UNIVERSITÉ DE MONTRÉAL

MODEL-CHECKING UML DESIGNS USING A TEMPORAL EXTENSION OF THE OBJECT CONSTRAINT LANGUAGE

MATHIEU BERGERON
GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
AOÛT 2004

© Mathieu Bergeron, 2004.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-97927-X
Our file Notre référence
ISBN: 0-612-97927-X

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

MODEL-CHECKING UML DESIGNS USING A TEMPORAL EXTENSION OF THE OBJECT CONSTRAINT LANGUAGE

présenté par : BERGERON Mathieu

en vue de l'obtention du diplôme de : Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de :

M. GUIBAULT François, Ph.D., président
M. MULLINS John, Ph.D., membre et directeur de recherche
M. KORT Skander, Ph.D., membre

To Lise and Roger

ACKNOWLEDGEMENTS

First of all, I want to thank John Mullins sincerely for sharing his knowledge in computer science, mathematics, jazz, and many other subjects. I warmly thank François Guibault and Skander Kort for evaluating this thesis. I also want to thank Robert Charpentier of `RDDC` Valcartier for his trust and support. I salute Gaétan Hains for all kinds of valuable lessons, especially for communicating his interest in functional programming languages. Finally, I thank Louis-Étienne Pigeon who generously accepted to read this thesis.

On a more personal note, I thank Lise and Roger for their incredible patience, and Julie and Marc-André for similar reasons. I also thank all of my friends for occasional libations and various other entertainments. Special thanks to Julien Bourbeau who was particularly available for the latter. Finally, thanks to musician friends who provided the sound-track of this thesis: `La Balconade`, `Skarazula`, `les Globe Glotters` and `l'ensemble Cercamon`.

ABSTRACT

A good design is crucial to the development of quality software. Despite this, behavior correctness is validated on partially or fully implemented software. Yet, it is estimated that correcting an error is 50-200 times more costly during implementation than during design [BP88]. Hence the need for design validation techniques.

We propose one such technique for the Unified Modeling Language (UML) notation, which is the *de facto* industrial design language. It consists of a model-checking framework for the Object Constraint Language (OCL), which is a integral part of the UML standard. Model-checking is a highly automated formal method that has found industrial applications [Cim01]. This thesis also aims at designing tool support for the described model-checking framework. The tool implements the automatic extraction and compilation of a UML model, the computation of the model's execution graph and the model-checking of OCL constraints. However, it clearly lacks the robustness of state-of-the-art model-checkers.

First, we develop a formal semantics for a fragment of the Unified Modeling Language in the Abstract State Machine (ASM) formalism. The semantics supports

important object-oriented features such as inheritance and object creation. It includes the class diagram, the object diagram and the statechart diagram. The static structure and dynamic configuration of the UML model are captured by ASM vocabularies. Well-formedness constraints on these vocabularies capture the static semantics of the chosen diagrams. The tool verifies the constraints while compiling a UML model and warns the designer of errors. An ASM transition rule captures the step semantics, i.e. how a model evolves from one configuration to another.

The OCL language is divided into expressions and constraints. OCL expressions are integrated into the UML semantics in guards, method calls and assignation actions. Their semantics is developed as a recursive function over the formal representation of a UML model configuration. Given a configuration and a variable environment (containing at least a value for the special variable **self**), the function returns the value of the evaluated expression by recursively evaluating subexpressions.

OCL constraints allow for the specification of temporal contracts on the behavior of a model. We use the μ-calculus [Cle90], a well-known temporal logic, to formalize OCL constraints. A constraint is mapped to a set of μ-formulas, where atomic properties are replaced by the corresponding OCL expressions. The constraint is verified if the UML model execution graph satisfies every μ-formula.

CONDENSÉ

Introduction

La phase de conception est cruciale pour le développement de logiciels de qualité. Pourtant, le comportement d'un logiciel est la plupart du temps validé après son implantation, à l'aide de techniques de test. Or, il est estimé que rectifier une erreur après l'implantation est de 50 à 100 fois plus dispendieux que de le faire lors de la conception [BP88]. Ainsi, il existe un besoin réel de techniques de validation dès la phase de conception.

Dans ce mémoire, nous proposons une technique de validation pour la notation UML (Unified Modeling Language). Cette notation est le standard industriel *de facto* en ce qui concerne les langages de conception orientés-objet. Nous proposons d'adapter la vérification automatique (*model-checking*) afin de vérifier des contraintes OCL (Object Constraint Language) sur le comportement d'un modèle UML. Le langage OCL est intégré au standard UML et permet d'exprimer des invariants et des pré/post-conditions.

Nous utilisons le formalisme ASM (Abstract State Machine) pour donner une sémantique à un fragment significatif de la notation UML. La sémantique supporte des mécanismes importants du paradigme orienté-objet tels que l'héritage et la création d'objet. Nous interprétons une contrainte OCL comme un patron contenant des propriétés atomiques sous forme d'expressions OCL et une relation temporelle sous forme d'une μ-formule [Cle90]. Ainsi, la première sémantique permet de calculer le graphe d'exécution d'un modèle UML alors que la seconde indique si une contrainte OCL est satisfaite par le graphe. Il s'agit des deux premiers volets d'une approche de vérification automatique. Le troisième volet, qui n'est pas abordé dans ce mémoire, consiste à développer une repésentation compacte du graphe d'exécution et des algorithmes efficaces de vérification. Un prototype est cependant conçu et implanté à partir du travail de ce mémoire.

Les Abstract State Machines

Le formalisme ASM est conçu pour spécifier des systèmes états-transitions. Il est particulièrement approprié pour les systèmes où les états contiennent beaucoup de données, telles les configurations d'un modèle UML. Une Abstract State Machine contient trois parties: un vocabulaire, une règle de transition et un état initial.

Vocabulaire Un vocabulaire, noté Υ, contient une collection de sortes et un ensemble de signature de fonctions. Les sortes permettent de catégoriser les éléments

contenus dans un état. En plus des éléments simples, nous permettons de spécifier des sortes construites: ensembles, piles, tuples, etc. Une signature sert à déclarer une fonction et à spécifier de quelle façon elle peut être appliquée. Pour chaque sorte construite, nous supposons des fonctions représentant les opérateurs usuels: union, intersection, concatenation, etc.

État Un état respectant le vocabulaire Υ contient deux parties: une collection d'ensembles porteurs et un ensemble d'interprétations de fonctions. Un ensemble porteur contient tous les éléments d'une sorte alors qu'une interprétation de fonction énumère les valeurs prisent par la fonction selon les arguments.

Règle de transition La règle de transition contient des mises à jour de fonction (p. ex. "$f(a) := b$") et des règles de contrôle. Les règles de contrôle supportent le non-déterminisme, ce qui permet de spécifier plusieurs états suivants. La règle est évaluée sur un état et retourne une collection d'ensembles de mises à jour. Appliquer un ensemble de misea à jour correspond à ajuster les valeurs de certaines fonctions pour certains arguments. Le graphe d'exécution d'une `ASM` est calculé en itérant l'évaluation de la règle de transition.

Sémantique des modèles UML

Nous considérons qu'un modèle UML est un diagramme de classe, un diagramme d'états pour chaque classe et un diagramme d'objets. L'aspect dynamique de la sémantique UML est concentré autour du diagramme d'états. Tel qu'il est expliqué dans [OMG03b], la sémantique des diagrammes d'états est une extension orientée-objet des statecharts de Harel [HN96]. L'extension orientée-objet est obtenue en sélectionnant le fragment approprié des actions décrites dans [OMG03b]. L'aspect statique d'un modèle UML est principalement concentré dans le diagramme de classe. La sémantique de ce diagramme est directement inspirée de la sémantique de Java de Stark et al. [SSB01]. Le diagramme d'objets sert à spécifier la configuration initiale du modèle.

Fragment d'UML considéré La Figure 1 est un exemple de diagramme de classe. Une classe spécifie un type d'objet. Elle contient un ensemble de champs de données et un ensemble de méthodes (procédures). La relation d'héritage agit comme une relation de sous-typage. Là où un objet de type **A** peut apparaître, un objet de type **B** peut aussi apparaître. L'association entre classes est toujours considérée comme un champ appartenant à la classe source. La multiplicité est une contrainte restreignant la taille du champ issu de l'association.

La Figure 2 donne un exemple d'un diagramme d'objet. Il s'agit d'un ensemble d'objets possédant chacun un nom, un type et l'évaluation des champs. Remarquez

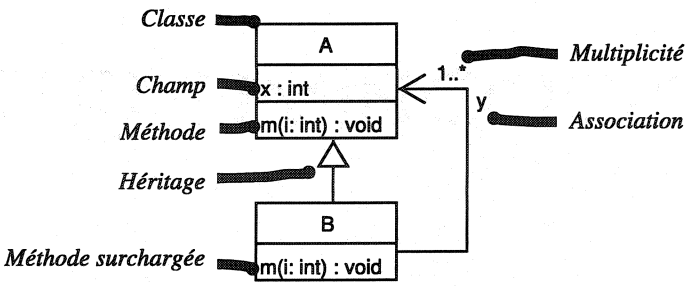


Figure 1: Diagramme de classes UML

que l'objet de type **B** hérite du champs x. Dans le paradigme orienté-objet, l'objet fournit à la fois un entrepôt de données et un contexte d'exécution pour les méthodes. De plus, si l'objet est un fil d'exécution (*thread*), il possède une pile d'appels.

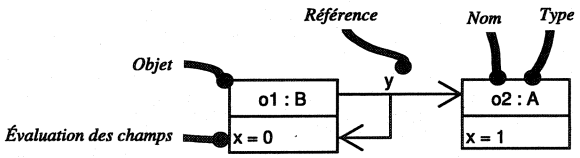


Figure 2: Diagramme d'objets UML

Un diagramme d'état permet de spécifier le comportement d'une classe. La Figure 3 donne un exemple d'un tel diagramme. Le flot de contrôle est spécifié à l'aide d'états et de transitions. Les instructions sont spécifiées à l'aide d'actions. Les déclencheurs permettent de préciser l'appartenance de ces instructions à une méthode. Chaque objet possède sa propre instance (machine à états) du diagramme d'états de sa classe. L'ensemble des états actifs de la machine représente la configuration locale de l'objet.

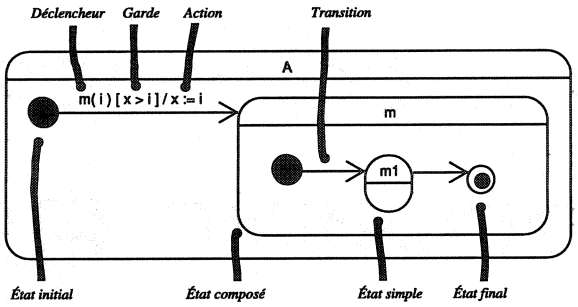


Figure 3: Diagramme d'états UML

Pour qu'une transition soit tirée, l'état source doit être actif, le déclencheur doit correspondre à un événement actif, le garde doit être satisfait et il doit être possible d'exécuter les actions. Les déclencheurs possibles sont présentés à la Figure 4. Un déclencheur vide est toujours satisfait. Les signaux sont des événements globaux qui sont désactivés après chaque pas de l'évolution du modèle. Un déclencheur de méthode est satisfait si cette méthode est en cours d'exécution. Un déclencheur de retour de méthode est satisfait si cette méthode est la dernière à avoir terminé sont exécution.

$$
\begin{array}{l}
Trigger ::= \\
\quad | \quad \star \\
\quad | \quad Signal \\
\quad | \quad Msig \\
\quad | \quad \%Msig \\
\\
Msig ::= \quad Meth(Param : Type, \ldots, Param : Type) : Type
\end{array}
$$

Figure 4: Syntaxe des déclencheurs

Le garde d'une transition est une expression OCL booléenne respectant la syntaxe

présentée à la Figure 5. L'expression "$e.f$" retourne la valeur du champ f appartenant à l'objet retourné par l'expression "e". Les expressions "$\triangle\ e$" et "$e_1 \triangle e_2$" correspondent à l'application des opérateurs unaires et binaires usuels sur les booléens, les entiers et les listes. L'expression "e_1.iterate$(v_1; v_2 = e_2 \mid e_3)$" est évaluée en laissant v_1 itérer à travers la liste retournée par e_1 et en entreposant successivement la valeur de e_3 dans v_2 (qui vaut e_2 au début de l'évaluation). La valeur de l'expression correspond à la valeur finale de v_2.

$$\begin{array}{lll} Exp & ::= & Val \mid Var \mid Exp\ .\ Field \mid Exp\ Bop\ Exp \mid Uop\ Exp \\ & & \mid Exp\ .\ \textbf{iterate}(Var;\ Var = Exp \mid Exp) \end{array}$$

Figure 5: Syntaxe des expressions OCL

Finalement, les actions possibles sont énumérées à la Figure 6. L'envoie d'un signal peut être effectué à partir de n'importe quelle configuration. Les autres actions correspondent intuitivement aux instructions d'une méthode. Conséquemment, elles peuvent être tirées seulement lorsque l'objet les possédant exécute une méthode. De plus, les actions de création/destruction d'objets et d'assignation doivent respecter les contraintes de multiplicité.

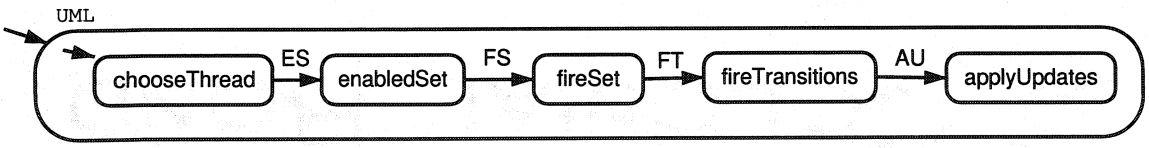
Sémantique Statique La structure d'un modèle est représentée par le vocabulaire ASM $\Upsilon_{\texttt{stat}}$. Le diagramme de classe et les diagrammes d'états sont compilés pour former un état structuré selon ce vocabulaire. La configuration dynamique d'un modèle est représentée par le vocabulaire ASM $\Upsilon_{\texttt{uml}}$. Le diagramme d'objet est

Action ::=
| *Signal*
| **new** *Field*
| **delete** *Field*
| *Field* = *OclExp*
| *OclExp* . *Meth* (*OclExp* , ... , *OclExp*)
| **super** . *Meth* (*OclExp* , ... , *OclExp*)
| % *MSig* [*OclExp*]

Figure 6: Syntaxe des actions

compilé pour former un état structuré selon ce vocabulaire. Lors de la compilation, des contraintes sont vérifiées et des fonctions sont élaborées. Les contraintes permettent de s'assurer de la cohérence des diagrammes entre eux. Par exemple, si un diagramme d'états contient un appel de méthode, on vérifie que cette méthode existe. Les fonctions élaborées sont nécessaires pour définir correctement la sémantique dynamique. Par exemple, la fonction *lookup* indique, lors d'un appel de méthode, si l'on doit exécuter le comportement de la classe appelée ou d'une classe parent de cette dernière. Cette fonction est élaborée selon l'information contenu dans le diagramme de classe et dépend à la fois de la relation d'héritage et des déclarations des méthodes.

Sémantique Dynamique L'évolution d'un modèle `UML` s'effectue par des pas d'exécution. Chaque pas produit une nouvelle configuration. Nous utilisons une règle de transition `ASM` pour calculer ces configurations. La règle est structurée de la sorte:



Dans un premier temps, un fil d'exécution est choisi de façon non-déterministe. Ce fil détermine la méthode en cours d'exécution et les variables locales (arguments de la méthode). Les deux étapes suivantes sélectionnent un ensemble de transitions actives et compatibles. Deux transitions sont compatibles si elles peuvent être tirées dans le même pas. Par exemple, deux transitions assignant des valeurs différentes au même champ ne sont pas compatibles. Cette sélection est non-déterministe et peut entraîner plusieurs pas. Finalement, chaque transition est tirée. L'effet de chacune des transitions est calculé en itérant la liste des actions et en exécutant chaque action.

Intégration des expressions `OCL` La sémantique des expressions `OCL` est donnée comme un fonction d'évaluation. Cette fonction reçoit une expression `OCL`, un environnement de variables et retourne une valeur:

$$\begin{array}{lllll} \textbf{sort} & \mathit{OclEnv} & = & \mathit{Var} \mapsto \mathit{Val} \\ \textbf{function} & [\![\_]\!]\_ & : & \mathit{OclExp} \times \mathit{OclEnv} \rightarrow \mathit{Val} \end{array}$$

Une valeur est soit un booléen, un entier ou un objet; soit une liste de ces valeurs simples. L'interprétation de cette fonction évolue selon la configuration du modèle. Par exemple, l'expression "$e.f$" nécessite d'accéder à la fonction $\mathit{heap} \in \Upsilon_{\texttt{uml}}$, qui contient les objets et leurs champs.

Sémantique des contraintes OCL

Il y a trois sortes de contraintes OCL: les invariants de classe, les préconditions de méthode et les postconditions de méthodes. Un invariant de classe spécifie qu'une condition doit être satisfaite pour toutes les instances de la classe et pour toutes les configurations du modèle UML. Une précondition de méthode spécifie qu'une condition doit être satisfaite avant l'appel de la méthode. Une postcondition de méthode spécifie qu'une condition doit être satisfaite au retour de la méthode. Les conditions sont énoncées à l'aide d'expressions OCL qui deviennent les propriétés atomiques de la μ-formule induite par la contrainte. Cette formule permet d'exprimer une relation temporelle entre les propriétés atomiques.

Les propriétés atomiques sont évaluées durant le calcul du graphe d'exécution du modèle. Nous supposons que leurs valeurs sont entreposées dans les fonctions suivantes:

function $CVals : Obj \rightarrow \mathcal{L}(Val)$
function $@now : Obj \times MSig \rightarrow Bool$
function $@post : Obj \times MSig \rightarrow Bool$

Formellement, une contrainte OCL est un quadruplet $(C, m, exps, \phi)$ où C est une classe, m une signature de méthode, $exps$ une liste d'expressions OCL et ϕ une μ-formule contenant la variable libre r. Par exemple, une pré/post condition est exprimée par le patron suivant:

$$\begin{array}{l} \textbf{context } \text{C::MSig} \\ \quad \textbf{pre: } e_1 \\ \quad \textbf{post: } e_2 \end{array} \quad \mapsto \quad \begin{array}{l} (C, m, \{e_1, e_2\}, \nu X.\Box X \wedge \phi \wedge \psi) \\ \\ \phi \equiv \diamond @n(o, m) \Rightarrow CVals(r).1 \\ \psi \equiv @p(o, m) \Rightarrow CVals(r).2 \end{array}$$

La contrainte doit être interprétée comme suit. Si le modèle est dans une configuration précédant l'appel de la méthode, e_1 doit être vraie. Si le modèle est dans une configuration coïncidant avec le retour de la méthode, alors e_2 doit être vrai.

La sémantique d'une contrainte est donnée comme suit. Pour tout objet o étant une instance de la classe C, une μ-formule $\phi[r \mapsto o]$ est obtenue en remplaçant la variable libre par l'objet. Le graphe d'exécution du modèle `UML` doit satisfaire toutes les μ-formules ainsi induites.

Conclusion

Un prototype a été implanté à partir de ce mémoire. Le modèle `UML` est extrait d'un outil d'édition, compilé, élaboré et soumis à des vérifications statiques. Ensuite, le graphe d'exécution et calculé par un interpréteur d'`ASM` modifié. Finalement, la vérification des contraintes `OCL` est effectuée par un algorithme de vérification reconnu [Cle90]. Le prototype permet de vérifier des modèles `UML` produisant de petits graphes d'exécution. Par contre, il ne possède pas de la robustesse des vérificateurs automatiques de pointe. Les travaux futurs consistent principalement à appliquer des techniques éprouvées de vérification afin d'améliorer la puissance de l'outil.

Deux approches semblent prometteuses. Premièrement, adapter des techniques de vérification symboliques afin d'effectuer les vérifications sur une représentation compacte du graphe d'exécution. La difficulté d'appliquer de telles techniques réside dans le fait que la sémantique des modèles `UML` utilise des structures de données non-bornées, telles des piles et des listes. À cet égard, la sémantique est similaire à la sémantique d'un langage de programmation. Récemment, des techniques intéressantes sont apparues pour la vérification automatique de programmes. Par exemple, le projet Bandera [HDZ00] vise à vérifier automatiquement des programmes Java. Au coeur de la technique réside un algorithme de *slicing* qui extrait automatiquement le modèle à vérifier. Nous pensons qu'il s'agit de l'approche à privilégier lors de travaux futurs.

LIST OF CONTENTS

DEDICATION iv

ACKNOWLEDGMENTS v

ABSTRACT vi

CONDENSÉ viii

LIST OF CONTENTS xx

LIST OF FIGURES xxiv

LIST OF APPENDICES xxvi

LIST OF ACRONYMS AND SYMBOLS xxvii

CHAPTER 1 INTRODUCTION 1

1.1 Motivation 1

1.2 About UML 4

1.3 About OCL 7

1.4 Motivating Example 10

1.5 Objectives and Methodology 14

1.5.1 Using the ASM Formalism 16

1.5.2 Formalizing UML 17
1.6 Related Work 19
1.7 Outline 22

CHAPTER 2 ABSTRACT STATE MACHINES 24
2.1 Introductory Example 24
2.2 State 28
2.3 Transition 29
2.4 Execution Graph 31
2.5 Observations 33
2.6 Specifications 33
2.6.1 Constructed Sorts 34
2.6.2 Syntactic Sugar 34
2.6.3 Control Flow Graphs 36

CHAPTER 3 UML MODEL SEMANTICS 38
3.1 Formalization Rationales 38
3.2 Considered Fragment 40
3.3 Using the ASM Formalism 43
3.4 Υ_{STAT} 44
3.4.1 Diagram Information 45
3.4.2 Elaborated Information 46
3.4.3 Well-formedness 47
3.5 Υ_{UML} 48
3.5.1 Well-formedness 49
3.6 Υ_{AUX} 50
3.6.1 Well-formedness 51
3.7 Step 51
3.7.1 Step Computation 52

3.7.2 Behavior Inheritance 55
3.7.3 Action Semantics 56
3.8 Execution Graph 59
3.9 Safety Properties 60

CHAPTER 4 OCL SEMANTICS 61
4.1 Expressions 62
4.1.1 Syntax 62
4.1.2 Semantics 64
4.2 Constraints 66
4.2.1 Syntax 66
4.2.2 Semantics 68
4.2.3 Templates 69
4.3 UML Model Semantics Extensions 71

CHAPTER 5 TOOL DESIGN 74
5.1 Design Workflow 74
5.2 Architecture 75
5.2.1 UML Compiler 77
5.2.2 Graph Generation 78
5.2.3 Model-checking 79
5.2.4 Diagnosis Generation 79
5.2.5 Visualization 80

CHAPTER 6 CONCLUSION 81
6.1 Motivation 81
6.2 Achievements 82
6.3 Issues 82
6.3.1 Semantics Issues 82

6.3.2 Robustness Issues 84
6.4 Future Work 85

BIBLIOGRAPHY 87

APPENDICES 93

LIST OF FIGURES

Figure 1 Diagramme de classes UML xii
Figure 2 Diagramme d'objets UML xii
Figure 3 Diagramme d'états UML xiii
Figure 4 Syntaxe des déclencheurs xiii
Figure 5 Syntaxe des expressions OCL xiv
Figure 6 Syntaxe des actions xv

Figure 1.1 UML Class Diagram 6
Figure 1.2 UML Object Diagram 6
Figure 1.3 UML Statechart Diagram 7
Figure 1.4 Example of an Invariant 10
Figure 1.5 Example of a Postcondition 10
Figure 1.6 Example of a Memory Cell 11
Figure 1.7 Cell's Behavior 11
Figure 1.8 Extended Memory Cell 12
Figure 1.9 BackupCell's Behavior 13
Figure 1.10 Postcondition for Method inc 13
Figure 1.11 Example of an Execution Graph 13

Figure 2.1 Example of an Execution Path 28
Figure 2.2 Sort Constructors 34

Figure 2.3 Example of Named Rules 35
Figure 2.4 Example of Distinguished Elements 36
Figure 2.5 Example of a Named Tuple 36
Figure 2.6 Example of a Control Flow Graph 37

Figure 3.1 Trigger Syntax 41
Figure 3.2 Action Syntax 42
Figure 3.3 ASM Execution Graph and UML Step 44

Figure 4.1 Collection Operators 63
Figure 4.2 μ-Calculus Syntax 67

Figure 5.1 Design Workflow 75
Figure 5.2 Prototype Architecture 76
Figure 5.3 Graphical Interface 77
Figure 5.4 Model Simulation 79

Figure 6.1 Backward Navigation Example 83
Figure 6.2 Excerpt of Cell's Behavior 84

Figure A.1 Predicate Semantics 95
Figure A.2 Rule Semantics 99

Figure B.1 Method Inheritance Example 113

Figure C.1 OCL Expressions Type System 130
Figure C.2 Predefined Types for OCL Expressions 132
Figure C.3 μ-Calculus Semantics 133

LIST OF APPENDICES

APPENDIX A ASM DETAILS 93
APPENDIX B UML SEMANTICS DETAILS 100
APPENDIX C OCL SEMANTICS DETAILS 129

LIST OF ACRONYMS AND SYMBOLS

Acronyms

ASM	Abstract State Machine
CTL	Computation Tree Logic
CRAC	Conception et Réalisation des Applications Complexes
LTL	Linear Temporal Logic
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
RFP	Request for Proposals
UML	Unified Modeling Language

Symbols

Υ	ASM vocabulary
$\mathfrak{A}, \mathfrak{B}, \ldots$	ASM states
$\perp$	Undefined value
ζ	Variable assignement

$\Phi_{\mathtt{a}}, \Phi_{\mathtt{b}}, \ldots$	ASM predicates
$\Pi_{\mathtt{a}}, \Pi_{\mathtt{b}}, \ldots$	ASM transition rules
$\mathcal{S}(A)$	Sets of elements of A
$\mathcal{L}(A)$	Lists of elements of A
$\mathcal{T}(A_1, \ldots, A_n)$	n-tuples in $A_1 \times \ldots \times A_n$
$\mathcal{R}(A)$	Relations over A
$A_1 \mid \ldots \mid A_n$	Elements of $A_1 \cup \ldots \cup A_n$
$A_1 \mapsto A_2$	Finite mappings from A_1 to A_2
A^*	Stacks of elements of A
$\emptyset$	Empty set
$\langle\rangle$	Empty list, empty stack
Θ	ASM execution graph
$\rightarrow$	ASM transition relation
Λ	UML execution graph
$\longrightarrow$	UML step relation
$e \vdash \tau$	Expression e is of type τ
$\sqsubseteq$	Subtype relation
$\prec_d$	Direct inheritance relation
$\preceq_h$	Inheritance relation
$\preceq_s$	More specific relation
$\preceq_o$	Overriding relation
$\Theta_{\mid\Phi}$	Observation of Θ through predicate Φ
$\Theta \models \phi$	The μ-formula ϕ holds in Θ
$\Lambda \models c$	The OCL constraint c holds in Λ

CHAPTER 1

INTRODUCTION

1.1 Motivation

Dream not of a Software-Free World Software is omnipresent. Although some would complain about the latter statement, few would disagree. And it is not likely to be falsified in the near future. Software is a flourishing industry and research domain. First of all, because the only limit to the pouring out of new software is the human imagination, which is (collectively) quite vast. Secondly, and more importantly, the development of correct and reliable software is still an open issue. Surveys show that large software projects have a huge probability of failure - in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than succeed [1] .

Software Crisis That state of things is often refered to as the software crisis, a term coined by Doug McIlroy at the 1968 NATO Conference in Software Engineering.

[1] http://www.omg.org/gettingstarted/what_is_uml.htm

However, as the crisis as yet to be resolved, it may be best described as a chronic affliction (as suggested by Daniel Tiechrow of the University of Michigan), an affliction that sets apart the software engineering discipline from other engineering disciplines. Software projects are notoriously difficult to manage. The correlation between the manpower assigned to development tasks and the speed at which things get done is ambiguous to say the least. As Brooks puts it in his famous essay *The Mythical Man-Month* [BBF+75], "adding people to a late software project makes it later". To make things worse, assessing the quality of software is a difficult task that cannot rely solely on quantitative measurements.

Software Quality Indeed, there are many factors to consider when evaluating software quality, many of which are qualitative. Among the factors usually considered are correctness, reliability, efficiency, security, usability, maintainability and flexibility. Arguably the single most important is correctness, i.e. making sure the software does what it is supposed to do. Developing a correct software requires a precise specification of the intended behavior and a validation strategy.

Software Development Software is developed through design, implementation and testing. The behavior correctness is validated in the testing phase. One well-known problem with this approach is that behavioral errors often result from design flaws. Correcting a design flaw on a partially or fully implemented software is intricate. In fact, it is estimated that correcting such an error is 50-200 times more costly during implementation than during design [BP88]. Hence the need to complement testing with design validation techniques.

