Formalisation et automatisation de la mesure des points de fonction

par

Hassan Diab

thèse présentée au Département de mathématiques
et d'informatique en vue de l'obtention du grade de doctorat (Ph.D.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, juillet 2003
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.

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.

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 les extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

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.
Sommaire

La méthode des points de fonction, proposée par Albrecht, permet de mesurer la taille fonctionnelle d'un logiciel durant la phase de spécification des besoins. Elle a été améliorée par l'IFPUG (International Function Points User Group). Cette méthode des points de fonction a été étendue par le groupe COSMIC (Common Software Measurement International Consortium) pour la mesure des systèmes temps réels, appelée COSMIC-FFP. Cependant, les définitions de ces deux méthodes sont ambiguës, ce qui les rend difficiles à automatiser.

Dans cette thèse, nous avons formalisé la définition des points de fonction selon la méthode d'IFPUG pour mesurer des spécifications écrites en langage B. Nous avons également formalisé la définition de COSMIC-FFP pour des spécifications écrites avec la notation de Rational Rose RealTime (RRRT). De plus, nous avons développé un outil, µROSE, qui permet de mesurer automatiquement COSMIC-FFP pour des spécifications RRRT. Nos définitions formelles permettent de lever plusieurs ambiguïtés et de rendre ces mesures objectives, ce qui permettra d'éliminer la variance dans le processus de mesure. La formalisation permet également l'automatisation de la mesure.
Remerciements

Je tiens à exprimer tout d'abord mes plus sincères remerciements à mes professeurs Marc Frappier et Richard St-Denis, respectivement directeur et co-directeur, qui m'ont appuyé par leurs conseils, leurs compétences, leurs soutiens financiers, leur grande patience et la confiance qu'ils m'ont accordée durant toute la période des mes études doctorales. Ceci a grandement facilité la réalisation du présent ouvrage. Qu'ils trouvent ici l'expression de ma gratitude pour l'intérêt qu'ils portent à ce travail.

Je voudrais aussi exprimer mes remerciements aux professeurs, Francis Bordeleau, Sylvain Giroux et Henri Habrias, et qui m'ont fait l'honneur de participer au jury de cette thèse.

Merci également à mon collègue Fouad Koukane pour sa patience et sa précieuse collaboration dans ce qui constitue la conception de l'outil \textit{\muROSE}. J'aimerais aussi remercier mes collègues Stéphane Beaudoin, Ning-Sun Cao, Ly Thanh-Le, Huu-Baoan Trinh et Hai-Pho Truong pour leur aide dans ce qui concerne le codage de l'outil \textit{\muROSE}.

Qu'il me soit également permis d'adresser mes remerciements à tout le personnel du Département de mathématiques et d'informatique de l'Université de Sherbrooke pour son affabilité et sa sympathie.

Enfin, à mes parents et amis dont le soutien fut indéfectible; merci du fond du coeur, ce travail vous est dédié.
Table des matières

Sommaire ii

Remerciements iii

Table des matières iv

Liste des tableaux ix

Liste des figures x

Introduction 1

1 Formalisation de la mesure d'IFPUG-FP dans le contexte du langage B 9

1.1 Overview of Functions Points ........................................ 14
1.2 Overview of the B Method .......................................... 15
1.2.1 Example of a B Specification .................................. 16
1.3 Formal Definition Summary ....................................... 18
1.4 Classifying Components ........................................ 19
1.5 Weighing Components ........................................... 23
1.6 The UFP Counting Algorithm .................................... 24
1.7 Case study ......................................................... 25
2 Formalisation de la méthode COSMIC-FFP selon le langage ROOM

2.1. Background ........................................ 36
   2.1.1. COSMIC Full Functions Points (COSMIC-FFP) .... 36
   2.1.2. The ROOM Language ............................ 37
2.2. Definition of COSMIC-FFP Elementary Concepts ......... 38
   2.2.1. Boundary ...................................... 38
   2.2.2. Layer ......................................... 38
   2.2.3. Functional Process ............................ 39
2.3. Classifying Components ................................ 43
   2.3.1. Data Group .................................... 43
   2.3.2. Data Attribute ................................. 46
   2.3.3. Data Movements ................................. 47
2.4. Conclusion ......................................... 49
References ............................................. 51

3 Automatisation de la mesure de COSMIC-FFP pour Rational Rose Real-Time

3.1 A Brief Overview of COSMIC-FFP ...................... 52
3.2 An Overview RRRT .................................. 57
3.3 A Tour of $\mu_cROSE$ ................................ 58
   3.3.1 The Architecture ............................... 61
   3.3.2 The Interface .................................. 61
3.4 The Class Models of $\mu_cROSE$ ....................... 63
   3.4.1 The RRRT Class Model ......................... 63
   3.4.2 The COSMIC-FFP Class Model ................. 65
3.4.3 The Mapping between COSMIC-FFP and RRRT Class Models . 69
3.5 Validation and Verification of $\mu_cROSE$ .......................... 72
3.6 Conclusion ...................................................... 74

Conclusion ...................................................... 79

Bibliographie .................................................... 84

Glossaire (Glossary) .............................................. 88

A Contexte (Background) .......................................... 90
A.1 Functional Size Measurement .................................... 90
A.1.1 Usage of Size Measurement in Software Engineering .... 90
A.1.2 Measuring Functional Size Versus Lines of Code ............ 92
A.2 Function Points for Functional Size Measurement ............ 92
A.2.1 The Characteristics of Function Point Techniques ............ 93
A.2.2 Function Point Measurement Process ......................... 93
A.3 Problems with IFPUG-FP and COSMIC-FFP ................. 95
A.4 Contributions ................................................. 96
A.4.1 General Contributions ....................................... 96
A.4.2 Why B and RRRT? ........................................... 97
A.5 Related Work and Comparison .................................. 98
A.5.1 IFPUG-FP and COSMIC-FFP Measuring Tools ............... 98
A.5.2 Mapping IFPUG-FP and COSMIC-FFP to OO Methods ....... 99
A.6 Methodology .................................................. 100
A.7 Evaluation of Specification Languages for Formalizing IFPUG-FP and COSMIC-FFP .......................... 101
B Des définitions formelles de la mesure d’IFPUG-FP pour le langage B
(Formal Rules for Measuring IFPUG-FP from B Specifications) 104

B.1 Definition of Elementary Concepts ........................................... 105
   B.1.1 Elementary Process .................................................. 105
   B.1.2 Logical Data Group and Data Element Type ...................... 105
   B.1.3 Record Element Types (RET) ...................................... 107

B.2 Classifying Components ...................................................... 108
   B.2.1 Internal Logical Files (ILF) ....................................... 108
   B.2.2 External Interface Files (EIF) .................................... 110
   B.2.3 External Inputs (EI) ............................................... 111
   B.2.4 External Outputs (EO) ............................................. 112
   B.2.5 External Inquiries (EQ) ........................................... 114
   B.2.6 Verifying the Completeness of FP ................................. 115

B.3 Weighing Components .......................................................... 117
B.4 Equivalent Operations .......................................................... 120

C Cas d'étude: la mesure d’IFPUG-FP à partir d'une spécification B
(Measurement of IFPUG-FP for a B Specification using Formal Rules) 122

C.1 List of transactions for the Library System ................................. 122
C.2 The B Specification of the Library System ................................. 124
C.3 Detailed Calculations of IFPUG’s FP Counts ............................... 137

D Le document des spécifications de l'outil $\mu_cROSE$ (Specification Doc-
ument of $\mu_cROSE$) ............................................................. 141

D.1 Purpose of $\mu_cROSE$ ....................................................... 141
D.2 Formal Specification of $\mu_cROSE$ ..................................... 142
D.3 Architecture ................................................................. 159
D.4 C++ Syntax Analyzer of $\mu_cROSE$ .................................... 160
D.4.1 Algorithm .................................................. 160
D.4.2 C++ Operators .......................................... 163
D.5 Class Models of \( \mu_c\text{ROSE} \) ............................. 164

E \( \mu_c\text{ROSE} \) : cas d'études (Case Studies of \( \mu_c\text{ROSE} \)) ......................................................... 166
E.1 Experimental Evaluation of \( \mu_c\text{ROSE} \) ................. 166
E.2 Test Cases .................................................... 166
E.3 Results of Some Case Studies ................................ 168
E.4 A Simple Fast and Serious Measurement ................... 169
   E.4.1 Measurement of data group .......................... 170
   E.4.2 Measurement of functional process .................. 173
<table>
<thead>
<tr>
<th>No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The interpretation of FP concepts within the B notation</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>Correspondence between COSMIC-FFP and RRRT concepts</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>Some RRRT models used during validation and verification with their FS</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>Processes classification according to IFPUG CPM and the formal rules</td>
<td>116</td>
</tr>
<tr>
<td>5</td>
<td>Complexity weights of an ILF according to IFPUG rules</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>Complexity weights of an ILF according to the formal rules</td>
<td>120</td>
</tr>
<tr>
<td>7</td>
<td>Unadjusted function points of the library system using UML</td>
<td>138</td>
</tr>
<tr>
<td>8</td>
<td>The components of the library system using our formal rules</td>
<td>139</td>
</tr>
<tr>
<td>9</td>
<td>The components of the library system using IFPUG rules</td>
<td>140</td>
</tr>
<tr>
<td>10</td>
<td>Cfsu of RRRT model using $\mu_{ROSE}$</td>
<td>168</td>
</tr>
</tbody>
</table>
Liste des figures

1. The structure diagram of the factory system and the behavior diagram of actor Controller ........................................ 41
2. The structure diagram of the factory system and state machine diagram of capsule Controller ........................................ 60
3. The functional architecture of $\mu_cROSE$ .................................................. 62
4. The RRRT model tab box ................................................................. 63
5. Summary by process of COSMIC-FFP measurement ........................................ 64
6. Details of COSMIC-FFP measurement .................................................... 64
7. Summary by type of COSMIC-FFP measurement .......................................... 65
8. Class model for RRRT models .......................................................... 66
9. Class model for COSMIC-FFP software models ......................................... 68
10. $\mu_cROSE$ main modules ............................................................. 160
Introduction

Les disciplines traditionnelles d’ingénierie sont constamment concernées par l’utilisation de mesures pour contrôler la production dans le but d’atteindre des objectifs avec un coût acceptable. Ainsi, il est essentiel pour une organisation de bien préparer et de bien contrôler les plans d’exécution du processus de production dans le but de réaliser ses objectifs. En informatique, le gestionnaire a besoin d’établir des plans relatifs à l’exécution du processus de développement qui doivent contenir des descriptions sur les activités associées au développement. C’est pourquoi des normes de ISO (International Standard Organization) décrivent un ensemble de mesures externes et internes qui sont nécessaires afin que le gestionnaire puisse, notamment, inclure dans son plan de développement des éléments essentiels comme :

  - un calendrier de réalisation des activités dans le respect des délais ;
  - une estimation de l’effort requis ;
  - une estimation des ressources adéquates nécessaires pour l’exécution des activités ;
  - un calcul des coûts associés au développement.

Parmi ces mesures, on trouve la mesure de la taille fonctionnelle du logiciel qui est une des mesures internes les plus importantes. Elle a été définie comme étant la valeur numérique calculée par la quantification des besoins fonctionnels du logiciel.

L’absence des éléments fondamentaux d’ingénierie dans un projet de développement de logiciel peut être partiellement attribuée à la nature variante du logiciel. Cette caractéristique du logiciel le rend aussi difficile à mesurer. Malgré cette difficulté, plusieurs
techniques ont été proposées pour mesurer la taille fonctionnelle d'un logiciel à partir de sa spécification des besoins.

La méthode la plus dominante en industrie, qui permet de mesurer la taille fonctionnelle d'un système durant les premières phases du cycle de vie, est la méthode des points de fonction originalement proposée par Albrecht. Elle a été restructurée par IFPUG (International Function Points User Group). Notons que nous appellerons cette mesure IFPUG-FP dans ce document. Récemment, la méthode des points de fonction a été étendue par le groupe COSMIC (Common Software Measurement International Consortium) pour la mesure des systèmes temps-réels. Cette mesure est appelée COSMIC-FFP (COSMIC Full Function Points). Notons que cette dernière peut aussi être utilisée pour mesurer d'autres types de logiciel (par exemple des systèmes d'information).

**Problème à résoudre**

Comme pour de nombreux autres domaines, lorsqu'il s'agit de mesurer la taille d'un produit, il est indispensable de pouvoir appliquer les règles de mesure d'une manière objective afin d'obtenir des résultats reproductibles. Par ailleurs, l'objectivité et la reproductibilité sont parmi les caractéristiques qui ont été considérées dans le norme ISO/IEC DTR 9126-4 et qui doivent être satisfaites lors du mesurage.

Dans cette thèse, nous considérons les problèmes de la subjectivité des règles de mesure et de la non reproductibilité de mesure lors de l'utilisation d'IFPUG-FP ou de COSMIC-FFP, surtout par des non-experts. En effet, les méthodes de points de fonction ont leurs propres sources d'ambiguïté. Elles ont été définies en langage naturel et elles sont généralement applicables à n'importe quel langage de spécification. L'inconvénient de cette applicabilité générale est que certains concepts adoptés par ces méthodes n'ont pas une représentation directe dans le contexte des langages de spécification. Cette lacune rend les règles de mesure plus ambiguës, subjectives et dépendantes de l'opinion du mesureur. Pour cela, il est indispensable que l'application de ces techniques soit faite par
des experts (qui ne sont pas toujours disponibles).

D’autre part, il existe peu d’outils qui permettent de mesurer directement les points de fonction d’IFPUG-FP (aucun pour COSMIC-FFP) à partir des besoins fonctionnels du logiciel. Certains concepts de ces deux méthodes n'ont pas une représentation directe en termes d'attributs observables des spécifications, ce qui les rend difficiles à automatiser.

**Approche proposée et contributions**

Pour faire face à ces problèmes, il est important d'éliminer la subjectivité des règles de mesure d’IFPUG-FP et de COSMIC-FFP et de rendre ces règles indépendantes de l'interprétation du mesuré afin d'obtenir des résultats reproductibles, notamment lors de l'application d’IFPUG-FP ou de COSMIC-FFP par des non-experts.

Afin d'atteindre cet objectif, nous avons défini des règles de mesure en utilisant des notations mathématiques pour calculer :

- les points de fonction d’IFPUG-FP à partir d’une spécification écrite en langage B et

- les points de fonction de COSMIC-FFP à partir d’une spécification produite par l'outil Rational Rose Real-Time (RRRT).

Grâce à ces définitions formelles, nous avons réussi à lever plusieurs ambiguïtés de la définition des points de fonction d’IFPUG-FP et de COSMIC-FFP et de rendre automatisable leur application. Ceci permet d’appliquer les règles de mesure d’une manière objective et d’obtenir des résultats reproductibles, ce qui est conforme aux caractéristiques recommandées par ISO, notamment l’objectivité et la reproductibilité.

De plus, nous avons utilisé nos définitions formelles de COSMIC-FFP pour développer l’outil \( \mu_\text{ROSE} \). \( \mu_\text{ROSE} \) est le premier outil qui automatise la mesure de COSMIC-FFP pour une spécification produite par l’outil RRRT. Un des avantages principaux de \( \mu_\text{ROSE} \) est que les résultats de mesure peuvent être obtenus d’une manière simple et
rapide et avec un minimum d’assistance humaine, ce qui réduit le coût et le délai de la mesure. Par ailleurs, plusieurs compagnies ont introduit ou commencent à introduire la technologie Rational Rose Real-Time dans leurs processus de développement et qui sont concernées par l’utilisation de mesures. Ainsi, il sera utile de mettre à la disposition de ces compagnies un outil comme $\mu_{ROSE}$.

**Portée de la thèse**

Il faut bien noter que l’objectif de notre travail est limité à la formalisation et à l’automatisation des règles de mesure d’IFPUG-FP et de COSMIC-FFP. Il ne porte surtout pas sur la validation de l’applicabilité des points de fonction aux systèmes temps-réels ou autre. Pour valider une telle applicabilité, il faudrait mesurer les points de fonction pour des applications déjà développées. Ensuite, nous ferions des estimations basées sur les points de fonction calculés aux fins de comparaison entre les valeurs estimées et les valeurs réelles. Dépendamment des résultats de ces comparaisons, on pourrait décider si les points de fonction sont applicables aux systèmes temps-réels ou non. Des telles expérimentations exigent des ressources qui dépassent le cadre de notre projet de recherche. Vous pouvez visiter le site de COSMIC-FFP [1] pour consulter des expérimentations qui ont été faites en mesurant COSMIC-FFP pour des systèmes temps-réels.

**Pourquoi B et RRRT**

Notre choix du langage B et de RRRT pour formaliser les définitions d’IFPUG-FP et de COSMIC-FFP est basé sur les faits suivants :

- l’adéquation entre certains concepts du langage B et IFPUG-FP d’une part, et entre d’autres concepts de RRRT et de COSMIC-FFP d’autre part ;

- l’utilisation industrielle de RRRT et du langage B.

Pour avoir plus de détails sur ce sujet, nous référons le lecteur à l’annexe A.
Discussion

Le paradigme approprié pour les points de fonction

Afin de mesurer les points de fonction, le document de spécification doit comporter des descriptions qui se situent à différents niveaux d'abstraction. Ce document doit fournir des descriptions sur toutes les fonctionnalités du point de vue de l'utilisateur. D'autre part, des informations doivent également être présentées concernant le type de traitement porté sur chaque donnée élémentaire et sur le regroupement de ces données, lors de l'exécution d'une fonctionnalité. Dans ce sens, le paradigme qui nous permet d'avoir ces informations est le paradigme le plus approprié pour les points de fonction. On remarque bien que les résultats de mesure ne dépendent pas seulement de la bonne ou mauvaise application des règles de mesure ; il faut que le document de spécification couvre l'ensemble des informations pertinentes pour la mesure des points de fonction. Pour cela, il est essentiel que l'analyste prenne en considération toutes ces informations lors de la modélisation.

La complexité du logiciel et les points de fonction

En pratique, l'objectif ultime est de pouvoir mesurer la taille fonctionnelle du logiciel qui reflète adéquatement l'effort, la productivité, etc. Il existe deux tendances sur la validité et l'applicabilité des points de fonction. Certains se plaignent que les méthodes des points de fonction mettent lourdement l'emphasis sur les fonctionnalités du logiciel du point de vue de l'utilisateur et ne couvrent véritablement pas la complexité du système. D'autres affirment l'utilité et l'applicabilité des points de fonction en industrie. En réalité, les méthodes des points de fonction prennent en considération les deux aspects : la fonctionnalité et la complexité. En effet, IFPUG-FP assigne un poids de complexité aux composants de données et aux composants fonctionnels. De plus, la complexité est représentée par les 14 facteurs d'ajustement. Cependant, la vraie question qui demeure sans réponse est : est-ce-que le poids assigné et ces facteurs reflètent correctement la complexité du
système? Pour avoir une réponse à cette question, des expérimentations devront être établies en terrain industriel très actif et particulièrement sensible de recherche. Une chose est certaine : sans avoir des données quantitatives qui montrent l'applicabilité des points de fonction par domaine d'application, cette question et d'autres resteront sans réponses. Pour cette raison notamment, nous avons entrepris le présent projet de recherche dans le but d'améliorer la reproductibilité des résultats de mesure, ce qui est indispensable pour mener des expérimentations.

La théorie de la mesure versus la mesure des points de fonction

Selon Kitchenham, Pfleeger et Fenton [15], la mesure des points de fonction d'IFPUG-FP est inconsistante par rapport à la théorie de la mesure. Ceci est dû à la combinaison de deux échelles (ordinale et rapport) utilisées par les points de fonction d'IFPUG. La première échelle est utilisée pour assigner un poids à un composant et la deuxième est utilisée lors du comptage. Il faut bien noter que ce problème a été bien résolu par la méthode COSMIC-FFP puisqu'elle utilise seulement l'échelle rapport.

Malgré tous les problèmes rapportés (la limitation du domaine d'application, la dépendance technologique, etc.) dans la littérature, la mesure des points de fonction a déjà montré son utilité en industrie par rapport à la mesure des lignes de code ; surtout quand elle est appliquée avec prudence et avec une bonne compréhension de ses limitations et de l'application des règles de mesure. D'où l'importance de la formalisation des règles de mesure afin qu'elles soient adéquatement appliquées.

Application du \( \mu \text{-ROSE} \) à lui-même

Rappelons que le paradigme RRRT a été spécifiquement conçu pour modéliser des systèmes temps-réels. Pour cela, il n'était pas approprié de modéliser un système d'information, comme \( \mu \text{-ROSE} \), en utilisant la notation RRRT. Pour cette raison, nous avons modélisé \( \mu \text{-ROSE} \) en utilisant les notations B et UML plutôt que d'utiliser la notation
RRRT. Ainsi, on ne peut pas appliquer l'outil $\mu_cROSE$ à lui-même. Cependant, il est toujours possible de re-modéliser $\mu_cROSE$ en utilisant la notation RRRT. Dans ce sens, il serait possible d'appliquer $\mu_cROSE$ à son propre modèle RRRT.

Structure de la thèse

Cette thèse est structurée comme suit. Les chapitres 1 et 2 représentent deux de nos articles qui ont été publiés et introduisent les règles de mesure d'IFPUG-FP et de COSMIC-FFP pour le langage B et la notation de RRRT, respectivement. Le chapitre 3 présente un de nos articles qui a été soumis pour publication. Dans ce chapitre, nous présentons les composants, l'interface et l'architecture de l'outil $\mu_cROSE$. Dans le dernier chapitre, des conclusions et des travaux futurs sont présentés. Finalement, les problèmes d'IFPUG-FP et de COSMIC-FFP, les définitions proposées, les spécifications de l'outil $\mu_cROSE$ et les résultats de nos experimentation sont décrits dans les annexes A, B, C, D et E comme suit :

- l'annexe A : donne un aperçu sur le rôle de la taille fonctionnelle du logiciel dans un projet de développement, présente le problème et la solution proposée et décrit la méthodologie pour la réalisation de cette solution ;
- l'annexe B : introduit les définitions formelles de la mesure d'IFPUG-FP pour le langage B ;
- l'annexe C : présente une application des définitions formelles, présentées dans l'annexe B, sur une spécification écrite en langage B ;
- l'annexe D : fournit les spécifications de l'outil $\mu_cROSE$ ;
- l'annexe E : décrit les résultats de plusieurs cas d'études analysés par $\mu_cROSE$.

Publications

Voici la liste des articles qui ont été publiés ou soumis pour publication.
Conferences


Journal

Chapitre 1

Formalisation de la mesure d’IFPUG-FP dans le contexte du langage B

Dans ce chapitre, nous présentons un de nos articles qui a été publié dans les actes de la conférence 4th International Conference on Formal Engineering Methods (ICFEM 2002). Dans cet article, nous proposons une définition formelle de la mesure d’IFPUG-FP pour des spécifications écrites en langage B. Ces définitions contribuent à :

- réduire la variance entre les mesures d’IFPUG-FP causée par la subjectivité des définitions informelles d’IFPUG-FP ;
- fournir des instructions claires et précises sur l’application des règles de mesure d’IFPUG-FP pour des spécifications écrites en B ;
- améliorer la compréhension de l’application de la mesure d’IFPUG-FP en général ;
- identifier des lacunes dans les règles de mesure d’IFPUG-FP et proposer des modifications afin d’assurer la complétude de nos définitions.
Finalement, les définitions proposées peuvent être utilisées comme une base afin de développer un outil supportant la mesure d'IFPUG-FP pour des spécifications écrites en B.
Counting Function Points From B Specifications

H. Diab, M. Frappier, R. St-Denis
Département de mathématiques et d'informatique, Université de
Sherbrooke
Sherbrooke, Québec, Canada, J1K 2R1
Phone: (819) 821 8000 ext. 2096, Fax: (819) 821 8200
e-mail: {Hassan.Diab, Marc.Frappier, Richard.St-Denis}@dmi.usherbro.ca
Key words: software engineering, software size measurement, formal specification.

Abstract

This paper proposes a formalization of the IFPUG Function Point (FP) definition for automated measurement of B specifications. This formal definition allows to: i) reduce the variance in FP counts due to rater interpretation of the IFPUG FP informal definition; ii) provide a better understanding of how the IFPUG FP definition should be applied; iii) automate the FP counts for B specifications, which can reduce measurement costs; and iv) identify specific holes in the IFPUG FP definition. We propose modifications to ensure completeness.

Introduction

Functional software size is an essential component for managing and controlling a software development project. FP analysis is one of the prominent industrial methods to measure functional size. It does not depend on a particular functional specification notation or on a system implementation. Albrecht proposed the first definition of this
measure [3]. Its users are now structured as a user group, the International Function Point Users Group (IFPUG), which provides a Counting Practice Manual (CPM) [24]. Their definition of FP is the following: "A function point is a synthetic metric that is comprised of the weighted totals of the inputs, outputs, inquiries, logical files or user data groups, and interfaces belonging to an application."

The FP measure may be used to: estimate development effort, by relating it with FP; evaluate software quality, by computing defect density; evaluate software productivity; manage outsourcing contracts, by determining annual maintenance cost per FP; and to compare systems, specified in different languages, in terms of productivity, quality, and maintenance costs.

IFPUG FP have several weaknesses. First, the counting rules are given in plain natural language and are subject to interpretation by measurers. This problem introduces some variances on FP count depending on the rater. Studies noted a difference varying from 11% to 30% between several counts of the same specification [18, 33, 34]. Furthermore, variations from 400% to 2000% are observed between the FP count of an initial specification and FP count of the resulting system [20]. Moreover, FP calculation is mostly a manual process which is quite expensive. Typically, an expert can count 57 unadjusted function points per hour. For some companies with a large software portfolio (e.g., 325 000 FP), it would take around three person-years to precisely measure their portfolio.

Since the root of these weaknesses is the lack of a precise definition for the IFPUG FP method, our goal in this paper is to provide a clear and unambiguous definition for computing the unadjusted function points of IFPUG FP. The IFPUG CPM provides rules for counting the unadjusted function points [24]. It is these rules that must be formalized. In [17], the adequacy of several formal specification languages for the formalization of IFPUG FP has been evaluated. It was found that the B notation [2] provides some details which are essential to automatically compute the number of FP. The B method is a formal, industrial software development method for specification, design, and
implementation of software products.

Since software requirement specifications depend on the language in use, the drawback of the general applicability of IFPUG FP is the lack of a direct representation of some IFPUG concepts within any specification language, the B notation in particular. It is unlikely that different measurers will consistently use the same mapping between B and IFPUG FP. For instance, Data Element Types (DET) and Record Element Types (RET) in IFPUG FP do not directly map to B concepts. The weights of function and data components in IFPUG FP are computed based on the number of DETs, RETs, and files. So any misuse of the counting rules for identifying these elements may negatively affect the measurement results. Therefore, mapping rules between B and IFPUG FP are required.

When using B in large scale projects, it is important to connect B to software metrics for an effective software engineering management. Our formal definition allows users of the B method to exploit the advantages due to the use of FP. For instance, an interesting research would be to take a number of B case studies, count the number of FP of each, and check what the implementation turns out to cost. The variation induced by implementation decisions could be an interesting factor and may mean that we should look at classes of problems.

In addition, our formal definition allows the automation of unadjusted FP counting for B specifications. It removes measurement variance and ensures perfect repeatability, because the formal definition is objective and independent of the measurer interpretation. Furthermore, the formal definition of IFPUG FP has allowed us to verify the completeness of the IFPUG FP counting rules. Interestingly, we have found that some components of a specification do not fall into any category in the IFPUG CPM. Hence, the IFPUG CPM definitions are incomplete. We have proposed modifications to ensure completeness.

This paper is structured as follows. Sections 2 and 3 provide overviews of IFPUG FP and the B method. Sections 4 and 5 provide the formal definition of IFPUG FP concepts.
within the B notation. Section B.3 briefly describes how components are weighed. Section 1.6 introduces the algorithm of the unadjusted FP counting for B specifications. Section 1.7 introduces a case study. Finally, Section 1.8 concludes with an appraisal of this work.

1.1 Overview of Functions Points

In this section, we introduce the basic concepts of IFPUG FP as defined in the IFPUG CPM version 4.1. The following equation describes how function points, denoted by $FP$, are counted: $FP = UFP \times VAF$.

