Services and Business Process Reengineering

Novarun Deb
Nabendu Chaki

Business Standard Compliance and Requirements Validation Using Goal Models

Services and Business Process Reengineering

Series Editors

Nabendu Chaki, Department of Computer Science and Engineering, University of Calcutta, Kolkata, India
Agostino Cortesi, DAIS, Ca' Foscari University, Venice, Italy

The book series aims at bringing together valuable and novel scientific contributions that address the critical issues of software services and business processes reengineering, providing innovative ideas, methodologies, technologies and platforms that have an impact in this diverse and fast-changing research community in academia and industry.

The areas to be covered are

® Service Design
® Deployment of Services on Cloud and Edge Computing Platform
® Web Services
® IoT Services
® Requirements Engineering for Software Services
® Privacy in Software Services
® Business Process Management
® Business Process Redesign
® Software Design and Process Autonomy
® Security as a Service
® IoT Services and Privacy
® Business Analytics and Autonomic Software Management
® Service Reengineering
® Business Applications and Service Planning
® Policy Based Software Development
® Software Analysis and Verification
® Enterprise Architecture

The series serves as a qualified repository for collecting and promoting state-of-the art research trends in the broad area of software services and business processes reengineering in the context of enterprise scenarios. The series will include monographs, edited volumes and selected proceedings.

More information about this series at http://www.springer.com/series/16135

Novarun Deb · Nabendu Chaki

Business Standard Compliance and Requirements Validation Using Goal Models

Springer

Novarun Deb
Department of Environmental Science,
Informatics and Statistics
Ca' Foscari University
Venice, Italy

Nabendu Chaki
Department of Computer Science
and Engineering
University of Calcutta
Kolkata, India

ISSN 2524-5503 ISSN 2524-5511 (electronic)
Services and Business Process Reengineering
ISBN 978-981-15-2500-1 ISBN 978-981-15-2501-8 (eBook)
https://doi.org/10.1007/978-981-15-2501-8

© Springer Nature Singapore Pte Ltd. 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

You have to fight to reach your dream. You have to sacrifice and work hard for it. Sometimes you have to accept you can't win all the time. There are more important things in life than winning or losing.

—Lionel Andrés Messi

To my grandmother Kalpana Dutta,
my father Parimal Deb and my mother Sapna Deb, who could not complete her own Ph.D. due to filial responsibilities.
—Novarun Deb

To Prof. Swapan Bhattacharya, my mentor.
—Nabendu Chaki

Preface

My research career under the guidance and supervision of my co-author, Prof. Nabendu Chaki, began in the latter half of 2009. He was the supervisor for my master's (MS) thesis. At that time, I was eager to explore the domain of intrusion detection mechanisms for wireless ad-hoc network security. Even during my M.Tech., we continued to work on intrusion detection algorithms for wireless ad-hoc networks. This research thrust resulted in conference and journal publications as well as some book chapters. All this boosted my research interests and it grew beyond my masters' theses.

However, after completing my M.Tech. in 2012, when I started working on my Ph.D., I decided to take a holistic approach on security of systems. What I concluded from my study of the state-of-the-art and industry practices at that time was that *Security* is a multi-dimensional objective. Intrusion detection was just one of these dimensions. Security can be thought of as a chain having multiple links, and the security solution for a system or enterprise is only as strong as the weakest link in that chain. So the current industry standards were *Security Compliance* rather than *Security* as a whole. Enterprises have certain standards for each security dimension. A 100% compliance to each of these security standards is more important than a 100% secure solution.

In my efforts to incorporate *Security* within the requirements of a system-to-be, I started exploring the domain of goal-oriented requirements engineering. Prof. Aditya Ghose at the University of Wollongong, Australia, has been the joint supervisor of my Ph.D. thesis. He suggested that non-functional requirements like *Security* can be represented within goal models with softgoals and Softgoal Interdependency Graphs (SIGs). However, since softgoals open up a whole new can of worms which needs to be handled with caution and a different level of expertise, we consciously decided to focus on functional requirements of a system (only) for my Ph.D. This book is mainly focused on the enterprise modelling and requirements analysis techniques for functional requirements that have been proposed during my six years of doctoral research.

The book also goes beyond the works documented in my Ph.D. thesis and gives an insight to the reader on how the proposed algorithms, architectures, and frameworks, can be developed into tools and extended for future research. Chapter 6 documents some of these works that have been done during my post-doctoral research period. These works address the future research directions that were mentioned in my Ph.D. thesis. Of particular importance is the GRL2APK framework on which we are currently working. We are trying to develop an optimized version of this framework that tries to resolve Non-Functional Requirement (NFR) conflicts and generate an optimal solution based on developer choices and priorities. This requires exploring how different NFRs—like *Security, Privacy, Efficiency, etc.*—interact and conflict with each other. We would like to thank Souvik Das (a Ph.D. scholar working under Prof. Chaki at the Department of Computer Science and Engineering, University of Calcutta) for helping us in building the GRL2APK tool.

Thus, in conclusion, I would like to mention that I began my research career with *Security* (as a defence mechanism) and currently I am again working with *Security* (as a non-functional requirement). This is, I guess, what they say…. *The Circle of Life!!!*

Venice, Italy Novarun Deb
November 2019

Contents

1 Introduction . . . 1
1.1 The i* Modelling Notation . . . 2
1.1.1 Case Study: Healthcare . . . 3
1.2 Research Directions . . . 4
1.2.1 Goal Model Hierarchies (RQ-1) . . . 6
1.2.2 Goal Model Checking (RQ-2) . . . 7
1.2.3 Semantic Analysis of Goal Models (RQ-3) . . . 7
1.3 Analysis of Results . . . 8
1.4 Organization of the Book . . . 9
References . . . 9

2 State-of-the-Art . . . 11
2.1 Formal Requirements Engineering Techniques . . . 12
2.2 Requirement Refinement Hierarchies . . . 13
2.3 Model Checking with i* . . . 15
2.4 Semantic Annotations of Goal Models . . . 17
References . . . 19

3 i* and Enterprise Hierarchies . . . 23
3.1 Hierarchic Correlations . . . 28
3.2 Relative Completeness Checking . . . 33
3.2.1 Consequence of Relative Completeness . . . 35
3.3 Possible Heuristics . . . 35
3.3.1 Formalizing the Heuristics . . . 38
3.3.2 Applying Heuristics for Relative Completeness Checking . . . 40
3.3.3 Results . . . 42
3.4 Conclusion . . . 43
References . . . 44

4 Model Checking with i* . . . 45
4.1 Developing Finite State Models from an i* Model . . . 47
4.1.1 The Naïve Algorithm . . . 48
4.1.2 The Semantic Implosion Algorithm (SIA) . . . 54
4.1.3 Soundness and Completeness . . . 61
4.2 Complexity Analysis . . . 62
4.2.1 Actor Internal Analytics . . . 63
4.2.2 Inter-Actor Analytics . . . 65
4.2.3 SIA Analytics . . . 69
4.3 The i*ToNuSMV Tool . . . 71
4.3.1 i*ToNuSMV Input . . . 71
4.3.2 The Preprocessing Module . . . 73
4.3.3 The Model Transformation Module . . . 73
4.3.4 The Mapper Module . . . 73
4.3.5 i*ToNuSMV Output . . . 74
4.3.6 The i*ToNuSMV Algorithm . . . 76
4.3.7 Platforms Used . . . 76
4.3.8 Application Scenario . . . 76
4.4 i*ToNuSMV Version Manager . . . 77
4.5 Contact and URL . . . 77
4.6 Conclusion . . . 78
References . . . 79

5 Goal Model Maintenance . . . 81
5.1 Semantic Reconciliation . . . 83
5.1.1 ORGMod Extraction . . . 88
5.1.2 Semantic Reconciliation Operators . . . 94
5.1.3 Illustrative Examples . . . 99
5.2 Resolving Conflicts Using Model Refactoring . . . 104
5.2.1 Entailment Issues . . . 104
5.2.2 Consistency Issues . . . 109
5.3 An Implementation Roadmap . . . 113
5.3.1 The Generalized Framework . . . 113
5.3.2 Taxonomy of Goal Model Proximity Measures . . . 117
5.3.3 Evaluating Goal Model Proximity . . . 117
5.4 Using AFSR on the i* Framework . . . 119
5.4.1 Dependency Reconciliation Operator . . . 119
5.4.2 Implementation Roadmap for i* . . . 121
5.5 Experimental Evaluation . . . 124
5.5.1 Indicators and Drivers . . . 124
5.5.2 Experimental Preliminaries . . . 124
5.5.3 Process and Results . . . 125