Software Design A design is not an easy thing to validate, however, as it generally cannot be executed and tested. Indeed, it is meant to be a model or representation

of the software that will be built later [Pre87]. It is meant to abstract away from data and behavioral details. Still, the design model must be written down in some language. Nowadays, the *de facto* industrial standard is the Unified Modeling Language (UML). UML supports the specification of object-oriented software, a popular paradigm that favors data and control encapsulation in an effort to increase reuse of software components. As we will see in the motivating example (Section 1.4), object-orientation includes a behavior refinement mechanism. This increases reuse possibilities and makes design (or source code) more compact. Unfortunately, it also makes behavior difficult to understand and design more difficult to achieve.

Formal Methods This is an umbrella term regrouping specification and validation techniques based on mathematically precise descriptions of computer systems or softwares. As they can support abstract descriptions, it is a natural choice for software design validation. The main drawback of formal methods is that they require highly trained operators, as some techniques are not fully automated. Nonetheless, formal methods have found many successful applications [CW96]. To preserve UML's industrial appeal, highly automated methods are more appropriate. Model-checking is one such technique and it has proven its efficiency in various applications, including industrial applications [Cim01].

Model-checking Bérard et al. [Bro01] describe model-checking as verifying some properties of the *model* of a system. This requires three things: a formal language to describe the model, a formal language to describe the properties and efficient verification techniques. The semantics of the model description maps a model to a representation of its behavior. Conceptually, that representation is a graph depicting every possible evolution of the model. Nodes are labelled with atomic properties, i.e. conditions on a model's configuration. Properties are specified in some kind of tem-

poral logic. It allows a temporal relation to be defined between atomic properties. For example, one could specify that if the model reaches condition b_1, it will eventually reach condition b_2. Again the property is specified in a formal language with a formal semantics, which defines whether a model satisfies a given property. Conceptually, the verification of the formula is done by computing the execution graph of the model and exploring it with respect to the temporal property. In addition, a complete model-checking framework includes a set of techniques that allow the graph to be encoded in a compact manner and properties to be verified efficiently.

A practical problem with model-checking is that the designer has to be familiar with the formal language used to describe the system. Moreover, he/she has to be familiar with the property language used to specify the correct behavior of the system. One way of circumventing that problem is to offer the model-checking of UML models. Indeed, UML already integrates a property language, namely the Object Constraint Language (OCL) constraints. The resultant would be a design validation technique appealing to the software industry. Our aim is to provide the first two steps towards a model-checking framework of OCL constraints: formal semantics for UML models and OCL constraints.

1.2 About UML

UML Inception In the nineteen seventies and eighties, several object-oriented design notations and methodologies were developed. This culminated in the early nineties with over than 50 methodologies competing in what is known as the *method wars*. As these methods matured and inspired each other, the need for an unified notation became clear. In the mid-nineties, Grady Booch and Jim Rumbaugh joined efforts in the fusion of the Booch and OMT (Object Modeling Technique) methods. Then Ivar Jacobson incorporated his OOSE (Object-Oriented Software Engineering)

method, which resulted in the Unified Modeling Language. The first specification documents of UML (UML 0.9) were proposed to the OMG (Object Management Group) in 1996. The OMG issued an RFP (Request for Proposals) as a first step toward a UML 1.0 definition. The consortium built around that RFP included organizations such as Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI and Unisys. UML 1.4 is the current version of the standard but the 2.0 version is nearly complete [OMG03a, OMG03b].

UML Diagrams The specification of the Unified Modeling Language presents a collection of 12 kinds of diagrams. Structural diagrams allow us to specify the static structure of the application. These include class diagrams, object diagrams, component diagrams and deployment diagrams. Behavioral diagrams represent different aspects of the behavior of the application. They include use case diagrams, sequence diagrams, activity diagrams, collaboration diagrams and statechart diagrams. Model management diagrams are meant to support project management by documenting how development can be broken down into modules and subsystems; they include package diagrams, subsystem diagrams and model diagrams. Rumbaugh gives a complete introduction to UML in [RJB98]. In this thesis, we concentrate on class diagrams, object diagrams and statechart diagrams.

Modeling Data Structures in UML Figure 1.1 shows an example of a class diagram containing every feature covered in this thesis. A class is the object-oriented mechanism for type definition. A class specification contains both data (fields) and procedures (methods). The object-oriented sub-typing mechanism is called inheritance. For example, class **B** inherits the fields and methods of class **A**. Wherever an instance of class **A** may occur, an instance of class **B** may also occur. In addition, class **B** can redefine (override) methods of class **A**. An association specifies a seman-

tic relationship that can occur between typed instances [OMG03b]. The multiplicity modifier restricts the number of participating instances in that relationship.

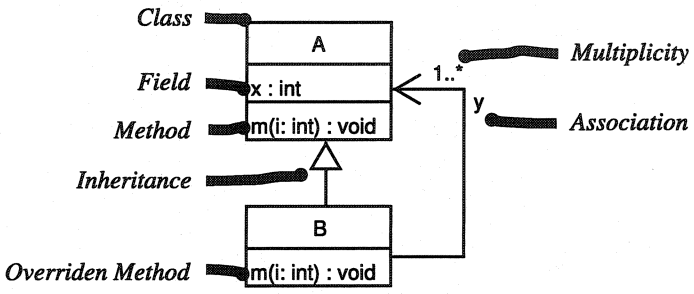


Figure 1.1: UML Class Diagram

Modeling Data in UML Figure 1.2 shows an example of an object diagram containing every feature covered in this thesis. Objects act both as data store and as context for the execution of methods. Notice how an object is specified by its name, its type and a valuation of its fields. Notice also how field x is inherited from class **A**. A reference is an instantiated association. The object `o1` is allowed to call methods of the objects `o1` or `o2` through the reference y.

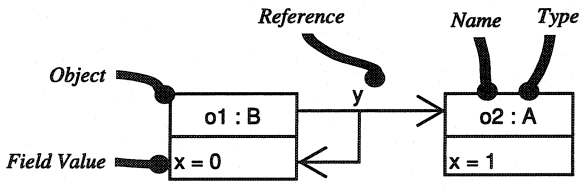


Figure 1.2: UML Object Diagram

Modeling Behavior in UML Figure 1.3 shows an example of a statechart diagram containing nearly every feature covered in this thesis. For each class, the UML model must contain a statechart that specifies its behavior. The class control flow

is modeled through states and transitions. Note that some states (called composite states) contain sub-states. Instructions for the class methods are modeled by labeling transitions with triggers, guards and actions. A trigger may indicate that a transition is part of a particular method. A guard states a condition on the model configuration that must be met for the transition to be fired. A transition's actions model instructions such as method calls, field assignations, etc.

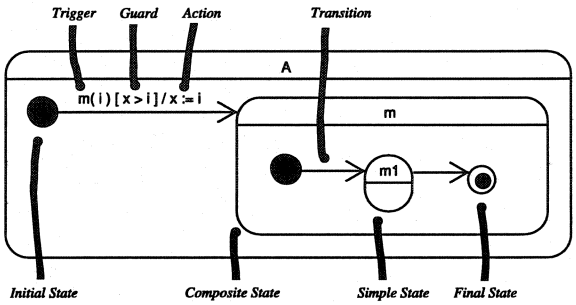


Figure 1.3: UML Statechart Diagram

Each object of the object diagram has its own state-machine. It consists of a set of active states belonging to the statechart of the object's class. One step in the model evolution corresponds to the firing of a set of state-machine transitions. As a result, active states are updated by deactivating source states of the fired transitions and activating target states. If a final state is reached, the enclosing state is deactivated, possibly deactivating the state-machine. It can be reactivated by a method call.

1.3 About OCL

The Object Constraint Language was developed at IBM by Jos Warmer as part of the Syntropy method. It was integrated into UML 1.0 and has since gained increasing interest. The purpose of this language is to complement the notation by allowing details that are not easily captured by diagrams to be specified. OCL is easy to

understand, yet precise enough for software design. It is divided into OCL expressions and OCL constraints [OMG02]. The latter enable the designer to specify contracts on the intended behavior of the model. The former are used in a UML model wherever an expression is needed.

OCL Expressions

OCL expressions are strongly typed and have no side-effects , i.e. their evaluation does not change the model configuration. Assuming some restrictions on the OCL semantics (see Section 1.5), an OCL expression evaluates to a precise value given a model configuration. OCL expressions are therefore used to specify values that will change as the model evolves. As such, expressions are useful in a number of places in a UML model:

- To specify the initial value of a field (which may depend on other values)
- To specify the value of a parameter in a method call
- To specify a new value for a field
- To specify a guard in a statechart diagram.

Evaluation Context An OCL expression is always evaluated (and type-checked) according to a context. Statically, the context is a class. During evaluation, that context is an instance of that class. The special variable **self** always refers to that instance. For example, consider the class diagram of Fig. 1.1. Given class **B** as context, the expression "**self**.x" is a legitimate expression. This expression simply accesses the value of field x. The expression is of integer type. Evaluated on the object diagram of Fig. 1.2 with the object o1 as context, it yields the value 0.

Basic Operators Like any programming-language expressions, OCL expressions support literals and basic operators on booleans and integers. Continuing our example, the expression "**self**.x < 1" is a legitimate expression of boolean type. It yields the value **true**.

Navigation The main distinction between OCL expressions and programming language expressions is the support for collections. An OCL expression may *navigate* object references and collect the resulting set of objects. From a collection, the "." operator can be reapplied to further navigate the configuration. For example, the expression "**self**.y" is a legitimate expression. The type of the expression is a collection of type **A**. Evaluated on the object diagram of Fig. 1.2, it returns $\{o1, o2\}$. Further navigation takes the "**self**.y.x" form. Evaluated on our example, it returns $\{0, 1\}$, i.e. the value of field x for the objects o1 and o2.

Collection Operators OCL expressions also include collection operators such as **forall**, **exists**, **filter**, etc. For example, the expression "**self**.y.x.**forall**$\{z \mid z = 1\}$" is a legetimate OCL expression of boolean type. Once evaluated, it returns **false**.

Constraints

There are three kinds of OCL constraints: class invariants, method preconditions and method postconditions. A class invariant states that a condition must always be met by all instances of the class. A method precondition states that a condition must be met before the method is invoked. A method postcondition states that a condition must be met after the method has returned. For every kind of constraint, the conditions are specified using an OCL expression of boolean type. Note that a constraint does not modify a model's behavior. Rather, it is used *a posteriori* to

verify whether it satisfies some behavioral requirement.

Context When an OCL expression is used as a guard in a statechart, the context is implicit. In fact, it is always the class associated with the statechart. For OCL constraints, however, the context has to be specified explicitly. For example, Fig. 1.4 shows an invariant for class **B**, stating that it may never reference an object whose field x is 10 or higher.

context: B
inv: **self**.y.**forall**$\{z \mid z.\mathrm{x} < 10\}$

Figure 1.4: Example of an Invariant

For method preconditions and postconditions, the context is a class and a method signature. For example, Fig. 1.5 states that, after every call to method **set**, field x must hold the value of the method parameter i.

context: A::set(i:int)
post: **self**.x = i

Figure 1.5: Example of a Postcondition

1.4 Motivating Example

Object-oriented software design is a difficult task, particularly when behavior inheritance is involved. As an illustration, we present the example of a memory cell. Imagine a simple object-oriented component acting as a memory cell. The cell must support updates, incrementations and retrievals of its value. To specify the cell, we create a class **Cell** with the appropriate fields and methods (Fig. 1.6).

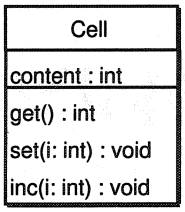


Figure 1.6: Example of a Memory Cell

The class behavior is specified by the statechart of Fig. 1.7 (the action $\%m$ stands for the return of method m). Notice how method **inc** calls methods **get** and **set** to retrieve and update the cell value. This is good practice, as these methods may display refined behavior in subclasses. Imagine that we want to extend the memory cell specification to include a method that cancels the last update and returns to the previous value. We do that by specifying a new class that inherits from class **Cell**. The new class diagram is shown in Fig. 1.8.

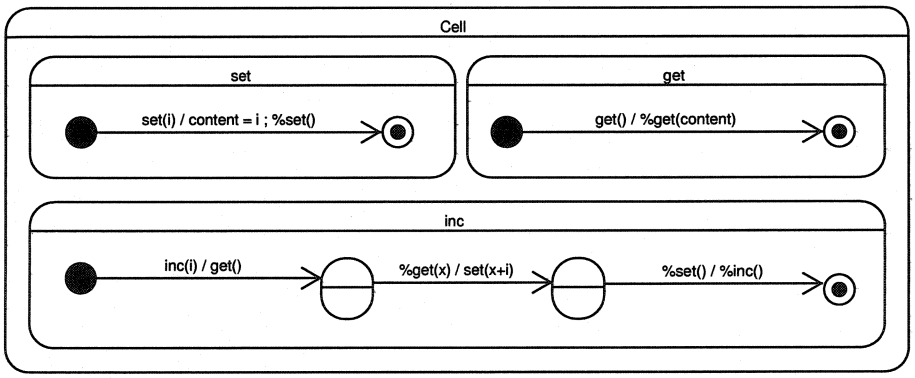


Figure 1.7: **Cell**'s Behavior

Class **BackupCell** adds field *backup* and method **restore** to the memory cell specification. Its behavior is specified by the statechart diagram of Fig. 1.9. Notice how method **set** is redefined to copy the value of the cell in field *backup*.

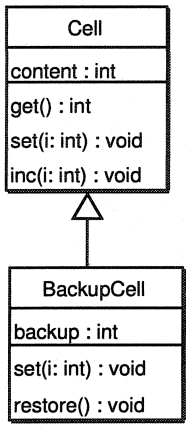


Figure 1.8: Extended Memory Cell

Consider that method **inc**, which is not redefined, is called on an instance of class **BackupCell**. The call is answered by the behavior inherited from class **Cell**. Method **get** is called to retrieve the current value. That value is incremented by the integer received as parameter. Then, method **set** is used to set the content of the memory cell to the incremented value. Since method **inc** executes on an instance of class **BackupCell**, the overridden method **set** is called to execute that task. Thus, the behavior defined by the statechart of **BackupCell** is executed and the old content is stored in the *backup* field.

We see how the object-oriented behavior inheritance mechanism can make design difficult. In this example, however, it is quite simple to specify behavioral correctness using an `OCL` constraint (see Fig. 1.10). It states that, after a call to method **inc**, the *backup* field must hold the value that the *content* field had prior to the method call. This is the meaning of the special ***@pre*** operator.

In order to verify the constraint, the designer must specify the initial configuration of the model using an object diagram. From that configuration, the execution graph of the model is computed according to the `UML` model semantics. During that com-

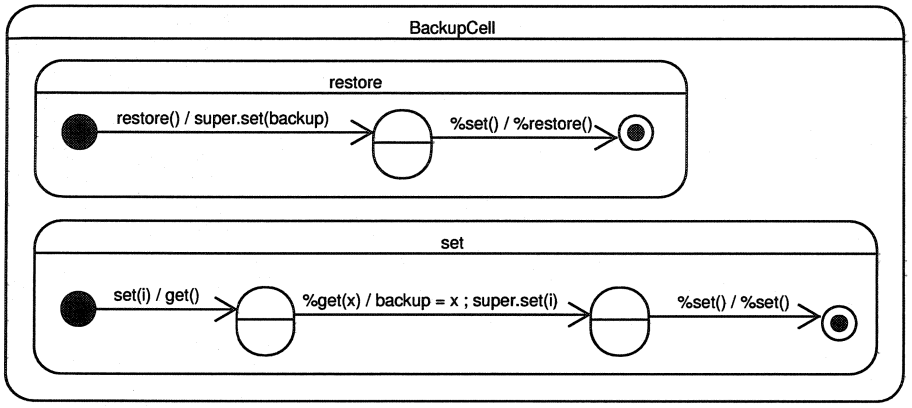


Figure 1.9: **BackupCell**'s Behavior

context: BackupCell::inc(i:int)
post: **self**.backup = (**self**.content)@*pre*

Figure 1.10: Postcondition for Method **inc**

putation, the OCL expression "**self**.backup = content@*pre*" is evaluated over every configuration of the model. The evaluated expression becomes an atomic property, namely b_1. In addition, configurations are labeled with another atomic property (b_2) indicating the return of the **inc**. Figure 1.11 presents an example of such an execution graph.

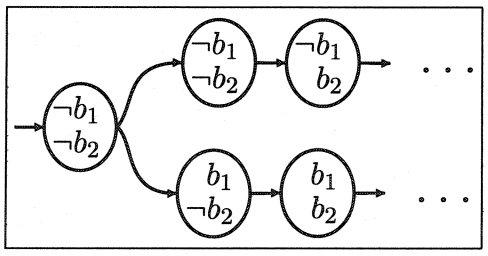


Figure 1.11: Example of an Execution Graph

The next verification step is to derive a temporal relation from the OCL constraint.

In this case, the relation would state that when the model reaches a configuration where b_2 is true, b_1 is true. Of course, this relation is formally expressed in a temporal logic. Conceptually, it is verified by searching a path in the graph leading to a configuration where b_2 and $\neg b_1$. If such a path exists, the constraint is violated (as in the case of Fig. 1.11). The designer can then scrutinize the execution graph in order to correct his/her design.

1.5 Objectives and Methodology

UML Semantics The main objective of this thesis is to develop a UML model semantics that integrates OCL expressions and supports the verification of OCL constraints. We develop the semantics in the Abstract State Machine (ASM) formalism, as explained in Subsection 1.5.1. It is operational, i.e. it allows us to compute an execution graph where every possible evolution of the model is depicted.

As we consider UML to be an object-oriented software design language, the semantics has to integrate important object-oriented features such as object creation and inheritance. Furthermore, UML model configurations must contain all necessary information for the evaluation of OCL expressions. This includes an explicit representation of objects and method instances. Subsection 1.5.2 further discusses the formalization of UML.

OCL Constraints Semantics We also aim to give a temporal semantics to OCL constraints. This requires a formalization of class invariants, method preconditions and method postconditions, which, in turn, requires a formalization of OCL expressions. We use the μ-calculus [Cle90], a well-known temporal logic, to formalize OCL constraints. A constraint is mapped to a set of μ-formulas, where atomic properties

correspond to OCL expressions and method instances. Each formula applies to an instance of the constraint context. The constraint is verified if the model execution graph satisfies every μ-formula.

OCL Expression Semantics We choose a significant fragment of OCL expressions for formalization. In order to simplify, we do not consider OCL to be a three-valued logic. The semantics of OCL expressions is given in terms of a recursive evaluation function. Given a UML configuration and a variable environment (containing at least a value for the special variable **self**), the function returns the value of the evaluated expression by recursively evaluating subexpressions. The function is incorporated into the ASM semantics as an external function. OCL expressions already have a formal syntax and semantics [OMG02]. The only ambiguity is the application of collection operations on collections such as sets and so-called *bags* (multi-sets). The problem is that such operations become non-deterministic, i.e. may return multiple values. To avoid the ambiguity, the only collections we consider are lists.

Tool Support With the semantics are defined, the verification of OCL constraints on UML models yielding finite execution graphs is possible. We design a tool to handle the extraction and compilation of the UML model, computation of the model execution graph and actual model-checking of OCL constraints. This tool is a first step toward a practical model-checking framework for OCL constraints. It required us to design a model-checker from scratch. The prototype is divided into three main modules. The first module is a UML compiler. It extracts diagram information from the UML diagram editor and elaborates the necessary static functions. In addition, it verifies static well-formedness constraints and warns the designer of compilation errors. The second module is a specialized ASM interpreter. It computes the UML model execution graph by interpreting the ASM rule on the initial configuration and

computing some graph observations. The last module is the model-checker itself. It transforms an OCL constraint into a set of μ-formulas and verifies each μ-formula.

1.5.1 Using the ASM Formalism

The ASM formalism is designed to specify state-transition systems. It is most suited for systems where states carry complex data structures, such as the UML model configuration. A state is a first-order structure, i.e. a set of first-order function interpretations over carrier sets. Transitions are specified with a transition rule supporting non-determinism. Evaluation of the rule yields a set of updates to be applied to the state. The execution graph is computed by iterating the transition rule.

Formal Semantics in ASM The Montages framework [KP97a] is an excellent example of how formal semantics are developed using the ASM formalism. First, the framework allows the static aspects of a programming language to be described. The dynamic semantics is captured through an ASM transition rule. A program is mapped to an ASM initial state. Static functions hold the abstract syntax of the program. The static semantics specifies how the program is elaborated before execution, i.e. exactly how the program is mapped to the initial state. Dynamic functions hold the program state and temporary variables such as an abstract program counter. The program evolution is described through the ASM evolution.

A similar approach has been used to formalize many programming languages in the ASM formalism, including Java [SSB01], C [HS00], Oberon [KP97b], Occam [GM90], Smalltalk [Mlo01] and ML [CH99].

1.5.2 Formalizing UML

Developing a formal semantics for the entire Unified Modeling Language is out of the scope of this thesis. In fact, UML is often considered as a family of languages. The language is meant to be adapted for the different purposes at hand. As a result, a complete formal semantics should include a formal derivation mechanism for adapting UML. Furthermore, the formalization of 12 diagrams representing a considerable amount of work is also beyond the scope of this thesis.

Lifting Ambiguities Description of the UML semantics in natural language is notoriously ambiguous. In [RW], Reggio and Wieringa make a survey of ambiguities, inconsistencies, incompleteness and other issues related to the UML informal semantics.

Many ambiguities are caused directly by an unspecified event dispatch mechanism. First, it is not specified how the receiver object of a message is chosen. We resolve that problem by forcing the receiver object to be specified by an OCL expression, which is evaluated before the message is sent. Subsequent ambiguities are resolved using a simple scheme: signals are considered as global events, while method calls are explicitly processed in calling stacks.

Another set of ambiguities exist because threads of execution are left unspecified. We resolve such ambiguities by providing a thread semantics similar to the Java thread semantics of [SSB01]. Each thread has its own calling stack. A (non-deterministic) thread scheduler chooses a thread at the beginning of the step. The method on top of the thread calling stack is the current method for that step.

Another issue is data coherence. For example, in the UML semantics, there is no discussion on how to prevent parallel actions from incoherently updating the same

variable. We solve that problem by extending the notion of transition conflict to include a data coherence verification, that is, two transitions that have (dynamically) incoherent effects are not fired in parallel.

Although UML suggests that behavior inheritance should go along with class inheritance, the inheritance mechanism is left unspecified. We remove that ambiguity by adapting the inheritance semantics of Java to UML. Essentially, a class statechart is the composition of every one of its superclass statecharts. However, the transitions belonging to a superclass behavior are only enabled when an inherited (and not overridden) method is called. The dynamic semantics is defined on top of the static semantics of inheritance, which is again directly inspired by [SSB01].

Static Semantics The static semantics of a UML model dictates how diagrams are mapped to static ASM functions. Additional functions are elaborated such as the inheritance relation, a set of activated and deactivated states for each transition, a method lookup function, etc. Well-formedness constraints indicate which models are statically correct. This includes a definition of diagram coherence, e.g. when the object diagram is a correct initial configuration.

Dynamic Semantics The UML model configuration is captured by dynamic functions. The step semantics is captured by the ASM transition rule, which is centered around three central concepts:

1. When a transition is enabled
2. When two transitions cannot be fired in the same step
3. The effect of a transition.

A UML model evolves from the initial configuration by firing state machine transitions. Hence each step is the firing of a set of state-machine transitions. The next configuration is computed by applying the updates corresponding to the effect of each fired transition. The rule is structured in four sequential steps:

1. Collecting enabled transitions
2. Computing maximal sets of non-conflicting transitions
3. Firing the sets of transitions
4. Applying updates generated by the firing of transitions.

1.6 Related Work

Some formal work around UML and OCL aims to clarify the notation [FEL97]. Other work focuses on the simulation of UML models [CRS04]. We concentrate on verification methods.

Theorem-proving Theorem-proving is a formal method where it is proved that desired properties are satisfied by a system. The system is described in a declarative way, i.e. as a first-order or higher-order logical formula. The property is described in the same formalism. Support tools, called theorem-provers, allow the proving process to be partially mechanized. Automated proof strategies may be applied. The tool generates proof-obligations, i.e. residual proof that has to be achieved manually. The main drawback of the theorem-proving approach is that it requires highly trained personnel to verify a model.

The work of Traoré is a good example of the theorem proving approach applied to UML specifications. In [Tra00], he develops a UML semantics in PVS, a second-order

theorem prover. UML class diagrams are mechanically translated in the formalism of the prover. Properties are then stated and proved in that formalism.

In [KC00], Kim and Carrington develop a similar framework for the Z formal method. In [ML02], Marcano and Levy propose a verification framework based on the B formal method.

Perhaps the best effort at formal verification using OCL constraints is the Key project [ABB+03]. The system is specified as a UML model, which is mapped to a Java CARD program (a subset of Java). Desired properties are expressed as OCL constraints. The tool uses a dynamic logic (an extended Hoare calculus) semantics of Java CARD to generate proof obligations. A theorem prover is provided to prove the properties on the generated Java CARD programs.

In [RWH01], Reus et al. propose a similar approach. A specialized Hoare calculus is developed for proving OCL constraints on Java programs. In [BW02], Brucker and Wolff propose a formal semantics of OCL constraints in the Isabelle/HOL [NPW02] theorem prover.

Model-checking Many model-checking frameworks have been proposed for the verification of UML models. They use the same scheme. UML diagrams are compiled in the input language of a known model-checker. This means the UML statechart semantics is restricted to guarantee finiteness. Then, temporal properties are expressed in the specification language of that model-checker. The main drawback of such an approach is that the designer has to be familiar with the model-checker. Inversely, allowing OCL constraints to be verified is closer to the average designer experience.

A good example of such an approach is the work of Compton et al. presented in [CGHS00]. It describes a UML model-checker based on an ASM semantics of state-

charts by Börger [BCR00]. UML diagrams are compiled in an ASM specification, which in turn is translated into the input language of SMV [McM92].

The same approach has also been applied for the model-checkers SPIN [E. 98, LP99, GHP97] and JACK [GLM99].

Finally, in [Jür02], Jurjens extends the UML semantics of Börger to include cryptographic primitives. The approach is focused on the specification of security properties. As it is based on ASM, one may expect a model-checking framework to be developed around those extensions. For now, it supports only manual reasoning.

The semantics of this thesis brings the following extensions to the semantics found in related work:

- Taking into account three kinds of diagrams
- Formalization of a static semantics
- Integration of OCL expressions
- Explicit representation of objects and method instances
- Support for inheritance and object creation.

To our best knowledge, there is no available toolset for the model-checking of OCL constraints on UML models. There is some interesting work, however, on the development of a temporal logic semantics for OCL constraints.

In [OMG02], a formal semantics of OCL constraints is given. It states that an invariant is satisfied if the OCL expression is satisfied on every configuration of the model. Method preconditions and postconditions are satisfied if every pair of configurations corresponding to a method instance satisfies the corresponding OCL expressions.

These definitions have two major shortcomings with regard to model-checking: first, they do not state how the configurations are computed and, second, they do not state how the pair of configurations are collected.

In [BFS02], Bradfield et al. propose an approach to deal with the second shortcoming. The idea is to use μ-formulas to collect the appropriate configurations on which the OCL expressions of a constraint must be evaluated. To do so, the atomic properties of μ-calculus are replaced by OCL expressions. However, the semantics of OCL expressions is left unspecified. Distefano uses the same approach in [DKR00] but goes further by giving a semantics to a fragment of OCL expressions. The semantics is given on an abstract description of a UML model execution graph.

1.7 Outline

This thesis is structured as follows.

Chapter 2 We introduce the Abstract State Machine formalism. We explain how an ASM vocabulary specifies the structure of a state. We define the notion of transition rule and explain how it allows the ASM to evolve. We give a formal definition of an ASM execution graph. We propose observation operators that are used in the formal definition of the UML model execution graph and OCL constraints semantics. We enumerate conventions used to specify ASMs. Some details are relegated to Appendix A.

Chapter 3 We present a high-level description of the UML model semantics. We use the notion of ASM vocabulary to specify the structure of a UML model configuration. We sketch the static semantics of UML models. We explain how the step semantics is

captured through an ASM transition rule. We formally define the UML model execution graph. We explain how behavior inheritance is formalized and detail the semantics of actions. The complete semantics is given in Appendix B.

Chapter 4 We present the semantics of OCL expressions and OCL constraints. We explain how an OCL expression is evaluated on a UML model configuration as an external function and how atomic properties are stored in the configuration. We give a formal semantics to OCL constraints in terms of the μ-calculus temporal logic. We explain how atomic properties are evaluated during the computation of the model execution graph. Again, some details are relegated to Appendix C.

Chapter 5 We explain the design of a prototype tool based on the current thesis. This tool was implemented by a development team led by the author at the CRAC (Conception et Réalisation des Applications Complexes[2]) laboratory of École Polytechnique de Montréal.

Chapter 6 We summerize achievements of this thesis. We enumerate issues about the current semantics and give possible solutions. We explain the limitations of the current prototype and discuss future works that should increase its robustness and usability.