The variable $UFP$ denotes the number of unadjusted FP and the variable $VAF$ denotes the value adjustment factor. The variable $UFP$ is determined using two main categories of components: data components (e.g., files, database relations) and function components or elementary processes (updates, inquiries, reports). Data components are further divided in two categories of file: external interface files (EIF) and internal logical files (ILF). Note that in the rest of this paper we refer to a file as a data group. Elementary processes are divided into three categories: external inputs (EI), external outputs (EO), and external inquiries (EQ). To compute $UFP$, each component $c$ is assigned a weight $w_c$ according to its category (EIF, ILF, EI, EO, or EQ) and other parameters: for data components, these parameters are the number of data element types (DET) and the number of record element types (RET); for function components, they are the number of DET and the number of EIF and ILF referenced. The UFP is simply the sum of the component weights: $UFP = \sum_c w_c$.

The following equation describes the calculation of $VAF$ which is determined by evaluating 14 factors, $f_i$, on a scale of 0 to 5: $VAF = 0.65 + 0.01 \times \sum_{i=1}^{14} f_i$. There are also rules to evaluate each factor, but they are not easily formalizable. Indeed, they directly depend on human judgment of the technical elements that may have an impact on the system complexity, data communications, application type, and performance objectives.
for instance. It should be noted that the effort required to evaluate $\text{VAF}$ only represents a small percentage of the total effort for FP count. Furthermore, these factors have been removed from the FP techniques that have been recently proposed, COSMIC Full Function Points [1] in particular. Hence, there is little interest for their formalization.

1.2 Overview of the B Method

The B method is founded on first order logic, set theory, and the theory of refinement. It is representative of the family of methods known as model oriented which represent a software by data, characterized by their invariant properties, and by services which handle these data [2]. The B method uses three notations as a specification language for software development: the mathematical notation, the generalized substitution notation, and the abstract machine notation. Set theory is the primarily mathematical notation used in B. It allows to specify the data and the software properties. Unlike classical set theory, the set theory of B is typed: a set consists of elements having all the same fundamental structure. The generalized substitution notation allows to describe the services of a software. In other words, it allows to describe the actions that a software can carry out to fulfill its functions.

The abstract machine notation allows to define sets, constants, variables, and operations which abstractly represent the data and services of a software. By doing this, it defines the operational interface of a software because it provides the mode and use conditions of its components. The sets and constants of an abstract machine represent immutable data. There are two kinds of set: abstract set and enumerated set. The definition of abstract set elements is deferred to the implementation. An enumerated set is a set defined by a complete list of its elements. The constants are characterized by a predicate of the mathematical notation. The variables of an abstract machine represent the modifiable data. As for the constants, variables are defined by a predicate called an invariant, which defines the properties that their actual values must always satisfy.
Finally, the B method includes various kinds of access relations between machines. The main relations are: **includes**, **uses**, **sees**, **imports**, and **extends**. Each relation imposes constraints on the access mode from the referencing machine to the referenced machine components.

### 1.2.1 Example of a B Specification

In this section, we use the B specification language to model the transactions of a simplified version of a library system. Only three transactions are modeled: **Acquire**, **ListBookAuthor**, and **FlagBadMember**. The last transaction is given in Section 5. Note that we will use this example in the rest of this paper to illustrate the application of our definition.

The machine **Library** describes the state variables, their invariant, and operations. The name of the machine, **Library**, is given in the first clause. **Library** has one input parameter **MaxLoanDuration**. The **SETS** clause defines the abstract sets of **Library**. The state of **Library** is defined by ten variables given in the **VARIABLES** clause. Each variable has a type given in the **INARIANT** clause. The **INITIALIZATION** clause defines the initial state of the machine. A value is assigned to each variable by using an *elementary substitution*. Machine **Library** has several operations defined in the **OPERATIONS** clause. The main substitution of those operations is a *precondition substitution* of the form **PRE p THEN S END**, where **p** is a condition and **S** a set of elementary substitutions. This construct means that the elementary substitution (corresponding to **S**) is only well-defined when **p** holds.

**MACHINE**  \textit{Library} (\textit{MaxLoanDuration})

**CONSTRAINTS**

\textit{MaxLoanDuration} \in \mathbb{N}

**SETS**

\textit{BOOK}; \textit{TITLE}; \textit{AUTHOR}; \textit{DATE}; \textit{MEMBER};

16
LOAN_STATUS = \{ InLibrary, Loaned \};
MEM_STATUS = \{ Bad, Good \}

VARIABLES
member, book, title, author; acquisitionDate, borrower,
lastLoanDate, loanStatus, loanCount, memStatus,

INVARIANT
member \subseteq MEMBER \land book \subseteq BOOK \land title \in book \rightarrow TITLE \land
author \in book \rightarrow AUTHOR \land acquisitionDate \in book \rightarrow DATE \land
borrower \in book \rightarrow member \land lastLoanDate \in book \rightarrow DATE \land
loanStatus \in book \rightarrow LOAN_STATUS \land loanCount \in book \rightarrow N \land
memStatus \in member \rightarrow MEM_STATUS

INITIALIZATION
member:= \emptyset \land book:= \emptyset \land title:= \emptyset \land author:= \emptyset \land
acquisitionDate:= \emptyset \land borrower:= \emptyset \land lastLoanDate:= \emptyset \land
loanStatus:= \emptyset \land memStatus:= \emptyset \land loanCount:= 0

OPERATIONS
Acquire(bbookid,btitle,bauthor,bdate) \Rightarrow

PRE
bbookid \in BOOK - book \land book \neq BOOK \land btitle \in TITLE \land
bauthor \in AUTHOR \land bdate \in DATE

THEN
book:= book \cup \{ bbookid \} \land title(bbookid):= btitle \land
author(bbookid):= bauthor \land acquisitionDate(bbookid):= bdate \land
loanStatus(bbookid):= InLibrary \land loanCount(bbookid):= 0

END;

END;

authorlist \leftarrow ListBookAuthor(bauthor) \Rightarrow

PRE
\( \text{bauthor} \in \text{AUTHOR} \)

THEN

ANY \(\text{bookseq} \) WHERE

\(\text{bookseq} \in \text{seq}(\text{TITLE} \times \mathbb{N}) \land\)

\(\text{ran}(\text{bookseq}) = \{ tt,nn \mid tt \in \text{TITLE} \land nn \in \mathbb{N} \land\)

\(\exists \text{bbookid} . (\text{bbookid} \in \text{author}^{-1}[\{\text{bauthor}\}] \land\)

\(tt = \text{title}(\text{bbookid}) \land nn = \text{loanCount}(\text{bbookid})\} \land\)

\(\text{sorted}(\text{bookseq},\text{prj1}(\text{TITLE},\mathbb{N}))\)

THEN

\(\text{authorlist} := ((\text{bookseq} \mapsto \text{card}(\text{dom}(\text{bookseq}))))\)

END

END

1.3 Formal Definition Summary

The definitions provided in this paper are based on a syntactic analysis of B specifications. Moreover, we assume that the machines have been flattened. In other words, we consider that the access relations have been replaced by virtual machines. A virtual machine is the machine resulting from the use of an access relation between two machines. For instance, assume that a machine \(M_1\) includes a machine \(M_2\). We consider the resulting machine \(M'_1\) contains \(M_1\) and \(M_2\). The flattening of a specification changes its structure but it does not change its behavior from the user viewpoint. Therefore, no bias is introduced in the counting when using the flattened specification, since the functional measure is produced from the user viewpoint [24].

Table 1 presents IFPUG FP concepts as defined in the IFPUG CPM version 4.1. The first column introduces each concept with a name and a brief description. The second column presents our interpretation of each concept within the B notation.
### Tab. 1 - The interpretation of FP concepts within the B notation

<table>
<thead>
<tr>
<th>IFPUG concepts</th>
<th>CPM</th>
<th>B notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary</td>
<td>set of operations</td>
<td></td>
</tr>
<tr>
<td>elementary process</td>
<td>operation</td>
<td></td>
</tr>
<tr>
<td>data group (file)</td>
<td>subset of an abstract set</td>
<td></td>
</tr>
<tr>
<td>Data Element Type (DET)</td>
<td>variable S or a function variable f from S to T</td>
<td></td>
</tr>
<tr>
<td>Record Element Type (RET)</td>
<td>DETs are partitioned in two RETs: total and partial function variables</td>
<td></td>
</tr>
</tbody>
</table>

For instance, the *application boundary* in the IFPUG CPM corresponds to a set of operations in B specifications, because, given a set of operations, it possible to derive the set of data components which are modified or referenced by these operations. Therefore, the input of the FP counting process consists of the text of B specifications and the set of operation names to count. We denote the application boundary by $F$.

### 1.4 Classifying Components

In this section, we provide formal rules of the IFPUG FP definition allowing to identify the IFPUG FP component types for B specifications. Our rules are provided using conventional mathematical notations.

**Data Group and Data Element Types (DET).**

Typical examples of a data group in IFPUG FP are a group of logically related data or a relation in a relational database. It corresponds to a variable $S$ which is a subset of an abstract set $AS$ (i.e., $S \subseteq AS$) and the domain of a variable $f$ which is a partial function to some other set $T$.

Let $AS$ be the set of all abstract sets defined in a specification $A$. Let $\mathcal{V}$ be the set of all variables defined in $A$. Let $\mathcal{FV}$ be the set of all variables defined as functions in $A$. 

19
We denote by $DG$ the set of data groups in specification $A$. It is defined as follows.

$$DG \triangleq \{ S \mid S \in \mathcal{V} \land \exists AS, f, T : AS \in AS \land S \subseteq AS \land f \in S \rightarrow T \}$$  (1)

A DET in IFPUG FP is an attribute of a data group, defined as the smallest information that has a meaning. It corresponds to either a variable $S$, which is a subset of an abstract set $AS$ (i.e., $S \subseteq AS$), or a variable $f$, which is a function from $S$ to some other set $T$. We denote by $DET(S)$ the set of DETs of a data group $S \in DG$. It is defined as follows.

$$DET(S) \triangleq \{ S \} \cup \{ f \mid f \in \mathcal{FV} \land \exists T : f \in S \rightarrow T \}$$  (2)

Record Element Types (RET).

The DETs of a data group are partitioned into RETs based on the IFPUG FP definition. RETs are classified as either mandatory or optional. “Optional subgroups are those that the user has the option of using one or none of the subgroups during an elementary process that adds or creates an instance of the data. Mandatory subgroups are subgroups where the user must use at least one.”

Obviously, it is possible to classify a DET as either optional or mandatory using the INARIANT clause of a B specification. However, we do not know how to partition the DETs into RETs using only the specification text itself. It seems that this choice must be made by the user (which may be a subjective choice). A compromise is to define by default one RET which contains all the mandatory DETs of a data group and one RET which contains all the optional DETs of the data group.

Formally, let $\mathcal{FV}$ be the set of all machine variables defined as partial functions and $\mathcal{T FV}$ be the set of all machine variables defined as total functions. Let $RET_m(S)$ be the mandatory RET of a data group $S$ and $RET_o(S)$ be the optional RET of a data group $S$. Rules 3 and 4 provide the formal definitions of mandatory and optional RETs.

$$RET_m(S) \triangleq \{ S \} \cup \{ v \mid v \in DET(S) \land v \in \mathcal{T FV} \}$$  (3)
\[ \text{RET}_o(S) \triangleq \{ v \mid v \in \text{DET}(S) \land v \in \mathcal{PV} \} \] (4)

Internal Logical Files (ILF) and External Interface Files (EIF).

A data group may be classified as an ILF or EIF. Rules 5 and 6 allow to classify the type of a data group. We denote by \( \text{ILF} \) and \( \text{EIF} \) the set of internal logical files and set of external interface files of a boundary \( F \), respectively. We define \( \text{ILF} \) (resp. \( \text{EIF} \)) as the set of data groups which are maintained (resp. consulted without modification) by at least one operation of \( F \).

\[ \text{ILF} \triangleq \{ S \mid S \in \text{DG} \land \exists o : o \in F \land \text{maintains}(o, S) \} \] (5)

\[ \text{EIF} \triangleq \{ S \mid S \in \text{DG} \land \exists o : o \in F \land \text{consults}(o, S) \land \neg (\exists o : o \in F \land \text{maintains}(o, S)) \} \] (6)

where \( \text{maintains}(o, S) \) denotes that operation \( o \) maintains data group \( S \) and \( \text{consults}(o, S) \) denotes that operation \( o \) consults data group \( S \) without changes.

An operation \( o \) is said to maintain a data group \( S \) if and only if there exists an elementary substitution \( b \) in \( o \) such that the left-hand side of \( b \), \( \text{LHS}(b) \), is a \( \text{DET} \) of \( S \). Predicate \( \text{maintains}(o, S) \) is defined as follows. Let \( B_o \) be the set of elementary substitutions appearing in the definition of operation \( o \).

\[ \text{maintains}(o, S) \iff \exists b, v : b \in B_o \land v \in \text{DET}(S) \land v = \text{LHS}(b) \] (7)

An operation \( o \) is said to consult a data group \( S \) if and only if \( o \) does not maintain \( S \) and there exists a \( \text{DET} \) \( v \) of \( S \) such that \( v \) occurs in the definition of \( o \). Let \( \text{occurs}(v, o) \) be that \( \text{DET} \) \( v \) appears in operation \( o \). Predicate \( \text{consults}(o, S) \) is defined as follows.

\[ \text{consults}(o, S) \iff \exists v : v \in \text{DET}(S) \land \text{occurs}(v, o) \land \neg \text{maintains}(o, S) \] (8)

By applying rules 1 to 8 to the library example, we conclude that: i) the subsets book and member are data groups classified as ILF; and ii) book, title, author, acquisitionDate, borrower, lastLoanDate, and loanStatus, (resp. member and memberStatus) are DETs of book (resp. member). Finally, book, contains two RETs: \( \text{RET}_o(\text{book}) \) and \( \text{RET}_m(\text{book}) \) and member has one RET: \( \text{RET}_m(\text{member}) \).
External Inputs (EI).

An EI is an elementary process that maintains data or control information. It corresponds to an operation in the B language that has inputs and maintains at least one ILF. Let $EI$ be the set of external inputs of a specification. Let $inParam(o,p)$ denote that $p$ is an input parameter of operation $o$ and let $outParam(o,p)$ denote that $p$ is an output parameter of operation $o$. Set $EI$ is defined as follows.

$$EI \triangleq \{ o \mid o \in F \land (\exists \ p, S : inParam(o,p) \land S \in DG \land \text{maintains}(o,S)) \land \neg(\exists \ p' : outParam(o,p')) \}$$

(9)

External Outputs (EO).

An EO is an elementary process that derives data or control information sent to outside the application boundary and possibly maintains data within the boundary. It corresponds to an operation $o$ within the boundary that: i) has at least one output parameter considered as a derived data; ii) has at least one output parameter and there is no input parameter; or iii) has at least one output parameter and maintains an ILF; Let $derives(o,p)$ denote that operation $o$ has a derived data $p$. Let $EO$ be the set of external outputs of a specification. Set $EO$ is defined as follows. Note that a detailed description of the formalization of the concept derived data is outside the scope of this paper. The reader may consult [11] for a more detailed description of this formalization.

$$EO \triangleq \{ o \mid o \in F \land \exists \ p : outParam(o,p) \land \left( \begin{array}{c} \text{derives}(o,p) \lor \\
\neg(\exists q : inParam(o,q)) \lor \\
(\exists S : S \in ILF \land \text{maintains}(o,S)) \end{array} \right) \}$$

(10)

By applying rules 9 and 10 to the library example, we conclude that Acquire is an EI and ListBookAuthor is an EO.
Completeness of IFPUG FP.

In [11], a truth table, that summarizes how operations in B specification are classified in the IFPUG FP context, has been built. This table indicates the category in which an operation is classified according to our interpretation of the IFPUG CPM definitions and proposes extension to the IFPUG CPM definitions in order to have a more complete classification. Based on this table, we have been able to identify some cases that are not explicitly covered in the IFPUG CPM. Furthermore, an operation, that maintains the internal system state and has no input/output parameters, is not defined or discussed in IFPUG CPM. We believe that it should be counted as an EI. For instance, the operation FlagBadMember sets the status of all the members who are the current borrowers of overdue books to Bad. The operation has no input, but it modifies the system state in a way which can be recognized by the user.

\[
\text{FlagBadMember} \triangleq \\
\text{memStatus} := \text{memStatus} \leftarrow\\n\{ \ bmemberid, \ newStatus \mid bmemberid \in \text{member} \land \\
\quad \newStatus \in \text{MEM\_STATUS} \land \exists \ bbookid . \ ( bbookid \in \text{book} \land \\
\quad \quad \ bmemberid = \text{borrower}(bbookid) \land \newStatus = \text{Bad} \land \\
\quad \quad \quad \text{currentDate} - \text{lastLoanDate}(bbookid) > \text{MaxLoanDuration} \} \\
\]

To count operations of this kind, we propose to modify rule 9 as follows.

\[
EI' \triangleq \{ o \mid o \in F \land \exists \ S : S \in DG \land \text{maintains}(o,S) \land \\
\quad \neg(\exists \ p : \text{outParam}(o,p)) \} \\
\]

(11)

1.5 Weighing Components

The last step of the IFPUG FP count is to compute component weights. The weight (complexity level) of a data group (i.e., an ILF or an EIF) depends on the number of DETs and the number of RETs. We have already defined how to compute DETs and
RETs of a data group in Section 5.1. The weight of a transaction (i.e., an EI, an EO, or an EQ) depends on the number of DETs and DGs referenced by function components. Additional details about the formal rules that identify the DETs and DGs referenced by a transaction are described in [11].

Once the referenced DETs, RETs, and DGs are known, we can compute a complexity level for each component type (ILF, EIF, EI, EO, or EQ) by using the complexity level tables of components as defined in IFPUG CPM.

In Section 5, we have defined the DETs, RETs, referenced DETs, and DGs as finite sets. In order to compute a complexity level for data or function components, we need to compute the number of elements in each finite set defined in the formal rules. To do so, we use the cardinality of a finite set $E$, denoted by $\text{card}(E)$. For instance, we use $\text{card}(\text{DET}(S))$ to calculate the number of DETs of $S$.

1.6 The UFP Counting Algorithm

To compute the number of unadjusted function points (UFP) for a B specification that has been flattened, the only information needed as input (aside from the specification itself), is the list of operations included in the boundary $F$. This list can be provided in either of the following forms. i) A list of machine names: in that case, all operations of these machines are considered to be included in the boundary $F$. ii) A list of machine names and, for each machine name, a list of operation names: in that case, only the operations specifically identified are included in $F$.

The counting algorithm is composed of six steps, as described below. In each step, one or several formal rules, given so far, are used to compute the FP components. For instance, rule 1 is used in step 1 to compute the set of data groups $DG$. Recall that the formal rules are based on a syntactic analysis of a B specification.

1. Compute set $DG$ for all the machines identified in the input
2. Compute sets \( RET \) and \( DET \) for each data group in \( DG \)
3. Compute sets \( EI, EO, \) and \( EQ \)
4. Compute sets \( ILF \) and \( EIF \)
5. Compute the weight of each element of \( ILF, EIF, EI, EO, \) and \( EQ \)
6. Compute the sum of the weights to obtain \( UFP \)

### 1.7 Case study

In this section, we provide an illustration of the suitability and objectivity of our definitions by applying them to the full version of the library system, presented in [11] and comparing the results with three other counts. The first two counts were obtained with the IFPUG FP definitions, using two different user viewpoints. The third count is derived using a definition of IFPUG FP for use case diagrams from the UML notation. The details of this last approach are provided in [35]. The informal description of the library system is provided in [35].

For the library system, we see at least two viewpoints. In the first viewpoint, there are three data groups (book, reservation, and member). In the second viewpoint, reservation and book are considered to form a single data group, because reservations do not exist without books. Hence, they are sufficiently logically related to form a single data group. This choice has a significant impact on the UFP count, because the number of referenced data groups is used in the calculation of transaction weights.

The UFPs of the library system are: 91, 88, 75, and 112 using B formal rules, CPM1 (viewpoint 1), CPM2 (viewpoint 2), and Use cases of [35], respectively. Two people participated in the counts: the first three were made by the same person; the last count was made by another. The UFP count using our formal rules is 91, which is very close to the IFPUG CPM count based on the first viewpoint (88). The 3.4% difference arises from the evaluation of function component complexity.
The CPM2 count is very different from both the B count (21 %) and the CPM1 count (17 %), due to the fact that books and reservations are combined into a single data group. It is an example of the subjectivity of the function point measure as defined in the IFPUG CPM.

Finally, the UCD (Use Case Diagram) count is significantly higher than the B count (23 %), because it identifies more function components (26 instead of 19). In the UCD approach of [35], function components are identified using events from sequence diagrams. The UCD model presented in [35] decomposes some B operations into several events, which results into a higher number of function components in the UCD count. If another UCD specification style was used in [35], the difference could be smaller.

1.8 Conclusion

The idea of this work was to show the subjectivity and incompleteness of the IFPUG FP definition, on the one hand. On the other hand, it was to provide a formal basis that can be used to develop a tool supporting the counting of FP for B specifications. The main contributions of this study are:

- removing the variance between measures in order to facilitate the validation and evaluation of the measure accuracy;
- providing a better understanding about the application of the IFPUG FP counting rules;
- identifying holes and ensure completeness of the IFPUG FP counting rules which is very helpful for IFPUG users and authors;
- connecting the B language to software metrics;
- increasing the objectivity of the IFPUG FP counting rules;
- providing a basis for the development of a tool that supports the FP measure in order to reduce measurement cost.
The formal rules provided in this paper are, however, specific to the B notation, hence automation is directly applicable to a B specification. Furthermore, these rules might be indirectly applicable to a specification written in OMT or UML, using translation rules from object-oriented notation to the B notation. We also hope that these rules, being formal, will help in reducing the subjectivity of FP counts in other specification languages, because they are more precise than IFPUG CPM counting rules. We also identified cases which were not covered by the IFPUG CPM rules. Therefore, it was necessary to propose modifications to take those cases into account and ensure the completeness of our formal rules.

The interpretation we made of the IFPUG CPM rules is perhaps not exactly the same as the ones used by some experts in the field. However, there is probably no universal agreement on a single interpretation of the IFPUG CPM rules. What is most important is that FP becomes an objective measure, repeatable and free of measurement variance due to human interpretation. That can only improve all activities based on FP measurement. The actual number of FP for a system is not in itself of interest for software managers. What is interesting is cost, quality and productivity. The interest in FP lies in its ability to predict cost, to compute quality ratio (defect / FP), productivity ratio (FP / effort), to benchmark and compare systems based on size (ex: system A is twice as big as system B). The less variance there is in FP measurement, the more reliable these management measures become.

From an analytical viewpoint, there are some open issues to resolve with the formalization of the IFPUG FP definition. More research is needed in order to address the following problems: i) our rules of a data group are based on a classical specification style in B. They may not properly identify a data group if the specification is written in another style; and ii) more investigations are needed to provide an objective definition of derived data and distinguish between optional RETs and mandatory RETs.
Bibliographie


28


Chapitre 2

Formalisation de la méthode
COSMIC-FFP selon le langage ROOM

La formalisation de la mesure de COSMIC-FFP constitue le deuxième volet de notre travail. Ce chapitre présente la première version de cette formalisation pour la notation ROOM (RealTime Object Oriented Modeling), qui a été publié dans les actes de la conférence ACS/IEEE International Conference on Computer Systems and Applications en 2001. Dans cette version, nous avons pris en considération les cas de base. Par exemple, la règle permettant d'identifier les processus fonctionnels ne traite pas les messages échangés entre des capsules situées dans une même couche.

Au cours de nos travaux, la notation ROOM a évolué. ROOM fut d'abord outillée par la compagnie ObjecTime, fondée par les inventeurs de la notation ROOM. En 2001, ObjecTime fut acquis par Rational. Suite à cette acquisition, l'outil d'ObjecTime fut renommé Rational Rose RealTime (RRRT). On apporta au passage quelques modifications au niveau de la notation ROOM, de la génération de code et de certains mécanismes de communication. Ce chapitre a été écrit avant que nous puissions prendre connaissance
de toutes les modifications effectuées par Rational. L'annexe D présente une version complète de la formalisation de COSMIC-FFP pour Rational Rose RealTime qui tient compte des différences et apports entre RRRT et ROOM. Le chapitre 3 présente un outil, $\mu^cROSE$, qui est fondé sur les définitions de l'annexe D.
Formalizing COSMIC-FFP Using ROOM

H. Diab, M. Frappier, R. St-Denis
Département de mathématiques et d’informatique, Université de Sherbrooke
Sherbrooke, Québec, Canada, J1K 2R1
Phone: (819) 821 8000 ext. 2096, Fax: (819) 821 8200
e-mail: {Hassan.Diab, Marc.Frappier, Richard.St-Denis}@dmi.usherb.ca

Abstract

We propose a formalization of the COSMIC Full Function Point (COSMIC-FFP) measure for the Real-time Object Oriented Modeling (ROOM) language. COSMIC-FFP is a measure of the functional size of a software. It has been proposed by the COSMIC group as an adaptation of the function point measure for real-time systems. The definition of COSMIC-FFP is general and can be applied to any specification language. The benefits of our formalization are twofold. First it eliminates measurement variance, because the COSMIC informal definition is subject to interpretation by COSMIC-FFP raters, which may lead to different counts for the same specification, depending on the interpretation made by each rater. Second it allows the automation of COSMIC-FFP measurement for ROOM specifications, which reduces measurement costs. Finally, the formal definition of COSMIC-FFP can provide a clear and unambiguous characterization of COSMIC-FFP concepts which is helpful for measuring COSMIC-FFP for other object-oriented notations like UML.
Introduction

Software development activities must be managed in order to effectively accomplish them. A high risk and cost is associated with these activities. The management of aspects like effort, quality, productivity, benchmarking, and outsourcing depends, to a large extent, on the availability of an appropriate software size measure. Size is a key factor to estimate effort, to compute productivity and quality ratio. Benchmarking relies on the ability to compare systems of similar size. The cost of outsourcing contracts for maintenance and evolution is also driven by the size of the software. Hence, measuring software size is an important goal for any organization developing or using software.

Function point analysis [3] is one of the prominent methods to measure the functional size of a software. It does not depend on a particular functional specification notation or on the implementation of a system. It has gained a considerable popularity in the software industry: more than 1200 people are using it worldwide [24]. Albrecht [3] proposed the first definition of this measure. Its users are now structured as a user group, the International Function Point Users Group (IFPUG), which provides a Counting Practice Manual (CPM) [24]. Function points was designed to measure information systems (data intensive and transaction-based systems). Although it is conceptually applicable to real-time systems, several reports [30, 40, 46] advocate that it is not very well adapted for real-time systems. Different adaptations of function points for real-time software have been proposed. However, it seems that none of these versions has gained a large acceptance. Recently, function points were extended by the Common Software Measurement International Consortium (COSMIC) group to more adequately address real-time software. The resulting measure is called COSMIC Full Function Points (FFP) [1]. COSMIC-FFP is a measure of the functional size of a software.

The COSMIC-FFP measurement function definition is stated in [1] as follows: "The
COSMIC-FFP measurement function is a mathematical function which assigns a numerical value based on the COSMIC-FFP measurement standard. The argument of the COSMIC-FFP measurement function is the sub-process (data movement). The COSMIC-FFP measurement standard, 1 CFSU (Cosmic Functional Size Unit), is defined as one elementary data movement (data entry, exit, read, or write).

The COSMIC-FFP measure may be used to estimate development effort, evaluate software quality, manage outsourcing contracts, and compare systems, specified in different languages, in terms of productivity, quality, and maintenance costs. Consequently, these measurement aspects can be used as the basis to improve and achieve the appropriate Capability Maturity Model (CMM) level of an organization.

In [1], a measurement procedure is defined to enable a consistent application of the COSMIC-FFP method. However, the measurement rules are given in plain natural language, which makes them difficult to apply in a consistent manner and to use as the basis for automated measurement, because some specification concepts do not have a direct representation in the context of COSMIC-FFP (some object-oriented concepts for instance). The lack of a direct representation makes the rules subject to interpretation by COSMIC-FFP raters. This may introduce some variances on COSMIC-FFP measure when different persons apply the measurement rules, in different contexts and at different times, to the same software specification. These variances negatively affect the accuracy of the software size measure. Therefore, mapping rules between COSMIC-FFP and ROOM concepts are required. Finally, COSMIC-FFP calculation is mostly a manual process which is quite expensive; there is no tool for identifying the COSMIC-FFP elements directly from software requirement specifications.

