/ecp/024/002/ecp2407002.pdf

Pratt, M. J. (2001). *Introduction to ISO 10303—the STEP Standard for Product Data Exchange. Journal of Computing and Information Science in Engineering* (Vol. 1, p. 102). doi:10.1115/1.1354995

Ruzzi, M. (2004). *Data Integration Issues in Research Supporting Sustainable Natural Resource Management. Geographical Research* (Vol. 24, pp. 230-386). Roma. doi:10.1111/j.1745-5871.2007.00476.x

Seng, J.-lang, & Kong, I. L. (2009). A schema and ontology-aided intelligent information integration. *Expert Systems With Applications*, *36*(7), 10538-10550. Elsevier Ltd. doi:10.1016/j.eswa.2009.02.067

Sheth, A. P., & Larson, J. A. (1990). Federated Database Systems for Managing Distributed , Heterogeneous , and Autonomous Databases. *ACM Computing Surveys*, *22*(3).

Sheth, A., Ramakrishnan, C., & Thomas, C. (2005). Semantics for the Semantic Web: The Implicit, the Formal and the Powerful. *International Journal on Semantic Web & Information Systems*, *1*(1), 1-18.

Silva, C. D. A. F. D. (2007). *Découverte de correspondances sémantiques entre ressources hétérogènes dans un environnement coopératif*. UNIVERSITE CLAUDE BERNARD - LYON 1.

Simon Zayas, D., Monceaux, A., & Ait-Ameur, Y. (2010). Knowledge models to reduce the gap between heterogeneous models Application to aircraft systems engineering. *15th IEEE International Conference on Engineering of Complex Computer Systems - UML&AADL Workshop*. Oxford: IEEE Computer Society.

Simon Zayas, D., Monceaux, A., & Ait-Ameur, Y. (2011). Using knowledge and expressions to validate inter-model constraints. *18th World Congress IFAC 2011*. IFAC.

Simon Zayas, D., Monceaux, A., & Ait-ameur, Y. (2011). Knowledge Based Characterization Of Cross-Models Constraints To Check Design And Modeling Requirements. *DASIA 2011*. San Anton, Malta: EUROSPACE.

Sirin, E., & Parsia, B. (2004). Pellet: An OWL DL Reasoner. In V. Haarslev & Ralf Möller (Eds.), *Description Logics*. CEUR-WS.org.

Snodgrass, T., & Kassi, M. (1986). Function Analysis The Stepping Stones to Good Value. University of Wisconsin-Madison.

Spiby, P. (2007). STEP AP233 Systems Engineering. *Systems Engineering*. Eurostep Group. Retrieved from http://www2.pdteurope.com/media/54159/1b. step ap233 systems engineering.pdf

Sweet, C. N. (2004). The C2 Constellation A US Air Force Network Centric Warfare Program Network Centric Applications and C4ISR Architecture. *Info*, 1-31.

Team, J. (2011). Jena - A Semantic Web Framework for Java. Retrieved from http://jena.sourceforge.net/index.html

Technologies, E. (2011a). SCADE Suite. Retrieved a from http://www.esterel-

technologies.com/products/scade-suite/

Technologies, E. (2011b). SCADE System. Retrieved b from http://www.esterel-technologies.com/products/scade-system

Tenorio, C. D., Mavris, D., Garcia, E., & Armstrong, M. (2008). Methodology for Aircraft System Architecture Sizing. *ICAS*. ICAS.

Tolvanen, J.-P., & Kelly, S. (2008). *Domain-Specific modeling: Enabling Full Code Generation* (p. 254). New Jersey, USA: Wiley-IEEE Press.

Topcased.org. (2011). TOPCASED The Open-Source Toolkit for Critical Systems. Retrieved from http://www.topcased.org

Tran, T. N., Khan, K. M., & Lan, Y.-C. (2004). A Framework for Transforming Artifacts from Data Flow Diagrams to UML. *Proceeding of the IASTED International Conference on Software Engineering (SE 2004)*. ACTA Press.

Tudorache, T. (2006). *Employing Ontologies for an Improved Development Process in Collaborative Engineering*. Elektrotechnik und Informatik at Berlin.

USAF. (1969). MIL-STD-499 System Engineering Management. USAF.

Uren, V., Hall, W., & Keynes, M. (2006). Semantic Annotation for Knowledge Management : Requirements and a Survey of the State of the Art. *Web Semantics: Science, Services and Agents on the World Wide Web*, *4*(1), 14-28. doi:10.1016/j.websem.2005.10.002

Uschold, M. (2002). Where are the Semantics in the Semantic Web ? *AI Magazine*.

Vajna, S. (2002). Approaches of Knowledge-based Design. *Proceedings of the International Design Conference*.

Verries, J. (2010). *Approche pour la Conception de Systèmes Aéronautiques Innovants en Vue d'Optimiser l'Architecture Application au Système Portes Passagers*. Université de Toulouse.

Vinoski, S. (1997). CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine*, *35*(2), 46-55. doi:10.1109/35.565655

Vitech Corporation. (2007a). System Definition Guide. Vitech Corporation.

Vitech Corporation. (2007b). Architecture Definition Guide (DoDAV v1.5). Vitech Corporation.

Vitech Corporation. (2011). A Primer for Model-Based Systems Engineering. Vitech Corporation.

W3C. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Recommendation*. Retrieved from http://www.w3.org/TR/rdf-schema/

W3C. (2008). Cool URIs for the Semantic Web. *W3C Interest Group Note*. Retrieved from http://www.w3.org/TR/cooluris/#hashuri

Warner, J. B., & Kleppe, A. G. (1998). *The Object Constraint Language: Precise Modeling with UML* (p. 144). Addison-Wesley.

Yoshimura, M. (2007). System Design Optimization for Product Manufacturing. *Concurrent Engineering*, *15*(4), 329-343. doi:10.1177/1063293X07083087

Zouggar, N., Vallespir, B., & Chen, D. (2008). Semantic Enrichment of Enterprise Models. *IWEI 2008*. IEEE.

Annex A

This annex contains the general approach models described in Chapter V. They are formalized in EXPRESS modeling language.

TOP_SCHEMA

```
--THIS SCHEMA CONTAINS THE ELEMENTS IN COMMON TO THE REST OF SCHEMAS
SCHEMA TOP_SCHEMA;

ENTITY T_DATE;
      DAY: INTEGER; --BETWEEN 1 AND 31
      MONTH: INTEGER;   --BETWEEN 1 AND 12
      YEAR: INTEGER; --BETWEEN -9999 AND 9999
      HOUR: OPTIONAL INTEGER; --BETWEEN 0 AND 23
      MINUTE: OPTIONAL INTEGER;-- BETWEEN 0 AND 59
      SECOND: OPTIONAL INTEGER; --BETWEEN 0 AND 59
END_ENTITY;

--THE MAIN ELEMENT, AN ENTITY
ENTITY ENTITY_CLASS
      ABSTRACT SUPERTYPE;
      NAME: OPTIONAL STRING; --IDENTIFIER OF THE ENTITY
END_ENTITY;

-- TYPES
TYPE T_NUMBER = NUMBER;
END_TYPE;

TYPE T_BOOLEAN = BOOLEAN;
END_TYPE;

TYPE T_STRING = STRING;
END_TYPE;

-- TYPE DOMAINE
TYPE T_DOMAINE = SELECT (T_NUMBER,T_BOOLEAN,T_STRING,T_DATE,ENTITY_CLASS) ;
END_TYPE;

--THIS ENTITY REPRESENTS A MODEL LANGUAGE
ENTITY MODELING_LANGUAGE
      ABSTRACT SUPERTYPE;
END_ENTITY;

--THIS ENTITY REPRESENTS ONE MODEL
ENTITY MODEL
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CLASS);
      DESCRIPTION: OPTIONAL STRING;
      CREATION: T_DATE;
      LAST_MODIFICATION: OPTIONAL T_DATE;
      MODELING_LANGUAGE: MODELING_LANGUAGE;
END_ENTITY;

END_SCHEMA;
```

ANNOTATION_SCHEMA

```
--THIS SCHEMA CONTAINS THE ENTITIES NEEDED FOR THE ANNOTATION
SCHEMA ANNOTATION_SCHEMA;
 (********************** KNOWLEDGE **************************)

--THIS ENTITY REPRESENTS AN UNIQUE IDENTIFIER (URI)
ENTITY URI;
      URI_VALUE: STRING;
INVERSE
      THE_CLASS: KNOWLEDGE_CLASS FOR MY_URI;
UNIQUE
      URI: URI_VALUE;
END_ENTITY;

ENTITY ANNOTATION_CLASS;
      NAME: STRING;
      MY_KNOWLEDGE: LIST OF URI;
      MY_ENTITIES: LIST OF ENTITY_CLASS;
END_ENTITY;


ENTITY KNOWLEDGE_CLASS
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CLASS);
      MY_URI: URI;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'KNOWLEDGE_CLASS';
END_ENTITY;

ENTITY KNOWLEDGE_LITERAL_CLASS
      ABSTRACT SUPERTYPE
      SUBTYPE OF (KNOWLEDGE_CLASS);
      THE_VALUE: T_DOMAINE;
END_ENTITY;

ENTITY KNOWLEDGE_LITERAL_STRING
      SUBTYPE OF (KNOWLEDGE_LITERAL_CLASS);
      SELF\KNOWLEDGE_LITERAL_CLASS.THE_VALUE:STRING;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'KNOWLEDGE_LITERAL_STRING';
END_ENTITY;

ENTITY KNOWLEDGE_LITERAL_NUMERIC
      SUBTYPE OF (KNOWLEDGE_LITERAL_CLASS);
      SELF\KNOWLEDGE_LITERAL_CLASS.THE_VALUE:NUMBER;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'KNOWLEDGE_LITERAL_NUMERIC';
END_ENTITY;

ENTITY KNOWLEDGE_LITERAL_BOOLEAN
      SUBTYPE OF (KNOWLEDGE_LITERAL_CLASS);
      SELF\KNOWLEDGE_LITERAL_CLASS.THE_VALUE:BOOLEAN;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'KNOWLEDGE_LITERAL_BOOLEAN';
END_ENTITY;

END_SCHEMA;
```

RELATION_SCHEMA

```
--THIS SCHEMA CONTAINS THE ENTITIES REPRESENTING THE INTER-MODEL RELATIONS
SCHEMA RELATION_SCHEMA;

(**********************************************************************)
(********************** RELATIONS **************************)
(**********************************************************************)

--THIS ENTITY REPRESENTS THE ATTRIBUTS WHICH CAN BE ADDED TO A RELATION
ENTITY ATTRIBUTE_RELATION;
      NAME: STRING;
      REPRESENTATION_ENTITY: OPTIONAL ENTITY_CLASS;
      REPRESENTATION_STRING: OPTIONAL STRING;
      REPRESENTATION_INTEGER: OPTIONAL INTEGER;
      REPRESENTATION_REAL: OPTIONAL REAL;
      REPRESENTATION_BOOLEAN: OPTIONAL BOOLEAN;
      REPRESENTATION_DATE: OPTIONAL T_DATE;
      --CONSTRAINTS
      --ONLY ONE OF THE REPRESENTATION ITEMS CAN HAVE A VALUE
END_ENTITY;

--THIS ENTITY REPRESENTS A RELATION BETWEEN 2 OR MORE ELEMENTS OF DIFFERENT
MODELS
ENTITY RELATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CLASS); --WE ALLOW RELATIONS BETWEEN RELATIONS!
      ENTITY_ORIGIN:  SET [1:?] OF ENTITY_CLASS;
      ENTITY_DESTINATION:  SET [1:?] OF ENTITY_CLASS;
      ATTRIBUTS: SET[0:?] OF ATTRIBUTE_RELATION;
END_ENTITY;

--THIS ENTITY GROUPS THE RELATIONS OF KIND SET
ENTITY SET_RELATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF(RELATION);
END_ENTITY;

--THIS ENTITY GROUPS THE RELATIONS OF KIND LOGICAL
ENTITY LOGICAL_RELATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF(RELATION);
END_ENTITY;

--THIS ENTITY GROUPS THE RELATIONS CONCERNING THE BEHAVIOR
ENTITY BEHAVIOR_RELATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF(RELATION);
END_ENTITY;

--THIS ENTITY GROUPS THE RELATIONS DERIVED FROM THE DESIGN PROCESS
ENTITY DESIGN_RELATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF(RELATION);
END_ENTITY;
```

```
--ENTITY_ORIGIN IS COMPOSED OF ENTITY_DESTINATION
ENTITY COMPOSITION
      SUBTYPE OF(SET_RELATION);
END_ENTITY;

--ENTITY_ORIGIN GROUPS ENTITY_DESTINATION
ENTITY AGGREGATION
      SUBTYPE OF(SET_RELATION);
END_ENTITY;

--ENTITY_ORIGIN SAME AS ENTITY_DESTINATION
ENTITY EQUIVALENCE
      SUBTYPE OF(LOGICAL_RELATION);
END_ENTITY;

--ENTITY_ORIGIN INTERFACES WITH ENTITY_DESTINATION
ENTITY INTERFACE
      SUBTYPE OF(BEHAVIOR_RELATION);
END_ENTITY;

--ENTITY_ORIGIN TRIGGERS ENTITY_DESTINATION
ENTITY TRIGGER
      SUBTYPE OF(INTERFACE);
END_ENTITY;

--ENTITY_ORIGIN INPUTS ENTITY_DESTINATION
ENTITY SENDING
      SUBTYPE OF(INTERFACE);
END_ENTITY;

--ENTITY_ORIGIN RECEIVES FROM ENTITY_DESTINATION
ENTITY RECEPTION
      SUBTYPE OF(INTERFACE);
END_ENTITY;

--ENTITY_ORIGIN SPECIFIES WITH ENTITY_DESTINATION
ENTITY SPECIFICATION
      SUBTYPE OF(BEHAVIOR_RELATION);
END_ENTITY;

END_SCHEMA;
```

EXPRESSION_SCHEMA

--THIS SCHEMA CONTAINS THE ENTITIES USED TO BUILD EXPRESSIONS

```
SCHEMA EXPRESSION_SCHEMA;

REFERENCE FROM TOP_SCHEMA;
REFERENCE FROM META_SCHEMA;
```

(***)
(*********************** EXPRESSIONS **************************)
(***)

--THIS ENTITY IS THE MAIN ENTRY TO REPRESENT CONSTRAINTS, BASED ON ISO TC184/SC4/WG2 N 375

```
ENTITY GENERIC_EXPRESSION
ABSTRACT SUPERTYPE
OF(ONEOF(SIMPLE_GENERIC_EXPRESSION,UNARY_GENERIC_EXPRESSION,BINARY_GENERIC_
EXPRESSION,MULTIPLE_ARITY_GENERIC_EXPRESSION));
WHERE
      WR1: IS_ACYCLIC(SELF);
END_ENTITY;


ENTITY SIMPLE_GENERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF(GENERIC_LITERAL, GENERIC_VARIABLE))
SUBTYPE OF (GENERIC_EXPRESSION);
END_ENTITY;

ENTITY GENERIC_LITERAL
ABSTRACT SUPERTYPE
SUBTYPE OF (SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY GENERIC_VARIABLE
ABSTRACT SUPERTYPE
SUBTYPE OF (SIMPLE_GENERIC_EXPRESSION);
      IDENTIFIER: STRING;
INVERSE
      INTERPRETATION : ENVIRONMENT FOR SYNTACTIC_REPRESENTATION;
END_ENTITY;
```

--A VARIABLE_SEMANTICS ENTITY IS USED TO REPRESENT THE MEANING OF A GENERIC_VARIABLE.
--IT IS AN ABSTRACT SUPERTYPE THAT SHALL BE SUBTYPED WHEREVER A VARIABLE_SEMANTICS IS USED.
--A VARIABLE_SEMANTICS SHALL SPECIFY THE CONTEXT WITHIN WHICH THE VARIABLE SHALL BE USED TOGETHER WITH THE INTERPRETATION
--FUNCTION THAT ASSOCIATES A VALUE WITH THIS VARIABLE

```
ENTITY VARIABLE_SEMANTICS
ABSTRACT SUPERTYPE;
END_ENTITY;

ENTITY ENVIRONMENT_VAR;
      SYNTACTIC_REPRESENTATION: OPTIONAL GENERIC_VARIABLE;
      SEMANTICS: OPTIONAL VARIABLE_SEMANTICS;
END_ENTITY;
```

```
ENTITY UNARY_GENERIC_EXPRESSION
ABSTRACT SUPERTYPE
SUBTYPE OF(GENERIC_EXPRESSION);
      OPERAND: GENERIC_EXPRESSION;
END_ENTITY;

ENTITY BINARY_GENERIC_EXPRESSION
ABSTRACT SUPERTYPE
SUBTYPE OF(GENERIC_EXPRESSION);
      OPERANDS: LIST [2:2] OF GENERIC_EXPRESSION;
END_ENTITY;

ENTITY MULTIPLE_ARITY_GENERIC_EXPRESSION
ABSTRACT SUPERTYPE
SUBTYPE OF(GENERIC_EXPRESSION);
      OPERANDS: LIST [2:?] OF GENERIC_EXPRESSION;
END_ENTITY;

ENTITY EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(NUMERIC_EXPRESSION,BOOLEAN_EXPRESSION,STRING_EXPRESSION))
SUBTYPE OF (GENERIC_EXPRESSION);
END_ENTITY;

ENTITY VARIABLE
ABSTRACT SUPERTYPE OF (ONEOF
(NUMERIC_VARIABLE,BOOLEAN_VARIABLE,STRING_VARIABLE))
SUBTYPE OF(GENERIC_VARIABLE);
END_ENTITY;

ENTITY DEFINED_FUNCTION
ABSTRACT SUPERTYPE OF ((ONEOF (NUMERIC_DEFINED_FUNCTION,
                        STRING_DEFINED_FUNCTION,
                        BOOLEAN_DEFINED_FUNCTION)
                        )
                        );
END_ENTITY;
```

--NUMERIC SECTION

```
ENTITY NUMERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(SIMPLE_NUMERIC_EXPRESSION,UNARY_NUMERIC_EXPRESSION,BINARY_NUMERIC_EXPRESSI
ON,MULTIPLE_ARITY_NUMERIC_EXPRESSION,
                        LENGTH_FUNCTION, VALUE_FUNCTION,
NUMERIC_DEFINED_FUNCTION))
SUBTYPE OF (EXPRESSION);
      THE_VALUE: OPTIONAL NUMBER;
DERIVE
      IS_INT: BOOLEAN := IS_INT_EXPR (SELF);
END_ENTITY;

ENTITY SIMPLE_NUMERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (LITERAL_NUMBER, NUMERIC_VARIABLE))
SUBTYPE OF (NUMERIC_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY LITERAL_NUMBER
ABSTRACT SUPERTYPE OF (ONEOF (INT_LITERAL, REAL_LITERAL))
SUBTYPE OF (SIMPLE_NUMERIC_EXPRESSION, GENERIC_LITERAL);
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER;
```

```
END_ENTITY;

ENTITY INT_LITERAL
SUBTYPE OF (LITERAL_NUMBER);
      SELF\LITERAL_NUMBER.THE_VALUE: INTEGER;
END_ENTITY;

ENTITY REAL_LITERAL
SUBTYPE OF (LITERAL_NUMBER);
      SELF\LITERAL_NUMBER.THE_VALUE: REAL;
END_ENTITY;

ENTITY NUMERIC_VARIABLE
SUPERTYPE OF (ONEOF (INT_NUMERIC_VARIABLE, REAL_NUMERIC_VARIABLE))
SUBTYPE OF (SIMPLE_NUMERIC_EXPRESSION, VARIABLE);
WHERE
      WR1:  ('EXPRESSION_SCHEMA.INT_NUMERIC_VARIABLE'
                  IN TYPEOF(SELF) ) OR
                  ('EXPRESSION_SCHEMA.REAL_NUMERIC_VARIABLE'
                  IN TYPEOF(SELF) );
END_ENTITY;

ENTITY INT_NUMERIC_VARIABLE
SUBTYPE OF (NUMERIC_VARIABLE);
END_ENTITY;

ENTITY REAL_NUMERIC_VARIABLE
SUBTYPE OF (NUMERIC_VARIABLE);
END_ENTITY;

ENTITY UNARY_NUMERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (UNARY_FUNCTION_CALL))
SUBTYPE OF (NUMERIC_EXPRESSION, UNARY_GENERIC_EXPRESSION);
      SELF\UNARY_GENERIC_EXPRESSION.OPERAND    : NUMERIC_EXPRESSION;
END_ENTITY;

ENTITY BINARY_NUMERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(MINUS_EXPRESSION,DIV_EXPRESSION,MOD_EXPRESSION,SLASH_EXPRESSION,POWER_EXPR
ESSION))
SUBTYPE OF (NUMERIC_EXPRESSION, BINARY_GENERIC_EXPRESSION);
      SELF\BINARY_GENERIC_EXPRESSION.OPERANDS : LIST [2:2] OF
NUMERIC_EXPRESSION;
END_ENTITY;

ENTITY MULTIPLE_ARITY_NUMERIC_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (PLUS_EXPRESSION,
MULT_EXPRESSION,MULTIPLE_ARITY_FUNCTION_CALL))
SUBTYPE OF (NUMERIC_EXPRESSION, MULTIPLE_ARITY_GENERIC_EXPRESSION);
      SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS: LIST [2:?] OF
NUMERIC_EXPRESSION;
END_ENTITY;

ENTITY LENGTH_FUNCTION
SUBTYPE OF (NUMERIC_EXPRESSION, UNARY_GENERIC_EXPRESSION);
      SELF\UNARY_GENERIC_EXPRESSION.OPERAND: STRING_EXPRESSION;
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: INTEGER := LENGTH_FCT(SELF);
END_ENTITY;

ENTITY VALUE_FUNCTION
```

```
SUPERTYPE OF (INT_VALUE_FUNCTION)
SUBTYPE OF (NUMERIC_EXPRESSION, UNARY_GENERIC_EXPRESSION);
      SELF\UNARY_GENERIC_EXPRESSION.OPERAND: STRING_EXPRESSION;
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := VALUE_FCT(SELF);
END_ENTITY;

ENTITY INT_VALUE_FUNCTION
SUBTYPE OF (VALUE_FUNCTION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: INTEGER := INT_VALUE_FCT(SELF);
END_ENTITY;

ENTITY NUMERIC_DEFINED_FUNCTION
ABSTRACT SUPERTYPE OF (ONEOF (INTEGER_DEFINED_FUNCTION,
                        REAL_DEFINED_FUNCTION))
SUBTYPE OF (NUMERIC_EXPRESSION, DEFINED_FUNCTION);
END_ENTITY;

ENTITY PLUS_EXPRESSION
SUBTYPE OF (MULTIPLE_ARITY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := PLUS_FUNCTION(SELF);
END_ENTITY;

ENTITY MULT_EXPRESSION
SUBTYPE OF (MULTIPLE_ARITY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MULT_FUNCTION(SELF);
END_ENTITY;

ENTITY MINUS_EXPRESSION
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MINUS_FCT(SELF);
END_ENTITY;

ENTITY DIV_EXPRESSION
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := DIV_FCT(SELF);
END_ENTITY;

ENTITY MOD_EXPRESSION
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MOD_FCT(SELF);
END_ENTITY;

ENTITY SLASH_EXPRESSION
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := SLASH_FCT(SELF);
END_ENTITY;

ENTITY POWER_EXPRESSION
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := POWER_FCT(SELF);
END_ENTITY;
```

```
ENTITY UNARY_FUNCTION_CALL
ABSTRACT SUPERTYPE
                        OF (ONEOF (
                        ABS_FUNCTION,
                        MINUS_FUNCTION,
                        SQUARE_ROOT_FUNCTION
                        ))
SUBTYPE OF (UNARY_NUMERIC_EXPRESSION);
END_ENTITY;

ENTITY BINARY_FUNCTION_CALL
ABSTRACT SUPERTYPE
SUBTYPE OF (BINARY_NUMERIC_EXPRESSION);
END_ENTITY;

ENTITY MULTIPLE_ARITY_FUNCTION_CALL
ABSTRACT SUPERTYPE OF (ONEOF (MAXIMUM_FUNCTION,
                        MINIMUM_FUNCTION))
SUBTYPE OF (MULTIPLE_ARITY_NUMERIC_EXPRESSION);
END_ENTITY;

ENTITY ABS_FUNCTION
SUBTYPE OF (UNARY_FUNCTION_CALL);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := ABS_FCT(SELF);
END_ENTITY;

ENTITY SQUARE_ROOT_FUNCTION
SUBTYPE OF (UNARY_FUNCTION_CALL);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := SQUARE_FCT(SELF);
END_ENTITY;

ENTITY MINUS_FUNCTION
SUBTYPE OF (UNARY_FUNCTION_CALL);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MINUS_UNARY_FCT(SELF);
END_ENTITY;

ENTITY MAXIMUM_FUNCTION
SUBTYPE OF (MULTIPLE_ARITY_FUNCTION_CALL);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MAXIMUM_FCT(SELF);
END_ENTITY;

ENTITY MINIMUM_FUNCTION
SUBTYPE OF (MULTIPLE_ARITY_FUNCTION_CALL);
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: NUMBER := MINIMUM_FCT(SELF);
END_ENTITY;

ENTITY INTEGER_DEFINED_FUNCTION
ABSTRACT SUPERTYPE
SUBTYPE OF (NUMERIC_DEFINED_FUNCTION);
END_ENTITY ;

ENTITY REAL_DEFINED_FUNCTION
ABSTRACT SUPERTYPE
SUBTYPE OF (NUMERIC_DEFINED_FUNCTION);
END_ENTITY ;
```

--BOOLEAN SECTION

```
ENTITY BOOLEAN_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(SIMPLE_BOOLEAN_EXPRESSION,UNARY_BOOLEAN_EXPRESSION,BINARY_BOOLEAN_EXPRESSI
ON,MULTIPLE_ARITY_BOOLEAN_EXPRESSION,COMPARISON_EXPRESSION,
                        INTERVAL_EXPRESSION,BOOLEAN_DEFINED_FUNCTION))
SUBTYPE OF (EXPRESSION);
      THE_VALUE: OPTIONAL BOOLEAN;
END_ENTITY;

ENTITY SIMPLE_BOOLEAN_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (BOOLEAN_LITERAL,
                        BOOLEAN_VARIABLE))
SUBTYPE OF (BOOLEAN_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY BOOLEAN_LITERAL
SUBTYPE OF (SIMPLE_BOOLEAN_EXPRESSION, GENERIC_LITERAL);
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN;
END_ENTITY;

ENTITY BOOLEAN_VARIABLE
SUBTYPE OF (SIMPLE_BOOLEAN_EXPRESSION, VARIABLE);
END_ENTITY;

ENTITY UNARY_BOOLEAN_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (NOT_EXPRESSION, ODD_FUNCTION))
SUBTYPE OF (BOOLEAN_EXPRESSION, UNARY_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY NOT_EXPRESSION
SUBTYPE OF (UNARY_BOOLEAN_EXPRESSION);
SELF\UNARY_GENERIC_EXPRESSION.OPERAND: BOOLEAN_EXPRESSION;
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := NOT_FCT(SELF);
END_ENTITY;

ENTITY ODD_FUNCTION
SUBTYPE OF (UNARY_BOOLEAN_EXPRESSION);
SELF\UNARY_GENERIC_EXPRESSION.OPERAND: NUMERIC_EXPRESSION;
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := ODD_FCT(SELF);
END_ENTITY;

ENTITY BINARY_BOOLEAN_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (XOR_EXPRESSION, EQUALS_EXPRESSION))
SUBTYPE OF (BOOLEAN_EXPRESSION, BINARY_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY MULTIPLE_ARITY_BOOLEAN_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (AND_EXPRESSION, OR_EXPRESSION))
SUBTYPE OF (BOOLEAN_EXPRESSION, MULTIPLE_ARITY_GENERIC_EXPRESSION);
      SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS: LIST [2:?] OF
BOOLEAN_EXPRESSION;
END_ENTITY;

ENTITY XOR_EXPRESSION
SUBTYPE OF (BINARY_BOOLEAN_EXPRESSION);
      SELF\BINARY_GENERIC_EXPRESSION.OPERANDS: LIST [2:2] OF
BOOLEAN_EXPRESSION;
```

```
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := XOR_FCT(SELF);
END_ENTITY;

ENTITY EQUALS_EXPRESSION
SUBTYPE OF (BINARY_BOOLEAN_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := EQUALS_FCT(SELF);
END_ENTITY;

ENTITY AND_EXPRESSION
SUBTYPE OF (MULTIPLE_ARITY_BOOLEAN_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := AND_FCT(SELF);
END_ENTITY;

ENTITY OR_EXPRESSION
SUBTYPE OF (MULTIPLE_ARITY_BOOLEAN_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := OR_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(COMPARISON_EQUAL,COMPARISON_GREATER,COMPARISON_GREATER_EQUAL,COMPARISON_LE
SS,COMPARISON_LESS_EQUAL,COMPARISON_NOT_EQUAL,LIKE_EXPRESSION))
SUBTYPE OF (BOOLEAN_EXPRESSION, BINARY_GENERIC_EXPRESSION);
      SELF\BINARY_GENERIC_EXPRESSION.OPERANDS  : LIST [2:2] OF EXPRESSION;
END_ENTITY;

ENTITY COMPARISON_EQUAL
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_EQUAL_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_GREATER
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_GREATER_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_GREATER_EQUAL
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_GREATER_EQUAL_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_LESS
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_LESS_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_LESS_EQUAL
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
```

```
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_LESS_EQUAL_FCT(SELF);
END_ENTITY;

ENTITY COMPARISON_NOT_EQUAL
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
COMPARISON_NOT_EQUAL_FCT(SELF);
END_ENTITY;

ENTITY LIKE_EXPRESSION
SUBTYPE OF (COMPARISON_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := LIKE_FCT(SELF);
WHERE
      WR1: ('EXPRESSION_SCHEMA.STRING_EXPRESSION'
            IN TYPEOF(SELF\COMPARISON_EXPRESSION.OPERANDS[1])) AND
            ('EXPRESSION_SCHEMA.STRING_EXPRESSION'
            IN TYPEOF(SELF\COMPARISON_EXPRESSION.OPERANDS[2]));
END_ENTITY;

ENTITY INTERVAL_EXPRESSION
SUBTYPE OF (BOOLEAN_EXPRESSION, MULTIPLE_ARITY_BOOLEAN_EXPRESSION) ;
DERIVE
      INTERVAL_LOW: GENERIC_EXPRESSION
                  := SELF\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[1];
      INTERVAL_ITEM: GENERIC_EXPRESSION
                  := SELF\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[2];
      INTERVAL_HIGH: GENERIC_EXPRESSION
                  := SELF\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[3];
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN := INTERVAL_FCT(SELF);
WHERE
      WR1:('EXPRESSION_SCHEMA.EXPRESSION'
                        IN TYPEOF(INTERVAL_LOW))
            AND ('EXPRESSION_SCHEMA.EXPRESSION'
                        IN TYPEOF(INTERVAL_ITEM) )
            AND ('EXPRESSION_SCHEMA.EXPRESSION'
                        IN TYPEOF(INTERVAL_HIGH));
      WR2:(('EXPRESSION_SCHEMA.STRING_EXPRESSION'
                        IN TYPEOF (SELF.INTERVAL_LOW))
                  AND ('EXPRESSION_SCHEMA.STRING_EXPRESSION'
                        IN TYPEOF (SELF.INTERVAL_HIGH))
                  AND ('EXPRESSION_SCHEMA.STRING_EXPRESSION'
                        IN TYPEOF (SELF.INTERVAL_ITEM)))
            OR
            (('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION'
                        IN TYPEOF(SELF.INTERVAL_LOW))
                  AND ('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION'
                        IN TYPEOF(SELF.INTERVAL_ITEM))
                  AND ('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION'
                        IN TYPEOF(SELF.INTERVAL_HIGH)));
END_ENTITY;

ENTITY BOOLEAN_DEFINED_FUNCTION
ABSTRACT SUPERTYPE
SUBTYPE OF (DEFINED_FUNCTION, BOOLEAN_EXPRESSION);
END_ENTITY ;
```