[2]Design and Implementation of Complex Applications

CHAPTER 2

ABSTRACT STATE MACHINES

This chapter describes the ASM formalism. An introductory example illustrates important notions: vocabulary, state, term and transition rule. These notions are then defined formally along with the notion of execution graph. We then propose observation operators that we later use to define the UML model and OCL constraints semantics. Finally, we explain how an ASM is specified and enumerate conventions used in the following chapters. Complementary definitions can be found in Appendix A.

2.1 Introductory Example

This section introduces the ASM formalism with a simple example. It models a system where an agent produces messages while another consumes and stores them.

Vocabulary An ASM vocabulary specifies how states are structured. It consists of a collection of sorts and a set of function signatures. Sorts are used to differentiate

values. The following sorts identify that there are three kinds of values in our example: agents, messages and message queues:

sort $Agent$
sort Msg
sort $Queue = \mathcal{L}(Msg)$

The latter is a constructed sort, which contains lists of messages. Other constructors include sets, tuples, maps and stacks.

Functions are used to store values within a state. The following functions are used in the example:

static	AGENTS	:	$\mathcal{S}(Agent)$
dynamic	q	:	$Queue$
dynamic	r	:	$Msg \rightarrow Bool$

The first two respectively hold the set of agents involved in the example and the message queue used for communication. The third one is used to indicate whether a message is known to the receiver agent.

There are three kinds of functions: static, dynamic and external. Static functions are invariant throughout the evolution of the ASM. Dynamic functions allow the ASM to evolve. For example, the function q will be updated as messages are exchanged. External functions are defined in terms of static and dynamic functions. Hence, their value may evolve as dynamic functions evolve. This mechanism is later used to integrate the evaluation of OCL expressions into the UML model semantics. The evaluation is done with respect to a fixed definition that fetches values in the UML model configuration. External functions are also used to add operators to constructed sorts. For example, the following external functions are used to manipulate message queues:

external $\_@\_ : Queue \times Queue \rightarrow Queue$
external $\langle\_\rangle : Msg \rightarrow Queue$
external $\texttt{hd} : Queue \rightarrow Msg$
external $\texttt{tail} : Queue \rightarrow Queue$

These functions are assumed to be part of the vocabulary by virtue of the fact that *Queue* is a constructed sort. Their interpretation corresponds to the usual interpretation of list operators.

Initial State An ASM state defines a carrier set for each sort and assigns an interpretation to each function. The carrier set contains every possible element of a given sort and is infinite in general. A function interpretation is a (potentially) infinite enumeration of the values taken by a function. The initial state of our example is defined as follows:

$$\begin{array}{lcll} \text{AGENTS} & = & \{\text{SND}, \text{RCV}\} & \\ q & = & \langle\rangle & \\ r(m) & = & \bot & \textbf{for all } m \textbf{ in } Msg \end{array}$$

The $\bot$ symbol represents the undefined element, which is assumed present for each sort.

Transition Rule The evolution of the ASM is specified by the following transition rule:

$\epsilon_{a \in \text{AGENTS}}$:
  case a **of**
    SND$\longrightarrow$
      $\partial_{m:Msg} : q := q@\langle m \rangle$
    RCV$\longrightarrow$
      $q \neq \langle\rangle$
        ? $\parallel$ $q := \texttt{tail}(q)$
          $\parallel$ $r(\texttt{hd}(q)) := \textbf{true}$

First, an agent is chosen non-deterministically. If it is the sender agent, a new message is created and appended to the queue. If it is the receiver agent, a message is dequeued (given the queue is not empty). The message becomes *known* to the receiver as indicated by the update of the function r. In general, a rule comprises variables, terms, predicates and subrules.

Variables are created and bound in rules. For example, the rule "$\partial_{m:Msg}$:" introduces the variable m and binds it to a fresh message. The variable is available in the subsequent rules and terms. Terms allow functions to be applied. For example, the term "$q@\langle m \rangle$" is the application of the concatenation function to the terms "q" and "$\langle m \rangle$". Predicates are special terms that always evaluate to a boolean value. They are used to parameterize some rules. For example, $q \neq \langle\rangle$ is a predicate that decides if the **?** clause is evaluated.

Rules serve two purposes. First, the update rule allows us to specify how dynamic functions evolve. For example, "$r(\mathtt{hd}(q)) := \mathbf{true}$" is an update that will assign the boolean value **true** to the function "r" at location "$\mathtt{hd}(q)$". Other rules are used to model the control flow of the system or to parameterize updates. For example, the choose rule (e.g. $\epsilon_{a \in \text{AGENTS}}$) allows us to make a non-deterministic choice before evaluating the remainder of the rule.

Execution From a given state, the next states are computed by evaluating the transition rule. This yields a collection of update sets. Each update describes how a dynamic function will take a new value for a given location. The updates of an update set are applied in no particular order, which means they have to be consistent. For example, "$q := \langle\rangle$" and "$q := \langle m \rangle$" would not be allowed in the same update set. Figure 2.1 gives an example of a path in the execution graph.

First, the transition rule is evaluated in the initial state. The sender agent is

$$
\begin{array}{lcc|c|c|c|c}
q & = & \langle\rangle & \langle m_1\rangle & \langle\rangle & \langle m_2\rangle & \\
r & = & m_1 \mapsto \bot & m_1 \mapsto \bot & m_1 \mapsto \textbf{true} & m_1 \mapsto \textbf{true} & \\
 & & m_2 \mapsto \bot & m_2 \mapsto \bot & m_2 \mapsto \bot & m_2 \mapsto \bot & \ldots \\
 & & m_3 \mapsto \bot & m_3 \mapsto \bot & m_3 \mapsto \bot & m_3 \mapsto \bot & \\
 & & [\ldots] & [\ldots] & [\ldots] & [\ldots] &
\end{array}
$$

Figure 2.1: Example of an Execution Path

selected by the choose rule. The fresh message m_1 is created by the "$\partial_{m:Msg}$:" rule and appended to q by the "$q := q@\langle m\rangle$" update rule. This yields the second state. The transition rule is evaluated again. The receiver agent is selected by the choose rule. As the queue is not empty, the **?** clause is evaluated, yielding the following updates: $q \mapsto \langle\rangle$ and $r(m_1) \mapsto \textbf{true}$. This illustrates how the updates "$q := \texttt{tail}(q)$" and "$r(\texttt{hd}(q)) := \textbf{true}$" are not evaluated in sequence. If they were, the term "$\texttt{hd}(q)$" in the second update would be undefined. The third transition corresponds to the selection of the sender agent again.

2.2 State

Definition 2.1 (ASM Vocabulary) An ASM vocabulary Υ is a couple (S, Ω), where:

- $S = (S_i)_{i \in I}$ is a collection of sorts
- Ω is a set of function signatures of the form $f : S_1 \times \ldots \times S_k \rightarrow S_n$.

We assume that Ω is partitioned into $\Omega_{\texttt{stat}}$, $\Omega_{\texttt{dyn}}$ and $\Omega_{\texttt{ext}}$, respectively static functions, dynamic functions and external functions.

For each sort S_i, we assume a static nullary function $\bot : S_i$ (interpreted as the undefined element). Finally, we assume a sort $Bool \in S$ representing booleans and

two static nullary functions **true** : *Bool* and **false** : *Bool*.

Definition 2.2 (ASM State) An ASM state $\mathfrak{A}$ is a couple $(S^{\mathfrak{A}}, \Omega^{\mathfrak{A}})$ over a vocabulary $\Upsilon = (S, \Omega)$, where:

- $S^{\mathfrak{A}} = (S_i^{\mathfrak{A}})_{i \in I}$ is a collection of infinite carrier sets
- $\Omega^{\mathfrak{A}}$ is a set of function interpretations of the form $f^{\mathfrak{A}} \subseteq (S_1^{\mathfrak{A}} \times \ldots \times S_k^{\mathfrak{A}}) \times S_n^{\mathfrak{A}}$.

For each sort S_i, we assume that the carrier set $S_i^{\mathfrak{A}}$ contains the element $\perp_{S_i}^{\mathfrak{A}}$. The interpretation of $\perp : S_i$ is always $\perp_{S_i}^{\mathfrak{A}}$. In addition, we assume that $\mathbf{true}^{\mathfrak{A}} \neq \mathbf{false}^{\mathfrak{A}}$.

2.3 Transition

Definition 2.3 (Terms) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$ and a set of variables $v_1 : S_1, \ldots, v_n : S_n$. The set of terms over Υ, written Δ, is defined inductively as follows:

- $v_1 : S_1, \ldots, v_k : S_k \in \Delta$
- $f : S_i \in \Omega$, then $f : S_i \in \Delta$
- If $t_1 : S_1, \ldots, t_k : S_k \in \Delta$ and $f : S_1 \times \ldots \times S_k \rightarrow S_n \in \Omega$, then
  - $f(t_1, \ldots, t_k) : S_n \in \Delta$.

The evaluation of a term is defined in Section A.1.

Definition 2.4 (Predicates) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$. Let t range

over the boolean terms over Υ. The set of predicates over Υ (ranged over by Φ, Ψ) is inductively defined according to the following syntax.

$$\Phi ::= t \mid \neg\Phi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \Phi \Rightarrow \Psi \mid \Phi \Leftarrow \Psi \mid \Phi \Leftrightarrow \Psi \mid \exists_{x:S_i} \cdot \Phi \mid \forall_{x:S_i} \cdot \Phi$$

The evaluation of predicates is as usual. For definiteness, we give their semantics in Section A.3. It relies on the notion of domain, defined in Section A.2.

Definition 2.5 (Transition Rule) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$. Let Φ range over the predicates over Υ. The set of possible transition rules over Υ (ranged over by Π) is defined according to the following syntax.

$$\Pi ::= \mathsf{o} \mid f(\vec{t}) := s \mid \Phi \ ? \ \Pi : \Pi \mid \Pi \parallel \Pi \mid \Sigma_{x:S_i} \cdot \Phi \colon \Pi \mid \epsilon_{x:S_i} \cdot \Phi : \Pi \mid x := t : \Pi$$

The informal meaning of the other rules is given below.

Skip	o	No effect
Update	$f(\vec{t}) := s$	Updates the dynamic function f at location $\vec{t}$ to the value of s
Conditional	$\Phi \ ? \ \Pi_1 : \Pi_2$	Π_1 is evaluated if Φ is true, else Π_2 is evaluated
Composition	$\Pi_1 \parallel \Pi_2$	Both Π_1 and Π_2 are evaluated
Σ-Composition	$\Sigma_{x:S_i} . \Phi : \Pi$	Π is evaluated for every value of x that renders Φ true
Choose	$\epsilon_{x:S_i} . \Phi : \Pi$	A value for x that renders Φ true is chosen and Π is evaluated
Let	$x := s : \Pi$	x is given the value of s and Π is evaluated

The formal semantics of transition rules is given in Section A.4. Note that the rule $\partial_{x:S_i} : \Pi$ used in the example is in fact a syntactic shortcut. This is explained in Section A.6.

2.4 Execution Graph

Formally, an Abstract State Machine is defined as follows.

Definition 2.6 (Abstract State Machine) An ASM M is a triple $(\Upsilon, \Pi, \mathfrak{A})$, where:

- Υ is a vocabulary
- Π is a transition rule over Υ
- $\mathfrak{A}$ is a state over Υ (the initial state).

From a given state, the evaluation of the transition rule yields a collection of update sets. The formal rule semantics is explained in Section A.4. It relies on the evaluation of terms given in Section A.1 and the predicate semantics given in Section A.3.

Definition 2.7 (Update Set) Consider an ASM $M = (\Upsilon, \Pi, \mathfrak{A})$ and $\mathfrak{B}$ a state over Υ. The evaluation of Π in state $\mathfrak{B}$, written $[\![\Pi]\!]^{\mathfrak{B}}$ yields a collection $(U)_{i \in I}$ of update sets. Every update set U_i contains elements of the form $(f, (a_1, \ldots, a_k), b)$, where:

- $f : S_1 \times \ldots \times S_k \rightarrow S_n$,
- $a_1 : S_1^{\mathfrak{A}}, \ldots, a_k : S_k^{\mathfrak{A}}$ and
- $b : S_n^{\mathfrak{A}}$.

Definition 2.8 (Update Set Consistency) Consider an ASM vocabulary Υ and a state $\mathfrak{A}$ over that vocabulary. An update set U is consistent if:

$$\forall u_1, u_2 \in U.\ u_1 = (f, (a_1, \ldots, a_k), b) \ \wedge \ u_2 = (f, (a_1, \ldots, a_k), c) \ \Rightarrow \ b = c$$

Definition 2.9 (Applying an Update Set) Consider an ASM vocabulary Υ and a state $\mathfrak{A}$ over that vocabulary. Let U be a consistent update set. The result of applying U to $\mathfrak{A}$ is a new state $\mathfrak{B}$ (written $\mathfrak{A} \xrightarrow{U} \mathfrak{B}$), such that:

- For every dynamic function $f \in \Omega$, $f^{\mathfrak{B}}$ is defined as follows:

$$f^{\mathfrak{B}}(\vec{a}) = \begin{cases} b & \text{if } ((f, \vec{a}), b) \in U \\ f^{\mathfrak{A}}(\vec{a}) & \textit{otherwise} \end{cases}$$

- For every external function $f \in \Omega$, $f^{\mathfrak{B}}$ is consistent with the external definition of f and the updated interpretations of the dynamic functions of Ω.

Definition 2.10 (Execution Graph) Consider the ASM $M = (\Upsilon, \Pi, \mathfrak{A})$. Its execution graph Θ_M is a triple $(\mathcal{S}, \rightarrow, \mathfrak{A})$, where $\mathcal{S}$ and $\rightarrow$ are defined inductively as follows:

- $\mathfrak{A} \in \mathcal{S}$
- If $\mathfrak{B} \in \mathcal{S}$, $U \in [\![\Pi]\!]^{\mathfrak{B}}$, U is consistent and $\mathfrak{B} \xrightarrow{U} \mathfrak{C}$, then:
  - $\mathfrak{C} \in \mathcal{S}$
  - $(\mathfrak{B}, \mathfrak{C}) \in \rightarrow$.

Hence, $\mathcal{S}$ contains the states of the graph, $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation and $\mathfrak{A}$ is the initial state.

2.5 Observations

State Restriction We define a state restriction operator that is used to formally define a UML model execution graph in Chapter 3.

Definition 2.11 Consider a state $\mathfrak{A} = (S^{\mathfrak{A}}, \Omega^{\mathfrak{A}})$ over the $\Upsilon = (S, \Omega)$ vocabulary and another vocabulary $\Upsilon' = (S', \Omega')$ such that $S' \subseteq S$ and $\Omega' \subseteq \Omega$.

The restricted state $\mathfrak{A}_{|\Upsilon'}$ is the pair $(S^{\mathfrak{A}'}, \Omega^{\mathfrak{A}'})$ such that:

- $S^{\mathfrak{A}'} = (S_i^{\mathfrak{A}})_{S_i \in S'}$
- $\Omega^{\mathfrak{A}'} = \{f^{\mathfrak{A}} \mid f \in \Omega'\}$.

Graph Observation We define an execution graph observation criterion to be used in Chapter 4.

Definition 2.12 Given an execution graph Θ and a first-order predicate Φ on the vocabulary of Θ, the observation of Θ through Φ, written $\Theta_{|\Phi}$, is defined as follows:

- $\mathfrak{A}_0 \rightarrow \mathfrak{A}_n \in \Theta_{|\Phi}$ iff there is some $\mathfrak{A}_0 \rightarrow \mathfrak{A}_1 \rightarrow \ldots \rightarrow \mathfrak{A}_n \in \Theta$ such that Φ is **true** in $\mathfrak{A}_0$ and $\mathfrak{A}_n$ and **false** in $\mathfrak{A}_i$ for any $1 \leq i < n$.

2.6 Specifications

This sections lists conventions that we use to specify vocabularies and transition rules throughout this thesis.

2.6.1 Constructed Sorts

By default, a sort contains simple elements. As illustrated in the example, we allow constructed sorts to be specified. Figure 2.2 lists the constructors considered. Constructed sorts come with the usual functions, defined as external functions. These are listed in Section A.5.

$\mathcal{S}(A)$	Sets of elements of A
$\mathcal{L}(A)$	Lists of elements of A
$\mathcal{R}(A)$	Relations over elements of A
$\mathcal{T}(A_1, \ldots, A_n)$	n-tuples in $A_1 \times \ldots \times A_n$
$A_1 \mid \ldots \mid A_n$	Elements of $A_1 \cup \ldots \cup A_n$
$A_1 \mapsto A_2$	Finite mappings from A_1 to A_2
A^*	Stacks of elements of A

Figure 2.2: Sort Constructors

Most of these external functions are conventional. The functions "$.$", "$\oplus$" and "`buildMap`" ought to be explained. The first fetches the n^{th} element of a list. For example, the term "$\langle a, b, c\rangle \,.\, 2$" will evaluate to b. The second allows us to update a mapping. For example, the term "$\{a \mapsto 1, b \mapsto 2\} \oplus \{b \mapsto 3, c \mapsto 4\}$" will evaluate to $\{a \mapsto 1, b \mapsto 3, c \mapsto 4\}$. The last one builds a mapping from two lists. For example, the term "`buildMap`$(\langle a, b\rangle, \langle 1, 3\rangle)$" will evaluate to $\{a \mapsto 1, b \mapsto 3\}$.

2.6.2 Syntactic Sugar

Shortcuts

Some shorthand notations are listed below.

Long Form		**Short Form**	
$t = t_1$			
$\quad ?\ \Pi_1$		**case** t **of**	
$\quad :\ t = t_2$		$\quad t_1 \rightarrow \Pi_1$	
$\quad\quad ?\ \Pi_2$	$\mapsto$	$\quad t_2 \rightarrow \Pi_2$	
$\quad\quad :\ t = t_3$		$\quad t_3 \rightarrow \Pi_3$	
$\quad\quad\quad ?\ \Pi_3$		$\quad \ldots$	
$\quad\quad\quad \ldots$			
$\epsilon_{x:A} \cdot x \in A : \Pi$	$\mapsto$	$\epsilon_{x \in \mathrm{F}} : \Pi$	**with** $\mathrm{F} : \mathcal{S}(A)$
$\forall_{x:A} \cdot x \in A : \Pi$	$\mapsto$	$\forall_{x \in \mathrm{F}} : \Pi$	**with** $\mathrm{F} : \mathcal{S}(A)$
$\Phi\ ?\ \Pi : \circ$	$\mapsto$	$\Phi\ ?\ \Pi$	
$\exists_{x_1:S_1} \cdot \Phi_1 \wedge \exists_{x_2:S_2} \cdot \Phi_2$		$\epsilon_{x_1:S_1,\, x_2:S_2} \cdot \Phi_1\ ;\ \Phi_2 :$	
$\quad ?\ \epsilon_{x_1:S_1} \cdot \Phi_1 : \epsilon_{x_2:S_2} \cdot \Phi_2 : \Pi_1$	$\mapsto$	$\quad ?\ \Pi_1$	
$\quad :\ \Pi_2$		$\quad :\ \Pi_2$	

Named Rules and Predicates Naming rules and predicates allows us to break long rules into smaller, more readable, rules. For example, the transition rule of the introductory example may be broken up as shown by Fig. 2.3.

$\epsilon_{a \in \text{AGENTS}}$:
case a **of**
SND $\rightarrow$
$\partial_{m:Msg} : q := q@\langle m \rangle$
RCV $\rightarrow$
$\neg \Phi_{emtpyQueue}(q)$
$?\ \Pi_{\texttt{dequeueMsg}}(q)$
$:\ \circ$

$\Phi_{\texttt{emptyQueue}}(q) \equiv$
$\quad q = \langle\rangle$

$\Pi_{\texttt{dequeueMsg}}(q) \equiv$
$\quad \| \ q := \texttt{tail}(q)$
$\quad \| \ r(\texttt{hd}(q)) := \textbf{true}$

Figure 2.3: Example of Named Rules

Distinguished Elements A sort must sometimes contain distinguished elements bound to static nullary functions. Figure 2.4 gives an example of such a case along with the abbreviated form.

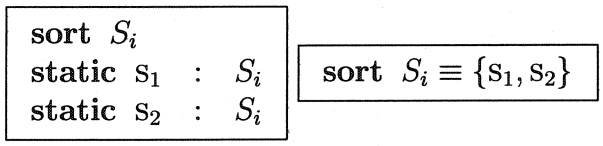


Figure 2.4: Example of Distinguished Elements

Named Tuples To simplify the manipulation of tuples, we name the positions of a tuple. Figure 2.5 gives an example of a named tuple. Given an element $t : \mathcal{T}(n_1 : A_1, \ldots, n_n : A_n)$, the term "$t.n_i$" extracts the element in the i^{th} position.

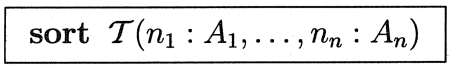


Figure 2.5: Example of a Named Tuple

2.6.3 Control Flow Graphs

In Chapter 3 and Appendix B, we use control flow graphs to simplify rule specifications. The approach is similar to Montage [KP97a], albeit simpler. For example, imagine we wish to change the control flow of the introductory example to alternate deterministically between the sender and receiver agents. The control graph is depicted in Fig. 2.6 along with the vocabulary extensions it induces. Note that, according to the graph, the initial value of *ctr* is SND.

The transitions rule of the example are now as follows:

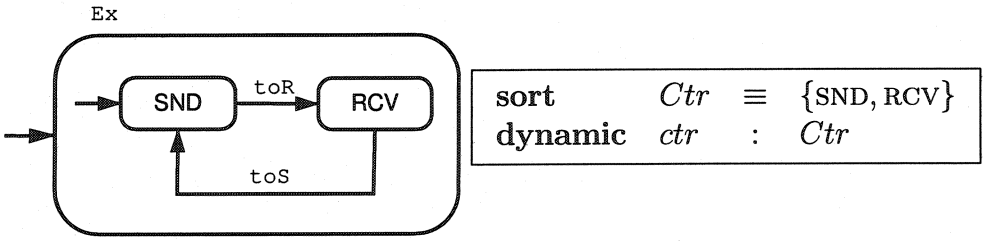
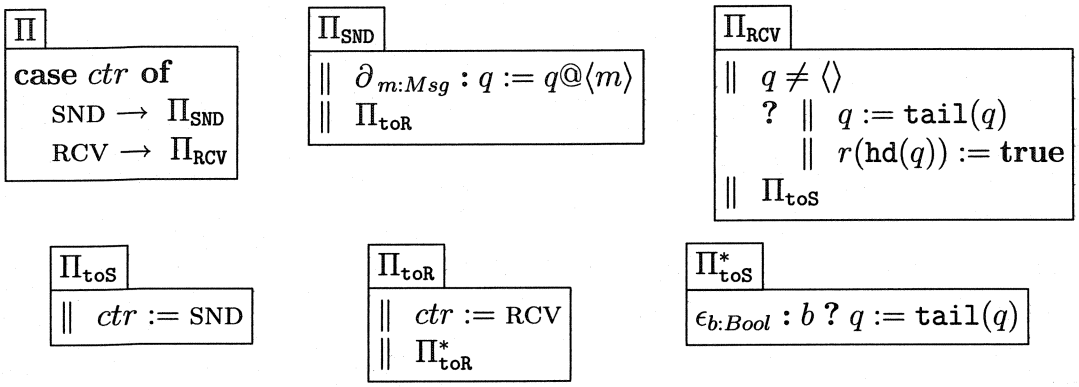


Figure 2.6: Example of a Control Flow Graph



The main rule Π and subrules $\Pi_{\texttt{toS}}$ and $\Pi_{\texttt{toR}}$ are induced from the graph. Note how the rule $\Pi_{\texttt{toR}}$ calls the user-defined rule $\Pi^*_{\texttt{toR}}$. This allows us to add user-defined behavior to transitions. In our example, we use this mechanism to model the fact that a message may be lost.

CHAPTER 3

UML MODEL SEMANTICS

This chapter presents a high-level description of the formal UML model semantics. We begin by detailing the formalization rationales and completing the informal description of the UML fragment considered. We explain how we use the ASM formalism to capture the concepts of configuration and step. We sketch the formalization of a model's static semantics. We explain how behavior inheritance is captured. We detail the semantics of actions. Finally, we give a formal definition of a UML model execution graph and define safety properties for the semantics. The complete list of formal definitions is given in Appendix B.

3.1 Formalization Rationales

As explained in [OMG03b], the UML statechart semantics is meant to be an object-oriented extension of Harel's statechart semantics. We formalized statechart semantics by directly adapting a fragment of Harel's statechart semantics, as given in [HN96]. The object-oriented extensions are supported by selecting a fragment

of the triggers and actions proposed in [OMG03b]. The static semantics is directly inspired by the ASM Java semantics of Stark et al. [SSB01].

Statechart The notion of OR-states and AND-states is supported. We also adopt the idea that transitions effects are applied all at once. We formalize the notion of transition conflict: two transitions are in conflict when they cannot be fired in the same step. The notion of transition priority is dropped. We extend the notion of conflict to include a dynamic computation of conflicts. For example, transitions resulting in incoherent assignation to a field are detected as conflictual.

We apply some simplifications. We do not consider compound transitions. In particular, this means that a transition exiting the initial state of a composite state is not fired in the same step as the transition activating its parent. Instead of *pseudo* states, initial states are considered as normal states. They are automatically activated when the direct parent state is activated. We do not consider actions associated with entering or exiting a state. We do not consider actions associated with a state being active. Finally, history states are dropped.

Classes We consider associations between classes as a field of the client class. The type of the field is always a list of the supplier class. Multiplicity is supported, but no other special meaning of the association relation is taken into account, e.g. agglomerates are not supported. Finally, we do not consider interfaces.

Methods Since transitions are fired in no particular order, method call actions are always in conflict. Otherwise, firing two transitions may result in an inconsistent update of a calling stack. Consequently, there is at most one method call per step.

Threads Concurrency is modelled through a multi-threaded scheduler. The current thread is chosen non-deterministically at the beginning of the step computation. The current thread must be the same throughout the step computation since OCL expressions are evaluated according to an environment containing local variables, i.e. parameters of the currently executing method.

3.2 Considered Fragment

As explained in the introduction, we consider a UML model to be exactly one class diagram, one statechart diagram for each class and one object diagram.

Configuration A configuration of the model consists of a set of active signals and a set of objects. Each object is specified by its class and the values taken by its fields. If the object is a thread, it also possesses a calling stack. Finally, each object has a state-machine.

Step One step of the UML model is the firing of a set of state-machine transitions. The active states of an object's state-machine denote this object's local configuration. States allow us to model the basic control flow of the object. Actions are equivalent to instructions that the object can execute. Triggers are used to restrict when such instructions are executed, e.g. by which method.

States There are five kinds of states. Initial states are automatically activated when their parent state is entered. Final states automatically deactivate their parent state when entered. Simple states have no special meaning. Finally, there are two kinds of composite states: OR-states, which may contain only one active substate

and AND-states, which may contain only OR-states as substates. As long as the AND-state is active, every enclosed OR-state is also active. As soon as one enclosed OR-state is deactivated, the AND-state is also deactivated.

Transitions For a transition to be enabled, its source state must be active. The set of states deactivated and activated by the firing of a transition is statically computed. The definition depends on the notion of a transition scope. The scope of a transition is the lowest state (in the state hierarchy) that contains both the source state and the target state of the transition. When a transition is fired, the scope state is deactivated along with every state it encloses. Then, the scope state is reactivated along with the target state. The only exception is when the target state is a final state.

Triggers For a transition to be enabled, its trigger must correspond to an active event of the configuration. Figure 3.1 presents the possible triggers of the considered fragment.

$Trigger ::=$		
	$\star$	Empty trigger
|	$Signal$	Signal trigger
|	$Msig$	Method call
|	$\%Msig$	Method return
$Msig ::=$	$Meth(Param : Type, \ldots, Param : Type) : Type$	

Figure 3.1: Trigger Syntax

The empty trigger $\star$ is always satisfied. A signal trigger is satisfied if the corresponding signal is active in the configuration. A method call trigger is satisfied if that method is the currently executing method, i.e. if the method is on top of the

current thread calling stack. A method return trigger is satisfied if the method was the last to return.

Guards A guard is an OCL expression of boolean type. Section 4.1 gives the formal syntax and semantics of OCL expressions. The guard is evaluated on the current configuration and must evaluate to **true** for the transition to be enabled.

Actions Figure 3.2 gives the syntax of actions. A transition contains a list of actions. When a transition is fired, the list is iterated and the effect of every action is computed. A signal action activates the corresponding signal in the next configuration. An assignation action updates the value of a field. An object creation action adds a new object to a reference field. A new state-machine is created for that object. An object deletion action removes one object from a reference field. Another way of removing objects from a reference field is to use an assignation action, which reassigns the field to a subset of the referenced objects. A method call action places a new call on the current thread's calling stack. A method return action removes the current method from the current stack.

Action ::=
| *Signal*
| **new** *Field*
| **delete** *Field*
| *Field* = *OclExp*
| *OclExp* . *Meth* (*OclExp* , ... , *OclExp*)
| **super** . *Meth* (*OclExp* , ... , *OclExp*)
| % *MSig* [*OclExp*]

Figure 3.2: Action Syntax

Notice how the method signature is not given explicitly in method calls. Rather, is it elaborated according to the type of the OCL expressions. This is explained in

Section B.1.4. Furthermore, there are two kinds of method calls: one calls the first applicable method according to the inheritance relation, while the second calls the method of the superclass. This is part of the object-oriented behavioral inheritance mechanism.