In this paper we propose a formal definition of the COSMIC-FFP measurement procedure given in [1] for mapping COSMIC-FFP concepts to ROOM concepts. This formal definition will eliminate the variance in COSMIC-FFP measurement for ROOM specifications, because this definition is stated in mathematical terms based on the syntactic
structure of a ROOM specification, which should improve the accuracy of software size measurement. It allows to always obtain the same COSMIC-FFP value and in a systematic way. It also allows the automation of COSMIC-FFP measurement for ROOM specifications, which reduces measurement costs of COSMIC-FFP and avoids counting errors. To the extent of our knowledge there is only one approach that has been published on measuring COSMIC-FFP for a notation: UML [7]. It does not, however, address the issues of formalization or automated measurement.

The ROOM language is widely used for developing real-time and distributed software. Rational Corporation provides a case tool (Rational Rose RealTime powered by ObjectTime) that allows the edition, validation, and simulation of ROOM specifications. This includes the ability to execute an incomplete model. ROOM is representative of the family of object-oriented methods for real-time systems. The ROOM language has a formal framework to abstractly and concretely describe the software. It integrates concepts of data encapsulation, inheritance, and information hiding, incorporates detail level programming languages, and provides an executable model at high levels of abstraction.

The structure of a ROOM model facilitates the automation of COSMIC-FFP measurement. The ROOM notation provides the details which are essential to automatically compute COSMIC-FFP. For instance, the use of executable instructions in the body of actions of a transition between states allows to formally identify the data movements and distinguish between them. We cannot formally identify these details in a specification written in semi-formal specification languages like UML. Since ROOM is an object-oriented method, our formal definition could be used as the basis for COSMIC-FFP calculation for other object-oriented methods like UML.

This paper is structured as follows. Section 2.1 provides an overview of the COSMIC-FFP method and the ROOM language. Section 2.2 formalizes some elementary concepts of COSMIC-FFP in the ROOM context which are then used in Section 2.3 to classify software components. Finally, we conclude with an appraisal of this work in Section 2.4,
identifying its strengths, weaknesses, limits, and future work.

2.1. Background

2.1.1. COSMIC Full Functions Points (COSMIC-FFP)

The following equation describes how COSMIC-FFP, denoted by $\text{Size}_{CFSU}(\text{layer}_i)$, are measured:

$$
\text{Size}_{CFSU}(\text{layer}_i) = \Sigma \text{size(entries)} + \Sigma \text{size(exits)} + \\
\Sigma \text{size(reads)} + \Sigma \text{size(writes)}
$$

Variable $\text{Size}_{CFSU}(\text{layer}_i)$ denotes the COSMIC-FFP size of the software in $\text{layer}_i$. A layer is defined as the result of the functional partitioning of the software environment. To obtain that functional size, a set of rules must be applied. These rules consist of mapping the software to measure onto an implicit software model and then measuring the components of this software model. There are two main categories of components: data groups and data movements. A data group is defined as a set of data attributes that are logically related based on the functional perspective. Data movements are divided in four categories: entry, exit, read, and write. A data movement is defined as a function which refers to a set of data attributes.

The size of a data movement is 1 CFSU. The number of all the "entries", "exits", "reads", and "writes" that are identified in the software to be measured are computed by the expression $\text{Size}_{CFSU}(\text{layer}_i)$.

The COSMIC-FFP Measurement Manual (MM) version 2.0 [1] provides rules for software measurement. It also provides rules and procedures for identifying, categorizing, and aggregating components. We want to develop a tool to automatically evaluate these rules.
2.1.2. The ROOM Language

ROOM is a formal modeling language for the development of systems [41]. It is well adapted to real-time systems. The design notation is a combination of graphical and textual specifications. This includes embedded segments of executable code of an object-oriented programming language.

In a ROOM model, the design might be observed via two different view points: structure and behavior. The structure represents the architecture of the model components and the links between these components. The behavior shows how the system may evolve over time. It is affected by the time and the occurrence of some events. The events are generated by the system or by its environment.

A ROOM model is based on three kinds of entity: actors, protocols, and data objects. A complete model is a combination of these entities. An actor is defined as an active object. Indeed, an object exhibits both static and dynamic semantics. The ROOM language integrates the data encapsulation concept. An actor is encapsulated and has restricted visibility of and by other actors. The access relation between the objects is by reference only. An actor offers its services that might be required by the communication channels. Each request of a service corresponds to a specific message. A protocol represents a set of messages that can be exchanged between the actors. Finally, a data object is the basic unit of the system data. It is sent or received in conjunction with the message. It is also used to define actor variables.

The dynamic part of a model is defined in the behavior section of the specification. The behavior of an actor is described as an extended finite state machine. In the ROOM environment, the state machines are called ROOMcharts. They are based on the State-charts formalism of Harel [21]. A ROOMchart is a hierarchical state machine. A state may be decomposed in sub-states. States that are not decomposed are called leaf states. A ROOMchart is defined by a state context. This state context may have variables, entry
code, exit code, a set of sub-states, or a set of transitions. Executable state machines are a feature of the ROOM language.

2.2. Definition of COSMIC-FFP Elementary Concepts

In order to give a formal definition of the measurement rules, we need to first define some elementary concepts of the COSMIC-FFP measure as defined in the COSMIC-FFP MM [1]. We map these concepts to the ROOM notation.

2.2.1. Boundary

"A boundary of a piece of software is the conceptual frontier between this piece and the environment in which it operates, as it is perceived externally from the perspective of its users. The boundary allows the measurer to distinguish, without ambiguity, what is included inside the measured software from what is part of the measured software's operating environment" [1].

Given a set of actors in a ROOM model, it is possible to derive the set of data and protocol classes referenced by these actors. Therefore, the input of the measuring process consists of the text of a ROOM specification and the set of actor names to be measured. This set of actors represents the application boundary to be measured. We denote the application boundary by $B$.

2.2.2. Layer

"A layer is the result of the functional partitioning of the software environment such that all included functional processes perform at the same level of abstraction" [1]. Each layer can be measured separately in COSMIC-FFP. In ROOM, a layer serves as an abstraction of the information hiding mechanism. It is similar to the encapsulation shell of an actor with the addition of a vertical relationship to communicate with other layers.
This means that several layers might be defined at the same level of abstraction. There is no direct mapping between the layer concept of ROOM and the layer concept of COSMIC-FFP, which makes it difficult to automatically identify a layer. Therefore, a human judgment is required to select the subset of actors from the application boundary $B$ that perform at the same level of abstraction. The selected subset corresponds to a layer in COSMIC-FFP. This provides the opportunity to measure any part of the system by selecting the set of actors representing it. Of course the layer boundary is the conceptual demarcation between this subset of actors and the other pieces of the application.

2.2.3. Functional Process

"A functional process in COSMIC-FFP MM is a non-empty set of data movements. The execution of these data movements changes the system state. A functional process implements a cohesive set of functional user requirements. The execution of the functional process must fulfill those functional user requirements" [1]. A functional process corresponds to a transition in the ROOM notation. A transition is implemented by a cohesive set of actions acting on inputs, changing the system state and/or extended state variables, and possibly producing a result.

Typically, a transition between two states is composed of a set of functions. There are six kinds of functions: guard condition appearing in the transition trigger, exit action, choicepoint condition, label action appearing in the transition label, internal function, and entry action. In this paper we refer to each of these functions as an elementary action. A transition is triggered by the arrival of a message through the actor interface (its ports). A message must be declared in the transition trigger. The arrival of a message triggers the transition execution in the following order: guard function of the trigger, exit action of the source state, choicepoint condition, action declared in the transition label, internal function, and finally entry action of the destination state.

Formally we define a transition by a 5-tuple:
\[ T = (s_o, Input_T, F_T, Output_T, s_w) \]

where \( s_o \) denotes the source state; \( Input_T \) denotes the set of all the incoming messages appearing in the trigger of \( T \); \( F_T \) denotes the set of all the elementary actions associated with the transition label, \( s_o \), and \( s_w \); \( Output_T \) denotes the set of all the outgoing messages generated by the elementary actions of \( T \) and sent to outside the boundary; and \( s_w \) denotes the destination state. Note that due to the limited number of pages we omit in this paper the formal definitions of several symbols (e.g., \( Input_T \) and \( Output_T \)) and some elementary concepts (e.g., "exit" and "read").

As an example, we use the specification of an elementary factory system. The factory system includes two \textit{Producer} machines and a \textit{Consumer} machine with a table separating them. In a cycle, a producer machine takes components from its own basket, produces a part, and puts the part on the table. Similarly, a consumer machine removes a part from the table, performs some operations, and feeds its own basket. An attributed controller must supervise the factory in accordance with two constraints: i) the number of parts on the table must not exceed its capacity: 3; ii) a consumer machine must not try to take a part from an empty table. The structure diagram of the factory system is depicted in Figure 1. It contains the two Producer machines represented by replicated actor \textit{Producer} with an instance factor (2), the Consumer machine represented by the actor \textit{Consumer}, and the Controller actor.

For the sake of simplicity, we would like to keep our example small. Only the Controller actor (called \textit{capsule} in the Rational Rose RealTime toolset) will be considered. Its behavior diagram is illustrated in Figure 1. All its substates are leaf states. In this example, we assume that the boundary only contains the Controller actor. In the rest of this paper we will use this example to illustrate the application of the formal rules.

To illustrate how we identify a functional process, consider the excerpt of the Controller actor specification provided below.
FIG. 1 - The structure diagram of the factory system and the behavior diagram of actor Controller

(1) State: Controlling
(2) Type: State
(3) Parent State: TOP
(4) Actions:
(5) Entry:
(6) IF numberOfPart +

    numberOfProducer <3
    && P1_busy == FALSE
    /* Producer_1 is not busy */

    {Put[1].Start_Machine().send();
        numberOfProducer =
        numberOfProducer+1;}

(7) IF numberOfPart >0

    {Remove.Start_Machine().send();}
(8) IF numberOfPart +
    numberOfProducer < 3
    && P2_busy == FALSE
    {Put[2].Start_Machine().send();
     numberOfProducer =
     numberOfProducer + 1;
    }

(9) Transition: putDoneP1

(10) From: : WorkingP1:

(11) To: : Controlling

(12) Event Guards:

(13) Event Guard:

(14) Ports: Put[1]

(15) Signals: Machine.Busy

(16) Guard: return (*rtdata == FALSE)

(17) Action:

(18) numberOfProducer =
    numberOfProducer - 1;

(19) numberOfPart = numberOfPart + 1;

(20) P1_busy = *rtdata;

The execution of transition putDoneP1 (between the WorkingP1 and Controlling states) will be triggered by the arrival of message Machine.Busy and if the guard condition is true. Following (1), putDoneP1 is formally defined as follow:

\[
\text{putDoneP1} \triangleq (\text{WorkingP1}, \text{Machine.Busy},
\text{F}_{\text{putDoneP1}}, \text{Start_Machine},
\text{Controlling}),
\]

where WorkingP1 is the source state; Machine.Busy is the incoming message; \( F_{\text{putDoneP1}} \) represents the executable statements defined within the entry action block of Controlling state and the action blocks of transition putDoneP1; Start_Machine denotes the outgoing
message generated during the transition processing; and Controlling is the destination state.

2.3. Classifying Components

In the COSMIC-FFP definition, there are six types of component to count: data group, data attribute, entry, exit, read, and write. In this section, we provide informal definitions of the COSMIC-FFP measurement rules as reported in the COSMIC-FFP MM followed by our formal definition for the ROOM notation.

2.3.1. Data Group

"A data group is a distinct, non-empty, non-ordered and non-redundant set of data attributes where each data attribute describes a complementary aspect of the same object of interest. A data group is characterized by its persistence" [1]. An object in the object-oriented paradigm and an entity in the entity/relationship model are typical candidates of a data group.

In a ROOM model, actor variables and data included in messages within protocol classes represent data of the system. Actors and protocol classes refer to data classes and predefined data types in order to define the type of their components. An actor may have simple variables and/or complex variables. A variable is said to be simple if its type is a scalar data type (e.g., integer, boolean, string), an array of a scalar type, or a predefined data class. A variable is said to be complex if a non-predefined data class is used to define its type. The set of simple variables of actor $a$ is denoted by $V_{simple}(a)$ which is defined as follow:

$$V_{simple}(a) \triangleq \{ v \mid v \in variables(a) \land \neg(\exists c : c \in DC \land type(v,c))\}$$ (2)
where $\mathcal{DC}$ denotes the set of all the non-predefined data classes declared in the system; $\text{variables}(a)$ denotes the set of all the variables declared in actor $a$; and $\text{type}(v, c)$ denotes that the type of variable $v$ is data class $c$.

Similarly, a message within a protocol may include data. This data may be simple or complex, as actor variables. The set of simple data of message $m$ is denoted by $D_{\text{simple}}(m)$ which is defined as follow:

$$D_{\text{simple}}(m) \triangleq \{d \mid \text{handles}(m, d) \land \\
\neg(\exists \ c : c \in \mathcal{DC} \land \\
\text{type}(d, c))\}$$

(3)

where $\text{type}(d, c)$ denotes that the type of data $d$ is data class $c$; and $\text{handles}(m, d)$ denotes that message $m$ includes data $d$.

A data group in COSMIC-FFP corresponds to a data class, a set of actor simple variables, or a set of simple data included within message. There are four kinds of data groups that must be identified. The following rules are given: Rule (4) provides the set of data classes used by the actors within the boundary; Rule (5) provides the set of actor simple variable groups; Rules (6) provides the set of data classes embedded in messages sent and/or received by the actors within the boundary; Rule (7) provides the set of simple data embedded in messages sent and/or received by the actors within the boundary;

$$\text{DataClass} \triangleq \{c \mid c \in \mathcal{DC} \land \\
(\exists \ a, v : a \in B \land \\
v \in \text{variables}(a) \land \\
\text{type}(v, c))\}$$

(4)

$$\text{DataVariables} \triangleq \{(a, V_{\text{simple}}(a)) \mid a \in B\}$$

(5)
\[ DataMessage \triangleq \{ c \mid c \in DC \land \\
(\exists \ m,d : m \in M \land \\
type(d,c) \land \text{handles}(m,d)) \} \]  
(6)

\[ \text{SimpleDataMessage} \triangleq \{(m,D_{\text{simple}}(m)) \mid m \in M\} \]  
(7)

where \( M \) is the set of all the messages exchanged between the actors within the boundary and the environment via the external ports and the messages exchanged between the actors within the boundary.

Finally, the set of data groups, denoted by \( DG \), used by the actors and protocols within the boundary is defined as the union of the four sets: \( DataClass \), \( DataVariables \), \( DataMessage \), and \( SimpleDataMessage \).

\[ DG \triangleq DataClass \cup DataVariables \cup \\
DataMessage \cup SimpleDataMessage \]  
(8)

To illustrate how we identify the set of data groups \( DG \), consider the excerpt of the Controller actor specification provided below.

(1) **Capsule:** Controller
(2) ...
(3) **Public Interface:**
(4) **Attributes:**
(5) \( \text{numberOfPart} : \text{int} = 0 \)
(6) \( \text{numberOfProducer} : \text{int} = 0 \)
(7) \( P1\_busy : \text{bool} = \text{FALSE} \)
(8) \( P2\_busy : \text{bool} = \text{FALSE} \)
(9) ...
(10) **Protocol:** ControlMachines
The Controller actor has four attributes: \textit{numberOfPart}, \textit{numberOfProducer}, \textit{P1\_busy}, and \textit{P2\_busy}. By applying rules (2), (4), and (5) to the Controller actor specification, we conclude that there is one data group: \((\text{Controller}, V_{simple}(\text{Controller}))\) denotes the set of simple variables that contains the four attributes. To identify the data groups included in messages that are exchanged during the system processing, we apply rules (3), (6), and (7) to protocol \textit{ControlMachines} which is used as the type of \textit{Put} and \textit{Remove} ports of Controller actor. There is one message, \textit{Machine\_Busy}, that includes simple data of type boolean. In this case, \((\text{Machine\_Busy}, D_{simple}(\text{Machine\_Busy}))\) is identified as a data group. Finally, we apply rule (8) to identify the data groups that must be included in \(DG\). In this example, \(DG\) contains two data groups: \((\text{Controller}, V_{simple}(\text{Controller}))\) and \((\text{Machine\_Busy}, D_{simple}(\text{Machine\_Busy}))\).

2.3.2. Data Attribute

"An attribute is the smallest parcel of information, within an identified data group, carrying a meaning from the perspective of the software's FURs" [1]. In ROOM, a data attribute is defined either as a simple actor variable or an attribute of a data class. An attribute may be simple or complex. A complex attribute may be decomposed into attributes that are defined at a lower level. We refer to an attribute that is not decomposed into finer attributes as an "elementary attribute". This kind of attributes represents the smallest information in the ROOM notation which corresponds to a data attribute in
COSMIC-FFP. Attributes(dg) denotes the set of attributes of a data group dg ∈ DG.

2.3.3. Data Movements

4.3.1. Entry. "An entry (E) is a movement of the data attributes found in one data
group from the user’s side of the software boundary to the inside the software boundary.
An “entry” does not update the data it moves. Functionally, an “entry” sub-process
brings data lying on the user’s side of the software boundary within reach of the functional
process to which it belongs. Note also that in COSMIC-FFP, an “entry” is considered to
include certain associated data manipulation (validation) sub-process” [1].

In a ROOM model, an incoming message handles a request from outside to inside the
boundary through the external ports on the interlayer boundary. The number of data
groups referred by a message determines the number of entries to which an incoming
message must be mapped. Therefore, for each incoming external message the set of data
groups that are referred in it should be identified (which corresponds to the number of
entries). Note that the same principle is applied to the other data movements (“read”,
“write”, and “exit”).

Rule (9) provides a formal definition that allows to identify the data groups carried
out in message m within the incoming message set of transition T through the external
ports.

\[
Entry(T) \equiv \{ dg \mid dg \in DG \land \\
\exists m,d : m \in Input_T \land \\
m \in InMessage_{ext} \land \\
(type(d,dg) \lor \\
d \in Attributes(dg)) \land \\
handles(m,d) \} \\
\text{(9)}
\]

where \(type(d, dg)\) denotes that the type of data object \(d\) is data group \(dg\) and \(InMessage_{ext}\)
denotes the set of all the incoming messages sent from outside the boundary.
By applying rule (9) to transition putDoneP1, we can identify one data group, \((Machine\_Busy, D_{simple}(Machine\_Busy))\), that has been included in message \(Machine\_Busy\) sent from outside the boundary, since message \(Machine\_Busy\) is received via port \(Put\) which communicates with other actor (e.g., \(Producer\)) (lines (14) and (15) in the specification of transition \(putDoneP1\)). The \(Entry(putDoneP1)\) contains one data group. Consequently, its weight is computed by the use of the set cardinality of \(Entry(putDoneP1)\) which is one CFSU.

4.3.2. Write. “A write \((W)\) refers to data attributes found in one data group. Functionally, a “write” sub-process sends data lying inside the functional process to which it belongs to storage. Note also that in COSMIC-FFP, a “write” is considered to include certain associated data manipulation sub-process” [1].

In a ROOM model, only the elementary actions, \(F_T\), of transition \(T\) might refer to data attributes and possibly change their values. \(F_T\) is said to maintain a data group \(dg\) if and only if there is an instruction in an elementary action within \(F_T\) that at least modifies the value of one attribute of \(dg\). The group of attributes that are referred to and modified by the elementary actions within \(F_T\) will be used to determine the number of “writes” to which those elementary actions should be mapped.

Rule (10) provides the set of data groups maintained by \(F_T\) during the execution of transition \(T\).

\[
Write(T) \triangleq \{dg \mid dg \in \text{DG} \land \\
\exists \; v : v \in \text{Attributes}(dg) \land \\
maintains(F_T, v)\}
\]

(10)

where \(maintains(F_T, v)\) denotes that there is an executable instruction in an elementary action within \(F_T\) that changes the value of variable \(v\), considered as an attribute of data group \(dg\).

During the processing of transition \(putDoneP1\), there is one data group, (Controller,
\( V_{\text{simple}}(\text{Controller}) \), that will be maintained, since there is at least one attribute (e.g., \textit{numberOfProducer} of (Controller, \( V_{\text{simple}}(\text{Controller}) \)) which is maintained by executable statements defined in the action blocks of state \textit{Controlling} and transition \textit{putDoneP1} (lines 6 and 8) in the specification of state \textit{Controlling} and lines (18, 19, and 20) in the specification of transition \textit{putDoneP1}. In this case \textit{Write(putDoneP1)} has one data group. Consequently, its weight is computed by using the set cardinality of \textit{Write(putDoneP1)} which is one CFSU.

The COSMIC-FFP functional size of this example is 25 CFSU, when all data movements are measured.

2.4. Conclusion

In this paper, we have proposed formal measurement rules for ROOM specifications according to COSMIC-FFP informal definitions given in [1]. Being formal, our rules are objective, which should improve the accuracy of the software size estimates.

Formal rules are founded on first order logic and set theory. They could be used as the basis for developing a software size measurement tool that supports the specification language. This tool can be integrated in the ObjeCTime/Rational Rose RealTime toolset that supports ROOM. Automation provides a good opportunity to reduce measurement costs of COSMIC-FFP measure and to avoid measurement errors, while making the measurement process objective. Our rules are, however, specific to the ROOM notation, hence automation is directly applicable to ROOM specifications.

The formal rules reported in this paper are just the first step of our work. During the formalization process, we found several interpretations of the COSMIC-FFP rules. We have chosen the interpretation which seemed the most "reasonable", but further validations with a group of COSMIC-FFP experts will be necessary. We need to apply the formal rules to several systems and compare the results with manual measurements conducted by several experts to validate their suitability. The difference between the
results will be analyzed in order to identify the elements that cause it. Implementing the formal rules in the Rational Rose RealTime toolset will streamline this validation.

In the future, we want to analyze how COSMIC-FFP could be measured from partial ROOM specifications, in order to estimate COSMIC-FFP during specification construction. We also plan to address the estimation of COSMIC-FFP for the UML notation, although UML documents do not always contain all the necessary details to measure COSMIC-FFP automatically. We are thinking of a semi-automated measurement approach where heuristics are used to identify COSMIC-FFP components from various UML diagrams.
Bibliographie


Chapitre 3

Automatisation de la mesure de COSMIC-FFP pour *Rational Rose RealTime*

Durant la dernière décennie, plusieurs organisations ont exprimé un grand intérêt dans l'évolution de leur processus de développement selon le modèle CMM (*Capability Maturity Model*). Puisque la mesure joue un rôle important dans le modèle CMM, il est essentiel que des outils supportant la mesure de la taille du logiciel soient exploités. Dans ce chapitre, nous présentons l'outil \( \mu.ROSE \) qui mesure la taille fonctionnelle d'un logiciel selon la méthode COSMIC-FFP pour des spécifications produites par l'outil Rational Rose RealTime. \( \mu.ROSE \) permet une mesure systématique, reproductible et consistante avec une intervention minimale des utilisateurs, tout en réduisant le coût du calcul de la mesure. Notons que \( \mu.ROSE \) est le premier outil qui supporte la mesure automatique de la méthode COSMIC-FFP. Il peut facilement être intégré dans l'outil Rational Rose RealTime. Cet article a été soumis pour publication à la revue *Empirical Software Engineering.*
\( \mu_c\text{ROSE} \): Automated Measurement of COSMIC-FFP for Rational Rose RealTime

H. Diab, F. Koukane, M. Frappier, R. St-Denis
Département de mathématiques et d’informatique, Université de Sherbrooke
Sherbrooke, Québec, Canada, J1K 2R1
Phone: (819) 821 8000 ext. 2096, Fax: (819) 821 8200
e-mail: \{Hassan.Diab, Fouad.Koukane, Marc.Frappier, Richard.St-Denis\}@dmi.usherb.ca

Abstract

During the last ten years, many organizations have invested resources and energy in order to be rated at the highest level as possible according to the Capability Maturity Model. Since measures play an important role in maturity models, it is essential that CASE tools offer facilities to automatically measure the sizes of various documents produced using them. This paper introduces a tool, called \( \mu_c\text{ROSE} \), that automatically measures the functional software size, as defined by the COSMIC-FFP method, for Rational Rose RealTime models. \( \mu_c\text{ROSE} \) streamlines the measurement process, ensuring repeatability and consistency in measurement while reducing measurement cost. It is the first tool to address automatic measurement of COSMIC-FFP and it can be integrated into the Rational Rose RealTime toolset.
Introduction

This paper describes $\mu$ROSE (pronounced McRose), a CASE tool that automatically measures COSMIC-FFP for Rational Rose RealTime (RRRT) models. COSMIC-FFP is a functional size (FS) measure initially designed for real-time systems [4]. It was inspired from Albrecht's function points (FP), widely used in industry for information systems [2]. COSMIC-FFP addresses some shortcomings of FP for real-time systems and simplifies the measurement task. It should be noted that COSMIC-FFP is not restricted to real-time systems; it can also be used for information systems. It is the first method that conforms to ISO/IEC 14143, which specifies a set of generic mandatory requirements for a method to be called a functional size measurement method; COSMIC-FFP has also been recently adopted as an ISO/IEC standard (19761).

RRRT is a CASE tool for designing embedded, real-time, and distributed systems. It originates from the acquisition and evolution of the ObjecTime toolset by Rational. ObjecTime was created to support the Real-Time Object-Oriented Modeling (ROOM) language [16]. RRRRT is mainly used in telecommunications, avionics, and process control. The main reasons for selecting RRRRT as the first target for COSMIC-FFP measurement automation are its market penetration and close correspondence with COSMIC-FFP.

FS measures like COSMIC-FFP and FP are applied in several key areas of the software process. The first motivation behind the development of FS measures was to estimate the software development cost (effort), because size is the main factor influencing cost. FS is more relevant than lines of code to estimate cost early on in the software development process, the latter being measured too late. FP and COSMIC-FFP have been successfully used to build cost estimation models. They can also be used to measure productivity (by computing FS unit per effort unit) or quality (by computing defect per FS unit). An organization may also benchmark its productivity across several organizations using FS measures. For instance, the International Software Benchmarking Standards Group [11]
collects and provides data to its members, while preserving anonymity. Members can benchmark their productivity based on FP or COSMIC-FFP.

COSMIC-FFP can be measured from various documents and at various times during the software process. At the beginning of a project, it is measured from the functional requirements and can be applied in a cost estimation model to estimate cost. Costs derived in the requirements definition phase can be adjusted during design or implementation by re-measuring COSMIC-FFP from design documents or the source code, since the functionalities usually evolve during a project. When a project is coming to an end, the final COSMIC-FFP size and actual effort can be stored in a database of completed projects in order to build or improve a cost estimation model.

Currently, the COSMIC-FFP measurement process is manual, performed by experts who must gather the appropriate information from project documentation. COSMIC-FFP is defined in plain natural language and therefore not specific to a particular functional specification notation. It is formulated in terms of simple concepts like functional process, data group, and data movement. A measurer must interpret these concepts for the specification notation at hand. This interpretation can be subjective (i.e., the measurement may vary depending on the measurer) and may lead to repeatability problems (measurement variance). A recent report on field trials of COSMIC-FFP notes that nearly perfect repeatability is obtained by experienced engineers, but junior engineers show poor repeatability [1]. The measurement rules are subject to interpretation, as noted in a preliminary study based on an experiment with a substantial system [7]. FP is also subject to repeatability problems and interpretations. Studies revealed differences varying from 11% to 30% between several measurements of the same specification by different measurers [8, 12, 14, 13]. These figures hold when the count boundary is well-defined, but larger variations are observed when boundaries are not clearly delineated. There are various causes for measurement variance: different (acceptable) interpretation of a measure, lack of proper training on measurement, and ambiguous or incomplete
functional requirements document.

$\mu_c\text{ROSE}$ improves the COSMIC-FFP measurement process in two ways. First, it almost eliminates measurement costs and advanced measurement training, because measurement is automatic. The measurer only has to identify the subset of the system that has to be measured; $\mu_c\text{ROSE}$ handles all the analysis and calculations of COSMIC-FFP by processing an RRRT model. Second, it removes measurement variance and ensures perfect repeatability, because the measurement algorithm is completely automated. This algorithm is based on the author's interpretation of the COSMIC-FFP definition and has been validated by experts who participated in defining the COSMIC-FFP measure. This interpretation has been formalized in a mathematical definition that is publicly available in a companion paper [5]. $\mu_c\text{ROSE}$ implements this mathematical definition.

The current version of $\mu_c\text{ROSE}$ requires an all-inclusive RRRT model, so that it can be used from the end of the programming phase to measure the actual value of COSMIC-FFP for the corresponding system. This means that $\mu_c\text{ROSE}$ can assist in building a database of completed development projects (containing actual COSMIC-FFP size, actual cost, and other measures) in order to build cost estimation and defect prediction models, and manage software maintenance and outsourcing, all of which are based on functional software size.