5.6 Conclusion 128
References 130

6 Conclusion and Future Work 131
6.1 Summary of the Work 131
6.2 Future Research Directions 132
6.2.1 Extracting Business Compliant Finite State Models 133
6.2.2 The CARGo Tool 140
6.2.3 Building Mobile Applications from Goal Model Specifications 143
References 152

Abbreviations

AFSR	Annotation of Functional Semantics and their Reconciliation
AI	Artificial Intelligence
AoURN	Aspect-oriented User Requirements Notation
AT	Activity Theory
BIDE	BI-Directional Extension
BPIC	Business Process Intelligence Challenge
BPMN	Business Process Modelling Notation
BRC	Bi-directional Relative Completeness
CCA	Cloud Component Approach
CNF	Created Not Fulfilled
CRA	Consistency Resolution Algorithm
CTL	Computational Tree Logic
DDL	Dependency Links
DSO	Decomposition Sequence Objects
E-FSM	Extended Finite State Models
ERA	Entailment Resolution Algorithm
FSM	Finite State Model/Machine
FSMS	Finite State Model Space
FSMS-A	Finite State Model Space for the actor set A
FSMS-N	Finite State Model Space for the Naïve Algorithm
FSMS-S	Finite State Model Space for the Semantic Implosion Algorithm
FU	Fulfilled
GBRAM	Goal-Based Requirements Analysis Method
GORE	Goal-Oriented Requirements Engineering
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
KAOS	Knowledge Acquisition in autOmated Specification
LTL	Linear Temporal Logic
MDSE	Model Driven Service Engineering
MEL	Means End Links

NA	Naïve Algorithm
NC	Not Created
NFR	Non-Functional Requirements
NL	No Links
NuSMV	New Symbolic Model Verifier
OCF	Optional Condition Formulae
OCL	Only Child Links
OO-SPL	Object-Oriented Software Product Line
OPL	Only Parent Links
ORGMod	OR-Refined Goal Models
OWL	Ontology Web Language
PCL	Parent and Child Links
PCTk	Process Compliance Toolkit
PoC	Proof-of-Concept
RAM	Reusable Aspect Models
RE	Requirements Engineering
RI	Relative Incompleteness
RML	Requirements Modelling Language
RSML	Requirements State Machine Language
SADT	Structured Analysis and Design Technique
SCR	Software Cost Reduction
SD	Strategic Dependency
SIA	Semantic Implosion Algorithm
SPMF	Sequential Pattern Mining Framework
SR	Strategic Rationale
SRA	Semantic Reconciliation Algorithm
TDL	Task Decomposition Links
t-GRL	Textual Goal Requirements Language
UML	Unified Modelling Language
URC	Unidirectional Relative Completeness

Notations

Generic

$\boldsymbol{A_i}$	Actor
$\boldsymbol{G_i}$	Goal
$\boldsymbol{T_i}$	Task
$\boldsymbol{R_i}$	Resource

Chapter 3

$\boldsymbol{n}$	Number of levels in the requirement refinement hierarchy
$\boldsymbol{L_i}$	Ontology used in level-*i* of the hierarchy
$\boldsymbol{ME(\{G_1, G_2, \ldots, G_k\}, G)}$	Goal G connected to goals G_1, G_2, ..., G_k using Means-End links
$\boldsymbol{TD(\{G_1, G_2, \ldots, G_k\}, G)}$	Goal G connected to goals G_1, G_2, ..., G_k using Task Decomposition links
$\boldsymbol{MEL_A^i}$	Means End Links in actor A at level-*i*
$\boldsymbol{TDL_A^i}$	Task Decomposition Links in actor A at level-*i*
$\boldsymbol{DD(A, B)}$	Dependency between a depender A and dependee B
$G_j^i \leftrightarrow \boldsymbol{G_k^{i+1}}$	1-1 correlation between goal G_j at level-*i* and goal G_k at level-(i + 1)
$\boldsymbol{G_j^i \leftrightarrow G_{k1}^{i+1}, G_{k2}^{i+1}, \ldots, G_{kn}^{i+1}}$	1-Many correlation between goal G_j at level-*i* and goals $G_{k1}, G_{k2}, \cdots, G_{kn}$ at level-(i + 1)
$\boldsymbol{I_k^*}$	i* model at level-*k*
$\boldsymbol{R[i^*(k)]}$	Set of requirements captured by the i* model at level-*k*

$\lvert \boldsymbol{I}_k^* \rvert$	Number of model elements in the i* model at levels k
OC#i	1-1 correlation
$\boldsymbol{OC\#i}^{-1}$	Inverse 1-1 correlation
HR#i	1-Many correlation
$\boldsymbol{HR\#i}^{-1}$	Inverse 1-Many correlation
$\lvert \boldsymbol{HR\#i} \rvert$	Number of model elements in I_{k+1}^* that have a 1-Many correlation with some model element in I_k^*
$\langle \boldsymbol{T}_i, \boldsymbol{T}_{i+1} \rangle$	Contiguous events in the level-x event log
$\boldsymbol{T}_{ij}$	j-th event in the level-$(x+1)$ event log that is related to event T_i in the level-x event log
$\langle\langle \boldsymbol{T}_{i1}, \ldots, \boldsymbol{T}_{in} \rangle, \boldsymbol{T}_i \rangle$	A joined log representing a sequential pattern of interest
$\boldsymbol{min}_{support}$	Threshold provided to the BIDE+ pattern miner

Chapter 4

$\boldsymbol{G}_{(N)}$	Goal in the Not Created state
$\boldsymbol{G}_{(C)}$	Goal in the Created Not Fulfilled state
$\boldsymbol{G}_{(F)}$	Goal in the Fulfilled state
$\boldsymbol{G}_{SS}$	State Sequence Graph
$\bar{\boldsymbol{G}}_i$	Goal in either of three states NC, CNF, or FU
$(\bar{\boldsymbol{G}}_1, \bar{\boldsymbol{G}}_2, \ldots, \bar{\boldsymbol{G}}_n)$	Vertex label in G_{SS}
$(\boldsymbol{G}_{1(N)}, \boldsymbol{G}_{2(N)}, \ldots, \boldsymbol{G}_{n(N)})$	Source vertex of G_{SS}
$(\boldsymbol{G}_{1(F)}, \boldsymbol{G}_{2(F)}, \ldots, \ldots \boldsymbol{G}_{n(F)})$	Sink vertex of G_{SS}
$\boldsymbol{L}_P$	Number of paths from source vertex to sink vertex in G_{SS}
$\boldsymbol{V}_i = \boldsymbol{0}$	Goal G_i in state NC
$\boldsymbol{V}_i = \boldsymbol{1}$	Goal G_i in state CNF
$\boldsymbol{V}_i = \boldsymbol{2}$	Goal G_i in state FU
$\boldsymbol{V}_i: \boldsymbol{0} \rightarrow \boldsymbol{1}$	Goal G_i making a transition from $G_{i(N)}$ to $G_{i(C)}$
$\boldsymbol{V}_i: \boldsymbol{1} \rightarrow \boldsymbol{2}$	Goal G_i making a transition from $G_{i(C)}$ to $G_{i(F)}$
$\boldsymbol{\#Seq}_i$	Number of finite state sequences for the i-th task decomposition within an actor
$\boldsymbol{S}_i$	Size of the finite state model space for actor A_i

Chapter 5

$\langle\langle \boldsymbol{G_i, G_j} \rangle, [\boldsymbol{G_i, G_k}] \rangle$	Path label for OR-decompositions
$\langle\langle \boldsymbol{G_i}, \{\langle \boldsymbol{G_j} \rangle, \langle \boldsymbol{G_k} \rangle\}\rangle, [\varnothing] \rangle$	Path label for AND-decompositions
$\boldsymbol{G}_{i(X)}$	Goal G_i undergoes OR-decomposition
$\boldsymbol{G}_{i(A)}$	Goal G_i undergoes AND-decomposition
IE(G)	Immediate semantic annotations provided for goal G
CE(G)	Cumulative semantic annotations derived for goal G
rec(G_i, G_j)	Semantic reconciliation operator between parent G_i and child G_j
ANDrec$(\boldsymbol{G_i, G_j, G_k})$	Semantic reconciliation operator for AND-decompositions
ORrec$(\boldsymbol{G_i, G_j, G_k})$	Semantic reconciliation operator for OR-decompositions
DEPrec$(\boldsymbol{G_i})$	Semantic reconciliation operator for dependences
$\mathbb{D}$	Deficiency List for a goal with entailment conflict
$\mathbb{A}(d_i)$	Availability function providing the paths for the deficient semantic d_i
α	Set of change constraints for the goal model maintenance problem
$<_G$	Goal model proximity operator
∇_G	Goal model distance measurement operator