3.3 Using the ASM Formalism

The static structure and the dynamic configuration of a UML model are captured by different ASM vocabularies, respectively Υ_{STAT} and Υ_{UML}. A third vocabulary, namely Υ_{AUX}, captures auxiliary functions. The step semantics is captured by an ASM transition rule, namely Π_{UML}.

Formally, a UML model configuration is an ASM state over the Υ_{UML} vocabulary. In order to compute the set of the next configurations, that state is extended with states over the Υ_{STAT} and Υ_{AUX} vocabularies. An ASM execution graph is computed from that extended state using the Π_{UML} rule. The final states of that graph contain the next configurations. Each such state is restricted to the Υ_{UML} vocabulary in order to extract the configuration. Transitions corresponding to the computed steps are added to the UML model execution graph. Figure 3.3 illustrates the approach. The process is iterated in order to compute the entire UML model execution graph. The formal definition of a UML model execution graph is given in Section 3.8.

The Υ_{STAT} vocabulary contains static information about the UML model. This includes the set of classes, the inheritance relation, the set of method signatures, etc. The Υ_{UML} vocabulary contains the UML model configuration itself. This includes objects, method instances, active states, etc. The Υ_{AUX} vocabulary contains auxiliary functions, e.g. functions to iterate through objects and transitions.

For each of these vocabularies, we define well-formedness constraints on the cor-

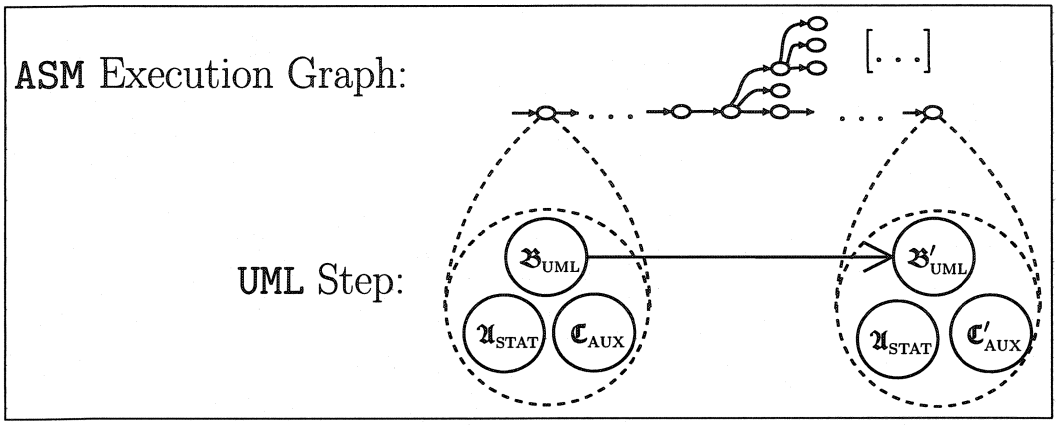


Figure 3.3: ASM Execution Graph and UML Step

responding ASM states. A well-formedness constraint for Υ_{STAT} corresponds to a syntactic correctness property. It insures that the UML model is well-formed, that statecharts are valid, that types are coherent, etc. A well-formedness constraint for Υ_{UML} refers to a valid UML configuration. For example, it insures that values are of the right type. Another type of well-formedness property for the Υ_{UML} vocabulary is constraints on the initial configuration of the model. A well-formedness property for Υ_{AUX} corresponds to the initial values that the auxiliary functions must take.

The static semantics is captured with the constraints on the Υ_{STAT} vocabulary and on the initial configuration. The constraints on the Υ_{UML} vocabulary capture the idea of type safety (see Section 3.9). The next three sections outline the vocabularies of the formal UML model semantics.

3.4 Υ_{STAT}

The Υ_{STAT} defines a data structure to which diagrams are compiled. It consists of a direct representation of the diagrams' syntax and a set of elaborated functions. The

complete specification of the $\Upsilon_{\texttt{stat}}$ vocabulary can be found in Section B.1.

3.4.1 Diagram Information

The UML model diagrams are captured using a set of simple and constructed sorts. Classes, objects and signals are simple sorts:

sort $Class$
sort Obj
sort $Signal$

States and transitions are captured by constructed sorts:

sort	$StateName$		
sort	$StateType$	$\equiv$	$\{\text{INIT}, \text{FINAL}, \text{SIMPLE}, \text{OR}, \text{AND}\}$
sort	$State$	$=$	$\mathcal{T}(n : StateName, ty : StateType)$
sort	$Trans$	$=$	$\mathcal{T}(n : TransName, src : State, \ldots, tgt : State)$

Fields are captured as follows:

sort	$FieldName$		
sort	$Field$	$=$	$\mathcal{T}(n : FieldName, ty : Type, \ldots)$
sort	$ClassField$	$=$	$\mathcal{T}(c : Class, f : Field)$

Method signatures are captured as follows:

sort	$MethName$		
sort	$ParamName$		
sort	$Param$	$=$	$\mathcal{T}(n : ParamName, ty : Type)$
sort	$MSig$	$=$	$\mathcal{T}(meth : MethName, args : \mathcal{L}(Param), \ldots)$
sort	$CMSig$	$=$	$\mathcal{T}(c : Class, msig : MSig)$

Triggers are captured as follows:

sort	$TriggerType$	$\equiv$	$\{\text{SIGNAL}, \text{METHOD} \ldots\}$
sort	$TSignal$	$=$	$\mathcal{T}(s : Signal, ty : TriggerType)$
sort	$TMsig$	$=$	$\mathcal{T}(msig : Msig, ty : TriggerType)$
	$[\ldots]$		
sort	$Trigger$	$=$	$TSignal \mid TMsig \mid \ldots$

Actions are captured as follows:

$$
\begin{array}{lll}
\textbf{sort} & \mathit{ActionType} & \equiv \{\text{SIGNAL, MCALL} \ldots\} \\
\textbf{sort} & \mathit{CallType} & \equiv \{\text{EXPLICIT, VIRTUAL, SUPER}\} \\
\textbf{sort} & \mathit{ASignal} & = \mathcal{T}(s : \mathit{Signal}, ty : \mathit{ActionType}) \\
\textbf{sort} & \mathit{AMCall} & = \mathcal{T}(rcv : \mathit{OclExp}, c : \mathit{Class}, msig : \mathit{Msig}, \\
 & & \quad cty : \mathit{CallType}, args : \mathcal{L}(\mathit{OclExp}), ty : \mathit{ActionType}) \\
 & [\ldots] & \\
\textbf{sort} & \mathit{Action} & = \mathit{ASignal} \mid \mathit{AMsig} \mid \ldots
\end{array}
$$

Static functions are used to actually store the diagrams. For example, the following functions hold the statechart diagrams of a UML model:

$$
\begin{array}{lll}
\textbf{static} & \text{STATES} & : \mathit{Class} \rightarrow \mathcal{S}(\mathit{State}) \\
\textbf{static} & \text{TRANS} & : \mathcal{S}(\mathit{Trans}) \\
\textbf{static} & \text{TOP} & : \mathit{Class} \rightarrow \mathit{State} \\
\textbf{static} & \text{CHILD} & : \mathit{State} \rightarrow \mathcal{S}(\mathit{State}) \\
\textbf{static} & \text{UP} & : \mathit{State} \rightarrow \mathit{State}
\end{array}
$$

3.4.2 Elaborated Information

Some important information is elaborated and stored in static functions. For example, the following functions capture elaborated information about statechart diagrams:

$$
\begin{array}{lll}
\textbf{static} & \text{CHILD}^* & : \mathit{State} \rightarrow \mathcal{S}(\mathit{State}) \\
\textbf{static} & \text{DEACTIVATES} & : \mathit{Trans} \rightarrow \mathcal{S}(\mathit{State}) \\
\textbf{static} & \text{ACTIVATES} & : \mathit{Trans} \rightarrow \mathcal{S}(\mathit{State}) \\
\textbf{static} & \text{INITSTATES} & : \mathit{Class} \rightarrow \mathcal{S}(\mathit{State})
\end{array}
$$

The function CHILD* returns all the children of a state (i.e. CHILD* is the transitive closure of CHILD). The functions DEACTIVATES and ACTIVATES indicate which states are deactivated or activated by a transition, respectively. The function INITSTATES indicates which states should be initially active.

The following functions are elaborated from the information of the class diagram:

static	$\preceq_h$	:	$\mathcal{R}(Class)$
static	FIELDS*	:	$Class \rightarrow \mathcal{S}(ClassField)$
static	METHOD*	:	$Class \rightarrow \mathcal{S}(MSig)$
static	$\preceq_o$	:	$\mathcal{R}(ClassMSig)$
static	LOOKUP	:	$Class \times ClassMSig \rightarrow Class$

The inheritance relation is computed from the direct inheritance relation present in the model. The $\preceq_o$ relation indicates whether a method overrides one of the method of its superclass. The static function FIELDS* returns every field of a class, including inherited fields. It is defined as follows:

Elaborated Function 3.1 (Fields of a Class) Given a class A, FIELDS*(A) is defined as follows:

$$\text{FIELDS}^*(A) = \{(B, f) \mid A \preceq_h B \ \wedge \ f \in \text{FIELDS}(B)\}$$

The complete list of elaborated functions, along with their definitions, can be found in Section B.1.2.

3.4.3 Well-formedness

We give two examples of well-formedness properties on states over the Υ_{STAT} vocabulary. The first property states that an OR-state may have only one initial state.

Well-formed Model 3.1 (OR-state Structure) An OR-state s is well-formed if:

$$!\exists\ s' \in (\text{CHILD}(s) \setminus \{s\})\ .\ s'.ty = \text{INIT}$$

The second property states that an AND-state may have only OR-states as direct children.

Well-formed Model 3.2 (AND-state Structure) An AND-state s is well-formed if:

$$(\text{CHILD}(s) \setminus \{s\}) \neq \emptyset \textbf{ and } \forall\, s' \in (\text{CHILD}(s) \setminus \{s\}) \,.\, s'.ty = \text{OR}$$

The complete list of well-formedness constraints over the $\Upsilon_{\texttt{stat}}$ vocabulary can be found in Section B.1.3.

3.5 Υ_{UML}

The $\Upsilon_{\texttt{uml}}$ vocabulary holds the model configuration. It is also a direct representation of the model's object diagram. A complete specification of the vocabulary can be found in Section B.2.1.

State-machines An UML model configuration contains active states for each state-machine and the set of active signals:

dynamic $signals$: $\mathcal{S}(Signal)$
dynamic $actstates$: $Obj \rightarrow \mathcal{S}(State)$

Objects In order to represent objects, we need a sort to specify the values that a field may take. A value is either a boolean, an integer, an object or a list of these simple values:

$$
\begin{array}{lll}
\textbf{sort} & Int & \equiv \{\ldots, -1, 0, 1, \ldots\} \\
\textbf{sort} & Bool & \equiv \{\text{TRUE}, \text{FALSE}\} \\
\textbf{sort} & SimpleVal & = Int \mid Bool \mid Obj \\
\textbf{sort} & ListVal & = \mathcal{L}(SimpleVal) \\
\textbf{sort} & Val & = SimpleVal \mid ListVal
\end{array}
$$

The function *heap* holds the configuration's objects. An object is a couple containing its class and the values taken by its fields:

$$
\begin{array}{lll}
\textbf{sort} & ObjEnv & = ClassField \mapsto Val \\
\textbf{sort} & Heap & = \mathcal{T}(c : Class, env : ObjEnv) \\
\textbf{dynamic} & heap & : Obj \rightarrow Heap
\end{array}
$$

Methods The function *cont* (shorthand for *execution context*) holds a calling stack for every thread of the configuration. Each frame of the stack contains the current method, a local variable assignation, the sender object and the receiver object:

$$
\begin{array}{lll}
\textbf{sort} & FrameEnv & = Var \mapsto Val \\
\textbf{sort} & Frame & = \mathcal{T}(mth : ClassMSig, env : FrameEnv, \\
 & & \quad snd : Obj, rcv : Obj) \\
\textbf{sort} & CallStack & = Frame^* \\
\textbf{dynamic} & cont & : Obj \rightarrow CallStack \\
\textbf{dynamic} & returns & : Obj \rightarrow MSig
\end{array}
$$

3.5.1 Well-formedness

We give an example of a well-formedness constraint over the Υ_{UML} vocabulary.

Well-formed Configuration 3.1 (Environment) For every object $o : Obj$ such that $heap(o) \neq \bot$. Let C be that object's class (i.e. $C = heap(o).c$). Then, we must have:

$$\forall\, (D, f) \in \text{FIELDS}^*(C)\ .\ heap(o).env((D, f)) \neq \bot$$

It states that every field of an object, including inherited fields, has a value. The complete list of well-formedness constraints on the $\Upsilon_{\texttt{uml}}$ vocabulary can be found in Section B.2.2.

3.6 Υ_{AUX}

Additional vocabulary is necessary to capture the step semantics. The complete specification of the vocabulary can be found in Section B.3. We need sets containing enabled transitions and transitions to be fired. In addition, we use a boolean nullary function to indicate that the step computation is finished:

dynamic	$enabledSet$	:	$\mathcal{S}(\mathcal{T}(Obj, Trans))$
dynamic	$fireSet$	:	$\mathcal{S}(\mathcal{T}(Obj, Trans))$
dynamic	$step$	:	$Bool$

We also need functions to hold the current thread, current object, etc.:

dynamic	$\tilde{th}$	:	Obj	*Current thread*
dynamic	$\tilde{o}$	:	Obj	*Current object*
dynamic	$\tilde{t}$	:	$Trans$	*Current transition*
dynamic	$\tilde{a}$	:	$\mathcal{L}(Action)$	*Current actions*

In addition, an external function is defined in Chapter 4 for evaluating `OCL` expressions:

sort	Var		
sort	$OclEnv$	:	$Var \mapsto Val$
external	$[\![\_]\!]\_$	:	$OclExp \times OclEnv \rightarrow Val$

The function maps an `OCL` expression and a variable environment to a value. The variable environment contains the variable **self**, which is mapped to the current object and the local variables located on the top of the calling stack.

Finally, we need to hold a temporary configuration. For each function in the Υ_{UML}

vocabulary, we require that the Υ_{aux} vocabulary contain a counterpart. For example, the function $\overrightarrow{heap}$ is defined in the Υ_{aux} vocabulary and is required to hold the same information as the function *heap* at the beginning of the step computation.

3.6.1 Well-formedness

We give two examples of well-formedness constraints on the Υ_{aux} vocabulary. Both define the value that a function must take at the beginning of the step computation.

Additional Vocabulary 3.1 (Enabled Set)

$$enabledSet = \emptyset$$

Additional Vocabulary 3.2 (Fire Set)

$$fireSet = \emptyset$$

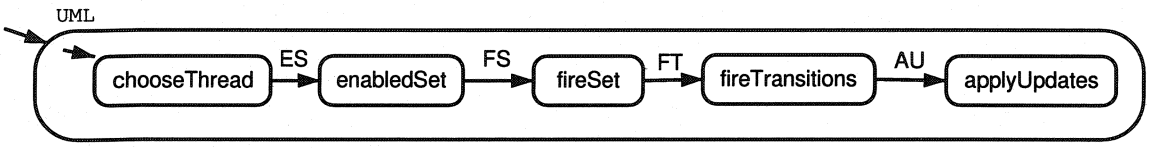
The complete specification of well-formedness constraints of the Υ_{aux} vocabulary is given in Section B.3.1.

3.7 Step

The basic step semantics is to accumulate enabled transitions and fire them all at once. As some transitions may be in conflict, a subset of non-conflicting transitions

is non-deterministically chosen. For a transition to be enabled, its actions must be allowed in the current configuration. Similarly, transitions are conflicting if their actions are in conflict.

The **ASM** rule $\Pi_{\texttt{uml}}$ capturing the step semantics has the following control flow:



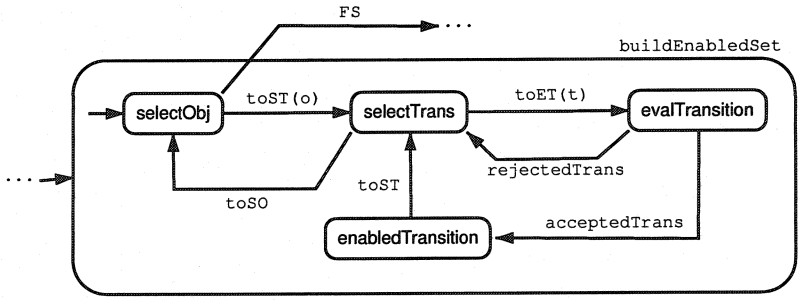
The next section describes the role of each subrule. Section 3.7.2 explains how object-oriented behavior inheritance is formalized. Section 3.7.3 details the semantics of actions. The complete specification of the transition rule is given in Section B.4.

3.7.1 Step Computation

Choosing a Thread The first subrule chooses the current thread of execution. It models the controller of a thread-based concurrency scheme. The choice is made non-deterministically as follows:

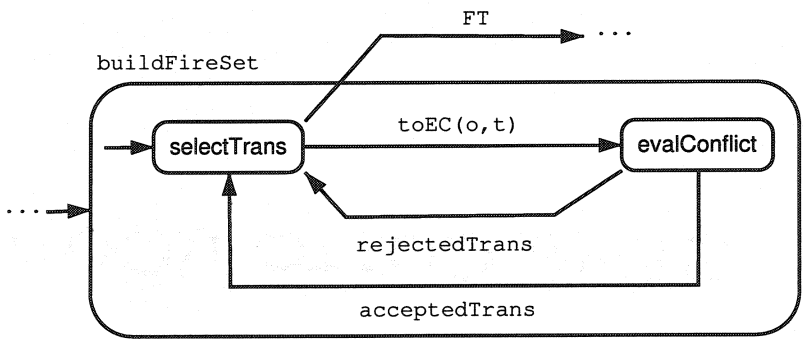
$$\epsilon_{o:Obj} \bullet heap(o).c \in \text{THREADS} :$$
$$\| \quad \tilde{th} := o$$

Collecting Enabled Transitions The second subrule collects enabled transitions by iterating objects and transitions:



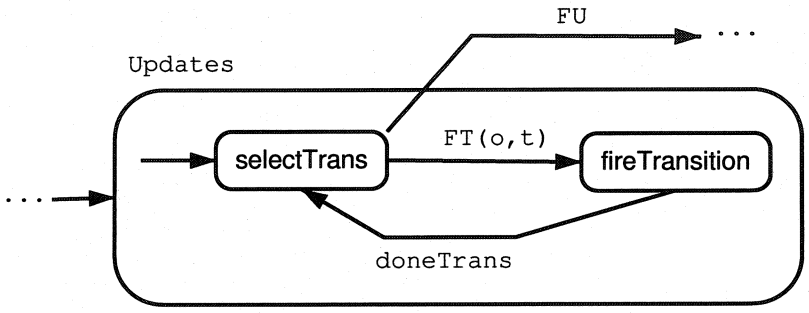
A transition is selected if it is enabled with respect to its source state and trigger. The rule $\Pi_{\texttt{evalTransition}}$ determines whether the transition is enabled with respect to its actions. A signal action can be fired in any configuration. For other actions, we require that the current object $\tilde{o}$ be on top of the current calling stack (as the receiver object). This is necessary because these actions will update either *heap* or *cont*. Intuitively, these actions are instructions of the currently executing method. Finally, object creation, object deletion and assignation actions must not violate the multiplicity of the field they update.

Selecting Transitions to Fire The third subrule consists of the non-deterministic choice of a (maximal) set of non-conflicting transitions. The fact that transitions are chosen non-deterministically guarantees that every possible set is considered. To guarantee that the set is maximal, we require that the rule $\Pi_{\mathbf{FT}}$ be called only when no more transitions can be added:



A transition is selected for addition only if it is not in conflict with a previously added transition (with respect to the states it deactivates). The rule $\Pi_{\texttt{evalConflict}}$ determines whether a transition can be added with respect to its actions. This allows us to enforce assignation coherence and, also, the fact that only one method call may occur in a step.

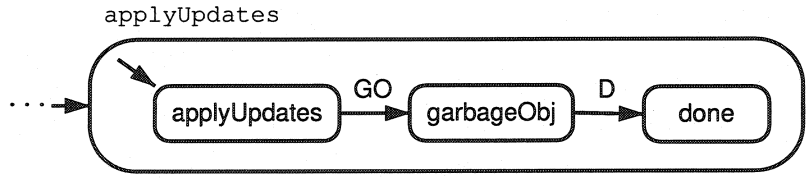
Firing Transitions The fourth subrule fires each transition:



When a transition is fired, its actions are iterated and fired according to the action semantics of Section 3.7.3. In addition, given a transition t belonging to the state-machine of object o, the active states are updated as follows:

$$\overrightarrow{actstates}(o) := (\overrightarrow{actstates}(o) \setminus \text{DEACTIVATES}(t)) \cup \text{ACTIVATES}(t)$$

Applying Updates Once every transition is fired, the next configuration is computed. This includes applying updates and garbaging objects:



Updates are applied according to the following rule:

$$
\begin{array}{l}
\| \quad signals := \overrightarrow{signals} \\
\| \quad \Sigma_{o:Obj} : actstates(o) := \overrightarrow{actstates}(o) \\
\| \quad \Sigma_{o:Obj} : heap(o) := \overrightarrow{heap}(o) \\
\| \quad cont(\tilde{th}) := \overrightarrow{cont}(\tilde{th}) \\
\| \quad returns(\tilde{th}) := \overrightarrow{returns}(\tilde{th})
\end{array}
$$

Objects deleted during step computation are garbaged as follows:

$$\| \quad \Sigma_{o:Obj} \cdot \neg \, \Phi_{\texttt{referencedObj}}(o) : heap(o) := \bot$$

The $\Phi_{\texttt{referencedObj}}$ predicate indicates whether a reference to an object exists , either in the heap or in a calling stack. Finally, rule $\Pi_{\texttt{done}}$ indicates that the step computation is finished by setting *step* to **true**.

3.7.2 Behavior Inheritance

With the appropriate static semantics, formalizing behavioral inheritance is straightforward. First of all, the state-machine of an object is the concatenation of its class statechart and its superclasses statecharts. This allows the object to execute inherited behavior. This is achieved by correctly defining the function INITSTATES.

We then define a mechanism to decide whether an object executes its own behavior or an inherited behavior. According to object-orientation, this is done when a

method call is fired. The function LOOKUP is used to select which class defines the behavior that should answer the call. That class is either the receiving object class or a superclass of the latter. We make sure the inherited behavior is ready to answer the call by reactivating the initial states using the static function INITSTATES.

Subsequently, we add the following condition for a transition t of object o to be enabled:

$$\begin{array}{ll} & \mathtt{top}(cont(\tilde{th})).rcv = o \\ \wedge & t.src \in \text{STATES}(\mathtt{top}(cont(\tilde{th})).cmsig.c) \end{array}$$

The predicate states that the transition is enabled if the object is currently executing. Furthermore, it verifies that the transition corresponds to a behavior inherited from the class selected to answer the call.

3.7.3 Action Semantics

Given an action a, its effects are computed according to its kind.

Signal Action The signal is simply added to the set of active signals:

$$\overrightarrow{signals} := \overrightarrow{signals} \cup \{a.s\}$$

Assignation Action The field environment of the corresponding object is updated:

$$
\begin{array}{l}
C := heap(\tilde{o}).c : \\
env := heap(\tilde{o}).env : \\
val := [\![a.exp]\!]_{\zeta} : \\
\quad \overrightarrow{heap}(\tilde{o}) := (C, env \oplus \{a.cf \mapsto val\})
\end{array}
$$

Note that *exp* is the OCL expression of the assignation action and ζ is the variable environment needed to evaluate the expression.

Object Creation The object creation action is fired as follows:

$$
\begin{array}{l}
\partial_{o':Obj} : \\
\quad C := a.cf.f.t : \\
\quad\quad \| \;\; env := \overrightarrow{heap}(\tilde{o}).env : \\
\quad\quad\quad\quad val := o' :: env(a.cf) : \\
\quad\quad\quad\quad\quad\quad \overrightarrow{heap}(\tilde{o}) := (C, env \oplus \{a.cf \mapsto val\}) \\
\quad\quad \| \;\; \overrightarrow{heap}(o') := \text{NEWOBJ}(C) \\
\quad\quad \| \;\; \overrightarrow{actstates}(o') := \text{INITSTATES}(C) \\
\quad\quad \| \;\; \Pi_{\texttt{createThread}}(C, o')
\end{array}
$$

First, a new element is added to the sort *Obj* using the sort extension mechanism of the ASM formalism. The new object is added to the current object's appropriate field. The new object is created and added to function *heap*. The static function NEWOBJ returns an object with default values for each field. Its initial states are added to function *actstates*. Finally, the rule $\Pi_{\texttt{createThread}}(C, o')$ verifies whether the new object is a thread and, if so, correctly updates the function *cont*.

Object Deletion Executing an object deletion action is straightforward. An object is selected from the object reference field specified in the object deletion action.

The reference is removed from the list and the object's field environment is updated. If it is the last reference to that object, it will be garbaged when updates are applied.

Method Call Action Firing a method call action is done as follows. First, the receiver object *rcv* and the method parameters *vals* are computed using the OCL expression evaluation function. If it is a virtual method call, the class C, which defines the behavior that answers the call, is $\text{LOOKUP}(heap(rcv).c, a.cmsig)$. If it is a super method call, that class C is $\text{LOOKUP}(\text{SUPER}(heap(rcv).c), a.cmsig)$. Once this information is computed, the action is fired according to the following rule:

$$\begin{array}{ll} \| & \overrightarrow{actstates(\tilde{o})} := \overrightarrow{actstates(\tilde{o})} \cup \text{INITSTATES}(C) \\ \| & env := \texttt{buildMap}(msig.p, vals) : \\ & \quad \overrightarrow{cont}(\tilde{th}) := \texttt{push}(cont(\tilde{th}), ((C, a.cmsig.msig), env, \tilde{o}, rcv)) \end{array}$$

It reactivates the initial states corresponding to the statechart that should answer the call. It pushes the method on the current thread calling stack. The external function `buildMap` builds the frame environment given a list of parameter names and a list of values.

Method Return Action If the action is a method return, the current executing method is popped. In addition, the returned value is stored. This is done by the following rule:

$$\begin{array}{ll} \| & \overrightarrow{cont}(\tilde{th}) := \texttt{pop}(cont(\tilde{th})) \\ \| & returns(\tilde{th}) := (a.msig, [\![a.exp]\!]_\varsigma) \end{array}$$

3.8 Execution Graph

Assuming the vocabularies Υ_{STAT}, Υ_{UML}, Υ_{AUX}, well-formedness properties for each vocabulary and the ASM transition rule Π_{UML}, we formally define the notion of UML step. The relation $\rightarrow^*$ is the transitive closure of the ASM transition relation. The ASM state $\mathfrak{E}_{|\Upsilon_{\text{UML}}}$ is the restriction of the state $\mathfrak{E}$ to the Υ_{UML} vocabulary (see Section 2.5).

Definition 3.1 (UML Step) Let $\mathfrak{A}, \mathfrak{A}'$ be two ASM states over the Υ_{UML} vocabulary. Let $\mathfrak{B}$ be a well-formed ASM state over Υ_{STAT}. Let $\mathfrak{C}$ be a well-formed ASM state over Υ_{AUX}. Let Π_{UML} be the ASM rule capturing the step semantics of UML and let $\rightarrow$ be the transition relation defined by Π_{UML} over $\Upsilon_{\text{STAT}} \cup \Upsilon_{\text{AUX}} \cup \Upsilon_{\text{UML}}$.

There is a UML step from $\mathfrak{A}$ to $\mathfrak{A}'$ (written $\mathfrak{A} \longrightarrow \mathfrak{A}'$) iff there exists an ASM state $\mathfrak{E}$ over $\Upsilon_{\text{STAT}} \cup \Upsilon_{\text{AUX}} \cup \Upsilon_{\text{UML}}$ such that:

- $(\mathfrak{A} \cup \mathfrak{B} \cup \mathfrak{C}) \rightarrow^* \mathfrak{E}$
- $step^{\mathfrak{E}} = \mathbf{true}$
- $\mathfrak{A}' = \mathfrak{E}_{|\Upsilon_{\text{UML}}}$.

Using that definition, we define the notion of UML execution graph.

Definition 3.2 (UML Execution Graph) An UML execution graph Λ is a triple $(\mathcal{U}, \mathfrak{A}_0, \longrightarrow)$, where:

- $\mathcal{U}$ is a set of configurations
- $\mathfrak{A}_0 \in \mathcal{U}$ is the initial, well-formed, configuration
- $\longrightarrow \subseteq \mathcal{U} \times \mathcal{U}$ is the step relation.

3.9 Safety Properties

The correctness of an operational semantics is usually proved with respect to a formal declarative description of the semantics. This is not possible for a UML semantics since the declarative description is informal. However, we will define two correctness properties: execution safety and type safety. The first one guarantees that the computation of a step always finishes. The second one guarantees that dynamic values are always of the correct type. Proving these properties is beyond the scope of this thesis.