Automation of FS measurement has attracted the interest of both case tools vendors and researchers. According to Mendes et al. [15], most commercial tools provide clerical support for storing FP components and calculating FP from a list of FP components. Eight tools claim to measure FP directly from functional requirements models (e.g., data flow diagrams, entity-relationship diagrams, or object models), but their accuracy has not been independently validated. Moreover, their measurement algorithm has not been published. HierarchyMaster FFP is a clerical tool that allows the recording of functional processes and data movements, and the computation of COSMIC-FFP [10]. In contrast, $\mu_c\text{ROSE}$ automatically extract the list of functional processes, data groups, and data
movements from an RRRT model.

The rest of this paper is organized as follows. Sections 3.1 and 3.2 summarize the main concepts of COSMIC-FFP and RRRT, respectively. They also introduce the terminology used in this paper. Section 3.3 presents a tour of $\mu_cROSE$, particularly its main functionalities, architecture, and interface. Section 3.4 gives two class models that capture concepts of COSMIC-FFP and RRRT related to the measurement process and establishes connections between them. Section 3.5 concerns the validation and verification of $\mu_cROSE$. Finally, Section 3.6 concludes with a summary of the key points presented in this paper, $\mu_cROSE$ limitations, and future directions.

3.1 A Brief Overview of COSMIC-FFP

In COSMIC-FFP, a system can be decomposed into layers. Each layer represents a level of abstraction that performs a set of functional processes. In this context, software in one layer exchanges data with software in another layer through their respective functional processes. A functional process is defined as a unique set of data movements whose execution is triggered by an event that occurs in the environment. A data movement is defined as a particular subprocess that is either an entry, an exit, a read, or a write. The four types of subprocess represent an input received from the environment (entry), an output sent to the environment (exit), data read from the storage side (read), and data updated within the storage side (write). The environment is either another software layer, a user, or a physical device. Elementary data used by a functional process is called data attribute. A data group is defined as a set of data attributes that are logically related based on a functional perspective.

As stated in the COSMIC-FFP measurement manual [4], the measurement function and measurement standard are as follows:

The COSMIC-FFP measurement function is a mathematical function which
assigns a value to its argument based on the COSMIC-FFP measurement standard. The argument of the COSMIC-FFP measurement function is the sub-process.

The COSMIC-FFP measurement standard, 1 Cfsu (Cosmic Functional Size Unit), is defined as one elementary data movement.

The following expression describes how the number of COSMIC-FFP is measured:

\[
Size_{Cfsu}(layer_1) \triangleq Card(\text{entries}) + Card(\text{exits}) + \\
Card(\text{reads}) + Card(\text{writes})
\]

The term \(Size_{Cfsu}(layer_1)\) denotes the FS of a software at \(layer_1\). Variables \(\text{entries}, \text{exits}, \text{reads}, \text{writes}\) denote the sets of all the entries, exits, reads, and writes that are identified in \(layer_1\), respectively. To compute the number of elements into each of these sets, the cardinality operator is used. For instance, the term \(\text{Card}(\text{entries})\) represents the number of elements into \text{entries}.

The aforementioned expression is used in the measurement phase of the COSMIC-FFP measurement method. Prior to this phase is the mapping of the software to be measured to the COSMIC-FFP software model based on rules and procedures prescribed by the COSMIC-FFP measurement method. \(\mu\text{ROSE}\) formalizes theses rules and procedures, and applies them to RRR models.

### 3.2 An Overview RRRT

The paradigm behind RRRT allows developers to model architectures and designs of software systems that are highly event-driven, concurrent, and distributed. In addition, these systems must generally satisfy real-time constraints. RRRT models result from the use of a graphical and textual notation based mainly on UML [3]. Embedded segments of executable code written in an object-oriented programming language like C++ form
a large part of RRRT models. Hence, RRRT supports a software life cycle from use case analysis through design, code generation, and model execution. A toolset, which comprises model editors, navigators, an incremental model compiler, static and dynamic checkers, and code generators, makes it possible to create, modify, simulate, and implement models.

In an RRRT model, the design might be observed via two different viewpoints: structure and behavior. The structure describes the system's architecture in terms of components and links between them. The behavior represents system dynamics. It shows how the system may evolve over time and depends on the time and occurrence of some events.

The structure of an RRRT model is built from three kinds of entity: capsules, protocols, and data classes. A capsule is an active entity that exhibits both static and dynamic semantics. The RRRT paradigm integrates the encapsulation principle. A capsule is encapsulated and has restricted visibility with respect to other capsules. A capsule offers its services that might be reached through its ports and connectors. Each service request corresponds to a specific message, also called signal. A port must refer to a protocol that represents a set of possible messages exchanged between capsules. Finally, a data class is the basic unit for data representation. An object of a data class may be sent or received in conjunction with a message. A data class is also used to define capsule attributes.

The behavior of an RRRT model specifies the dynamic aspects of each capsule by using an extended finite state machine inspired from Harel's Statecharts [9]. A state may be decomposed into substates. States that are not decomposed are called leaf states. A state machine is defined by a state context that may have variables, an entry action, an exit action, substates, and transitions. Executable state machines are a feature of RRRT. An action associated with a transition or state may incorporate code written in an object-oriented programming language.

To illustrate, Figure 2 contains a part of an RRRT model for an elementary factory system that includes two producer machines and one consumer machine with a table
separating them. In a cycle, a producer machine takes components from its own basket, produces a part, and puts the part on the table. Similarly, a consumer machine removes a part from the table, performs some operations, and feeds its own basket. A controller must supervise the factory in accordance with two constraints: i) the number of parts on the table must not exceed its capacity; ii) a consumer machine must not try to take a part from an empty table. The structure diagram (left part in Figure 2) identifies the components of the factory system represented by the capsules Producer (a replicated capsule with an instance factor fixed to 2), Consumer, and Controller. The state machine (right part in Figure 2) gives the behavior of the capsule Controller. For instance, the transition labeled requestPutP1 from the state Controlling to the state WorkingP1 indicates that the first producer has the permission to put a part on the table. The transition labeled putDoneP1 from WorkingP1 to Controlling represents the fact that the first producer has effectively put a part on the table. The other transitions act in a similar way. Segments of executable code written in C++ are added to actions associated with the different transitions.

**FIG. 2** - The structure diagram of the factory system and state machine diagram of capsule Controller
3.3 A Tour of $\mu_cROS$E

$\mu_cROS$E offers basic facilities to aid measurers and project managers in the measurement process. It accepts software requirements of any size written in the RRRT notation and calculates its FS based on the COSMIC-FFP method. A user can select any collection of capsules included in an RRRT model and obtain measurement data with a small amount of interactions. The main functionalities supported by $\mu_cROS$E are: i) visual support of the COSMIC-FFP measurement process; ii) generation of an RRRT model in XML format; iii) extraction of RRRT entities required in the measurement process; iv) analysis of C++ code included in the RRRT model; v) identification of functional processes, data groups, and data movements; vi) calculation of COSMIC-FFP, representing the FS; and vii) aggregation and reporting of measurement results.

These functionalities represent the initial core to be considered as a minimum to make experiments on COSMIC-FFP with large real-time software systems over a short period and validate more deeply the theoretical framework implemented in $\mu_cROS$E, which maps RRRT concepts to COSMIC-FFP concepts. Finally, $\mu_cROS$E has been developed in Java using Sun microsystems JDK 1.3 on a Windows NT platform.

3.3.1 The Architecture

Figure 3 shows the functional architecture of $\mu_cROS$E. It comprises four main modules organized around the Manager module, which coordinates their activations and manages data exchange.

The XML Generator reads an RTMDL file, which contains the internal representation of an RRRT model, analyzes it, and produces a corresponding lightened file in XML format. The analysis includes the extraction of RRRT entities that are pertinent for the application of the COSMIC-FFP method and omission of irrelevant data. XML Generator output is used by the Object Constructor as input in order to build and store in the
main memory instances of Java classes that correspond to the extracted RRRT entities. The Object Constructor also supports the visualization of these entities, since it constructs a hierarchy of RRRT entities that is presented in a tree view by the Graphical User Interface module.

From the selected capsules, the COSMIC-FFP Component Builder determines the functional processes, data groups, and data movements inside the boundary. It classifies data movements according to their type, assigns a numerical value to each functional process, and aggregates the measurement results. This module is strongly connected with the C++ Parser module that accepts a piece of C++ code and a list of attributes. The parser analyzes the syntax of the code to characterize the attributes as modified attributes or referred attributes in order to classify data movements. Finally, the Graphical User Interface module provides a user-friendly interaction between the user and $\muROSE$.

![Diagram](image)

**FIG. 3** - The functional architecture of $\muROSE$
3.3.2 The Interface

The interface includes a main window with four tab boxes. In the following figures, the tab boxes contain data specific to the factory system introduced in Section 3.2. As shown in Figure 4 the RRRT Model tab box is composed of three panels: a tree view of RRRT entities, a list of attributes, and a list of selected capsules.

![Image of RRRT Model tab box]

**Fig. 4.** The RRRT model tab box

The measurement process starts by opening an RTMDL file, which contains an RRRT model, from the File menu. The entities extracted from the RRRT model by the XML Generator are brought in the tree view panel. The user then selects capsules one-by-one from the tree view panel and adds them to the selected capsules panel. The selected capsules must belong to the same level of abstraction to be considered as a layer in the COSMIC-FFP context. Note that μROSE cannot verify if the selected capsules belong to the same layer. The concept of layer is subjective, hence it is difficult to formalize. The Add (Remove) button is used to add (remove) capsules to (from) the
selected capsules panel. Finally, the functional size is calculated by clicking the Measure Cosmic-FFP button. A summary of the measurement results is then displayed in the Cfsu Summary by Process tab box (see Figure 5). It contains the functional processes, number of Cfsu assigned to each functional process, and total number of Cfsu representing the FS. More details about the measurement results can be displayed by selecting the Cfsu Details (Figure 6) and Cfsu by Subprocess Type (Figure 7) tab boxes.

**FIG. 5.** Summary by process of COSMIC-FFP measurement

**FIG. 6.** Details of COSMIC-FFP measurement
3.4 The Class Models of $\mu_c$ROSE

In addition to the functional architecture introduced in Section 3.3.1, several models have been derived during the design of $\mu_c$ROSE, in particular RRRRT and COSMIC-FFP class models. These models capture RRRRT entities (e.g., capsules, protocols, ports, messages, attributes) and COSMIC-FFP components (e.g., functional processes, data groups, data movements), respectively. Thus, each class is considered an abstraction of a group of objects (entities or components) that share the same attributes, operations, relationships, and semantics.

The processing performed by the Object Constructor and COSMIC-FFP Component Builder largely depends on these models. On one hand, the Object Constructor creates instances of classes in $\mu_c$ROSE’s RRRRT model to represent entities from which different measures can be derived. On the other hand, the COSMIC-FFP Component Builder uses these instances in a specific context. It instantiates objects in $\mu_c$ROSE’s COSMIC-FFP model in order to calculate COSMIC-FFP. This processing expresses a mapping from a set of entities in an RRRRT model to a representation in a COSMIC-FFP model.

3.4.1 The RRRRT Class Model

A capsule is the fundamental modeling element of an RRRRT model. A capsule may have the following properties: attributes, ports, operations, behavior, supercapsule, and subcapsules. Figure 8 shows the class Capsule and its associations with the classes
Attribute, Port, Action (representing capsule operations), and State (representing capsule behavior). These associations describe the relationships between a capsule and its properties. The class Capsule also has two reflexive associations. The association inherits describes the inheritance relationship between capsules while the association is_composed_of describes the relationship between a composite capsule and its components. A capsule is said to be composite if it contains one or several capsules.

![Class model for RRRT models](image)

**Fig. 8**: Class model for RRRT models

The associations between the classes Port, Protocol, and Message reflect the exchanges of messages between capsules. A capsule is awakened by the arrival of a message through one of its ports. A port refers to a protocol that specifies a set of incoming messages that can be received and a set of outgoing messages that can be sent through the port. In an RRRT model, the inheritance concept is also applicable to protocols.

A hierarchical state machine is used to describe a capsule's behavior. In Figure 8,
a hierarchical state machine is bound to the class State. It has a reflexive association necessary to express the concept of composite state that contains a lower-level state machine. The class Action plays a central role in the RRRT class model. An action is materialized by statements written in a programming language. It may be associated with a state as an entry or exit action. An entry action is executed whenever the corresponding state is entered. An exit action is executed whenever an outgoing transition from the corresponding state is taken. An action may also be associated with a transition and is executed whenever the transition is triggered, but after the exit action of its source state and before the entry action of its target state. The class Transition has associations with others classes (e.g., State, Trigger). Besides an action, a transition has a source state, a target state, and triggers that specify the names of signals that trigger the transition and ports from which these signals are received. Furthermore, a trigger may have an optional guard condition defined as a Boolean expression. This is captured by the association may_have between Trigger and Action classes.

A data class in an RRRT model is similar to a class in C++ or Java. It includes one or several attributes (data fields) and operations (methods). A data class may be used to define the type of an attribute peculiar to a capsule or another data class. An operation is a service that affects the internal state of an object. Data classes provided by the RRRT toolset as well as C++ primitive data types like int and char are considered as instances of Predefined_Class. Classes defined in an RRRT model (i.e., introduced by designers) are considered as instances of Defined_Class. During the computation of COSMIC-FFP, all attributes of type Predefined_Class in a capsule are grouped together to form a single data group. Each capsule attribute of type Defined_Class is identified as a single data group. As shown in Figure 8, classes Defined_Class and Predefined_Class are two subclasses of the superclass Data_Type. The class Data_Type has an association with the class Attribute, since each capsule attribute must be typed.
3.4.2 The COSMIC-FFP Class Model

A class model for COSMIC-FFP components is shown in Figure 9. It shows only those components for measurement automation. For instance, the class `Functional_Process` denotes the functional processes of a COSMIC-FFP software model. It has associations with different classes to take into consideration definitions and limitations specified by the COSMIC-FFP method.

Fig. 9 – Class model for COSMIC-FFP software models

First, the association `initiates` reflects the fact that the execution of a functional process must be triggered by the occurrence of an event from outside the boundary. A triggering event may initiate one or several functional processes.

Second, the association `performs_at` indicates that a functional process implements a service that is provided at a specific layer. According to the COSMIC-FFP method, the measurement process is separately applied to each layer of a software model. A boundary represents the conceptual demarcation of a layer.

Third, a functional process is defined as a unique set of data movements. Based on the way attributes are used by a functional process, data movements can be identified as well as their type. A functional process may: i) receive one or several data groups
from outside the boundary as input; ii) produce one or several data groups to outside the boundary as output; iii) refer to one or several data groups from the storage side without modification of their values; and iv) refer to one or several data groups from the storage side with modification of their values. These four cases are captured by the association is_moved_by between the classes Data_Movement and Data_Group and the specialization of the class Data_Movement.

Finally, it should be noted that a functional process must have at least one entry and one write or exit.

3.4.3 The Mapping between COSMIC-FFP and RRRT Class Models

The COSMIC-FFP measurement rules can be applied to any specification language, in particular ROOM. Unfortunately, there are two major sources of misinterpretation. First, some concepts of RRRT do not have a direct representation in the COSMIC-FFP method; other concepts of RRRT are broader than COSMIC-FFP concepts. Second, in the FS software metrics area, the greater the degree of independence from specification languages, the lower the degree of formality used in their definition. Indeed, COSMIC-FFP measurement rules are defined in natural language. These drawbacks may lead two persons to come up with different size measures when applying FS measurement methods to the same software document.

Ambiguities arise when measurers must interpret, for example, the definition of functional process given on page 26 of the COSMIC-FFP measurement manual [4].

A functional process is a unique set of data movements (entry, exit, read, write) implementing a cohesive and logically indivisible set of Functional User Requirements. It is triggered directly, or indirectly via an ‘actor’, by an Event (-type) and is complete when it has executed all that is required to be done in response to the triggering Event (-type).
The expressions *cohesive* and *logically indivisible* appearing in this definition are highly subjective. Furthermore, they are very difficult to define precisely. Another example is the following sentence: *A triggering event initiates one or several functional processes.* It is not clear what kind of triggering events may initiate several functional processes and how to be sure to not count them as one functional process.

**Tab. 2 – Correspondence between COSMIC-FFP and RRRT concepts**

<table>
<thead>
<tr>
<th>COSMIC-FFP Concepts</th>
<th>RRRT Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>layer</td>
<td>set of capsules</td>
</tr>
<tr>
<td>boundary</td>
<td>conceptual demarcation of a set of capsules</td>
</tr>
<tr>
<td>data group</td>
<td>message or a data type</td>
</tr>
<tr>
<td>functional process</td>
<td>one or several transitions</td>
</tr>
<tr>
<td>triggering event</td>
<td>arrival of an external message</td>
</tr>
<tr>
<td>attribute</td>
<td>attribute</td>
</tr>
<tr>
<td>entry</td>
<td>incoming external message</td>
</tr>
<tr>
<td>exit</td>
<td>outgoing external message</td>
</tr>
<tr>
<td>read</td>
<td>referred attributes in an action</td>
</tr>
<tr>
<td>write</td>
<td>updated attributes in an action</td>
</tr>
</tbody>
</table>

To overcome the misinterpretation problem, $\mu ROSE$ is based on a theoretical framework [5] that maps RRRT concepts to COSMIC-FFP concepts. The mapping is based on mathematical definitions of COSMIC-FFP concepts with respect to RRRT concepts. A mathematical approach helps removing ambiguities and eliminates subjective interpretations. A summary of the mapping, implemented through the COSMIC-FFP Component Builder, is given in Table 2 and informally explained in the following subsections.

**Identification of Layers**

In an RRRT model, software is represented as a collection of capsules that might perform at one or several levels of abstraction. Given capsules, it is difficult to automatically determine these levels of abstraction. Therefore, human judgment is required to identify
the capsules forming a layer. It is assumed that the capsules selected by a user correspond to a COSMIC-FFP layer. Once a selection is made, an instance of the class Layer is created with appropriate references to the selected capsules.

Identification of Data Groups

A data group in the COSMIC-FFP class model can be derived from:

- a collection of simple attributes used in a selected capsule (an attribute is simple if its type is an instance of the class Predefined_Class);
- an instance of the class Defined_Class used to specify the type of an attribute that belongs to a selected capsule; or
- an instance of the class Message that participates in a communication between capsules within the boundary and the environment.

Identification of Functional Processes

A functional process in COSMIC-FFP corresponds to a chain of transitions in an RRRRT model. A chain must contain:

- transitions triggered by the same message sent from outside the boundary (called external transitions); and possibly
- the transitions that are recursively triggered by messages generated and exchanged between capsules within the boundary during the execution of an action associated with an external transition.

Thus, an instance of the class Functional_Process in the COSMIC-FFP class model is created from one or several instances of the class Transition in the RRRRT class model. An instance of the class Triggering_Event corresponds to the arrival of a message from outside the boundary in an RRRRT model.
Identification of Data Movements

An instance of the class Entry (Exit) in the COSMIC-FFP class model is created for each instance of the class Message in the RRRT class model that:

- refers to the timer; or
- is sent from the environment (by a capsule within the boundary) and received by a capsule within the boundary (to the environment).

Finally, an instance of the class Read (Write) is created from an instance of the class Data_Group when one of its attributes is used (updated) in an action associated with a transition (including its source and target states as well as its guards) without (with) modification of its value. Of course, the transition should be a part of a functional process.

3.5 Validation and Verification of $\mu_cROSE$

The validation process of $\mu_cROSE$ started early by examining and reviewing the theoretical framework with a COSMIC-FFP expert. The main goal of this process was to uncover errors due to improper formalization or a misinterpretation of COSMIC-FFP measurement rules. Errors were detected during the review process and modifications have been made to ensure correctness and completeness.

During the development phase, unit testing and integration testing were conducted conventionally. For system testing, a set of scenario-based tests for verifying the functionalities and ensuring quality have been systematically derived from the formal specification of the measurement rules. Scenarios have been classified under two categories.

1. Scenarios for testing classes that allow identification of data groups – This includes trivial and nontrivial cases for verifying the identification of the data groups from capsule attributes and messages exchanged between capsules within a boundary and its environment. Trivial cases correspond to the "lower" bound of input domains (e.g., a class without attributes, a class with one attribute).
2. Scenarios for testing classes that allow identification of functional processes and their data movements – This includes the analysis of code embedded in actions associated with states and transitions in order to determine the type of each attribute (referred or modified) involved in the actions. This also includes the identification of chains of transitions associated with a functional process.

**Tab. 3**: Some RRRT models used during validation and verification with their FS

<table>
<thead>
<tr>
<th>RRRT Model</th>
<th>Capsules in the Boundary</th>
<th>Cfsu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory System</td>
<td>Controller</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Controller and Consumer</td>
<td>47</td>
</tr>
<tr>
<td>Manufacturing System</td>
<td>Controller</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Controller and Consumer</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Controller and Producer</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Consumer and Producer</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Controller, Consumer, and Producer</td>
<td>27</td>
</tr>
<tr>
<td>Integrating Data</td>
<td>Receiver</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Receiver and Sender</td>
<td>11</td>
</tr>
</tbody>
</table>

$\muROSE$ has been executed from various RRRT models according to different boundaries. Table 3 presents some results of experiments conducted with $\muROSE$. The first model (Factory System) is the model introduced in Section 3.2. The second model (Manufacturing System) is similar to the first model. The main difference concerns how a transition is triggered. In the first model, each transition of the state machine, that gives the behavior of the capsule Controller, is triggered by a particular message while there are transitions that are triggered by the same message in the second model. Thus, there are more data groups in the first model. This allows to show what are the effects of some small modifications in a model with respect to COSMIC-FFP. The critical points concern the identification of functional processes represented by a sequence of transitions instead of only one transition and the partitioning of internal and external messages with respect to a boundary. The last model (Integrating Data) comes from RRRT. It contains two
capsules: i) the Receiver that receives messages containing data of various types and uses the log service to display the data values and ii) the Sender that sends messages containing data of various types. When the two capsules are selected, the messages sent by the Sender to the Receiver are now considered as internal messages. This reduces the number of entries and the number exits with respect to the case in which only the Receiver is selected.

3.6 Conclusion

The ever-growing need for effective support to software project management has been exemplified with the advent of maturity models that advocate monitoring and controlling development and maintenance activities related to the software process. In order to satisfy this need, measurement components suitable for evaluating both the process and documents must be integrated into existing CASE tools. $\mu_c\text{ROSE}$ constitutes a valid contribution to this effort because it is the first measurement software component to provide a systematic, stepwise, and complete way to measure the FS of RRRR models according to the COSMIC-FFP method. It yields an objective and unobtrusive measurement process with perfect repeatability involving minimal human assistance.

$\mu_c\text{ROSE}$ represents the tangible part of a research project oriented towards the formalization of rules and procedures prescribed by methods related to FS measures not only for RRRR models, but also for B models [6]. The fundamental part of this research project was a theoretical framework that formally expresses FS measure concepts and knowledge accumulated by experts in this area as well as their mapping with formal specification languages (ROOM and B) in a mathematical form. This kind of framework provides the means for better understanding COSMIC-FFP and clarifying some of its obscure points. Although some details of the framework will need to be changed and refined according to the evolution of COSMIC-FFP and intense use of $\mu_c\text{ROSE}$, it turns out that the development of a mathematical framework and class models has
provided a sound foundation for addressing automatic measurement of COSMIC-FFP. Each modification or refinement will be repeatedly transposed into future versions of $\mu_c$ROSE.

Presently, the main limitation of $\mu_c$ROSE concerns the C++ Parser module that performs the code analysis by only examining some basic operators (e.g., assignment, arithmetic and logical operators) of the C++ programming language, because arrays, pointers, and functions may cause serious problems. For instance, the parser does not analyze the code of functions called in actions associated with transitions and states. So, it is assumed that there is no side effect in the measurement when such complex operators appear in actions. These problems are not specific to $\mu_c$ROSE but are encountered in program slicing and static data-flow analysis. According to the present state of art, no general solutions exist, but the use of heuristics allows to obtain non-optimum results when applied in some specific contexts. Heuristics appropriate to static data-flow analysis in the context of COSMIC-FFP measurement should be proposed to improve the C++ Parser module. We hope to address this weakness in a subsequent version of $\mu_c$ROSE.

In addition to the aforementioned issues, we plan to fully integrate $\mu_c$ROSE into the RRRT toolset in such a way that it loses its own identity. We also plan to connect $\mu_c$ROSE to a project database in order to automatically populate it with FS of finished projects. As suggested in the literature, such data are required to build and improve estimation models. Finally, we will investigate how $\mu_c$ROSE could be adapted to measure the FS of incomplete RRRT models. This means that $\mu_c$ROSE should support the measurement of FS both after and throughout the construction of RRRT models. These improvements should significantly reduce the measurement cost for RRRT models and yield a better control and monitoring during their development and maintenance.
Acknowledgements

We would like to express our thanks to Jean-Marc Desharnais for his useful comments during the validation of the theoretical framework implemented in \( \mu_c\text{ROSE} \). Of course, any remaining errors in this framework are ours. We would also like to express our thanks to Stéphane Beaudoin, Ning-Sun Cao, Ly Thanh-Le, Huu-Baoan Trinh, and Hai-Pho Truong for the intense work carried out in programming \( \mu_c\text{ROSE} \).

The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds pour la formation de chercheurs et l'aide à la recherche of Québec (FCAR).
Bibliographie


Conclusion

Ce chapitre évalue la portée de nos travaux sur la formalisation et l'automatisation de la mesure de taille fonctionnelle ainsi que leur extension.

Application des travaux présentés

Plusieurs travaux connexes proposent des interprétations des concepts d'IFPUG-FP et de COSMIC-FFP dans le contexte d'une variété de langages de spécification. Cependant aucun traite la mesure des points de fonction dans le contexte du langage B ou de Rational Rose RealTime (RRRT). Contrairement à ces travaux, qui définissent textuellement ces interprétations, cette thèse propose des définitions mathématiques pour interpréter : i) les concepts d'IFPUG-FP dans le contexte du langage B et ii) les concepts de COSMIC-FFP dans le contexte de RRRT. Ces définitions contribuent à :

- réduire la subjectivité des règles de mesure d'IFPUG-FP et de COSMIC-FFP ;
- identifier et analyser les difficultés rencontrées lors de l'application de ces règles ;
- faciliter et mettre en évidence l'application de ces règles de mesure ;
- éliminer la variance entre les mesures ;
- développer un outil qui supporte la mesure de la taille fonctionnelle du logiciel à partir d'une spécification de besoins écrite en langage B (ou en notation de RRRT) ;
- connecter les méthodes formelles de spécification à la métrique du logiciel, ce qui

79
permet aux organisations, qui utilisent des méthodes formelles, d'exploiter les avantages de la mesure des points de fonction.

L'outil \( \mu_{\text{ROSE}} \), que nous avons développé, implémente la définition formelle de la méthode COSMIC-FFP pour RRR, ce qui permet d'automatiser complètement la mesure. On peut donc obtenir quasi instantanément une mesure uniforme de COSMIC-FFP, sans avoir recours à une expertise externe généralement difficile à obtenir et coûteuse. En conséquence, \( \mu_{\text{ROSE}} \) devrait grandement faciliter l'implémentation d'un processus de mesure de COSMIC-FFP dans les organisations. Les fonctions principales de l'outil \( \mu_{\text{ROSE}} \) sont les suivantes :

- la conversion d'un fichier RTMDL, représentant une spécification produite par l'outil RRRR, en format XML;
- la visualisation du processus de mesure basée sur la méthode COSMIC-FFP;
- l'identification des objets requis pour le processus de mesure et leur extraction à partir d'une spécification produite par l'outil RRRR;
- l'analyse syntaxique de code C++ intégré dans un modèle RRRR;
- la mesure de COSMIC-FFP ainsi que l'agrégation et la documentation des résultats de la mesure.

Un autre gain réalisé par cette thèse est l'identification d'une lacune dans les règles de mesure d'IFPUG-FFP. En effet, la formalisation des règles d'IFPUG-FFP a permis d'identifier des cas qui ne sont pas pris en considération. Nous avons proposé des modifications afin de combler cette lacune et d'assurer la complétude de nos définitions.