--STRING SECTION

```
ENTITY STRING_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF
(SIMPLE_STRING_EXPRESSION,INDEX_EXPRESSION,SUBSTRING_EXPRESSION,CONCAT_EXPR
ESSION,FORMAT_FUNCTION,STRING_DEFINED_FUNCTION))
SUBTYPE OF (EXPRESSION);
      THE_VALUE: OPTIONAL STRING;
END_ENTITY;

ENTITY SIMPLE_STRING_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (STRING_LITERAL, STRING_VARIABLE))
SUBTYPE OF (STRING_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

ENTITY STRING_LITERAL
SUBTYPE OF (SIMPLE_STRING_EXPRESSION, GENERIC_LITERAL);
      SELF\STRING_EXPRESSION.THE_VALUE: STRING;
END_ENTITY;

ENTITY STRING_VARIABLE
SUBTYPE OF (SIMPLE_STRING_EXPRESSION, VARIABLE);
END_ENTITY;

ENTITY INDEX_EXPRESSION
SUBTYPE OF (STRING_EXPRESSION, BINARY_GENERIC_EXPRESSION);
DERIVE
      OPERAND:GENERIC_EXPRESSION:=
SELF\BINARY_GENERIC_EXPRESSION.OPERANDS[1];
      INDEX:GENERIC_EXPRESSION:=
SELF\BINARY_GENERIC_EXPRESSION.OPERANDS[2];
      SELF\STRING_EXPRESSION.THE_VALUE: STRING := INDEX_FCT(SELF);
WHERE
      WR1: ('EXPRESSION_SCHEMA.STRING_EXPRESSION' IN TYPEOF(OPERAND))
            AND ('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION'
                  IN TYPEOF(INDEX));
      WR2: IS_INT_EXPR (INDEX);
END_ENTITY;

ENTITY SUBSTRING_EXPRESSION
SUBTYPE OF (STRING_EXPRESSION, MULTIPLE_ARITY_GENERIC_EXPRESSION);
DERIVE
      OPERAND:GENERIC_EXPRESSION:=SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OP
ERANDS[1];
      INDEX1:GENERIC_EXPRESSION:=
      SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS[2];
      INDEX2:GENERIC_EXPRESSION:=
      SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS[3];
      SELF\STRING_EXPRESSION.THE_VALUE: STRING := SUBSTRING_FCT(SELF);
WHERE
      WR1: ('EXPRESSION_SCHEMA.STRING_EXPRESSION' IN TYPEOF(OPERAND))
                  AND ('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION'    IN
TYPEOF(INDEX1))
                  AND ('EXPRESSION_SCHEMA.EXPRESSIONS_SCHEMA' IN
TYPEOF(INDEX2));
      WR2: SIZEOF(SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS)=3;
      WR3: IS_INT_EXPR (INDEX1);
      WR4: IS_INT_EXPR (INDEX2);
END_ENTITY;

ENTITY CONCAT_EXPRESSION
SUBTYPE OF (STRING_EXPRESSION, MULTIPLE_ARITY_GENERIC_EXPRESSION);
```

```
      SELF\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS: LIST [2 : ?] OF
STRING_EXPRESSION;
DERIVE
      SELF\STRING_EXPRESSION.THE_VALUE: STRING := CONCAT_FCT(SELF);
END_ENTITY;


ENTITY FORMAT_FUNCTION
SUBTYPE OF (STRING_EXPRESSION, BINARY_GENERIC_EXPRESSION);
DERIVE
      VALUE_TO_FORMAT: GENERIC_EXPRESSION:=
SELF\BINARY_GENERIC_EXPRESSION.OPERANDS[1];
      FORMAT_STRING: GENERIC_EXPRESSION:=
SELF\BINARY_GENERIC_EXPRESSION.OPERANDS[2];
      SELF\STRING_EXPRESSION.THE_VALUE: STRING := FORMAT_FCT(SELF);
WHERE
      WR1: (('EXPRESSION_SCHEMA.NUMERIC_EXPRESSION') IN
TYPEOF(VALUE_TO_FORMAT))
                  AND (('EXPRESSION_SCHEMA.STRING_EXPRESSION') IN
TYPEOF(FORMAT_STRING));
END_ENTITY;

ENTITY STRING_DEFINED_FUNCTION
ABSTRACT SUPERTYPE
SUBTYPE OF (DEFINED_FUNCTION, STRING_EXPRESSION);
END_ENTITY ;
```

(***)
(***** GENERAL FUNCTIONS *****)
(***)

```
FUNCTION IS_INT_EXPR (ARG: NUMERIC_EXPRESSION) : BOOLEAN;

LOCAL
      I: INTEGER;
END_LOCAL;

IF  'EXPRESSION_SCHEMA.INT_LITERAL' IN TYPEOF(ARG)
THEN
      RETURN (TRUE);
END_IF;
IF  'EXPRESSION_SCHEMA.REAL_LITERAL'   IN TYPEOF(ARG)
THEN
      RETURN (FALSE);
END_IF;
IF  'EXPRESSION_SCHEMA.INT_NUMERIC_VARIABLE'   IN TYPEOF(ARG)
THEN
      RETURN (TRUE);
END_IF;
IF  'EXPRESSION_SCHEMA.REAL_NUMERIC_VARIABLE'   IN TYPEOF(ARG)
THEN
      RETURN (FALSE);
END_IF;
IF  'EXPRESSION_SCHEMA.ABS_FUNCTION'   IN TYPEOF(ARG)
THEN
      RETURN (IS_INT_EXPR(ARG\UNARY_NUMERIC_EXPRESSION.OPERAND));
END_IF;
IF  'EXPRESSION_SCHEMA.MINUS_FUNCTION'   IN TYPEOF(ARG)
THEN
      RETURN (IS_INT_EXPR(ARG\UNARY_NUMERIC_EXPRESSION.OPERAND));
END_IF;
```

```
IF    ( 'EXPRESSION_SCHEMA.PLUS_EXPRESSION'   IN TYPEOF(ARG))
            OR ( 'EXPRESSION_SCHEMA.MULT_EXPRESSION'
                                                IN TYPEOF(ARG))
            OR ( 'EXPRESSION_SCHEMA.MAXIMUM_FUNCTION'
                                                IN TYPEOF(ARG))
            OR ( 'EXPRESSION_SCHEMA.MINIMUM_FUNCTION'
                                                IN TYPEOF(ARG))
THEN
      REPEAT I :=1 TO SIZEOF (
                  ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS);
      IF NOT
            IS_INT_EXPR(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I])
      THEN
            RETURN (FALSE);
      END_IF;
      END_REPEAT;
      RETURN (TRUE);
END_IF;
IF    ( 'EXPRESSION_SCHEMA.MINUS_EXPRESSION'   IN TYPEOF(ARG))
            OR ( 'EXPRESSION_SCHEMA.POWER_EXPRESSION'
                                                IN TYPEOF(ARG))
THEN
      RETURN (IS_INT_EXPR(ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1])
            AND IS_INT_EXPR(ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2]));
END_IF;
IF    ( 'EXPRESSION_SCHEMA.DIV_EXPRESSION'   IN TYPEOF(ARG))
            OR ( 'EXPRESSION_SCHEMA.MOD_EXPRESSION'   IN TYPEOF(ARG))
THEN
      RETURN(TRUE);     (*ALWAYS DELIVER AN INTEGER RESULT *)
END_IF;
IF  'EXPRESSION_SCHEMA.SLASH_EXPRESSION'   IN TYPEOF(ARG)
THEN
      RETURN (FALSE);   (* ALWAYS DELIVERS A REAL RESULT *)
END_IF;
IF  'EXPRESSION_SCHEMA.LENGTH_FUNCTION'   IN TYPEOF(ARG)
THEN
      RETURN (TRUE);
END_IF;
IF  'EXPRESSION_SCHEMA.VALUE_FUNCTION'   IN TYPEOF(ARG)
THEN
      IF  'EXPRESSION_SCHEMA.INT_VALUE_FUNCTION'
                                                IN TYPEOF(ARG)
      THEN
            RETURN (TRUE);
      ELSE
            RETURN (FALSE);
      END_IF;
END_IF;
IF  'EXPRESSION_SCHEMA.INTEGER_DEFINED_FUNCTION'
                                                IN TYPEOF(ARG)
THEN
      RETURN(TRUE) ;
END_IF;
IF 'EXPRESSION_SCHEMA.REAL_DEFINED_FUNCTION'   IN TYPEOF(ARG)
THEN
      RETURN(FALSE) ;
END_IF ;
IF  'EXPRESSION_SCHEMA.BOOLEAN_DEFINED_FUNCTION'
                                                IN TYPEOF(ARG)
THEN
      RETURN(FALSE) ;
```

```
END_IF ;
IF  'EXPRESSION_SCHEMA.STRING_DEFINED_FUNCTION'
                                                IN TYPEOF(ARG)
THEN
      RETURN (FALSE) ;
END_IF ;
(* IF ANOTHER GENERIC_EXPRESSION IS INVOLVED THAT IS NOT A SUBTYPE OF
INTEGER_DEFINED_FUNCTION THEN ITS RESULT IS NOT INTEGER. *)
RETURN (FALSE);

END_FUNCTION; -- IS_INT_EXPR

FUNCTION IS_ACYCLIC (ARG: GENERIC_EXPRESSION): BOOLEAN;
RETURN (ACYCLIC (ARG, []));
END_FUNCTION ; -- IS_ACYCLIC

FUNCTION ACYCLIC (ARG1: GENERIC_EXPRESSION;
                  ARG2: SET OF GENERIC_EXPRESSION): BOOLEAN;

LOCAL
      RESULT: BOOLEAN := TRUE;
END_LOCAL;

IF ('EXPRESSION_SCHEMA.SIMPLE_GENERIC_EXPRESSION'
      IN TYPEOF (ARG1))
THEN
      RETURN (TRUE);
END_IF;

IF ARG1 IN ARG2
THEN
      RETURN (FALSE);
END_IF;

IF 'EXPRESSION_SCHEMA.UNARY_GENERIC_EXPRESSION'
      IN TYPEOF (ARG1)
THEN
      RETURN
      (ACYCLIC(ARG1\UNARY_GENERIC_EXPRESSION.OPERAND,ARG2+[ARG1]));
END_IF;

IF 'EXPRESSION_SCHEMA.BINARY_GENERIC_EXPRESSION'
      IN TYPEOF (ARG1)
THEN
      RETURN
      (ACYCLIC(ARG1\BINARY_GENERIC_EXPRESSION.OPERANDS[1],ARG2+[ARG1])
      AND
      ACYCLIC(ARG1\BINARY_GENERIC_EXPRESSION.OPERANDS[2],ARG2+[ARG1]));
END_IF;

IF
'EXPRESSION_SCHEMA.MULTIPLE_ARITY_GENERIC_EXPRESSION'
      IN TYPEOF (ARG1)
THEN
      RESULT := TRUE;
      REPEAT I := 1 TO
                  SIZEOF (ARG1\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS);
            RESULT := RESULT AND
            ACYCLIC(ARG1\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS[I],
ARG2+[ARG1]);
      END_REPEAT;
```

```
      RETURN (RESULT);
END_IF;

RETURN (RESULT);
END_FUNCTION; -- ACYCLIC


FUNCTION PLUS_FUNCTION (ARG: NUMERIC_EXPRESSION) : NUMBER;

LOCAL
      I: INTEGER;
      SUM: NUMBER :=0;
END_LOCAL;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE
EVALUATED/OBTAINED
      REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS);
            SUM := SUM +
ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
      END_REPEAT;
      RETURN (SUM);

END_FUNCTION; -- PLUS_FUNCTION

FUNCTION MULT_FUNCTION (ARG: NUMERIC_EXPRESSION) : NUMBER;

LOCAL
      I: INTEGER;
      MULT: NUMBER :=1;
END_LOCAL;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE
EVALUATED/OBTAINED
      REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS);
            MULT := MULT *
ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
      END_REPEAT;
      RETURN (MULT);

END_FUNCTION; -- MULT_FUNCTION

FUNCTION MINUS_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1].THE_VALUE-
ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2].THE_VALUE);

END_FUNCTION; -- MINUS_FCT

FUNCTION DIV_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1].THE_VALUE DIV
ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2].THE_VALUE);

END_FUNCTION; -- DIV_FUNCTION

FUNCTION MOD_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1].THE_VALUE MOD
ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
```

```
END_FUNCTION; -- MOD_FUNCTION

FUNCTION SLASH_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
	--HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
	RETURN (ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1].THE_VALUE /
ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
END_FUNCTION; -- SLASH_FUNCTION

FUNCTION POWER_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
	--HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
	RETURN (ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[1].THE_VALUE **
ARG\BINARY_NUMERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
END_FUNCTION; -- POWER_FUNCTION

FUNCTION MINUS_UNARY_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
	--HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
	RETURN (-ARG\UNARY_NUMERIC_EXPRESSION.OPERAND.THE_VALUE);
END_FUNCTION; -- MINUS_UNARY_FUNCTION

FUNCTION MAXIMUM_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
LOCAL
	I: INTEGER;
	RES: NUMBER := 0;
END_LOCAL;
	--HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
	REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS);
		IF (I=1) THEN RES :=
ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
		END_IF;
		IF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE>RES) THEN RES
:= ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
		END_IF;
	END_REPEAT;
	RETURN (RES);
END_FUNCTION; -- MAXIMUM_FUNCTION

FUNCTION MINIMUM_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
LOCAL
	I: INTEGER;
	RES: NUMBER := 0;
END_LOCAL;
	--HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE
EVALUATED/OBTAINED
	REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS);
		IF (I=1) THEN RES :=
ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
		END_IF;
		IF
(ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE<RES) THEN RES
:= ARG\MULTIPLE_ARITY_NUMERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
		END_IF;
	END_REPEAT;
	RETURN (RES);
END_FUNCTION; -- MINIMUM_FUNCTION

FUNCTION LENGTH_FCT (ARG: LENGTH_FUNCTION) : INTEGER;
LOCAL
	STR: STRING;
```

```
END_LOCAL;
      STR := ARG.OPERAND.THE_VALUE;
      RETURN (LENGTH(STR));
END_FUNCTION; -- LENGTH_FUNCTION

FUNCTION VALUE_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (VALUE(ARG\VALUE_FUNCTION.OPERAND.THE_VALUE));
END_FUNCTION; -- VALUE_FUNCTION

FUNCTION INT_VALUE_FCT (ARG: NUMERIC_EXPRESSION) : INTEGER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (VALUE(ARG\VALUE_FUNCTION.OPERAND.THE_VALUE));
END_FUNCTION; -- INT_VALUE_FUNCTION

FUNCTION ABS_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE EVALUATED/OBTAINED
      RETURN (ABS(ARG\UNARY_NUMERIC_EXPRESSION.OPERAND.THE_VALUE));
END_FUNCTION; -- ABS_FUNCTION


FUNCTION NOT_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN (NOT(ARG\NOT_EXPRESSION.OPERAND.THE_VALUE));
END_FUNCTION; -- NOT_FUNCTION

FUNCTION ODD_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN (ODD(ARG\ODD_FUNCTION.OPERAND.THE_VALUE));
END_FUNCTION; -- ODD_FUNCTION

FUNCTION XOR_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN (ARG\XOR_EXPRESSION.OPERANDS[1].THE_VALUE XOR
ARG\XOR_EXPRESSION.OPERANDS[2].THE_VALUE);
END_FUNCTION; -- XOR_FUNCTION

FUNCTION EQUALS_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE :=:
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
END_FUNCTION; -- EQUALS_FUNCTION


FUNCTION AND_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;

LOCAL
      I: INTEGER;
END_LOCAL;
      REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS);
            IF ('EXPRESSION_SCHEMA.BOOLEAN_EXPRESSION' IN TYPEOF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I])) THEN
                  IF
(NOT(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I].THE_VALUE)) THEN
                        RETURN(FALSE);
                  END_IF;
            ELSE
                  RETURN(FALSE);
            END_IF;
      END_REPEAT;
      RETURN (TRUE);
END_FUNCTION; -- AND_FUNCTION

FUNCTION OR_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
```

```
LOCAL
      I: INTEGER;
END_LOCAL;
      REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS);
            IF ('EXPRESSION_SCHEMA.BOOLEAN_EXPRESSION' IN TYPEOF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I])) THEN
                  IF
(ARG\MULTIPLE_ARITY_BOOLEAN_EXPRESSION.OPERANDS[I].THE_VALUE) THEN
                        RETURN(TRUE);
                  END_IF;
            ELSE
                  RETURN(FALSE);
            END_IF;
      END_REPEAT;
      RETURN (FALSE);
END_FUNCTION; -- OR_FUNCTION

FUNCTION COMPARISON_EQUAL_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE =
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'EQUAL TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'EQUAL FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_EQUAL_FUNCTION

FUNCTION COMPARISON_GREATER_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE >
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'GREATER TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'GREATER FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_GREATER_FUNCTION

FUNCTION COMPARISON_GREATER_EQUAL_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE >=
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'EQUAL GREATER TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'EQUAL GREATER FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_GREATER_EQUAL_FUNCTION
```

```
FUNCTION COMPARISON_LESS_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE <
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'LESS TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'LESS FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_LESS_FUNCTION

FUNCTION COMPARISON_LESS_EQUAL_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE <=
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'LESS EQUAL TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'LESS EQUAL FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_LESS_EQUAL_FUNCTION

FUNCTION COMPARISON_NOT_EQUAL_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
LOCAL
      COMPARE: BOOLEAN;
END_LOCAL;
      COMPARE := (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE <>
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
      IF (COMPARE) THEN
            TRACE_FUNCTION(ARG,'NOT EQUAL TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'NOT EQUAL FALSE');
      END_IF;
      RETURN(COMPARE);
END_FUNCTION; -- COMPARISON_NOT_EQUAL_FUNCTION

FUNCTION LIKE_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN (ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[1].THE_VALUE LIKE
ARG\BINARY_GENERIC_EXPRESSION.OPERANDS[2].THE_VALUE);
END_FUNCTION; -- LIKE_FUNCTION

FUNCTION INTERVAL_FCT (ARG: BOOLEAN_EXPRESSION) : BOOLEAN;
      RETURN ((ARG\INTERVAL_EXPRESSION.INTERVAL_LOW.THE_VALUE <=
ARG\INTERVAL_EXPRESSION.INTERVAL_ITEM.THE_VALUE) AND
(ARG\INTERVAL_EXPRESSION.INTERVAL_ITEM.THE_VALUE <=
ARG\INTERVAL_EXPRESSION.INTERVAL_HIGH.THE_VALUE));
END_FUNCTION; -- INTERVAL_FUNCTION

FUNCTION INDEX_FCT (ARG: STRING_EXPRESSION) : STRING;
      RETURN
(ARG\INDEX_EXPRESSION.OPERAND.THE_VALUE[ARG\INDEX_EXPRESSION.INDEX.THE_VALU
E]);
END_FUNCTION; -- INDEX_FUNCTION

FUNCTION SUBSTRING_FCT (ARG: STRING_EXPRESSION) : STRING;
```

```
      RETURN
(ARG\SUBSTRING_EXPRESSION.OPERAND.THE_VALUE[ARG\SUBSTRING_EXPRESSION.INDEX1
.THE_VALUE:ARG\SUBSTRING_EXPRESSION.INDEX2.THE_VALUE]);
END_FUNCTION; -- SUBSTRING_FUNCTION

FUNCTION CONCAT_FCT (ARG: STRING_EXPRESSION) : STRING;

LOCAL
      I: INTEGER;
      STR_CONCAT: STRING :='';
END_LOCAL;
      REPEAT I :=1 TO SIZEOF
(ARG\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS);
             STR_CONCAT := STR_CONCAT +
ARG\MULTIPLE_ARITY_GENERIC_EXPRESSION.OPERANDS[I].THE_VALUE;
      END_REPEAT;
      RETURN (STR_CONCAT);
END_FUNCTION; -- CONCAT_FUNCTION

FUNCTION FORMAT_FCT (ARG: STRING_EXPRESSION) : STRING;
      RETURN
(FORMAT(ARG\FORMAT_FUNCTION.VALUE_TO_FORMAT.THE_VALUE,ARG\FORMAT_FUNCTION.F
ORMAT_STRING.THE_VALUE));
END_FUNCTION; -- SUBSTRING_FUNCTION

FUNCTION SQUARE_FCT (ARG: NUMERIC_EXPRESSION) : NUMBER;
      --HYP: THE OPERANDS ARE CORRECT NUMBERS THAT CAN BE
EVALUATED/OBTAINED
      RETURN (SQRT(ARG\UNARY_NUMERIC_EXPRESSION.OPERAND.THE_VALUE));
END_FUNCTION; -- SQUARE_FUNCTION

(**********************************************************************)

--THIS ENTITY IS THE MAIN ENTRY TO REPRESENT CONSTRAINTS OF INTER MODEL
--RELATIONS (THIS IS AN EXAMPLE SINCE IT IS A GENERATED ENTITY IN OUR
--FRAMEWORK)
--ENTITY INTER_MODEL_CONSTRAINT
--SUBTYPE OF (BOOLEAN_EXPRESSION);
      --NAME : STRING;
      --CONSTRAINED_RELATION: OPTIONAL RELATION;
      --INITIAL_CONTEXT: SET OF GENERIC_VARIABLE;
      --PROPERTIES: SET OF EXPRESSION;
      --PROPERTY1: EXPRESSION;
      --PROPERTY2: EXPRESSION;
--DERIVE
      --SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=
EVALUATE_INTER_MODEL_CONSTRAINTS(SELF);
      --THE_VALUE1: BOOLEAN := SELF.PROPERTY1.THE_VALUE;
      --THE_VALUE2: BOOLEAN := PROPERTY2.THE_VALUE;
      --IT RETURNS TRUE IF ALL PROPERTIES ARE RESPECTED, FALSE THE FIRST
TIME ONE PROPERTY EVALUTES TO FALSE
--WHERE
     --PROPERTIES : THE_VALUE = TRUE ;
     --PROPERTY1 : THE_VALUE1 = TRUE ;
     --PROPERTY2 : THE_VALUE2 = TRUE ;
--END_ENTITY;


(**********************************************)
(*** VARIABLES *******)
```

```
(*********************************************)

--IT REPRESENTS A SET OF VALUES TO BE TREATED IN AN EXPRESSION
ENTITY EXPRESSION_DOMAIN
ABSTRACT;
END_ENTITY;

--UNDER THIS CLASS WE FIND THE PRIMITIVE TYPES
ENTITY PRIMITIVE_DOMAIN
ABSTRACT
SUBTYPE OF (EXPRESSION_DOMAIN);
END_ENTITY;

--IT CONTAINS STRINGS
ENTITY STRING_DOMAIN
SUBTYPE OF (PRIMITIVE_DOMAIN);
      THE_VALUE: SET OF STRING;
END_ENTITY;

--IT CONTAINS NUMERICS
ENTITY NUMERIC_DOMAIN
SUBTYPE OF (PRIMITIVE_DOMAIN);
      THE_VALUE: SET OF NUMBER;
END_ENTITY;

--IT CONTAINS BOOLEANS
ENTITY BOOLEAN_DOMAIN
SUBTYPE OF (PRIMITIVE_DOMAIN);
      THE_VALUE: SET OF BOOLEAN;
END_ENTITY;

--UNDER THIS CLASS WE FIND THE OBJECT TYPES
ENTITY OBJECT_DOMAIN
ABSTRACT
SUBTYPE OF (EXPRESSION_DOMAIN);
END_ENTITY;

--IT CONTAINS INSTANCES OF ENTITY_CLASS
ENTITY ENTITY_DOMAIN
SUBTYPE OF (OBJECT_DOMAIN);
      THE_VALUE: SET OF ENTITY_CLASS;
END_ENTITY;

--IT PUTS IN RELATION A VARIABLE WITH ITS DOMAIN
ENTITY VARIABLE_DOMAIN;
      THE_DOMAIN: EXPRESSION_DOMAIN;
      THE_VARIABLE: GENERIC_VARIABLE;
END_ENTITY;

--ONE EXPRESSION WHICH RETURNS AN ENTITY_CLASS
ENTITY ENTITY_EXPRESSION
ABSTRACT
SUBTYPE OF (EXPRESSION);
      THE_VALUE: OPTIONAL ENTITY_CLASS;
END_ENTITY;

--ONE EXPRESSION WHICH RETURNS AN ARRAY OF SOMETHING
ENTITY ARRAY_EXPRESSION
ABSTRACT
SUBTYPE OF (GENERIC_EXPRESSION);
      THE_VALUE: OPTIONAL SET OF T_DOMAINE;
```

```
END_ENTITY;
```

--ONE EXPRESSION WHICH RETURNS AN ARRAY OF STRING

```
ENTITY STRING_ARRAY_EXPRESSION
ABSTRACT
SUBTYPE OF (ARRAY_EXPRESSION);
      SELF\ARRAY_EXPRESSION.THE_VALUE: OPTIONAL SET OF STRING;
END_ENTITY;
```

--ONE EXPRESSION WHICH RETURNS AN ARRAY OF NUMERIC

```
ENTITY NUMERIC_ARRAY_EXPRESSION
ABSTRACT
SUBTYPE OF (ARRAY_EXPRESSION);
      SELF\ARRAY_EXPRESSION.THE_VALUE: OPTIONAL SET OF NUMBER;
END_ENTITY;
```

--ONE EXPRESSION WHICH RETURNS AN ARRAY OF BOOLEAN

```
ENTITY BOOLEAN_ARRAY_EXPRESSION
ABSTRACT
SUBTYPE OF (ARRAY_EXPRESSION);
      SELF\ARRAY_EXPRESSION.THE_VALUE: OPTIONAL SET OF BOOLEAN;
END_ENTITY;
```

--ONE EXPRESSION WHICH RETURNS AN ARRAY OF ENTITY_CLASS

```
ENTITY ENTITY_ARRAY_EXPRESSION
ABSTRACT
SUBTYPE OF (ARRAY_EXPRESSION);
      SELF\ARRAY_EXPRESSION.THE_VALUE: OPTIONAL SET OF ENTITY_CLASS;
END_ENTITY;
```

--ONE SIMPLE EXPRESSION WHICH RETURNS AN ENTITY_CLASS

```
ENTITY SIMPLE_ENTITY_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (ENTITY_LITERAL, ENTITY_VARIABLE))
SUBTYPE OF (ENTITY_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;
```

--LITERAL REPRESENTING AN ENTITY_CLASS

```
ENTITY ENTITY_LITERAL
SUBTYPE OF (SIMPLE_ENTITY_EXPRESSION, GENERIC_LITERAL);
      SELF\ENTITY_EXPRESSION.THE_VALUE: ENTITY_CLASS;
END_ENTITY;
```