Definition 3.3 (Execution Safety) Let $\mathfrak{A}$ be a well-formed UML model configuration. Let $\mathfrak{B}$ be a well-formed ASM state over Υ_{STAT}. Let $\mathfrak{C}$ be a well-formed ASM state over Υ_{AUX}. Let Π_{UML} be the ASM rule capturing the step semantics of UML.

The UML model semantics is execution-safe if the ASM graph of $(\mathcal{S}, \Pi_{\text{UML}}, \mathfrak{A} \cup \mathfrak{B} \cup \mathfrak{C})$ is finite and acyclic.

Definition 3.4 (Type Safety) Let $\mathfrak{A}$ be a well-formed UML model configuration.

The UML model semantics is type-safe if every configuration $\mathfrak{B}'$ such that $\mathfrak{B} \longrightarrow \mathfrak{B}'$ is a well-formed configuration.

CHAPTER 4

OCL SEMANTICS

This chapter presents the semantics of OCL expressions and OCL constraints. We define the syntax of OCL expressions and develop their semantics as a recursive function. The function requires a variable assignement and fetches values in the UML model configuration. We give the formal syntax of OCL constraints in terms of an adapted μ-calculus. We explain how it relies on extensions to the $\Upsilon_{\texttt{uml}}$ vocabulary. We show how class invariants, method preconditions and method postconditions are captured as templates. We give a formal semantics to OCL constraints in terms of the adapted μ-calculus semantics. Finally, we explain how the atomic properties of a constraint are evaluated during the computation of the UML model execution graph. Some details are relegated to Appendix C.

4.1 Expressions

4.1.1 Syntax

Sorts OCL expressions are captured by the following sorts, some of which have already been introduced:

sort Val
sort Var
sort $OclOp \equiv \{., \textbf{\textit{iterate}}, \textbf{\textit{@pre}}\}$
sort Uop
sort Bop
sort $ClassField$
sort $OclExp$

Variables include method parameters, OCL variables appearing in collection operators and the special variable **self**. We consider three OCL operators: the navigation operator "**.**", the collection operator "**iterate**" and the special "***@pre***" operator. The sorts Bop and Uop represent the usual binary and unary operators on integers, booleans and lists. Figure C.2 in Appendix C gives the list of the considered operators along with their predefined types. Finally, the sort $OclExp$ contains elements constructed according to the OCL expression syntax.

Definition 4.1 (OCL Expression Syntax) Let v range over Val, x range over Var, $\triangle$ range over Bop and Uop, cf range over $ClassField$ and e range over $OclExp$. An OCL expression $e : OclExp$ must satisfy the following syntax.

$$e \ ::= \ v \mid x \mid e \triangle e \mid \triangle e \mid e \,.\, cf \mid e \,.\, \textbf{iterate}\ (x\ ;\ x = e \mid e\ ) \mid e\ \textbf{\textit{@pre}}$$

Figure 4.1 shows how collection operators are encoded using the **iterate** operator. Note that "|" is the boolean OR operator, while "||" is its exclusive XOR counterpart.

size	**iterate**(v_1; $v_2 = 0 \mid v_2 + 1$)	Size of a collection
forall$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **true** $\mid v_2$ & exp)	Universal quantifier
exists$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **false** $\mid v_2 \mid exp$)	Existential quantifier
unique$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **false** $\mid v_2 \mid\mid exp$)	Unicity

Figure 4.1: Collection Operators

Functions The OCL expression semantics is integrated into the UML model semantics by means of four external functions:

sort	$OclEnv$	$=$	$Var \mapsto Val$
external	$[\![\_]\!]\_$	:	$OclExp \times OclEnv \to Val$
external	$\{\!\{\_\}\!\}\_$	:	$OclExp \times OclEnv \to OclExp$
external	$[\![\_]\!]^{\mathcal{L}}\_$	:	$\mathcal{L}(OclExp) \times OclEnv \to \mathcal{L}(Val)$
external	$\{\!\{\_\}\!\}^{\mathcal{L}}\_$	:	$\mathcal{L}(OclExp) \times OclEnv \to \mathcal{L}(OclExp)$

Given an UML model configuration, these functions allow OCL expressions to be evaluated. There are two kinds of evaluation. A complete evaluation ($[\![\_]\!]\_$) takes an expression and a variable assignment and returns the value that the expression takes in the current configuration. A partial evaluation ($\{\!\{\_\}\!\}\_$) takes an expression and a variable assignment and evaluates every expression of the form "e **@*pre***". For both kinds of evaluation, we define point-wise extension to lists ($[\![\_]\!]^{\mathcal{L}}\_$ and $\{\!\{\_\}\!\}^{\mathcal{L}}\_$). This simplifies the evaluation of a list of method parameters, which are OCL expressions.

The evaluation functions require three auxiliary external functions:

external	bop	:	$Bop \times Val \times Val \to Val$
external	uop	:	$Uop \times Val \to Val$
external	$\vdash$	:	$OclExp \to Type$

The first two give the standard interpretations of binary and unary operators. The $\vdash$ function assigns a type to an OCL expression. We write $e \vdash \tau$ when expression e if of type τ. This function is defined by derivation rules, as explained in Section C.1.

4.1.2 Semantics

Given the required external functions, the semantics of an OCL expression is defined as follows:

External Function 4.1 (Evaluating an Expression) Consider an OCL expression e and a variable environment ζ. The evaluation function $[\![e]\!]_\zeta$ is defined as follows. In the case of values, variables and simple operators, we have:

$$
\begin{array}{ll}
e \equiv v: & v \\
e \equiv x: & \zeta(x) \\
e \equiv e_1 \bigtriangleup e_2: & \mathtt{bop}(\bigtriangleup, [\![e_1]\!]_\zeta, [\![e_2]\!]_\zeta) \\
e \equiv \bigtriangleup e_1: & \mathtt{uop}(\bigtriangleup, [\![e_1]\!]_\zeta)
\end{array}
$$

In the case of navigation, i.e. $e \equiv e_1 \,.\, cf$, $[\![e]\!]_\zeta$ is defined according to the three following cases:

if $e_1 \vdash C$ **then**
$heap([\![e_1]\!]_\zeta).env(cf)$

if $e_1 \vdash \mathrm{L}(C)$ **and** $cf.f.ty \in SimpleType$ **then**
let $\langle o_1, \ldots, o_n \rangle = [\![e_1]\!]_\zeta$ **in**
$\langle heap(o_1).env(cf), \ldots, heap(o_n).env(cf) \rangle$

if $e_1 \vdash \mathrm{L}(C)$ **and** $cf.f.ty \in ListType$ **then**
let $\langle o_1, \ldots, o_n \rangle = [\![e_1]\!]_\zeta$ **in**
$heap(o_1).env(cf)$ @ ... @ $heap(o_n).env(cf)$

Note that $cf.f.ty$ corresponds to the declared type of a field.

If $e \equiv e_1.\mathbf{iterate}\{x_1; x_2 = e_2 \mid e_3\}$, then $[\![e]\!]_\zeta$ is defined as follows:

$$
\begin{array}{l}
\textbf{let } v_1 = [\![e_1]\!]_\zeta \textbf{ in} \\
\quad \textbf{if } \mathtt{size}(v_1) = 0 \textbf{ then} \\
\quad\quad \langle\rangle \\
\quad \textbf{else} \\
\quad\quad \textbf{let } v_2 = [\![e_2]\!]_\zeta \textbf{ in} \\
\quad\quad \textbf{let } v_3 = \mathtt{hd}(v_1) \textbf{ in} \\
\quad\quad \textbf{let } v_4 = [\![e_3]\!]_{\zeta[x_1 \mapsto v_3, x_2 \mapsto v_2]} \textbf{ in} \\
\quad\quad\quad \textbf{if } \mathtt{size}(v_1) = 1 \textbf{ then} \\
\quad\quad\quad\quad v_4 \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad [\![\mathtt{tail}(v_1).\mathbf{iterate}\{x_1; x_2 = v_4 \mid e_3\}]\!]_\zeta
\end{array}
$$

External Function 4.2 (Partially Evaluating an Expression) Consider an OCL expression e and a variable environment ζ. In the case of values and variables, we have $\{\!\{e\}\!\}_\zeta = e$. The other cases are defined as follows:

$e \equiv e_1 \triangle e_2$:	$\{\!\{e_1\}\!\}_\zeta \triangle \{\!\{e_2\}\!\}_\zeta$
$e \equiv \triangle e_1$:	$\triangle \{\!\{e_1\}\!\}_\zeta$
$e \equiv e_1.cf$:	$\{\!\{e_1\}\!\}_\zeta.cf$
$e \equiv e_1.\mathbf{iterate}\{x_1; x_2 = e_2 \mid e_3\}$:	$\{\!\{e_1\}\!\}_\zeta.\mathbf{iterate}\{x_1; x_2 = \{\!\{e_2\}\!\}_\zeta \mid \{\!\{e_3\}\!\}_\zeta\}$

Finally, when $e \equiv e_1@\boldsymbol{pre}$, the subexpression is evaluated, i.e. $\{\!\{e_1@\boldsymbol{pre}\}\!\}_\zeta = [\![e_1]\!]_\zeta$.

The point-wise extensions to list are defined straightforwardly as described below.

External Function 4.3 (Evaluating a List of Expressions) Consider a list of OCL expressions $\langle e_1, \ldots, e_n\rangle$ and a variable environment ζ. The function $[\![\_]\!]^{\mathcal{L}}_\zeta$ is defined as the point-wise extension of $[\![\_]\!]_\zeta$, that is:

$$[\![\langle e_1, \ldots, e_n \rangle]\!]^{\mathcal{L}}_{\zeta} = \langle [\![e_1]\!]_{\zeta}, \ldots, [\![e_n]\!]_{\zeta} \rangle$$

External Function 4.4 (Partially Evaluating a List of Expressions) Consider a list of OCL expressions $\langle e_1, \ldots, e_n \rangle$ and a variable environment ζ. The $\{\!\{\_\}\!\}^{\mathcal{L}}_{\zeta}$ is defined as the point-wise extension of $\{\!\{\_\}\!\}_{\zeta}$, that is:

$$\{\!\{\langle e_1, \ldots, e_n \rangle\}\!\}^{\mathcal{L}}_{\zeta} = \langle \{\!\{e_1\}\!\}_{\zeta}, \ldots, \{\!\{e_n\}\!\}_{\zeta} \rangle$$

4.2 Constraints

4.2.1 Syntax

An OCL constraint is defined by a context (a class and optionally a method signature), a list of OCL expressions acting as atomic properties and a μ-formula that expresses the temporal relation between these properties.

The syntax of a μ-formula is given by Fig. 4.2. The symbol Φ denotes an ASM first-order predicate. The symbol X is a μ-variable. The construct $\diamond\phi$ means: from the current configuration, a transition exists leading to a configuration where ϕ holds. The construct $\Box\phi$ means: from the current configuration, every possible transition leads to a configuration where ϕ holds. The constructs $\mu X.\phi$ and $\nu X.\phi$ allows ϕ to be iterated. It is assumed that ϕ contains the free variable X. Thus, ϕ will be

tested for successive values of X, which is intuitively considered as a set of current configurations. In particular, $\mu X.\phi$ is intuitively considered finitely iterating ϕ and $\nu X.\phi$ is intuitively considered as infinitely iterating ϕ.

$$\begin{array}{rcl} \phi & ::= & \Phi \mid X \mid \neg\phi \mid \phi \vee \phi \mid \diamond\phi \mid \Box\phi \mid \\ & & \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \mu X.\phi \mid \nu X.\ \phi \end{array}$$

Figure 4.2: μ-Calculus Syntax

ASM predicates are used to fetch the value of evaluated OCL expressions and to check the status of method instances. That information is stored in three dynamic functions:

$$\begin{array}{lll} \textbf{dynamic} & CVals & : \ Obj \rightarrow \mathcal{L}(Val) \\ \textbf{dynamic} & @now & : \ Obj \times MSig \rightarrow Bool \\ \textbf{dynamic} & @post & : \ Obj \times MSig \rightarrow Bool \end{array}$$

In any configuration, we assume that these functions correctly depict atomic properties of specified constraints and method instances. They are evaluated during the computation of the UML model execution graph as explained in Section 4.3. For definiteness, we restrict ASM predicates.

Definition 4.2 (Allowed Predicates) A predicate Φ of a μ-formula has one of the following form:

- **true**
- **false**
- $CVals(o).n$ where o is a free variable and n an integer
- $@n(o,m) \equiv \exists_{m':Msig} \cdot (c,m') \preceq_o (c,m) \wedge @now(o,m')$ where o is a free variable, m a method signature and c a class

- $@p(o, m) \equiv \exists_{m':Msig} \cdot (c, m') \preceq_o (c, m) \wedge @post(o, m')$ where o is a free variable, m a method signature and c a class.

The "." operator applied to $CVals(o)$ allows us to retrieve the n^{th} element of the list. Notice how the $@n(o, m)$ and $@p(o, m)$ predicates take into consideration overridden methods. This is necessary as a contraint applies to subclasses and overridden methods.

The abstract syntax of an OCL constraint is defined below.

Definition 4.3 (OCL Constraint) An OCL constraint is a 4-tuple $(C, m, exps, \phi)$, where:

- $C : Class$ is the class context of the constraint
- $m : Msig$ is the (optional) method context of the constraint
- $exps$ is a list of OCL expressions
- ϕ is μ-formula containing only allowed ASM predicates (using C as the class and m as the method signature).

4.2.2 Semantics

A UML model satisfies an OCL constraint $(C, m, exps, \phi)$ if the μ-formula is satisfied for every instance of C (and its subclasses). The free variable in the predicates of the μ-formula is a place-holder for an instance of C. The formal definition requires the observation criteria of Section 2.5. In addition, it requires the relation $\Lambda \models \phi$, which means that the graph Λ satisfies the formula ϕ. This corresponds to the formal semantics of the adapted μ-calculus, which is defined in Section C.2.

Definition 4.4 (OCL Constraints Semantics) Consider an UML execution graph Λ and an OCL constraint $c = (C, MSig, exps, \phi)$ with r as the only free variable of the predicates of ϕ. The graph Λ satisfies c if:

- For every $o \in Obj$ such that $\exists\, \mathfrak{A} \in \Lambda\ .\ heap(o)^{\mathfrak{A}} \neq \bot\ \wedge\ heap(o)^{\mathfrak{A}}.c \preceq_h C$, the following holds:
  - $\Lambda_{|heap(o)\neq\bot} \models \phi[r \mapsto o]$.

The expression "$[r \mapsto o]$" denotes the substitution of the free variable r with the object o in the predicates of ϕ. Note how the graph is observed to make sure the formula is verified on the segment of the graph where the object exists.

4.2.3 Templates

Class Invariants A class invariant template is translated to an OCL constraint as follows:

$$\begin{array}{l}\textbf{context } C \\ \textbf{inv: } e_1\end{array} \quad \mapsto \quad (C, \bot, \{e_1\}, \nu X.\Box X \wedge CVals(o).1)$$

Informally, it reads as: the OCL expression e_1 always holds for an object of type C.

Method Pre/post Conditions A pre/post condition template is translated to an OCL constraint as follows:

$$\begin{array}{l} \textbf{context } \text{C::MSig} \\ \quad \textbf{pre: } e_1 \\ \quad \textbf{post: } e_2 \end{array} \quad \mapsto \quad \begin{array}{l} (C, m, \{e_1, e_2\}, \nu X.\Box X \wedge \phi \wedge \psi) \\ \\ \phi \equiv \diamond @n(o,m) \Rightarrow CVals(o).1 \\ \psi \equiv @p(o,m) \Rightarrow CVals(o).2 \end{array}$$

Informally, it reads as: if the system is in a $@pre$ configuration, then e_1 holds; if the system is in a $@post$ configuration, then e_2 holds. Note how the $@pre$ configuration is encoded as: there exists a next configuration where $@now$ holds.

Specifying Other Constraints According to the semantics developed , `OCL` constraints are not restricted to invariants and pre/post conditions. For example, an after/eventually template may be translated to an `OCL` constraint as follows:

$$\begin{array}{l} \textbf{context } \text{C} \\ \quad \textbf{after: } e_1 \\ \quad \textbf{eventually: } e_2 \end{array} \quad \mapsto \quad \begin{array}{l} (C, \bot, \{e_1, e_2\}, \nu X.\Box X \wedge \phi \Rightarrow \psi) \\ \\ \phi \equiv CVals(o).1 \\ \psi \equiv \mu Y.\Box Y \ \vee \ CVals(o).2 \ \wedge \ \diamond \textbf{true} \end{array}$$

Informally, it reads as: if e_1 holds in some configurations, then e_2 will eventually hold in a future configuration. The μ-formula is more intricate. It first reads as: it is always true that $\phi \Rightarrow \psi$. Then ϕ is **true** if e_1 holds. The formula ψ may be understood as follows. For a particular path, if a future configuration is found such that e_2 holds, then ψ is **true** for that path. If a future configuration has not yet been found on the path such that e_2 holds and there is no more successor configuration (i.e. the use of $\diamond$**true**), then ψ is false on that path. The formula ψ must hold for every path.

4.3 UML Model Semantics Extensions

The OCL *@pre* operator requires that the *@pre* configuration be matched with the corresponding *@post* configuration. This is not easily expressed by a μ-formula. In fact, it may even be impossible but it is beyond the scope of this thesis to prove such a claim. To overcome the problem, we evaluate the OCL expressions of an OCL constraint during the computation of the UML model execution graph. We push the partially evaluated OCL expressions on the stack. This insures the required correspondence.

This can be done with simple extensions to the semantics. First, we extend the $\Upsilon_{\mathtt{stat}}$ vocabulary with a function containing a list of OCL expressions associated with a class:

static CONS : $Class \rightarrow \mathcal{L}(OclExp)$

Then we extend the $\Upsilon_{\mathtt{uml}}$ vocabulary with a function containing a list of evaluated OCL expressions:

dynamic $CExps$: $Obj \rightarrow \mathcal{L}(OclExp)$

Finally, we redefine the sort $Frame$ to include a list of partially evaluted OCL expressions:

sort $Frame = \mathcal{T}(mth : ClassMSig, \ldots, ocl : \mathcal{L}(OclExp))$

Method Instances The functions *@now* and *@post* need to be initialized at the beginning of the step by the rule $\Pi_{\mathtt{chooseThread}}$:

$$
\begin{array}{l}
\| \ [\ldots] \\
\| \ \Sigma_{o:Obj} : \Sigma_{m:Msig} : @now(o,m) := \textbf{false} \\
\| \ \Sigma_{o:Obj} : \Sigma_{m:Msig} : @post(o,m) := \textbf{false}
\end{array}
$$

Then, when a method is pushed on the stack, the $@now$ function is updated. The partially evaluated OCL expressions are also pushed on the stack. This is done in the $\Pi_{\texttt{pushMethod}}(c, msig, \ldots)$ rule:

$$
\begin{array}{l}
\| \ [\ldots] \ \texttt{push}(cont(\tilde{th}), ((c, msig), \ldots, \{\!\{CExps(\tilde{o})\}\!\}^{@}_{\zeta})) \\
\| \ @now(snd, msig) := \textbf{true}
\end{array}
$$

When a method is popped, the rule $\Pi_{\texttt{popMethod}}$ updates the $@post$ function. Also, the partially evaluated OCL expressions are taken from the frame and put into the $CExps$ function:

$$
\begin{array}{l}
\| \ [\ldots] \\
\| \ tf := \texttt{top}(cont(\tilde{th})) : \\
\qquad \| \ CExps(\tilde{o}) := tf.ocl \\
\qquad \| \ @post(\tilde{o}, tf.cmsig.msig) := \textbf{true}
\end{array}
$$

Object Creation The object creation rule $\Pi_{\texttt{createObj}}(o, cf, C)$ is extended by adding a list of OCL expressions for the newly created object:

$$
\begin{array}{l}
\partial\, o' : Obj : \\
\qquad \| \ [\ldots] \\
\qquad \| \ CExps(o') := \textsc{cons}(C)
\end{array}
$$

Evaluating Atomic Properties Finally, the evaluation of atomic properties is done by the $\Pi_{\texttt{fireUpdates}}$ rule:

$$\begin{array}{l} \| \ [\ldots] \\ \| \ \Sigma_{o:Obj} : CVals(o) := [\![CExps(o)]\!]^{\mathcal{L}}_{\zeta} \end{array}$$

CHAPTER 5

TOOL DESIGN

This chapter presents a high-level description of a model-checker prototype based on the work of this thesis. We first present the context in which such a tool is assumed to be used. Then, we describe the architecture of the tool. The tool was implemented at the CRAC laboratory of École Polytechnique de Montréal from July 2003 to March 2004.

5.1 Design Workflow

Figure 5.1 shows a design workflow involving the model-checker prototype.

Modeling The first task of the designer is to model the system using the UML fragment presented in sections 1.2 and 3.2. This includes describing the data structure through a class diagram, describing the initial configuration through an object diagram and describing class behavior through statechart diagrams.

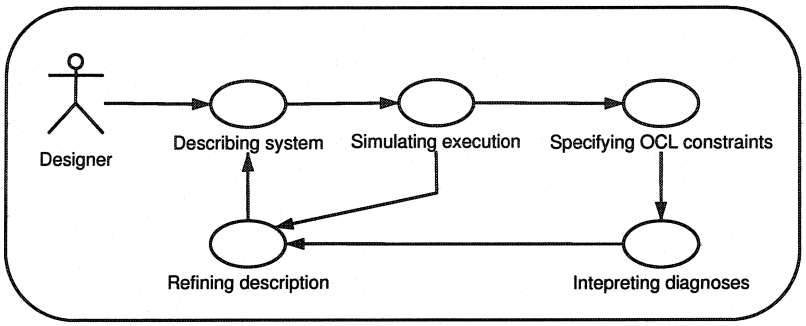


Figure 5.1: Design Workflow

Simulating Execution Once the model is completed, the designer can simulate it right away to detect simple errors that might have occurred in the description.

Specifying OCL Constraints In order to detect more intricate errors, the designer can specify OCL constraints.

Interpreting Diagnosis Once constraints are specified, the prototype will verify that the model satisfies the specification. If the model fails to satisfy a constraint, the tool will issue a diagnosis. The designer must analyze the diagnosis in order to understand the design flaw in his/her model.

Refining Description Once a design flaw is correctly identified, the designer can modify his/her design and go through the previous steps again to validate it.

5.2 Architecture

Figure 5.2 presents the architecture of the prototype. The tool is divided into three executables.

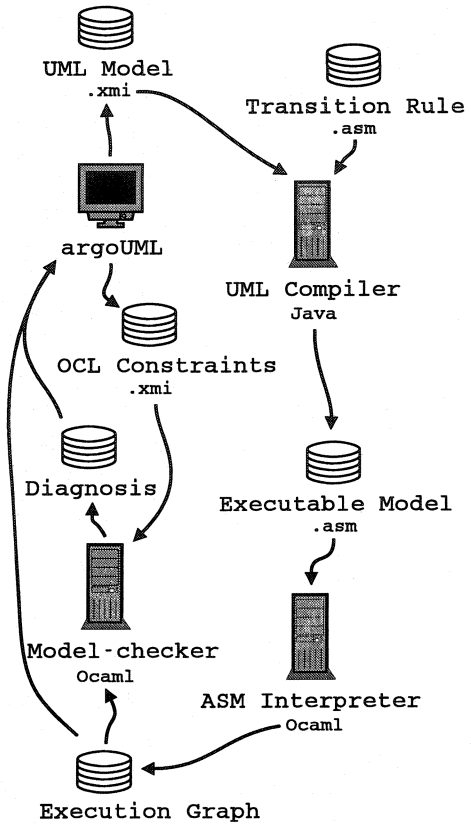


Figure 5.2: Prototype Architecture

The first executable consists of a modified version of the open-source UML editing tool ArgoUML[1]. The modification is aimed at compiling the UML model and visualizing results from the model-checking engine. The second executable is an ASM interpreter written from scratch in the OCaml[2] programming language. It computes the execution graph and evaluates OCL expressions. The third executable verifies OCL constraints and generates verification diagnoses.

Interaction between the executables is done through a set of files. Human-machine interaction is done through the graphical interface of ArgoUML. Figure 5.3 shows a

[1]http://argouml.tigris.org
[2]http://caml.inria.fr/ocaml

typical session. The background window is **argoUML**. The bottom window shows the progress of the execution graph computation and **OCL** constraints verification. The right window is a graphical representation of the execution graph.

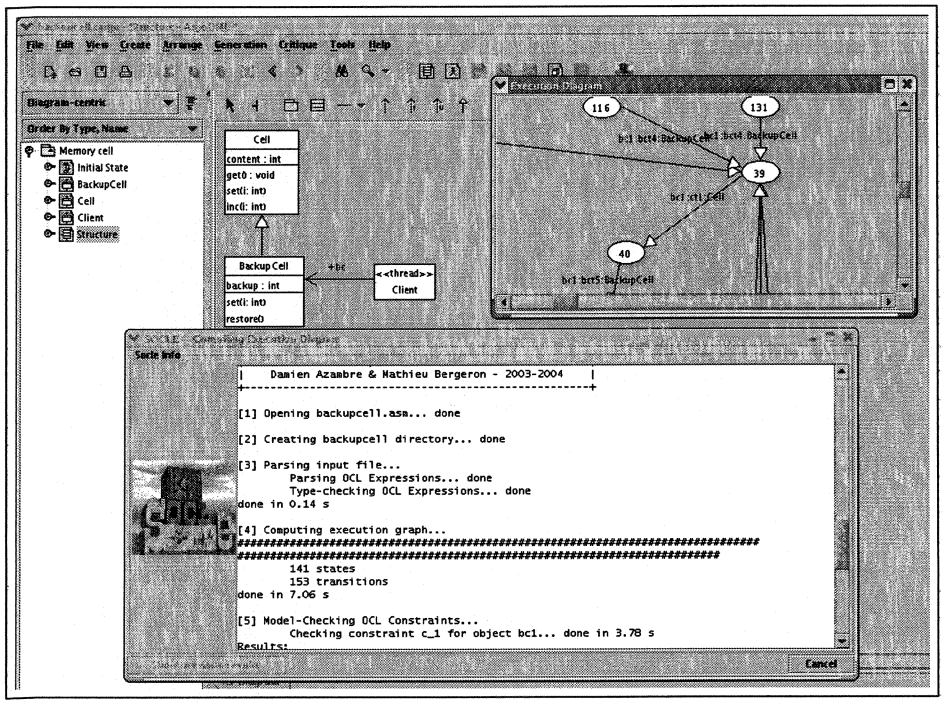


Figure 5.3: Graphical Interface

5.2.1 UML Compiler

That module corresponds to the first three phases of a compiler: parsing, elaboration and type-checking.

Extraction and Parsing The **UML** model is extracted and stored in the **ASM** sorts and functions described in Section B.1. This requires parsing the triggers and the actions. **OCL** expressions and **OCL** constraints are parsed according to sections 4.1 and 4.2.

Elaboration The elaboration is done according to Section B.1.2. In addition, the method call actions are elaborated according to Section B.1.4.

Type-Checking The well-formedness constraints of sections B.1.3 and B.2.2 are verified. In an error occurs, the tool warns the designer.

ASM Specification Generation Finally, an ASM specification is generated. The first half of the specification, which corresponds to the initial state, is generated from the information gathered during parsing, elaboration and type-checking. In addition, the initial state must respect the well-formedness constraints described Section B.3.1. The second half of the specification is directly copied from a template file. It corresponds to the transition rules described in Section B.4. Finally, for each OCL constraint, a μ-formula is generated and included in the ASM specification.

5.2.2 Graph Generation

This module is a modified ASM interpreter developed by Damien Azambre [Aza03] at the CRAC laboratory. It parses the ASM specification corresponding to the current UML model. Then it generates the execution graph in accordance with the UML model execution graph definition (Section 3.8) and the ASM rule semantics (Section A.4). The graph is written to a file for visualization purposes. Figure 5.4 shows the simulation of a model through the graphical interface. When a user selects a step of the execution graph, the interface answers by displaying one transition contained in that step. This consists of highlighting the fired transition and the activated states, while shadowing the deactivated states.

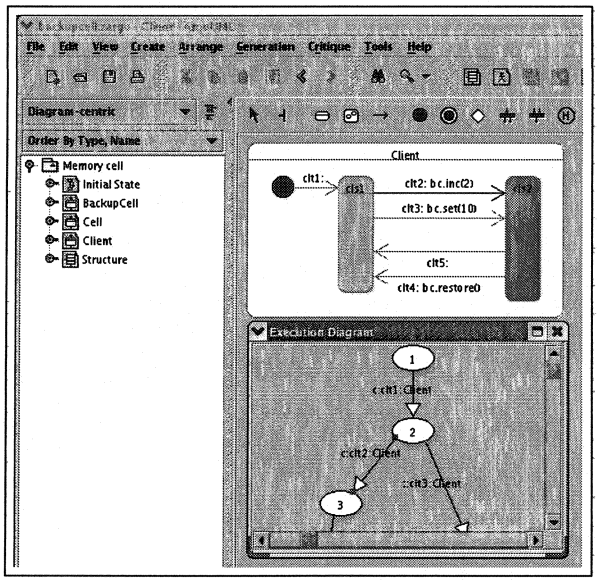


Figure 5.4: Model Simulation

5.2.3 Model-checking

This module verifies μ-formulas corresponding to OCL constraints. This is done according to the OCL constraint semantics presented in Section 4.2.2, which includes instantiation of the generated μ-formula by replacing the free variable by the appropriate object name. Each μ-formula is then verified according to the semantics of Section C.2.