Finalement, la définition formelle de la méthode COSMIC-FFP pour RRRR est aussi partiellement applicable à certaines notations orientées objets, notamment UML. En effet, UML et RRRR ont des concepts en commun, par exemple une classe, un attribut d'une classe, etc. Mais il y a aussi des différences entre les deux notations. Par exemple, une capsule est un élément fondamental dans un modèle RRRR, mais il n'existe pas en UML. En conséquence, toutes les règles de mesure pour RRRR, proposées dans le
présent document, qui concernent ces concepts communs sont aussi applicables à un modèle UML. Ceci permet aux utilisateurs d’UML d’avoir une solide connaissance sur comment les règles de mesure de COSMIC-FFP doivent être appliquées sur ces concepts.

Limitations

Concernant l’outil $\mu$ROSE, le module permettant d’analyser du code en C++ intégré dans un modèle RRRT ne traite pas tous les cas possibles. L’algorithme d’analyse du code source traite les cas typiques de référence aux groupes de données dans les systèmes temps réel. Toutefois, il ne traite pas les cas d’alias par pointeur, les références aux attributs d’un objet et les appels de méthode. Pour cela il est possible qu’une partie du code intégré dans un modèle RRRT ne soit pas traité. Dans ce cas, le résultat retourné par ce module représente partiellement le code analysé, car certains “Read” et “Write” ne sont pas identifiés. En conséquence, le nombre de COSMIC-FFP calculé par l’outil pourrait être inférieur à la taille fonctionnelle réelle du logiciel mesuré. Par ailleurs, l’outil $\mu$ROSE ne supporte pas la mesure d’un modèle RRRT incomplet. Il est nécessaire que tous les composants du modèle RRRT soient définis et aient des noms distincts. Finalement, les définitions formelles, permettant d’identifier les groupes de données dans le contexte d’IFPUG-FP, peuvent être appliquées seulement à une spécification B écrite selon un style assez classique. Si une spécification est écrite différemment, il est possible que ces définitions n’identifient pas proprement tous les groupes de données.

Travaux futurs

L’évaluation des résultats obtenus dans cette thèse a été faite de deux manières: révision par des experts du domaine et expérimentation sur des spécifications de taille réduite. Avant d’implémenter nos résultats à une échelle industrielle, il sera nécessaire de mener des expériences sur le terrain avec des systèmes d’envergure industrielle et de
natures diverses. L'ampleur de cet exercice de validation dépasse le cadre normal d'un travail de doctorat, car il nécessite des ressources humaines (expertise en mesure de taille fonctionnelle), logicielles (cas d'étude d'envergure) et financières importantes (embauche des ressources humaines) difficiles à obtenir dans un contexte académique.

Au niveau du prolongement des travaux de développement, nous envisageons les éléments suivants :

- le développement d'un module permettant d'analyser syntaxiquement les cas qui ne sont pas traités par le présent module et de l'intégrer dans $\mu_{ROSE}$ ;
- l'ajout d'une nouvelle fonctionnalité à l'outil $\mu_{ROSE}$ permettant de mesurer la taille fonctionnelle d'un modèle RRRT durant sa construction ;
- le développement d'un outil qui supporte la mesure de la taille fonctionnelle du logiciel à partir d'une spécification écrite en B en se basant sur les définitions formelles présentées dans cette thèse.

Le développement d'un outil supportant la mesure de la taille fonctionnelle dans le contexte d'IFPUG-FP nécessite d'abord de faire d'autres activités de recherche afin de résoudre les problèmes suivants :

- groupe de données : la généralisation de l'application de la définition formelle qui permet l'identification des groupes de données indépendamment du style de la spécification ;
- sous-groupe de données ou "Record Element Types" (RET) : le manuel d'IFPUG-FP ne fournit pas de règles permettant de déterminer les sous-groupes de données dans un GDI ou un GDE. Nous avons proposé une définition qui reste à être validée ;
- processus élémentaire : il n'y a pas de règles dans le manuel d'IFPUG-FP [24] permettant de classifier le type de certaines opérations en B. Nous avons fait des suggestions pour s'assurer de la complétude de nos définitions. Mais des validations seront nécessaires avec des experts afin d'évaluer l'exactitude de nos interprétations.
D'autres activités de recherche pourraient être intéressantes en prenant un certain nombre de cas d'études (des spécifications écrites en B ou RRRT), en mesurant la taille fonctionnelle de chaque cas, en comparant la variation entre des spécifications alternatives et en déterminant les facteurs qui ont un effet sur le coût de développement et la qualité d'une spécification par rapport à une autre.
Bibliographie


87
Glossaire (Glossary)

The following terms have been used throughout this document. Note that, some terms have been adopted as defined in the COSMIC-FFP measurement manual, IFPUG counting practice manual, or ISO (International Standard Organization) documents.

- CFSU: COSMIC Functional Size Unit;
- COSMIC-FFP: Common Software Measurement International Consortium of Full Function Points;
- CPM: Counting Practice Manual of IFPUG;
- DET: Data Element Types of IFPUG;
- E: Entry data movement in the COSMIC-FFP context;
- EI: External Inputs of IFPUG;
- EIF: External Interface Files of IFPUG;
- EO: External Outputs of IFPUG;
- EQ: External Inquiries of IFPUG;
- FP: Function Points;
- Formal Method: is a method for specifying, designing, and developing a software based on mathematical notations;
- Formalization and Formalizing: is providing a definition using mathematical notations;
- FSM: Functional Size Measurement;
- FUR: Functional User Requirements;
- IFPUG or IFPUG-FP: International Function Points Users Group;
- ILF: Internal Logical Files of IFPUG;
- LOC: Lines Of Code;
- $\mu$ROSE: Measurement of COSMIC-FFP for Rational Rose Real-Time models;
- OMT: Object Modeling Technique;
- OO: Object Oriented;
- R: Read data movement in the COSMIC-FFP context;
- RET: Record Element Types of IFPUG;
- RRRT: Rational Rose Real-Time;
- ROOM: Real-Time Object Oriented Modeling;
- UCD: Use Case Diagram from the Unified Modeling Language (UML);
- UFP: Unadjusted Function Points;
- VAF: Value Adjustment Factors;
- W: Write data movement in the COSMIC-FFP context;
- X: Exit data movement in the COSMIC-FFP context;
Annexe A

Contexte (Background)

A.1  Functional Size Measurement

The software industry is one of the greatest industrial phenomena of the history. In the span of a few decades, it has become an essential component of everyday life in modern society. Software plays an increasing role in many domains (commerce, industry, government, health, transport, telecommunication, etc.). Therefore, software must fulfill an expected quality standard. Software measurement is a prerequisite for managing and controlling software development in order to achieve a predictable quality level and controlling development cost.

In this section, we provide an overview of the role of measurement in software engineering and the main benefits of its use.

A.1.1  Usage of Size Measurement in Software Engineering

Measurement is considered as one of the essential features of a real engineering approach to software development, maintenance, and operation. It has been highlighted in the IEEE definition of software engineering [23] as follows: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of
software; that is, the application of engineering to software. To ensure that a software is quantifiable, a measurement process is required. It has been realized that an integration of measurement activities into the software development process could be of great benefit for many uses:

- it allows project managers to estimate cost and to control projects;
- it allows software engineers to evaluate the suitability and efficiency of several methods and tools used during software development;
- it helps contract organizations to select a supplier, to control the supplier’s performance, and to track actual performance versus established plans and objectives;
- it helps investors for analyzing the cost/benefit of future projects;
- it can be used to improve, compare, and benchmark performance when comparing performances across projects using different technologies.

Hence, measuring software is an important goal for any organization developing or using software.

The first step of a measurement process is the identification of measurable attributes. These attributes may belong to products, process, or resources, depending on the measurement goals. In [15], Fenton has defined two kinds of attributes: internal and external. Internal attributes of products, process, or resources are those that can be measured by statically examining the product, process, or resources on its own and independently of its behavior, for instance size (product), effort (process), number of requirements changes (process), etc. External attributes of products, process, or resources are those that can be measured by dynamically examining the behavior of the product, process, or resources and how they are related to their environment, for instance quality (product), cost (process), productivity (resources), etc. Software size is one of the main attributes in a measurement process. It is used to compute several internal and external attributes (as quality report, productivity report, cost, plan effort, defect removal efficiency, etc.) that are very helpful to improve software development process and software quality.
A.1.2 Measuring Functional Size Versus Lines of Code

Software developers tried to use lines of source code (LOC) as a unit of measure for software size. A consistent estimation of LOC can be performed at implementation and testing stages. However, there are difficulties in using LOC, such as how to estimate it during requirements analysis or design, how to convert LOC in order to compare systems written in different languages, and which line of code should be counted (physical or logical) [30]. Furthermore, in [29] an analysis of the mathematical problems and paradoxes associated with LOC measures has been reported. It proved that LOC measures were incapable of measuring productivity in the economic sense.

One of the measurement alternatives to LOC measures is functional software size measures. The functional software size, representing what users care about (software services to be provided from the user's viewpoint), could be used for measuring productivity according to the economic definition of productivity. The function point method of Albrecht [3] was the first to measure the functional software size. The main goal of the function point method was to eliminate LOC paradoxical problems and deal with the software services. In the software industry, LOC measures are sidestepped by functional software size measures for several management activities. Functional size measurement methods, function point methods in particular, have gained a considerable popularity in the industry. More than 1 200 organizations are using function point methods world-wide.

A.2 Function Points for Functional Size Measurement

In this section, we briefly introduce function points techniques and their main characteristics. We also describe the measurement processes of IFPUG-FP and COSMIC-FFP.
A Brief History of Function Point Techniques

Function points [3, 13] is one of the prominent methods to measure functional size. Albrecht [3] proposed the first definition of this measure. Its users are now structured as a user group, the International Function Point User Group (IFPUG), which provides a Counting Practice Manual (CPM) [24].

Another version of function points, Mark II Function Point Analysis, has been proposed by Symons [44], and it is widely used in the United Kingdom. Recently, function points were extended by the COSMIC group (Common Software Measurement International Consortium) to more adequately address real-time software. The resulting measure is called COSMIC Full Function Points (COSMIC-FFP) [1]. This method has rapidly gained international support and acceptance as a new functional size measurement method. Note that this thesis only focus on COSMIC-FFP and IFPUG-FP.

A.2.1 The Characteristics of Function Point Techniques

The main characteristics of function point techniques are as follows: they measure the size of the software services provided to the users (or the system’s environment); they do not depend on a particular functional specification notation; they do not depend on the implementation of the system.

An international benchmarking center has been established in Australia [27]. It helps member organizations to provide and retrieve data from its database in a controlled manner, ensuring quality and anonymity of data. It includes IFPUG-FP and COSMIC-FFP measures.

A.2.2 Function Point Measurement Process

Function point techniques (COSMIC-FFP and IFPUG-FP in particular) are based on objects comprised in a given model. The model may be produced at the capture of user
requirements, specification, or design phases. Function points can also be estimated from the number of LOC through backfiring (conversion of LOC in function points).

**IFPUG-FP:** After determining the boundary of the application, measuring the number of function points is a four steps process:

1. identifying elementary processes, data groups, data element types (DET), and record element types (RET);
2. classifying elementary processes (there are three types: external inputs (EI), external outputs (EO), and external inquiries (EQ)) and data groups (there are two types: internal logical files (ILF) and external interfaces files (EIF));
3. assigning a numerical value for each EI, EO, EQ, ILF, and EIF (representing the weight of each component);
4. computing the sum of all the weights which represents the functional software size.

**COSMIC-FFP:** In the Measurement Manual (MM) of COSMIC-FFP [1], there are eight steps for measuring the number of full function points:

1. identifying layers of the software to measure (a layer is a set of software components identified at the same level of functional abstraction);
2. identifying the boundary of each layer;
3. identifying functional processes within the boundary;
4. identifying data groups within the boundary;
5. identifying data movements within each functional process;
6. classifying each data movement (there are four types: entry (E), exit (X), read (R), and write (W));
7. assigning a numerical value for each data movement of each functional process and computing the sum of all the values which represents the size of the functional process in question;
8. computing the sum of the sizes assigned to all the functional processes which represents the functional size of the software.

A.3 Problems with IFPUG-FP and COSMIC-FFP

Functional measurement methods have their own sources of ambiguity, IFPUG-FP and COSMIC-FFP in particular. The measurement rules are only given in natural language and may easily be misused or misinterpreted depending on the measurer's experience using them. In addition, the lack of a direct representation of some specification concepts in the context of a measurement method and/or the subjectivity of the measurement rules are the main causes of this misuse (therefore experts are always needed, but they are not easily available). For instance, a DET and a RET in IFPUG-FP do not directly map to concepts of some specification languages like B [2]. The weights of transactions and files are based on detailed rules requiring the determination of DETs and RETs and file types referenced (FTR). But, in B this information is not easy to find. Analysts would have to deepen their investigations in order to identify and collect them for the sake of measurement. In COSMIC-FFP, some concepts of a formal modeling language like Rational Rose RealTime (RRRT) are broader than COSMIC-FFP concepts, (e.g., a transition in RRRT versus a functional process in COSMIC-FFP). Unless the application of measurement rules is independent of the measurer, it will always produce a variance between measures when different persons apply the measurement rules to the same specification at different times. In IFPUG-FP counts, studies noted a difference varying from 11% to 30% [18, 28, 33, 34]. Another problem is that the estimated size does not meet the actual. The difference between the IFPUG-FP count of the initial specification and IFPUG-FP count of the resulting system varies between 400% and 2000% [34].

On the other hand, for many organizations no size measures have been produced to

---

1. These figures hold when the count boundary is well-defined. Larger variations are observed when boundaries are not clearly delineated.
evaluate projects in terms of quality and productivity and to calibrate their own estimation models. Some of the major reasons given by organizations not implementing measurement systems are: the measurement cost is quite expensive and the duration to manually obtain these measures is not short. Furthermore, in most cases requirements are incomplete and ambiguous. Therefore, re-computation of the software size will be necessary after each change of requirements. Consequently, the cost of manually measuring the functional software size will considerably increase. Organizations with successful measurement systems have found an inexpensive means of gathering data through the use of automated tools [45]. There are quite a number of tools that allow to estimate effort and schedule, build plans, and statistically analyze data in an estimation model for software development projects. One of the basic inputs of these tools is the software size. Note that there are few tools supporting an automated measurement of IFPUG-FP from software requirements and there is no tool for automated measurement of COSMIC-FFP [37].

A.4 Contributions

In this section, we introduce the main contributions of our work and the major factors that lead us to select formal specification languages like B and RRRT for formalizing the measurement rules of IFPUG-FP and COSMIC-FFP, respectively.

A.4.1 General Contributions

This thesis removes the subjectivity from the measurement rules of IFPUG-FP and COSMIC-FFP. It proposes theoretical frameworks for mapping IFPUG-FP concepts to B concepts and COSMIC-FFP concepts to RRRT concepts using mathematical notations. That is, there is one and only one interpretation for each formal measurement rule. This allows to always obtain the same number of function points whatever the measurer experience is (expert or not). The natural languages that humans use to communicate, such as English or French, are full of ambiguities. A sentence can often have two or more
different valid meanings. A formula does not allow such ambiguities to exist, because its semantic is well defined. Hence, our formal measurement rules eliminate the variance between measures and can be used as the basis for an automated measurement.

During the formalization, we have found that IFPUG-FP was incomplete, in the sense that some transactions did not fall into any category defined in IFPUG-FP. We have proposed modifications to ensure completeness.

Several researchers have proposed translation rules from object-oriented notations like OMT and UML to the B notation [14, 38]. Consequently, we could use the mapping rules to compute IFPUG-FP not only for a B specification but also for a specification written in OMT or UML.

The formal definition of COSMIC-FFP is implemented in the $\mu$ROSE tool. $\mu$ROSE can read an RRRT model and automatically compute the COSMIC-FFP size of the selected layers. It is the first tool that supports automated measurement of COSMIC-FFP directly from a software description. Having a tool for measuring the functional size of a software is an inexpensive mean for gathering data about software size. This allows organizations to easily measure software size, and maintain data about their previous projects which are essential to build their estimation models for future projects. A particular advantage of this tool is that repeated measures can be easily taken after each change of requirements. This allows to track size variations along the software life cycle.

A.4.2 Why B and RRRT?

The adequacy of several families of specification languages for formalizing IFPUG-FP and COSMIC-FFP has been evaluated [17]. A summary of this evaluation is presented in Section 1.8. It was found that the B and RRRT notations [2, 41] were the most appropriate languages for formalizing measurement rules of IFPUG-FP and COSMIC-FFP, respectively.
A.5 Related Work and Comparison

In this section we introduce a brief literature review on measurement tools, IFPUG-FP counting tools in particular, and approaches that have been proposed to map IFPUG-FP and COSMIC-FFP measurement rules to some object-oriented methods. A comparison between these approaches and our approach is also presented.

A.5.1 IFPUG-FP and COSMIC-FFP Measuring Tools

In [15], Fenton and Pfleeger identify five types of measurement tools: planning tools, forecast tools, data collection tools, data analysis tools, and data storage tools. The planning tools allow to build plans for software development projects. The forecast tools allow to estimate effort and schedule. The data collection tools allow to measure the size and structure of programs using static and dynamic code analyzers. The data analysis tools allow to analyze the collected data using statistical tools. The data storage tools allow to store the collected data in a database management system (DBMS) which manipulates stored data in a simple manner.

IFPUG-FP measurement is implemented in many forecast tools. Most implementations of IFPUG-FP are restricted to an automatic calculation of the sum of function points. The measurement tools read the numbers of files, processes, and attributes that are manually determined and produce the functional software size. In other words, most works on automation of IFPUG-FP counting do not calculate the functional software size directly from a software requirements specification. To the extent of our knowledge, there are eight tools [37] claiming to automate the identification of the IFPUG-FP standard parameters directly from a software requirements specification (no document has been published about the algorithms implemented in these tools) and there is no tool supporting COSMIC-FFP measures.
A.5.2 Mapping IFPUG-FP and COSMIC-FFP to OO Methods

Most approaches that have been proposed for adapting IFPUG-FP or COSMIC-FFP to object-oriented methods do not provide formal rules for measuring the number of function points, as we provide in this thesis. These approaches do not totally eliminate the human judgment factor from the measurement process. As consequence, variances may still exist and it will be difficult to develop a tool for measuring the number of IFPUG-FP or COSMIC-FFP based on these approaches. For instance, Withmire in [46] makes the following statement for mapping the inheritance concept to the IFPUG-FP context: "if the generalization is truly part of the application domain, it is counted a separate data group. If the generalization was built for the ease of modeling we count the general class with each specialized class". Determining if a generalization was built for the ease of modeling depends on the measurer's knowledge of the system. Two measurers with different knowledges about the system will produce different measures. Another approach reported in [16] states: "sub-objects of a concrete object are candidates for a data group or for a sub-group of that object. If these sub-objects are not counted as data groups themselves, they are optional sub-groups of the data group related to their super-object". In [16], the authors do not specify when a sub-object must be counted as a data group or as sub-group; that decision is left to the measurer.

None of the proposed approaches provides mapping rules for measuring IFPUG-FP and COSMIC-FFP from B and RRRT specifications, respectively. There is only one approach that has been published on COSMIC-FFP in the context of the UML notation [7]. There are many concepts which are used in RRRT notation but which are not in UML. For instance, capsules are the basic modeling element in RRRT model, but they do not exist in UML. Therefore, we cannot measure the number of COSMIC-FFP from RRRT model based on the approach of [7]. Hence, mapping rules between RRRT and COSMIC-FFP concepts are required.
It should be noted that the related works discussed in this section are limited to the approaches based on IFPUG-FP and COSMIC-FFP. Approaches that define variants of IFPUG have not been considered.

A.6 Methodology

This section briefly outlines the procedure used for formalizing the IFPUG-FP and COSMIC-FFP measurement rules. It is composed of the following steps.

- **Step 1**: evaluate the adequacy of several specification languages.
- **Step 2**: formalize IFPUG-FP counting rules for the B language.
- **Step 3**: validate the suitability of the formal rules defined in step 2.
- **Step 4**: formalize COSMIC-FFP measurement rules for the RRRT notation.
- **Step 5**: validate the suitability of the formal rules defined in Step 4 using experts.
- **Step 6**: develop a tool (μROSE) to automate the measurement of COSMIC-FFP for RRRT. The development process was composed of the following activities.
  - identify what data are to be processed, what functions are required, and what validation criteria are required based on the formal rules defined in step 4;
  - create class diagrams (attributes, methods, and relationships between classes) and establish interface;
  - define the tool's main modules, procedure details, different components, and interface's prototype;
  - translate the design into a collection of programs written in the Java language;
  - perform testing to detect errors and ensure that the tool produces the required results.
A.7 Evaluation of Specification Languages for Formalizing IFPUG-FP and COSMIC-FFP

The main goal for which IFPUG-FP and COSMIC-FFP were proposed is for enabling a measurement of software services from the user's viewpoint. To measure IFPUG-FP or COSMIC-FFP, it is essential that a software document contains details allowing to identify separately those services. Furthermore, this document should provide information allowing the measurer to identify and distinguish data groups that might be entered, modified, referred, and produced during the processing of each service. Therefore, an evaluation of the adequacy of a specification language for the formalization of IFPUG-FP and COSMIC-FFP measurement rules and to automate the measurement process should be based on these aspects.

In the rest of this section, we briefly evaluate the adequacy of several families of specification languages. More details are given in [17].

Trace-based specifications are not adequate for the formalization of IFPUG-FP or COSMIC-FFP measurement rules. Because they abstract from states, it is not possible to identify data groups. Consequently, the calculation of weights for services cannot be fully formalized, because data groups are required.

Algebraic specifications are not adequate for formalizing IFPUG-FP or COSMIC-FFP measurement rules. Operations are not always elementary processes (in IFPUG-FP) or functional processes (in COSMIC-FFP). Some of them may be auxiliary operations used in the definition of operations required by the user. We do not see any systematic way of distinguishing them. There is a similar problem with data groups. Sorts may represent data groups or data attributes, without any systematic way of distinguishing between them.

Because process algebra do not distinguish between inputs and outputs, it seems difficult to identify software services. There is a similar problem with data groups. In process
algebra, event parameters may either be input parameters, output parameters or state information. Hence, it is difficult to identify data and weigh them. Because of these difficulties, process algebra are not adequate for formalizing IFPUG-FP or COSMIC-FFP measurement rules.

There is a natural mapping between IFPUG-FP/COSMIC-FFP and model-based specifications, but not all the notations are easy to formalize or appropriate. For instance, semi-formal object-oriented notations are difficult to formalize (e.g., UML and OMT). The main difficulty with these notations is that behavior diagrams are not precise enough to properly identify data that might be modified and/or referred during service execution. In general, these details are described in natural language. In the Z notation, it is not always obvious to distinguish between a modified variable and a referred variable that are used in a predicate.

A model-based specification written in the B language is defined as a set of state machines. Each machine is composed of data and operations which represent services of a software. A simple analogy with information systems is to consider that machines are specifications of modules interacting with the users. An operation corresponds to an elementary process in IFPUG-FP and a functional process in COSMIC-FFP. Signatures of an operation may be used to identify the software inputs and outputs. Data that are modified and/or referred during the processing of an operation are described in the operation body. Furthermore, the use of elementary substitution in the body of operation facilitates the identification of these data and their classification.

In industry, several systems, in different domains, are successfully developed by a number of industrial organizations using the B method, mainly in United Kingdom and France [6]. Furthermore, the B notation is very close to a programming language, which makes it easy to use and learn. Although it is sufficiently abstract to support good specification practices.

In formal object-oriented notations, there is also a natural mapping between Rational
Rose Real-Time (RRRT) concepts and COSMIC-FFP concepts. An RRRT model is defined as a set of capsules. The behavior of a capsule is defined by a state machine. Each machine is composed of a set of transitions between states. An action is defined as a set of executable instructions written in a programming language. An action may be associated to a transition or a state. The structure of a transition allows to easily identify and distinguish the different types of data processing, since a transition is already decomposed into actions. A chain of transitions triggered by the arrival of an external event corresponds to a functional process in COSMIC-FFP. Messages exchanged between the software and its environment may be used to identify the software inputs and outputs. Data that are modified and/or referred during the processing of an action are described in the action body. Furthermore, the use of executable instructions in the body of action facilitates the identification of those data and their classification.

In addition, COSMIC-FFP was originally proposed for measuring the functional size of real-time software. RRRT was defined for specifying, designing, and developing real-time software. All aspects of a real-time software have been already taken into account and represented in COSMIC-FFP and RRRT. This facilitates the mapping between them.

Based on these investigations, it was found appropriate to formalize the COSMIC-FFP measurement rules for the RRRT notation and IFPUG-FP counting rules for the B notation.
Annexe B

Des définitions formelles de la mesure d'IFPUG-FP pour le langage B (Formal Rules for Measuring IFPUG-FP from B Specifications)

In this chapter, we propose a formalization of the IFPUG definition of function points using the B notation. In order to formalize this definition, we need to first define some elementary concepts of the function points measure. We map these concepts to B definitions.

Note that all our definitions are based on a syntactic analysis of a B specification. Almost no semantic reasoning is required to compute function points from a B specification. Some minor aspects would be better evaluated using a semantic viewpoint, but they would be more difficult to compute because a proof would be required.
B.1 Definition of Elementary Concepts

B.1.1 Elementary Process

An elementary process in the function point measure is the smallest unit of function which is recognized by (visible to) the user. An elementary process corresponds to an operation in the B notation.

The application boundary of a function points count is a set of elementary processes and data components that are modified or referenced by these elementary processes. In B, given a set of operations, it is possible to derive the set of data components which are modified or referenced by these operations. Therefore, we may simply define the boundary as a set of operations that are recognized by the user. We denote the boundary by $F$.

B.1.2 Logical Data Group and Data Element Type

Typical examples of a logical data group are a file or a relation in a relational database. The best way to map this concept in the B notation is to use abstract sets. In the B notation, sets, constants, and variables of a machine represent system data. An abstract set $AS$ is typically used to represent a set of logically related data. The abstract set represents the set of all possible values that the system may use during its execution. The actual set of values used during the execution is stored in a machine variable $S$ which is a subset of $AS$.

A data element type (DET) is an attribute of a logical data group. In B, a DET corresponds to either a variable $S$ which is a subset of an abstract set $AS$ (i.e. $S \subseteq AS$) or a variable $f$ which is a function from $S$ to some other set $T$ denoted by $f \in S \leftrightarrow T$. In this case, $T$ denotes the type of the attribute.

We may now provide formal definitions for logical data group and DET. Let $AS$ be the set of all abstract sets defined in a specification $A$. Let $\mathcal{V}$ be the set of all variables defined in $A$. Let $\mathcal{FV}$ be the set of all variables defined as functions in $A$. We denote by
$DG$ the set of logical data groups in specification $A$. It is defined as follows.

$$DG \triangleq \{ S \mid S \in \mathcal{V} \land \exists AS, f, T : AS \in AS \land S \subseteq AS \land f \in S \rightarrow T \}.$$ 

We denote by $DET(S)$ the set of DETs of a logical data group $S$. It is defined as follows.

$$DET(S) \triangleq \{ S \} \cup \{ f \mid f \in \mathcal{FV} \land \exists T : f \in S \rightarrow T \}.$$ 

As an elementary example, suppose that we want to count function points for the specification of a library system written in B [11]. In this example, we have one machine, "Machine Library". We suppose that all the operations defined in the machine are recognized by the user.

To illustrate how we establish a data group, we use the part listed below of the library specification. There are two abstract sets, $AS$, defined in the SETS clause which are the following: "BOOK" and "TITLE". In the VARIABLES clause, there are two machine variables: "book" and "title". Finally, in the INVARIANT clause, there are the predicates on the variables "book" and "title" which must be preserved by operations (i.e., "book", $S$, is defined as a subset of "BOOK" and "title", $f$, is defined as a total function).

SETS

BOOK;
TITLE;

VARIABLES

book,
title,

INVARIANT

book $\subseteq$ BOOK $\land$

$\quad$ title $\in$ book $\rightarrow$ TITLE

By applying the formal definition for identifying a data group, we have identified the subset "book" as a data group because there is a variable "title" which uses "book"
(book \subseteq \text{BOOK}) as a function domain. Formally we can write "book" $\in DG$. Further, we have identified "book" and "title" as two DETs of "book" using the formal definition "DET(S)".

Note that in the rest of this paper we will use the same library specification to illustrate the application of the formal rules.

B.1.3 Record Element Types (RET)

In the CPM of IFPUG, the DETs of a data group are partitioned into RETs (record element types). Furthermore, RETs are classified as either mandatory or optional according to the following definitions:

- mandatory: each set of DETs that an elementary process needs to provide for adding an element in the data group;
- optional: each set of DETs that an elementary process possibly uses during the processing.