--IT REPRESENTS A VARIABLE CONTAINING AN INSTANCE OF AN OBJECT

```
ENTITY OBJECT_VARIABLE
ABSTRACT
SUBTYPE OF (VARIABLE);
END_ENTITY;
```

--VARIABLE POINTING TO AN ENTITY_CLASS

```
ENTITY ENTITY_VARIABLE
SUBTYPE OF (SIMPLE_ENTITY_EXPRESSION, OBJECT_VARIABLE);
END_ENTITY;
```

--ONE SIMPLE EXPRESSION WHICH RETURNS A SET OF STRING

```
ENTITY SIMPLE_STRING_ARRAY_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (STRING_ARRAY_LITERAL, STRING_ARRAY_VARIABLE))
SUBTYPE OF (STRING_ARRAY_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;
```

--LITERAL REPRESENTING A SET OF STRING

```
ENTITY STRING_ARRAY_LITERAL
SUBTYPE OF (SIMPLE_STRING_ARRAY_EXPRESSION, GENERIC_LITERAL);
END_ENTITY;

--VARIABLE POINTING TO A SET OF STRING
ENTITY STRING_ARRAY_VARIABLE
SUBTYPE OF (SIMPLE_STRING_ARRAY_EXPRESSION, VARIABLE);
END_ENTITY;

--ONE SIMPLE EXPRESSION WHICH RETURNS A SET OF NUMERIC
ENTITY SIMPLE_NUMERIC_ARRAY_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (NUMERIC_ARRAY_LITERAL,
NUMERIC_ARRAY_VARIABLE))
SUBTYPE OF (NUMERIC_ARRAY_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

--LITERAL REPRESENTING A SET OF NUMERIC
ENTITY NUMERIC_ARRAY_LITERAL
SUBTYPE OF (SIMPLE_NUMERIC_ARRAY_EXPRESSION, GENERIC_LITERAL);
END_ENTITY;

--VARIABLE POINTING TO A SET OF NUMERIC
ENTITY NUMERIC_ARRAY_VARIABLE
SUBTYPE OF (SIMPLE_NUMERIC_ARRAY_EXPRESSION, VARIABLE);
END_ENTITY;

--ONE SIMPLE EXPRESSION WHICH RETURNS A SET OF BOOLEAN
ENTITY SIMPLE_BOOLEAN_ARRAY_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (BOOLEAN_ARRAY_LITERAL,
BOOLEAN_ARRAY_VARIABLE))
SUBTYPE OF (BOOLEAN_ARRAY_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

--LITERAL REPRESENTING A SET OF BOOLEAN
ENTITY BOOLEAN_ARRAY_LITERAL
SUBTYPE OF (SIMPLE_BOOLEAN_ARRAY_EXPRESSION, GENERIC_LITERAL);
END_ENTITY;

--VARIABLE POINTING TO A SET OF BOOLEAN
ENTITY BOOLEAN_ARRAY_VARIABLE
SUBTYPE OF (SIMPLE_BOOLEAN_ARRAY_EXPRESSION, VARIABLE);
END_ENTITY;

--ONE SIMPLE EXPRESSION WHICH RETURNS A SET OF ENTITY_CLASS
ENTITY SIMPLE_ENTITY_ARRAY_EXPRESSION
ABSTRACT SUPERTYPE OF (ONEOF (ENTITY_ARRAY_LITERAL, ENTITY_ARRAY_VARIABLE))
SUBTYPE OF (ENTITY_ARRAY_EXPRESSION, SIMPLE_GENERIC_EXPRESSION);
END_ENTITY;

--LITERAL REPRESENTING  A SET OF ENTITY
ENTITY ENTITY_ARRAY_LITERAL
SUBTYPE OF (SIMPLE_ENTITY_ARRAY_EXPRESSION);
END_ENTITY;

--VARIABLE POINTING TO A SET OF ENTITY
ENTITY ENTITY_ARRAY_VARIABLE
SUBTYPE OF (SIMPLE_ENTITY_ARRAY_EXPRESSION, VARIABLE);
END_ENTITY;

--THIS ENTITY REPRESENTS THE ACCESS TO ONE ENTITY OR ATTRIBUTE OF AN ENTITY
ENTITY PATH_VARIABLE
```

```
ABSTRACT
SUBTYPE OF (GENERIC_VARIABLE);
      SOURCE_VARIABLE: ENTITY_VARIABLE; --VARIABLE POINTING TO THE ENTITY
FROM WHICH WE WANT TO GET THE ATTRIBUTE/ANNOTATION
      ATTRIBUTE_NAME: OPTIONAL STRING; --ATTRIBUTE OF THE ENTITY_NAME OR
NAME OF THE ANNOTATION IF IS_ANNOTATION=TRUE
      IS_ANNOTATION: BOOLEAN; --TRUE IF ATTRIBUTE NAME CONTAINS THE NAME OF
ONE ANNOTATION
END_ENTITY;

--PATH VARIABLE POINTING TO AN ENTITY_CLASS
ENTITY ENTITY_PATH_VARIABLE
SUBTYPE OF (ENTITY_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A STRING
ENTITY STRING_PATH_VARIABLE
SUBTYPE OF (STRING_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A NUMERIC
ENTITY NUMERIC_PATH_VARIABLE
SUBTYPE OF (NUMERIC_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A BOOLEAN
ENTITY BOOLEAN_PATH_VARIABLE
SUBTYPE OF (BOOLEAN_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO AN ENTITY_CLASS
ENTITY ENTITY_ARRAY_PATH_VARIABLE
SUBTYPE OF (ENTITY_ARRAY_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A STRING
ENTITY STRING_ARRAY_PATH_VARIABLE
SUBTYPE OF (STRING_ARRAY_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A NUMERIC
ENTITY NUMERIC_ARRAY_PATH_VARIABLE
SUBTYPE OF (NUMERIC_ARRAY_VARIABLE, PATH_VARIABLE);
END_ENTITY;

--PATH VARIABLE POINTING TO A BOOLEAN
ENTITY BOOLEAN_ARRAY_PATH_VARIABLE
SUBTYPE OF (BOOLEAN_ARRAY_VARIABLE, PATH_VARIABLE);
END_ENTITY;

(*** LOGICAL EXPRESSIONS ***)

--THIS ENTITY REPRESENTS THE FOL (FIRST ORDER LOGIC) EXPRESSIONS
ENTITY FOL_EXPRESSION
SUBTYPE OF (BOOLEAN_EXPRESSION);
      CONTEXT_VARIABLES: OPTIONAL SET OF GENERIC_VARIABLE; --THE CONTEXT OF
EVALUATION (BRACKETED VARIABLES)
      EXPRESSION_VARIABLES: SET OF VARIABLE_DOMAIN; --THE VARIABLES
DIRECTED LINKED TO THE EXPRESSION AND THEIR DOMAINS (EVALUATED ELEMENTS)
      PREDICATE: BOOLEAN_EXPRESSION;--THE PREDICATE TO BE EVALUATED
END_ENTITY;
```

```
--THIS ENTITY REPRESENTS THE EXISTS FOL ASSERTION
ENTITY EXISTS_EXPRESSION
SUBTYPE OF (FOL_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=EXISTS_FUNCTION(SELF);
END_ENTITY;

--THIS ENTITY REPRESENTS THE ALL FOL ASSERTION
ENTITY ALL_EXPRESSION
SUBTYPE OF (FOL_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE: BOOLEAN :=ALL_FUNCTION(SELF);
END_ENTITY;

--THIS ENTITY REPRESENTS THE ALL SUM FOL ASSERTION ?
ENTITY ALL_SUM_EXPRESSION
SUBTYPE OF (NUMERIC_EXPRESSION);
      CONTEXT_VARIABLES: OPTIONAL SET OF GENERIC_VARIABLE; --THE CONTEXT OF
EVALUATION (BRACKETED VARIABLES)
      EXPRESSION_VARIABLES: SET OF VARIABLE_DOMAIN; --THE VARIABLES
DIRECTED LINKED TO THE EXPRESSION AND THEIR DOMAINS (EVALUATED ELEMENTS)
      PREDICATE: BOOLEAN_EXPRESSION;--THE PREDICATE TO BE EVALUATED
DERIVE
      SELF\NUMERIC_EXPRESSION.THE_VALUE: INTEGER :=ALL_SUM_FUNCTION(SELF);
END_ENTITY;

--THIS ENTITY REPRESENTS THE BELONGING TO A SET
--THE FIRST OPERAND IS THE VALUE TO SEARCH IN THE SECOND OPERAND (A SET)
ENTITY BELONG_BOOLEAN_EXPRESSION
SUBTYPE OF (BINARY_BOOLEAN_EXPRESSION);
DERIVE
      SELF\BOOLEAN_EXPRESSION.THE_VALUE:BOOLEAN := BELONG_FUNCTION(SELF);
END_ENTITY;

--THIS ENTITY ALLOWS COMPARING 2 VALUES WHICH MUST BE EQUAL
ENTITY COMPARISON_EQUAL_CONTEXT_EXPRESSION
SUBTYPE OF (COMPARISON_EQUAL);
DERIVE
      SELF\COMPARISON_EQUAL.THE_VALUE: BOOLEAN :=EQUAL_FUNCTION(SELF);
END_ENTITY;

(**** DEPRECATED: REPLACE WITH (NOT(EXPRESSION1) OR (EXPRESSION2)) ****)
--THIS ENTITY REPRESENTS THE IMPLICATION FOL ASSERTION
ENTITY IMPLICATION_EXPRESSION
SUBTYPE OF (FOL_EXPRESSION);
END_ENTITY;

ENTITY TRACE;
      INSTANCE_ID: STRING;
      DESCRIPTION: OPTIONAL STRING;
END_ENTITY;

--THIS ENTITY STORES THE EXECUTION OF A FOL EXPRESSION
ENTITY FOL_TRACE;
      ID: OPTIONAL SET OF INTEGER;
      DESC: OPTIONAL SET OF STRING;
END_ENTITY;

(*************************************)
```

(*** FOL FUNCTIONS ***)
(************************************)

```
FUNCTION TRACE_FUNCTION(INST:GENERIC; DESCRIPTION:STRING): STRING;
LOCAL
        CURRENT_TRACE: STRING;
        ALL_TRACES : SET OF FOL_TRACE;
        CURRENT_FOL_TRACE : FOL_TRACE;
        LAST_TRACE : INTEGER;
        einfo: entity_info;
        ID: INTEGER;
        NUMBER_OF_TRACES : INTEGER;
END_LOCAL;
      einfo := get_type_info(INST);
      ID :=       einfo.id;
      CURRENT_TRACE := FORMAT(id,'2I') +':'+description+';';
      ALL_TRACES := POPULATION('EXPRESSION_SCHEMA.FOL_TRACE');
      LAST_TRACE := SIZEOF(ALL_TRACES);
      IF (LAST_TRACE > 0) THEN
            CURRENT_FOL_TRACE := ALL_TRACES[LAST_TRACE];
            --Treatment of first VALUE (null)
            CURRENT_FOL_TRACE.id := NVL(CURRENT_FOL_TRACE.id,[]);
            CURRENT_FOL_TRACE.desc := NVL(CURRENT_FOL_TRACE.desc,[]);
            CURRENT_FOL_TRACE.id := CURRENT_FOL_TRACE.id + ID;
            CURRENT_FOL_TRACE.desc := CURRENT_FOL_TRACE.desc + description;
      END_IF;

RETURN (CURRENT_TRACE);
END_FUNCTION;
```

--THIS FUNCTION EVALUATES ALL AND EXISTS EXPRESSIONS DEPENDING ON THE VALUE OF WHEN_STOP PARAMETER

```
FUNCTION QUANTIFIER_FUNCTION(ARG:FOL_EXPRESSION; WHEN_STOP:BOOLEAN):
BOOLEAN;
LOCAL
        SIZE : INTEGER := 0;
        I : INTEGER := 1;
        BOOL_EXP : BOOLEAN_EXPRESSION;
        VAR_DOM: VARIABLE_DOMAIN;
        DOM: EXPRESSION_DOMAIN;
        VARI: GENERIC_VARIABLE; --EXPRESSION; --VARIABLE
        VALEUR : T_DOMAINE;
        LIST_VALEURS: SET OF T_DOMAINE;
        BOOLEAN_VALUE: BOOLEAN;
END_LOCAL;
      --NUMBER OF VALUES TO CHECK
      VAR_DOM := ARG.EXPRESSION_VARIABLES[1];
      SIZE := SIZEOF (VAR_DOM.THE_DOMAIN.THE_VALUE);
      --EXPRESSION TO CHECK, IT SHOULD ACCESS TO THE VARIABLE
      BOOL_EXP := ARG.PREDICATE;
      --WE ASSIGN EACH VALUE TO THE VARIABLE
      REPEAT I:= 1 TO SIZE;
            --WE ASSIGN THE CURRENT CONCEPT OF THE DOMAIN TO THE RELATED
VARIABLE (THEY HAVE TO BE TYPE CONSISTENT)
            DOM := VAR_DOM.THE_DOMAIN;
            VARI := VAR_DOM.THE_VARIABLE;
            LIST_VALEURS := DOM.THE_VALUE;
            VALEUR := LIST_VALEURS[I];
            VARI.THE_VALUE := VALEUR;
            --THE BOOLEAN_EXPRESSION RELATED TO THE EXISTS MUST BE TRUE FOR AT
LEAST ONE CONCEPT OF DOMAIN
```

```
            BOOLEAN_VALUE := BOOL_EXP.THE_VALUE;
            IF (BOOLEAN_VALUE = WHEN_STOP) THEN
                  RETURN(WHEN_STOP);
            END_IF;
      END_REPEAT;
RETURN (NOT(WHEN_STOP));
END_FUNCTION;

--IMPLEMENTATION OF EXISTS EXPRESSION
FUNCTION EXISTS_FUNCTION(ARG:EXISTS_EXPRESSION): BOOLEAN;
LOCAL
        BOOLEAN_VALUE: BOOLEAN;
END_LOCAL;
      --WE FINISH WHEN 1 VARIABLE FULFILLS THE PREDICATE (SECOND PARAMETER
TRUE)
      BOOLEAN_VALUE := QUANTIFIER_FUNCTION(ARG,TRUE);
      IF (BOOLEAN_VALUE) THEN
            TRACE_FUNCTION(ARG,'EXISTS TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'EXISTS FALSE');
      END_IF;
      RETURN(BOOLEAN_VALUE);
END_FUNCTION;

--IMPLEMENTATION OF ALL EXPRESSION
FUNCTION ALL_FUNCTION(ARG:ALL_EXPRESSION): BOOLEAN;
LOCAL
        BOOLEAN_VALUE: BOOLEAN;
END_LOCAL;
      BOOLEAN_VALUE := QUANTIFIER_FUNCTION(ARG,FALSE);
      IF (BOOLEAN_VALUE) THEN
            TRACE_FUNCTION(ARG,'ALL TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'ALL FALSE');
      END_IF;
      --WE FINISH WHEN ALL VARIABLES FULFILLS THE PREDICATE (SECOND PARAMETER
FALSE)
      RETURN(BOOLEAN_VALUE);
END_FUNCTION;

--IMPLEMENTATION OF ALL EXPRESSION
FUNCTION ALL_SUM_FUNCTION(ARG:ALL_SUM_EXPRESSION): INTEGER;
LOCAL
        SIZE : INTEGER := 0;
        I : INTEGER := 1;
        BOOL_EXP : BOOLEAN_EXPRESSION;
        VAR_DOM: VARIABLE_DOMAIN;
        DOM: EXPRESSION_DOMAIN;
        VARI: GENERIC_VARIABLE; --EXPRESSION; --VARIABLE
        VALEUR : T_DOMAINE;
        LIST_VALEURS: SET OF T_DOMAINE;
        SUM_OK : INTEGER := 0;
END_LOCAL;

      --NUMBER OF VALUES TO CHECK
      VAR_DOM := ARG.EXPRESSION_VARIABLES[1];
      SIZE := SIZEOF (VAR_DOM.THE_DOMAIN.THE_VALUE);
      --EXPRESSION TO CHECK, IT SHOULD ACCESS TO THE VARIABLE
      BOOL_EXP := ARG.PREDICATE;
      --WE ASSIGN EACH VALUE TO THE VARIABLE
      REPEAT I:= 1 TO SIZE;
```

```
              --WE ASSIGN THE CURRENT CONCEPT OF THE DOMAIN TO THE RELATED
VARIABLE (THEY HAVE TO BE TYPE CONSISTENT)
              DOM := VAR_DOM.THE_DOMAIN;
              VARI := VAR_DOM.THE_VARIABLE;
              LIST_VALEURS := DOM.THE_VALUE;
              VALEUR := LIST_VALEURS[I];
              VARI.THE_VALUE := VALEUR;
              --THE BOOLEAN_EXPRESSION RELATED TO THE EXISTS MUST BE TRUE FOR AT
LEAST ONE CONCEPT OF DOMAIN
              IF (BOOL_EXP.THE_VALUE = TRUE) THEN
              SUM_OK := SUM_OK + 1;
              END_IF;
       END_REPEAT;
RETURN (SUM_OK);
END_FUNCTION;

FUNCTION FIND_ATTRIBUTE_VALUE(ARG:GENERIC;ATT_NAME:STRING): T_DOMAINE;
LOCAL
  einfo: entity_info;
  atts: LIST OF attribute;
  SIZE_ATTRIBUTES : INTEGER := 0;
END_LOCAL;
       einfo := get_type_info(ARG);
       --FIRST WE LOOK FOR THE EXPLICIT ATTRIBUTES
       atts := einfo.explicit;
       SIZE_ATTRIBUTES := SIZEOF (atts);
       --WE LOOK FOR THE ATTRIBUTE
       REPEAT I:= 1 TO SIZE_ATTRIBUTES;
              IF (atts[I].name = ATT_NAME) THEN
                     RETURN(atts[I].access(ARG));
              END_IF;
       END_REPEAT;

       --NEXT WE LOOK FOR THE DERIVED ATTRIBUTES
       atts := einfo.derived;
       SIZE_ATTRIBUTES := SIZEOF (atts);
       --WE LOOK FOR THE ATTRIBUTE
       REPEAT I:= 1 TO SIZE_ATTRIBUTES;
              IF (atts[I].name = ATT_NAME) THEN
                     RETURN(atts[I].access(ARG));
              END_IF;
       END_REPEAT;

       --FINALLY WE LOOK FOR THE INVERSE ATTRIBUTES
       atts := einfo.inv;
       SIZE_ATTRIBUTES := SIZEOF (atts);
       --WE LOOK FOR THE ATTRIBUTE
       REPEAT I:= 1 TO SIZE_ATTRIBUTES;
              IF (atts[I].name = ATT_NAME) THEN
                     RETURN(atts[I].access(ARG));
              END_IF;
       END_REPEAT;

RETURN ('ERROR: ATTRIBUTE NOT FOUND');
END_FUNCTION;

--THIS FUNCTION GET A VALUE FROM A VARIABLE OR FROM AN EXPRESSION
FUNCTION GET_INDIVIDUAL_VALUE(ARG:GENERIC_EXPRESSION): T_DOMAINE;
LOCAL
       ENTITY_OF_ATTRIBUTES: ENTITY_CLASS;
         SIZE_ATTRIBUTES : INTEGER := 0;
```

```
        SIZE_ANNOTATIONS : INTEGER := 0;
        ALL_ANNOTATIONS: SET OF ANNOTATION_CLASS;
        FILTERED_ANNOTATIONS: SET OF ANNOTATION_CLASS;

END_LOCAL;
      --FIRSTLY WE TREAT THE ATTRIBUTES OF AN ENTITY
      IF ('EXPRESSION_SCHEMA.PATH_VARIABLE' IN TYPEOF (ARG)) THEN
            ENTITY_OF_ATTRIBUTES := ARG.SOURCE_VARIABLE.THE_VALUE;
            IF (ARG.IS_ANNOTATION) THEN --CASE ANNOTATION
                  ALL_ANNOTATIONS :=
POPULATION('ANNOTATION_SCHEMA.ANNOTATION_CLASS');
                  FILTERED_ANNOTATIONS := QUERY(x <* ALL_ANNOTATIONS |
VALUE_IN(x.MY_ENTITIES,ENTITY_OF_ATTRIBUTES) AND (x.NAME =
ARG.ATTRIBUTE_NAME));
                  IF (SIZEOF(FILTERED_ANNOTATIONS)>0) THEN

      RETURN(FILTERED_ANNOTATIONS[1].MY_KNOWLEDGE[1].THE_CLASS);
                  ELSE
                        RETURN('NOT ANNOTATION');
                  END_IF;
            END_IF;
            --CASE IS NOT AN ANNOTATION

      RETURN(FIND_ATTRIBUTE_VALUE(ENTITY_OF_ATTRIBUTES,ARG.ATTRIBUTE_NAME))
;
      END_IF;
      --IF NOT, WE TRY WITH ANOTHER KIND OF VARIABLE
      IF ('EXPRESSION_SCHEMA.VARIABLE' IN TYPEOF (ARG)) THEN
            RETURN (ARG.THE_VALUE);
      END_IF;
      --IF NOT, WE GET DIRECTLY THE VALUE
      RETURN (ARG.THE_VALUE);
END_FUNCTION;


FUNCTION FIND_ATTRIBUTE_VALUES(ARG:GENERIC;ATT_NAME:STRING): SET OF
T_DOMAINE;
LOCAL
  einfo: entity_info;
  atts: LIST OF attribute;
  SIZE_ATTRIBUTES : INTEGER := 0;
END_LOCAL;
      einfo := get_type_info(ARG);
      --FIRST WE LOOK FOR THE EXPLICIT ATTRIBUTES
      atts := einfo.explicit;
      SIZE_ATTRIBUTES := SIZEOF (atts);
      --WE LOOK FOR THE ATTRIBUTE
      REPEAT I:= 1 TO SIZE_ATTRIBUTES;
            IF (atts[I].name = ATT_NAME) THEN
                  RETURN(atts[I].access(ARG));
            END_IF;
      END_REPEAT;

      --NEXT WE LOOK FOR THE DERIVED ATTRIBUTES
      atts := einfo.derived;
      SIZE_ATTRIBUTES := SIZEOF (atts);
      --WE LOOK FOR THE ATTRIBUTE
      REPEAT I:= 1 TO SIZE_ATTRIBUTES;
            IF (atts[I].name = ATT_NAME) THEN
                  RETURN(atts[I].access(ARG));
            END_IF;
```

```
	END_REPEAT;

	--FINALLY WE LOOK FOR THE INVERSE ATTRIBUTES
	atts := einfo.inv;
	SIZE_ATTRIBUTES := SIZEOF (atts);
	--WE LOOK FOR THE ATTRIBUTE
	REPEAT I:= 1 TO SIZE_ATTRIBUTES;
		IF (atts[I].name = ATT_NAME) THEN
			RETURN(atts[I].access(ARG));
		END_IF;
	END_REPEAT;

RETURN ([]);
END_FUNCTION;

--THIS FUNCTION GET A SET OF VALUES FROM A VARIABLE OR FROM AN EXPRESSION
FUNCTION GET_SET_OF_VALUES(ARG:GENERIC_EXPRESSION): SET OF T_DOMAINE;
LOCAL
	ENTITY_OF_ATTRIBUTES: ENTITY_CLASS;
	  SIZE_ATTRIBUTES : INTEGER := 0;
	  SIZE_ANNOTATIONS : INTEGER := 0;
	ARG_ARRAY : GENERIC_EXPRESSION; --ARRAY_EXPRESSION;
	  ALL_ANNOTATIONS: SET OF ANNOTATION_CLASS;
	  FILTERED_ANNOTATIONS: SET OF ANNOTATION_CLASS;

END_LOCAL;
	--FIRSTLY WE TREAT THE ATTRIBUTES OF AN ENTITY
	IF (('EXPRESSION_SCHEMA.ENTITY_ARRAY_PATH_VARIABLE' IN TYPEOF (ARG))
OR ('EXPRESSION_SCHEMA.STRING_ARRAY_PATH_VARIABLE' IN TYPEOF (ARG)) OR
('EXPRESSION_SCHEMA.NUMERIC_ARRAY_PATH_VARIABLE' IN TYPEOF (ARG)) OR
('EXPRESSION_SCHEMA.BOOLEAN_ARRAY_PATH_VARIABLE' IN TYPEOF (ARG))) THEN
		ENTITY_OF_ATTRIBUTES := ARG.SOURCE_VARIABLE.THE_VALUE;
		IF (ARG.IS_ANNOTATION) THEN --CASE ANNOTATION
			ALL_ANNOTATIONS :=
POPULATION('ANNOTATION_SCHEMA.ANNOTATION_CLASS');
			FILTERED_ANNOTATIONS := QUERY(x <* ALL_ANNOTATIONS |
VALUE_IN(x.MY_ENTITIES,ENTITY_OF_ATTRIBUTES) AND (x.NAME =
ARG.ATTRIBUTE_NAME));
			IF (SIZEOF(FILTERED_ANNOTATIONS)>0) THEN
				RETURN([]);
RETURN(FILTERED_ANNOTATIONS[1].MY_KNOWLEDGE[1].THE_CLASS.MY_ATTRIBUTES);
			ELSE
				RETURN(['NOT ANNOTATION']);
			END_IF;

		END_IF;
		--CASE IS NOT AN ANNOTATION

	RETURN(FIND_ATTRIBUTE_VALUES(ENTITY_OF_ATTRIBUTES,ARG.ATTRIBUTE_NAME)
);
	END_IF;
	--IF NOT, WE TRY WITH ANOTHER KIND OF VARIABLE
	IF ('EXPRESSION_SCHEMA.ARRAY_EXPRESSION' IN TYPEOF (ARG)) THEN
		ARG_ARRAY := ARG;
		RETURN (ARG_ARRAY.THE_VALUE);
	END_IF;
	--IF NOT, WE GET DIRECTLY THE VALUE
	RETURN (ARG.THE_VALUE);
END_FUNCTION;
```

```
--THIS FUNCTION EVALUATES WHETHER ARG.OPERANDS[1] IS CONTAINED IN
ARG.OPERANDS[2]
FUNCTION BELONG_FUNCTION(ARG:BELONG_BOOLEAN_EXPRESSION): BOOLEAN;
LOCAL
      GENERIC_VALUE_TO_LOOK_FOR: T_DOMAINE;
      LIST_OF_GENERIC: SET OF T_DOMAINE;
      BOOLEAN_VALUE : BOOLEAN;
END_LOCAL;
      GENERIC_VALUE_TO_LOOK_FOR := GET_INDIVIDUAL_VALUE(ARG.OPERANDS[1]);
      --PRE: THE LIST MUST BE OF THE SAME TYPE
      LIST_OF_GENERIC := GET_SET_OF_VALUES(ARG.OPERANDS[2]);
      --WE LOOK FOR THE VALUE
      BOOLEAN_VALUE := VALUE_IN(LIST_OF_GENERIC,GENERIC_VALUE_TO_LOOK_FOR);

      IF (BOOLEAN_VALUE) THEN
            TRACE_FUNCTION(ARG,'BELONG TRUE');
      ELSE
            TRACE_FUNCTION(ARG,'BELONG FALSE');
      END_IF;

      RETURN (BOOLEAN_VALUE);
END_FUNCTION;

--THIS FUNCTION COMPARES 2 VALUES WHICH MUST BE EQUAL
FUNCTION EQUAL_FUNCTION(ARG:COMPARISON_EQUAL_CONTEXT_EXPRESSION): BOOLEAN;
LOCAL
      VALUE1: T_DOMAINE;
      VALUE2: T_DOMAINE;
END_LOCAL;
            VALUE1 := GET_INDIVIDUAL_VALUE(ARG.OPERANDS[1]);
            VALUE2 := GET_INDIVIDUAL_VALUE(ARG.OPERANDS[2]);
            IF (TYPEOF(VALUE1)=TYPEOF(VALUE2)) THEN
                  IF (VALUE1=VALUE2) THEN
                        TRACE_FUNCTION(ARG,'EQUAL TRUE');
                        RETURN (TRUE);
                  ELSE
                        TRACE_FUNCTION(ARG,'EQUAL FALSE');
                        RETURN (FALSE);
                  END_IF;
            ELSE
                  TRACE_FUNCTION(ARG,'EQUAL FALSE');
                  RETURN (FALSE);
            END_IF;
END_FUNCTION;

END_SCHEMA;
```

Annex B

This annex introduces the CORE and SysML meta-models formalized in EXPRESS modeling language and used in our case studies.