5.2.4 Diagnosis Generation

This module generates the diagnosis for the verified OCL constraint. The diagnosis is written to a file for visualization purposes. It indicates in which configuration the OCL expressions are falsified and offers an explanation by recursively analyzing the evaluation of the expression.

5.2.5 Visualization

This module parses the files containing the model execution graph and `OCL` constraint diagnosis. It displays the information in the graphical interface of `ArgoUML`.

CHAPTER 6

CONCLUSION

6.1 Motivation

The main goal of this thesis was to develop formal semantics for UML models and OCL constraints. The former enables us to compute the execution graph of a UML model. The latter defines when a constraint is satified by such an execution graph, hence by a UML model. This is the first step towards a model-checking framework for OCL constraints. The latter offers an appealing design validation technique. A good design is crucial to the development of quality software. Moreover, discovering behavioral errors during design, i.e. before implementation, is notoriously cost-effective. The Unified Modeling Language was chosen as the design notation because it is the *de facto* industry standard. The Object Constraint Language was chosen as a property specification language because it is part of the UML and easy to understand. Since a software design is more abstract than the source code, it seemed natural to apply formal methods rather than testing, for example. Of the known formal methods, model-checking was selected as the most pertinent because it is highly automated.

6.2 Achievements

A formal semantics for a significant fragment of the Unified Modeling Language was developed. The semantics includes object-oriented extensions such as behavior inheritance and object creation. An OCL expression semantics was defined according to the UML configuration vocabulary and integrated into the UML model semantics in guards, assignation actions and method calls. Using this OCL expression semantics, an OCL constraint semantics was proposed using the μ-calculus modal logic. That formalization supports class invariants, method preconditions and method postconditions. Other constraints may be constructed by specifying a temporal relation between a set of OCL expressions. Finally, a prototype was implemented which relies on three modules: UML diagram extraction and compilation, execution graph computation and constraint verification.

6.3 Issues

Two kinds of issues were raised during the course of this thesis. Semantics issues concerns particularities of the semantics that may render modelling difficult. The main robustness issue is the fact that the computation of the execution graph will never return on some models. Leads are proposed to address these issues in Section 6.4.

6.3.1 Semantics Issues

Backward Navigation Backward navigation, a useful feature of OCL expressions, was not formalized. This would be particularly useful in method postconditions. In the memory cell example, consider that we want to specify that the client will never set two cells to the same value. It is possible, however, that two cells acquire the

same value as a result of incrementation. Thus, the constraint is not an invariant of the system. We need to specify that, upon the return of the method **set**, there will be no duplicate cell. This requires naming the client side of the association. Suppose it is attributed the name *owner*. Figure 6.1 gives the OCL method postcondition that specifies the desired property.

context: Cell set(i:int)
post: **self**.owner.cells.**forall**$\{x_1, x_2 \mid x_1 \neq x_2 \Rightarrow (x_1.\text{content} \neq x_2.\text{content})\}$

Figure 6.1: Backward Navigation Example

Modeling External Behavior Another useful addition to the semantics would be the possibility to model the behavior of external agents. In the memory cell example, this could be the user's behavior. The user could be represented by a state-machine not bound to any object or any class would call methods and activate signals to simulate utilization scenarios.

Modeling Method Calls The integration of method calls in the statechart semantics requires careful modeling. In the memory cell example, consider that two clients access the same cell. Figure 6.2 reproduces an excerpt of the **Cell** statechart. Consider that both clients concurrently call the method **inc**. The first transition will be fired twice, causing the target state to be active but the following transition will be fired only once. So, only one instance of the **inc** will return.

A solution to this problem would be to include active states in the frame of the calling stack, much as the program counter is included in the frame of a programming language. However, this raises an ontological issue, as active states represent the local configuration of the object, not of the method instance.

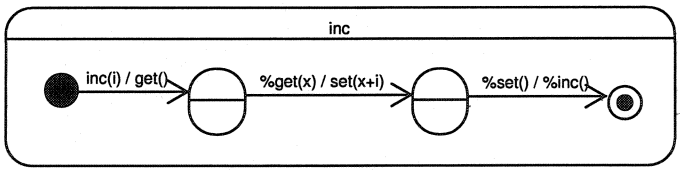


Figure 6.2: Excerpt of **Cell**'s Behavior

6.3.2 Robustness Issues

The prototype based on the work of this thesis allows OCL constraints to be verified on UML models that yield small execution graphs. However, it clearly lacks the robustness of state-of-the-art model-checkers.

Computability The first problem is that it is not detected prior to computation if a UML model yields an infinite graph. On some UML models, the prototype will never finish its analysis. Usually, a model-checker does not allow the user to specify a model it cannot analyze. However, doing so for the model-checking of OCL constraints may be difficult. As OCL constraints require evaluation of OCL expressions, objects need to be represented in the configuration. Furthermore, verification of the method pre/post conditions require a representation of method instances. This forces the presence of unbounded data structures such as lists and stacks in the configuration. Hence, the state-space of a non-trivial UML model is infinite. Of course, with careful modeling of recursive method calls and object creation, the reachable state-space is finite, although it is difficult to statically determine if it is the case.

Concrete Execution Graph An additional problem is that the execution graph is stored in concrete data structures. In order to process large graphs, it is necessary to encode the graph efficiently. In [BCM+90], Burch et al. show how to symbolically

represent an execution graph in order to verify large graphs. The encoding allows us to process graphs containing 10^{20} states, which are of course too large to process explicitly. However, the technique only applies to bounded data structures such as sets (over finite domains) and boolean variables.

***A Posteriori* Verification** Another problem is that the verification starts only after the computation graph is fully computed. Efficient model-checking algorithms usually support *on-the-fly* verification. This allows us to verify a property while the execution graph is computed. Errors are reported as soon as they are found. In [BCG95], Bhat et al. describe one such model-checking algorithm for the temporal logic CTL*.

6.4 Future Work

Robust Model-Checking The UML model semantics developed in this thesis is similar to a programming language semantics. Model-checking software raises the same robustness issues discussed earlier. Recently, some research led to interesting results in that field. For example, the Bandera tool set is a model-checking framework for multi-threaded Java programs. At the heart of the framework is a slicing algorithm that generates the model to verify [HDZ00]. The safety of the slicing algorithm is proved with respect to LTL (Linear Temporal Logic) properties. The model is then verified by SPIN or SMV. The diagnosis is mapped back to source code, in order to be intelligble to the programmer. Properties are expressed in a high-level language. The language supports assertions and various temporal templates. The atomic properties language is similar to the expression sublanguage of Java but unlike OCL, does not allow collections to be iterated.

A similar approach would allow us to take advantage of existing state-of-the art model-checkers. This means adapting the UML model semantics to generate some kind of control flow graph. We also need to adapt the slicing algorithm to take OCL expressions into consideration.

Semantics Improvements Another area to consider for future work is extending the semantics to facilitate modeling. For example, activity diagrams could be added as a means of modeling method calls more closely. The activity diagram acts as a flow graph. An instance of such a diagram can represent a method instance. As such, its active states can legitimately be pushed on the stack.

Diagnosis Improvements Another improvement would be to design a diagnosis generation algorithm. This algorithm would map an error to a sequence diagram explaining the problem to the designer. This would increase the usability of the model-checking framework in industrial applications.

BIBLIOGRAPHY

[ABB+03] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager and Peter H. Schmitt. The KeY Tool. Technical Report 2003-05, Department of Computing Science, Chalmers University of Technology and Göteborg University, March 2003.

[Aza03] Damien Azambre. Réalisation d'un interpréteur d'Abstract State Machines adapté à la validation de modèles UML-OCL. Master's thesis, École Polytechnique de Montréal and INSA de Lyon, 2003.

[BBF+75] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie. *Systems and Software Verification.* Springer, 1975.

[BCG95] Girish Bhat, Rance Cleaveland and Orna Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 388–397, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages

428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[BCR00] Egon Börger, Alessandra Cavarra and Elvinia Riccobene. Modeling the Dynamics of UML State Machines. *Lecture Notes in Computer Science*, 1912:223–241, 2000.

[BFS02] Julian Bradfield, Juliana Küster Filipe and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. *Lecture Notes in Computer Science*, 2306:203–217, 2002.

[BP88] Barry W. Boehm and Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1988.

[Bro01] F. P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley Publishing Company, 2001.

[BW02] Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. In César Muñoz, Sophiène Tahar and Víctor Carreño, editors, *Theorem Proving in Higher Order Logics*, number 2410 in Lecture Notes in Computer Science, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.

[CGHS00] K. Compton, Y. Gurevich, J. Huggins and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, EECS Department, University of Michigan, 2000.

[CH99] S. Cater and J. Huggins. An ASM Dynamic Semantics for Standard ML. Technical Report CPSC-1999-2, Kettering University, October 1999.

[Cim01] Alessandro Cimatti. Industrial Applications of Model Checking. pages 153–168, 2001.

[Cle90] R. Cleaveland. Tableaux-Based Model Checking in the Propositional μ-calculus. *Acta Informatica*, 27:725–747, 1990.

[CRS04] Alessandra Cavarra, Elvinia Riccobene and Patrizia Scandurra. A Framework to Simulate UML Models: Moving from a Semi-Formal to a Formal Environment. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1519–1523. ACM Press, 2004.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. Technical Report CMU-CS-96-178, Carnegie Mellon University (CMU), September 1996.

[DKR00] D. Distefano, J. Katoen and A. Rensink. Towards Model Checking OCL. In *Proceedings of ECOOP Workshop on Defining a Precise Semantics for UML, 2000.*, 2000.

[E. 98] G. J. Holzmann E. Mikk, Y. Lakhnech, M. Siegel. Verifying Statecharts with SPIN. *Proc. Workshop on Industrial-strength Formal specification Techniques*, pages 90–101, October 1998.

[FEL97] Robert B. France, Andy Evans and Kevin Lano. The UML as a Formal Modeling Notation. In Haim Kilov, Bernhard Rumpe and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[GHP97] Jean-Charles Grégoire, Gerard J. Holzmann and Doron A. Peled, editors. *The Spin Verification System*, volume 32 of *DIMACS series*. American Mathematical Society, 1997. ISBN 0-8218-0680-7, 203p.

[GLM99] S. Gnesi, Diego Latella and Mieke Massink. Model Checking UML Statechart Diagrams using JACK. In Raymond Paul and Catherine Meadows,

editors, *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering.* IEEE, 1999.

[GM90] Y. Gurevich and L. Moss. Algebraic Operational Semantics and Occam. In E. Börger, H. Kleine Büning and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 176–192. Springer, 1990.

[HDZ00] John Hatcliff, Matthew B. Dwyer and Hongjun Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[HS00] J. Huggins and W. Shen. The Static and Dynamic Semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, 2000.

[Jür02] Jan Jürjens. *Principle for Secure Systems Design.* PhD thesis, Oxford University, 2002.

[KC00] Soon-Kyeong Kim and David Carrington. A Formal Mapping between UML Models and Object-Z Specifications. *Lecture Notes in Computer Science*, 1878, 2000.

[KP97a] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[KP97b] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[LP99] Johan Lilius and Ivan Porres Paltor. vUML: a Tool for Verifying UML Models. Technical Report TUCS-TR-272, Turku Centre for Computer Science, Finland, May 18, 1999.

[McM92] K. L. McMillan. The SMV System, Symbolic Model Checking - an Approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[ML02] R. Marcano and N. Levy. Using B Formal Specifications for Analysis and Verification of UML/OCL Models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.

[Mlo01] M. Mlotkowski. *Specification and Optimization of the Smalltalk Programs.* PhD thesis, University of Wroclaw, 2001.

[NPW02] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Springer Verlag, 2002.

[OMG02] OMG. Response to the UML 2.0 OCL RfP (ad/2000-09-03). Technical Report ad/2002-05-09, 2002.

[OMG03a] OMG. Unified Modeling Language: Infrastructure (ad/2000-03-01). Technical Report OMG RFP ad/00-09-01, 2003.

[OMG03b] OMG. Unified Modeling Language: Superstructure (ad/2000-09-03). Technical Report OMG RFP ad/00-09-02, 2003.

[Pre87] R. S. Pressman. *Software Engineering—A Practitioner's Approach.* McGraw-Hill, 2nd edition, 1987.

[RJB98] J. Rumbaugh, I. Jacobson and G. Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley Publishing Company, 1998.

[RW] G. Reggio and R.J. Wieringa. Thirty-one Problems in the Semantics of UML 1.3 Dynamics.

[RWH01] Bernhard Reus, Martin Wirsing and Rolf Hennicker. A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. *Lecture Notes in Computer Science*, 2029:300–317, 2001.

[SSB01] Robert Stärk, Joachim Schmid and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

[Tra00] I. Traoré. An Outline of PVS Semantics for UML Statecharts. *J.UCS: Journal of Universal Computer Science*, 6(11), November 2000.

APPENDIX A

ASM DETAILS

This appendix contains details about the **ASM** formalism. We define the notion of domain. We give the formal semantics of **ASM** predicates and of transition rules.

A.1 Term Evaluation

Variable Assignment Evaluating a term requires a state and a variable assignment. A variable assignment is a finite mapping from variables of an **ASM** transition rule and elements of the appropriate carrier sets of a current state. The initial mapping ζ maps every variable to $\perp^{\mathfrak{A}}$. Updating the variable assignment is as usual.

Definition 1.1 The update of ζ^* on x is defined as follows:

$$\zeta^*_{x \mapsto a}(v) = \begin{cases} a & \text{if } v = x \\ \zeta^*(v) & \textit{otherwise} \end{cases}$$

Term Interpretation The terms are evaluated recursively. We define a function $[\![\_]\!]^{\mathfrak{A}}_{\zeta^*} : \Delta \to \bigcup_{i \in I} S_i^{\mathfrak{A}}$ to denote this evaluation.

Definition 1.2 Consider a state $\mathfrak{A}$ and a variable assignment ζ^*, the term evaluation function $([\![\_]\!]^{\mathfrak{A}}_{\zeta^*})$ is defined as follows:

- $[\![v_i]\!]^{\mathfrak{A}}_{\zeta^*} = \zeta^*(v_i)$
- $[\![f]\!]^{\mathfrak{A}}_{\zeta^*} = f^{\mathfrak{A}}$
- $[\![f(t_1, \ldots, t_k)]\!]^{\mathfrak{A}}_{\zeta^*} = f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta^*}, \ldots, [\![t_k]\!]^{\mathfrak{A}}_{\zeta^*})$.

A.2 Domains

In general, a carrier set is infinite. We define the notion of domain, which is a finite subset of that carrier set. Formally, this is as follows:

Definition 1.3 (Domain) Let $\mathfrak{A}$ be a state and S_i be a sort. The domain $\tilde{S}_i^{\mathfrak{A}} \subseteq S_i^{\mathfrak{A}}$ of S_i in $\mathfrak{A}$ is defined as follows:

$$\tilde{S}_i^{\mathfrak{A}} = \{a \in S_i^{\mathfrak{A}} \mid \exists f \in \Omega_{stat} \cup \Omega_{dyn} \,.\, f^{\mathfrak{A}}(\ldots, a, \ldots) \neq \bot^{\mathfrak{A}} \vee f^{\mathfrak{A}}(\ldots) = a\}$$

A.3 Predicate Semantics

Given a state $\mathfrak{A}$ and a variable assignment ζ^*, Fig. A.1 gives the semantics of a predicate Φ. The evaluation function $[\![\_]\!]^{\mathfrak{A}}_{\zeta^*}$ takes a predicate and returns a boolean

value (i.e. an element of sort *Bool*). The only subtlety is that quantifiers are bounded to domains rather than ranging over the whole carrier set.

$$
\begin{array}{lcl}
[\![v]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & \zeta^*(v) \\
[\![f(\vec{t})]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![f(\vec{t})]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\neg\Phi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & \neg[\![\Phi]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\Phi \wedge \Psi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\Phi]\!]^{\mathfrak{A}}_{\zeta^*} \text{ and } [\![\Psi]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\exists x : S_i.\ \Phi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\Phi]\!]^{\mathfrak{A}}_{\zeta^*_{x\mapsto a}} \text{ for some } a \in \tilde{S}_i^{\mathfrak{A}} \\
[\![\forall x : S_i.\ \Phi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\Phi]\!]^{\mathfrak{A}}_{\zeta^*_{x\mapsto a}} \text{ for all } a \in \tilde{S}_i^{\mathfrak{A}} \\
[\![\Phi \vee \Psi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\neg(\neg\Phi \wedge \neg\Psi)]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\Phi \Rightarrow \Psi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\neg\Phi \vee \Psi]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\Phi \Leftarrow \Psi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\Psi \Rightarrow \Phi]\!]^{\mathfrak{A}}_{\zeta^*} \\
[\![\Phi \Leftrightarrow \Psi]\!]^{\mathfrak{A}}_{\zeta^*} & \Leftrightarrow & [\![\Phi \Rightarrow \Psi \wedge \Phi \Leftarrow \Psi]\!]^{\mathfrak{A}}_{\zeta^*}
\end{array}
$$

Figure A.1: Predicate Semantics

A.4 Rule Semantics

The semantics of transition rules is given in terms of the function $[\![\_]\!]^{\mathfrak{A}}_{\zeta}$ which, given a state $\mathfrak{A}$ and the initial variable assignment ζ, takes a rule and returns a collection of update sets. The calculus of Fig. A.2 defines the semantics function Note that in the Σ and ϵ rules, we assume that sets in $(I_a)_{a\in J}$ are pairwise different, i.e. $\forall_{a,a'\in J}.\ I_a \cap I_{a'} = \emptyset$. The function yields a collection of update sets of form $(U_i)_{i\in I}$. The reason is that many transitions may occur from one state as an effect of the non-deterministic choose rule $(\epsilon_{x:S_i}.\Phi : \Pi)$.

A.5 External Functions

Given an ASM M containing the sort S_i, then we assume that $S_i^{\mathfrak{A}}$ is totally ordered by a relation $\preceq_{S_i}$ and that Ω contains the following external functions:

$$\begin{array}{lll} \textbf{external} & \_ = \_ & : \quad S_i \times S_i \to \mathit{Bool} \\ \textbf{external} & \_ \preceq_{S_i} \_ & : \quad S_i \times S_i \to \mathit{Bool} \end{array}$$

The following external functions are assumed to be correctly defined on a constructed sort $\mathcal{S}(A)$:

$$\begin{array}{lll} \textbf{external} & \emptyset & : \quad \mathcal{S}(A) \\ \textbf{external} & \_ \cup \_ & : \quad \mathcal{S}(A) \times \mathcal{S}(A) \to \mathcal{S}(A) \\ \textbf{external} & \_ \setminus \_ & : \quad \mathcal{S}(A) \times \mathcal{S}(A) \to \mathcal{S}(A) \\ \textbf{external} & \_ \in \_ & : \quad A \times \mathcal{S}(A) \to \mathit{Bool} \\ \textbf{external} & \{\_\} & : \quad A \to \mathcal{S}(A) \end{array}$$

The following external functions are assumed to be correctly defined on a constructed sort $\mathcal{L}(A)$:

$$\begin{array}{lll} \textbf{external} & \langle\rangle & : \quad \mathcal{L}(A) \\ \textbf{external} & \_ @ \_ & : \quad \mathcal{L}(A) \times \mathcal{L}(A) \to \mathcal{L}(A) \\ \textbf{external} & \_ :: \_ & : \quad A \times \mathcal{L}(A) \to \mathcal{L}(A) \\ \textbf{external} & \_ \in \_ & : \quad A \times \mathcal{L}(A) \to \mathit{Bool} \\ \textbf{external} & \texttt{hd} & : \quad \mathcal{L}(A) \to A \\ \textbf{external} & \texttt{tail} & : \quad \mathcal{L}(A) \to \mathcal{L}(A) \\ \textbf{external} & \_ \bullet \_ & : \quad \mathcal{L}(A) \times \mathit{Int} \to A \\ \textbf{external} & \langle\_\rangle & : \quad A \to \mathcal{L}(A) \end{array}$$

The following external functions are assumed to be correctly defined on a constructed sort $\mathcal{T}(A_1, \ldots, A_n)$:

$$\begin{array}{lll} \textbf{external} & \pi_1 & : \quad \mathcal{T}(A_1, \ldots, A_n) \to A_1 \\ \textbf{external} & \vdots & \quad \vdots \\ \textbf{external} & \pi_n & : \quad \mathcal{T}(A_1, \ldots, A_n) \to A_n \end{array}$$

The functions π_i correspond to the projection operators on tuples.

The following external functions are assumed to be correctly defined on a con-

structed sort $A_1 \mapsto A_2$:

external $\emptyset$: $A_1 \mapsto A_2$
external $\_ \in \_$: $A_1 \times (A_1 \mapsto A_2) \to Bool$
external $\_(\_)$: $(A_1 \mapsto A_2) \times A_1 \to A_2$
external $\{\_ \mapsto \_\}$: $A_1 \times A_2 \to (A_1 \mapsto A_2)$
external $\_ \oplus \_$: $(A_1 \mapsto A_2) \times (A_1 \mapsto A_2) \to (A_1 \mapsto A_2)$

The function $\_(\_)$ searches a finite mapping $A_1 \mapsto A_2$ for a particular value $a : A_1$. If there exists a couple $(a, b) : (A_1, A_2)$ in the mapping, then b is returned. Otherwise, $\perp$ is returned.

The second function $\oplus$ concatenates two mappings. It is defined as follows:

External Function 1.1 (Mapping Concatenation) Given two mappings $m_1 : A_1 \mapsto A_2$ and $m_2 : A_1 \mapsto A_2$, the concatenation is defined as follows:

$$m_1 \oplus m_2(a) = \begin{cases} m_2(a) & \textbf{if } m_2(a) \neq \perp \\ m_1(a) & \textbf{else} \end{cases}$$

The following external functions are assumed to be correctly defined on a constructed sort A^*:

external $\langle\rangle$: A^*
external $\_ \in \_$: $A \times A^* \to Bool$
external `push` : $A^* \times A \to A^*$
external `pop` : $A^* \to A^*$
external `top` : $A^* \to A$

These functions correspond to the usual operators on stacks, except for $\_[\_ \mapsto \_]$.

Finally, we define an external function taking two lists and returning a mapping.

external BUILDMAP : $\mathcal{L}(A) \times \mathcal{L}(B) \to A \mapsto B$

External Function 1.2 (Building a Mapping) Given two lists $l_1 = \langle a_1, \ldots, a_n \rangle$

and $l_2 = \langle b_1, \ldots, b_n \rangle$:

$$buildMap(l_1, l_2) = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$$

A.6 Extend Rule

The extend rule "$\partial_{x:S_i} : \Pi$" binds x to an element $s \in S_i^{\mathfrak{A}} \setminus \tilde{S}_i^{\mathfrak{A}}$ and evaluates Π. It is a shortcut for $x = \partial_{S_i} : \Pi$, where ∂_{S_i} is an external function that returns the biggest element (according to $\preceq_{S_i}$) not in the domain of S_i. Removing an element from the domain of S_i is achieved by correctly updating dynamic functions to the undefined element.

$$\circ \quad \frac{}{[\![\circ]\!]^{\mathfrak{A}}_{\zeta} \;\to\; \{\emptyset\}}$$

$$:= \quad \frac{}{[\![f(\vec{t}) := s]\!]^{\mathfrak{A}}_{\zeta} \;\to\; \{\{((f,\vec{a}),b)\}\}} \qquad \text{if } \vec{a} = [\![\vec{t}]\!]^{\mathfrak{A}}_{\zeta} \text{ and } b = [\![s]\!]^{\mathfrak{A}}_{\zeta}$$

$$? \quad \frac{[\![\Pi_1]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I}}{[\![\Phi?\Pi_1 : \Pi_2]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I}} \qquad \text{if } [\![\Phi]\!]^{\mathfrak{A}}_{\zeta} = \mathbf{true}$$

$$? \quad \frac{[\![\Pi_2]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I}}{[\![\Phi?\Pi_1 : \Pi_2]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I}} \qquad \text{if } [\![\Phi]\!]^{\mathfrak{A}}_{\zeta} = \mathbf{false}$$

$$\| \quad \frac{[\![\Pi_1]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I} \quad [\![\Pi_2]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (V_j)_{j\in J}}{[\![\Pi_1 \| \Pi_2]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i \cup V_j)_{(i,j)\in I\times J}}$$

$$\Sigma \quad \frac{[\![\Pi]\!]^{\mathfrak{A}}_{\zeta_{x\mapsto a}} \;\to\; (U_i)_{i\in I_a} \text{ for each } a \in J}{[\![\Sigma_{x:S_i}.\Phi : \Pi]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (\bigcup_{i\in\vec{\imath}} U_i)_{\vec{\imath}\in\prod_{a\in J} I_a}} \qquad \text{if } J = \{a \in \tilde{S}_i^{\mathfrak{A}} \mid [\![\Phi]\!]^{\mathfrak{A}}_{\zeta_{x\mapsto a}} = \mathbf{true}\}$$

$$\epsilon \quad \frac{[\![\Pi]\!]^{\mathfrak{A}}_{\zeta_{x\mapsto a}} \;\to\; (U_i)_{i\in I_a} \text{ for each } a \in J}{[\![\epsilon_{x:S_i}.\Phi : \Pi]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in\bigcup_{a\in J} I_a}} \qquad \text{if } J = \{a \in \tilde{S}_i^{\mathfrak{A}} \mid [\![\Phi]\!]^{\mathfrak{A}}_{\zeta_{x\mapsto a}} = \mathbf{true}\}$$

$$\mathbf{Let} \quad \frac{[\![\Pi]\!]^{\mathfrak{A}}_{\zeta_{x\mapsto a}} \;\to\; (U_i)_{i\in I}}{[\![x := t : \Pi]\!]^{\mathfrak{A}}_{\zeta} \;\to\; (U_i)_{i\in I}} \qquad \text{if } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

Figure A.2: Rule Semantics

APPENDIX B

UML SEMANTICS DETAILS

This chapter presents the complete specification of the UML model semantics developed in this thesis. It is presented as shortly as possible and comments are added only when deemed necessary. The $\Upsilon_{\texttt{stat}}$, $\Upsilon_{\texttt{uml}}$ and $\Upsilon_{\texttt{aux}}$ vocabularies are presented along with the corresponding elaborated functions and well-formedness constraints. Lastly, the complete transition rule is presented.

B.1 Static Vocabulary

B.1.1 Extracted Information

The first part of the $\Upsilon_{\texttt{stat}}$ vocabulary is a straightforward representation of a model's class and statechart diagrams. Sorts are used to structure how the information is stored while static functions actually hold the diagrams.