List of Figures

Fig. 1.1 SD model of a medical insurance enterprise 3
Fig. 1.2 SR model of the claims manager actor within the medical insurance enterprise. 4
Fig. 3.1 Requirement specifications modelled by the university at abstract higher levels of the requirement refinement hierarchy 25
Fig. 3.2 Requirement specifications modelled by individual departments at the lower levels of the requirement refinement hierarchy 26
Fig. 3.3 Hierarchic correlation in an Actor Invariant refinement hierarchy 29
Fig. 3.4 Hierarchic OCL goal correlation in an Actor Invariant refinement hierarchy 30
Fig. 3.5 Hierarchic OPL goal correlation in an Actor Invariant refinement hierarchy 31
Fig. 3.6 Hierarchic PCL task correlation in an Actor Invariant refinement hierarchy 31
Fig. 3.7 Possible Hierarchic correlation in an Actor Variant refinement hierarchy 32
Fig. 3.8 1-Many correlation between lower granular goal *Science Courses* and higher granular goals *Pure Science Courses* and *Engineering Science Courses* 37
Fig. 3.9 The i* model obtained for the lower levels of the requirement refinement hierarchy after incorporating the 1-Many correlation 38
Fig. 4.1 Block diagram of standard model verifiers 46
Fig. 4.2 Problem with i* model verification 46
Fig. 4.3 **a** Actor *A* with a single goal *G*; **b** The only semantically correct finite state sequence; **c–g** Other possible finite state sequences that can be derived by permuting the state space but which are semantically incorrect. 49

Fig. 4.4 **a** Actors A_1 and A_2 with goals G_1 and G_2, respectively; **b** The State Sequence Graph over the set of $3^2 = 9$ possible states . . . 49
Fig. 4.5 Graph depicting the rate of growth of the state space and finite state model space with respect to the number of model elements in the i* model for the Naïve Algorithm . . . 53
Fig. 4.6 **a** Actor A_1 with goal G_1; **b** The corresponding finite state model . . . 55
Fig. 4.7 **a** Actor A_1 with goals G_1, G_2 and G_3 connected through a task decomposition; **b** The corresponding set of all possible finite state models captured in a state sequence graph. . . . 56
Fig. 4.8 **a** Actor A_1 with goals G_1, G_2, G_3 and G_4 connected through a means-end decomposition; **b** The corresponding finite state model . . . 58
Fig. 4.9 The state sequence graph corresponding to a nested decomposition. A higher level means-end decomposition contains another means-end decomposition along the leftmost link and a task decomposition along the rightmost link . . . 59
Fig. 4.10 **a** Goal G_3 of actor A_1 dependant on Goal G_4 of actor A_2; **b** Temporary transition from $G_{3(C)}$ to $G_{3(F)}$ introduced; **c** Resolution of the dependency by replacing the temporary transition with two permanent transitions . . . 60
Fig. 4.11 Behaviour analysis with respect to the finite state model space of individual actors for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the number of model elements in the i* model varies. . . . 64
Fig. 4.12 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the density of actors in the i* model varies . . . 67
Fig. 4.13 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the distribution of model elements within actors in the i* model varies . . . 69
Fig. 4.14 Behaviour analysis of the *Semantic Implosion Algorithm* (w.r.t. the finite state model space) as the distribution of model elements within actors and the actor density in the i* model are both varied . . . 70
Fig. 4.15 The *i**ToNuSMV* tool . . . 72
Fig. 4.16 An i* model of a single actor *Doctor* . . . 72
Fig. 5.1 Goal model of an existing healthcare enterprise . . . 84
Fig. 5.2 An OR-refined Goal Model highlighted within the goal model . . . 87

Fig. 5.3 Goal models illustrating the two different types of decompositions or splits that artefacts can undergo. 88
Fig. 5.4 Decomposition sequence segmentation of the path list for the goal model in Fig. 5.2. 90
Fig. 5.5 Decomposition Sequence Objects (DSO) derived from the decomposition sequences in Fig. 5.4. 90
Fig. 5.6 The four possible *ORGMods* that have been derived for goal G_1, highlighted within the goal model of actor A_i 94
Fig. 5.7 Two *ORGMod*s highlighted for goal G_1 which undergoes a *OR-decomposition* within the goal model of actor A_i. 99
Fig. 5.8 An illustrative example showing how semantic reconciliation can be used to detect problems in *entailment* although *consistency* is ensured 100
Fig. 5.9 An example showing a *consistency* conflict between the immediate annotations of parent goal G and the cumulative annotations of child goal G_2 101
Fig. 5.10 An example showing how *entailment* and *consistency* are both satisfied. 103
Fig. 5.11 A sample goal model showing failure of *entailment* at goal G that undergoes OR-decomposition. 106
Fig. 5.12 Temporary high-level goals GT_1 and GT_2 are used to merge goals G_1 and G_2 with the temporary goal CT_1. 107
Fig. 5.13 A sample goal model showing failure of *entailment* at goal G that undergoes AND-decomposition 107
Fig. 5.14 Temporary goal CT_1 is merged with goals G_1 and G_2 to obtain the cumulative semantic annotation of G 108
Fig. 5.15 *Hierarchic* inconsistency at goal G arising out of the immediate satisfaction condition q of G and the cumulative satisfaction condition $\neg q$ of goal G_1 109
Fig. 5.16 Eliminating inconsistencies in the semantic reconciliation process governed by Eq. 5.2 in Theorem 5.1 110
Fig. 5.17 *Sibling* inconsistency at goal G arising out of the immediate satisfaction condition r of goal G_1 and the immediate satisfaction condition $\neg r$ of goal G_2 110
Fig. 5.18 `Solution 1`: eliminates the effect annotation r of goal G_1. `Solution 2`: eliminates the effect annotation $\neg r$ of goal G_2 111
Fig. 5.19 Modified goal model incorporating the two business environment changes `Change-1` and `Change-2` 114
Fig. 5.20 An *ORGMod* that extends beyond the boundary of actor A_i as the resource R_{i1} depends on task T_{j2} of actor A_j. 120
Fig. 5.21 Variation of execution time for random generation of annotated goal models 126

Fig. 5.22 Number of calls to the semantic reconciliation operators (ERA and CRA) for randomly generated goal models with $n_{eff} = 10$ . . . 127
Fig. 5.23 Number of calls to the effect reconciliation operators (ERA and CRA) for varying values of the drivers *level* and n_{eff} . . . 129
Fig. 6.1 The workflow of the *i*ToNuSMV*3.0 deployment framework. . . . 135
Fig. 6.2 A simple goal model for accessing a locker . . . 135
Fig. 6.3 FSM generated by *i*ToNuSMV 2.02* . . . 136
Fig. 6.4 FSM generated by *i*ToNuSMV 3.0* . . . 137
Fig. 6.5 Number of state transitions in the final FSM . . . 138
Fig. 6.6 Execution time for deriving the final FSMs . . . 138
Fig. 6.7 *tree_node* structure . . . 140
Fig. 6.8 Abstracted view of an example SAi* Net . . . 141
Fig. 6.9 The proposed framework for app orchestration from goal models using selective composition of NFRs . . . 144
Fig. 6.10 The implementation framework with the process steps numbered in black circles . . . 146
Fig. 6.11 Conflict identified between operationalization PPM (for Data-space performance) and DES_Encryption (for security) . . . 150
Fig. 6.12 Workflow generated for LZ (for Data-space performance) and AES_Encryption (for security) . . . 150
Fig. 6.13 Screenshot of the app . . . 151
Fig. 6.14 Screenshot of patient database . . . 152

List of Tables

Table 1.1 List of existential compliance rules 5
Table 4.1 Rate of growth of space w.r.t. the number of model elements 53
Table 4.2 Actor internal analytics 64
Table 4.3 Inter-actor analytics obtained by varying actor density. 67
Table 4.4 Inter-actor analytics obtained by varying the distribution of goals 68
Table 4.5 Inter-actor analytics obtained by varying both actor density and distribution of goals for the *Semantic Implosion Algorithm*. 70
Table 4.6 State variable listing of entities 74
Table 4.7 Finite state model recorded in *STT.opm* 75
Table 5.1 List of primitive goal modification operators 115
Table 5.2 List of compound goal modification operators 122
Table 6.1 Feature comparison between verions 2.02 and 3.0 139

Chapter 1
Introduction

Any software or product goes through a development life cycle. Once the product is launched it undergoes rigorous maintenance as long as it is alive and does not become obsolete. Developers are constrained to minimize the cost and risks associated with bugs (or errors) that may be detected during the development or maintenance lifecycles [1, 2]. These errors can range from simple logistics to complex non-compliance issues.