CORE Meta-Model

```
--THIS SCHEMA CONTAINS THE ENTITIES DESCRIBING THE CORE METAMODEL
SCHEMA CORE_SCHEMA;

REFERENCE FROM TOP_SCHEMA;

(*************************************************************************)
(*********************** CORE METAMODEL ***************************)
(*************************************************************************)

--THIS TYPE REPRESENTS DIFFERENT TYPES OF DURATION
TYPE CALCULATION_KIND = SELECT (CONSTANT_CORE,RANDOM_CORE, SCRIPT_CORE);
END_TYPE;

--THIS TYPE REPRESENTS DIFFERENT TYPES OF AMOUNTS
TYPE AMOUNT_KIND = ENUMERATION OF (FLOATS,INTEGERS);
END_TYPE;

--TYPES AND ENTITIES OF CALCULATION
TYPE CONSTANT_CORE = REAL; END_TYPE;
TYPE SCRIPT_CORE = STRING; END_TYPE;
TYPE DISTRIBUTION_KIND = ENUMERATION OF
(BERNOULLI,BETA,BINOMIAL,CHISQUARED,DISCRETEUNFIROM,ERLANG,EXPONENTIAL,F,GA
MMA,GEOMETRIC,LAPLACE,LOGNORMAL,NEGATIVEBINOMIAL,NORMAL,POISSON,T,TRIANGULA
R,UNIFORMM,WEIBULL); END_TYPE;

(**********************************************************)
(**********************************************************)


ENTITY RANDOM_CORE;
      DISTRIBUTION : DISTRIBUTION_KIND;
      MEAN: REAL;
      STANDARD_DEVIATION: REAL;
      RANDOM_NUMBER_STREAM: INTEGER; -- >0
      RESULT: REAL;
END_ENTITY;

--THIS ENTITY REPRESENTS CORE MODELING LANGUAGE
ENTITY CORE_MODELING_LANGUAGE
      SUBTYPE OF (MODELING_LANGUAGE);
      VERSION: OPTIONAL STRING;
DERIVE
      NAME:STRING := 'CORE';
END_ENTITY;

--THIS ENTITY REPRESENTS ONE PARTIAL CORE MODEL
```

```
ENTITY MODEL_CORE
      SUBTYPE OF (MODEL);
      ITS_ENTITIES: OPTIONAL SET[0:?] OF ENTITY_CORE;
      ITS_FLOWS: OPTIONAL SET[0:?] OF FLOW_CORE;
      SELF\MODEL.MODELING_LANGUAGE:CORE_MODELING_LANGUAGE;
END_ENTITY;

--THIS ENTITY REPRESENTS THE COMMON ATTRIBUTS OF CORE ENTITIES
ENTITY ENTITY_CORE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CLASS);
      CREATION_STAMP: T_DATE;
      CREATOR: STRING;
      DESCRIPTION: OPTIONAL STRING;
      MODIFICATION_STAMP: T_DATE;
      NUMBER_ATT: OPTIONAL STRING;
END_ENTITY;


--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF REQUIREMENTS
ENTITY REQ_TYPE;
      NAME: STRING;
END_ENTITY;

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF ORIGINS FOR A REQUIREMENT
ENTITY ORIGIN_TYPE;
      NAME: STRING;
END_ENTITY;

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF QUEUES
ENTITY QUEUE_TYPE;
      NAME: STRING;
END_ENTITY;

--THIS ENTITY REPRESENTS THE REQUIREMENTS OF THE MODELLED SYSTEM
ENTITY REQUIREMENT
      SUBTYPE OF (ENTITY_CORE);
      REQ_LIST: OPTIONAL REQ_TYPE;
      ORIGIN_LIST: OPTIONAL ORIGIN_TYPE;
      RATIONALE: OPTIONAL STRING;
      BASIS_OF: SET[0:?] OF ENTITY_CORE;
      RESULT_OF: SET[0:?] OF ENTITY_CORE;
      REFINES: SET[0:?] OF REQUIREMENT;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'REQUIREMENT';
INVERSE
      REFINED : SET[0:?] OF REQUIREMENT FOR REFINES;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATIONS OF CORE: USED WHEN ATTRIBUTS ARE NEEDED
(ASSOCIATION CLASS)
ENTITY CORE_RELATION
      SUBTYPE OF (ENTITY_CLASS);
      DESTINATION: ENTITY_CORE;
END_ENTITY;

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF BEHAVIORS FOR
ALLOCATED_TO_RELATION
ENTITY BEHAVIOR_TYPE;
      NAME: STRING;
END_ENTITY;
```

```
--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE FUNCTION_CORE AND ONE
COMPONENT
ENTITY ALLOCATED_TO_RELATION
	SUBTYPE OF (CORE_RELATION);
	SELF\CORE_RELATION.DESTINATION: COMPONENT;
	BEHAVIOR_TYPE: BEHAVIOR_TYPE;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE
OPERATIONAL_INFORMATION/ITEM_CORE AND ONE OPERATION/FUNCTION
ENTITY TRIGGERED_BY_RELATION
	SUBTYPE OF (CORE_RELATION);
	SELF\CORE_RELATION.DESTINATION: ITEM_CORE;
	QUEUE_TYPE: QUEUE_TYPE;
END_ENTITY;

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF EXITS
ENTITY EXIT_TYPE;
	NAME: STRING;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE FUNCTION AND ONE EXIT ELEMENT
ENTITY EXIT_RELATION
	SUBTYPE OF (CORE_RELATION);
	SELF\CORE_RELATION.DESTINATION: EXIT;
	SELECTION_PROBABILITY: OPTIONAL REAL; --PROBABILITY TO USE THE EXIT
	EXIT_TYPE: EXIT_TYPE;
END_ENTITY;

--THIS ENTITY REPRESENTS THE FUNCTIONS SUPPORTING OPERATIONS
ENTITY FUNCTION_CORE
	SUBTYPE OF (ENTITY_CORE);
	DURATION: OPTIONAL CALCULATION_KIND;
	SCRIPT: OPTIONAL STRING;
	TIME_OUT: OPTIONAL CALCULATION_KIND;
	EXECUTE_DECOMPOSITION: OPTIONAL BOOLEAN;
	DECOMPOSES: OPTIONAL SET[0:?] OF FUNCTION_CORE;
	SERVICES: OPTIONAL SET[0:?] OF LINK;
	SPECIFIED_BY: OPTIONAL SET[0:?] OF REQUIREMENT;
	ALLOCATED_TO: OPTIONAL ALLOCATED_TO_RELATION;
	INPUTS: OPTIONAL SET[0:?] OF ITEM_CORE;
	OUTPUTS: OPTIONAL SET[0:?] OF ITEM_CORE;
	TRIGGERED_BY: OPTIONAL SET[0:?] OF TRIGGERED_BY_RELATION;--ITEM_CORE;
	EXITS_BY: OPTIONAL SET[0:?] OF EXIT_RELATION;--EXIT;
	PRODUCES: OPTIONAL SET[0:?] OF PRODUCES_RELATION;
	CAPTURES: OPTIONAL SET[0:?] OF CAPTURES_RELATION;
	CONSUMES: OPTIONAL SET[0:?] OF CONSUMES_RELATION;
DERIVE
	SELF\ENTITY_CLASS.NAME:STRING := 'FUNCTION';
INVERSE
	BASEDON: SET[0:?] OF REQUIREMENT FOR BASIS_OF;
	DECOMPOSED_BY: SET[0:?] OF FUNCTION_CORE FOR DECOMPOSES;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY PRODUCES_RELATION
	SUBTYPE OF (CORE_RELATION);
	SELF\CORE_RELATION.DESTINATION: RESOURCE;
	AMOUNT: OPTIONAL CALCULATION_KIND;
```

```
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY CAPTURES_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: RESOURCE;
      AMOUNT: OPTIONAL CALCULATION_KIND;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY CONSUMES_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: RESOURCE;
      AMOUNT: OPTIONAL CALCULATION_KIND;
END_ENTITY;

--THIS ENTITY REPRESENTS THE FUNCTIONS SUPPORTING OPERATIONS
ENTITY RESOURCE
      SUBTYPE OF (ENTITY_CORE);
      INITIAL_AMOUNT: OPTIONAL REAL;
      MAXIMUM_AMOUNT: OPTIONAL REAL;
      AMOUNT_TYPE: OPTIONAL AMOUNT_KIND;
      UNITS: OPTIONAL REAL;
      PRODUCED_BY: OPTIONAL SET[0:?] OF PRODUCED_RELATION;
      CAPTURED_BY: OPTIONAL SET[0:?] OF CAPTURED_RELATION;
      CONSUMED_BY: OPTIONAL SET[0:?] OF CONSUMED_RELATION;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'RESOURCE';
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY PRODUCED_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: FUNCTION_CORE;
      AMOUNT: OPTIONAL CALCULATION_KIND;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY CAPTURED_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: RESOURCE;
      AMOUNT: OPTIONAL CALCULATION_KIND;
END_ENTITY;

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE RESOURCE AND ONE
FUNCTION_CORE
ENTITY CONSUMED_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: RESOURCE;
      AMOUNT: OPTIONAL CALCULATION_KIND;
END_ENTITY;

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF COMPONENTS
ENTITY COMPONENT_TYPE;
      NAME: STRING;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE COMPONENT AND ONE FUNCTION_CORE

```
ENTITY PERFORMS_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: FUNCTION_CORE;
      BEHAVIOR_TYPE: BEHAVIOR_TYPE;
END_ENTITY;
```

--THIS ENTITY REPRESENTS A PHYSICAL COMPONENT OF THE MODELLED SYSTEM

```
ENTITY COMPONENT
      SUBTYPE OF (ENTITY_CORE);
      ABBREVIATION: OPTIONAL STRING;
      COMP_TYPE: OPTIONAL COMPONENT_TYPE;
      PURPOSE: OPTIONAL STRING;
      MISSION: OPTIONAL STRING;
      BUILT_IN: SET[0:?] OF COMPONENT;
      CONNECTED_TO: SET[0:?] OF LINK;
      CONNECTED_THROUGH: SET[0:?] OF LINK; --DERIVED: THE LINKS ALLOCATED
TO A COMPONENT ARE ALSO ALLOCATED TO AN UPPER COMPONENT (BUILT FROM
RELATION)
      JOINED_TO: SET[0:?] OF INTERFACE_CORE;
      JOINED_THROUGH: SET[0:?] OF INTERFACE_CORE; --DERIVED: THE INTERFACES
ALLOCATED TO A COMPONENT ARE ALSO ALLOCATED TO AN UPPER COMPONENT (BUILT
FROM RELATION)
      PERFORMS: SET[0:?] OF PERFORMS_RELATION; --COMPLEMENT OF ALLOCATED_TO
      SPECIFIED_BY: SET[0:?] OF REQUIREMENT;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'COMPONENT';
INVERSE
      BUILT_FROM: SET[0:?] OF COMPONENT FOR BUILT_IN;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE INTERFACES OF THE MODELLED SYSTEM

```
ENTITY INTERFACE_CORE
      SUBTYPE OF (ENTITY_CORE);
      COMPRISED_OF: SET[0:?] OF LINK;
      SPECIFIED_BY: SET[0:?] OF REQUIREMENT;
      --CONSTRAINTS
      --IT SHOULD BE CONSISTENT WITH COMPRISES RELATION OF LINK (IF A LINK
CONNECTS 2 COMPONENTS:
      --THEN THE COMPONENTS JOINED BY AN INTERFACE COMPRISED BY THE LINK
SHOULD INCLUDE THEM)
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'INTERFACE_CORE';
INVERSE
      JOINS_THROUGH: SET[0:?] OF COMPONENT FOR JOINED_THROUGH;
      JOINS_TO: SET[0:?] OF COMPONENT FOR JOINED_TO;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE LINKS OF THE MODELLED SYSTEM

```
ENTITY LINK
      SUBTYPE OF (ENTITY_CORE);
      CAPACITY: OPTIONAL CALCULATION_KIND;
      CAPACITY_UNITS: OPTIONAL STRING;
      DELAI: OPTIONAL CALCULATION_KIND;
      DELAI_UNITS: OPTIONAL STRING;
      PROTOCOL: OPTIONAL STRING;
      SPECIFIED_BY: SET[0:?] OF REQUIREMENT;
```

```
	TRANSFERS:  SET[0:?] OF ITEM_CORE; --NOT NECESSARILY RELATED TO A
SERVICED_BY FUNCTION: BUT THE TRANSFERRED ITEM_CORES SHOULD BE THE INPUT OF
ONE OF THE FUNCTIONS OF SERVICED_BY RELATIONSHIP
	--CONSTRAINTS
	--THE TRANSFERRED ITEM_CORES SHOULD BE THE INPUT OF ONE OF THE
FUNCTIONS OF SERVICED_BY RELATIONSHIP
DERIVE
	SELF\ENTITY_CLASS.NAME:STRING := 'LINK';
INVERSE
	CONNECTS_THROUGH: SET[0:?] OF COMPONENT FOR CONNECTED_THROUGH;
	CONNECTS_TO: SET[0:?] OF COMPONENT FOR CONNECTED_TO;
	SERVICED_BY: SET[0:?] OF FUNCTION_CORE FOR SERVICES;
	COMPRISES: SET[0:?] OF INTERFACE_CORE FOR COMPRISED_OF;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE OPERATIONAL_INFORMATION AND ONE OPERATION

```
ENTITY TRIGGERS_OPERATION_RELATION
	SUBTYPE OF (CORE_RELATION);
	QUEUE_TYPE: QUEUE_TYPE;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF ITEM_CORES

```
ENTITY ITEM_CORE_TYPE;
	NAME: STRING;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE DIFFERENT TYPES OF MEDIAS SUPPORTED BY AN ITEM_CORE

```
ENTITY MEDIA_TYPE;
	NAME: STRING;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE OPERATIONAL_INFORMATION/ITEM_CORE AND ONE OPERATION/FUNCTION

```
ENTITY TRIGGERS_FUNCTION_RELATION
	SUBTYPE OF (CORE_RELATION);
	SELF\CORE_RELATION.DESTINATION: FUNCTION_CORE;
	QUEUE_TYPE: QUEUE_TYPE;
END_ENTITY;
```

--THIS ENTITY DESCRIBES THE ITEM_CORES PROCESSED BY THE FUNCTIONS

```
ENTITY ITEM_CORE
	SUBTYPE OF (ENTITY_CORE);
	ITEM_CORE_TYPE:  OPTIONAL ITEM_CORE_TYPE;
	SIZE: OPTIONAL CALCULATION_KIND;
	SIZE_UNITS: OPTIONAL STRING;
	MEDIA_TYPE:  OPTIONAL MEDIA_TYPE;
	PRIORITY: OPTIONAL STRING;
	ACCURACY: OPTIONAL STRING;
	TIMELINESS: OPTIONAL STRING;
	FORMAT_TYPE: OPTIONAL STRING;
	FIELDS: SET[0:?] OF STRING;
	DECOMPOSES: SET[0:?] OF ITEM_CORE;
	TRIGGERS: SET[0:?] OF TRIGGERS_FUNCTION_RELATION;
	SPECIFIED_BY: SET[0:?] OF REQUIREMENT;
DERIVE
	SELF\ENTITY_CLASS.NAME:STRING := 'ITEM';
INVERSE
	DECOMPOSED_BY: SET[0:?] OF ITEM_CORE FOR DECOMPOSES;
	INPUT_TO: SET[0:?] OF FUNCTION_CORE FOR INPUTS;
	OUTPUT_FROM: SET[0:?] OF FUNCTION_CORE FOR OUTPUTS;
```

```
      TRANSFERRED_BY: SET[0:?] OF LINK FOR TRANSFERS;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE RELATION BETWEEN ONE FUNCTION AND ONE EXIT ELEMENT

```
ENTITY EXIT_FOR_RELATION
      SUBTYPE OF (CORE_RELATION);
      SELF\CORE_RELATION.DESTINATION: FUNCTION_CORE;
      SELECTION_PROBABILITY: OPTIONAL REAL; --PROBABILITY TO USE THE EXIT
      EXIT_TYPE: EXIT_TYPE;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE CONDITIONS TO EXIT (FINISH) A FUNCTION

```
ENTITY EXIT
      SUBTYPE OF (ENTITY_CORE);
      EXIT_FOR: SET[0:?] OF EXIT_FOR_RELATION; --FUNCTION_CORE;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'EXIT';
END_ENTITY;
```

--THIS ENTITY DESCRIBES THE BEHAVIOUR OF THE MODELLED SYSTEM: TO BE COMPLETED

```
ENTITY FLOW_CORE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CORE);
END_ENTITY;
```

--THIS ENTITY REPRESENTS AN EFFBD DIAGRAM

```
ENTITY EFFBD
      SUBTYPE OF (FLOW_CORE);
      ITS_CONSTRUCTS: SET[0:?] OF EFFBD_CONSTRUCT;
      REPRESENTS: FUNCTION_CORE;
END_ENTITY;
```

--THIS ENTITY REPRESENTS AN EFFBD CONSTRUCT

```
ENTITY EFFBD_CONSTRUCT
      ABSTRACT SUPERTYPE ;
END_ENTITY;
```

--THIS ENTITY REPRESENTS A REPLICATE CONSTRUCT

```
ENTITY REPLICATE_CONSTRUCT
      SUBTYPE OF (EFFBD_CONSTRUCT);
      CONTAINS: SET[0:?] OF EFFBD_CONSTRUCT;
END_ENTITY;
```

--THIS ENTITY REPRESENTS A LOOP

```
ENTITY LOOP_CONSTRUCT
      SUBTYPE OF (REPLICATE_CONSTRUCT);
END_ENTITY;
```

--THIS ENTITY REPRESENTS AN ITERATION

```
ENTITY ITERATE_CONSTRUCT
      SUBTYPE OF (REPLICATE_CONSTRUCT);
END_ENTITY;
```

-- TYPE REPRESENTING POSSIBLE INPUTS/OUTPUTS OF A BRANCH

```
TYPE T_BRANCH_TARGET = SELECT (PARALLEL_CONSTRUCT,FUNCTION_CORE) ;
END_TYPE ;
```

--THIS ENTITY REPRESENTS A BRANCH

```
ENTITY BRANCH_CONSTRUCT
      SUBTYPE OF (EFFBD_CONSTRUCT);
```

```
      INPUTS: T_BRANCH_TARGET;
      OUTPUTS: T_BRANCH_TARGET;
      EXITS_BY: OPTIONAL EXIT;
      ANNOTATION: OPTIONAL STRING;
END_ENTITY;

--THIS ENTITY REPRESENTS A PARALLEL CONSTRUCT
ENTITY PARALLEL_CONSTRUCT
      ABSTRACT SUPERTYPE
      SUBTYPE OF (EFFBD_CONSTRUCT);
END_ENTITY;

--THIS ENTITY REPRESENTS AN AND
ENTITY AND_CONSTRUCT
      SUBTYPE OF (PARALLEL_CONSTRUCT);
END_ENTITY;

--THIS ENTITY REPRESENTS AN OR
ENTITY OR_CONSTRUCT
      SUBTYPE OF (PARALLEL_CONSTRUCT);
END_ENTITY;

END_SCHEMA;
```

SysML Meta-Model

```
--THIS SCHEMA CONTAINS THE ENTITIES DESCRIBING THE SYSML METAMODEL
SCHEMA SYSML_SCHEMA;

REFERENCE FROM TOP_SCHEMA;

--THIS ENTITY REPRESENTS SYSML MODELING LANGUAGE
ENTITY SYSML_MODELING_LANGUAGE
      SUBTYPE OF (MODELING_LANGUAGE);
      VERSION: OPTIONAL STRING;
DERIVE
      NAME:STRING := 'SYSML';
END_ENTITY;

--THIS ENTITY REPRESENTS ONE SYSML MODEL, A MODEL IS AN ABSTRACTION OF A
PHYSICAL SYSTEM (UML SPECIFICATION)
ENTITY SYSML_MODEL
      SUBTYPE OF (MODEL);
      ITS_PACKAGES: OPTIONAL SET[0:?] OF PACKAGE;
      ITS_ELEMENTS: OPTIONAL SET[0:?] OF ELEMENT;
      SELF\MODEL.MODELING_LANGUAGE:SYSML_MODELING_LANGUAGE;
END_ENTITY;

(************************************************************************)
(*********************** UML 2.0 METAMODEL **************************)
(************************************************************************)

ENTITY ELEMENT
      SUBTYPE OF (ENTITY_CLASS);
      OWNED_ELEMENT: OPTIONAL SET[0:?] OF ELEMENT;
      OWNER: OPTIONAL ELEMENT;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'ELEMENT';
END_ENTITY;

(**********************************************)

--THIS ENTITY REPRESENTS AN ELEMENT IN A MODEL THAT MAY HAVE A NAME (PAGE 98 OF
UML SPEC)
ENTITY NAMED_ELEMENT
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ELEMENT);
      NAME_ELEMENT: OPTIONAL STRING;
      QUALIFIED_NAME: OPTIONAL STRING; --DERIVED
      VISIBILITY: OPTIONAL VISIBILITY_KIND;
      CLIENT_DEPENDENCY: OPTIONAL DEPENDENCY; --INDICATES THE DEPENDENCIES
THAT REFERENCE THE CLIENT
      NAMESPACE: OPTIONAL NAMESPACE; --DERIVED
DERIVE
      SELF\ELEMENT.NAME:STRING := 'NAMED_ELEMENT';
END_ENTITY;

(**********************************************)

--THIS ENTITY REPRESENTS AN ELEMENT THAT CAN BE EXPOSED AS A FORMAL TEMPLATE
PARAMETER FOR A TEMPLATE, OR SPECIFIED AS
```

--AN ACTUAL PARAMETER IN A BINDING OF A TEMPLATE (PAGE 623 OF UML SPEC)

```
ENTITY PACKAGEABLE_ELEMENT
	ABSTRACT SUPERTYPE
	SUBTYPE OF (ELEMENT);
DERIVE
	SELF\ELEMENT.NAME:STRING := 'PACKAGEABLE_ELEMENT';
END_ENTITY;
```

(***)

--IT IS A KIND OF CLASSIFIER THAT REPRESENTS A DECLARATION OF A SET OF COHERENT PUBLIC FEATURES AND OBLIGATIONS (UML SPEC, PAGE 86)

```
ENTITY INTERFACE_UML
	SUBTYPE OF (CLASSIFIER);
	OWNED_ATTRIBUTE: OPTIONAL SET[0:?] OF PROPERTY_UML;
	OWNED_OPERATION: OPTIONAL SET[0:?] OF OPERATION_UML;
	NESTED_CLASSIFIER: OPTIONAL SET[0:?] OF CLASSIFIER;
	REDEFINED_INTERFACE: OPTIONAL SET[0:?] OF INTERFACE_UML;
DERIVE
	SELF\CLASSIFIER.NAME:STRING := 'INTERFACE_UML';
END_ENTITY;
```

(***)

--A BEHAVIORAL FEATURE IS IMPLEMENTED (REALIZED) BY A BEHAVIOR (PAGE 432 OF UML SPEC)

```
ENTITY BEHAVIORAL_FEATURE
	ABSTRACT SUPERTYPE
	SUBTYPE OF (FEATURE);
	OWNED_PARAMETER: SET OF PARAMETER_UML;
	IS_ABSTRACT: OPTIONAL BOOLEAN;
	METHOD: OPTIONAL SET[0:?] OF BEHAVIOR;
	RAISED_EXCEPTION: OPTIONAL SET[0:?] OF CLASSIFIER;
DERIVE
	SELF\FEATURE.NAME:STRING := 'BEHAVIORAL_FEATURE';
END_ENTITY;
```

(***)

--THIS ENTITY IS A IS A BEHAVIORAL FEATURE OF A CLASSIFIER THAT SPECIFIES THE NAME, TYPE, PARAMETERS, AND CONSTRAINTS FOR INVOKING
--AN ASSOCIATED BEHAVIOR (FROM UML SPEC, PAGE 103)

```
ENTITY OPERATION_UML
	SUBTYPE OF (BEHAVIORAL_FEATURE);
	IS_ORDERED : OPTIONAL BOOLEAN; --DERIVED.
	IS_QUERY : OPTIONAL BOOLEAN;
	IS_UNIQUE : OPTIONAL BOOLEAN; --DERIVED.
	LOWER : OPTIONAL INTEGER; --DERIVED.
	UPPER : OPTIONAL INTEGER; -- >=0, DERIVED.
	CLASS : OPTIONAL CLASS_UML;
	BODY_CONDITION: OPTIONAL CONSTRAINT_UML; --AN OPTIONAL CONSTRAINT ON
THE RESULT VALUES OF AN INVOCATION OF THIS OPERATION.
	POST_CONDITION: SET [0:?] OF CONSTRAINT_UML;
	PRE_CONDITION: SET [0:?] OF CONSTRAINT_UML;
	REDEFINED_OOPERATION: SET [0:?] OF OPERATION_UML;
	RETURN_TYPE: OPTIONAL TYPE_UML; --DERIVED
	OWNER_INTERFACE: OPTIONAL INTERFACE_UML;
DERIVE
	SELF\BEHAVIORAL_FEATURE.NAME:STRING := 'OPERATION_UML';
END_ENTITY;
```

(***)

--THIS ENTITY IS A SPECIFICATION OF AN ARGUMENT USED TO PASS INFORMATION INTO OR OUT OF AN INVOCATION OF A BEHAVIORAL
--FEATURE. (PAGE 120 OF UML SPEC)

```
ENTITY PARAMETER_UML
      SUBTYPE OF (MULTIPLICITY_ELEMENT,TYPED_ELEMENT, CONNECTABLE_ELEMENT);
      DEFAULT: OPTIONAL STRING; --DERIVED
      DIRECTION: OPTIONAL PARAMETER_DIRECTION_KIND;
      OPERATION: OPTIONAL OPERATION_UML; --DERIVED
      DEFAULT_VALUE: OPTIONAL VALUE_SPECIFICATION;
DERIVE
      SELF\ELEMENT.NAME:STRING := 'PARAMETER_UML';
END_ENTITY;
```

(**)

--THIS ENTITY REPRESENTS AN ELEMENT THAT CONTAINS A SET OF NAMED ELEMENTS THAT CAN BE IDENTIFIED BY NAME (PAGE 100 OF UML SPEC)

```
ENTITY NAMESPACE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (NAMED_ELEMENT);
      OWNED_MEMBER: OPTIONAL SET[0:?] OF NAMED_ELEMENT; --DERIVED
      MEMBER: OPTIONAL NAMED_ELEMENT; --DERIVED
DERIVE
      SELF\NAMED_ELEMENT.NAME:STRING := 'NAMESPACE';
END_ENTITY;
```

--IT REPRESENTS THE DIFFERENT TYPES OF PARAMETER DIRECTION

```
TYPE VISIBILITY_KIND = ENUMERATION OF (PUBLIC,PRIVATE,PROTECTED,PACKAGE);
END_TYPE;
```

--IT REPRESENTS THE DIFFERENT TYPES OF AGGREGATION

```
TYPE AGGREGATION_KIND = ENUMERATION OF (NONE,SHARED,COMPOSITE);
END_TYPE;
```

--THIS ENTITY IS THE SPECIFICATION OF A (POSSIBLY EMPTY) SET OF INSTANCES, INCLUDING BOTH OBJECTS AND DATA VALUES (PAGE 137 OF UML SPEC)

```
ENTITY VALUE_SPECIFICATION
      SUBTYPE OF (PACKAGEABLE_ELEMENT,TYPED_ELEMENT);
END_ENTITY;
```

--A LITERAL SPECIFICATION IDENTIFIES A LITERAL CONSTANT BEING MODELED(PAGE 92 OF UML SPEC)

```
ENTITY LITERAL_SPECIFICATION
      SUBTYPE OF (VALUE_SPECIFICATION);
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF A BOOLEAN VALUE(PAGE 89 OF UML SPEC)

```
ENTITY LITERAL_BOOLEAN
      SUBTYPE OF (LITERAL_SPECIFICATION);
      THE_VALUE: BOOLEAN;
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF A INTEGER VALUE(PAGE 89 OF UML SPEC)

```
ENTITY LITERAL_INTEGER
	SUBTYPE OF (LITERAL_SPECIFICATION);
	THE_VALUE: NUMBER;
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF A STRING VALUE(PAGE 92 OF UML SPEC)

```
ENTITY LITERAL_STRING
	SUBTYPE OF (LITERAL_SPECIFICATION);
	THE_VALUE: STRING;
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF AN UNLIMITED NATURAL VALUE(PAGE 93 OF UML SPEC)

```
ENTITY LITERAL_UNLIMITED_NATURAL
	SUBTYPE OF (LITERAL_SPECIFICATION);
	THE_VALUE: NUMBER;
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF A LACK VALUE(PAGE 91 OF UML SPEC)

```
ENTITY LITERAL_NULL
	SUBTYPE OF (LITERAL_SPECIFICATION);
END_ENTITY;
```

--A CONSTRAINT IS A CONDITION OR RESTRICTION EXPRESSED IN NATURAL LANGUAGE TEXT OR IN A MACHINE READABLE LANGUAGE FOR THE
--PURPOSE OF DECLARING SOME OF THE SEMANTICS OF AN ELEMENT (PAGE 58 OF UML SPEC)

```
ENTITY CONSTRAINT_UML
	ABSTRACT SUPERTYPE
	SUBTYPE OF (PACKAGEABLE_ELEMENT);
	CONSTRAINED_ELEMENT: OPTIONAL SET[0:?] OF ELEMENT;
	CONTEXT_UML: OPTIONAL NAMESPACE; --DERIVED
	SPECIFICATION: VALUE_SPECIFICATION;
END_ENTITY;
```

--THIS ENTITY CONSTRAINS THE VALUES REPRESENTED BY A TYPED ELEMENT (PAGE 135 OF UML SPEC)

```
ENTITY TYPE_UML
	ABSTRACT SUPERTYPE
	SUBTYPE OF (PACKAGEABLE_ELEMENT);
DERIVE
	SELF\PACKAGEABLE_ELEMENT.NAME:STRING := 'TYPE_UML';
END_ENTITY;
```