Basics

$$\begin{array}{lll} \textbf{sort} & \mathit{Class} & \\ \textbf{sort} & \mathit{Obj} & \\ \textbf{sort} & \mathit{OclExp} & \end{array}$$

$$\begin{array}{lll} \textbf{sort} & \mathit{SimpleType} & \equiv \{\text{VOID}, \text{INT}, \text{BOOL}\} \cup \mathit{Class} \\ \textbf{sort} & \mathit{ListType} & \equiv \{\tau : \mathit{SimpleType} \mid \text{L}(\tau)\} \cup \{C : \mathit{Class} \mid \text{L}(C)\} \\ \textbf{sort} & \mathit{Type} & = \mathit{SimpleType} \mid \mathit{ListType} \end{array}$$

Fields

$$\begin{array}{lll} \textbf{sort} & \mathit{FieldName} & \\ \textbf{sort} & \mathit{Mult} & = \mathcal{T}(\mathit{low} : \mathit{Int}, \mathit{up} : \mathit{Int}) \\ \textbf{sort} & \mathit{SimpleField} & = \mathcal{T}(t : \mathit{SimpleType}, f : \mathit{FieldName}) \\ \textbf{sort} & \mathit{ListField} & = \mathcal{T}(t : \mathit{ListType}, f : \mathit{FieldName}, m : \mathit{Mult}) \\ \textbf{sort} & \mathit{Field} & = \mathit{SimpleField} \mid \mathit{ListField} \\ \textbf{sort} & \mathit{ClassField} & = \mathcal{T}(c : \mathit{Class}, f : \mathit{Field}) \end{array}$$

Method Signatures

$$\begin{array}{lll} \textbf{sort} & \mathit{Meth} & \\ \textbf{sort} & \mathit{ParamName} & \\ \textbf{sort} & \mathit{Param} & = \mathcal{T}(t : \mathit{Type}, p : \mathit{ParamName}) \\ \textbf{sort} & \mathit{MSig} & = \mathcal{T}(\mathit{mth} : \mathit{Meth}, p : \mathcal{L}(\mathit{Param}), r : \mathit{Param}) \\ \textbf{sort} & \mathit{ClassMSig} & = \mathcal{T}(c : \mathit{Class}, \mathit{msig} : \mathit{MSig}) \end{array}$$

Class Diagram In the following, note that a class association is mapped to a field named after the *supplier* end of the association. Also, the direct inheritance relation $\prec_d$ and the SUPER function are specified by model's *generalization* associations. In the model, a thread is identified by a class stereotype.

$$\begin{array}{lll} \textbf{static} & \text{CLASSES} & : \mathcal{S}(\mathit{Class}) \\ \textbf{static} & \prec_d & : \mathcal{R}(\mathit{Class}) \\ \textbf{static} & \text{SUPER} & : \mathit{Class} \rightarrow \mathit{Class} \\ \textbf{static} & \text{FIELDS} & : \mathit{Class} \rightarrow \mathcal{S}(\mathit{Field}) \\ \textbf{static} & \text{METHODS} & : \mathit{Class} \rightarrow \mathcal{S}(\mathit{MSig}) \\ \textbf{static} & \text{THREADS} & : \mathcal{S}(\mathit{Class}) \end{array}$$

Triggers

$$
\begin{array}{lll}
\textbf{sort} & Signal & \\
\textbf{sort} & TType & \equiv \{\text{EMPTY, SIGNAL, TCALL, TRETURN}\} \\
\textbf{sort} & TEmpty & = \mathcal{T}(ty : TType) \\
\textbf{sort} & TSignal & = \mathcal{T}(s : Signal, ty : TType) \\
\textbf{sort} & TCall & = \mathcal{T}(msig : MSig, ty : TType) \\
\textbf{sort} & TReturn & = \mathcal{T}(msig : MSig, ty : TType)
\end{array}
$$

Actions

$$
\begin{array}{lll}
\textbf{sort} & AType & \equiv \{\text{SIGNAL, NEW, DEL, ASSIGN, MRETURN, MCALL}\} \\
\textbf{sort} & CallType & \equiv \{\text{VIRTUAL, SUPER}\} \\
\textbf{sort} & ASignal & = \mathcal{T}(s : Singal, ty : AType) \\
\textbf{sort} & ANew & = \mathcal{T}(cf : ClassField, ty : AType) \\
\textbf{sort} & ADel & = \mathcal{T}(cf : ClassField, ty : AType) \\
\textbf{sort} & AAssign & = \mathcal{T}(cf : ClassField, exp : OclExp) \\
\textbf{sort} & AReturn & = \mathcal{T}(msig : MSig, exp : OclExp) \\
\textbf{sort} & ACall & = \mathcal{T}(rcv : OclExp, cmsig : ClassMSig, exps : \mathcal{L}(OclExp), \\
 & & \quad cty : CallType, ty : AType)
\end{array}
$$

States

$$
\begin{array}{lll}
\textbf{sort} & StateType & \equiv \{\text{INIT, FINAL, SIMPLE, OR, AND}\} \\
\textbf{sort} & StateName & \\
\textbf{sort} & State & = \mathcal{T}(n : StateName, ty : StateType)
\end{array}
$$

Transitions

$$
\begin{array}{lll}
\textbf{sort} & TransName & \\
\textbf{sort} & Trigger & = TSignal \mid TCall \mid TReturn \\
\textbf{sort} & Action & = ASignal \mid ANew \mid ADel \mid AAssign \mid ACall \mid AReturn \\
\textbf{sort} & Guard & = OclExp \\
\textbf{sort} & Trans & = \mathcal{T}(n : TransName, src : State, tr : Trigger, g : Guard, \\
 & & \quad acts : \mathcal{L}(Action), tgt : State)
\end{array}
$$

Statechart Diagram In the following, the functions TOP, CHILD and UP hold the hierarchical structure of a model's statechart diagrams. Exceptionally, the function TRANS requires some elaboration. This is explained in Section B.1.4.

static	STATES	:	$Class \rightarrow \mathcal{S}(State)$
static	TRANS	:	$\mathcal{S}(Trans)$
static	TOP	:	$Class \rightarrow State$
static	CHILD	:	$State \rightarrow \mathcal{S}(State)$
static	UP	:	$State \rightarrow State$

B.1.2 Elaborated Information

The second part of the $\Upsilon_{\mathbf{stat}}$ vocabulary contains elaborated information. Some elaborated function are needed in the dynamic semantics. Others are auxiliary functions used in the definition of the latter.

Class Diagram Elaborated Functions

The dynamic semantics uses the following elaborated functions:

static	$\preceq_h$	:	$\mathcal{R}(Class)$
static	FIELDS*	:	$Class \rightarrow \mathcal{S}(ClassField)$
static	METHODS*	:	$Class \rightarrow \mathcal{S}(MSig)$
static	LOOKUP	:	$Class \times ClassMSig \rightarrow Class$

The inheritance relation $\preceq_h$ is the transitive reflexive closure of the direct inheritance relation $\prec_d$. The function FIELDS* returns every field of a class, including inherited fields. The function METHODS* returns every method of a class, including inherited methods. The following auxiliary functions are necessary for defining the LOOKUP function and well-formedness constraints.

static	$\sqsubseteq$	:	$\mathcal{R}(Type)$
static	$\preceq_s$	:	$\mathcal{R}(MSig)$
static	APP	:	$ClassMSig \rightarrow \mathcal{S}(ClassMSig)$
static	$\underline{\text{APP}}$	:	$ClassMSig \rightarrow ClassMsig$
static	$\prec_o$	:	$\mathcal{R}(ClassMSig)$
static	$\preceq_o^*$	:	$\mathcal{R}(ClassMSig)$

The FIELDS* and METHODS* functions are defined straightforwardly using the inheritance relation.

Elaborated Function 2.1 (Fields of a Class) Given a class A, FIELDS*(A) is defined as follows:

$$\text{FIELDS}^*(A) = \{(B, f) \mid A \preceq_h B \ \wedge \ f \in \text{FIELDS}(B)\}$$

Elaborated Function 2.2 (Methods of a Class) Consider a class A. The function METHODS$^*(A)$ is defined as follows:

$$\text{METHODS}^*(A) = \{(B, m) \mid A \preceq_h B \ \wedge \ m \in \text{METHODS}(B)\}$$

Definition of the important LOOKUP requires more work. First, we need a subtype relation ($\sqsubseteq$). It is defined as follows.

Elaborated Function 2.3 (Subtype Relation) Given two types $\tau, \tau' : Type$, the subtype relation is defined as follows:

$$\tau \sqsubseteq \tau' \ \Leftrightarrow \begin{cases} \tau = \tau' \\ \tau \equiv C,\ \tau' \equiv D \textbf{ and } C \preceq_h D \\ \tau \equiv \text{L}(C),\ \tau' \equiv \text{L}(D) \textbf{ and } C \sqsubseteq D \end{cases}$$

In order to define the $\preceq_s$ (more specific) relation method signatures are required.

Elaborated Function 2.4 (More Specific) Consider two method signatures cm and cn, with

$$cm.msig.p = \langle(\tau_1^{cm}, p_1^{cm}), (\tau_2^{cm}, p_2^{cm}), \ldots, (\tau_i^{cm}, p_i^{cm})\rangle$$

and

$$cn.msig.p = \langle(\tau_1^{cn}, p_1^{cn}), (\tau_2^{cn}, p_2^{cn}), \ldots, (\tau_j^{cn}, p_j^{cn})\rangle$$

The relation $\preceq_s$ is defined as follows.

$$\begin{aligned} cm \preceq_s cn \Leftrightarrow & cm.msig.m = cn.msig.m \\ \textbf{and} \quad & cm.c \preceq_h cn.c \\ \textbf{and} \quad & i = j \\ \textbf{and} \quad & \tau_k^{cm} \sqsubseteq \tau_k^{cn} \textbf{ for } k = 1, \ldots, i \end{aligned}$$

The more specific relation is also required to define the overriding relation.

Elaborated Function 2.5 A method cm directly overrides cn if there is some class B such that:

1. $cm.msig = cn.msig$
2. $cm.c \prec_h B$
3. $cn \in \text{METHODS}^*(B)$

The relation $\preceq_o^*$ is the transitive reflexive closure of the direct overriding relation. Finally, we can define the method lookup function.

Elaborated Function 2.6

$$\text{LOOKUP}(A, cm) =$$

$$\begin{cases} A & \textbf{if there is some } n \in \text{METHODS}(A) \textbf{ such that } n \preceq_o^* cm.msig \\ C & \textbf{if } A \prec_d B \textbf{ and } C = \text{LOOKUP}(B, cmsig) \\ \bot & \textbf{otherwise} \end{cases}$$

In addition, we define a set of applicable methods in terms of a desired method call. This is used in the elaboration of the method call actions as described in Section B.1.4.

Elaborated Function 2.7 Given the desired method call cm, the set of applicable methods is defined as follows:

$$\text{APP}(cm) = \{cn \mid cm \preceq_s cn \ \wedge \ cn \in \text{METHODS}^*(cm.c)\}$$

This allows us to define the most applicable method for a given call, if it exists.

Elaborated Function 2.8 (Most Applicable Method) The applicability predicate is defined as follows:

$$\underline{\text{APP}}(cm) = cn \textbf{ such that } \forall cp \in \text{APP}(cm).\ cn \preceq_s cp$$

Statechart Diagram Elaborated Functions The following elaborated function are used in the dynamic semantics:

static	DEACTIVATES	:	$Trans \rightarrow \mathcal{S}(State)$
static	ACTIVATES	:	$Trans \rightarrow \mathcal{S}(State)$
static	INITSTATES	:	$Class \rightarrow \mathcal{S}(State)$
static	$\oslash$	:	$\mathcal{R}(Trans)$

The $\oslash$ relation indicates that two transitions are in conflict with respect to the state they deactivate. Definition of these functions requires the following auxiliary functions:

static	CHILD*	:	$State \rightarrow \mathcal{S}(State)$
static	ENC	:	$Trans \rightarrow \mathcal{S}(State)$
static	SCOPE	:	$Trans \rightarrow State$
static	ENTERS	:	$State \times Trans \rightarrow \mathcal{S}(State)$

The function CHILD* returns every child of a state. It is defined inductively as follows.

Elaborated Function 2.9 (State Hierarchy)

$$\text{CHILD}^*(s) = \begin{cases} \{s\} & \textbf{if } \text{CHILD}(s) = \{s\} \\ \bigcup s' \in \text{CHILD}(s) \,.\, \text{CHILD}^*(s') & \textbf{else} \end{cases}$$

The function ENC returns every state that contains both the source state and the target state of a transition.

Elaborated Function 2.10 (States Enclosing a Transition)

$$\text{ENC}(t) = \{s : State \mid t.src \in \text{CHILD}^*(s) \ \wedge \ t.tgt \in \text{CHILD}^*(s)\}$$

The scope of a transition is its lowest enclosing state, which is unique if the statechart is well-formed.

Elaborated Function 2.11 (Scope of a Transition)

$$\text{SCOPE}(t) = s \textbf{ such that } s \in \text{ENC}(t) \textbf{ and } \forall\, s' \in \text{ENC}(t)\ .\ s \in \text{CHILD}^*(s')$$

The function ENTERS returns the states that are activated upon entering a state s_1 as a result of the firing of a transition with target state s_2. As a shorthand notation, we use $\underline{\text{CHILD}}(s)$ instead of $\text{CHILD}(s) \setminus \{s\}$.

Elaborated Function 2.12 (Entering a State) Given a transition t and a state s, ENTERS is inductively defined as follows:

$$\text{ENTERS}(s_1, s_2) =$$

$$\begin{cases} \{s_1\} \cup \bigcup_{s' \in \underline{\text{CHILD}}(s_1)} \text{ENTERS}(s', s_2) & \textbf{if } s_1.ty = \text{AND} \\ \{s_1\} \cup \{s' \mid s' \in \text{CHILD}(s_1) \ \wedge\ s'.ty = \text{INIT}\} & \textbf{if } s_1.ty = \text{OR } \textbf{and } s_2 \notin \underline{\text{CHILD}}(s_1) \\ \{s_1\} \cup \text{ENTERS}(s_2, s_2) & \textbf{if } s_1.ty = \text{OR } \textbf{and } s_2 \in \underline{\text{CHILD}}(s) \\ \{s_1\} & \textbf{else} \end{cases}$$

The main elaborated functions are defined below.

Elaborated Function 2.13 (Deactivated States) Given a transition t, the function DEACTIVATES is defined as follows:

$$\text{DEACTIVATES}(t) = \begin{cases} \text{CHILD}^*(\text{SCOPE}(t)) & \textbf{if } t.tgt.ty \neq \text{FINAL} \\ \text{CHILD}^*(\text{SCOPE}(t)) & \textbf{if } t.tgt.ty = \text{FINAL} \\ & \textbf{and } \text{UP}(\text{UP}(t.tgt)).ty \neq \text{AND} \\ \text{CHILD}^*(\text{SCOPE}(t)) & \textbf{if } t.tgt.ty = \text{FINAL} \\ & \textbf{and } \text{UP}(\text{UP}(t.tgt)).ty = \text{AND} \\ & \textbf{and } \text{UP}(\text{UP}(t.tgt)) \in \text{SCOPE}(t) \\ \text{CHILD}^*(\text{UP}(\text{UP}(t.tgt))) & \textbf{else} \end{cases}$$

Elaborated Function 2.14 (Activated States) Given a transition t, ACTIVATES is defined as follows:

$$\text{ACTIVATES}(t) = \begin{cases} \text{ENTERS}(\text{SCOPE}(t), t.tgt) & \textbf{if } t.tgt.ty \neq \text{FINAL} \\ \emptyset & \textbf{else} \end{cases}$$

Elaborated Function 2.15 (Initial States) Given a class C , INITSTATES is defined as follows:

$$\text{INITSTATES}(C) = \bigcup_{C \preceq_h D} \text{INITSTATES}(\text{TOP}(D))$$

Elaborated Function 2.16 (Conflicting Transitions) Two transitions t_1, t_2 are in conflict if:

$$t_1 \oslash t_2 \Leftrightarrow t_1 \neq t_2 \textbf{ and } \text{DEACTIVATES}(t_1) \cap \text{DEACTIVATES}(t_2) \neq \emptyset$$

B.1.3 Well-Formedness

The following well-formedness constraints apply to the $\Upsilon_{\texttt{stat}}$ vocabulary. The notation $exp \vdash \tau$ means that the OCL expression exp is assigned type τ by the typing rules of Section C.1.

OCL Expression Well-formed Model 2.1 An OCL expression is well-formed is it is well-typed.

Class Diagram Well-Formedness

Well-formed Model 2.2 (Direct Inheritance Relation) A class has only one superclass:

$$\textbf{if } A \prec_d B \textbf{ and } A \prec_d C \textbf{ then } B = C$$

Well-formed Model 2.3 (Direct Inheritance Relation) The inheritance $\preceq_h$ relation must be acyclic:

$$\textbf{if} A \preceq_h B \textbf{ and } B \preceq_h A \textbf{ then } A = B$$

Action Well-Formedness

Well-formed Model 2.4 (Well-formed Action) Consider a transition t and an action $a \in t.a$. Assume that $\text{SCOPE}(t) \in \text{STATES}(C)$. The action is well-formed if:

- $a.ty = \text{SIGNAL}$
- $a.ty = \text{NEW}$ and $a.cf \in \text{FIELDS}^*(C)$
- $a.ty = \text{DEL}$ and $a.cf \in \text{FIELDS}^*(C)$
- $a.ty = \text{ASSIGN}$, $a.cf \in \text{FIELDS}^*(C)$,
  - $a.exp \vdash \tau : Type$ and
  - $\tau \sqsubseteq a.cf.ty$
- $a.ty = \text{MCALL}$,
  - $a.rcv \vdash D$,
  - $a.exps = \langle e_1, \ldots, e_n \rangle$,
  - $e_i \vdash \tau_i$ for $i = 1 \ldots n$ and
  - $\underline{\text{APP}}(D, (a.cmsig.msig, \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle)) \neq \bot$
- $a.ty = \text{MRETURN}$,
  - $a.msig \in \text{METHODS}^*(C)$, and
  - $a.exp \vdash \tau$ and $\tau \sqsubseteq a.msig.r.ty$

Transition Well-Formedness

Well-formed Model 2.5 (Well-formed Guard) Consider a transition t and the guard $t.g$. Assume that $\text{SCOPE}(t) \in \text{STATES}(C)$. The guard is well-formed if:

$$t.g \vdash \text{BOOL}$$

I.e. if it has the type BOOL.

State Diagram Well-Formedness

Well-formed Model 2.6 (OR-state Structure) An OR-state s is well-formed if:

$$!\exists\, s' \in \underline{\text{CHILD}}(s)\ .\ s'.ty = \text{INIT}$$

Well-formed Model 2.7 (AND-state Structure) An AND-state s is well-formed if:

$$\underline{\text{CHILD}}(s) \neq \emptyset \textbf{ and } \forall\, s' \in \underline{\text{CHILD}}(s)\ .\ s'.ty = \text{OR}$$

Well-formed Model 2.8 (State Type)

$$\forall\, s : State\ \textbf{.}\ \underline{\text{CHILD}(s)} \neq \emptyset\ \Rightarrow\ s.ty = \text{OR}\ \textbf{or}\ s.ty = \text{AND}$$

Well-formed Model 2.9 (State Hierarchy)

$$\begin{array}{ll} \forall\, s, s' : State\ \textbf{.} & s \neq s'\ \Rightarrow\ \text{CHILD}(s) \cap \text{CHILD}(s') = \emptyset \\ \textbf{and} & \text{CHILD}(s) = \text{CHILD}(s')\ \Rightarrow\ s = s' \\ \textbf{and} & \text{UP}(s) = s'\ \Rightarrow\ s \in \text{CHILD}(s') \\ \textbf{and} & \text{UP}(s) = \bot\ \Leftrightarrow\ \exists\, C : Class\ .\ s = \text{TOP}(C) \end{array}$$

Well-formed Model 2.10 (Transitions & Statecharts)

$$\forall\, t : \text{TRANS}\ \textbf{.}\ !\exists\, C \in \text{CLASSES}\ \textbf{.}\ \text{SCOPE}(t) \in \text{STATES}(C)$$

Well-Formed Statechart Defining whether a transition is well-formed with respect to the states it activates and deactivates requires the notion of state configuration. A state configuration $SC : \mathcal{S}(State)$ is a set of active states for a given statechart.

Well-formed Model 2.11 (AND-state Configuration) A state configuration $SC : \mathcal{S}(State)$ is well-formed with respect to an AND-state s if:

$$\textbf{if } s \in SC \textbf{ then } \forall\, s' \in \underline{\text{CHILD}}(s)\ .\ s' \in SC$$

Well-formed Model 2.12 (OR-state Configuration) A state configuration $SC : \mathcal{S}(State)$ is well-formed with respect to an OR-state s if:

$$\textbf{if } s \in SC \textbf{ then } !\exists\, s' \in \underline{\text{CHILD}}(s)\ .\ s' \in SC$$

Well-formed Configuration 2.1 (State Configuration) A state configuration $SC : \mathcal{S}(State)$ is well-formed if it is well-formed with respect to every AND-state and every OR-state.

Well-formed Model 2.13 (Well-formed Transition) A transition t is well-formed if:

- Every action in $t.a$ is well-formed
- The guard $t.g$ is well-formed
- Given a well-formed state configuration $SC : \mathcal{L}(State)$, then:
  - $(SC \setminus \text{DEACTIVATES}(t)) \cup \text{ACTIVATES}(t)$ is also a well-formed state configuration

Well-formed Model 2.14 (Statechart) A statechart is well-formed if:

- Every AND and OR states are well-formed
- Every transition is well-formed

Well-Formed UML Model **Well-formed Model 2.15 (UML Model)** A UML model is well-formed if:

- The class diagram is well-formed
- Every statechart diagram is well-formed
- The object diagram yields a well-formed configuation (defined on page 115).

B.1.4 Method Call Actions

Method call actions are a special case because they have to be elaborated. Hence, the TRANS static function is partially elaborated.

Consider the action $a : ACall$. The model contains the OCL expression $a.rcv$, a method name m, a list of parameters $a.exps$ and the calling type $a.ty$. The method signature $a.cmsig$ is computed as follows. First, the type of $a.rcv$ is computed according to the typing rules given in Section C.1. This must be a class. Let C be that class. The method signature $msig$ is elaborated according to the type of each parameter. The complete method signature $a.cmsig$ is elaborated according to the APP function, i.e. $a.cmsig = \text{APP}(C, msig)$. The method exists if the action is well-formed.

Dynamically, however, antother method may be called. The LOOKUP function is used to verify whether an overridden method is available. For example, consider the class diagram of Fig. B.1.

Assume that a transition of the **Client**'s statechart specifies the following action:

$$\text{self}.b.m(\text{self}.c)$$

This corresponds to a virtual method call. The expression providing the parameter value is of type B. Hence, the extracted action is the following tuple:

$$(\text{self}.b, (B, (m, \langle (B, \bot) \rangle, \bot)), \langle \text{self}.c \rangle, \text{VIRTUAL})$$

Now, the set of applicable methods is computed from the $(B, (m, \langle (x, B) \rangle, \bot))$ method signature. This gives the following set.

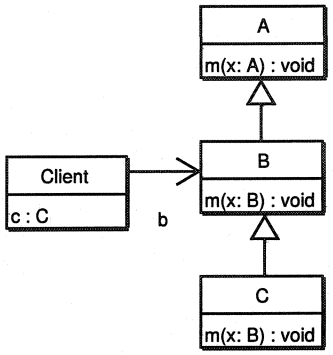


Figure B.1: Method Inheritance Example

$$\{(A, (m, \langle (A, x) \rangle, (\bot, r))), (B, (m, \langle (B, x) \rangle, (\bot, r)))\}$$

Clearly, the method defined in class **B** is more specific. The elaborated action is as follows:

$$(\text{self}.b, (B, (m, \langle (B, x) \rangle, (\bot, r))), \langle \text{self}.c \rangle, \text{VIRTUAL})$$

Now, assume that the action is fired with the b field referencing an object of type **C**. As the calling type is virtual, the LOOKUP function is used. The overridden method in class **C** will be called.

B.2 Configuration Vocabulary

B.2.1 Extracted Information

State-Machines and Signals

dynamic *signals* : $\mathcal{S}(Signal)$
dynamic *actstates* : $Obj \rightarrow \mathcal{S}(State)$

Values

sort	Int	$\equiv$	$\{\ldots, -1, 0, 1, \ldots\}$
sort	$Bool$	$\equiv$	$\{\text{TRUE}, \text{FALSE}\}$
sort	$SimpleVal$	$=$	$Int \mid Bool \mid Obj$
sort	$ListVal$	$=$	$\mathcal{L}(SimpleVal)$
sort	Val	$=$	$SimpleVal \mid ListVal$

Objects

sort	$ObjEnv$	$=$	$ClassField \mapsto Val$
sort	$Heap$	$=$	$\mathcal{T}(c : Class, env : ObjEnv)$
dynamic	$heap$	$:$	$Obj \rightarrow Heap$

Methods and Threads

sort	Var		
sort	$FrameEnv$	$=$	$Var \mapsto Val$
sort	$Frame$	$=$	$\mathcal{T}(mth : ClassMSig, env : FrameEnv, snd : Obj,$ $rcv : Obj)$
sort	$CallStack$	$=$	$Frame^*$
dynamic	$cont$	$:$	$Obj \rightarrow CallStack$
dynamic	$returns$	$:$	$Obj \rightarrow \mathcal{T}(msig : MSig, v : Val)$

B.2.2 Well-Formedness

Well-formed Configuration 2.2 (State-Machine) The function $actstates$ is well-formed if for every object o, $actstates(o)$ is a well-formed state configuration (defined on page 111).

Well-formed Configuration 2.3 (Well-Typed Value) Let $v : Val$ and $\tau : Type$, we say that v is well-typed if:

$$\boxdot(v, \tau) \Leftrightarrow \begin{cases} \tau = \text{BOOL} \Rightarrow v \in Bool \\ \tau = \text{INT} \Rightarrow v \in Int \\ \tau \in \text{CLASSES} \Rightarrow v \in Obj \textbf{ and } heap(v).o \preceq_h \tau \\ \tau = \mathcal{L}(\tau') \Rightarrow v \in \mathcal{L}(\tau) \textbf{ and } \forall\, v' \in v \,.\, \boxdot(v', \tau') \end{cases}$$

Well-formed Configuration 2.4 (Environment) For every object $o : Obj$ such

that $heap(o) \neq \bot$. Let C be that object's class (i.e. $C = heap(o).c$). Then we must have:

$$\forall\, (D, f) \in \text{FIELDS}^*(C)\ .\ heap(o).env((D, f)) \neq \bot$$

Well-formed Configuration 2.5 (Objects) Each value in the environment is well-typed.

$$\forall\, o \in Obj\ .\ \forall\, cf \in \text{FIELDS}^*(heap(o).c)\ \Rightarrow\ \boxdot(heap(o).env(cf), cf.ty)$$

Well-formed Configuration 2.6 (Calling Stack) For every object o such that $heap(o).c \in$ THREADS, then $cont(o) \neq \bot$. Furthermore, for every frame $f \in cont(o)$, the sender and the receiver object exists, i.e. $heap(f.snd) \neq \bot$ and $heap(f.rcv) \neq \bot$.

Well-formed Configuration 2.7 (Method Returns) The returned value is of the correct type. For every object o such that $heap(o).c \in$ THREADS and $returns(o) \neq \bot$. Let $(msig, v) = returns(o)$, then we have:

$$\boxdot(v, msig.r.ty)$$

Well-formed Configuration 2.8 (Configuration) A UML model configuration is well-formed if the functions *heap*, *cont* and *returns* are well-formed.

B.2.3 Initial Configuration

Initial Configuration 2.1 (Active States) Initially, active states correspond to INITSTATES:

$$actstates(o) = \text{INITSTATES}(heap(o).c)$$

Initial Configuration 2.2 (Signals) Initially, the set of active signals is empty, that is:

$$signals = \emptyset$$

Initial Configuration 2.3 In the initial configuration, every thread has the following calling stack:

$$\forall\, o : Obj\ .\ heap(o).c \in \text{THREADS} \Rightarrow cont(o) = \langle(\bot, \emptyset, o, o, \langle\rangle)\rangle$$

This is necessary for the thread to be active.

B.3 Additional Vocabulary

Temporary Configuration

dynamic	$\overrightarrow{signals}$	:	$\mathcal{S}(Signal)$
dynamic	$\overrightarrow{actstates}$	:	$Obj \to \mathcal{S}(State)$
dynamic	$\overrightarrow{heap}$	:	$Obj \to Heap$
dynamic	$\overrightarrow{cont}$	:	$Obj \to CallStack$
dynamic	$\overrightarrow{returns}$	:	$Obj \to \mathcal{T}(msig : MSig, env : FrameEnv)$

Current Values

dynamic	$\tilde{th}$	:	Obj	*Current thread*
dynamic	$\tilde{o}$	:	Obj	*Current object*
dynamic	$\tilde{t}$	:	$Trans$	*Current transition*
dynamic	$\tilde{a}$	:	$\mathcal{L}(Action)$	*Current actions*

Iteration

dynamic	$iterObj$	:	$Obj \to Bool$
dynamic	$iterTrans$	:	$Obj \times Trans \to Bool$

OCL Expressions

external	$\vdash$	:	$OclExp \times Type$
external	$[\![\_]\!]\_$	:	$OclExp \times OclEnv \to Val$
external	$[\![\_]\!]^{\mathcal{L}}\_$	:	$\mathcal{L}(OclExp) \times OclEnv \to \mathcal{L}(Val)$

Processing Transitions Notice how the *enabledSet* and $fireSet$ contain couples holding an object and a transition. The object indicates the state-machine to which the transition belongs.

dynamic $enabledSet$: $\mathcal{S}(\mathcal{T}(Obj, Trans))$
dynamic $fireSet$: $\mathcal{S}(\mathcal{T}(Obj, Trans))$

Object Creation

static NEWOBJ : $Class \rightarrow Heap$

Consistency Checks

dynamic diff : $Obj \times ClassField \rightarrow Int$
dynamic $objEnvUpdates$: $Obj \times ClassField \rightarrow Val$
dynamic $stepCall$: $Bool$

End of Computation

dynamic $done$: $Bool$

B.3.1 Well-Formedness

Note that these well-formedness constraints are re-evaluated before every step computation. This allows us to consider newly created objects, for example.

Additional Vocabulary 2.1 (Object Iteration)

$$\forall\, o \in Obj\ .\ iterObj(o) = \textbf{true}$$

Additional Vocabulary 2.2 (Transition Iteration)

$$iterTrans(t, o) = \textbf{true} \Leftrightarrow \text{SCOPE}(t) \in \text{STATES}^*(heap(o).c)$$

Additional Vocabulary 2.3 (Enabled Set)

$$enabledSet = \emptyset$$

Additional Vocabulary 2.4 (Fire Set)

$$fireSet = \emptyset$$

Additional Vocabulary 2.5 (Initial Object) The initial object $\tilde{o}$ is $\perp$.