The cost of rectifying errors in the later phases of the development lifecycle has a direct impact in hiking the cost. The requirement analysis phase is the most critical phase in the development and maintenance of a system and accounts for almost 95% of the errors that are detected in the later phases [3, 4]. The current practice is that the developer and client sign agreements after finalizing the requirements. Any failures that occur later are resolved by delegation.

The main motivation behind this book is to help designers and developers identify and rectify errors in the requirements phase itself, before the requirements are formally documented and specified. Goal Modelling techniques can be used to identify and detect errors, conflicts, or issues that may arise in the later phases of the lifecycle. Early detection helps in reducing the cost by a great extent. In this book, we explore different scenarios where goal modelling techniques are yet to be deployed. We also propose techniques to leverage greater benefits from goal models by extending their analytical capabilities. The correctness and the efficiency of the different methodologies are established through theoretical and mathematical analysis, simulation and developing proof-of-concepts (PoCs).

The rest of this chapter is organized as follows. We begin with a brief introduction of the i* modelling framework in Sect. 1.1 that was proposed in [1]. We present a case study of the healthcare enterprise and how i* modelling constructs can be derived for individual requirements [5]. This is followed by Sect. 1.2 that presents the research directions that will be explored in this book. Section 1.3 presents an overview of the findings in this research. The chapter concludes with an organization of this book in Sect. 1.4.

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_1

1.1 The i* Modelling Notation

The i* framework has two main components—the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. The SD model is used to model actors and their inter-dependencies. Both actors and their inter-dependencies are said to be *intentional* in the i* framework. Every actor has its own set of abilities, goals and assumptions. Dependency between actors increases the vulnerability of the depender on the dependee. Actors represent the stakeholders of the enterprise as well as agents of the system being designed. Actors are also *strategic* in the sense that they optimize their risks and opportunities in order to achieve their ultimate objectives.

Actors can be classified into three types—agents, roles, and positions. Agents represent real-life humans or systems with unique capabilities. A role is an "abstract actor embodying expectations and responsibilities" [6]. An agent of the system can play multiple roles in different social contexts. Inter-actor dependencies are also said to be *intentional* if they result from an agent's strategy to achieve some objective. i* supports four different types of dependencies—goal, task, resource, and softgoal.

The SD model is used to represent actors and how they depend on each other. The dependencies capture the intentions of individual processes within the organization. Every dependency represents an agreement between the depender and the dependee. The type of dependency defined the nature of this agreement. Goal dependencies represent the delegation of responsibility to the dependee for fulfilling some goal of the depender. Softgoal dependencies are similar to goal dependencies except that their fulfillment cannot be guaranteed. Task dependencies represent some activity that the dependee must do for the depender. Resource dependencies are used to capture the provisioning of a resource to the depender by the dependee. All other finer granularities of the enterprise actors are abstracted. This model is used in the late requirements phase and helps in performing vulnerability analysis of individual actors. Figure 1.1 shows one such SD model that has been illustrated in [1].

The SR model provides a lower level more detailed rationale behind the objectives of individual actors and how they intend to achieve them. It uses the concepts of goals, tasks, resources, and softgoals to analyse the internal processes of each actor as well as all the alternatives. Goals, tasks and resources can be related to subgoals, subtasks, or softgoals using means-end links or task decomposition links. These links can be used to generate AND/OR refinement hierarchies. Goals, tasks and softgoals can be related to higher level softgoals using contribution links that are labelled with positive or negative contributions. Softgoals are used to capture the non-functional requirements of the system. The SR model is also strategic in the sense that model elements are included only if they effect the achievement of the objectives of individual actors. Figure 1.2 shows the SR model of the Claims Manager actor [1] within the Medical Insurance enterprise.

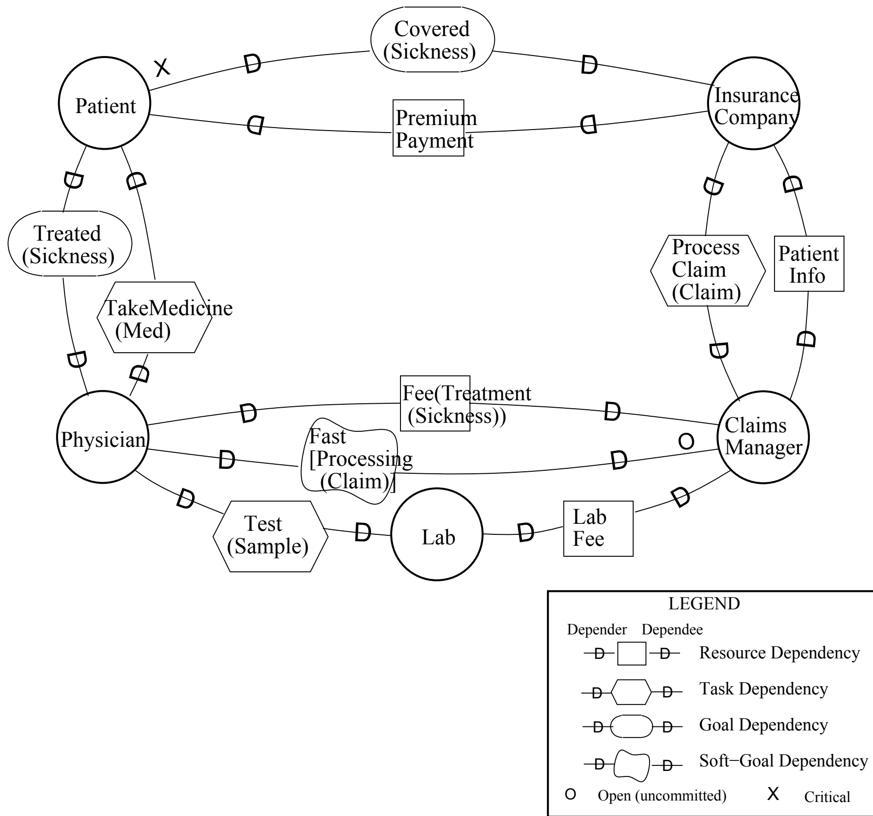


Fig. 1.1 SD model of a medical insurance enterprise [1]

1.1.1 Case Study: Healthcare

A remote healthcare system often depends heavily on cutting edge technologies like cellular network or cloud to outreach patients in remote places. The primary motivation for designing a remote healthcare system is to provide healthcare to patients as and when required. This may often require the patients to rely on handheld devices like cell phones or smart-phones to access these healthcare services in a remote manner. In [5], we try to look into the requirement specifications of such a remote healthcare enterprise and try to model them with i* modelling constructs. This helps in deriving an abstract goal model of existing legacy enterprises. Table 1.1 lists some examples of how requirements specified using predicate calculus can be converted into i* model constructs.

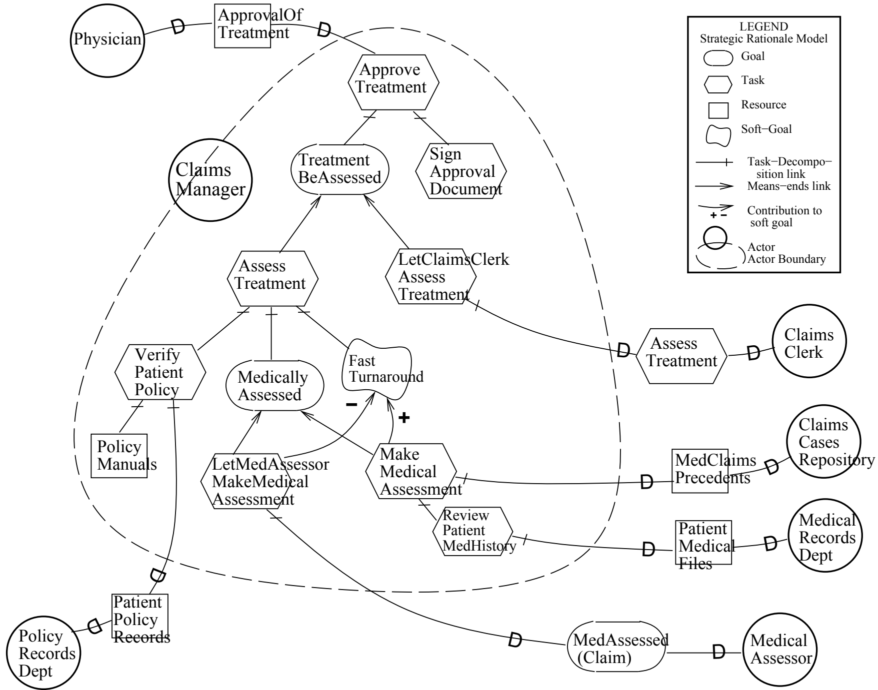


Fig. 1.2 SR model of the claims manager actor within the medical insurance enterprise [1]

1.2 Research Directions

In this book, we typically identify three different research directions and try to make some significant contributions for addressing each of these research problems. The research directions can be captured with the help three specific research questions as follows:

RQ-1: Has the community been able to model diverse real life enterprise scenarios?