--THIS ENTITY IS AN ELEMENT THAT, WHEN DEFINED IN THE CONTEXT OF A CLASSIFIER, CAN BE REDEFINED MORE SPECIFICALLY OR
--DIFFERENTLY IN THE CONTEXT OF ANOTHER CLASSIFIER THAT SPECIALIZES (DIRECTLY OR INDIRECTLY) THE CONTEXT CLASSIFIER (PAGE 130 OF UML SPEC)

```
ENTITY REDEFINABLE_ELEMENT
	ABSTRACT SUPERTYPE
	SUBTYPE OF (NAMED_ELEMENT);
	IS_LEAF: OPTIONAL BOOLEAN;
	REDEFINED_ELEMENT: OPTIONAL SET[0:?] OF REDEFINABLE_ELEMENT; --
DERIVED (THE REDEFINABLE ELEMENT THAT IS BEING REDEFINED BY THIS ELEMENT)
	REDEFINED_CONTEXT: OPTIONAL SET[0:?] OF CLASSIFIER; --DERIVED
DERIVE
	SELF\NAMED_ELEMENT.NAME:STRING := 'REDEFINABLE_ELEMENT';
END_ENTITY;
```

--THIS ENTITY REPRESENTS A LINK BETWEEN TWO OR MORE CONNECTABLE ELEMENTS (FROM UML SPEC, PAGE 175)

```
ENTITY CONNECTOR
      SUBTYPE OF (FEATURE);
      END_CONNECTOR: SET[2:?] OF CONNECTOR_END;
      TYPE_CONNECTOR: OPTIONAL ASSOCIATION;
      REDEFINED_CONNECTOR: OPTIONAL SET[0:?] OF CONNECTOR;
DERIVE
      SELF\FEATURE.NAME:STRING := 'CONNECTOR';
END_ENTITY;
```

(***
*****)

--THIS ENTITY IS A CLASSIFICATION OF INSTANCES, IT DESCRIBES A SET OF INSTANCES THAT HAVE FEATURES IN COMMON (FROM UML SPEC, PAGE 52)

```
ENTITY CLASSIFIER
      ABSTRACT SUPERTYPE
      SUBTYPE OF (NAMESPACE,REDEFINABLE_ELEMENT,TYPE_UML);
      IS_ENCAPSULATED: OPTIONAL BOOLEAN;
      ATTRIBUTE: OPTIONAL SET[0:?] OF PROPERTY_UML; --DERIVED
      FEATURE: OPTIONAL SET[0:?] OF FEATURE; --DERIVED
      GENERAL: OPTIONAL SET[0:?] OF CLASSIFIER; --DERIVED
      GENERALIZATION: OPTIONAL SET[0:?] OF GENERALIZATION;
      INHERITED_ELEMENT: OPTIONAL SET[0:?] OF NAMED_ELEMENT; --DERIVED
      REDEFINED_CLASSIFIER: OPTIONAL SET[0:?] OF CLASSIFIER; --DERIVED
(REFERENCES THE CLASSIFIERS THAT ARE REDEFINED BY THIS CLASSIFIER)
      COLLABORATION_USE: OPTIONAL COLLABORATION_USE;
      REPRESENTATION: OPTIONAL COLLABORATION_USE;
DERIVE
      SELF\NAMESPACE.NAME:STRING := 'CLASSIFIER';
END_ENTITY;
```

(***
******)

--THIS ENTITY REPRESENTS A PACKAGE (FROM UML 2 SPEC PAGE 108)

```
ENTITY PACKAGE
      SUBTYPE OF (NAMED_ELEMENT,PACKAGEABLE_ELEMENT);
      NESTED_PACKAGE: OPTIONAL SET[0:?] OF PACKAGE;
      PACKAGED_ELEMENT: OPTIONAL SET[0:?] OF PACKAGEABLE_ELEMENT;
      NESTING_PACKAGE: OPTIONAL PACKAGE;
DERIVE
      SELF\PACKAGEABLE_ELEMENT.NAME:STRING := 'PACKAGE';
END_ENTITY;
```

(***
******)

--THIS ENTITY IS AN ABSTRACT METACLASS THAT REPRESENTS ANY CLASSIFIER WHOSE BEHAVIOR CAN BE FULLY OR PARTLY
--DESCRIBED BY THE COLLABORATION OF OWNED OR REFERENCED INSTANCES. (FROM UML SPEC, PAGE 186)

```
ENTITY STRUCTURED_CLASSIFIER
      ABSTRACT SUPERTYPE
      SUBTYPE OF (CLASSIFIER);
      ROLE: OPTIONAL SET[0:?] OF CONNECTABLE_ELEMENT; --DERIVED
      OWNED_ATTRIBUTE: OPTIONAL SET[0:?] OF PROPERTY_UML;
      PART: OPTIONAL SET[0:?] OF PROPERTY_UML; --DERIVED
      OWNED_CONNECTOR: OPTIONAL SET[0:?] OF CONNECTOR;
```

```
DERIVE
      SELF\CLASSIFIER.NAME:STRING := 'STRUCTURED_CLASSIFIER';
END_ENTITY;
```

(***
******)
--THIS ENTITY IS AN ENDPOINT OF A CONNECTOR, WHICH ATTACHES THE CONNECTOR TO A CONNECTABLE ELEMENT. EACH CONNECTOR
--END IS PART OF ONE CONNECTOR. (FROM UML SPEC, PAGE 176)

```
ENTITY CONNECTOR_END
      SUBTYPE OF (MULTIPLICITY_ELEMENT);
      ROLE: CONNECTABLE_ELEMENT;
      DEFINING_END: OPTIONAL PROPERTY_UML;
      PART_WITH_PORT: OPTIONAL PROPERTY_UML;
DERIVE
      SELF\MULTIPLICITY_ELEMENT.NAME:STRING := 'CONNECTOR_END';
END_ENTITY;
```

(***
******)
--THIS ENTITY IS AN ABSTRACT METACLASS REPRESENTING A SET OF INSTANCES THAT PLAY ROLES OF A CLASSIFIER. (FROM UML SPEC, PAGE 174)

```
ENTITY CONNECTABLE_ELEMENT
      ABSTRACT SUPERTYPE
      SUBTYPE OF (TYPED_ELEMENT);
      END_CONNECTOR: OPTIONAL CONNECTOR_END;
END_ENTITY;
```

--THIS ENTITY DESCRIBES A STRUCTURE OF COLLABORATING ELEMENTS (ROLES), EACH PERFORMING A SPECIALIZED FUNCTION, WHICH
--COLLECTIVELY ACCOMPLISH SOME DESIRED FUNCTIONALITY. (FROM UML SPEC, PAGE 168)

```
ENTITY COLLABORATION
      ABSTRACT SUPERTYPE
      SUBTYPE OF (BEHAVIORED_CLASSIFIER,STRUCTURED_CLASSIFIER);
      COLLABORATION_ROLE: OPTIONAL SET[0:?] OF CONNECTABLE_ELEMENT;
END_ENTITY;
```

--THIS ENTITY REPRESENTS THE APPLICATION OF THE PATTERN DESCRIBED BY A COLLABORATION TO A SPECIFIC SITUATION INVOLVING
--SPECIFIC CLASSES OR INSTANCES PLAYING THE ROLES OF THE COLLABORATION (FROM UML SPEC, PAGE 171)

```
ENTITY COLLABORATION_USE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (NAMED_ELEMENT);
      TYPE_COLLABORATION: COLLABORATION;
      ROLE_BINDING: OPTIONAL SET[0:?] OF DEPENDENCY;
END_ENTITY;
```

(***
******)

--THIS EXTENDS A CLASSIFIER WITH THE ABILITY TO OWN PORTS AS SPECIFIC AND TYPE CHECKED INTERACTION POINTS (FROM UML SPEC, PAGE 178)

```
ENTITY ENCAPSULATED_CLASSIFIER
      ABSTRACT SUPERTYPE
      SUBTYPE OF (STRUCTURED_CLASSIFIER);
      OWNED_PORT: OPTIONAL SET[0:?] OF PORT;
DERIVE
      SELF\STRUCTURED_CLASSIFIER.NAME:STRING := 'ENCAPSULATED_CLASSIFIER';
END_ENTITY;
```

(***
******)

--THIS ENTITY DESCRIBES A SET OF OBJECTS THAT SHARE THE SAME SPECIFICATIONS OF FEATURES, CONSTRAINTS, AND SEMANTICS (FROM UML SPEC, PAGE 49)

```
ENTITY CLASS_UML
	ABSTRACT SUPERTYPE
	SUBTYPE OF (CLASSIFIER, ENCAPSULATED_CLASSIFIER);
	IS_ABSTRACT: OPTIONAL BOOLEAN;
	NESTED_CLASSIFIER: OPTIONAL SET[0:?] OF CLASSIFIER;
	SUPER_CLASS: OPTIONAL SET[0:?] OF CLASS_UML; --DERIVED
DERIVE
	SELF\ENCAPSULATED_CLASSIFIER.NAME:STRING := 'CLASS_UML';
END_ENTITY;
```

(***
******)

--THIS ENTITY IS A DEFINITION OF AN INCLUSIVE INTERVAL OF NON-NEGATIVE INTEGERS BEGINNING WITH A LOWER BOUND AND ENDING
--WITH A (POSSIBLY INFINITE) UPPER BOUND. A MULTIPLICITY ELEMENT EMBEDS THIS INFORMATION TO SPECIFY THE ALLOWABLE
--CARDINALITIES FOR AN INSTANTIATION OF THIS ELEMENT. (FROM UML SPEC, PAGE 94)

```
ENTITY MULTIPLICITY_ELEMENT
	ABSTRACT SUPERTYPE
	SUBTYPE OF (NAMED_ELEMENT);
	IS_ORDERED: OPTIONAL BOOLEAN;
	IS_UNIQUE: OPTIONAL BOOLEAN;
	LOWER: OPTIONAL INTEGER; --DERIVED
	UPPER: OPTIONAL INTEGER; --DERIVED, UNLIMITED NATURAL >=0
	LOWER_VALUE: OPTIONAL VALUE_SPECIFICATION;
	UPPER_VALUE: OPTIONAL VALUE_SPECIFICATION;
DERIVE
	SELF\NAMED_ELEMENT.NAME:STRING := 'MULTIPLICITY_ELEMENT';
END_ENTITY;
```

(***
******)
--THIS ENTITY HAS A TYPE (FROM UML SPEC, PAGE 136)

```
ENTITY TYPED_ELEMENT
	SUBTYPE OF (NAMED_ELEMENT);
	TYPE_UML: OPTIONAL TYPE_UML;
DERIVE
	SELF\NAMED_ELEMENT.NAME:STRING := 'TYPED_ELEMENT';
END_ENTITY;
```

(***
******)
--THIS ENTITY DECLARES A BEHAVIORAL OR STRUCTURAL CHARACTERISTIC OF INSTANCES OF CLASSIFIERS (FROM UML SPEC, PAGE 70)

```
ENTITY FEATURE
	ABSTRACT SUPERTYPE
	SUBTYPE OF (REDEFINABLE_ELEMENT);
	IS_STATIC: OPTIONAL BOOLEAN;
	FEATURING_CLASSIFIER: OPTIONAL SET[0:?] OF CLASSIFIER; --DERIVED
DERIVE
	SELF\REDEFINABLE_ELEMENT.NAME:STRING := 'FEATURE';
END_ENTITY;
```

(**)

--THIS ENTITY IS A IS A TYPED FEATURE OF A CLASSIFIER THAT SPECIFIES THE STRUCTURE OF INSTANCES OF THE CLASSIFIER (FROM UML SPEC, PAGE 133)

```
ENTITY STRUCTURAL_FEATURE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (FEATURE,MULTIPLICITY_ELEMENT,TYPED_ELEMENT);
      IS_READ_ONLY: OPTIONAL BOOLEAN;
DERIVE
      SELF\FEATURE.NAME:STRING := 'STRUCTURAL_FEATURE';
END_ENTITY;
```

(**)

--THIS ENTITY IS AN ABSTRACT CONCEPT THAT SPECIFIES SOME KIND OF RELATIONSHIP BETWEEN ELEMENTS (FROM UML SPEC, PAGE 131)

```
ENTITY RELATIONSHIP
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ELEMENT);
      RELATED_ELEMENT: OPTIONAL SET [1:?] OF ELEMENT; --DERIVED
DERIVE
      SELF\ELEMENT.NAME:STRING := 'RELATIONSHIP';
END_ENTITY;
```

(**)

--THIS ENTITY DESCRIBES A SET OF TUPLES WHOSE VALUES REFER TO TYPED INSTANCES (FROM UML SPEC, PAGE 39)

```
ENTITY ASSOCIATION
      SUBTYPE OF (CLASSIFIER,RELATIONSHIP);
      IS_DERIVED: OPTIONAL BOOLEAN;
      MEMBER_END: SET [2:?] OF PROPERTY_UML;
      OWNED_END: OPTIONAL SET [0:?] OF PROPERTY_UML;
      NAVIGABLE_OWNED_END: OPTIONAL SET [0:?] OF PROPERTY_UML;
      ENDTYPE: OPTIONAL SET [1:?] OF TYPE_UML;
DERIVE
      SELF\RELATIONSHIP.NAME:STRING := 'ASSOCIATION';
END_ENTITY;
```

(**)

--THIS ENTITY IS REFERENCES ONE OR MORE SOURCE ELEMENTS AND ONE OR MORE TARGET ELEMENTS (FROM UML SPEC, PAGE 63)

```
ENTITY DIRECTED_RELATIONSHIP
      ABSTRACT SUPERTYPE
      SUBTYPE OF (RELATIONSHIP);
      SOURCE: SET [1:?] OF ELEMENT; --DERIVED
      TARGET: SET [1:?] OF ELEMENT; --DERIVED
DERIVE
      SELF\RELATIONSHIP.NAME:STRING := 'DIRECTED_RELATIONSHIP';
END_ENTITY;
```

(**)

--THIS ENTITY IS A RELATIONSHIP THAT SIGNIFIES THAT A SINGLE OR A SET OF MODEL ELEMENTS REQUIRES OTHER MODEL ELEMENTS FOR
--THEIR SPECIFICATION OR IMPLEMENTATION (FROM UML SPEC, PAGE 62)

```
ENTITY DEPENDENCY
      SUBTYPE OF (DIRECTED_RELATIONSHIP,PACKAGEABLE_ELEMENT);
      CLIENT: SET [1:?] OF NAMED_ELEMENT;
```

```
    SUPPLIER: SET [1:?] OF NAMED_ELEMENT;
DERIVE
    SELF\DIRECTED_RELATIONSHIP.NAME:STRING := 'DEPENDENCY';
END_ENTITY;
```

(**)
--THIS ENTITY IS A TAXONOMIC RELATIONSHIP BETWEEN A MORE GENERAL CLASSIFIER AND A MORE SPECIFIC CLASSIFIER. EACH
--INSTANCE OF THE SPECIFIC CLASSIFIER IS ALSO AN INDIRECT INSTANCE OF THE GENERAL CLASSIFIER. (UML SPEC., PAGE 71)

```
ENTITY GENERALIZATION
    SUBTYPE OF (DIRECTED_RELATIONSHIP);
    IS_SUBSTITUTABLE: OPTIONAL BOOLEAN;
    GENERAL: CLASSIFIER;
    SPECIFIC: CLASSIFIER;
    GENERALIZATION_SET: OPTIONAL SET[0:?] OF GENERALIZATION; --DESIGNATES
A SET IN WHICH INSTANCES OF GENERALIZATION ARE CONSIDERED MEMBERS
END_ENTITY;
```

(**)
--THIS ENTITY IS A RELATIONSHIP THAT RELATES TWO ELEMENTS OR SETS OF ELEMENTS THAT REPRESENT THE SAME CONCEPT AT DIFFERENT
--LEVELS OF ABSTRACTION OR FROM DIFFERENT VIEWPOINTS. IN THE METAMODEL, AN ABSTRACTION IS A DEPENDENCY IN WHICH THERE IS A
--MAPPING BETWEEN THE SUPPLIER AND THE CLIENT (UML SPEC., PAGE 38)

```
ENTITY ABSTRACTION
    ABSTRACT SUPERTYPE
    SUBTYPE OF (DEPENDENCY);
END_ENTITY;
```

--THIS ENTITY IS A SPECIALIZED ABSTRACTION RELATIONSHIP BETWEEN TWO SETS OF MODEL ELEMENTS, ONE REPRESENTING A SPECIFICATION
--(THE SUPPLIER) AND THE OTHER REPRESENTS AN IMPLEMENTATION OF THE LATTER (THE CLIENT). REALIZATION CAN BE USED TO MODEL
--STEPWISE REFINEMENT, OPTIMIZATIONS, TRANSFORMATIONS, TEMPLATES, MODEL SYNTHESIS, FRAMEWORK COMPOSITION, ETC.. (UML SPEC., PAGE 129)

```
ENTITY REALIZATION
    SUBTYPE OF (ABSTRACTION);
END_ENTITY;
```

--IT REPRESENTS THE DIFFERENT TYPES OF PARAMETER DIRECTION

```
TYPE PARAMETER_DIRECTION_KIND=ENUMERATION OF
(IN_UML,OUT,INOUT,RETURN_UML);--IN AND RETURN EXIST IN EXPRESS --YAMINE
END_TYPE;
```

--A CLASSIFIER CAN HAVE BEHAVIOR SPECIFICATIONS DEFINED IN ITS NAMESPACE (PAGE 434 OF UML SPEC)

```
ENTITY BEHAVIORED_CLASSIFIER
    ABSTRACT SUPERTYPE
    SUBTYPE OF (CLASSIFIER);
    OWNED_BEHAVIOR: OPTIONAL SET[0:?] OF BEHAVIOR;
    CLASSIFIER_BEHAVIOR: OPTIONAL BEHAVIOR;
    OWNED_TRIGGER: OPTIONAL SET[0:?] OF TRIGGER_UML;
END_ENTITY;
```

--THIS ENTITY IS A SPECIFICATION OF HOW ITS CONTEXT CLASSIFIER CHANGES STATE OVER TIME (PAGE 430 OF UML SPEC)

```
ENTITY BEHAVIOR
      ABSTRACT SUPERTYPE
      SUBTYPE OF (CLASS_UML);
      IS_REENTRANT: OPTIONAL BOOLEAN;
      SPECIFICATION: OPTIONAL BEHAVIORAL_FEATURE;
      CONTEXT_UML: OPTIONAL BEHAVIORED_CLASSIFIER; --DERIVED
      OWNED_PARAMETER: OPTIONAL SET[0:?] OF PARAMETER_UML;
      REDEFINED_BEHAVIOR: OPTIONAL BEHAVIOR;
      PRECONDITION: OPTIONAL SET[0:?] OF CONSTRAINT_UML;
      POSTCONDITION: OPTIONAL SET[0:?] OF CONSTRAINT_UML;
END_ENTITY;
```

(**
******)

--THIS ENTITY DESCRIBES A TYPE WHOSE INSTANCES ARE IDENTIFIED ONLY BY THEIR VALUE. A DATATYPE MAY CONTAIN ATTRIBUTES TO SUPPORT THE
--MODELING OF STRUCTURED DATA TYPES. (UML SPEC., PAGE 60)

```
ENTITY DATA_TYPE_UML
      SUBTYPE OF (CLASSIFIER);
      OWNED_ATTRIBUTE: LIST OF PROPERTY_UML;--THE ATTRIBUTES OWNED BY THE
DATATYPE. THIS IS AN ORDERED COLLECTION.
      OWNED_OPERATION:  LIST OF OPERATION_UML;
DERIVE
      SELF\CLASSIFIER.NAME:STRING := 'DATA_TYPE';
END_ENTITY;
```

(**
******)

--THIS ENTITY DEFINES A PREDEFINED DATA TYPE, WITHOUT ANY RELEVANT SUBSTRUCTURE (I.E., IT HAS NO PARTS IN THE CONTEXT OF
--UML)(PAGE 122 OF UML SPEC)

```
ENTITY PRIMITIVE_TYPE_UML
      SUBTYPE OF (DATA_TYPE_UML);
END_ENTITY;
```

(**
******)

--THIS ENTITY IS A STRUCTURAL FEATURE (FROM UML SPEC, PAGE 122)

```
ENTITY PROPERTY_UML
      SUBTYPE OF (STRUCTURAL_FEATURE, CONNECTABLE_ELEMENT);
      AGGREGATION: OPTIONAL AGGREGATION_KIND;
      DEFAULT: OPTIONAL STRING; --DERIVED
      IS_COMPOSITE: OPTIONAL BOOLEAN; --DERIVED
      IS_DERIVED: OPTIONAL BOOLEAN;
      IS_DERIVED_UNION: OPTIONAL BOOLEAN;
      SUPER_CLASS: OPTIONAL SET[0:?] OF CLASS_UML; --DERIVED
      ASSOCIATION: OPTIONAL ASSOCIATION;
      OWNING_ASSOCIATION: OPTIONAL ASSOCIATION;
      DATA_TYPE: OPTIONAL DATA_TYPE_UML;
      DEFAULT_VALUE: OPTIONAL VALUE_SPECIFICATION;
      REDEFINED_PROPERTY: OPTIONAL SET[0:?] OF PROPERTY_UML;
      SUBSETTED_PROPERTY: OPTIONAL SET[0:?] OF PROPERTY_UML;
      OPPOSITE: OPTIONAL PROPERTY_UML;
      CLASS_UML: OPTIONAL SET[0:?] OF CLASS_UML;
      ASSOCIATION_END: OPTIONAL PROPERTY_UML;
```

```
      QUALIFIER: OPTIONAL LIST OF PROPERTY_UML; --AN OPTIONAL LIST OF
ORDERED QUALIFIER ATTRIBUTES FOR THE END. IF THE LIST IS EMPTY, THEN THE
ASSOCIATION IS NOT QUALIFIED
DERIVE
      SELF\STRUCTURAL_FEATURE.NAME:STRING := 'PROPERTY_UML';
END_ENTITY;
```

(**
******)

--THIS ENTITY IS THE SPECIFICATION OF SOME OCCURRENCE THAT MAY POTENTIALLY TRIGGER EFFECTS BY AN OBJECT (SPEC. UML, PAGE 442)

```
ENTITY EVENT
      ABSTRACT SUPERTYPE
      SUBTYPE OF (PACKAGEABLE_ELEMENT);
END_ENTITY;
```

(**
******)

--THIS ENTITY REPRESENTS A PORT, FOR DATA AND CONTROL FLOW

```
ENTITY PORT
      SUBTYPE OF (PROPERTY_UML);
      IS_SERVICE: OPTIONAL BOOLEAN;
      IS_BEHAVIOR: OPTIONAL BOOLEAN;
      REQUIRED: OPTIONAL SET[0:?] OF INTERFACE_UML; --DERIVED
      PROVIDED: OPTIONAL SET[0:?] OF INTERFACE_UML; --DERIVED
      REDEFINED_PORT: OPTIONAL PORT;
      --CONSTRAINTS
      --THE REQUIRED INTERFACES OF A PORT MUST BE PROVIDED BY ELEMENTS TO
WHICH THE PORT IS CONNECTED.
      --PORT.AGGREGATION MUST BE COMPOSITE.
      --WHEN A PORT IS DESTROYED, ALL CONNECTORS ATTACHED TO THIS PORT WILL
BE DESTROYED ALSO.
      --A DEFAULTVALUE FOR PORT CANNOT BE SPECIFIED WHEN THE TYPE OF THE
PORT IS AN INTERFACE.
DERIVE
      SELF\PROPERTY_UML.NAME:STRING := 'PORT';
END_ENTITY;
```

(**
******)

--THIS ENTITY RELATES AN EVENT TO A BEHAVIOR THAT MAY AFFECT AN INSTANCE OF THE CLASSIFIER (PAGE 456 OF UML SPEC)

```
ENTITY TRIGGER_UML
      ABSTRACT SUPERTYPE
      SUBTYPE OF (NAMED_ELEMENT);
      EVENT : EVENT;
      PORT: OPTIONAL SET[0:?] OF PORT;
END_ENTITY;
```

--IT IS A SPECIFICATION OF SEND REQUEST INSTANCES COMMUNICATED BETWEEN OBJECTS (UML SPEC, PAGE 449)

```
ENTITY SIGNAL
      SUBTYPE OF (CLASSIFIER);
      OWNED_ATTRIBUTE: OPTIONAL SET[0:?] OF PROPERTY_UML;
END_ENTITY;
```

```
(***************************************************************************************
***********************)
(********************* SYSML PART *************************)
(***************************************************************************************
***********************)
TYPE ENVIRONMENT_TYPE = SELECT(PACKAGE,ENVIRONMENT); END_TYPE;


--AN ACTOR SPECIFIES A ROLE PLAYED BY A USER OR ANY OTHER SYSTEM THAT INTERACTS
WITH THE SUBJECT (UML SPEC, PAGE 588)
ENTITY ACTOR
      SUBTYPE OF (BEHAVIORED_CLASSIFIER);
END_ENTITY;

--THIS ENTITY REPRESENTS THE ENVIRONMENT OF THE SYSTEM (TYPICALLY ACTOR WITH
THEIR INTERRELATIONSHIPS
ENTITY ENVIRONMENT
      SUBTYPE OF (ELEMENT);
      ACTORS: OPTIONAL SET[0:?] OF ACTOR;
      DEPENDENCIES: OPTIONAL SET[0:?] OF DEPENDENCY;
      ASSOCIATIONS: OPTIONAL SET[0:?] OF ASSOCIATION;
END_ENTITY;

--THIS ENTITY REPRESENTS A BLOCK, THE MAIN SYSML UNIT (FROM SYSML SPEC PAGE 46)
ENTITY BLOCK
      SUBTYPE OF (CLASS_UML);
END_ENTITY;

--THIS ENTITY DESCRIBES A STATE MACHINE
ENTITY STATE_MACHINE
      SUBTYPE OF (BEHAVIOR);
      REGION: SET[1:?] OF REGION;
END_ENTITY;

--THIS ENTITY MODELS A SITUATION DURING WHICH SOME (USUALLY IMPLICIT) INVARIANT
CONDITION HOLD (UML SPEC., PAGE 550)
ENTITY STATE
      SUBTYPE OF (NAMESPACE,REDEFINABLE_ELEMENT,VERTEX);
      IS_COMPOSITE : OPTIONAL BOOLEAN;
      IS_ORTHOGONAL: OPTIONAL BOOLEAN;
      IS_SIMPLE: OPTIONAL BOOLEAN;
      IS_SUBMACHINE_STATE: OPTIONAL BOOLEAN;
      CONNECTION: OPTIONAL SET[0:?] OF CONNECTION_POINT_REFERENCE;
      DEFERRABLE_TRIGGER: OPTIONAL SET[0:?] OF TRIGGER_UML; --A LIST OF
TRIGGERS THAT ARE CANDIDATES TO BE RETAINED BY THE STATE MACHINE IF THEY
TRIGGER NO TRANSITIONS OUT OF THE STATE (NOT
      --CONSUMED). A DEFERRED TRIGGER IS RETAINED UNTIL THE STATE MACHINE
REACHES A STATE CONFIGURATION WHERE IT IS NO LONGER
      --DEFERRED.
      DO_ACTIVITY: OPTIONAL BEHAVIOR;
      ENTRY:OPTIONAL BEHAVIOR;
      EXIT: OPTIONAL BEHAVIOR;
      REGION: OPTIONAL SET[0:?] OF REGION;
      SUBMACHINE: OPTIONAL STATE_MACHINE;
      REDEFINITION_CONTEXT: OPTIONAL CLASSIFIER;
END_ENTITY;


--IT SPECIFIES A SPECIAL KIND OF STATE SIGNIFYING THAT THE ENCLOSING REGION IS
COMPLETED. (UML SPEC, PAGE 532)
ENTITY FINAL_STATE
```

```
      SUBTYPE OF (STATE);
END_ENTITY;
```

--IT SPECIFIES THE RECEIPT BY AN OBJECT OF EITHER A CALL OR A SIGNAL. (UML SPEC, PAGE 445)

```
ENTITY MESSAGE_EVENT
      SUBTYPE OF (EVENT);
END_ENTITY;
```

--THIS ENTITY MODELS THE RECEIPT BY AN OBJECT OF A MESSAGE INVOKING A CALL OF AN OPERATION (UML SPEC., PAGE 436)

```
ENTITY CALL_EVENT
      SUBTYPE OF (MESSAGE_EVENT);
      OPERATION: OPERATION_UML;
END_ENTITY;
```

--IT REPRESENTS THE RECEIPT OF AN ASYNCHRONOUS SIGNAL INSTANCE (UML SPEC, PAGE 450)

```
ENTITY SIGNAL_EVENT
      SUBTYPE OF (MESSAGE_EVENT);
      SIGNAL: SIGNAL;
END_ENTITY;
```

--IT REPRESENTS THE DIFFERENT TYPES OF TRANSITION

```
TYPE TRANSITION_KIND=ENUMERATION OF (EXTERNAL,INTERNAL,LOCAL_UML); --LOCAL
EXISTS IN EXPRESS
END_TYPE;
```

--THIS ENTITY REPRESENTS A DIRECTED RELATIONSHIP BETWEEN A SOURCE VERTEX AND A TARGET VERTEX (PAGE 572 OF UML SPEC)

```
ENTITY TRANSITION
      SUBTYPE OF (NAMESPACE,REDEFINABLE_ELEMENT);
      KIND: TRANSITION_KIND;
      TRIGGER: OPTIONAL SET[0:?] OF TRIGGER_UML; --SPECIFIES THE TRIGGERS
THAT MAY FIRE THE TRANSITION, I.E. AN EVENT
      GUARD: OPTIONAL CONSTRAINT_UML;
      EFFECT: OPTIONAL BEHAVIOR; --E.G. TO CALL A METHOD
      SOURCE: VERTEX; --E.G. A STATE
      TARGET: VERTEX; --E.G. A STATE
      REDEFINED_TRANSITION: OPTIONAL TRANSITION;
      REDEFINITION_CONTEXT: OPTIONAL CLASSIFIER;--DERIVED
      CONTAINER: OPTIONAL REGION;
END_ENTITY;
```