We may formalize these definitions using the concepts of partial functions and total functions. Recall that the DETs of a data group are defined as machine variables which are either a subset of an abstract set or a function from a subset of an abstract set to some typing set. It is possible to specify constraints on functions in B. In particular, it is possible to state that a function is total or partial. A function $f$ with domain $D$ and range $E$ is total iff it is defined for every element of its domain $D$. A function is partial if it may be undefined for some elements of $D$. In B, a total function is noted as $D \rightarrow E$, and a partial function is noted $D \rightarrow E$. Other decoration symbols may be added to the arrow to specify other constraints (like injectivity and surjectivity).

Recall from our introduction on B that operations must preserve the invariant. Hence, if an operation creates a new element of a data group, it must specify the value for DETs which are defined using total functions. Therefore, these DETs are mandatory. DETs defined with partial functions may or may not be specified. Therefore, these DETs are
optional.

Following these observations, it is possible to classify a DET as either optional or mandatory using the INVARnANT clause of a B specification. However, we do not know how to partition the DETs into RETs using only the specification text itself. It seems that this choice must be made by the user (which may be a subjective choice). A compromise is to define by default one RET which contains all the mandatory RETs of a data group and one RET which contains all the optional DETs of the data group.

Formally, let $\mathcal{PFV}$ be the set of all machine variables defined as partial functions and $\mathcal{TFV}$ be the set of all machine variables defined as total functions. Let $RET_m(S)$ be the mandatory RET of a data group $S$ and $RET_o(S)$ be the optional RET of a data group $S$. They are defined as follows.

$$RET_m(S) \triangleq \{S\} \cup \{v \mid v \in DET(S) \land v \in \mathcal{TFV}\}$$

$$RET_o(S) \triangleq \{v \mid v \in DET(S) \land v \in \mathcal{PFV}\}$$

### B.2 Classifying Components

In the function points definition, there are five types of component to count: internal logical files, external logical files, external inputs, external outputs and external inquiries. In this section, we provide a formal definition of the IFPUG rules [24] to identify these component types.

#### B.2.1 Internal Logical Files (ILF)

According to the IFPUG CPM, an internal logical file is a set of data or control information, logically related, identifiable by the user and maintained by an elementary process within the boundary of the application under evaluation.
We may formalize this definition in B as follows. We denote by $ILF$ the set of internal logical files of a boundary $F$. We define $ILF$ as the set of data groups which are maintained by at least one of operation of $F$. Our formal definition is the following.

$$ILF \triangleq \{ S \mid S \in DG \land \exists o : o \in F \land \text{maintains}(o, S) \}$$

where $\text{maintains}(o, S)$ denotes that operation $o$ maintains data group $S$. The rest of this section provides a definition of predicate $\text{maintains}$.

In B, an operation is specified using elementary substitutions. An elementary substitution is similar to an assignment statement in a programming language. It is a statement $b$ of the form $v := E$, where $v$ is a machine variable or an output parameter, and $E$ is an expression. We refer to $v$ as the left-hand side of $E$, noted $LHS(b)$, and $E$ as the right-hand side of $E$, noted $RHS(b)$.

An operation $o$ is said to maintain a data group $S$ if and only if there exists an elementary substitution $b$ in $o$ such that the left-hand side of $b$ is a DET of $S$. Predicate $\text{maintains}(o, S)$ is defined as follows. Let $B_o$ be the set of elementary substitutions appearing in the definition of operation $o$.

$$\text{maintains}(o, S) \iff \exists b, v : b \in B_o \land v \in DET(S) \land v = LHS(b)$$

To illustrate the application of the ILF definition, we use operation "Acquire", as listed below, from the library specification. The purpose of this operation is to introduce a new book in the library system. This operation maintains the value of the data group "book", $\text{maintains(\text{Acquire, book})}$, because it contains, at least, an elementary substitution whose left-hand side refers to a DET of "book". For instance, the left-hand side of the following elementary substitution is variable "title" (identified as a DET of "book"):

$$\text{title(bbookid)} := \text{btitle}$$

So, $\text{maintains(\text{Acquire, book})}$ is true, "Acquire" $\in F$, and recall that "book" $\in DG$. Consequently, "book" is an ILF.
Acquire(bbookid,btitle,bauthor,bdate) = 

PRE 
  book ≠ BOOK ∧ 
  bbookid ∈ BOOK - book ∧ 
  btitle ∈ TITLE ∧ 
  bauthor ∈ AUTHOR ∧ 
  bdate ∈ DATE 

THEN 
  book := book ∪ {bbookid} || 
  title(bbookid) := btitle || 
  author(bbookid) := bauthor || 
  acquisitionDate(bbookid) := bdate || 
  loanStatus(bbookid) := InLibrary || 
  loanCount(bbookid) := 0 

END;

B.2.2 External Interface Files (EIF)

An external interface file is similar to an internal logical file, except that is maintained outside the boundary. According to the IFPUG CPM, an EIF is data or control information, logically related, referenced by the application under evaluation but maintained by an elementary process within the boundary of another application.

Our formal definition in B of an EIF is very similar to the definition of an ILF. We define an EIF as a data group which is referenced by an operation of F, but which is not maintained by any operation of F. Let EIF be the set of external interface files of
a specification. Set \( EIF \) is defined as follows.

\[
EIF \triangleq \{ S \mid S \in DG \land \exists o : o \in F \land \text{consults}(o, S) \land \\
\neg (\exists o : o \in F \land \text{maintains}(o, S)) \}
\]

where \( \text{maintains}(o, S) \) denotes that operation \( o \) maintains data group \( S \) and \( \text{consults}(o, S) \) denotes that operation \( o \) consults data group \( S \) without changes.

An operation \( o \) is said to consult a data group \( S \) if and only if \( o \) does not maintain \( S \) and there exists a \( \text{DET} \) \( v \) of \( S \) such that \( v \) occurs in the definition of \( o \). Predicate \( \text{consults}(o, S) \) is defined as follows. Let \( \text{occurs}(v, x) \) be that \( \text{DET} \) \( v \) appears in \( x \) (\( x \) might be an operation, expression, or set of conditions).

\[
\text{consult}(o, S) \leftrightarrow \neg\text{maintains}(o, S) \land \exists v : v \in \text{DET}(S) \land \text{occurs}(v, o)
\]

In the case of the library system, there is no EIF because all the identified data groups are maintained by at least one operation from inside the application boundary.

### B.2.3 External Inputs (EI)

In the IFPUG CPM, an external input is an elementary process that maintains data or control information. Using the concepts defined so far, it is quite straightforward to characterize an external input in \( B \). Let \( EI \) be the set of external inputs of a specification. Let \( \text{inParam}(o, p) \) be that \( p \) is an input parameter of operation \( o \) and let \( \text{outParam}(o, p) \) denote that \( p \) is an output parameter of operation \( o \). Set \( EI \) is defined as follows.

\[
EI \triangleq \{ o \mid o \in F \land (\exists p, S : \text{inParam}(o, p) \land S \in DG \land \\
\text{maintains}(o, S)) \land \neg (\exists p' : \text{outParam}(o, p')) \}
\]

For instance, operation “Acquire” has input parameters, no output parameters, and maintains “book” identified as an ILF. Using the formal definition \( EI \), “Acquire” is recognized as an EI.
B.2.4 External Outputs (EO)

In the IFPUG CPM, an external output is an elementary process that generates data or control information sent outside the application boundary. In B, it corresponds to an operation that has an output parameter to which derived data might be assigned. It is very important to determine if an operation has a derived data or not because it might change the type of operation. For instance, suppose that an operation does not maintain data, has input parameters and output parameters. If there is at least one output parameter considered as a derived data, this operation is recognized as an EO. Otherwise, this operation is recognized as an EQ. The difficulty here is to formalized the concept of derived data, the CPM fails to give a precise definition of what a derived data is. We say that a term $E$ of a substitution is derived if one of its elementary components is not a DET or a constant. The elementary components of a term $E$ are the operands of the data structure operations composing $E$. The data structures available in B are sets, tuples, relations, functions, sequences, and trees. These data structures are used to specify outputs like reports. For instance, a report listing sales by region is a sequence of tuples, each tuple representing a report line.

$$[(\text{region}_1 \rightarrow \text{amount}_1), \ldots , (\text{region}_n \rightarrow \text{amount}_n)]$$

Operation $[\ldots ]$ is sequence constructor. Operation "$\rightarrow$" is a tuple constructor. The elementary components of this report are $\text{region}_i$ and $\text{amount}_i$. In the expression specifying the report, if $\text{amount}_i$ is not a DET or a constant, then we say that this report contains derived data. For instance, if the $\text{amount}_i$ is the sum of sales by salesman in a region, then the report contains derived data.

A detailed description of the formalization of the concept derived data is outside the scope of this paper. The reader may consult [11] for a more detailed description of this formalization. For the sake of simplicity and concision, let $\text{derives}(o,p)$ denotes that operation $o$ has a derived data $p$. Let $EO$ be the set of external outputs of a specification.
Set $EO$ is defined as follows.

$$EO \triangleq \{ o \mid o \in F \land \exists p : outParam(o,p) \land \left[ derives(o,p) \lor \neg(\exists q : inParam(o,q)) \lor \left( \exists S : S \in ILF \land maintains(o,S) \right) \right] \}$$

To illustrate the application of this definition we use operation "ListBooksAuthor" as listed below. The purpose of this operation is to list the books of a particular author. This operation contains derived data "\text{card(dom(bookseq))}", and has input parameter "bauthor". Using the formal definition $EO$, "ListBooksAuthor" is recognized as an EO.

authorlist $\leftarrow$ ListBooksAuthor(bauthor) =

**PRE**

\hspace{1em} bauthor $\in$ AUTHOR

**THEN**

\hspace{1em} ANY bookseq WHERE

\hspace{2em} bookseq $\in$ seq(TITLE×NAT) $\land$

\hspace{2em} ran(bookseq) =

\hspace{3em} \{tt,nn |$

\hspace{4em} tt,nn$ $\in$ TITLE×NAT $\land$

\hspace{5em} $\exists$ bbookid . (bbookid $\in$ author$^{-1}$[\{bauthor\}] $\land$

\hspace{6em} tt = title(bbookid)$\land$

\hspace{7em} nn = loanCount(bbookid)) $\land$

\hspace{8em} sorted(bookseq,prj1(TITLE,NAT))

\hspace{2em} THEN

\hspace{3em} authorlist := \{(bookseq $\mapsto$ card(dom(bookseq)))\}

**END**

END;
B.2.5 External Inquiries (EQ)

According to IFPUG's CPM, an external inquiry is an elementary process composed of an input/output combination that extracts and displays data. The displayed data must not be derived (otherwise, the process is counted as an external output). Now that we have defined the concept of derived data in the previous section, it is straightforward to define \( EQ \), the set of external inquiries.

\[
EQ \triangleq \{ o \mid o \in F \land (\exists p,q : outParam(o,p) \land inParam(o,q)) \land
\neg(\exists S : S \in ILF \land maintains(o,S)) \land \neg(\exists p : derives(o,p))\}
\]

To illustrate the application of this definition we use operation "ListBookDescription" as listed below. The purpose of this operation is to list the description of a particular book. This operation has input parameter "bbookid", output parameters but no derived data "bbookid, title(bbookid), author(bbookid)" and does not maintain any data group. Using the formal definition \( EQ \), "ListBookDescription" is recognized as an EQ.

```
bookdesc, response ← ListBookDescription(bbookid) =

PRE
    bbookid ∈ BOOK

THEN
    IF
        bbookid ∈ book
    THEN
        bookdesc := \{(bbookid ↦ (title(bbookid) ↦ author(bbookid)))\} ||
        response := BookFound
    ELSE
        bookdesc := \{ \} ||
        response := BookNotFound
```
B.2.6 Verifying the Completeness of FP

There are two properties that we would like to verify for the definition of the measure.

1. For every variable \( v \) of the VARIABLE clause, there exist a data group \( S \) such that \( v \in \text{DET}(S) \).

   This property is not satisfied by our definition. A variable which is neither a subset of an abstract set nor a function from an abstract set is not counted in any data group. The IFPUG CPM is not precise enough about this issue. Our best interpretation of the IFPUG CPM is that these left out variables should be counted as control information. It is also not clear on what basis they should be aggregated into data groups. For the sake of simplicity, we have chosen to count a single data group for all these variables, and one DET for each variable.

2. For every operation \( o \), we would like the following formula to hold:

\[
o \in EI \lor o \in EQ \lor o \in EO
\]

This property is meant to check that an operation is counted at least once. Note that the FP measure allows an elementary process to belong to several categories. For instance, an operation that updates the system state and that delivers derived data belongs to \( EI \) and \( EO \). This property does not hold for our definition, which means that the FP definition is not complete, at least from the interpretation we made of the IFPUG CPM. The truth table in TAB. 4 summarizes how operations are classified. The first four column headings are shorthands for predicates used in the formal definitions. Column CPM indicates the category in which an operation is classified according to our interpretation of the IFPUG CPM. Column B

115
indicates a proposed extension to IFPUG’s definition in order to have a more complete classification. There are twelve cases to consider (rather than sixteen), since derives \(\Rightarrow\) outParam. Lines 1 and 7 denote cases where the operation does nothing.

**Tab. 4 - Processes classification according to IFPUG CPM and the formal rules**

<table>
<thead>
<tr>
<th>no.</th>
<th>inParam</th>
<th>maintains</th>
<th>derives</th>
<th>outParam</th>
<th>CPM</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>V</td>
<td>EO</td>
<td>EO</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>V</td>
<td>-</td>
<td>EO</td>
<td>EO</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>V</td>
<td>F</td>
<td>F</td>
<td>no rule</td>
<td>EI</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>V</td>
<td>F</td>
<td>V</td>
<td>EO</td>
<td>EI and EO</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>V</td>
<td>V</td>
<td>-</td>
<td>EO</td>
<td>EI and EO</td>
</tr>
<tr>
<td>7</td>
<td>V</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>8</td>
<td>V</td>
<td>F</td>
<td>F</td>
<td>V</td>
<td>EQ</td>
<td>EQ</td>
</tr>
<tr>
<td>9</td>
<td>V</td>
<td>F</td>
<td>V</td>
<td>-</td>
<td>EO</td>
<td>EO</td>
</tr>
<tr>
<td>10</td>
<td>V</td>
<td>V</td>
<td>F</td>
<td>F</td>
<td>EI</td>
<td>EI</td>
</tr>
<tr>
<td>11</td>
<td>V</td>
<td>V</td>
<td>F</td>
<td>V</td>
<td>EI and EO</td>
<td>EI and EO</td>
</tr>
<tr>
<td>12</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>-</td>
<td>EI and EO</td>
<td>EI and EO</td>
</tr>
</tbody>
</table>

meaningful for the user: the operation has no parameters nor does it maintain the system state. Hence, the operation can not be counted in the function points. The cases of lines 4, 5 and 6 denote operations which do not have input parameters, but that maintain the internal system state. An operation satisfying line 4 is not counted in function points. We believe that it should, since it maintains the internal system state. For instance, one may think of an operation that increments a counter when an interrupt is received. The operation has no input, but it does modify the system state in a way which can be recognized by the user. To count operations of this type, we propose to extend the definition of set EI as follows:

\[
EI' \triangleq \{o \mid o \in F \land \exists S : S \in DG \land maintains(o,S) \land \\
\neg(\exists p : outParam(o,p))\}
\]
Consequently, cases 5 and 6 must also be counted as ELs; following IFPUG’s CPM, they should only be counted as EO.

B.3 Weighing Components

The last step of a function point count is to compute component weights. The weight of a data group (i.e. an ILF or an EIF) depends on the number of DETs and the number of RETs. We have already defined how to compute DETs and RETs of a data group in Sections B.1.2 and B.1.3. The weight of a transaction (i.e. an EI, an EO, or an EQ) depends on the number of DETs and DGs referenced by the transaction. In this section, we define sets to identify the referenced DGs, which are called file type references (FTRs) in function point terminology, and the referenced DETs of components.

To identify referenced DETs, we need one definition for EI, one for EO, and one for the input part of an EQ, which are denoted by $EI_{RDET}(o)$, $EO_{RDET}(o)$, and $EQ_{IRDET}(o)$, respectively. To identify the set of DETs for the output part of operation $o$ identified as an EQ, the set $EO_{RDET}(o)$ is used.

Set $EI_{RDET}(o)$ must include the input parameters of $o$ and the DETs referenced or maintained by the operation. When a DET $v$ is maintained using an assignment of the form $v := p$, where $p$ is an input parameter, then it is not counted, since the input parameter is counted. We first define set $ODET(o)$ which denotes the set of DETs occurring in an operation $o$.

$$ODET(o) \triangleq \{ v \mid \text{occurs}(v,o) \land \exists S : S \in DG \land v \in DET(S) \}$$

Using $ODET(o)$, we may now define set $EI_{RDET}(o)$. Let $MB_o$ be a set of all the elementary substitutions of operation $o$ which maintain data (the LHS of each is a machine variable). Let $OuterSubstitutions(b)$ be the conditions of outer substitutions.
which must hold for elementary substitution \( b \) to execute.

\[
EI_{-}RDET(o) \triangleq \text{IN\_PARAM}(o) \cup \text{REF\_DET}(o) \cup \text{MAINT\_DET}(o)
\]

\[
\text{IN\_PARAM}(o) \triangleq \{ p \mid \text{inParam}(p,o) \}
\]

\[
\text{REF\_DET}(o) \triangleq \{ v \mid v \in \text{ODET}(o) \land (\exists b : b \in MB_o \land \neg \text{inParam}(o,RHS(b)) \lor \text{occurs}(v,\text{OuterSubstitutions}(b))) \}
\]

\[
\text{MAINT\_DET}(o) \triangleq \{ v \mid v \in \text{ODET}(o) \land
\exists b : b \in MB_o \land v = LHS(b) \land
\neg \text{inParam}(o,RHS(b)) \}
\]

Set \( EI\_FTR(o) \) denotes the set of FTRs (File Type References or referenced data groups) of operation \( o \) identified as an EI.

\[
EI\_FTR(o) \triangleq \{ S \mid \exists v,b : v \in \text{DET}(S) \land b \in MB_o \land
(v = LHS(b) \lor v \in \text{REF\_DET}(o)) \}
\]

Set \( EO\_RDET(o) \) must include the \textit{elementary types} occurring in the \textit{type expression} of each output parameter of \( o \). We omit here the formal definition of an elementary type and a type expression. We refer the reader to [2] for a formal definition. To illustrate these concepts, consider the sales report of Section B.2.4. The type expression of this output parameter is \( SEQ(TUPLE(STRING,N)) \), and its elementary type occurrences are \( STRING \) and \( N \). Therefore, this output parameter has 2 DETs. Let \( \text{ElemTypes}(p) \) be the occurrences of elementary types in output parameter \( p \).

\[
EO\_RDET(o) \triangleq \{ e \mid \exists p : \text{outParam}(p,o) \land e \in \text{ElemTypes}(p) \}
\]
Set $EO_{-FTR}(o)$ denotes the set of FTR (referenced data groups) of operation $o$ identified as an EO.

$$EO_{-FTR}(o) \triangleq \{ S \mid \exists p, b, v : outParam(p, o) \land b \in B_o \land p = LHS(b) \land v \in DET(S) \land (\text{occurs}(v, RHS(b)) \lor \text{occurs}(v, \text{OuterSubstitutions}(b))) \}$$

Set $EQ_{-IRDET}(o)$ must include the input parameters of $o$ and the referenced DETs occurring in the conditions applied to input parameters. Let $Predicates(p)$ be the conditions that input parameter $p$ must fulfill for operation $o$ to execute.

$$EQ_{-IRDET}(o) \triangleq IN_{-PARAM}(o) \cup REF_{-IDET}(o)$$

$$REF_{-IDET}(o) \triangleq \{ v \mid \exists p : inParam(o, p) \land \text{occurs}(v, Predicates(p)) \land v \in DET(S) \}$$

Set $EQ_{-IFTR}(o)$ denotes the set of FTR (referenced data groups) for the input part of operation $o$ identified as an EQ.

$$EQ_{-IFTR}(o) \triangleq \{ S \mid \exists v : v \in DET(S) \land v \in REF_{-IDET}(o) \}$$

To identify the set of FTR for the output part of operation $o$ identified as an EQ, the set $EO_{-FTR}(o)$ is used.

Once we have identified the referenced DETs, RETs, and DGs (FTR), we could now assign a complexity weight to all the component types ($ILF, EIF, EI, EO, \text{or} \ EQ$). According to the IFPUG CPM, the complexity weights of these components are defined in term of a table. For instance, to compute the complexity weights of an $ILF$ we use the complexity table of the form shown in TAB. 5.

In the formal rules, we have defined the DETs, RETs, referenced DETs and DGs as finite sets. Recall that the complexity weight of a transaction ($EI, EO, \text{or} \ EQ$) depends
on the number of the DETs and DGs referenced by the transaction and the complexity weight of data group (ILF or EIF) depends on the DETs and RETs numbers of the data group. In order to contribute a complexity weights to the component types, we need to compute the number of elements in each finite set defined in the formal rules. To do that, we use the cardinal of a finite set $E$, denoted by $\text{Card}(E)$, which returns the number of elements contained in $E$. For instance, suppose we want to compute the complexity weight of a data group $S$ identified as an ILF. In this case, we use "$\text{Card}(DET(S))$" to compute the DETs number of the data group $S$, and "$\text{Card}(RET_m(S))$ plus $\text{Card}(RET_c(S))$" to compute the RETs number of $S$. For the sake of simplicity, we denote "$\text{Card}(RET_m(S))$ plus $\text{Card}(RET_c(S))$" by "$\text{Card}(RET(S))$" as defined in the complexity table of the form shown in TAB. 6.

**TAB. 6 – Complexity weights of an ILF according to the formal rules**

<table>
<thead>
<tr>
<th>$\text{Card}(RET(S))$</th>
<th>$1 \leq \text{Card}(DET(S)) \leq 19$</th>
<th>$20 \leq \text{Card}(DET(S)) \leq 50$</th>
<th>$\text{Card}(DET(S)) \geq 51$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq 1$</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>$2 \leq \text{Card}(RET(S)) \leq 5$</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>$\geq 6$</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

**B.4 Equivalent Operations**

According to [24], if there are two functions which have the same logical processing and the same input/output parameters then we should only count one of both.
In a B language context, if there is an operation $o'$ which has the same elementary substitutions (corresponds to logical processing in function points) and the same input/output parameters as operation $o$ then $o'$ is equivalent to $o$, formally we write $o \equiv o'$. In this case, we count only one of both. For example, in the library specification there are two operations "Discard" and "Sell" which are identified as two EIs. Both has the same elementary substitutions and input parameters. In this case, we should count "Discard" as an EI and ignore "Sell".

The formal definition of $o \equiv o'$ is the following. Let $B_o$ and $B_o'$ be sets of all elementary substitutions of $o$ and $o'$ respectively.

\[
(\forall o,o' \mid o \in F \land o' \in F \land \\
EI_RDET(o) = EI_RDET(o') \land \\
EO_RDET(o) = EO_RDET(o') \land \\
(\forall b,b' \mid b \in B_o \land b' \in B_o' \Rightarrow b = b') \land \\
OuterSubstitutions(b) = OuterSubstitutions(b')) \leftrightarrow o \equiv o'
\]
Annexe C

Cas d’étude: la mesure d’IFPUG-FP à partir d’une spécification B
(Measurement of IFPUG-FP for a B Specification using Formal Rules)

In order to illustrate the application of our definition, we use the B specification language to model the transactions of a library system and to count the number of IFPUG-FP based on our definition. We provide a comparison of the results with three other counts. The first two counts were obtained with the IFPUG FP definitions, based on two different user viewpoints. The third count is derived using a definition of IFPUG FP for use case diagrams from the UML notation. The details of this last approach are provided in [35]. The informal description of the library system is provided in [35].

C.1 List of transactions for the Library System

1. Acquire(Book_Id, Title, Author, Date): adds a new book to the library;
2. Lend(Book_Id, Member_Id, Date): allows a member to borrow a book;

3. Renew(Book_Id, Date): renews a book loan by the same member;

4. Reserve(Reservation_Id, Book_Id, Member_Id, Date): makes a reservation of a book for borrowing by a member;

5. Take(Reservation_Id, Date): borrows a book to a member based on the book reservation that has been made;

6. Return(Book_Id, Date)
   output: Book_Id, Title, Date, Member_Id: allows a member to return a borrowed book to the library. This transaction produces, during its execution, the book identifier, the current date, and the member identifier.

7. Cancel(Reservation_Id): allows a member to cancel the reservation of the specified book;

8. Sell(Book_Id): allows the library staff to sell a book, which means removing the book from the library system;

9. Discard(Book_Id): is similar to the "sell" transaction;

10. Join(Member_Id, Name, Phone, Loan_Limit): adds a new member to the library;

11. Modify_Member(Member_Id, Name, Phone, Loan_Limit): changes the personal information about a member;

12. Leave(Member_Id): allows to cancel the membership of a particular member;

13. List_Book_Descr(Book_Id)
   output: Book_Id, Title, Author: produces a description about a particular book. The description is composed of the book identifier, its title, and the name of its author;

14. List_Books_of_Author(Author)
   output: (Title, Number_of_Loans)*, Total_Number_of_Loans: returns, as output, a list of books and the number of loans per book of a particular author;
15. List_Books_for_Title(Title)
   output: Book_Id*: takes the book title as input and returns the identifier of the
   specified book as output;

16. List_Overdue_Books(Date)
   output: (Member_Id, Book_Id)*: produces a list of overdue books;

17. List_New_Books(Date)
   output: (Title, Author, Book_Id)*: produces a list of all the books that have been
   added after a specific date;

18. List_Reservations_of_Book(Book_Id)
   output: (Member_Id, Name, Phone, Reservation_Id)*: returns a list of reservations
   of a particular book as output;

19. List_Reservations_for_Overdue_Books(Date)
   output: (Member_Id, Name, Phone, Book_Id, Title)*: returns information about
   the reservations of all the overdue books.