- **1a**: How do the granularity of the goal models change in an enterprise hierarchy?
- **1b**: Can we correlate the different levels of a goal model hierarchy?
- **1c**: Is the degree of correlation measurable?
- **1d**: Do hierarchies really manifest themselves in process/event logs?

RQ-2: Can we extend the analytical capabilities of goal models beyond *Ability, Workability, Viability, etc.*?

- **2a**: Can model checking be applied to goal models?
- **2b**: Is it really efficient and scalable?

Table 1.1 List of existential compliance rules

Existential operators	Function	i* model
`∀X,achieves(X,G)` Ex: ∀doctors D, achieves (D, $Provide\ Healthcare$)	*Goal existence*: X is an actor in the i* model and G is a goal within the actor boundary of X	Doctor Provide Healthcare
`∀X,does(X,T)` Ex: ∀patients P, does (P, $Contact Doctor$)	*Task existence*: X is an actor in the i* model and T is a task within the actor boundary of X	Patient Contact Doctor
`∀X,acquires(X,R)` Ex: ∀doctors D, acquires (D, *Patient EMR*)	*Resource existence*: X is an actor in the i* model and R is a resource within the actor boundary of X	Doctor Patient EMR
`∀X,tries(X,S)` Ex: ∀patients P, tries (P, $Accurate\ Symptoms$)	*Softgoal existence*: X is an actor in the i* model and S is a softgoal within the actor boundary of X	Patient Accurate Symptoms
`∃jobs` (e_1,e_2,$\cdots$,e_k), `actor` X, `task` T, `s.t.` $\wedge_{i=1}^{k}$`executes`$(X, e_i) \longrightarrow$ `executes`(X, T) Ex: ∃ goal $ObtainSymp-$ $-toms$, resource $PatientEMR$, tasks $ConsultSpecialist$, $ProvideTreatment$ soft–goal $Fast$, doctor D, s.t. achieves (D,$ObtainSympt-$ $-oms$)∧acquires (D,$Patient$ EMR)∧does (D,$Consult$ $Specialist$)∧tries (D,$Fast$) ⟶does (D,$ProvideTreat-$ $-ment$)	*Task decomposition existence*: Executing jobs e_1,e_2,$\cdots$,e_k is the equivalent of actor X doing the higher level task T and e_1,e_2,$\cdots$, e_k are connected to T using task decomposition links	Doctor Provide Treatment Obtain Symptoms Patient EMR Consult Specialist Fast
`∃jobs` (m_1,m_2,$\cdots$,m_k,E), `actor` X, `s.t.` $\vee_{i=1}^{k}$`executes`$(X, m_i)\longrightarrow$ `executes`(X, E) Ex: ∃ goals $ProvideHealth-$ $-care$,$SendAmbulance$, task $ProvideTreatment$, doctor D, s.t. does (D, $ProvideTreat-$ $-ment$)∨achieves (D, $Send$ $Ambulance$)⟶achieves (D, $ProvideHealthcare$)	*Means end existence*: Executing either of the jobs m_1,m_2,$\cdots$,m_k provides actor X with alternate means to obtain the end objective E; m_1,m_2,$\cdots$,m_k are connected to E using means end links	Doctor Provide Healthcare Provide Treatment Send Ambulance for Transfer

RQ-3: Are goal models really meaningful, i.e., useful?

- **3a**: Are goal models really meaningful?
- **3b**: Can we propose goal nomenclature to capture goal semantics?
- **3c**: Can we perform semantic analysis of goal models?
- **3d**: Is the solution really scalable?

We elaborate on these research directions in this section.

1.2.1 Goal Model Hierarchies (RQ-1)

We identify unexplored scenarios and use-cases existing within real world enterprises where goal modelling techniques have not been deployed previously-typically *enterprise hierarchies*. The higher levels of the hierarchy within an enterprise are more concerned with managerial and administrative decision-making processes. As result, they have an abstract view of the system that is to be developed for a client/consumer. The lower levels of the hierarchy comprises of engineers, developers, and architects who have a better understanding of the system-to-be and their constituent components. The views derived from the lower levels of the enterprise hierarchy are, thus, more fine-tuned with respect to the client's requirements.

If different levels of the enterprise hierarchy are required to capture their perception of the system-to-be using goal models, then we observe that the higher level goal models are coarse-grained consisting of highly abstract goals whereas the lower level goal models are more fine-grained containing highly refined goals. Thus, corresponding to the enterprise hierarchy, we have a requirement refinement hierarchy consisting of multi-level goal models, where each level uses its own set of ontologies for describing the goals of the system being developed. We need to ensure that the goal models at different levels of the requirement refinement hierarchy are in harmony and do not give rise to conflicting set of requirements.

We explore the state-of-the-art in ontology integration and observe that such a mechanism or framework is not yet in place. We try to propose a framework in this direction so that we can ensure ontology integration in requirement refinement hierarchies. However, at some point, it may appear to the community that the whole notion of enterprise hierarchies is too far-fetched and seemingly hypothetical in the real world. To resolve this confusion, we try to determine whether such hierarchies manifest themselves in the real world. We try to mine goal refinement patterns from both synthetic and real-world event logs using sequential pattern mining and observe that our hypothesis about enterprise hierarchies is indeed manifested in the real world. This goes a long way in establishing the significance of our research.

1.2.2 Goal Model Checking (RQ-2)

Goal modelling is a powerful mechanism as it helps analysts to analyze the system prior to requirements specification. Every system has some set of properties or rules that they must always comply with after deployment. Compliance rules have a generic structure that specify some constraints on the ordering of event execution within the system. The following examples present some skeletal structures that compliance rules generally have.

– Whenever event X occurs, it must be followed by event Y, either immediately or some time in the future, OR
– If event X occurs, then it must be case that event Y has never occurred in the past, OR
– Events X and Y should not occur simultaneously, etc.

Thus, compliance rules try to impose an ordering of events or states within the system; but goal models are inherently sequence-agnostic. Goal models do not capture any temporal information with respect to the ordering of goal fulfillment. In the absence of such a temporal ordering we cannot perform compliance checks on a goal model.

In general, temporal properties can be easily verified on design or process models with the help of model checkers. *Model checking* can be used to verify finite state concurrent systems only but has the added advantage of full automation. Assuming that unbounded systems can be restricted to finite state under specific instances, we propose to apply model checking techniques on goal models itself. This requires us to perform model transformation and derive finite state models (FSMs) corresponding to a goal model. FSMs capture ordering of state transitions that a system can go through. We propose heuristics for such a model transformation that outperforms the existing approaches [7] for transforming goal models into FSMs. We have also developed a tool that enables model checking of i* models, using the NuSMV model verifier, as a proof-of-concept for our research contribution.

1.2.3 Semantic Analysis of Goal Models (RQ-3)

We also observe that goal nomenclature is very restricted as it depends on the interpretation of the modeller alone. Hence, we go beyond the simple nomenclature of the goals and explore the underlying semantics in order to identify any type of semantic conflicts—such as *entailment* or *consistency* issues. Analysing the semantics of goals within goal models is extremely important when we perform a goal model maintenance exercise. Given a goal model configuration we need to verify the existence of conflicts and then make changes to the configuration such that the newly derived goal model is free from such conflicts. However, it becomes quite infeasible to manually explore the space of all goal model configurations and identify the configuration that deviates minimally from the original configuration. This results

in requirement analysts coming up with sub-optimal solutions to the goal model maintenance problem.