--IT GROUPS THE DIFFERENT TYPES OF SYSML DIAGRAMS, TO BE COMPLETED

```
ENTITY DIAGRAM_SYSML
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ENTITY_CLASS);
      --NAME: STRING;
END_ENTITY;
```

--THIS ENTITY REPRESENTS A USAGE (AS PART OF A SUBMACHINE STATE) OF AN ENTRY/EXIT POINT DEFINED IN THE
--STATEMACHINE REFERENCE BY THE SUBMACHINE STATE. (PAGE 529 OF UML SPEC)

```
ENTITY CONNECTION_POINT_REFERENCE
      SUBTYPE OF (VERTEX);
      ENTRY: OPTIONAL SET[0:?] OF PSEUDOSTATE;
      EXIT: OPTIONAL SET[0:?] OF PSEUDOSTATE;
      STATE: OPTIONAL STATE;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACTION THAT ENCOMPASSES DIFFERENT TYPES OF TRANSIENT VERTICES IN THE STATE MACHINE GRAPH. (PAGE 540 OF UML SPEC)

```
ENTITY PSEUDOSTATE
      SUBTYPE OF (VERTEX);
      KIND: OPTIONAL PSEUDOSTATE_KIND;
      STATE_MACHINE: OPTIONAL STATE_MACHINE;
      STATE: OPTIONAL STATE;
END_ENTITY;
```

--THIS ENTITY REPRESENTS A REGION, IT IS AN ORTHOGONAL PART OF EITHER A COMPOSITE STATE OR A STATE MACHINE. IT CONTAINS STATES AND TRANSITIONS

```
ENTITY REGION
      SUBTYPE OF (NAMESPACE, REDEFINABLE_ELEMENT);
      STATE_MACHINE: OPTIONAL STATE_MACHINE;
      STATE: OPTIONAL STATE ;
      TRANSITION: OPTIONAL SET[0:?] OF TRANSITION;
      SUBVERTEX: OPTIONAL SET[0:?] OF VERTEX;
      EXTENDED_REGION: OPTIONAL SET[0:?] OF REGION; --THE REGION OF WHICH
THIS REGION IS AN EXTENSION
      REDEFINITON_CONTEXT: OPTIONAL CLASSIFIER; --DERIVED, REFERENCES THE
CLASSIFIER IN WHICH CONTEXT THIS ELEMENT MAY BE REDEFINED
END_ENTITY;
```

--THIS ENTITY IS AN ENUMERATION OF TYPES OF PSEUDOSTATES
TYPE PSEUDOSTATE_KIND=ENUMERATION OF
(INITIAL,DEEPHISTORY,SHALLOWHISTORY,JOIN,FORK,JUNCTION,CHOICE,ENTRYPOINT,EXITPOINT,TERMINATE);

```
END_TYPE;
```

--THIS ENTITY SPECIFIES THE COORDINATION OF EXECUTIONS OF SUBORDINATE BEHAVIORS, USING A CONTROL AND DATA FLOW MODEL ((PAGE 316 OF UML SPEC))

```
ENTITY ACTIVITY
      SUBTYPE OF (BEHAVIOR);
      IS_READ_ONLY: OPTIONAL BOOLEAN;
      IS_SINGLE_EXECUTION: OPTIONAL BOOLEAN;
      GROUP: OPTIONAL SET[0:?] OF ACTIVITY_GROUP;
      NODE: OPTIONAL SET[0:?] OF ACTIVITY_NODE;
      EDGE: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
      PARTITION: OPTIONAL SET[0:?] OF ACTIVITY_PARTITION;
      STRUCTURED_NODE: OPTIONAL SET[0:?] OF STRUCTURED_ACTIVITY_NODE;
      VARIABLE: OPTIONAL SET[0:?] OF VARIABLE_UML;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACT CLASS FOR DEFINING SETS OF NODES AND EDGES IN AN ACTIVITY (PAGE 348 OF UML SPEC)

```
ENTITY ACTIVITY_GROUP
      ABSTRACT SUPERTYPE
      SUBTYPE OF (ELEMENT);
      IN_ACTIVITY: OPTIONAL ACTIVITY;
      CONTAINED_NODE: OPTIONAL SET[0:?] OF ACTIVITY_NODE;
      CONTAINED_EDGE: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
      SUPER_GROUP: OPTIONAL SET[0:1] OF ACTIVITY_GROUP;
      SUB_GROUP: OPTIONAL SET[0:?] OF ACTIVITY_GROUP;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACT CLASS FOR POINTS IN THE FLOW OF AN ACTIVITY CONNECTED BY EDGES (PAGE 349 OF UML SPEC)

```
ENTITY ACTIVITY_NODE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (NAMED_ELEMENT, REDEFINABLE_ELEMENT);
      ACTIVITY: OPTIONAL ACTIVITY;
```

```
      IN_GROUP: OPTIONAL SET[0:?] OF ACTIVITY_GROUP;
      INCOMING: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
      OUTGOING: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
      REDEFINED_NODE: OPTIONAL SET[0:?] OF ACTIVITY_NODE;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACT CLASS FOR THE CONNECTIONS ALONG WHICH TOKENS FLOW BETWEEN ACTIVITY NODES (PAGE 341 OF UML SPEC)

```
ENTITY ACTIVITY_EDGE
      ABSTRACT SUPERTYPE
      SUBTYPE OF (REDEFINABLE_ELEMENT);
      ACTIVITY: OPTIONAL ACTIVITY;
      IN_GROUP: OPTIONAL SET[0:?] OF ACTIVITY_GROUP;
      REDEFINED_EDGE: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
      SOURCE: ACTIVITY_NODE;
      TARGET: ACTIVITY_NODE;
END_ENTITY;
```

--THIS ENTITY IS A KIND OF ACTIVITY GROUP FOR IDENTIFYING ACTIONS THAT HAVE SOME CHARACTERISTIC IN COMMON (PAGE 356 OF UML SPEC)

```
ENTITY ACTIVITY_PARTITION
      SUBTYPE OF (ACTIVITY_GROUP, NAMED_ELEMENT);
      IS_DIMENSION: BOOLEAN;
      IS_EXTERNAL: BOOLEAN;
      SUPER_PARTITION: OPTIONAL ACTIVITY_PARTITION;
      REPRESENTS: OPTIONAL ELEMENT;
      SUB_PARTITION: OPTIONAL SET[0:?] OF ACTIVITY_GROUP;
      NODE: OPTIONAL SET[0:?] OF ACTIVITY_NODE;
      EDGE: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
END_ENTITY;
```

--THIS ENTITY IS AN EXECUTABLE ACTIVITY NODE THAT MAY HAVE AN EXPANSION INTO SUBORDINATE NODES AS AN ACTIVITYGROUP (PAGE 425 OF UML SPEC))

```
ENTITY STRUCTURED_ACTIVITY_NODE
      SUBTYPE OF  (ACTIVITY_GROUP,  NAMESPACE);
      MUST_ISOLATE: OPTIONAL BOOLEAN;
      ACTIVITY: OPTIONAL ACTIVITY;
      VARIABLE: OPTIONAL SET[0:?] OF VARIABLE_UML;
      NODE: OPTIONAL SET[0:?] OF ACTIVITY_NODE;
      EDGE: OPTIONAL SET[0:?] OF ACTIVITY_EDGE;
END_ENTITY;
```

--VARIABLES ARE ELEMENTS FOR PASSING DATA BETWEEN ACTIONS INDIRECTLY (PAGE 430 OF UML SPEC)

```
ENTITY VARIABLE_UML
      SUBTYPE OF (MULTIPLICITY_ELEMENT,TYPED_ELEMENT);
      SCOPE: OPTIONAL STRUCTURED_ACTIVITY_NODE;
      ACTIVITY: OPTIONAL ACTIVITY;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACTION OF A NODE IN A STATE MACHINE GRAPH. IN GENERAL, IT CAN BE THE SOURCE OR DESTINATION OF ANY NUMBEROF TRANSITIONS. (PAGE 582 OF UML SPEC)

```
ENTITY VERTEX
      SUBTYPE OF (NAMED_ELEMENT);
      OUTGOING: OPTIONAL SET[0:?] OF TRANSITION; --DERIVED
      INCOMING: OPTIONAL SET[0:?] OF TRANSITION; --DERIVED
      CONTAINER: OPTIONAL REGION;
END_ENTITY;
```

--THIS ENTITY IS AN ABSTRACTION OF A NODE IN A STATE MACHINE GRAPH. IN GENERAL, IT CAN BE THE SOURCE OR DESTINATION OF ANY NUMBEROF TRANSITIONS. (PAGE 582 OF UML SPEC)

```
ENTITY REQUIREMENT_SYSML
      SUBTYPE OF (NAMED_ELEMENT);
END_ENTITY;
```

--THIS ENTITY IS A SINGLE FLOW ELEMENT TO/FROM A BLOCK (PAGE 65 SYSML SPEC)

```
ENTITY FLOW_PROPERTY
      SUBTYPE OF (PROPERTY_UML);
      DIRECTION: OPTIONAL PARAMETER_DIRECTION_KIND;
END_ENTITY;

END_SCHEMA;
```

Annex C

This annex presents the meta-models and instances of the SIS and CIS messages case and of the different case studies described in Chapter VI.

1. SIS and CIS message models

Knowledge model

```
-- THIS SCHEMA CONTAINS THE ENTITIES MAKING EXPLICIT THE KNOWLEDGE RELATED TO
THE MESSAGES CASE
SCHEMA KMODEL_SCHEMA;

REFERENCE FROM TOP_SCHEMA;
REFERENCE FROM ANNOTATION_SCHEMA;


--This entity represents the origin of a message
ENTITY ORIGIN_OF_MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
END_ENTITY;

--This entity represents the destination of a message
ENTITY DESTINATION_OF_MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
END_ENTITY;

--This entity represents the copy destination of a message
ENTITY COPY_OF_MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
END_ENTITY;

--This entity represents the hidden copy destination of the message
ENTITY SECRET_COPY_OF_MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
END_ENTITY;


--This entity represents a Message
ENTITY MESSAGE
SUBTYPE OF (KNOWLEDGE_CLASS);
      PERSON_FROM: SET [0:?] OF ORIGIN_OF_MESSAGE;
      PERSON_TO: SET [0:?] OF DESTINATION_OF_MESSAGE;
      PERSON_CC: SET [0:?] OF COPY_OF_MESSAGE;
      PERSON_CCO: SET [0:?] OF SECRET_COPY_OF_MESSAGE;
      MESSAGE_PARAMETER: SET [1:?] OF STRING;
END_ENTITY;
```

--This entity represents the different type of communication protocols

```
ENTITY COMMUNICATION_PROTOCOL
SUBTYPE OF (KNOWLEDGE_CLASS);
      PROTOCOL_NAME: STRING;
      IS_SECURE: BOOLEAN;
END_ENTITY;

END_SCHEMA;
```

CORE SIS model instances

```
#1=T_DATE(2, 2010, 11, 13, 0, 0);
#2=CORE_MODELING_LANGUAGE('7');
#3=MODEL_CORE($, $, $, 'SIS', #1, #1, #2, (#13,#26,#39,#50,#247), $);
#13=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0001', $, $, $, $, $, $,
$, $, (), (), (), ());
#26=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0002', $, $, $, $, $, $,
$, $, (), (), (), ());
#39=ITEM_CORE(*, $, $, #1, 'dsz', $, #1, 'ITEM0003', $, $, $, $, $, $,
$, $, (), (), (), ());
#50=LINK(*, $, $, #1, 'dsz', $, #1, 'ExtComm', $, $, $, $, $, (),
(#39,#26,#13));
#247=INTERFACE_CORE(*, $, $, #1, 'dsz', $, #1, 'extcomm', (#50), ());
```

SysML CIS model instances

```
#70=T_DATE(16, 09, 2007, $, $, $);
#71=SYSML_MODELING_LANGUAGE('1.1');
#72=SYSML_MODEL($, $, $, 'CIS', #70, $, #71, $, (#78,#93,#100));
#78=INTERFACE_UML(*, $, $, (), $, 'External Communication', $,
.PRIVATE., $, $, (), $, $, (), (), $, (), (), (), (), (), (), $, $,
(), (#93), (), ());
#93=OPERATION_UML(*, $, $, (), $, $, 'receiveMessage', .PRIVATE., $,
$, $, (), (), .F., (), (#100), .F., (), (), .F., .F., .F., $, $, $, $,
(), (), (), $, #78);
#100=PARAMETER_UML(*, $, $, (), $, $, 'message', .PRIVATE., $, $, .T.,
.T., $, $, $, $, $, $, $, .IN_UML., #93, $);
```

Knowledge model instances

```
#115=URI('http://www.eads.net/thesedsz/knowledge/protocol/ex25''');
#116=URI('http://www.eads.net/thesedsz/knowledge/protocol/ethernet');
#117=URI('http://www.eads.net/thesedsz/knowledge/protocol/x25''');
#60=URI('http://www.eads.net/thesedsz/knowledge/message/maintenance');
#6=URI('http://www.eads.net/thesedsz/knowledge/persons/0001');
#7=URI('http://www.eads.net/thesedsz/knowledge/persons/0002');
#8=URI('http://www.eads.net/thesedsz/knowledge/persons/0003');
#9=URI('http://www.eads.net/thesedsz/knowledge/persons/0004');
#51=ORIGIN_OF_MESSAGE(*, *, $, #6, '0001');
#52=DESTINATION_OF_MESSAGE(*, *, $, #7 , '0002');
#53=COPY_OF_MESSAGE(*, *, $, #8, '0003');
#54=SECRET_COPY_OF_MESSAGE(*, *, $, #9, '0004');
#57=MESSAGE(*, *, $, #60, (#51), (#52), (#53), (#54), ('Test'));
#112=COMMUNICATION_PROTOCOL(*, *, $, #115, 'EX25', .T.);
#113=COMMUNICATION_PROTOCOL(*, *, $, #116, 'ETHERNET', .T.);
#114=COMMUNICATION_PROTOCOL(*, *, $, #117, 'X25', .F.);
```

Annotation instances

```
#105=ANNOTATION_CLASS('represents', (#60), (#100));
#59=ANNOTATION_CLASS('represents', (#60), (#13,#26,#39));
#118=ANNOTATION_CLASS('protocol', (#115), (#50));
#119=ANNOTATION_CLASS('protocol', (#115), (#78));
```

Expression model instances

```
#157=ENTITY_DOMAIN((#50));
#158=VARIABLE_DOMAIN(#157, #362);
#159=ALL_EXPRESSION(*, (), (#158), #352);
#320=ENTITY_VARIABLE(#100, 'p');
#321=ENTITY_VARIABLE(#93, 'o');
#322=ENTITY_ARRAY_PATH_VARIABLE($, 'o.owned_parameter', #321,
'owned_parameter', .F.);
#323=ENTITY_PATH_VARIABLE($, 'p\\{represents\\}', #320, 'represents',
.T.);
#324=ENTITY_LITERAL(#57);
#325=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#323,#324));
#326=BELONG_BOOLEAN_EXPRESSION(*, (#320,#322));
#327=AND_EXPRESSION(*, (#325,#326));
#329=ENTITY_DOMAIN((#100));
#330=VARIABLE_DOMAIN(#329, #320);
#331=ENTITY_VARIABLE(#78, 'X');
#332=ENTITY_ARRAY_PATH_VARIABLE($, 'X.owned_operation', #331,
'owned_operation', .F.);
```

```
#333=BELONG_BOOLEAN_EXPRESSION(*, (#321,#332));
#334=AND_EXPRESSION(*, (#333,#338));
#335=EXISTS_EXPRESSION(*, (#321), (#337), #334);
#336=ENTITY_DOMAIN((#93));
#337=VARIABLE_DOMAIN(#336, #321);
#338=EXISTS_EXPRESSION(*, (#321), (#330), #327);
#339=ENTITY_VARIABLE(#112, 'cp');
#340=BOOLEAN_ARRAY_PATH_VARIABLE($, 'cp.securised', #339, 'is_secure',
.F.);
#341=ENTITY_PATH_VARIABLE($, 'l\\{protocol\\}', #362, 'protocol',
.T.);
#342=ENTITY_PATH_VARIABLE($, 'X\\{protocol\\}', #331, 'protocol',
.T.);
#343=BOOLEAN_LITERAL(.T.);
#344=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#340,#343));
#345=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#341,#339));
#346=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#342,#339));
#347=AND_EXPRESSION(*, (#344,#345,#346));
#348=EXISTS_EXPRESSION(*, (), (#350), #347);
#349=ENTITY_DOMAIN((#114,#113,#112));
#350=VARIABLE_DOMAIN(#349, #339);
#351=NOT_EXPRESSION(*, #356);
#352=OR_EXPRESSION(*, (#351,#348));
#353=ENTITY_VARIABLE(#247, 'Y');
#354=ENTITY_ARRAY_PATH_VARIABLE($, 'Y.comprised_of', #353,
'comprised_of', .F.);
#355=BELONG_BOOLEAN_EXPRESSION(*, (#362,#354));
#356=AND_EXPRESSION(*, (#355,#369,#335));
#361=ENTITY_VARIABLE(#26, 'i');
#362=ENTITY_VARIABLE(#50, 'l');
#363=ENTITY_ARRAY_PATH_VARIABLE($, 'l.transfer', #362, 'transfers',
.F.);
#364=ENTITY_PATH_VARIABLE($, 'i\\{represents\\}', #361, 'represents',
.T.);
#365=ENTITY_LITERAL(#57);
#366=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#364,#365));
#367=BELONG_BOOLEAN_EXPRESSION(*, (#361,#363));
#368=AND_EXPRESSION(*, (#367,#366));
#369=EXISTS_EXPRESSION(*, (#362), (#371), #368);
#370=ENTITY_DOMAIN((#39,#26,#13));
#371=VARIABLE_DOMAIN(#370, #361);
```

2. Water and Waste System model

Knowledge model

```
-- THIS SCHEMA CONTAINS THE ENTITIES MAKING EXPLICIT THE KNOWLEDGE RELATED TO
THE WATER AND WASTE SYSTEM CASE STUDY
SCHEMA KMODEL_SCHEMA;

REFERENCE FROM TOP_SCHEMA;
REFERENCE FROM ANNOTATION_SCHEMA;

(********************** WWS Example **************************)
--This entity represents an aircraft
ENTITY AIRCRAFT_PROGRAM
SUBTYPE OF (KNOWLEDGE_CLASS);
      NUMBER_OF_DECKS: NUMBER;
      NUMBER_OF_ENGINES: NUMBER;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'AIRCRAFT_PROGRAM';
END_ENTITY;

--This entity represents an ATA chapter
ENTITY ATA
SUBTYPE OF (KNOWLEDGE_CLASS);
      CHAPTER_NUMBER: STRING;
      DESCRIBES: ATA_SYSTEM;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ATA';
END_ENTITY;

--This entity represents an ATA System
ENTITY ATA_SYSTEM
ABSTRACT SUPERTYPE
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ATA_SYSTEM';
END_ENTITY;

--This entity represents WWS
ENTITY WWS
SUBTYPE OF (ATA_SYSTEM);
      HAS_LAVATORY: SET[0:?] OF LAVATORY;
      HAS_GALLEY: SET[0:?] OF GALLEY;
      HAS_WASTE_TANK: WASTE_TANK;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'WWS';
END_ENTITY;

--This entity represents a lavatory
ENTITY LAVATORY
SUBTYPE OF (KNOWLEDGE_CLASS);
      HAS_TOILET: TOILET;
      DECK_NUMBER: NUMBER;
      HAS_WASH_BASIN: WASH_BASIN;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'LAVATORY';
END_ENTITY;
```

```
--This entity represents a toilet
ENTITY TOILET
SUBTYPE OF (KNOWLEDGE_CLASS);
      HAS_FLUSH_VALVE: FLUSH_VALVE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'TOILET';
END_ENTITY;

--This entity represents a flush valve
ENTITY FLUSH_VALVE
SUBTYPE OF (KNOWLEDGE_CLASS);
      CONNECTS_TO_FCU: OPTIONAL FCU;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FLUSH_VALVE';
END_ENTITY;

--This entity represents a flush control unit
ENTITY FCU
SUBTYPE OF (KNOWLEDGE_CLASS);
      CONNECTS: OPTIONAL WASTE_LINE;
      MANAGES: OPTIONAL SET [0:?] OF FLUSH_VALVE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FCU';
END_ENTITY;

TYPE CAPACITY_UNIT = ENUMERATION OF (Liter,m3);
END_TYPE;

ENTITY CAPACITY_CLASS
      SUBTYPE OF (ENTITY_CLASS);
      THE_VALUE: NUMBER;
      UNIT: CAPACITY_UNIT;
DERIVE
      SELF\ENTITY_CLASS.NAME:STRING := 'CAPACITY';
END_ENTITY;

--This entity represents a waste line
ENTITY WASTE_LINE
SUBTYPE OF (KNOWLEDGE_CLASS);
      CIRCUIT: WASTE_TANK;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'WASTE_LINE';
END_ENTITY;

--This entity represents a waste tank
ENTITY WASTE_TANK
SUBTYPE OF (KNOWLEDGE_CLASS);
      CAPACITY: OPTIONAL CAPACITY_CLASS;
      EJECTION: OPTIONAL DRAIN_VALVE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'WASTE_TANK';
END_ENTITY;

--This entity represents a galley
ENTITY GALLEY
SUBTYPE OF (KNOWLEDGE_CLASS);
      FLUSH: WASTE_LINE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'GALLEY';
END_ENTITY;
```

```
--This entity represents a wash basin
ENTITY WASH_BASIN
SUBTYPE OF (KNOWLEDGE_CLASS);
      FLUSH: WASTE_LINE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'WASH_BASIN';
END_ENTITY;

--This entity represents a drain valve
ENTITY DRAIN_VALVE
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'DRAIN_VALVE';
END_ENTITY;

END_SCHEMA;
```

SysML model instances

```
#20=T_DATE(16, 9, 2009, $, $, $);
#21=SYSML_MODELING_LANGUAGE('1.1');
#22=SYSML_MODEL($, $, $, $, #20, $, #21, $,
(#28,#31,#33,#35,#41,#42,#45,#46,#47,#48,#49,#50,#56,#59,#62,#65,#67,#
70,#73,#75,#78,#81,#83,#86,#89,#91,#92,#93,#94,#95,#96,#98,#100,#101,#
102,#103,#104));
#28=PORT(*, $, $, (), $, 'waste in', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#31=BLOCK(*, $, $, (), $, $, $, .PRIVATE., $, $, (), $, $, (), (), $,
(), (), (), (), (), (), $, $, (), (#28), (), (), (), $, (), ());
#33=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #28, $, #35);
#35=PROPERTY_UML(*, $, $, (), $, 'wasteline', $, .PRIVATE., $, $, $,
(), (), .F., (), .F., .F., $, $, $, $, #31, .F., $, .COMPOSITE., $,
.F., .F., .F., (), $, $, $, $, (), (), $, (), $, ());
#41=CONNECTOR(*, $, $, (), $, 'waste water', $, .PRIVATE., $, $, $,
(), (), .F., (), (#33,#56), $, ());
#42=BLOCK(*, $, $, (), $, 'WWS', $, .PRIVATE., $, $, (), $, $, (), (),
$, (), (), (), (), (), (), $, $, (), (#35), (), (#41), (), $, (), ());
#45=PROPERTY_UML(*, $, $, (), $, 'T1', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, #50, .F., $, .COMPOSITE., $, .F., .F.,
.F., (), $, $, $, $, (), (), $, (), $, ());
#46=PROPERTY_UML(*, $, $, (), $, 'T2', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, #50, .F., $, .COMPOSITE., $, .F., .F.,
.F., (), $, $, $, $, (), (), $, (), $, ());
#47=PROPERTY_UML(*, $, $, (), $, 'T3', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, #50, .F., $, .COMPOSITE., $, .F., .F.,
.F., (), $, $, $, $, (), (), $, (), $, ());
#48=PROPERTY_UML(*, $, $, (), $, 'T4', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, #50, .F., $, .COMPOSITE., $, .F., .F.,
.F., (), $, $, $, $, (), (), $, (), $, ());
#49=PROPERTY_UML(*, $, $, (), $, 'fcu', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, #50, .F., $, .COMPOSITE., $, .F., .F.,
.F., (), $, $, $, $, (), (), $, (), $, ());
#50=BLOCK(*, $, $, (), $, 'Toilet Unit', $, .PRIVATE., $, $, (), $, $,
(), (), $, (), (), (), (), (), (), $, $, (), (#92,#91,#94,#93,#96,
     #95), (), (), (), $, (), ());
```

```
#56=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #104, $, #49);
#59=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #93, $, #45);
#62=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #100, $, #49);
#65=CONNECTOR(*, $, $, (), $, 'flush 1', $, .PRIVATE., $, $, $, (),
(), .F., (), (#59,#62), $, ());
#67=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #94, $, #46);
#70=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #101, $, #49);
#73=CONNECTOR(*, $, $, (), $, 'flush 2', $, .PRIVATE., $, $, $, (),
(), .F., (), (#70,#67), $, ());
#75=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #95, $, #47);
#78=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #102, $, #49);
#81=CONNECTOR(*, $, $, (), $, 'flush 3', $, .PRIVATE., $, $, $, (),
(), .F., (), (#78,#75), $, ());
#83=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #96, $, #48);
#86=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #103, $, #49);
#89=CONNECTOR(*, $, $, (), $, 'flush 4', $, .PRIVATE., $, $, $, (),
(), .F., (), (#86,#83), $, ());
#91=PROPERTY_UML(*, $, $, (), $, 'capacity', $, .PRIVATE., $, $, $,
(), (), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, ());
#92=PROPERTY_UML(*, $, $, (), $, 'deck', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, ());
#93=PORT(*, $, $, (), $, 'Flush Valve 1', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#94=PORT(*, $, $, (), $, 'Flush Valve 2', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#95=PORT(*, $, $, (), $, 'Flush Valve 3', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#96=PORT(*, $, $, (), $, 'Flush Valve 4', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#98=BLOCK(*, $, $, (), $, 'Flush Control Unit', $, .PRIVATE., $, $,
(), $, $, (), (), $, (), (), (), (), (), (), $, $, (),
(#104,#102,#101,#100,#103), (), (), (), $, (), ());
#100=PORT(*, $, $, (), $, 'In1', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F., .F., .F.,
(), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#101=PORT(*, $, $, (), $, 'In2', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F., .F., .F.,
(), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#102=PORT(*, $, $, (), $, 'In3', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F., .F., .F.,
(), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#103=PORT(*, $, $, (), $, 'In4', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, .COMPOSITE., $, .F., .F., .F.,
(), $, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
```

Knowledge model instances

```
#105=LAVATORY(*, $, $, #108, #109, 1., #265);
#108=URI('http://www.eads.net/systems/wws/lavatory1');
#109=TOILET(*, $, $, #110, #111);
#110=URI('http://www.eads.net/systems/wws/toilet1');
#111=FLUSH_VALVE(*, $, $, #117, #119);
#113=URI('http://www.eads.net/systems/wws/a330');
#114=URI('http://www.eads.net/systems/wws/a380');
#115=URI('http://www.eads.net/systems/wws/wastetank1');
#116=URI('http://www.eads.net/systems/wws/fcu');
#117=URI('http://www.eads.net/systems/wws/flushvalve1');
#119=FCU(*, $, $, #116, #264, (#111,#146,#136,#156));
#121=WASTE_TANK(*, $, $, #115, $, $);
#123=CAPACITY_CLASS(*, $, $, 50., .LITER.);
#124=AIRCRAFT_PROGRAM(*, $, $, #114, 2., 4.);
#127=URI('http://www.eads.net/systems/wws/lavatory2');
#128=URI('http://www.eads.net/systems/wws/toilet2');
#129=URI('http://www.eads.net/systems/wws/flushvalve2');
#131=LAVATORY(*, $, $, #127, #134, 1., #266);
#134=TOILET(*, $, $, #128, #136);
#136=FLUSH_VALVE(*, $, $, #129, #119);
#138=URI('http://www.eads.net/systems/wws/lavatory3');
#139=URI('http://www.eads.net/systems/wws/toilet3');
#140=URI('http://www.eads.net/systems/wws/flushvalve3');
#141=LAVATORY(*, $, $, #138, #144, 2., #267);
#144=TOILET(*, $, $, #139, #146);
#146=FLUSH_VALVE(*, $, $, #140, #119);
#148=URI('http://www.eads.net/systems/wws/lavatory4');
#149=URI('http://www.eads.net/systems/wws/toilet4');
#150=URI('http://www.eads.net/systems/wws/flushvalve4');
#151=LAVATORY(*, $, $, #148, #154, 2., #268);
#154=TOILET(*, $, $, #149, #156);
#156=FLUSH_VALVE(*, $, $, #150, #119);
#158=AIRCRAFT_PROGRAM(*, $, $, #113, 1., 4.);
#259=URI('http://www.eads.net/systems/wws/wasteline');
#260=URI('http://www.eads.net/systems/wws/washbasin1');
#261=URI('http://www.eads.net/systems/wws/washbasin2');
#262=URI('http://www.eads.net/systems/wws/washbasin3');
#263=URI('http://www.eads.net/systems/wws/washbasin4');
#264=WASTE_LINE(*, $, $, #259, #121);
#265=WASH_BASIN(*, $, $, #260, #264);
#266=WASH_BASIN(*, $, $, #261, #264);
#267=WASH_BASIN(*, $, $, #262, #264);
#268=WASH_BASIN(*, $, $, #263, #264);
```

Annotation instances