C.2 The B Specification of the Library System

MACHINE

Library(MaxLoanDuration, Date_Day)

CONSTRAINTS

MaxLoanDuration ∈ NAT ∧ Date_Day ∈ NAT

SETS

BOOK;
TITLE;
AUTHOR;
DATE;
MEMBER;
RESERVATION;
LOAN_STATUS = {InLibrary, Loaned};
NAME;
PHONE;
ERROR_MESSAGE = {BookNotFound, BookFound}

DEFINITIONS

\[\text{sorted}(ss, ff) = \forall \ ii. (ii \in \text{dom}(ss) \land ii \neq \text{card}(ss) \rightarrow ff(ss(ii)) \leq ff(ss(ii+1)))\]

VARIABLES

book, title, author, acquisitionDate, curBorrower,
lastLoanDate, loanStatus, loanCnt,
reservation, resMemberId, resBookId, resDate,
member, memName, memPhone, memLoanLimit

INVARIANT

\[\text{member} \subseteq \text{MEMBER} \land \]
\[\text{book} \subseteq \text{BOOK} \land \]
\[\text{title} \in \text{book} \rightarrow \text{TITLE} \land \]
\[\text{author} \in \text{book} \rightarrow \text{AUTHOR} \land \]
\[\text{acquisitionDate} \in \text{book} \rightarrow \text{DATE} \land \]
\[\text{curBorrower} \in \text{book} \rightarrow \text{member} \land \]
\[\text{lastLoanDate} \in \text{book} \rightarrow \text{NAT} \land \]
loanStatus \in book \rightarrow \text{LOAN\_STATUS} \wedge 
loanCnt \in book \rightarrow \text{NAT} \wedge 
reservation \subseteq \text{RESERVATION} \wedge 
resMemberId \in reservation \rightarrow \text{member} \wedge 
resBookId \in reservation \rightarrow \text{book} \wedge 
resDate \in reservation \rightarrow \text{DATE} \wedge 
memName \in \text{member} \rightarrow \text{NAME} \wedge 
memPhone \in \text{member} \rightarrow \text{PHONE} \wedge 
memLoanLimit \in \text{member} \rightarrow \text{NAT} 

\text{INITIALISATION} 

\begin{align*}
\text{book} & := \emptyset \mid \\
\text{title} & := \emptyset \mid \\
\text{author} & := \emptyset \mid \\
\text{acquisitionDate} & := \emptyset \mid \\
\text{curBorrower} & := \emptyset \mid \\
\text{lastLoanDate} & := \emptyset \mid \\
\text{loanStatus} & := \emptyset \mid \\
\text{loanCnt} & := \emptyset \mid \\
\text{reservation} & := \emptyset \mid \\
\text{resMemberId} & := \emptyset \mid \\
\text{resBookId} & := \emptyset \mid \\
\text{resDate} & := \emptyset \mid \\
\text{member} & := \emptyset \mid \\
\text{memName} & := \emptyset \mid \\
\text{memPhone} & := \emptyset \mid 
\end{align*}
memLoanLimit := ∅

OPERATIONS

Acquire(bbookid, btitle, bauthor, bdate) =

PRE

book ≠ BOOK ∧
bbookid ∈ BOOK - book ∧
btitle ∈ TITLE ∧
bauthor ∈ AUTHOR ∧
bdate ∈ DATE

THEN

book := book ∪ {bbookid} ||
title(bbookid) := btitle ||
author(bbookid) := bauthor ||
acquisitionDate(bbookid) := bdate ||
loanStatus(bbookid) := InLibrary ||
loanCnt(bbookid) := 0

END;

Lend(bbookid, bmemberid, bdate) =

PRE

bbookid ∈ book ∧
bmemberid ∈ member ∧
bdate ∈ NAT ∧
loanStatus(bbookid) = InLibrary ∧
\card(\text{curBorrower}^{-1} \{\text{bmemberid}\}) \leq \text{memLoanLimit}(\text{bmemberid}) ∧
loanCnt(bbookid) + 1 ≤ MAXINT

THEN

curBorrower(bbookid) := bmemberid ||
lastLoanDate(bbookid) := bdate ||
loanStatus(bbookid) := Loaned ||
loanCnt(bbookid) := loanCnt(bbookid) + 1

END;

Renew(bbookid, bdate) =

PRE

bbookid ∈ book ∧

bdate ∈ NAT ∧

loanStatus(bbookid) = Loaned

THEN

lastLoanDate(bbookid) := bdate

END;

bb, tt, dd, cc ← Return(bbookid, bdate) =

PRE

bbookid ∈ book ∧

bdate ∈ DATE ∧

loanStatus(bbookid) = Loaned

THEN

curBorrower := {bbookid} ⋈ curBorrower ||
lastLoanDate := {bbookid} ⋈ lastLoanDate ||
loanStatus(bbookid) := InLibrary ||
\[ \text{bb, tt, dd, cc := } (\text{bbookid, title(bbookid), bdate, curBorrower(bbookid)}) \]

END;

\textbf{Sell(bbookid) =}

\textbf{PRE}
\[
\text{bbookid } \in \text{ book } \land \\
\text{loanStatus(bbookid) = InLibrary } \land \\
\text{resBookId}^{-1} [[\{\text{bbookid}\}] = \emptyset
\]

\textbf{THEN}
\[
\text{book := book - } \{\text{bbookid}\} \parallel \\
\text{title := } \{\text{bbookid}\} \triangleleft \text{ title } \parallel \\
\text{author := } \{\text{bbookid}\} \triangleleft \text{ author } \parallel \\
\text{acquisitionDate := } \{\text{bbookid}\} \triangleleft \text{ acquisitionDate } \parallel \\
\text{loanStatus := } \{\text{bbookid}\} \triangleleft \text{ loanStatus } \parallel \\
\text{lastLoanDate := } \{\text{bbookid}\} \triangleleft \text{ lastLoanDate } \parallel \\
\text{loanCnt := } \{\text{bbookid}\} \triangleleft \text{ loanCnt}
\]

\textbf{END;}

\textbf{Discard(bbookid) =}

\textbf{PRE}
\[
\text{bbookid } \in \text{ book } \land \\
\text{loanStatus(bbookid) = InLibrary } \land \\
\text{resBookId}^{-1} [[\{\text{bbookid}\}] = \emptyset
\]

\textbf{THEN}
\[
\text{book := book - } \{\text{bbookid}\} \parallel \\
\text{title := } \{\text{bbookid}\} \triangleleft \text{ title } \parallel \\
\text{author := } \{\text{bbookid}\} \triangleleft \text{ author } \parallel 
\]

129
acquisitionDate := {bbookid} \in acquisitionDate ||
loanStatus := {bbookid} \in loanStatus ||
lastLoanDate := {bbookid} \in lastLoanDate ||
loanCnt := {bbookid} \in loanCnt

END;

Reserve(breservid, bbookid, bmemberid, bdate) =

PRE
reservation \neq RESERVATION \land
breservid \in RESERVATION - reservation \land
bbookid \in book \land
bmemberid \in member \land
bdate \in DATE

THEN
reservation := reservation \cup \{breservid\} ||
resMemberId(breservid) := bmemberid ||
resBookId(breservid) := bbookid ||
resDate(breservid) := bdate

END;

Take(breservid, bmemberid, bdate, bbookid) =

PRE
breservid \in reservation \land
bmemberid \in member \land
bdate \in NAT \land
bbookid \in book \land
loanStatus(bbookid) = InLibrary \land
\[ \text{card}(\text{curBorrower}^{-1}[\{\text{bmemberid}\}]) \leq \text{memLoanLimit}(\text{bmemberid}) \land \]
\[ \text{loanCnt}(\text{resBookId}(\text{breservid})) + 1 \leq \text{MAXINT} \]

\text{THEN}

\begin{align*}
\text{reservation} & := \text{reservation} - \{\text{breservid}\} \land \\
\text{resMemberId} & := \{\text{breservid}\} \leq \text{resMemberId} \land \\
\text{resBookId} & := \{\text{breservid}\} \leq \text{resBookId} \land \\
\text{resDate} & := \{\text{breservid}\} \leq \text{resDate} \land \\
\text{curBorrower}(\text{resBookId}(\text{breservid})) & := \text{resMemberId}(\text{breservid}) \land \\
\text{lastLoanDate}(\text{resBookId}(\text{breservid})) & := \text{bdate} \land \\
\text{loanStatus}(\text{resBookId}(\text{breservid})) & := \text{Loaned} \land \\
\text{loanCnt}(\text{resBookId}(\text{breservid})) & := \text{loanCnt}(\text{resBookId}(\text{breservid})) + 1
\end{align*}

\text{END;}

\text{Cancel}(\text{breservid}) =

\text{PRE}

\[ \text{breservid} \in \text{reservation} \]

\text{THEN}

\begin{align*}
\text{reservation} & := \text{reservation} - \{\text{breservid}\} \land \\
\text{resMemberId} & := \{\text{breservid}\} \leq \text{resMemberId} \land \\
\text{resBookId} & := \{\text{breservid}\} \leq \text{resBookId} \land \\
\text{resDate} & := \{\text{breservid}\} \leq \text{resDate}
\end{align*}

\text{END;}

\text{Join}(\text{bmemberid}, \text{bname}, \text{bphone}, \text{loanlimit}) =

\text{PRE}

\[ \text{member} \neq \text{MEMBER} \land \\
\]

131
\[ b\text{memberid} \in \text{MEMBER} \land \]
\[ b\text{name} \in \text{NAME} \land \]
\[ b\text{phone} \in \text{PHONE} \land \]
\[ b\text{loanlimit} \in \text{NAT} \]

\text{THEN}
\begin{align*}
\text{member} & := \text{member} \cup \{b\text{memberid}\} \land \\
\text{memName}(b\text{memberid}) & := b\text{name} \land \\
\text{memPhone}(b\text{memberid}) & := b\text{phone} \land \\
\text{memLoanLimit}(b\text{memberid}) & := b\text{loanlimit} \\
\end{align*}

\text{END ;}

\text{Modify}(b\text{memberid}, b\text{name}, b\text{phone}, b\text{loanlimit}) =

\text{PRE}
\begin{align*}
b\text{memberid} & \in \text{member} \land \\
b\text{name} & \in \text{NAME} \land \\
b\text{phone} & \in \text{PHONE} \land \\
b\text{loanlimit} & \in \text{NAT} \\
\end{align*}

\text{THEN}
\begin{align*}
\text{memName}(b\text{memberid}) & := b\text{name} \land \\
\text{memPhone}(b\text{memberid}) & := b\text{phone} \land \\
\text{memLoanLimit}(b\text{memberid}) & := b\text{loanlimit} \\
\end{align*}

\text{END ;}

\text{Leave}(b\text{memberid}) =

\text{PRE}
\begin{align*}
b\text{memberid} & \in \text{member} \land \\
\text{resMemberId}^{-1} \{b\text{memberid}\} & = \emptyset \land \\
\end{align*}
\( \text{curBorrower}^{-1} \{b\text{memberid}\} = \emptyset \)

THEN

\[
\begin{align*}
\text{member} & := \text{member} - \{b\text{memberid}\} || \\
\text{memName} & := \{b\text{memberid}\} \triangleq \text{memName} || \\
\text{memPhone} & := \{b\text{memberid}\} \triangleq \text{memPhone} || \\
\text{memLoanLimit} & := \{b\text{memberid}\} \triangleq \text{memLoanLimit}
\end{align*}
\]

END;

\( \text{listBookdescription} \leftarrow \text{ListBookDescription}(b\text{bookid}) = \)

PRE

\( b\text{bookid} \in \text{book} \)

THEN

ANY \( \text{bookseq} \) WHERE

\[
\begin{align*}
\text{bookseq} & \in \text{seq}(\text{TITLE} \times \text{AUTHOR}) \land \\
\text{ran(bookseq)} & = \{tt, aa \mid tt, aa \in \text{TITLE} \times \text{AUTHOR} \land \\
tt & = \text{title}(b\text{bookid}) \land \\
\text{aa} & = \text{author}(b\text{bookid})\} \land \\
\text{sorted(bookseq,prj1(TITLE,AUTHOR))}
\end{align*}
\]

THEN

\( \text{listBookdescription} := \text{bookseq} \)

END

END;

\( \text{authorlist} \leftarrow \text{ListBookAuthor}(b\text{author}) = \)

PRE

\( b\text{author} \in \text{AUTHOR} \)

THEN
\textbf{ANY} \textit{bookseq} \textbf{WHERE}

\begin{align*}
\textit{bookseq} & \in \text{seq}(\text{TITLE} \times \text{NAT}) \land \\
\text{ran}(\textit{bookseq}) & = \{ tt, nn \mid tt, nn \in \text{TITLE} \times \text{NAT} \land \\
\exists \textit{bbookid}. (\textit{bbookid} \in \text{author}^{-1}[\{\textit{bauthor}\}] \land \\
\textit{tt} & = \text{title}(\textit{bbookid}) \land \\
\textit{nn} & = \text{loanCnt}(\textit{bbookid})\} \land \\
\text{sorted}(\textit{bookseq}, \text{prj1}(\text{TITLE}, \text{NAT}))
\end{align*}

\textbf{THEN}

\textit{authorlist} := \{(\textit{bookseq} \mapsto \text{card}(\text{dom}(\textit{bookseq})))\}

\textbf{END}

\textbf{END};

\textit{booklist} \leftarrow \text{ListBookTitle}(\textit{btitle}) =

\textbf{PRE}

\begin{align*}
\textit{btitle} & \in \text{TITLE}
\end{align*}

\textbf{THEN}

\textbf{ANY} \textit{bookseq} \textbf{WHERE}

\begin{align*}
\textit{bookseq} & \in \text{seq}(\text{BOOK}) \land \\
\text{ran}(\textit{bookseq}) & = \{ bb \mid bb \in \text{title}^{-1}[\{\textit{btitle}\}] \land \\
\text{loanStatus}(bb) & = \text{InLibrary} \} \land \\
\text{sorted}(\textit{bookseq}, \text{id}(\text{BOOK}))
\end{align*}

\textbf{THEN}

\begin{align*}
\textit{booklist} & := \textit{bookseq}
\end{align*}

\textbf{END}
END;

booklist ← ListOverdueBooks =

ANY bbookid, bmemberid, bookseq WHERE
    bmemberid ∈ member ∧
    bbookid ∈ book ∧
    bmemberid = curBorrower(bbookid) ∧
    Date.Day - lastLoanDate(bbookid) ≥ MaxLoanDuration ∧
    bookseq ∈ seq(MEMBER × BOOK) ∧
    ran(bookseq) = { mm,bb | bb ∈ book ∧
    mm = curBorrower(bb) ∧
    Date.Day - lastLoanDate(bb) ≥ MaxLoanDuration} ∧
    sorted(bookseq,prj1(MEMBER,BOOK))

THEN
    booklist := bookseq

END;

booklist ← ListNewBooks(bdate) =

PRE
    bdate ∈ DATE

THEN

ANY bookseq WHERE
    bookseq ∈ seq(TITLE × AUTHOR) ∧
    ran(bookseq) = { tt, aa | ∃ bb . (bb ∈ book ∧
    acquisitionDate(bb) ≥ bdate ∧
    tt = title(bb) ∧
    aa = author(bb) } ∧
sorted(bookseq, prj1(TITLE,AUTHOR))

THEN

booklist := bookseq

END

END;

booklist, response ← ListReservationOfBooks(bbookid) =

PRE

bbookid ∈ BOOK

THEN

IF

bbookid ∈ book

THEN

ANY bookseq WHERE

bookseq ∈ seq(MEMBER × NAME × PHONE × RESERVATION) ∧

ran(bookseq) = { mm,nn,pp,rr |

rr ∈ resBookId⁻¹ [{bbookid}] ∧

mm = resMemberId(rr) ∧

nn = memName(mm) ∧

pp = memPhone(mm) }

THEN

booklist := bookseq ||

response := BookFound

END

ELSE

booklist := ∅ ||

response := BookNotFound

136
booklist ← ListReservationForOverdueBooks(bdate) =

PRE
   bdate ∈ NAT
THEN
   ANY bookseq WHERE
      bookseq ∈ seq(MEMBER × NAME × PHONE × BOOK × TITLE) ∧
      ran(bookseq) = { mm, nn, pp, bb, tt | 
      bb ∈ ran(resBookId) ∧
      bdate - lastLoanDate(bb) ≥ MaxLoanDuration ∧
      ∃ rr. (rr ∈ resBookId⁻¹ [{bb}] ∧
      mm = resMemberId(rr) ∧
      nn = memName(mm) ∧
      pp = memPhone(mm) ∧
      tt = title(bb)) } THEN
      booklist := bookseq
   END
END

C.3 Detailed Calculations of IFPUG's FP Counts

Table 7 provides the list of components with their complexity level for the function point count obtained using UML. Table 8 provides the list of components with their
complexity level for the function point count obtained using our formal definition. Table 9 provides the list of components for the counts made with IFPUG’s CPM. In this table, CPM1 and CPM2 denote the two different view-points that a user may take for the data groups. In CPM1, the user identifies three data groups (member, book and reservation), whereas in CPM2 only two data groups are identified (member and book).

**Tab. 7. Unadjusted function points of the library system using UML**

<table>
<thead>
<tr>
<th>Type of component</th>
<th>Number of components by complexity</th>
<th>Weight by complexity</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low  Medium High</td>
<td>Low  Medium High</td>
<td></td>
</tr>
<tr>
<td>ILF</td>
<td>3     0      0</td>
<td>7     10    15</td>
<td>21</td>
</tr>
<tr>
<td>EIF</td>
<td>0     0      0</td>
<td>5     7      10</td>
<td>0</td>
</tr>
<tr>
<td>EI</td>
<td>13    0      0</td>
<td>3     4      6</td>
<td>39</td>
</tr>
<tr>
<td>EO</td>
<td>13    0      0</td>
<td>4     5      7</td>
<td>52</td>
</tr>
<tr>
<td>EQ</td>
<td>0     0      0</td>
<td>3     4      6</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>112</td>
</tr>
</tbody>
</table>
### Component Complexity Table

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Number of RETs/FTRs</th>
<th>Number of DETs</th>
<th>Complexity level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>ILF</td>
<td>2</td>
<td>8</td>
<td>Low</td>
</tr>
<tr>
<td>Reservation</td>
<td>ILF</td>
<td>1</td>
<td>4</td>
<td>Low</td>
</tr>
<tr>
<td>Member</td>
<td>ILF</td>
<td>1</td>
<td>4</td>
<td>Low</td>
</tr>
<tr>
<td>Acquire</td>
<td>EI</td>
<td>1</td>
<td>7</td>
<td>Low</td>
</tr>
<tr>
<td>Lend</td>
<td>EI</td>
<td>2</td>
<td>7</td>
<td>Medium</td>
</tr>
<tr>
<td>Renew</td>
<td>EI</td>
<td>1</td>
<td>3</td>
<td>Low</td>
</tr>
<tr>
<td>Return</td>
<td>EI</td>
<td>1</td>
<td>5</td>
<td>Low</td>
</tr>
<tr>
<td>Return</td>
<td>EO</td>
<td>1</td>
<td>4</td>
<td>Low</td>
</tr>
<tr>
<td>Sell/Discard</td>
<td>EI</td>
<td>2</td>
<td>9</td>
<td>Medium</td>
</tr>
<tr>
<td>Reserve</td>
<td>EI</td>
<td>3</td>
<td>5</td>
<td>High</td>
</tr>
<tr>
<td>Take</td>
<td>EI</td>
<td>3</td>
<td>12</td>
<td>High</td>
</tr>
<tr>
<td>Cancel</td>
<td>EI</td>
<td>1</td>
<td>5</td>
<td>Low</td>
</tr>
<tr>
<td>Join</td>
<td>EI</td>
<td>1</td>
<td>5</td>
<td>Low</td>
</tr>
<tr>
<td>ModifyMember</td>
<td>EI</td>
<td>1</td>
<td>4</td>
<td>Low</td>
</tr>
<tr>
<td>Leave</td>
<td>EI</td>
<td>3</td>
<td>7</td>
<td>High</td>
</tr>
<tr>
<td>ListBookDescription</td>
<td>EQ</td>
<td>1</td>
<td>3</td>
<td>Low</td>
</tr>
<tr>
<td>ListBooksAuthor</td>
<td>EQ</td>
<td>1</td>
<td>1</td>
<td>Low</td>
</tr>
<tr>
<td>ListBooksTitle</td>
<td>EQ</td>
<td>1</td>
<td>1</td>
<td>Low</td>
</tr>
<tr>
<td>ListOverdueBooks</td>
<td>EQ</td>
<td>2</td>
<td>2</td>
<td>Low</td>
</tr>
<tr>
<td>ListNewBooks</td>
<td>EQ</td>
<td>1</td>
<td>2</td>
<td>Low</td>
</tr>
<tr>
<td>ListReservationofBooks</td>
<td>EQ</td>
<td>3</td>
<td>4</td>
<td>Low</td>
</tr>
<tr>
<td>ListReservationFor-</td>
<td>EQ</td>
<td>3</td>
<td>5</td>
<td>Low</td>
</tr>
<tr>
<td>OverdueBooks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Tab. 8 - The components of the library system using our formal rules*
TAB. 9 – The components of the library system using IFPUG rules

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Number of RETs/FTRs</th>
<th>Number of DETs</th>
<th>Complexity level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPM1</td>
<td>CPM2</td>
<td>CPM1</td>
</tr>
<tr>
<td>Book</td>
<td>ILF</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Reservation</td>
<td>ILF</td>
<td>1</td>
<td>N/A</td>
<td>4</td>
</tr>
<tr>
<td>Member</td>
<td>ILF</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Acquire</td>
<td>EI</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Lend</td>
<td>EI</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Renew</td>
<td>EI</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Return</td>
<td>EI</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Return</td>
<td>EO</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Sell/Discard</td>
<td>EI</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Reserve</td>
<td>EI</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Take</td>
<td>EI</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Cancel</td>
<td>EI</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Join</td>
<td>EI</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>ModifyMember</td>
<td>EI</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Leave</td>
<td>EI</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ListBookDescription</td>
<td>EQ</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ListBooksAuthor</td>
<td>EO</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ListBooksTitle</td>
<td>EQ</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ListOverdueBooks</td>
<td>EQ</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ListNewBooks</td>
<td>EQ</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ListReservationofBooks</td>
<td>EQ</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>ListReservationFor-OverdueBooks</td>
<td>EQ</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Annexe D

Le document des spécifications de l'outil $\mu_cROSE$ (Specification Document of $\mu_cROSE$)

In this chapter, we introduce the purpose of $\mu_cROSE$ and provide a brief description of its architecture, its specification written in the B language, class models, syntax analyzer algorithm, and the results of some of the case studies.

D.1 Purpose of $\mu_cROSE$

The purpose of $\mu_cROSE$ is for measuring the software functional size based on the COSMIC-FFP method. The input to this tool is a software specification produced by the Rational Rose RealTime toolset. The output is the functional software size derived from the COSMIC-FFP measurement rules.

The tool manipulates an RRRT linear form design, consisting of a mixture of C++ code and RRRT notation. It provides a syntax analyzer, hierarchical RRRT model navigation facility, and graphical interface. Major characteristics of the tool are described as the
following.

- The tool provides a standard interface that is easy to learn and easy to use by employing icons, multiple windows, and a mouse. Each operation is performed by selecting the specification (set of capsules) to measure and a command from a menu.
- The tool is applicable to a specification written in RRRT modeling language (RTMDL file extension). It can easily incorporate new features and be integrated in the Rational Rose RealTime toolset.
- The tool reduces the time and cost of measuring the functional size of software.

D.2 Formal Specification of $\mu_cROSE$

MACHINE global\_microse

/* Declaration of abstract sets of the different RRRT objects required to perform the COSMIC-FFP measurement */

SETS
  CAPSULE;
  PROTOCOL;
  STATE;
  TRANSITION;
  ACTION;
  TRIGGER;
  MESSAGE;
  PORT;
  FUNCTIONAL\_PROCESS;
  ATTRIBUTE;
  DATA\_CLASS;
\textit{CAPSULE\_TYPE} = \{Selected, notSelected\};
\textit{DATA\_TYPE} = \{Predefined, Defined\};
\textit{PORT\_TYPE} = \{Internal, External\};
\textit{BOOLEAN} = \{True, False\}

\textbf{ABSTRACT\_CONSTANTS}

\textit{RELATED\_TRANSITION},

\textit{constructChain}

\(/^{*}\) Definition of a property that allows to establish a set of related transitions. Two transitions are related iff one triggers the processing of the other during its execution or they belong to the same group of transition segments \(^*/^{*}\)

\textbf{PROPERTIES}

\[
\textit{RELATED\_TRANSITION} \in \text{TRANSITION} \leftrightarrow \text{TRANSITION} \wedge
\]

\[
\textit{constructChain} \in ((\mathcal{P}(\text{TRANSITION}) \times \mathcal{P}(\text{RELATED\_TRANSITION})) \times \mathcal{P}(\text{TRANSITION})) \rightarrow \mathcal{P}(\text{TRANSITION}) \wedge
\]

\(/^{*}\) \textit{TT} denotes a set of transitions triggered by the arrival of a message. \textit{GG} represents a set of transitions related to transitions within \textit{TT}. \textit{VV} represents a set of transitions that have been identified during the \textit{constructChain} execution \(^*/^{*}\)

\[
\forall (\textit{TT,GG,VV}). (\textit{TT} \subseteq \text{TRANSITION} \wedge \textit{TT} \in \mathcal{P}(\text{TRANSITION}) \wedge
\textit{GG} \in \text{TRANSITION} \leftrightarrow \text{TRANSITION} \wedge \textit{GG} \subseteq \text{RELATED\_TRANSITION} \wedge
\textit{VV} \subseteq \text{TRANSITION} \wedge \textit{VV} \in \mathcal{P}(\text{TRANSITION}) \wedge
\textit{TT} \subseteq \textit{VV} \Rightarrow \text{constructChain}(\textit{TT,GG,VV}) = \emptyset \) \wedge
\]

143
∀ (TT, GG, VV). (TT ⊆ TRANSITION ∧ TT ∈ ℙ(TRANSITION) ∧
GG ∈ TRANSITION ↔ TRANSITION ∧ GG ⊆ RELATED_TRANSITION ∧
VV ⊆ TRANSITION ∧ VV ∈ ℙ(TRANSITION) ∧
TT ∩ (TRANSITION-VV) ≠ ∅ ⇒ constructChain(TT, GG, VV) = TT ∪
constructChain(GG[T], GG, (VV ∪ TT))

/* Definition of each variable needed to represent the different RRR objects. These
variables will be manipulated by the operations in order to identify the COSMIC-FFP
components and calculate the number of function points */

VARIABLES

capsule, capsuleInBoundary, boundary,
port, portType, portOfCapsule, portReferToProtocol,
portConnectedToPort, wiredPort, portInterne,
message, messageInProtocol, messageDataGroup,
messageSentByTransition, messageReceivedByTransition,
protocol,
attributeOfCapsule, attributeType, attribute,
dataClass, dataClassType, attributeOfDataClass,
attributeOfSimpleDataGroup,
simpleDataGroup, isDataGroup,
trigger, guardOfTrigger,
statePoint,
state, entryActionOfState,
exitActionOfState,
action,
transition, sourceStateOfTransition,
targetStateOfTransition, actionOfTransition,
triggerOfTransition, sourcePointStateOfTransition,  
targetPointStateOfTransition,  
transitionsOfCapsule,  
TransitionsTriggeredByMessage,  
chainOfRelatedTransitions,  
relatedTransitions,  
functionalProcess, messageTriggeringFunctionalProcess,  
entry, exit, read, write,  
consults, maintains  

/* Definition of each predicate that must be preserved by the execution of an operation */  

INVARIANT  

dataClass ⊆ DATA_CLASS ∧  
attribute ⊆ ATTRIBUTE ∧  
attributeOfDataClass ∈ attribute → dataClass ∧  
attributeType ∈ attribute → dataClass ∧  
dataClassType ∈ dataClass → DATA_TYPE ∧  
capsule ⊆ CAPSULE ∧  
boundary ⊆ capsule ∧  

capsuleInBoundary ∈ capsule → CAPSULE_TYPE ∧  
attributeOfCapsule ∈ attribute → capsule ∧  

isDataGroup ∈ dataClass → BOOLEAN ∧  
simpleDataGroup ⊆ DATA_CLASS ∧  
attributeOfSimpleDataGroup ∈ simpleDataGroup → P (attribute) ∧  

145
action \subseteq ACTION \land \\
\text{port} \subseteq PORT \land \\
\text{portType} \in \text{port} \rightarrow PORT\_TYPE \land \\
\text{protocol} \subseteq PROTOCOL \land \\
\text{wiredPort} \in \text{port} \rightarrow BOOLEAN \land \\
\text{portReferToProtocol} \in \text{port} \rightarrow \text{protocol} \land \\
\text{portOfCapsule} \in \text{port} \rightarrow \text{capsule} \land \\
\text{portConnectedToPort} \in \text{port} \rightarrow \text{port} \land \\
\\
\text{message} \subseteq MESSAGE \land \\
\text{messageInProtocol} \in \text{message} \leftrightarrow \text{protocol} \land \\
\\
\text{messageDataGroup} \in \text{capsule} \leftrightarrow P(\text{message}) \land \\
\\
\text{trigger} \subseteq TRIGGER \land \\
\text{guardOfTrigger} \in \text{trigger} \leftrightarrow \text{action} \land \\
\\
\text{state} \subseteq STATE \land \\
\text{statePoint} \subseteq STATE \land \\
\text{state} \cap \text{statePoint} = \emptyset \land \\
\\
\text{entryActionOfState} \in \text{state} \rightarrow \text{action} \land \\
\text{exitActionOfState} \in \text{state} \rightarrow \text{action} \land \\
\\
\text{transition} \subseteq TRANSITION \land \\
\text{sourceStateOfTransition} \in \text{transition} \rightarrow \text{state} \land
targetStateOfTransition ∈ transition → state ∧
actionOfTransition ∈ transition → action ∧
triggerOfTransition ∈ transition → trigger ∧

transitionsOfCapsule ∈ transition → capsule ∧

messageSentByTransition ∈ transition ↔ (message × port) ∧
messageReceivedByTransition : (message × port) ↔ transition ∧
TransitionsTriggeredByMessage ∈ message → P (transition) ∧

functionalProcess ⊆ FUNCTIONAL_PROCESS ∧
messageTriggeringFunctionalProcess ∈ message ↔ functionalProcess ∧

chainOfRelatedTransitions ∈ message → P (transition) ∧
relatedTransitions ∈ transition ↔ transition ∧
portInterne ⊆ port ∧

sourcePointStateOfTransition ∈ transition → startPoint ∧
targetPointStateOfTransition ∈ transition → startPoint ∧

entry ∈ functionalProcess → P (message) ∧
exit ∈ functionalProcess → P (message) ∧
read ∈ functionalProcess → P (dataClass) ∧
write ∈ functionalProcess → P (dataClass) ∧
\[
\text{consults} \in \text{action} \rightarrow \mathcal{P} \left( \text{attribute} \right) \land \\
\text{maintains} \in \text{action} \rightarrow \mathcal{P} \left( \text{attribute} \right) \land \\