We propose a new framework called the AFSR framework that proposes a new goal model nomenclature that helps analysts to capture the semantics associated with a goal. The framework defines a semantic reconciliation machinery to identify and detect points of conflict. The framework is quite robust as it provides re-factored configurations to the analysts that resolve the conflicts. This helps analysts to explore the complete space of goal model configurations. We map the goal model maintenance problem to the state space search problem and establish the admissibility and consistency of the heuristic path cost function. This allows us to apply A* search on the complete space of goal model configurations and derive a conflict-free configuration that deviates minimally from the original goal model configuration.

1.3 Analysis of Results

Most of the research done in this book have been in completely unexplored dimensions. As a result, there is a lack of existing data sets (or benchmarks) with which we can compare the performance of our proposed solutions. However, we have tried to use real life case studies and develop tools as proof of our concepts.

Solution to RQ-1: In our work on goal model hierarchies, we consider a University Admission System as a case study for the proposed hierarchy correlation framework. We use goal models to illustrate how the proposed ontology correlation can be achieved in goal-oriented requirements engineering. We also build heuristics on the same case study based on the framework proposed.

Solution to RQ-2: In our efforts to assess the analytical advantage of applying model checking on goal models, we have developed the `i*ToNuSMV` tool. This tool acts as a PoC for the proposed *Semantic Implosion Algorithm* where we can feed i* models and temporal specifications (in CTL) as input and check whether the specifications are being satisfied by the model. The NuSMV model verifier runs in the back end to verify the specification and generates a counterexample if it fails to satisfy.

Solution to RQ-3: The goal model maintenance framework has been supplemented with an implementation roadmap. We have spelled out the mechanism to map the reconciliation problem to a state space search problem. We apply the heuristic-based A* search algorithm to the state space and observe how A* search outperforms Uninformed search in deriving an optimal goal model configuration that is free from all conflicts.

1.4 Organization of the Book

The rest of this book is organized as follows. Chapter 2 provides a review on why we choose to work with the i* framework and documents the current state-of-the-art with respect to the research directions that we have identified. In chapter 3, we have addressed the research problem of how enterprise hierarchies get reflected in requirement refinement hierarchies. We provide techniques for modelling requirement refinement hierarchies. We show how such hierarchies exist within an enterprise by mining synthetic as well as real-life data sets. In chapter 4, we propose to apply model checking techniques to i* models. We propose two algorithms and show how the Semantic Implosion Algorithm drastically outperforms the Naïve Algorithm. We also discuss the `i*ToNuSMV` tool that we have developed for this purpose. Chapter 5 proposes the AFSR framework that performs goal model maintenance using semantic annotation of goals. We also discuss how this problem can be mapped to a state space search problem and then A* search can be applied to derive an optimal solution. Chapter 6 concludes the book with a brief discussion on future research directions.

References

1. Yu E, Modelling strategic relationships for process reengineering. PhD thesis, University of Toronto, Toronto, Canada
2. Hinge K, Ghose A, Koliadis G (2009) Process SEER: a tool for semantic effect annotation of business process models. In: Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing (EDOC), pp 54–63. https://doi.org/10.1109/EDOC.2009.24
3. van Lamsweerde A, Darimont R, Letier E (1998) Managing conflicts in goal-driven requirements engineering. Trans Softw Eng Special Issue Inconsistency Manage Softw Dev 24(11):908–926. https://doi.org/10.1109/32.730542
4. Horkoff J et al (2016) Goal-oriented requirements engineering: a systematic literature map. In: Proceedings of the IEEE 24th international requirements engineering conference (RE), Beijing, China, pp 106–115. https://doi.org/10.1109/RE.2016.41
5. Deb N, Chaki N (2014) Verification of i* models for existential compliance rules in remote healthcare systems. Appl Innov Mob Comput 60–66. https://doi.org/10.1109/AIMOC.2014.6785520
6. Schönböck J et al (2009) Catch me if you can—debugging support for model transformations. Models Workshops Lecture Notes Comput Sci 6002(2010):5–20. https://doi.org/10.1007/978-3-642-12261-3_2
7. Fuxman AD (2001) Formal analysis of early requirements specifications. MS thesis, Department of Computer Science, University of Toronto, Canada

Chapter 2
State-of-the-Art

Several languages and frameworks have been proposed in the domain of goal oriented requirements engineering that try to capture and model the requirement specifications of the system being developed. Some well documented articles have been published that compare and contrast these approaches and stress on the analytical capabilities of each approach [1–4]. For the purpose of this book, we provide a brief summary of the current state-of-the-art in the requirements engineering domain in Sect. 2.1.

The rest of this chapter is organized based on the structure of our book. Section 2.2 presents a review of some of the existing works on ontology integration and highlights how semantic integration of different levels of a requirement refinement hierarchy has not yet been addressed by the research community. This section also presents a study of different data mining techniques that have been applied in the domain of requirements engineering. Based on these reviews we present our work on i* refinement hierarchies and mining goal decomposition patterns in Chapter 3. Chapter 4 of this book presents an efficient solution for model checking goal models like i*. An extensive background study related to model transformation techniques has been documented in Sect. 2.3. Based on this study, we propose a model transformation scheme that allows analysts to perform model checking on i* models. We also present a survey of goal model annotation nomenclatures that have been proposed as part of the current state-of-the-art in Sect. 2.4. This survey forms the basis for the AFSR framework proposed in Chapter 5 which proposes a new goal model nomenclature for capturing the semantics of goal models. Each of these sections conclude with a gap analysis that helps in identifying the research question that we address in the subsequent chapters of this book.

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_2

2.1 Formal Requirements Engineering Techniques

One of the earliest requirements modelling language, Structured Analysis and Design Technique (SADT) [5], was proposed by Ross and Schoman in 1977. The language was founded on the principle of data/operation duality where data were defined by their source and destination operations while operations were defined on the basis of their input and output data. The main drawbacks of this early modelling language was the lack of precision and the absence of well-defined semantics. The syntax was also semi-formal and often interleaved with natural language assertions.

The first requirements modelling language that incorporated formal semantics was RML [6]. It introduced the semantic concept of entities, operations and constraints. Operations and constraints were expressed in formal assertion languages that supported temporal ordering of events. RML also introduced the concepts of generalization, aggregation, and classification. These formal semantics were mapped to first order predicate calculus.

Albert II [7] is a requirements language that was proposed with richer ontologies and modelled the requirements of agent-oriented systems. Entities were replaced by agents and modelled using graphical notations. Constraints on agent behavior were still modelled using textual notations that supported temporal logistics. Verification of constraints using formal analysis techniques, like animation, were possible. The drawback of this language was that it did not support enough abstraction for being used in the early phases of requirements engineering.

The state-of-the-art in requirements modelling focuses more on goal-oriented ontologies that capture the "*why*" requirements of an enterprise. The NFR framework [8, 9] specializes in the modelling and analysis of non-functional requirements or softgoals. The NFR framework deals with capturing non-functional requirements (NFRs) for the domain of interest, decomposing NFRs, identifying possible NFR operationalizations, NFR ambiguities, trade-offs, priorities, and NFR interdependencies [1].

Lamsweerde proposed the Knowledge Acquisition in autOmated Specification (KAOS) [4] framework that supports semi-formal modelling of goals, qualitative analysis of alternatives, and formal analysis for correct reasoning. The KAOS framework combines semantic nets [6], for modelling of concepts, and linear-time temporal logic for state-based specification of operations. Operations are declared by signatures over objects and have pre-, post-, and trigger conditions [1]. Goals can be decomposed using AND/OR refinement abstraction hierarchies. The KAOS framework does not provide any support for non-functional requirements or softgoals. However, the qualitative analysis techniques of the NFR framework can be integrated into KAOS. It has a solid formal framework that uses well-established formal techniques for goal refinement and operationalization [10, 11].

The Goal-Based Requirements Analysis Method (GBRAM) [12, 13] emphasizes on the identification and elicitation of goals from various documents as provided by the stakeholders of an enterprise. GBRAM distinguishes between achievement and maintenance goals. GBRAM tries to establish an ordering of goals in order to

establish inter-actor dependencies. This process is much more complex in comparison to the i* framework which can capture such dependencies quite efficiently. GBRAM does not provide a graphical interface; rather, it uses a textual notations in goal schemas for representing goals, goal refinements, goal precedence, agents, etc.

i* [2, 14] is an agent-oriented modelling framework that can be used for requirements engineering, business process re-engineering, organizational impact analysis, and software process modelling [1]. This i* framework can model activities prior to the freezing of requirement specifications. This allows the use of i* for both the early and late phases of requirements engineering. In the early phases, the i* model can capture the stakeholders of the enterprise, their objectives and how they depend on each other for achieving their objectives. The late phases of requirements engineering can use the i* modelling framework to incorporate changes and new processes that are aligned with the functional and non-functional requirements of the user.