```
#161=ANNOTATION_CLASS('is', (#110),(#45));
#162=ANNOTATION_CLASS('belongs', (#114),(#42));
#163=ANNOTATION_CLASS('is', (#128),(#46));
#164=ANNOTATION_CLASS('is', (#149),(#48));
#165=ANNOTATION_CLASS('is', (#139),(#47));
#166=ANNOTATION_CLASS('is', (#116),(#98));
```

Expression model instances

```
/********** More than 1 toilet ****************/
#167=ENTITY_VARIABLE(#48, 'p');
#168=BELONG_BOOLEAN_EXPRESSION(*, (#172,#169));
    /*     not for commercial use   */
#169=ENTITY_ARRAY_LITERAL((#109,#154,#144,#134));
#170=ENTITY_DOMAIN((#47,#46,#45,#48));
#171=VARIABLE_DOMAIN(#170, #167);
#172=ENTITY_PATH_VARIABLE($, 'p\\{is\\}', #167, 'is', .T.);
#173=ALL_SUM_EXPRESSION(*, (#167), (#171), #168);
#174=INT_LITERAL(1.);
#175=COMPARISON_GREATER(*, (#173,#174));

/********** All toilets connected to a Flush Control Unit? (FCU)
****************/
#176=ALL_EXPRESSION(*, (#167), (#171), #177);
#177=OR_EXPRESSION(*, (#178,#179));
#178=NOT_EXPRESSION(*, #168);
#179=EXISTS_EXPRESSION(*, (#180), (#182), #184);
#180=ENTITY_VARIABLE(*, 'c');
#181=ENTITY_DOMAIN((#41,#65,#73,#81,#89));
#182=VARIABLE_DOMAIN(#181, #180);
#184=EXISTS_EXPRESSION(*, (#185), (#187), #188);
#185=ENTITY_VARIABLE(*, 'e1');
#186=ENTITY_DOMAIN((#33,#56,#59,#62,#67,#70,#75,#78,#83,#86));
#187=VARIABLE_DOMAIN(#186, #185);
#188=AND_EXPRESSION(*, (#189,#191,#193));
#189=BELONG_BOOLEAN_EXPRESSION(*, (#185,#190));
#190=ENTITY_ARRAY_PATH_VARIABLE($, 'c.end_connector', #180,
'END_CONNECTOR', .F.);
#191=BELONG_BOOLEAN_EXPRESSION(*, (#167,#192));
#192=ENTITY_ARRAY_PATH_VARIABLE($, 'e1.PART_WITH_PORT', #185,
'PART_WITH_PORT', .F.);
#193=EXISTS_EXPRESSION(*, (#194), (#195), #196);
#194=ENTITY_VARIABLE(*, 'e2');
#195=VARIABLE_DOMAIN(#186, #194);
#196=AND_EXPRESSION(*, (#197,#198));
#197=BELONG_BOOLEAN_EXPRESSION(*, (#194,#190));
#198=EXISTS_EXPRESSION(*, (#199), (#200), #202);
#199=ENTITY_VARIABLE(*, 'b');
#200=VARIABLE_DOMAIN(#201, #199);
#201=ENTITY_DOMAIN((#31,#42,#50,#98));
#202=AND_EXPRESSION(*, (#203,#206));
#203=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#204,#205));
#204=ENTITY_PATH_VARIABLE($, 'b\\{is\\}', #199, 'is', .T.);
#205=ENTITY_LITERAL(#119);
#206=BELONG_BOOLEAN_EXPRESSION(*, (#207,#208));
#207=ENTITY_PATH_VARIABLE($, 'e2.ROLE', #194, 'ROLE', .F.);
#208=ENTITY_ARRAY_PATH_VARIABLE($, 'b.OWNED_ATTRIBUTE', #199,
'OWNED_ATTRIBUTE', .F.);
```

3. Hydraulic and Engine systems models

Knowledge model

-- THIS SCHEMA CONTAINS THE ENTITIES MAKING EXPLICIT THE KNOWLEDGE RELATED -- TO THE HYDRAULIC AND ENGINE SYSTEMS CASE STUDIES

```
SCHEMA KMODELATA_SCHEMA;

REFERENCE FROM TOP_SCHEMA;
REFERENCE FROM ANNOTATION_SCHEMA;
```

(********************* Hydraulic Example *************************)

--This entity represents an aircraft

```
ENTITY AIRCRAFT_PROGRAM
SUBTYPE OF (KNOWLEDGE_CLASS);
      NUMBER_OF_DECKS: NUMBER;
      NUMBER_OF_ENGINES: NUMBER;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'AIRCRAFT_PROGRAM';
END_ENTITY;
```

--This entity represents an ATA chapter

```
ENTITY ATA
SUBTYPE OF (KNOWLEDGE_CLASS);
      CHAPTER_NUMBER: STRING;
      DESCRIBES: ATA_SYSTEM;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ATA';
END_ENTITY;
```

--This entity represents an ATA System

```
ENTITY ATA_SYSTEM
ABSTRACT SUPERTYPE
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ATA_SYSTEM';
END_ENTITY;
```

--This entity represents HYDRAULIC_SYSTEM

```
ENTITY HYDRAULIC_SYSTEM
SUBTYPE OF (ATA_SYSTEM);
      ITS_PUMPS: SET[0:?] OF PUMP;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'HYDRAULIC_SYSTEM';
END_ENTITY;
```

--This entity represents an hydraulic flow

```
ENTITY PUMP
SUBTYPE OF (KNOWLEDGE_CLASS);
      GENERATES: FLOW;
      ITS_VALVES: SET[0:?] OF VALVE;
      IDENTIFIER: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'PUMP';
END_ENTITY;
```

```
--This entity represents an hydraulic flow
ENTITY VALVE
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'VALVE';
END_ENTITY;

--This entity represents an hydraulic flow
ENTITY VALVE_IN
SUBTYPE OF (VALVE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'VALVE_IN';
END_ENTITY;

--This entity represents an hydraulic flow
ENTITY VALVE_OUT
SUBTYPE OF (VALVE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'VALVE_OUT';
END_ENTITY;

--This entity represents an hydraulic flow
ENTITY EDP
SUBTYPE OF (PUMP);
      LOCATION: OPTIONAL ENGINE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'EDP';
END_ENTITY;

--This entity represents an hydraulic flow
ENTITY FLOW
SUBTYPE OF (KNOWLEDGE_CLASS);
      THE_VALUE: NUMBER;
      UNIT: FLOW_UNIT;
      PRESSURE_UNDER: PRESSURE;
      FREQUENCY_AT: FREQUENCY;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'HYDRAULIC_FLOW';
END_ENTITY;


--This entity represents a flow unit
ENTITY FLOW_UNIT
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FLOW_UNIT';
END_ENTITY;

--This entity represents an hydraulic pressure
ENTITY PRESSURE
SUBTYPE OF (KNOWLEDGE_CLASS);
      THE_VALUE: NUMBER;
      UNIT: PRESSURE_UNIT;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'PRESSURE';
END_ENTITY;

--This entity represents a pressure unit
ENTITY PRESSURE_UNIT
```

```
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'PRESSURE_UNIT';
END_ENTITY;
```

--This entity represents an hydraulic frequency

```
ENTITY FREQUENCY
SUBTYPE OF (KNOWLEDGE_CLASS);
      THE_VALUE: NUMBER;
      UNIT: FREQUENCY_UNIT;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FREQUENCY';
END_ENTITY;
```

--This entity represents a frequency unit

```
ENTITY FREQUENCY_UNIT
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FREQUENCY_UNIT';
END_ENTITY;
```

--This entity represents an engine

```
ENTITY ENGINE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ENGINE_NUMBER: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ENGINE';
END_ENTITY;

END_SCHEMA;
```

SysML Hydraulic model instances

```
#190=T_DATE(15, 09, 2008, $, $, $);
#191=SYSML_MODELING_LANGUAGE('1.1');
#192=SYSML_MODEL($, $, $, 'Hydraulic', #190, $, #191, $,
(#142,#145,#146,#147,#148,#163,#164,#167,#171,#172,#174,#175,#177,#181
,#182));
#142=BLOCK(*, $, $, (), $, 'Distribution system', $, .PRIVATE., $, $,
(), $, $, (), (), $, (), (), (), (), (), (), $, $, (),
(#145,#148,#146,#147), (), (), (), $, (), ());
#145=PORT(*, $, $, (), $, 'out_pump1', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#146=PORT(*, $, $, (), $, 'in_pump1', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#147=PORT(*, $, $, (), $, 'out_pump2', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#148=PORT(*, $, $, (), $, 'in_pump2', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
```

```
#163=BLOCK(*, $, $, (), $, 'Accumulator', $, .PRIVATE., $, $, (), $,
$, (), (), $, (), (), (), (), (), (), $, $, (), (), (), (), (), $, (),
());
#164=ASSOCIATION(*, $, $, (), $, 'provides', $, .PRIVATE., $, $, (),
$, $, (), (), $, (), (), (), (), (), (), $, $, $, .F., (#177,#167),
(), (#177), $);
#167=PROPERTY_UML(*, $, $, (), $, 'accumulator', $, .PRIVATE., $, $,
$, (), (), .F., (), .F., .F., $, $, #172, #171, #163, .F., $,
.COMPOSITE., $, .F., .F., .F., (), #164, $, $, $, (), (), $, (), $,
());
#171=VALUE_SPECIFICATION(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
#172=VALUE_SPECIFICATION(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
#174=TYPED_ELEMENT(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
#175=TYPED_ELEMENT(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
#177=PROPERTY_UML(*, $, $, (), $, 'distribution system', $, .PRIVATE.,
$, $, $, (), (), .F., (), .F., .F., $, $, #172, #171, $, .F., $,
.COMPOSITE., $, .F., .F., .F., (), #164, $, $, $, (), (), $, (), $,
());
#181=VALUE_SPECIFICATION(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
#182=TYPED_ELEMENT(*, $, $, (), $, $, $, .PRIVATE., $, $, $);
```

SysML Engine model instances

```
#1=T_DATE(16, 04, 2007, $, $, $);
#2=SYSML_MODELING_LANGUAGE('1.1');
#3=SYSML_MODEL($, $, $, 'Engine', #1, $, #2, $,
(#50,#51,#52,#53,#54,#56,#58,#60,#62,#64,#66,#68,#70,#72,#74,#76,#78,#
80,#82,#84,#86,#88,#90,#92,#94,#96,#99,#102,#104,#107,#110,#112,#115,#
118,#120,#123,#126,#128,#131,#134,#136,#139,#149,#152,#153,#154,#155,#
156,#157,#158,#161,#162));
#50=PORT(*, $, $, (), $, 'in_pumpA', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#51=PORT(*, $, $, (), $, 'out_pumpA', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#52=PORT(*, $, $, (), $, 'out_pumpB', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#53=PORT(*, $, $, (), $, 'in_pumpB', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#54=PROPERTY_UML(*, $, $, (), $, 'edp1', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, #149, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, ());
#56=PROPERTY_UML(*, $, $, (), $, 'edp2', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, #149, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, ());
#58=PROPERTY_UML(*, $, $, (), $, 'engine1', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, #158, .F., $, .COMPOSITE., $, .F.,
.F., .F., (), $, $, $, $, (), (), $, (), $, ());
#60=PORT(*, $, $, (), $, 'Port1', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
```

```
#62=PORT(*, $, $, (), $, 'in_edp1', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#64=PORT(*, $, $, (), $, 'out_edp1', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#66=PORT(*, $, $, (), $, 'in_edp2', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#68=PORT(*, $, $, (), $, 'out_edp2', $, .PRIVATE., $, $, $, (), (),
.F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $,
$, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#70=CONNECTOR(*, $, $, (), $, 'in_pressureB', $, .PRIVATE., $, $, $,
(), (), .F., (), (#74,#72), $, ());
#72=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #53, $, $);
#74=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #145, $, $);
#76=CONNECTOR(*, $, $, (), $, 'out_pressureB', $, .PRIVATE., $, $, $,
(), (), .F., (), (#80,#78), $, ());
#78=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #52, $, $);
#80=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #146, $, $);
#82=CONNECTOR(*, $, $, (), $, 'out_pressureA', $, .PRIVATE., $, $, $,
(), (), .F., (), (#86,#84), $, ());
#84=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #51, $, $);
#86=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #148, $, $);
#88=CONNECTOR(*, $, $, (), $, 'in_pressureA', $, .PRIVATE., $, $, $,
(), (), .F., (), (#92,#90), $, ());
#90=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #50, $, $);
#92=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #147, $, $);
#94=CONNECTOR(*, $, $, (), $, 'power', $, .PRIVATE., $, $, $, (), (),
.F., (), (#96,#99), $, ());
#96=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #152, $, #54);
#99=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #161, $, #58);
#102=CONNECTOR(*, $, $, (), $, 'power', $, .PRIVATE., $, $, $, (), (),
.F., (), (#107,#104), $, ());
#104=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #153, $, #56);
#107=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #162, $, #58);
#110=CONNECTOR(*, $, $, (), $, 'pressure_in', $, .PRIVATE., $, $, $,
(), (), .F., (), (#115,#112), $, ());
#112=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #62, $, $);
#115=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #154, $, #54);
```

```
#118=CONNECTOR(*, $, $, (), $, 'pressure_out', $, .PRIVATE., $, $, $,
(), (), .F., (), (#120,#123), $, ());
#120=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #64, $, $);
#123=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #155, $, #54);
#126=CONNECTOR(*, $, $, (), $, 'pressure_in', $, .PRIVATE., $, $, $,
(), (), .F., (), (#128,#131), $, ());
#128=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #66, $, $);
#131=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #161, $, #58);
#134=CONNECTOR(*, $, $, (), $, 'pressure_out', $, .PRIVATE., $, $, $,
(), (), .F., (), (#139,#136), $, ());
#136=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #68, $, $);
#139=CONNECTOR_END(*, $, $, (), $, $, $, .PRIVATE., $, $, .F., .F., $,
$, $, $, #157, $, #54);
#149=BLOCK(*, $, $, (), $, 'EDP', $, .PRIVATE., $, $, (), $, $, (),
(), $, (), (), (), (), (), (), $, $, (), (#152,#153,#155,#154,#157,
     #156), (), (), (), $, (), ());
#152=PORT(*, $, $, (), $, 'engine_power', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (),
$, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#153=PORT(*, $, $, (), $, 'engine_power', $, .PRIVATE., $, $, $, (),
(), .F., (), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (),
$, $, $, $, (), (), $, (), $, (), .F., .F., (), (), $);
#154=PORT(*, $, $, (), $, 'in', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
#155=PORT(*, $, $, (), $, 'out', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
#156=PORT(*, $, $, (), $, 'in', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
#157=PORT(*, $, $, (), $, 'out', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
#158=BLOCK(*, $, $, (), $, 'Engine', $, .PRIVATE., $, $, (), $, $, (),
(), $, (), (), (), (), (), (), $, $, (), (#161,#162), (), (), (), $,
(), ());
#161=PORT(*, $, $, (), $, 'edp1', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
#162=PORT(*, $, $, (), $, 'edp2', $, .PRIVATE., $, $, $, (), (), .F.,
(), .F., .F., $, $, $, $, $, .F., $, $, $, .F., .F., .F., (), $, $, $,
$, (), (), $, (), $, (), .F., .F., (), (), $);
```

SysML Alternative Engine model instances

```
#1=PROPERTY_UML(*, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #16, $, $, $, $, $, $, $, $, #11, $, $, $, $, $, $, $, $, $);
#2=PROPERTY_UML(*, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #16, $, $, .COMPOSITE., $, $, $, $, $, #12, $, $, $, $, $, $,
$, $, $);
#3=PROPERTY_UML(*, $, $, $, $, 'VALVE_IN', $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, #17, $, $, .COMPOSITE., $, $, $, $, $, #13, $, $, $,
$, $, $, $, $, $);
#4=PROPERTY_UML(*, $, $, $, $, 'edp1', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, #16, $, $, .COMPOSITE., $, $, $, $, $, #14, $, $, $, $,
$, $, $, $, $);
#5=PROPERTY_UML(*, $, $, $, $, '', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #19, $, $, .COMPOSITE., $, $, $, $, $, #14, $, $, $, $, $, $,
$, $, $);
#6=PROPERTY_UML(*, $, $, $, $, 'edp2', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, #16, $, $, .COMPOSITE., $, $, $, $, $, #15, $, $, $, $,
$, $, $, $, $);
#7=PROPERTY_UML(*, $, $, $, $, '', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #19, $, $, .COMPOSITE., $, $, $, $, $, #15, $, $, $, $, $, $,
$, $, $);
#8=PROPERTY_UML(*, $, $, $, $, '', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #17, $, $, .COMPOSITE., $, $, $, $, $, #11, $, $, $, $, $, $,
$, $, $);
#9=PROPERTY_UML(*, $, $, $, $, '', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, #18, $, $, .COMPOSITE., $, $, $, $, $, #12, $, $, $, $, $, $,
$, $, $);
#10=PROPERTY_UML(*, $, $, $, $, 'edp', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, #16, $, $, .COMPOSITE., $, $, $, $, $, #13, $, $, $, $,
$, $, $, $, $);
#11=ASSOCIATION(*, $, $, $, $, 'A_eDP_vALVE_IN', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, (#1,#8), $, $, $);
#12=ASSOCIATION(*, $, $, $, $, 'A_eDP_vALVE_OUT', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, (#2,#9), $, $, $);
#13=ASSOCIATION(*, $, $, $, $, '''A_VALVE_IN', $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#3,#10), $, $, $);
#14=ASSOCIATION(*, $, $, $, $, 'edp1', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, (#4,#5), $, $, $);
#15=ASSOCIATION(*, $, $, $, $, 'A_edp_engine_system', $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, (#6,#7), $, $, $);
#16=BLOCK(*, $, $, $, $, 'EDP', $, $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, (#9,#8), $, $, $, $, $, $);
#17=BLOCK(*, $, $, $, $, 'VALVE_IN', $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, (#10), $, $, $, $, $, $);
#18=BLOCK(*, $, $, $, $, 'VALVE_OUT', $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, (), $, $, $, $, $, $);
#19=BLOCK(*, $, $, $, $, 'Engine System', $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, (), $, $, $, $, $, $);
#20=PACKAGE(*, $, $, $, $, 'Alternative Engine', $, $, $, $, $,
(#12,#11,#13,#16,#17,
     #15,#14,#19,#18), $);
#21=SYSML_MODELING_LANGUAGE('1.1');
#22=T_DATE(15, 9, 2007, $, $, $);
#23=SYSML_MODEL($, $, $, $, #22, $, #21, (#20), $);
```

Knowledge model instances

```
#300=URI('http://www.eads.net/aircraftprogram/a330');
#301=URI('http://www.eads.net/aircraftprogram/a380');
#307=URI('http://www.eads.net/systems/hydraulic/flow30');
#308=URI('http://www.eads.net/systems/hydraulic/flow70');
#309=URI('http://www.eads.net/systems/hydraulic/flow35');
#310=URI('http://www.eads.net/systems/hydraulic/flow150');
#311=URI('http://www.eads.net/systems/hydraulic/flow_unit');
#312=URI('http://www.eads.net/systems/hydraulic/pressure');
#313=URI('http://www.eads.net/systems/hydraulic/pressure_unit');
#314=URI('http://www.eads.net/systems/hydraulic/frequency');
#315=URI('http://www.eads.net/systems/hydraulic/frequency_unit');
#316=AIRCRAFT_PROGRAM(*, $, $, #301, 2., 4.);
#327=FLOW(*, $, $, #307, 30., #331, #335, #336);
#328=FLOW(*, $, $, #308, 70., #331, #335, #336);
#331=FLOW_UNIT(*, $, $, #311, 'L/MN');
#335=PRESSURE(*, $, $, #312, 206., #339);
#336=FREQUENCY(*, $, $, #314, 400., #343);
#339=PRESSURE_UNIT(*, $, $, #313, 'Bar');
#343=FREQUENCY_UNIT(*, $, $, #315, 'Hz');
#357=FLOW(*, $, $, #309, 35., #331, #335, #336);
#362=FLOW(*, $, $, #310, 150., #331, #335, #336);
#367=AIRCRAFT_PROGRAM(*, $, $, #300, 1., 4.);
#372=URI('http://www.eads.net/systems/hydraulic/valve11');
#373=URI('http://www.eads.net/systems/hydraulic/valve12');
#374=VALVE_IN(*, $, $, #372);
#375=VALVE_OUT(*, $, $, #373);
#376=EDP(*, $, $, #382, #328, (#375,#374), 'edp1', $);
#377=URI('http://www.eads.net/systems/hydraulic/valve21');
#378=URI('http://www.eads.net/systems/hydraulic/valve22');
#379=VALVE_IN(*, $, $, #377);
#380=VALVE_OUT(*, $, $, #378);
#381=EDP(*, $, $, #383, #328, (#380,#379), 'edp2', $);
#382=URI('http://www.eads.net/systems/hydraulic/edp1');
#383=URI('http://www.eads.net/systems/hydraulic/edp2');
```

Annotation instances

```
/* in1 */
#370=ANNOTATION_CLASS('valve', (#372), (#146));
#371=ANNOTATION_CLASS('valve', (#372), (#62));
/* out1 */
#390=ANNOTATION_CLASS('valve', (#373), (#64));
#391=ANNOTATION_CLASS('valve', (#373), (#145));
/* in2 */
#392=ANNOTATION_CLASS('valve', (#377), (#66));
#393=ANNOTATION_CLASS('valve', (#377), (#148));
/* out2 */
#394=ANNOTATION_CLASS('valve', (#378), (#68));
#395=ANNOTATION_CLASS('valve', (#378), (#147));

/* flows */
#396=ANNOTATION_CLASS('flow', (#308), (#145,#146,#147,#148));
#397=ANNOTATION_CLASS('flow', (#308), (#62,#64,#66,#68));
```

Annotation instances (alternative Engine)

```
/*annotations*/
/* in1 */
#370=ANNOTATION_CLASS('valve', (#372), (#146));
#371=ANNOTATION_CLASS('valve', (#372), (#17,#4));
/* out1 */
#390=ANNOTATION_CLASS('valve', (#373), (#18,#4));
#391=ANNOTATION_CLASS('valve', (#373), (#145));
/* in2 */
#392=ANNOTATION_CLASS('valve', (#377), (#17,#6));
#393=ANNOTATION_CLASS('valve', (#377), (#148));
/* out2 */
#394=ANNOTATION_CLASS('valve', (#378), (#18,#6));
#395=ANNOTATION_CLASS('valve', (#378), (#147));

/* flows */
#389=ANNOTATION_CLASS('flow', (#308), (#17,#4));
#396=ANNOTATION_CLASS('flow', (#308), (#145,#146,#147,#148));
#397=ANNOTATION_CLASS('flow', (#308), (#18,#4));
#398=ANNOTATION_CLASS('flow', (#308), (#17,#6));
#399=ANNOTATION_CLASS('flow', (#308), (#18,#6));
```

Expression model instances

```
/* linked ports are compatible: same flow unit and same conditions of
flow production */
/* all ports from hydraulic */
#400=ENTITY_VARIABLE(#146, 'p1');
#401=VARIABLE_DOMAIN(#402, #400);
#402=ENTITY_DOMAIN((#145,#146,#147,#148));
#403=ALL_EXPRESSION(*, (#400), (#401), #407);
/* all ports from engine */
#404=ENTITY_VARIABLE(#60, 'p2');
#406=ENTITY_DOMAIN((#60,#62,#68,#64,#66,#152,#153,#154,#155,#156,#157)
);
#405=VARIABLE_DOMAIN(#406, #404);
#407=ALL_EXPRESSION(*, (#404), (#405), #413);
/* or expression */
#408=OR_EXPRESSION(*, (#409,#413));
/* they are not the same valve */
#409=NOT_EXPRESSION(*, #410);
#410=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#411,#412));
#411=ENTITY_PATH_VARIABLE($, 'p1\\{valve\\}', #400, 'valve', .T.);
#412=ENTITY_PATH_VARIABLE($, 'p2\\{valve\\}', #404, 'valve', .T.);
/* or they have the same flow */
#413=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#414,#415));
#414=ENTITY_PATH_VARIABLE($, 'p1\\{flow\\}', #400, 'valve', .T.);
#415=ENTITY_PATH_VARIABLE($, 'p2\\{flow\\}', #404, 'valve', .T.);
```

Expression model instances (alternative Engine)

```
/* linked elements are compatible: same flow unit and same conditions
of flow production */
/* all ports from hydraulic */
#400=ENTITY_VARIABLE(#146, 'p');
#401=VARIABLE_DOMAIN(#402, #400);
#402=ENTITY_DOMAIN((#145,#146,#147,#148));
#403=ALL_EXPRESSION(*, (#400), (#401), #407);
/* all blocks from engine */
#404=ENTITY_VARIABLE(#16, 'b');
#405=VARIABLE_DOMAIN(#406, #404);
#406=ENTITY_DOMAIN((#16,#17,#18,#19));
#407=ALL_EXPRESSION(*, (#404), (#405), #413);
/* or expression */
#408=OR_EXPRESSION(*, (#409,#413));
/* they are not the same valve */
#409=NOT_EXPRESSION(*, #410);
#410=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#411,#412));
#411=ENTITY_PATH_VARIABLE($, 'p\\{valve\\}', #400, 'valve', .T.);
#412=ENTITY_PATH_VARIABLE($, 'b\\{valve\\}', #404, 'valve', .T.);
/* or they have the same flow */
#413=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#414,#415));
#414=ENTITY_PATH_VARIABLE($, 'p\\{flow\\}', #400, 'valve', .T.);
#415=ENTITY_PATH_VARIABLE($, 'b\\{flow\\}', #404, 'valve', .T.);
```

4. Ram Air Turbine models

Knowledge model

-- THIS SCHEMA CONTAINS THE ENTITIES MAKING EXPLICIT THE KNOWLEDGE RELATED -- TO THE RAT CASE STUDY

```
SCHEMA KMODELAC_SCHEMA;

REFERENCE FROM TOP_SCHEMA;
REFERENCE FROM ANNOTATION_SCHEMA;
```

(********************** FLIGHT CYCLE KNOWLEDGE **********************)

--This entity represents a Flight Cycle

```
ENTITY FLIGHT_CYCLE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ITS_FLIGHT_PHASES: SET [0:?] OF FLIGHT_PHASE;
      ITS_GROUND_PHASES:  SET [0:?] OF GROUND_PHASE;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FLIGHT_CYCLE';
END_ENTITY;
```

--This entity represents an Aircraft Phase

```
ENTITY PHASE
ABSTRACT SUPERTYPE
SUBTYPE OF (KNOWLEDGE_CLASS);
      ID: OPTIONAL STRING;
      PHASE_NAME: STRING;
      DESCRIPTION: OPTIONAL STRING;
      SUBPHASE: OPTIONAL PHASE;
      PREVIOUS: OPTIONAL PHASE;
      NEXT: OPTIONAL PHASE;
      UNUSUAL_TRANSITION: OPTIONAL PHASE;
      CONFIGURATION: OPTIONAL SET [0:?] OF AIRCRAFT_CONFIGURATION;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'PHASE';
END_ENTITY;
```

--This entity represents a Flight Phase

```
ENTITY FLIGHT_PHASE
SUBTYPE OF (PHASE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'FLIGHT PHASE';
END_ENTITY;
```

--This entity represents a Ground Phase

```
ENTITY GROUND_PHASE
SUBTYPE OF (PHASE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'GROUND PHASE';
END_ENTITY;
```

--This entity represents an Aircraft Configuration

```
ENTITY AIRCRAFT_CONFIGURATION
ABSTRACT SUPERTYPE
```

```
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'AIRCRAFT CONFIGURATION';
END_ENTITY;
```

--This entity represents an aircraft Event

```
ENTITY EVENT
ABSTRACT SUPERTYPE
SUBTYPE OF (KNOWLEDGE_CLASS);
	STARTS: OPTIONAL PHASE;
	ENDS: OPTIONAL PHASE;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'EVENT';
END_ENTITY;
```

--This entity represents a Performance Parameter

```
ENTITY PERFORMANCE_PARAMETER
ABSTRACT SUPERTYPE
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'PERFORMANCE PARAMETER';
END_ENTITY;
```

--This entity represents a Environment Parameter

```
ENTITY ENVIRONMENT_PARAMETER
SUBTYPE OF (KNOWLEDGE_CLASS);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ENVIRONMENT PARAMETER';
END_ENTITY;
```

--This entity represents an operator Event

```
ENTITY OPERATOR_EVENT
SUBTYPE OF (EVENT);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'OPERATOR EVENT';
END_ENTITY;
```

--This entity represents a system Event

```
ENTITY SYSTEM_EVENT
SUBTYPE OF (EVENT);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'SYSTEM EVENT';
END_ENTITY;
```

--This entity represents one human operation

```
ENTITY AIRCRAFT_OPERATION
SUBTYPE OF (OPERATOR_EVENT);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'AIRCRAFT OPERATION';
END_ENTITY;
```

--This entity represents a distance