Additional Vocabulary 2.6 (Initial Transition) The initial transition $\tilde{t}$ is $\perp$.

Additional Vocabulary 2.7 (Multiplicity)

$$\forall\, o \in Obj\ .\ \mathrm{diff}(o) = 0$$

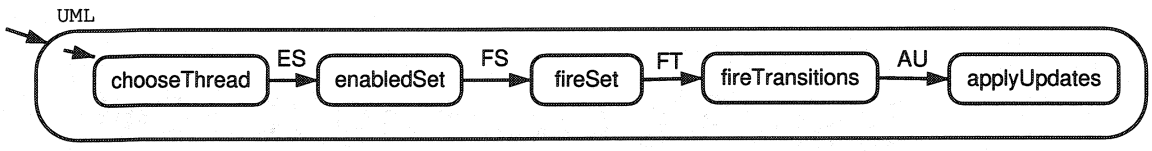
Additional Vocabulary 2.8 (Assignation Coherence)

$$\forall\, o \in Obj\ .\ \forall\, cf \in \textsc{fields}^*(heap(o).c)\ .\ objEnvUpdates(o, cf) = \perp$$

Additional Vocabulary 2.9 (Step Method Call)

$$stepCall = \textbf{false}$$

B.4 Transition Rule



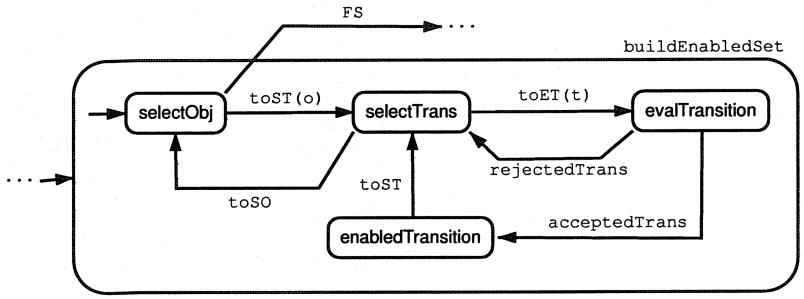
B.4.1 Choosing a Thread

$\Pi_{\texttt{chooseThread}}$

$\epsilon_{o:Obj} \cdot heap(o).c \in \text{THREADS} :$
$\quad \| \ \tilde{th} := o$
$\quad \| \ \Pi_{\texttt{ES}}$

B.4.2 Collecting Enabled Transitions

Control Flow



$\Pi^*_{\texttt{toST}}(o)$

$\| \ \tilde{o} := o$

$\Pi^*_{\texttt{toET}}(t)$

$\| \ \tilde{t} := t$
$\| \ \tilde{a} := t.a$

$\Pi^*_{\texttt{acceptedTrans}}$

$\| \ \text{enabledSet} := \text{enabledSet} \cup \{\tilde{t}\}$

Iterating Objects and Transitions

$\Pi_{\texttt{selectObj}}$

$\epsilon_{\text{o:Obj}} \cdot \text{iterObj(o)}$
$\quad ? \ \text{iterObj(o)} := \textbf{false}$
$\quad\quad \| \ \Pi_{\texttt{toST}}(\text{o})$
$\quad : \ \Pi_{\texttt{FS}}$

In the following, notice that a transition is iterated only if it is enabled with respect to its source state and its guard. This is verified by the $\Phi_{\texttt{enabled}}(o,t)$ predicate. Notice also the *diff* function is reinitialized. This function allows us to verify whether the object creation and deletion actions of a transition respect the multiplicity requirements of the reference field.

$\Pi_{\texttt{selectTrans}}$

$\in_{t \in \text{TRANS}} \cdot \text{iterTrans}(o, t) \wedge \Phi_{\texttt{enabled}}(\tilde{o}, t)$
? $\text{iterTrans}(\tilde{o}, t) := \textbf{false}$
|| $\Sigma_{cf \in \text{FIELDS}^*(\text{heap}(\tilde{o}).c)} : \text{diff}(o, cf) := 0$
|| $\Pi_{\texttt{toET}}(t)$
: $\Pi_{\texttt{toSO}}$

In the following predicate, we use the `OCL` expression evaluation function. Note that the variable environment ζ is always:

$$
\begin{array}{ll}
 & cont(\tilde{th}).env \\
\oplus & \{returns(\tilde{th}).msig.r.p \mapsto returns(\tilde{th}).val\} \\
\oplus & \{\text{SELF} \mapsto \tilde{o}\}
\end{array}
$$

$\Phi_{\texttt{enabled}}(o, t)$

$$
\begin{array}{ll}
 & t.src \in actstates(o) \\
\wedge & t.ty = \text{SIGNAL} \Rightarrow t.tr.s \in signals \\
\wedge & t.ty = \text{MCALL} \Rightarrow t.tr.cmsig = \texttt{top}(cont(\tilde{th})).meth \\
\wedge & t.ty = \text{MRETURN} \Rightarrow t.tr.cmsig.msig = returns(\tilde{th}).msig \\
\wedge & [\![t.g]\!]_\zeta = \textbf{true}
\end{array}
$$

Evaluating Actions The following rule verifies the possibility of firing a given action. A signal action can always be fired. Other actions correspond to a method's behavior, so they can be fired only if the object is currently executing a method. The predicate $\Phi_{\texttt{activeBehavior}}(o, t)$ also verifies that the transition belongs to the class defining the method. This is part of the behavior inheritance mechanism.

$\Pi_{\texttt{evalTransition}}$

$$
\begin{array}{l}
\tilde{a} \neq \langle\rangle \\
\quad ? \;\; \| \;\; \tilde{a} := \texttt{tail}(\tilde{a}) \\
\qquad \| \;\; a := \texttt{hd}(\tilde{a}) : \\
\qquad\quad a.ty = \text{SIGNAL} \\
\qquad\qquad ? \;\; \circ \\
\qquad\qquad : \;\; \Phi_{\texttt{activeBehavior}}(\tilde{o}, t) \\
\qquad\qquad\quad ? \;\; \Pi_{\texttt{evalAction}}(a) \\
\qquad\qquad\quad : \;\; \Pi_{\texttt{rejectedTrans}} \\
\quad : \quad \Pi_{\texttt{acceptedTrans}}
\end{array}
$$

$\Phi_{\texttt{activeBehavior}}(o, t)$

$$
\begin{array}{ll}
 & \texttt{top}(cont(\tilde{th})).rcv = o \\
\wedge & t.src \in \text{STATES}(\texttt{top}(cont(\tilde{th})).cmsig.c)
\end{array}
$$

The case of an assignation, object creation, or deletion calls for verification that multiplicity is not violated. This is done by computing the resulting size of a reference field upon the firing of the action and comparing it with the declared multiplicity (using the $\Phi_{\texttt{violatesMult}}(s, f)$ predicate). As a transition may fire many such actions, the $diff$ function is used to accumulate the size modifications.

$\Phi_{\texttt{violatesMult}}(s, f)$

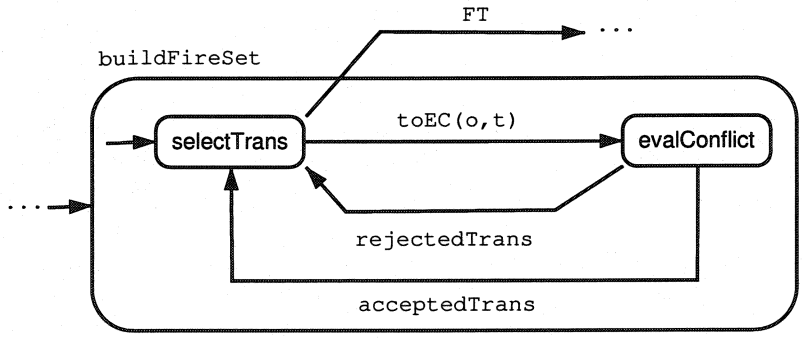
$$
\begin{array}{ll}
 & f.mult.up \neq \bot \;\Rightarrow\; s > f.mult.up \\
\wedge & f.mult.low \neq \bot \;\Rightarrow\; s < f.mult.low \\
\wedge & f.mult.low = \bot \;\Rightarrow\; s \leq 0
\end{array}
$$

$\Pi_{\texttt{evalAction(a)}}$

case $a.ty$ **of**
  NEW $\rightarrow$
    $d := diff(\tilde{o}, a.cf) + 1$:
      $\|$ $diff(\tilde{o}, a.cf) := d$
      $\|$ $\Phi_{\texttt{violatesMult}}(\texttt{size}(heap(\tilde{o}).env(a.cf)) + d, a.cf.f)$? $\Pi_{\texttt{rejectedTrans}}$
  DEL $\rightarrow$
    $d := diff(\tilde{o}, a.cf) - 1$:
      $\|$ $diff(\tilde{o}, a.cf) := d$
      $\|$ $\Phi_{\texttt{violatesMult}}(\texttt{size}(heap(\tilde{o}).env(a.cf)) + d, a.cf.f)$? $\Pi_{\texttt{rejectedTrans}}$
  ASSIGN $\rightarrow$
    $a.cf.ty \in ListType$
      ? $\Phi_{\texttt{violatesMult}}(\texttt{size}([\![a.exp]\!]_\zeta), a.cf.f)$? $\Pi_{\texttt{rejectedTrans}}$
      : o

B.4.3 Selecting Transitions to Fire

Control Flow



$\Pi^*_{\texttt{FS}}$

$\Sigma_{o:Obj} : \Sigma_{cf \in \text{FIELDS}^*(heap(o).c)} : diff(\tilde{o}, cf) := 0$

$\Pi^*_{\texttt{toEC}}(o,t)$

$\|$ $\tilde{o} := o$
$\|$ $\tilde{t} := t$
$\|$ $\tilde{a} := t.a$

$\Pi^*_{\texttt{toEC}}(o,t)$

$\|$ $\tilde{o} := o$
$\|$ $\tilde{t} := t$
$\|$ $\tilde{a} := t.a$

$\Pi^*_{\texttt{acceptedTrans}}$

$\|$ $fireSet := fireSet \cup \{(\tilde{o}, \tilde{t})\}$

Selecting Transitions When building the $fireSet$, a transition is selected for addition if it is in the *enabledSet* and if it is not conflicting with a transition already added to the $fireSet$ (using the $\Phi_{\texttt{fireable}}(o,t)$ predicate).

$\Pi_{\texttt{selectTrans}}$

$\epsilon_{o:Obj,\, t \in \text{TRANS}} \cdot \Phi_{\texttt{fireable}}(o,t)$:
  ? $enableSet := enabledSet \setminus \{(o,t)\}$
    || $\Pi_{\texttt{toEC}}(o,t)$
  : $\Pi_{\texttt{FT}}$

$\Phi_{\texttt{fireable}}(o,t)$

$(o,t) \in enabledSet$
$\wedge \quad \neg(\exists_{t':Trans} \cdot (o,t') \in fireSet \ \wedge \ t \oslash t')$

Evaluating Conflict Furthermore, the following rule determines whether adding the transition to the $fireSet$ would create a conflict. If so, the transition is rejected. Signal actions are always accepted. Object creation and deletion actions are rejected if they produce a multiplicity violation. Again, this is done with the $\Phi_{\texttt{violatesMult}}(s,f)$ predicate. However, the $diff$ function is initialized only once, before computing the $fireSet$. Hence, the size modifications are accumulated for each transition added to the $fireSet$. A transition is rejected if it violates multiplicity with respect to the effects of the previously added transitions. In the case of assignation action, the function *objEnvUpdate* and the $\Phi_{\texttt{coherentUpdate}}(o,cf,v)$ are used to verify whether the assignation is consistent with the assignations of the previously added transitions. In the case of method call and method return actions, the *stepCall* function is used to enforce the requirement that only one such action may be fired in one step.

$\Pi_{\texttt{evalConflict}}$

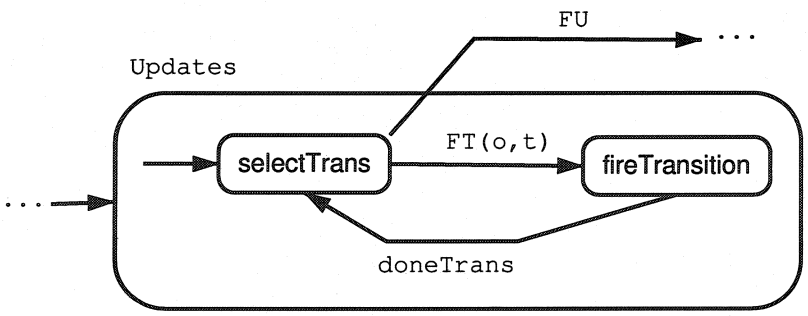
$\tilde{a} \neq \langle\rangle$
? $\|$ $\tilde{a} := \texttt{tail}(\tilde{a})$
$\|$ $a := \texttt{hd}(\tilde{a})$:
case $a.ty$ **of**
SIGNAL $\rightarrow$
$\circ$
NEW $\rightarrow$
$d := diff(\tilde{o}, a.cf) + 1$:
$\|$ $diff(\tilde{o}, a.cf) := d$
$\|$ $\Phi_{\texttt{violatesMult}}(\texttt{size}(heap(\tilde{o}).env(a.cf)) + d, a.cf.f)$
? $\Pi_{\texttt{rejectedTrans}}$
DEL $\rightarrow$
$d := diff(\tilde{o}, a.cf) - 1$:
$\|$ $diff(\tilde{o}, a.cf) := d$
$\|$ $\Phi_{\texttt{violatesMult}}(\texttt{size}(heap(\tilde{o}).env(a.cf)) + d, a.cf.f)$?
? $\Pi_{\texttt{rejectedTrans}}$
ASSIGN $\rightarrow$
$\Phi_{\texttt{conherentUpdate}}(\tilde{o}, a.cf, [\![a.e]\!]_\varsigma)$
? $objEnvUpdate(\tilde{o}, a.cf) := [\![a.e]\!]_\varsigma$
: $\Pi_{\texttt{rejectedTrans}}$
MCALL
MRETURN $\rightarrow$
$stepCall$
? $\Pi_{\texttt{rejectedTrans}}$
: $\|$ $stepCall :=$ **true**
$\|$ $\Pi_{\texttt{acceptedTrans}}$
: $\Pi_{\texttt{acceptedTrans}}$

$\Phi_{\texttt{coherentUpdate}}(o, cf, v)$

$objEnvUpdate(o, cf) = \bot$
$\vee$ $objEnvUpdate(o, cf) = v$

B.4.4 Firing Transitions

Control Flow



$\Phi^*_{\texttt{toFT}}(o,t)$

$\| \ \tilde{o} := o$
$\| \ \tilde{t} := t$
$\| \ \tilde{a} := t.a$

Select Transition

$\Pi_{\texttt{selectTrans}}$

$\epsilon_{o:Obj,\ t \in \text{TRANS}} \bullet (o,t) \in fireSet :$
$\quad ?\ fireSet := fireSet \setminus \{(o,t)\}$
$\quad\quad \| \ \overrightarrow{actstates}(o) := (\overrightarrow{actstates}(o) \setminus \text{DEACTIVATES}(t)) \cup \text{ACTIVATES}(t)$
$\quad\quad \| \ \Pi_{\texttt{toFT}}(o,t)$
$\quad : \Pi_{\texttt{AU}}$

Fire Actions In the following, note that $\oplus$ updates a map.

$\Pi_{\texttt{fireAction}}$

$\tilde{a} \neq \langle\rangle$
  ? || $\tilde{a} := \texttt{tail}(\tilde{a})$
  || $a := \texttt{hd}(\tilde{a})$:
  case $a.ty$ **of**
    SIGNAL $\rightarrow$
      $\overrightarrow{signals} := \overrightarrow{signals} \cup \{a.s\}$
    NEW $\rightarrow$
      $\Pi_{\texttt{createObj}}(\tilde{o}, a.cf, a.cf.f.t)$
    DEL $\rightarrow$
      $C := heap(\tilde{o}).c$:
      $env := heap(\tilde{o}).env$:
      $val := \texttt{hd}(env(cf))$:
        $\overrightarrow{heap}(\tilde{o}) := (C, env \oplus \{a.cf \mapsto val\})$
    ASSIGN $\rightarrow$
      $C := heap(\tilde{o}).c$:
      $env := heap(\tilde{o}).env$:
      $val := [\![a.exp]\!]_\varsigma$:
        $\overrightarrow{heap}(\tilde{o}) := (C, env \oplus \{a.cf \mapsto val\})$
    MCALL $\rightarrow$
      $\epsilon_{o:Obj} \cdot o \in [\![a.rcv]\!]_\varsigma$:
        $\Pi_{\texttt{fireCallAction}}(a.cmsig, o, [\![a.exps]\!]_\varsigma, heap(\tilde{o}).c, a.ty)$
    MRETURN $\rightarrow \Pi_{\texttt{popMethod}}$
  : || $\Pi_{\texttt{doneTrans}}$

In the following, notice how a new machine is instanciated when the object is created. This is done by setting its active states using INITSTATES.

$\Pi_{\texttt{createObj}}(o, cf, C)$

$\partial_{o':Obj}$:
  || $env := \overrightarrow{heap}(\tilde{o}).env$:
    $val := o' :: env(cf)$:
      $\overrightarrow{heap}(\tilde{o}) := (C, env \oplus \{cf \mapsto val\})$
  || $\overrightarrow{heap}(o') :=$ NEWOBJ(C)
  || $\overrightarrow{actstates}(o') :=$ INITSTATES(C)
  || $\Pi_{\texttt{createThread}}(C, o')$

$\Pi_{\texttt{fireCallAction}}(cmsig, rcv, vals, ctxt, ty)$

case ty **of**
  VIRTUAL $\rightarrow$
    $\Pi_{\texttt{pushMethod}}(cmsig.msig, \text{LOOKUP}(ctxt, cmsig), vals, \tilde{o}, rcv)$
  SUPER $\rightarrow$
    $\Pi_{\texttt{pushMethod}}(cmsig.msig, \text{LOOKUP}(\text{SUPER}(ctxt), cmsig), vals, \tilde{o}, rcv)$

In the following, notice how the initial states of an object are reactivated when a method call is pushed. This allows the object to answer the call.

$\Pi_{\texttt{pushMethod}}(msig, c, vals, snd, rcv)$

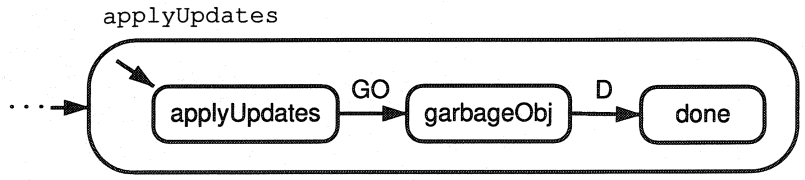
$\| \quad \overrightarrow{actstates(\tilde{o})} := \overrightarrow{actstates(\tilde{o})} \cup \text{INITSTATES}(c)$
$\| \quad env := \texttt{buildMap}(msig.p, vals)$:
$\qquad \overrightarrow{cont}(\tilde{th}) := \texttt{push}(cont(\tilde{th}), ((c, msig), env, snd, rcv))$

$\Pi_{\texttt{popMethod}}$

$\| \quad \overrightarrow{cont}(\tilde{th}) := \texttt{pop}(cont(\tilde{th}))$
$\| \quad returns(\tilde{th}) := (a.msig, [\![a.exp]\!]_\varsigma)$

B.4.5 Applying Updates

Control Flow



Applying Updates

$\Pi_{\texttt{fireUpdates}}$

$$
\begin{array}{l}
\| \quad signals := \overrightarrow{signals} \\
\| \quad \Sigma_{o:Obj} : actstates(o) := \overrightarrow{actstates}(o) \\
\| \quad \Sigma_{o:Obj} : heap(o) := \overrightarrow{heap}(o) \\
\| \quad cont(\tilde{th}) := \overrightarrow{cont}(\tilde{th}) \\
\| \quad returns(\tilde{th}) := \overrightarrow{returns}(\tilde{th}) \\
\| \quad \Pi_{\texttt{GO}}
\end{array}
$$

Garbaging Objects

$\Pi_{\texttt{garbageObjects}}$

$$
\begin{array}{l}
\| \quad \Sigma_{o:Obj} \cdot \neg\, \Phi_{\texttt{referencedObj}}(o) : heap(o) := \bot \\
\| \quad \Pi_{\texttt{D}}
\end{array}
$$

$\Phi_{\texttt{referencedObj}}(o)$

$$
\begin{array}{ll}
 & \Phi_{\texttt{heapReference}}(o) \\
\vee & \Phi_{\texttt{frameReference}}(o)
\end{array}
$$

$\Phi_{\texttt{heapReference}}(o)$

$$\exists\, o' : Obj\ .o \neq o'\ \wedge\ \exists\, cf : Field\ .\ o \in heap(o').env(cf)$$

$\Phi_{\texttt{frameReference}}(o)$

$$
\begin{array}{l}
\exists\, o' : Obj\ .\ heap(o').c \in \text{THREADS}\ \wedge\ \exists\, f : Frame\ .\ f \in cont(o')\ \wedge \\
\qquad o = f.rcv \\
\quad \vee\ o = f.snd \\
\quad \vee\ \exists\, v : Var\ .\ o \in f.env(v)
\end{array}
$$

End of Step Computation The update of the auxiliary function *done* indicates that the step computation is finished.

$\Pi_{\texttt{done}}$

$$\| \quad done := \textbf{true}$$

APPENDIX C

OCL SEMANTICS DETAILS

This chapter contains details about the semantics of OCL expressions and OCL constraints. It presents the type-checking of OCL expressions and the formal semantics of the adapted μ-calculus.

C.1 OCL Expressions Type System

Given an OCL expression e, its type is determined by recursively analyzing its subexpressions. The analysis is done according to the rules of Figure C.1. When a value, a unary, or a binary operator is reached in the analysis, its type is fetched from the predefined types listed in Figure C.2. The typing environment Γ is used to accumulate the types of the variables. In particular, we assume it holds SELF : τ, given that τ is the expression's context, which is always a class. If the OCL expression appears in a statechart, that statechart's class is the context. If the OCL expression appears in an OCL constraint, the class is explicitly specified.

$$(1)\quad \frac{}{\Gamma, exp : \tau \vdash exp : \tau}$$

$$(2)\quad \frac{}{\Gamma \vdash lit : \tau} \quad \text{if } lit :_d \tau$$

$$(3)\quad \frac{\Gamma \vdash exp_1 : \tau_1 \;\; \Gamma \vdash exp_2 : \tau_2}{\Gamma \vdash exp_1\ bop\ exp_2 : \tau_3} \quad \text{if } bop :_d \tau_1 \times \tau_2 \to \tau_3$$

$$(4)\quad \frac{\Gamma \vdash exp_1 : \tau_1}{\Gamma \vdash uop\ exp_1 : \tau_2} \quad \text{if } uop :_d \tau_1 \to \tau_2$$

$$(5)\quad \frac{\Gamma \vdash exp_1 : C \;\; \Gamma \vdash D/f : \tau}{\Gamma \vdash exp_1.D/f : \tau} \quad \text{if } C \preceq_h D \text{ and } (D, f) \in \text{FIELDS}^*(C)$$

$$(6)\quad \frac{\Gamma \vdash exp_1 : \text{L}(C) \;\; \Gamma \vdash D/f : \tau}{\Gamma \vdash exp_1.D/f : \text{L}(\tau)} \quad \text{if } \tau \neq \text{L}(\tau'),\ C \preceq_h D \text{ and } (D, f) \in \text{FIELDS}^*(C)$$

$$(7)\quad \frac{\Gamma \vdash exp_1 : \text{L}(C) \;\; \Gamma \vdash D/f : \text{L}(\tau)}{\Gamma \vdash exp_1.D/f : \text{L}(\tau)} \quad \text{if } C \preceq_h D \text{ and } (D, f) \in \text{FIELDS}^*(C)$$

$$(8)\quad \frac{\Gamma \vdash exp_1 : \text{L}(\tau_1) \;\; \Gamma \vdash exp_2 : \tau_2 \;\; \Gamma, var_1 : \tau_1, var_2 : \tau_2 \vdash exp_3 : \tau_3}{\Gamma \vdash exp_1.\textbf{iterate}(var_1;\ var_2 = exp_2 \mid exp_3) : \tau_3}$$

Figure C.1: OCL Expressions Type System

C.2 μ-Calculus Semantics

Given an ASM execution graph $\Theta = (\mathcal{S}, \to, \mathfrak{A}_0)$, the semantics of a μ-formula is presented in Fig. C.3. The function ϑ is a variable assignment that maps subsets of $\mathcal{S}$ to μ-variables. As usual, $\vartheta[X \mapsto S]$ denotes the updated assignment where X is mapped to S and other variables are mapped according to ϑ.

Note how the semantics maps a μ-formula to a set of states ($[\![\phi]\!]_\vartheta \subseteq \mathcal{S}$), which contains the states where the μ-formula holds. The execution graph Θ satisfies the μ-formula (written $\Theta \models \phi$) if $\mathfrak{A}_0 \in [\![\phi]\!]_\emptyset$, i.e. if the initial state satisfies the μ-formula

when evaluated with the empty variable assignment.

Val	*true*	$:_d$	BOOL	
	false	$:_d$	BOOL	
	$\ldots, -1, 0, 1, \ldots$	$:_d$	INT	
Bop	$+$	$:_d$	INT $\times$ INT $\rightarrow$ INT	Addition
	$-$	$:_d$	INT $\times$ INT $\rightarrow$ INT	Subtraction
	$*$	$:_d$	INT $\times$ INT $\rightarrow$ INT	Multiplication
	$/$	$:_d$	INT $\times$ INT $\rightarrow$ INT	Division
	$\%$	$:_d$	INT $\times$ INT $\rightarrow$ INT	Remainder
	$<$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Less than
	$<=$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Less than or equal to
	$>$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Greater than
	$>=$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Greater than or equal to
	$==$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Equal
	$!=$	$:_d$	INT $\times$ INT $\rightarrow$ BOOL	Not equal
	$\&$	$:_d$	BOOL $\times$ BOOL $\rightarrow$ BOOL	Boolean AND
	$\vert$	$:_d$	BOOL $\times$ BOOL $\rightarrow$ BOOL	Boolean OR
	$\vert\vert$	$:_d$	BOOL $\times$ BOOL $\rightarrow$ BOOL	Boolean XOR
	$\Rightarrow$	$:_d$	BOOL $\times$ BOOL $\rightarrow$ BOOL	Boolean IMPLIES
	@	$:_d$	$\mathrm{L}(\tau) \times \mathrm{L}(\tau) \rightarrow \mathrm{L}(\tau)$	List concatenation
	::	$:_d$	$\tau \times \mathrm{L}(\tau) \rightarrow \mathrm{L}(\tau)$	List construction
Uop	$+$	$:_d$	INT $\rightarrow$ INT	Unary plus
	hd	$:_d$	$\mathrm{L}(\tau) \rightarrow \tau$	Head of a list
	tail	$:_d$	$\mathrm{L}(\tau) \rightarrow \mathrm{L}(\tau)$	Tail of a list
	$-$	$:_d$	INT $\rightarrow$ INT	Unary minus
	$!$	$:_d$	BOOL $\rightarrow$ BOOL	Logical complement

Figure C.2: Predefined Types for OCL Expressions

$$
\begin{array}{lcl}
[\![\Phi]\!]_\vartheta & = & \{\mathfrak{A} \in \mathcal{S} \mid [\![\Phi]\!]^{\mathfrak{A}}_{\zeta_0} = \mathbf{true}\} \\
[\![X]\!]_\vartheta & = & \vartheta(X) \\
[\![\neg\phi]\!]_\vartheta & = & \mathcal{S} \setminus [\![\phi]\!]_\vartheta \\
[\![\phi_1 \vee \phi_2]\!]_\vartheta & = & [\![\phi_1]\!]_\vartheta \cup [\![\phi_2]\!]_\vartheta \\
[\![\diamond\phi]\!]_\vartheta & = & \{\mathfrak{A}' \in \mathcal{S} \mid \exists_{\mathfrak{A} \in [\![\phi]\!]_\vartheta} \cdot \mathfrak{A}' \rightarrow \mathfrak{A} \in \Theta\} \\
[\![\nu X.\phi]\!]_\vartheta & = & \bigcup\{S \subseteq \mathcal{S} \mid S \subseteq [\![\phi]\!]_{\vartheta[X \mapsto S]}\} \\
 & & \\
[\![\phi_1 \wedge \phi_2]\!]_\vartheta & = & [\![\neg(\neg\phi_1 \vee \neg\phi_2)]\!]_\vartheta \\
[\![\phi_1 \Rightarrow \phi_2]\!]_\vartheta & = & [\![\neg\phi_1 \vee \phi_2]\!]_\vartheta \\
[\![\Box\phi]\!]_\vartheta & = & [\![\neg \diamond \neg\phi]\!]_\vartheta \\
[\![\mu X.\phi]\!]_\vartheta & = & [\![\neg\nu X.\neg\phi]\!]_{\vartheta[X \mapsto [\![\neg X]\!]_\vartheta]}
\end{array}
$$

Figure C.3: μ-Calculus Semantics