2.2 Requirement Refinement Hierarchies

Different stakeholders within an enterprise may use partial or even completely non-intersecting vocabulary sets. This would lead to developing multiple i* models having independent ontologies. Collectively, these i* models define a requirements refinement hierarchy where different tiers of the hierarchy capture different levels of detail. In order to integrate such a distributed ontology, we require an appropriate mapping or correlate definitions between different ontologies at multiple levels. This section explores the ontology integration mechanisms that exist in the current state-of-the-art. We also try to document the research initiatives that have been taken in the last decade for applying data mining techniques in requirements engineering.

Ontology Integration Mechanisms

Graph matching of any structure is a well researched problem and several good works (like [15]) have been published in this domain. However, the real challenge being addressed here is conceptual matching of ontologies. Wang et al. [16] propose a tree similarity algorithm for ontology integration of multiple information sources available in the Semantic Web. Dynamic programming is used to effectively match concept trees while satisfying the maximum mapping theorem and keeping tree isomorphisms intact. PRIOR+ is an adaptive ontology mapping approach proposed in [17]. It consists of an information retrieval system that extracts similarities between ontologies, an adaptive similarity filter that aggregates these similarities, and, finally, a neural network based ontology constraint satisfier.

In [18], the authors bridge the gap between Description Logic based ontologies and Object Oriented systems by mapping OWL ontologies to Java interfaces and classes. STROMA [19] is a semantic ontology mapping scheme that goes beyond equality correlations and maps part-whole as well as IS_A relationships. Khattak et al. have highlighted how ontologies evolve over time [20]. A mapping reconciliation mechanism for evolving ontologies is proposed that reduces the

reconciliation time by tracking the change histories of such ontologies. [21] proposes relation mappings between ontologies by finding the least upper bound and greatest lower bounds of complex relations and then deriving the best upper and lower approximations of the relation. Kumar and Harding [22] use Description Logic based bridging rules for mapping complex concepts and roles between manufacturing and marketing enterprises. The Semantic Bridge Ontology [23] detects structural and semantic conflicts between Learning Resource Systems and resolves them by defining ontology mapping rules.

Data Mining in RE

Zawawy et al. have proposed a root-cause analysis framework [24] that mines natively generated log data to establish the relationship between a requirement and the pre- and post-conditions associated with that requirement. In [25], the authors have proposed techniques for mining dependencies from message logs and task-dependency correlations from process logs. There have been very interesting industrial and commercial applications of mining requirements from event logs. Formal verification of control systems have been performed by mining temporal requirements from simulation traces [26]. Qi et al. have provided big data commerce solutions by mining customer requirements from online reviews and suggest product improvement strategies [27]. REQAnalytics [28], proposed by Garcia and Paiva, mines the usage statistics of a website and provides a roadmap for the evolution of the website's requirements specification. [29] is another data mining technique that tries to address the inconsistencies that affect the contextual requirements of a system at runtime.

Sequential pattern mining has been frequently used for extracting statistically relevant patterns or sequences of values in data sets. StrProM [30], for instance, uses the Heuristics Miner algorithm to generate prefix-trees from the data stream and continuously prunes these trees to extract sequences of events. Sohrabi and Ghods use bitwise compression techniques to represent the data sequence as a 3-dimensional array and extract frequently occurring patterns from this compressed array [31]. Hassani et al. have proposed the PIVOTMiner [32] which considers activities as interval-based events rather than the conventional single-point events. Some researchers have also tried to improve the legacy sequential mining algorithm PrefixSpan (like [33–35]). Sequential pattern mining has also been used in interesting applications that range from detecting user behavior from online surveys [36] to mining electronic medical records and inferring the efficacy of medicines [37]. A detailed survey of sequential pattern mining algorithms is available in [38].

Previously workflow logs used to be mined for extracting the control flow within an organization and, hence, extensively used for developing process models. However, the mining process had no focus on extracting the hierarchical structure within an organization. Ni et al. have introduced the concept of executor similarity metrics and grid clustering for mining the organization structure of an enterprise from workflow logs [39]. Schönig and his group have proposed a framework to extract the organisational structure of business processes by mining human resource allocation information from event logs [40]. Also in prior work, non-functional requirements have been extracted from text [41].

Research Gap

A vast literature exists for ontology mappings within the domain of Semantic Web. However, there has been limited research on mapping model constructs between conceptual models derived from an enterprise hierarchy that use different ontologies. This research gap is identified and some ideas in the direction of bridging ontologies within hierarchic i* models developed by any enterprise is presented in Chapter 3.

2.3 Model Checking with i*

The importance of converting an i* model to a finite state model lies in the effort to perform model checking on i* models using industry standard model checkers like NuSMV and SPIN. Model checking helps requirement analysts to verify whether goal models comply with system regulations. Standard model checkers accept extended finite state models as input and verify temporal properties specified using Linear Temporal Logic (LTL) or Computational Tree Logic (CTL). i* models can be converted to other sequential models like activity diagrams or BPMNs for some specific business requirement, if the need arises. Transforming finite state models to other sequential models should be easier than transforming a sequence agnostic model, like i*, to activity diagrams or BPMNs. Model transformation represents the daunting challenge of converting higher-level abstraction models to platform-specific implementation models that may be used for automated code generation.

Model Transformation

Sendall and Kozaczynski had already identified model transformation as one of the major driving forces behind model-driven software development [42]. Most strategies work with lower levels of abstraction and encounter several limitations. In [43], the authors propose a Domain Specific Language over Coloured Petri-Nets—called Transformation Nets—that provides a high level of model transformation abstraction. An integrated view of places, transitions, and tokens, provide a clear insight into the previously hidden operational semantics.

Model transformation plays a vital role in bridging the gap between non-successive phases of the software development life cycle. [44] presents one such attempt to bridge the gap between system designers and system analysts. A model generated by the designer is transformed to a model suitable for conducting analysis. The outcome of the analysis is mapped back into the design domain. The authors work with *UML2Alloy*—a tool that takes a UML Class diagram augmented with OCL constraints and converts it into the Alloy formal representation. Design inconsistency analysis is done on the Alloy representation. Alloy creates counter examples for any such inconsistency and converts it back into a UML Object diagram. This paper tries to do model transformation for bridging the gap between the requirements phase and the design phase of the development life cycle.

Creating a wide array of formal models for enhancing the system engineering process, proves to have time and cost overheads. Kerzhner and Paredis use model transformations to achieve this objective, overcoming the overheads, in [45]. Formal models are used to specify the structures of varying design alternatives and design requirements, along with experiments that conform the two. These models are represented using the Object Management Group's Systems Modelling Language (OMG SysMLTM). Model transformation is then used to transform design structures into analysis models by combining the knowledge of reusable model libraries. Analysis models are transformed into executable simulations which help in identifying possible system alternatives. Model transformation plays a vital role in this work.

Mussbacher et al., have performed a detailed comparison of six different modelling approaches in [46]. The modelling approaches that were assessed include Aspect-oriented User Requirements Notation (*AoURN*) [47], Activity Theory (*AT*) [48], The Cloud Component Approach (*CCA*), Model Driven Service Engineering (*MDSE*) [49], Object-oriented Software Product Line Modelling (*OO-SPL*) [50], and Reusable Aspect Models (*RAM*) [51, 52]. The comparison criteria were grouped into two broad categories—*Modelling Dimensions* and *Key Concepts*. Modelling dimensions include properties like Phase, Notation, and Units of Encapsulation. Key concepts, on the other hand, provide an insight into parameters like Paradigm, Modularity, Composability, Traceability and Trade-off Analysis. Of these six approaches, *AoURN* [47, 53] and *OO-SPL* [50] are of interest to this work, as both these approaches are applicable in the Early and Late Requirements phases of software development. The i* modelling notation belongs to this approach. In fact, *AoURN* is based on the ITU-T Z.151 [54] standard that uses Goal-oriented Requirements Language (GRL), that is based on i* modelling. *AoURN* is machine analysable and can perform scenario regression tests, goal-model evaluation, and trade-off analysis. Unlike the other modelling approaches, *AoURN* provides structural, behavioural, and intentional views, along with generic support for qualities and non-functional properties. It is purely graphical in nature.