```
ENTITY DISTANCE
SUBTYPE OF (SYSTEM_EVENT, PERFORMANCE_PARAMETER);
	DFROM: STRING;
	DTO: STRING;
	THE_VALUE: NUMBER;
	UNIT: DISTANCE_UNIT;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'DISTANCE';
END_ENTITY;
```

```
--This entity represents a distance unit
ENTITY DISTANCE_UNIT
SUBTYPE OF (KNOWLEDGE_CLASS);
	ID: STRING;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'DISTANCE UNIT';
END_ENTITY;

--This entity represents a speed
ENTITY SPEED
SUBTYPE OF (SYSTEM_EVENT, PERFORMANCE_PARAMETER);
	THE_VALUE: NUMBER;
	UNIT: SPEED_UNIT;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'SPEED';
END_ENTITY;

--This entity represents a speed unit
ENTITY SPEED_UNIT
SUBTYPE OF (KNOWLEDGE_CLASS);
	ID: STRING;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'SPEED UNIT';
END_ENTITY;

--This entity represents an ALTITUDE
ENTITY ALTITUDE
SUBTYPE OF (SYSTEM_EVENT, PERFORMANCE_PARAMETER);
	THE_VALUE: NUMBER;
	UNIT: ALTITUDE_UNIT;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ALTITUDE';
END_ENTITY;

--This entity represents an ALTITUDE unit
ENTITY ALTITUDE_UNIT
SUBTYPE OF (KNOWLEDGE_CLASS);
	ID: STRING;
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ALTITUDE UNIT';
END_ENTITY;

--This entity represents a Landing Gear Position
ENTITY LANDING_GEAR_POSITION
SUBTYPE OF (SYSTEM_EVENT, AIRCRAFT_CONFIGURATION);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'LANDING GEAR POSITION';
END_ENTITY;

--This entity represents an engine rating
ENTITY ENGINE_RATING
SUBTYPE OF (SYSTEM_EVENT, AIRCRAFT_CONFIGURATION);
DERIVE
	SELF\KNOWLEDGE_CLASS.NAME:STRING := 'ENGINE_RATING';
END_ENTITY;

--This entity represents the AC power
ENTITY AC_POWER
SUBTYPE OF (SYSTEM_EVENT, AIRCRAFT_CONFIGURATION);
DERIVE
```

```
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'AC POWER';
END_ENTITY;

--This entity represents the slat flap configuration
ENTITY SLAT_FLAP_CONFIGURATION
SUBTYPE OF (AIRCRAFT_CONFIGURATION);
      SLAT_FLAP_NAME: STRING;
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'SLAT FLAP CONFIGURATION';
END_ENTITY;

--This entity represents the deceleration point
ENTITY DECELERATION_POINT
SUBTYPE OF (DISTANCE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'DECELERATION POINT';
END_ENTITY;

--This entity represents the Vlof speed
ENTITY VLOF
SUBTYPE OF (SPEED);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'VLOF';
END_ENTITY;

--This entity represents the V1 speed
ENTITY V1
SUBTYPE OF (SPEED);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'V1';
END_ENTITY;

--This entity represents the VR speed
ENTITY VR
SUBTYPE OF (SPEED);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'VR';
END_ENTITY;

--This entity represents the V2 speed
ENTITY V2
SUBTYPE OF (SPEED);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'V2';
END_ENTITY;

--This entity represents the Safely aborted speed
ENTITY SAFELY_ABORTED_TO
SUBTYPE OF (SPEED);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'SAFELY ABORTED TO';
END_ENTITY;

--This entity represents the en route climb altitude
ENTITY EN_ROUTE_CLIMB
SUBTYPE OF (ALTITUDE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'EN ROUTE CLIMB';
END_ENTITY;
```

--This entity represents the optimum cruise altitude
```
ENTITY OPTIMUM_CRUISE
SUBTYPE OF (ALTITUDE);
DERIVE
      SELF\KNOWLEDGE_CLASS.NAME:STRING := 'OPTIMUM CRUISE';
END_ENTITY;

END_SCHEMA;
```

RAT CORE model instances

```
#70=T_DATE(13, 11, 2009, $, $, $);
#41=RESOURCE(*, $, $, #70, 'MS', 'Load Capability', #70, $, $, $,
.FLOATS., 0.0, (#51,#53,#55,#54), $, $);
#42=FUNCTION_CORE(*, $, $, #70, 'MS', 'RAT Extension', #70, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $);
#43=FUNCTION_CORE(*, $, $, #70, 'MS', '140 175 Power Generation', #70,
$, $, $, $, $, $, $, $, $, $, $, $, $, (#47), $, $);
#44=FUNCTION_CORE(*, $, $, #70, 'MS', 'Landing Power Generation', #70,
$, $, $, $, $, $, $, $, $, $, $, $, $, (#48), $, $);
#45=FUNCTION_CORE(*, $, $, #70, 'MS', '140 Power Generation', #70, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#49), $, $);
#46=FUNCTION_CORE(*, $, $, #70, 'MS', '175 Power Generation', #70, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#50), $, $);
#47=PRODUCES_RELATION($, $, $, #41, CONSTANT_CORE(42.));
#48=PRODUCES_RELATION($, $, $, #41, CONSTANT_CORE(9.5));
#49=PRODUCES_RELATION($, $, $, #41, CONSTANT_CORE(30.));
#50=PRODUCES_RELATION($, $, $, #41, CONSTANT_CORE(50.));
#51=PRODUCED_RELATION($, $, $, #43, $);
#52=EFFBD($, $, $, #70, 'MS', 'RAT Extension', #70, $,
(#60,#59,#58,#65,#64,#63,#62,#61,#57,#56), #42);
#53=PRODUCED_RELATION($, $, $, #44, $);
#54=PRODUCED_RELATION($, $, $, #45, $);
#55=PRODUCED_RELATION($, $, $, #46, $);
#56=OR_CONSTRUCT();
#57=OR_CONSTRUCT();
#58=BRANCH_CONSTRUCT(#43, #56, $, '175kts < VC < 140 kts');
#59=BRANCH_CONSTRUCT(#44, #56, $, 'Landing Gear Deployment');
#60=BRANCH_CONSTRUCT(#45, #56, $, '140 Power Generation');
#61=BRANCH_CONSTRUCT(#46, #56, $, 'VC > 175kts');
#62=BRANCH_CONSTRUCT(#57, #43, $, '');
#63=BRANCH_CONSTRUCT(#57, #44, $, '');
#64=BRANCH_CONSTRUCT(#57, #45, $, '');
#65=BRANCH_CONSTRUCT(#57, #46, $, '');
#66=CORE_MODELING_LANGUAGE('7');
#67=MODEL_CORE($, $, $, 'RAT', #70, #70, #66,
(#45,#44,#43,#42,#41,#46), (#52));
```

Slats SysML model instances

```
#1=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 25.);
#2=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 25.);
#3=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 33.);
#4=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 25.);
#5=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 25.);
#6=LITERAL_INTEGER(*, $, $, (), $, $, $, $, $, $, $, 9.5);
#7=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #1, $, $, $, $, $,
$, $);
#8=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #2, $, $, $, $, $,
$, $);
#9=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #3, $, $, $, $, $,
$, $);
#10=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #4, $, $, $, $, $,
$, $);
#11=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #5, $, $, $, $, $,
$, $);
#12=FLOW_PROPERTY(*, $, $, $, $, 'load', $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, #6, $, $, $, $, $,
$, $);
#13=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#7), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#14=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#8), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#15=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#9), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#16=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#10), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#17=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#11), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#18=ACTIVITY(*, $, $, $, $, 'power consumption', $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, (#12), $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $);
#19=STATE(*, $, $, $, $, 'Take-Off', $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, #13, $, $, $, $, $);
#20=STATE(*, $, $, $, $, 'Climb', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, #14, $, $, $, $, $);
#21=STATE(*, $, $, $, $, 'Cruise', $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, #15, $, $, $, $, $);
#22=STATE(*, $, $, $, $, 'Approach', $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, #16, $, $, $, $, $);
#23=STATE(*, $, $, $, $, 'Landing', $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, #17, $, $, $, $, $);
#24=STATE(*, $, $, $, $, 'Landing Gear Deployment', $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, #18, $, $, $, $, $);
#25=PSEUDOSTATE(*, $, $, $, $, 'taxi-out', $, $, $, $, $, $, $, $, $,
$);
```

```
#26=FINAL_STATE(*, $, $, $, $, 'taxi-in', $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $);
#27=REGION(*, $, $, $, $, 'Flight', $, $, $, $, $, $, $, $, $, $, $,
(#31,#32,#33,#34,#35,
     #36,#37), (#22,#21,#20,#19,#26,
     #25,#24,#23), $, $);
#28=STATE_MACHINE(*, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $,
$, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $, $,
(#27));
#29=BLOCK(*, $, $, (#28), $, 'Slats', $, $, $, $, $, $, $, $, $, $, $,
(), $, $, $, $, $, $, $, $, $, $, $, $, $, $);
#30=PACKAGE(*, $, $, $, $, 'Slats System', $, $, $, $, $, (#29), $);
#31=TRANSITION(*, $, $, $, $, 'engine To power', $, $, $, $, $, $, $,
$, $, .LOCAL_UML., $, $, $, #25, #19, $, $, $);
#32=TRANSITION(*, $, $, $, $, 'Vr', $, $, $, $, $, $, $, $, $,
.LOCAL_UML., $, $, $, #19, #20, $, $, $);
#33=TRANSITION(*, $, $, $, $, 'cruise altitude', $, $, $, $, $, $, $,
$, $, .LOCAL_UML., $, $, $, #20, #21, $, $, $);
#34=TRANSITION(*, $, $, $, $, 'ATC clearance', $, $, $, $, $, $, $, $,
$, .LOCAL_UML., $, $, $, #21, #22, $, $, $);
#35=TRANSITION(*, $, $, $, $, 'Final approach', $, $, $, $, $, $, $,
$, $, .LOCAL_UML., $, $, $, #22, #23, $, $, $);
#36=TRANSITION(*, $, $, $, $, 'Landing Gears Compressed', $, $, $, $,
$, $, $, $, $, .LOCAL_UML., $, $, $, #23, #24, $, $, $);
#37=TRANSITION(*, $, $, $, $, 'exit runaway', $, $, $, $, $, $, $, $,
$, .LOCAL_UML., $, $, $, #24, #26, $, $, $);
#38=SYSML_MODELING_LANGUAGE('1.1');
#39=SYSML_MODEL($, $, $, 'Slats System', #40, $, #38, (#30), $);
#40=T_DATE(13, 11, 2009, $, $, $);
```

Knowledge model instances

```
#303=URI('http://www.eads.net/A350/FlightCycle/Initial_Climb');
#304=URI('http://www.eads.net/A350/FlightCycle/En_Route_Climb');
#305=URI('http://www.eads.net/A350/FlightCycle/Take_off');
#306=URI('http://www.eads.net/A350/FlightCycle/Cruise');
#307=URI('http://www.eads.net/A350/FlightCycle/Descent');
#308=URI('http://www.eads.net/A350/FlightCycle/Approach_to_Land');
#309=URI('http://www.eads.net/A350/FlightCycle/Landing');
#314=URI('http://www.eads.net/A350/FlightCycle/Taxi_out');
#315=URI('http://www.eads.net/A350/FlightCycle/Taxi_in');
#316=URI('http://www.eads.net/A350/FlightCycle/Push_back');
#317=URI('http://www.eads.net/A350/FlightCycle/Engine_start');
#318=URI('http://www.eads.net/A350/FlightCycle/Cabin_safety_briefing')
;
#319=URI('http://www.eads.net/A350/FlightCycle/Rolling_taxi_out');
#320=URI('http://www.eads.net/A350/FlightCycle/Rolling_taxi_in');
#321=URI('http://www.eads.net/A350/FlightCycle/Parking');
#322=FLIGHT_PHASE(*, $, $, #303, 'initialclimb', 'initial climb
phase', $, $, #324, #323, $, $);
#323=FLIGHT_PHASE(*, $, $, #304, 'enrouteclimb', 'en route climb
phase', $, $, #322, #325, $, $);
#324=FLIGHT_PHASE(*, $, $, #305, 'takeoff', 'takeoff phase', $, $,
#336, #322, $, $);
```

```
#325=FLIGHT_PHASE(*, $, $, #306, 'cruise', 'cruise phase', $, $, #323,
#326, $, $);
#326=FLIGHT_PHASE(*, $, $, #307, 'descent', 'descent phase', $, $,
#325, #327, $, $);
#327=FLIGHT_PHASE(*, $, $, #308, 'approachtoland', 'approach to land
phase', $, $, #326, #328, $, $);
#328=FLIGHT_PHASE(*, $, $, #309, 'landing', 'landing phase', $, $,
#327, #337, $, $);
#329=URI('http://www.eads.net/A350/FlightCycle/TO_step1');
#330=URI('http://www.eads.net/A350/FlightCycle/TO_step2');
#331=URI('http://www.eads.net/A350/FlightCycle/TO_confirmed');
#310=URI('http://www.eads.net/A350/FlightCycle/Approach');
#311=URI('http://www.eads.net/A350/FlightCycle/Final_Approach');
#332=FLIGHT_PHASE(*, $, $, #310, 'approach', 'approach subphase', $,
#327, #326, #333, $, $);
#333=FLIGHT_PHASE(*, $, $, #311, 'final_approach', 'final approach
subphase', $, #327, #332, #334, $, $);
#312=URI('http://www.eads.net/A350/FlightCycle/Derot');
#313=URI('http://www.eads.net/A350/FlightCycle/Brake');
#334=FLIGHT_PHASE(*, $, $, #312, 'derot', 'derot subphase', $, #328,
#333, #335, $, $);
#335=FLIGHT_PHASE(*, $, $, #313, 'brake', 'brake subphase', $, #328,
#334, #342, $, $);
#336=FLIGHT_PHASE(*, $, $, #314, 'taxi_out', 'taxi-out phase', $, $,
$, #324, $, $);
#337=FLIGHT_PHASE(*, $, $, #315, 'taxi_in', 'taxi-in phase', $, $,
#328, $, $, $);
#338=FLIGHT_PHASE(*, $, $, #316, 'push_back', 'push-back subphase', $,
#336, $, #341, $, $);
#339=FLIGHT_PHASE(*, $, $, #317, 'engine_start', 'engine start
subphase', $, #336, $, #341, $, $);
#340=FLIGHT_PHASE(*, $, $, #318, 'Cabin_safety_briefing', 'Cabin
safety briefing subphase', $, #336, $, #324, $, $);
#341=FLIGHT_PHASE(*, $, $, #319, 'Rolling_taxi_out', 'Rolling taxi out
subphase', $, #336, $, #324, $, $);
#342=FLIGHT_PHASE(*, $, $, #320, 'Rolling_taxi_in', 'Rolling taxi in
subphase', $, #337, #335, #343, $, $);
#343=FLIGHT_PHASE(*, $, $, #321, 'parking', 'parking subphase', $,
#337, #342, $, $, $);
#344=FLIGHT_PHASE(*, $, $, #329, 'TO_step1', 'TO step1 subphase', $,
#324, #336, #345, $, $);
#345=FLIGHT_PHASE(*, $, $, #330, 'TO_step2', 'TO step2 subphase', $,
#324, #344, #346, $, $);
#346=FLIGHT_PHASE(*, $, $, #331, 'TO_step_confirmed', 'TO step
confirmed subphase', $, #324, #345, #322, $, $);
```

Annotation instances

```
/* taxi-out */
#370=ANNOTATION_CLASS('phase', (#314), (#45,#25));
/* take-off */
#371=ANNOTATION_CLASS('phase', (#305), (#45,#19));
/* Initial Climb */
#372=ANNOTATION_CLASS('phase', (#303), (#43,#20));
/* En route climb */
#373=ANNOTATION_CLASS('phase', (#304), (#43,#20));
/* Cruise */
#374=ANNOTATION_CLASS('phase', (#306), (#46,#21));
/* Descent */
#375=ANNOTATION_CLASS('phase', (#307), (#43,#22));
/* Approach */
#376=ANNOTATION_CLASS('phase', (#310), (#43,#22));
/* Final approach */
#377=ANNOTATION_CLASS('phase', (#311), (#43,#44,#24));
/* Landing */
#378=ANNOTATION_CLASS('phase', (#309), (#45,#23));
/* Taxi-in */
#379=ANNOTATION_CLASS('phase', (#315), (#45,#26));
```

Expression model instances

```
/* "RAT load capability > (Slats power consumption + other systems
consumption)" for each significant flight configuration */
/*for all functions*/
#500=ENTITY_VARIABLE($, 'f');
#501=VARIABLE_DOMAIN(#502, #500);
#502=ENTITY_DOMAIN((#42,#43,#44,#45,#46));
#503=ALL_EXPRESSION(*, (#500), (#501), #507);
/*for all states*/
#504=ENTITY_VARIABLE($, 's');
#505=VARIABLE_DOMAIN(#506, #504);
#506=ENTITY_DOMAIN((#19,#20,#21,#22,#23,#24,#25,#26));
#507=ALL_EXPRESSION(*, (#504), (#505), #508);
/* or expression */
#508=OR_EXPRESSION(*, (#509,#516));
/* they are not the same valve */
#509=NOT_EXPRESSION(*, #510);
#510=COMPARISON_EQUAL_CONTEXT_EXPRESSION(*, (#511,#512));
#511=ENTITY_PATH_VARIABLE($, 'f\\{phase\\}', #500, 'phase', .T.);
#512=ENTITY_PATH_VARIABLE($, 's\\{phase\\}', #504, 'phase', .T.);
/* or exists a produces relation */
#513=ENTITY_VARIABLE($, 'pr');
#514=ENTITY_DOMAIN((#47,#48,#49,#50));
#515=VARIABLE_DOMAIN(#514, #513);
#516=EXISTS_EXPRESSION(*, (#513), (#515), #517);
/*and expression*/
#517=AND_EXPRESSION(*, (#519,#523));
/*relation belongs to function*/
#518=ENTITY_ARRAY_PATH_VARIABLE($, 'f.produces', #500, 'PRODUCES',
.F.);
#519=BELONG_BOOLEAN_EXPRESSION(*, (#513,#518));
```

```
/*exists activity*/
#520=ENTITY_VARIABLE($, 'a');
#521=ENTITY_DOMAIN((#13,#14,#15,#16,#17,#18));
#522=VARIABLE_DOMAIN(#521, #520);
#523=EXISTS_EXPRESSION(*, (#520), (#522), #527);
/*and expression*/
#527=AND_EXPRESSION(*, (#529,#533));
/*a belongs to state.do_activity*/
#528=ENTITY_ARRAY_PATH_VARIABLE($, 's.do_activity', #504,
'DO_ACTIVITY', .F.);
#529=BELONG_BOOLEAN_EXPRESSION(*, (#520,#528));
/*exists flow property*/
#530=ENTITY_VARIABLE($, 'fp');
#531=ENTITY_DOMAIN((#7,#8,#9,#10,#11,#12));
#532=VARIABLE_DOMAIN(#531, #530);
#533=EXISTS_EXPRESSION(*, (#530), (#532), #534);
/*and expression*/
#534=AND_EXPRESSION(*, (#536,#540));
/*fp belongs to activity.owned_attribute*/
#535=ENTITY_ARRAY_PATH_VARIABLE($, 'a.owned_attribute', #520,
'OWNED_ATTRIBUTE', .F.);
#536=BELONG_BOOLEAN_EXPRESSION(*, (#530,#535));
/*exists literal integer*/
#537=ENTITY_VARIABLE($, 'li');
#538=ENTITY_DOMAIN((#1,#2,#3,#4,#5,#6));
#539=VARIABLE_DOMAIN(#538, #537);
#540=EXISTS_EXPRESSION(*, (#537), (#539), #541);
/*and expression*/
#541=AND_EXPRESSION(*, (#543,#546));
/*li belongs to flow_property.default_value*/
#542=ENTITY_ARRAY_PATH_VARIABLE($, 'fp.default_value', #530,
'DEFAULT_VALUE', .F.);
#543=BELONG_BOOLEAN_EXPRESSION(*, (#537,#542));
/* rat produces more than slats needs*/
#544=ENTITY_ARRAY_PATH_VARIABLE($, 'li.the_value', #537, 'THE_VALUE',
.F.);
#545=ENTITY_ARRAY_PATH_VARIABLE($, 'pr.amount', #530, 'AMOUNT', .F.);
#546=COMPARISON_GREATER(*, (#545,#544));
```

Table of figures

Figure 1. SysML diagrams from OMG 11
Figure 2. eFFBD diagram illustration (Vitech Corporation, 2011) 12
Figure 3. Systems and Software Engineering standards evolution up to 2010 (Monzón, 2010) 13
Figure 4. Aircraft Process Development 14
Figure 5. The Architecture consists of Operational –a group of Operational Nodes- and System Architecture –a hierarchy of Components- 19
Figure 6. Transformation of models via meta-models 21
Figure 7. Entity and properties in EXPRESS 30
Figure 8. Example of a derived attribute in EXPRESS 30
Figure 9. Constraints in EXPRESS 31
Figure 10. Expressions top entity 31
Figure 11. Interpretation of an expression using EXPRESS 32
Figure 12. Airbus RBE process 40
Figure 13. Physical Block Diagram representing the communications from a subsystem to an external system. 47
Figure 14. Block Definition Diagram showing external interfaces of CIS model. 48
Figure 15. Method to validate inter-model constraints based on knowledge models. 50
Figure 16. Models in the Aircraft Development V-Cycle. 52
Figure 17. Focus on Export activity 54
Figure 18. UML diagram of the meta-model layer of our approach 56
Figure 19. An excerpt of the CORE meta-model, focus on ItemClass. 57
Figure 20. Focus on Annotation activity 58
Figure 21. Annotation class. 59
Figure 22. Knowledge model of messages and communication protocols 60
Figure 23. Knowledge base, instances of messages a communication protocols. 61
Figure 24. Instances of an annotation 61
Figure 25. Focus on Model Integration activity 61
Figure 26. Inter-model relations diagram 62
Figure 27. Focus on General Constraint Description activity 63
Figure 28. Excerpt of expressions structure in a UML diagram. 64
Figure 29. View of variables model in UML. 64
Figure 30. First Order Logic expression. 65
Figure 31. Messages in the CORE model. 66
Figure 32. Messages in the SysML model 66
Figure 33. Communication protocol must be secure 66
Figure 34. Instances of CORE Link class in SIS model 67
Figure 35. Meta-model of CORE language implemented in EXPRESS 72

Figure 36. Instances of CORE in ISO-10303-21 format 72
Figure 37. Knowledge model implemented in EXPRESS 73
Figure 38. EXPRESS instances representing part of the knowledge base 73
Figure 39. Annotation class implemented in EXPRESS 74
Figure 40. EXPRESS instances representing the annotated model 74
Figure 41. Equivalence class implemented in EXPRESS 74
Figure 42. EXPRESS instance of an equivalence relation 75
Figure 43. Exist expression in BNF form 76
Figure 44. Exist expression in the EXPRESS model 77
Figure 45. EXPRESS function implementing the interpretation of the expression 77
Figure 46. Excerpt of the instances implementing the inter-model constraint 78
Figure 47. Derivation of the value of attribute "the_value" for OR_EXPRESSION entity 79
Figure 48. Functional architecture of the operational validation 80
Figure 49. Schemas in EXPRESS 81
Figure 50. Creation of a project with ECCO toolkit 82
Figure 51. Edition of schema using ECCO toolkit 82
Figure 52. Creation of instances using ECCO toolkit 84
Figure 53. Check of instances with ECCO toolkit 85
Figure 54. Industrial validation strategy 89
Figure 55. Approach applied to the WWS case study 91
Figure 56. Internal Block Diagram of the WWS SysML model 92
Figure 57. Instances representing the WWS model in EXPRESS modeling language 92
Figure 58. Knowledge model according to ATA 38 architecture 93
Figure 59. Instances of ATA 38 knowledge model in EXPRESS modeling language 94
Figure 60. Instances of annotations using ATA 38 knowledge model 94
Figure 61. Instances of constraints in EXPRESS modeling language 95
Figure 62. Approach applied to the Hydraulic and Engine case studies 96
Figure 63. Engine model in SysML 97
Figure 64. Hydraulic model in SysML 98
Figure 65. Instances representing Hydraulic and Engine models in EXPRESS modeling language 98
Figure 66. Alternative SysML Engine model 99
Figure 67. Instances of the alternative Engine model in EXPRESS modeling language 99
Figure 68. Knowledge model according to ATA 29 100
Figure 69. Instances of the ATA 29 knowledge model in EXPRESS modeling language 101
Figure 70. Instances of the annotations using the ATA 29 knowledge model 101
Figure 71. Instances implementing the constraints in EXPRESS modeling language 102
Figure 72. Approach applied to the RAT case study 104

Figure 73. eFFBD diagram of RAT power generation functions.................................... 105
Figure 74. Relationship with the Resource and its value 105
Figure 75. Instances representing the RAT model in EXPRESS modeling language 105
Figure 76. State machine of Slats consumption during flight 106
Figure 77. Instances representing the Slats model in EXPRESS modeling language 106
Figure 78. Knowledge model of the Flight Cycle 108
Figure 79. Instances of the Flight Cycle knowledge model in EXPRESS modeling language 108
Figure 80. Instances of annotations using the Flight Cycle knowledge model 109
Figure 81. Instances implementing the constraint in EXPRESS modeling language 110
Figure 82. Configuration use cases 117
Figure 83. Operational use cases 118
Figure 84. Load of meta-model screen 123
Figure 85. Model meta-data screen 123
Figure 86. Knowledge browsing feature 124
Figure 87. Annotation description screen 124
Figure 88. Constraint meta-data screen 125
Figure 89. Coverage of needs in beta prototype 126
Figure 90. Future meta-model HCI with a SysML example 135
Figure 91. Drag and drop of an instance to annotate it 136
Figure 92. Equivalence relationship between instances from CORE and SysML models 136
Figure 93. Graphical construction of a FOL expression 137
Figure 94. Traceability of the executions of a constraint validation 137

// Summary

Nowadays, complexity of systems frequently implies different engineering teams handling various descriptive models. Each team having a variety of expertise backgrounds, domain knowledge and modeling practices, the heterogeneity of the models themselves is a logical consequence. Therefore, even individually models are well managed; their diversity becomes a problem when engineers need to share their models to perform some overall validations.

We defend the use of implicit knowledge as an important way of reducing the heterogeneity. This knowledge is implicit since it is in engineers' minds but has not been formalized in the models even though it is cardinal to understand them.

After the analysis of current approaches concerning model integration and formalization of implicit knowledge we propose a methodology permitting to complete (annotate) the functional and design models of a system using domain shared knowledge formalized by means of ontologies. These annotations ease the model integration and the cross-model checks. Moreover, it is a non-intrusive approach since the source models are not directly modified. Thus, they are exported into a unified framework by expressing their meta-models in a shared modeling language that permits the syntactical homogenization.

The approach has been formally validated by using the EXPRESS modeling language as shared language. Then, in order to validate it from an industrial point of view, three aircraft domain case studies have been implemented by applying the approach. This industrial aspect has been completed by the development of a prototype allowing engineers to work from a process perspective.

Keywords: Aeronautics, Ontologies (Information retrieval), Teams in the workplace--Data processing, Computer systems, Heterogeneous modeling, Meta-modeling, Inter-model relations, Implicit knowledge.

Résumé

De nos jours, la complexité des systèmes implique fréquemment la participation des différentes équipes d'ingénierie dans la gestion des modèles descriptifs. Chaque équipe ayant une diversité d'expériences, de connaissances du domaine et de pratiques de modélisation, l'hétérogénéité des modèles mêmes est une conséquence logique. Ainsi, malgré la bonne gestion des modèles d'un point de vue individuel, leur variabilité devient un problème quand les ingénieurs nécessitent partager leurs modèles afin d'effectuer des validations globales.

Nous défendons l'utilisation des connaissances implicites comme un moyen important de réduction de l'hétérogénéité. Ces connaissances sont implicites car elles sont dans la tête des ingénieurs mais elles n'ont pas été formalisées dans les modèles bien qu'elles soient essentielles pour les comprendre.

Après avoir analysé les approches actuelles concernant l'intégration de modèles et l'explicitation de connaissances implicites nous proposons une méthodologie qui permet de compléter (annoter) les modèles fonctionnels et de conception d'un système avec des connaissances partagées du domaine formalisées sous la forme d'ontologies. Ces annotations facilitent l'intégration des modèles et la validation de contraintes inter-modèles. En outre, il s'agit d'une approche non intrusive car les modèles originaux ne sont pas modifiés directement. En effet, ils sont exportés dans un environnement unifié en exprimant leurs méta-modèles dans un langage de modélisation partagé qui permet l'homogénéisation syntactique.

L'approche a été validée formellement en utilisant le langage de modélisation EXPRESS en tant que langage partagé. Ensuite, afin de la valider d'un point de vue industriel, trois cas d'étude du domaine aéronautique ont été implémentés en appliquant l'approche. Cet aspect industriel a été complété par le développement d'un prototype permettant de travailler avec les ingénieurs depuis une perspective processus.

Mots-clés : Aéronautique, Ontologies (informatique), Groupes de travail—Informatique, Systèmes informatiques, Modélisation hétérogène, Méta-modélisation, Relations inter-modèles, Connaissances implicites.