\forall \, \mathbf{f}. \, (\mathbf{f} \in \text{functionalProcess} \Rightarrow \\
\text{card(} \text{write}(\mathbf{f}) \text{)} > 0 \lor \text{card(} \text{exit}(\mathbf{f}) \text{)} > 0 )
\]

/* Initialisation of all the variables that have been defined in the VARIABLE clause. We assign the empty set as a value to each variable. This initialisation defines the initial state of } \mu_\text{ROSE} */

\textbf{INITIALISATION}
\[
\text{capsule} := \emptyset || \\
\text{boundary} := \emptyset || \\
\text{attribute} := \emptyset || \\
\text{dataClass} := \emptyset || \\
\text{capsuleInBoundary} := \emptyset || \\
\text{attributeOfClass} := \emptyset || \\
\text{attributeType} := \emptyset || \\
\text{dataClassType} := \emptyset || \\
\text{attributeOfDataClass} := \emptyset || \\
\text{attributeOfSimpleDataGroup} := \emptyset || \\
\text{isDataGroup} := \emptyset || \\
\text{simpleDataGroup} := \emptyset || \\
\text{wiredPort} := \emptyset || \\
\text{protocol} := \emptyset || \\
\text{portReferToProtocol} := \emptyset || \\
\]
portOfCapsule := 0 ||
portConnectedToPort := 0 ||
portInterne := 0 ||
messageInProtocol := 0 ||
messageDataGroup := 0 ||
action := 0 ||
message := 0 ||
port := 0 ||
portType := 0 ||
trigger := 0 ||
guardOfTrigger := 0 ||
state := 0 ||
statePoint := 0 ||
entryActionOfState := 0 ||
exitActionOfState := 0 ||
transition := 0 ||
sourceStateOfTransition := 0 ||
targetStateOfTransition := 0 ||
actionOfTransition := 0 ||
triggerOfTransition := 0 ||
transitionsOfCapsule := 0 ||
messageReceivedByTransition := 0 ||
messageSentByTransition := 0 ||
TransitionsTriggeredByMessage := 0 ||
functionalProcess := 0 ||
messageTriggeringFunctionalProcess := 0 ||
relatedTransitions := 0 ||
chainOfRelatedTransitions := \emptyset \ ||
sourcePointStateOfTransition := \emptyset \ ||
targetPointStateOfTransition := \emptyset \ ||
entry := \emptyset \ ||
exit := \emptyset \ ||
read := \emptyset \ ||
write := \emptyset \ ||
consults := \emptyset \ ||
maintains := \emptyset

OPERATIONS

/* This operation allows a user to select a set of capsules that perform at the same level of abstraction. This set corresponds to a layer in the context of COSMIC-FFP. The graphical demarcation of these capsules represents the boundary of the RRRRT model to be measured */

selectCapsule (cc) =

PRE
cc ∈ capsule

THEN

capsuleInBoundary(cc) := Selected ||
boundary := boundary ∪ \{cc\}

END;

/* This operation determines the type of a port. There are two possible types: external and internal. A port is called external iff it communicates with another port of a capsule from outside the boundary or refers to a time protocol. Otherwise, it is called internal

150
determineTypeOfPort(pp, cc) =

PRE
    pp \in port \land
    cc \in boundary \land
    portOfCapsule(pp) = cc

THEN
IF
  ( ( \exists (pp1, cc1). (pp1 \in port \land
    cc1 \in capsule \land cc1 \notin boundary \land
    portConnectedToPort(pp1) = pp \land
    portOfCapsule(pp1) = cc1 \land
    wiredPort(pp) = True
  )
  \lor
  (wiredPort(pp) = False)

THEN
  portType(pp) := External

ELSE
  portType(pp) := Internal ||
  portInterne:= portInterne \cup \{pp\}
END
END;
/* This operation identifies a data group based on a message exchanged between capsules over ports */

messageDataGroupByCapsule(cc,pp) =

PRE
  cc ∈ boundary ∧
  pp ∈ port ∧
  portType(pp) = External ∧
  portOfCapsule(pp) = cc
THEN
  messageDataGroup(cc) := messageInProtocol⁻¹ [{portReferToProtocol(pp)}]
∪ messageDataGroup(cc)
END;

/* This operation identifies a data group based on a set of selected capsule attributes that have been defined by using predefined data types (integer, boolean, etc.) */

simpleDataGroupByCapsule(cc) =

PRE
  cc ∈ boundary
THEN
  ANY dd WHERE dd ∈ DATA_CLASS ∧
  dd ∉ dataClass ∧
  dd ∉ simpleDataGroup ∧
  ∃ ee (ee ∈ attributeOfCapsule⁻¹ [{cc}] ∧
  dataClassType(attributeType(ee)) = Predefined)
THEN
  simpleDataGroup := simpleDataGroup ∪ {dd} ||
attributeOfSimpleDataGroup(dd) :=
   \{ aa | aa \in attributeOfCapsule^{-1}[[cc]] \land \\
   dataClassType(attributeType(aa)) = Predefined \}

END

END;

/* This operation identifies a data group based on a set of selected capsule attributes 
that have been defined by using defined data types (data classes) */

classDataGroupByCapsule(cc) =

PRE
   cc \in boundary
THEN
   isDataGroup := isDataGroup \equiv \lambda dd. (dd \in dataClass \land \\
   \exists aa. (aa \in attributeOfCapsule^{-1}[[cc]] \land \\
   attributeType(aa) = dd \land \\
   dataClassType(attributeType(aa)) = Defined) \lor \\
   True )

END;

/* This operation identifies a set of external transitions that might be triggered by the 
arrival of a message through external ports. A transition is called external iff it is fired 
by an incoming message over an external port */

determineTransitionsTriggeredByExternalMessage(mm) =

PRE
   mm \in message \land \\
   \exists cc.(cc \in boundary \land mm \in messageDataGroup(cc))
THEN

\[
\text{TransitionsTriggeredByMessage}(mm) := \{ tt \mid tt \in \text{transition} \land \\
\exists (pp, cc). (cc \in \text{boundary} \land \\
pp \in \text{portOfCapsule}^{-1} [(\{cc\})] \land \\
\text{transitionOfCapsule}^{-1} [(\{cc\})] \land \\
mm \mapsto pp \in \text{messageRecievedByTransition}^{-1} [\{tt\}] \land \\
\text{portType}(pp) = \text{External}\}
\]

END;

/* This operation identifies a set of internal transitions that are related to each other. A transition is called internal iff it belongs to a capsule from inside the boundary and is fired by an incoming message over an internal port */

determineSetOfRelatedInternalTransitions =

BEGIN

\text{relatedTransitions} := \{ tt, ss \mid \\
\exists (mm, pp, pp1).
\}\n
\begin{align*}
& (pp1 \in \text{port} \land pp \in \text{port} \land mm \in \text{message} \land \\
& mm \mapsto pp \in \text{messageSentByTransition}[\{tt\}] \land \\
& mm \mapsto pp1 \in \text{messageRecievedByTransition}^{-1} [\{ss\}] \land \\
& pp \mapsto pp1: (\text{portConnectedToPort}; \text{Id}(\text{portInternal}))
\end{align*}

\text{sourcePointStateOfTransition}(tt) =

\text{targetPointStateOfTransition}(ss)

\}

\}
END;

/* This operation identifies a set of internal and external transitions that are related to each other. In this operation, we use the property constructChain in order to determine all the related transitions */

establishChainOfRelatedTransitions(mm) =

PRE

    mm ∈ message ∧
    mm ∈ dom(TransitionsTriggeredByMessage)
THEN

    chainOfRelatedTransitions(mm) :=
    constructChain(TransitionsTriggeredByMessage(mm),
                    relatedTransitions, ∅)
END;

/* This operation allows to identify a functional process */

determineFunctionalProcess(mm) =

PRE

    mm ∈ message ∧
    mm ∈ dom(TransitionsTriggeredByMessage)
THEN

    ANY ff WHERE ff ∈ FUNCTIONAL_PROCESS ∧ ff ∉ functionalProcess
    THEN
    messageTriggeringFunctionalProcess(mm) := ff ||
    functionalProcess := functionalProcess ∪ {ff}
    END
END
/* This operation allows to identify all the "entries" of a functional process */

determineEntriesOfFunctionalProcess(ff) =

PRE
   ff ∈ functionalProcess
THEN
   entry(ff) := { mm |
      mm ∈ messageTriggeringFunctionalProcess⁻¹ [{ff}] ∧
      ∃ tt. (tt ∈ TransitionsTriggeredByMessage(mm) )
   }
END;

/* This operation allows to identify all the "exits" of a functional process */

determineExitsOfFunctionalProcess(ff) =

PRE
   ff ∈ functionalProcess
THEN
   exit(ff) := { mm |
      mm ∈ messageTriggeringFunctionalProcess⁻¹ [{ff}] ∧
      ∃ (tt, pp). (tt ∈ chainOfRelatedTransitions(mm) ∧
      mm → pp ∈ messageSentByTransition[{tt}] ∧
      portType(pp) = External)
   }
END;
/* This operation allows to identify all the "reads" of a functional process */

determineReadsOfFunctionalProcess(ff) =

PRE

ff ∈ functionalProcess

THEN

read(ff) := { dd | dd ∈ DATA_CLASS ∧

((dd ∈ dataClass ∧ isDataGroup(dd) = True) ∨

dd ∈ simpleDataGroup) ∧

∃ (aa, cc, tt, mm). (aa ∈ attribute ∧

mm ∈ messageTriggeringFunctionalProcess⁻¹ {[ff]} ∧

tt ∈ chainOfRelatedTransitions(mm) ∧

cc ∈ action ∧

(aa ∈ attributeOfDataClass⁻¹ {[dd]} ∨

aa ∈ attributeOfSimpleDataGroup(dd) ) ∧

(cc = entryActionOfState(targetStateOfTransition(tt)) ∨

cc = exitActionOfState(sourceStateOfTransition(tt)) ∨

cc = actionOfTransition(tt) ∨

cc = guardOfTrigger(triggerOfTransition(tt))

) ∧

aa ∈ consults(cc)

}

END;
/* This operation allows to identify all the “writes” of a functional process */

determineWritesOfFunctionalProcess(ff) =

PRE

ff ∈ functionalProcess

THEN

write(ff) := { dd | dd ∈ DATA_CLASS ∧

(dd ∈ dataClass ∧ isDataGroup(dd) = True) ∨

dd ∈ simpleDataGroup) ∧

∃ (aa, cc, tt, mm). (aa ∈ attribute ∧

mm ∈ messageTriggeringFunctionalProcess⁻¹ {[ff]} ∧

tt ∈ chainOfRelatedTransitions(mm) ∧

cc ∈ action ∧

(aa ∈ attributeOfDataClass⁻¹ {[dd]} ∨

aa ∈ attributeOfSimpleDataGroup(dd) ) ∧

(cc = entryActionOfState(targetStateOfTransition(tt)) ∨

cc = exitActionOfState(sourceStateOfTransition(tt)) ∨

cc = actionOfTransition(tt) ∨

cc = guardOfTrigger(triggerOfTransition(tt))

) ∧

aa ∈ maintains(cc)

)

END;

/* This operation calculates the sum of values assigned to the entries, reads, writes, and exits of each functional process. The result, produced by this operation, represents
the functional size of the selected capsules */

\[ \text{number\_of\_FFP} \leftarrow \text{determineFFPNumberOfFunctionalProcess}(ff) = \]

\text{PRE}
\[
ff \in \text{functionalProcess}
\]

\text{THEN}
\[
\text{number\_of\_FFP} := \text{card(entry}(ff)) + \\
\text{card(exit}(ff)) + \\
\text{card(read}(ff)) + \\
\text{card(write}(ff))
\]

\text{END}

\text{END}

D.3 Architecture

Figure FIG. 10 shows the main components of \( \mu \text{ROSE} \). The main modules are:

- the XML Generator allows to extract the relevant information from a RTMDL file (RRRT file extension) to an XML file;

- the Object Constructor allows to instantiate all the objects that are required during the measurement process;

- the COSMIC-FFP Components Builder allows to identify data groups and the data attributes of each identified group. It also allows to identify functional processes by using data groups and identifying data movements of each functional process. Finally, it computes the functional software size using the identified data groups, data movements and functional processes.
D.4 C++ Syntax Analyzer of $\mu$ROSE

In this section, we provide the syntax analyzer algorithm following by the list of C++ operators treated by the analyzer.

D.4.1 Algorithm

This algorithm receives a piece of C++ code to analyze and a list of capsule attributes, that may occur in the code, as inputs. It produces a list of attributes that have been found in the code and the processing type of each attribute (there are four types: read, write, entry, and exit). The following is the pseudo-code of this algorithm.

SyntaxAnalyzer(Code_to_Analyze, List_Of_Attributes)
{
  (Begin of SyntaxAnalyzer)
  Create(List_Of_Treated_Attributes)
  For (Attribute in List_Of_Attributes) in List_Of_Attributes
  {
    End_of_Code_to_Analyze = False
    Current_Attribute_Read = False
    Current_Attribute_Write = False
  }
Current_Attribute_Entry = False
Current_Attribute_Exit = False

While ( ( Not_End_of_Code_to_Analyze ) OR
( Current_Attribute_Read AND Current_Attribute_Write AND
Current_Attribute_Entry AND Current_Attribute_Exit ) )
Do
{
  Find(Attribute, Code_to_Analyze)
(Finds the first occurrence of Attribute in Code_to_Analyze
starting from the current position of cursor and returns
Attribute_Is_Found_In_Code or not)
If Attribute_Is_Found_In_Code_to_Analyze
  Then
  {  
    If Attribute_Is_Valid
(Validates if the word found in code == Attribute or not)
    Then
    {
      Determine_Index_Of(Attribute, Code_to_Analyze)
( Returns Index_Of_Attribute)
      Analyze_Before_And_After(Attribute, Index_Of_Attribute)
      If Attribute_Appear_Before_Operator
        Then
        {
          Current_Attribute_Write = True
          Add_To_List_Of_Treated_Attributes(Attribute, Write)
        }
If Attribute_Appears_After_Operator
  Then
  {
    Current_Attribute_Read = True
    Add_To_List_Of_Treated_Attributes(Attribute, Read)
  }

If Attribute_Appears_After_Entry_Operator
  Then
  {
    Current_Attribute_Entry = True
    Add_To_List_Of_Treated_Attributes(Attribute, Entry)
  }

If Attribute_Appears_After.Exit_Operator
  Then
  {
    Current_Attribute.Exit = True
    Add_To_List_Of_Treated_Attributes(Attribute, Exit)
  }

} (End If of Attribute_Is_Valid)

} (End If of Attribute_Is_Found_In_Code_to_Analyze)

Else
  Then
  {
    End_of_Code_to_Analyze = True
  } (End If Else of Attribute_Is_Found_In_Code_to_Analyze)

} (End While)

} (End For)
D.4.2 C++ Operators

Most of C++ operators have been considered in the code analyser of µROSE in order to identify the processing types (read, write, entry, and/or exit) of capsule attributes in an RRRT model that may occur in a piece of code to analyze.

- Index: [ ]
- Projection: -> and .
- Post-increment and Post-decrement: x++, x-
- Pre-increment and Pre-decrement: ++x, -x, +x and -x
- Complementary: ~x
- Logical And: & &
- Logical Not: !
- Logical Or: ||
- Or exclusive: ^
- Or inclusive: |
- C++ conditional expression "?:": a?b:c
- Assignment: =
- Arithmetical operators: +, -, *, /, and % (modulo)
- Mathematical functions: sin, cos, tan, sqrt, pow, and log
- Comparison operators: <,<=, >, and >=
- Equality and inequality: == and !=
- Operator and assignment: +=, -=, etc.
- Input and output: cin and cout
- Selection statement: If (condition) statement Else statement, Switch (expression)
  { case v1:... }
- Iteration statement: *While* (condition), *For* (init; condition; increment), *Do* (statement) *While* (condition).

Furthermore, the *rtdata* (which returns the data associated with a message) and *send* (used for sending messages) are also considered in the syntax analyzer. Note that, *rtdata* and *send* are two reserved words of the RRR toolset.

**D.5 Class Models of $\mu$ROSE**
Annexe E

$\mu_c ROSE$ : cas d'études (Case Studies of $\mu_c ROSE$)

E.1 Experimental Evaluation of $\mu_c ROSE$

The best way to determine if $\mu_c ROSE$ accurately measures the functional software size is to use it in industrial practice. The functional software size computed by the tool should be compared with the actual functional software sizes for a large number of systems from a wide range of application areas (embedded systems, real-time systems, and management information systems). This is, however, outside the scope of the present work. We have developed our own specifications of a variety of systems which have been used to evaluate the functionalities of $\mu_c ROSE$. Detailed results of some of the case studies are given in the next section.

E.2 Test Cases

A variety of scenarios have been defined to evaluate the functionalities of $\mu_c ROSE$, ensure its correctness, and detect errors. We have prepared data so that conditions force
execution to evaluate the identification of:

- functional processes in several ways, depending on the condition of transitions in the state machine of an RRRT model. For the identification of functional process procedure, we have specified the following cases:
  
  - Scenario1: a functional process is composed of one transition;
  
  - Scenario2: a functional process is composed of one or more transitions belonging to the same capsule;
  
  - Scenario3: a functional process is composed of one or more transitions belonging to different capsules in the boundary;
  
  - Scenario4: a functional process must never contain a transition belonging to a capsule from outside the boundary;

- data groups within the boundary:
  
  - Scenario1: a data group is derived from attributes of capsules within the boundary;
  
  - Scenario2: a data group is derived from messages exchanged by capsules within the boundary. In this case, we have taken into account all the possible types of port (wired, not wired, and protected) and protocols (defined and predefined);
  
  - Scenario3: a data group is derived from classes used to define the types of capsules attributes;
  
  - Scenario4: a data group must never be derived from attributes, messages or classes related to a capsule from outside the boundary;

- data movements within each functional process. In this procedure, we have defined scenarios to ensure that:
  
  - Scenario1: there does not exist two data movements "read" ("write", "entry", or "exit") which refer to the same data group in one functional process;
- Scenario 2: the type classification of each data movement and the computing of numerical values assigned for each data movement of each functional process are correct.

E.3 Results of Some Case Studies

Table TAB. 10 provides the list of components for the measurements made with $\mu ROSE$. In this table, the first column denotes the different RRRT models produced by the Rational Rose toolset. Note that, the "IntegratingData" model is one of the Rational Rose toolset examples. The second column describes the selected capsules, representing the boundary of the system to be measured. The third column denotes the number Cfsu representing the functional size of each case of the selected capsules.

<table>
<thead>
<tr>
<th>Model</th>
<th>Selected Capsules</th>
<th>Cfsu (Total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory</td>
<td>Producer</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Consumer</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Controller</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Producer and Consumer</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Producer and Controller</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Consumer and Controller</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Producer, Consumer, and Controller</td>
<td>20</td>
</tr>
<tr>
<td>Manufacturing</td>
<td>Producer</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Consumer</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Controller</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Producer and Consumer</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Producer and Controller</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Consumer and Controller</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Producer, Consumer, and Controller</td>
<td>27</td>
</tr>
<tr>
<td>IntegratingData</td>
<td>Receiver</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Sender</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Receiver and Sender</td>
<td>11</td>
</tr>
</tbody>
</table>
E.4 A Simple Fast and Serious Measurement

In this section we report on how \( \mu \text{ROSE} \) works through a RRRRT model for measuring the number of COSMIC-FFP. We have the IntegratingData file which describes the design specification. This model is one of the RRRRT toolset samples. It is designed in order to show how the RRRRT toolset may integrate a new data class into a RRRRT model. The RRRRT toolset can generate source code for classes with a simple behavior, for example constructors and destructors. But it cannot generate code when a class is more complex and contains well-defined operations, because it does not know anything about the class descriptor such as attributes. Indeed, the IntegratingData model allows the toolset to generate a type descriptor for this kind of classes.

Recall that, three principal constructs are used in a RRRRT model to describe software architecture: capsule, port, and connector. Dynamic behavior is modeled by using state machines and protocols. A protocol specifies messages exchanged between ports over a connector to allow communication between capsules. A state machine specifies the internal behavior of a capsule, with a communication capability. This capability is achieved by explicitly introducing executable instructions in states and transitions of a state machine (or state diagram). These transitions may have an additional remote effect of sending messages, and therefore firing transitions in the same state diagram or in an other state diagram. Finally, a class in a RRRRT model is used to define the type of a capsule attribute.

The main capsules of the IntegratingData model are:

- **Receiver**: this capsule receives messages, which contain data of different types, and
- **Sender**: this capsule sends messages, which contain data of different types.

The measurement process starts by selecting a set of capsules that perform at the same level of abstraction based on the measurer’s judgement. To present the measurement
of the different COSMIC-FFP components, we consider some excerpts of Sender and Receiver capsules specification, as described into the IntegrationData file, provided in the rest of the section.

The number of COSMIC-FFP is based on the measurement of the following components: data group, attribute, functional process, and data movements (entry, exit, read, and write). In the following sections we provide examples associated to each of these components.

E.4.1 Measurement of data group

Data group based on a data class: A class is concerned with various characteristics such as attributes. In a RRRT model, a class is used to define the type of a capsule attribute. A class is considered as a data group if it is used as the attribute type of a selected capsule. In the IntegratingData model, there is a class called Sample1, which is defined outside the toolset and has three attributes: x, y, and z (see below the description of this class).

(1) (object Class "Sample1"
(2) ...
(3) class_attributes (list class_attribute_list
(4) (object ClassAttribute "x"
(5) quid "378D3CC000AE"
(6) type "int"
(7) visibility "Public")
(8) (object ClassAttribute "y"
(9) quid "378D3CBE00CD"
(10) type "char"
(11) visibility "Public")
(12) (object ClassAttribute "z"
(13) quid "378D3CBB02D1"
(14) type "char"
(15) visibility "Public")
(16) language "C++")

The integration of class Sample1 allows the RRRT toolset to generate a type descriptor for this class. As described below in lines 4 and 6, Sample1 has been used as the type of attribute sample1, which is an attribute of capsule Sender. In this case, Sample1 is considered as a data group by $\mu_cROSE$.

(1) (object Capsule "Sender"
(2)   attributes (list Attribute_Set)
(3)   ... 
(4)   (object ClassAttribute "sample1"
(5)   quid "384D3F8D03B6"
(6)   type "Sample1")
(7)   (object ClassAttribute "payload2"
(8)   quid "378CF60F00DD"
(9)   type "int"
(10) visibility "Public")
(11) ...

Data group based on capsule attributes: The capsule Receiver contains two attributes: receiverTest and Buff (see below lines 13 and 17 in the description of Receiver). Since RTBoolean is a predefined data type provided by the RRRT toolset and used as the type of these attributes, $\mu_cROSE$ has established a data group called VSimple_Receiver. VSimple_Receiver contains two attributes: receiverTest and Buff.

(1) (object Capsule "Receiver"
(2)   ...
(3) (object Port "txRx"
(4) classifier "Logical View::Transmitters::TxRxProtocol"
(5) quid "373ADFE0297"
(6) isConjugated TRUE
(7) isRelay FALSE))
(8) ...
(9) class_attributes (list class_attribute_list
(10) (object ClassAttribute "receiverTest"
(11) quid "3A661700014"
(12) initv "FALSE"
(13) type "RTBoolean")
(14) (object ClassAttribute "Buff"
(15) quid "3B0B0AD40303"
(16) initv "FALSE"
(17) type "RTBoolean")
(18) ...

Data group based on a message: In the IntegratingData model, port txRx from Receiver refers to protocol TxRxProtocol, as described in lines 3 and 4 of the previous description associated to Receiver. For showing how $\mathcal{M}_{\text{ROSE}}$ identifies a data group based on a message, we suppose that capsule Receiver is the only selected capsule to be measured. In this case, each message that is specified within protocol TxRxProtocol will be considered as a data group during the measurement processing of $\mathcal{M}_{\text{ROSE}}$. For instance, message payloadSig, specified below in line 5, is a data group. Of course, sample1Sig, sample2Sig, pointerIntsSig, stringSig, and vecSigare are also data groups.

(1) (object Protocol "TxRxProtocol"

172
(2) ... 
(3) (object ProtocolRole "base"
(4) outSignals (list outSignalsList
(5) (object Signal "payloadSig"
(6) dataType "Logical View::SampleClasses::Payload"
(7) quidu "36D1D29B0148")
(8) (object Signal "sample1Sig"
(9) dataType "Logical View::SampleClasses::Sample1"
(10) quidu "378D3CA00169")
(11) (object Signal "sample2Sig"
(12) dataType "Logical View::SampleClasses::Sample2"
(13) quidu "378D4CB8035D")
(14) (object Signal "pointerIntsSig"
(15) dataType "Logical View::SampleClasses::PointerInts"
(16) quidu "378CE53402A2")
(17) (object Signal "stringSig"
(18) dataType "Logical View::SampleClasses::string"
(19) quidu "383D683B0367")
(20) (object Signal "vecSig"
(21) dataType "Logical View::SampleClasses::vector_string"
(22) quidu "383D665B0358")

E.4.2 Measurement of functional process

Before providing an example of how $\mu$ROSE works on a functional process, we will first explain the notion of related transitions. Suppose that $t_1$ is a state transition that sends a message $m$ during its execution and $m$ fires a state transition, $t_2$, from the state diagram of $t_1$ or an other state diagram of a selected capsule. In this case, $t_1$ and $t_2$ are
considered as related transitions and will belong to the same functional process. Also, $t_1$ and $t_2$ are considered as related transitions if they belong to the same group of transition segments, for instance the group of transition segments of a choice point state.

**Functional process based on one simple transition:** We have, here, transition $\text{sample1}$ from state diagram Receiver. This transition allows receiving data transition $\text{sample1}$ and will be fired by the arrival of message $\text{sample1Sig}$, as described below in line 14. Because it is fired by $\text{sample1Sig}$ incoming from outside the boundary (represented by Receiver) and it is not related to any transition, transition $\text{sample1}$ is a functional process.

(1) (object Trans "sample1"
(2) source "TOP:Ready:Junction4"
(3) target "TOP:Ready:Junction5"
(4) action (object Action
(5) quid "3E208C6F010F"
(6) body
(7) myData = *(const Sample1 *) getMsg() ->rtdata; )
(8) eventGuards (list eventGuardsList
(9) (object EventGuard
(10) Event (object PortEvent
(11) ports (list portsList
(12) ports "Structure:txRx")
(13) signals (list signalsList
(14) signals "base:sample1Sig")))))

**Functional process based on a chain of related transitions:** Suppose that Sender and Receiver have been selected to be measured. As described below in lines
8 to 14, transition initial from Sender creates and sends a set of messages containing a variety of data types when it will be triggered by the arrival of message timelog (see line 21). The code associated to Initial sends messages payloadSig, sample1Sig, sample2Sig, name, and vecSig from Receiver through port txRx during its execution. These messages fire the execution of transitions payload, sample1, sample2, pointerInts, string, and vector_string from Receiver, respectively. That Initial is fired by a message referring to time (timelog) and is related to payload, sample1, sample2, pointerInts, string, and vector_string. In summary, we have a functional process that is composed of all these transitions including Initial from Sender. μROSE assigns a name to this functional process by concatenating the transition names such as Initialpayloadsampl...
A data movement based on a message: functional process Initialpayloadsamp-
le1sample2pointerntstringvector_string has timelog as an “entry”, since time-
log triggers the execution of transition Initial (which triggers the execution of the
other transitions) and refers to time. It also has message sample1Sig as a “read”. Indeed,
message sample1Sig, firing transition sample1, is referred in the code associated to this
transition by using rtdata (see line 7 in the previous description of transition sample1),
which is a RRRT reserved word representing the value of the sample1Sig message.

Sub-process based on capsule attributes: Remember that VSimple_Sender is
considered as a data group and contains attribute payload2 (Sender has at least one
attribute, payload2, that is typed as integer). The $\mu$Rose syntax analyzer is able
to count VSimple_Sender as a “write” because payload2 occurs before the assignment
sign “=” in the code associated to the Initial transition (see above line 6). It is also
able to count VSimple_Sender as a “read” since the code associated to Initial contains
an executable instruction that refers to payload2 (see above line 9).

Finally, it should be noted that this example does not describe all the possible measure-
ment cases of the IntegratingData model. For the sake of simplicity, some cases have
been omitted. It was just to give a simple idea about the measurement processing of
$\mu$Rose.