Most model checking techniques are best suited for the design and subsequent phases of the development life cycle. Architectural [55] and detailed design [56] model checking have been proposed. Automated verification of requirement models requires a completely different set of ontologies and, hence, existing techniques cannot be extended to requirement models. However, work has been done on the application of model checking to requirement models. Heitmeyer et.al. have proposed a tool support for the SCR tabular notation for requirements specification [57, 58] that supports formal techniques for consistency and completeness checking, model checking, theorem proving and animation. The RSML language [59] has better structuring mechanisms than the SCR notation and also provides a tool support for completeness and consistency checking. Both these approaches are restricted to the domain of embedded systems and process control. Neither of these tools support the goal ontologies that have been proposed for early requirements engineering. Wang has explored the application of ConGolog [60] formalisms to the i* framework for analysing early requirement specification. The work tries to map i* SR diagram concepts to ConGolog primitives and control structures. ConGolog is based on *situation*

calculus [61] and provides a formal machinery for proving assertions on requirement specifications.

Telos [62] captures the i* meta-framework which describes all the semantics and constraints of the i* framework. Telos is equipped with the ability to perform different types of analysis and also check the consistency between i* models. Tropos [63] is an agent-oriented system development framework that utilizes the i* modelling framework to model agent requirements and system configurations. Formal Tropos [3, 64] associates the Tropos methodology to a formal specification language. Formal Tropos allows the specification of constraints, invariants, pre- and post- conditions, thereby capturing the semantics of the i* graphical models. These models can be validated using model checking.

Research Gap

Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target. Most model-driven engineering practises offer a black box view of the transformation logic making it difficult to observe the operational semantics of a transformation. In order to perform model checking on any sequence-agnostic goal model (like i*), we must first transform the model into some form of finite state model that provides a possible sequencing of activities within the enterprise. This research direction has been explored in Chapter 4.

2.4 Semantic Annotations of Goal Models

The domain of goal model maintenance requires analysts to explore the space of goal model configurations that can be derived from a given erroneous goal model. However, the research question becomes even more complex if we try to go beyond the structural features and consider the semantics of goal models. The nomenclature of goal models do not capture the semantics of the goals. There has been very limited work in the existing literature that annotates goal models with more information. This section highlights some of the research that has been done in the domain of annotating requirement models with different types of attributes.

Annotation of Goal Models

Liaskos and Mylopoulos [65] have identified the sequence agnostic nature of standard goal modelling notations like i* [2, 14] and annotated them with temporal logistics for deriving AI-based goal satisfaction planning. The authors introduce the notion of *precedence links* and *effect links* that annotate the i* model with preconditions and postconditions of fulfilling a goal. This kind of ordering allows formalization of goal models using temporal logics (like LTL, CTL, etc.). Although this method establishes some sort of a sequence between the tasks of a goal model, the notion of precedence does not remain intuitive for softgoals. Softgoal satisfaction can be facilitated with *hurt* and *help* contributions from tasks, hard goals, etc.

In [66], Liaskos et al. have highlighted the importance of augmenting goal models (like i*) with the optional requirements or preferences of the users. This paper uses the *precedence* and *effect* links proposed in [65]. Additionally, this work introduces the notion of weighted contribution links for evaluating the degree of satisfaction or denial for softgoals. Accumulation and propagation of these weighted contributions follow the rules prescribed in [67]. Optional user requirements are defined as *Optional Condition Formulae* (OCFs) using first order satisfaction and domain predicates. Preferences are captured as linear combinations of OCFs and these preference formulae may be weighted or non-weighted in nature. Alternate goal plans are evaluated based on the degree of satisfaction of the preferences.

Koliadis and Ghose have been working with semantic effect annotations of business process models [68–70]. In [68], the authors propose the GoalBPM methodology that maps business process models (using BPMN) to high-level stakeholder goals described using the KAOS framework [4]. This is done by defining two types of links—*traceability* and *satisfaction*. The former links goals to activities while the latter links goals to entire business processes. Satisfaction links require effect annotation of the business process model, followed by identification of a set of critical trajectories and, finally, identifying the subset of traceability links that represent the satisfaction links. In [69, 70], the authors have worked with semantic effect annotation and accumulation over business process models (Process SEER) and how it can be extended to check for business process compliance using the PCTk toolkit.

Kaiya et al. [71] have proposed the popular Attributed Goal-Oriented Requirements Analysis (AGORA) method that derives a goal graph from goal models and annotates the nodes and edges of the graph with attribute values and quality matrices. Attribute values consist of contribution values and preference matrices. Contribution values are used to annotate the edges of the goal graph and represent the contribution of the sub-goal towards the fulfillment of the higher-level goal. Preference matrices are the vertex annotations and represent the preference of respective goals to the concerned stakeholders. Both these attribute annotations can be used by analysts to choose among multiple alternate strategies, to perform conflict management and change management. Quality metrics are used to analyze the quality of the requirement specifications that are derived from the goal graphs. The metrics for measuring such quality may be correctness, unambiguity, completeness, inconsistency, etc. Yamamoto and Saeki [72] have extended the idea of using annotated goal graphs for requirements analysis to software component selection.

Research Gap

Researchers have attempted to annotate goal models with temporal information for simulation and model checking purposes. They have also tried to identify effects of goal fulfillment for evaluating user preferences. Semantic annotation of goal models may seem intuitive and analogous to effect annotation of business process models but it is not so. Goal models and process models have completely different objectives and characteristics. The most crucial differential characteristic being the sequence-agnostic nature of goal models. In this perspective, it becomes necessary to spell out a mechanism for semantic annotation of goal model artefacts, and how these goal

semantics can be reconciled over the entire enterprise for performing goal model maintenance. Chapter 5 proposes a framework for doing this and also provides a roadmap to implement it.

References

1. Lapouchnian A (2005) Goal-oriented requirements engineering: an overview of the current research, Depth Report. University of Toronto, Toronto, Canada
2. Yu E, Modelling strategic relationships for process reengineering. PhD thesis, University of Toronto, Toronto, Canada
3. Fuxman AD (2001) Formal analysis of early requirements specifications. MS thesis, Department of Computer Science, University of Toronto, Canada
4. van Lamsweerde A, Darimont R, Letier E (1998) Managing conflicts in goal-driven requirements engineering. Trans Softw Eng Special Issue Inconsistency Manage Softw Dev 24(11):908–926. https://doi.org/10.1109/32.730542
5. Ross D (1977) Structured analysis (SA): a language for communicating ideas. IEEE Trans Softw Eng 3(1):16–34. https://doi.org/10.1109/TSE.1977.229900
6. Greespan S, Borgida A, Mylopoulos J (1986) A requirements modeling language and its logic. Knowl Based Manage Syst 471–502. https://doi.org/10.1016/0306-4379(86)90020-7
7. Bois PD (1995) The albert ii language: on the design and use of a formal specification language for requirements analysis. PhD thesis, Notre Dame de la Paix, Namur, Belgium
8. Chung L et al, Non-functional requirements in software engineering. Kluwer Academic Publishers. https://doi.org/10.1007/978-1-4615-5269-7. ISBN 978-1-4615-5269-7
9. Mylopoulos J, Chung L, Nixon B, Representing and using non-functional requirements: a process-oriented approach. IEEE Trans Softw Eng 18(6). https://doi.org/10.1109/32.142871
10. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Program 20(1–2):3–50. https://doi.org/10.1016/0167-6423(93)90021-G
11. van Lamsweerde A, Letier E (2002) From object orientation to goal orientation: a paradigm shift for requirements engineering. In: Proceedings of the 9th international workshop on radical innovations of software and systems engineering in the future (Lecture Notes in Computer Science 2941), pp 325–340. https://doi.org/10.1007/978-3-540-24626-8_23
12. Anton A (1996) Goal-based requirements analysis. In: Proceedings of the 2nd IEEE international conference on requirements engineering (ICRE), pp 136–144. https://doi.org/10.1109/ICRE.1996.491438
13. Anton A (1997) Goal identification and refinement in the specification of software-based information systems. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA
14. Yu E (1997) Towards modeling and reasoning support for early-phase requirements engineering.In: Proceedings of the 3rd international symposium on requirements engineering (RE), pp 226–235. https://doi.org/10.1109/ISRE.1997.566873
15. Abbas S, Seba H (2012) A module-based approach for structural matching of process models. In: Proceedings of the 5th IEEE international conference on service-oriented computing and applications (SOCA), pp 1–8. https://doi.org/10.1109/SOCA.2012.6449441
16. Wang J, Liu H, Wang H (2014) A mapping-based tree similarity algorithm and its application to ontology alignment. Knowl Based Syst 56:97–107. https://doi.org/10.1016/j.knosys.2013.11.002
17. Mao M, Peng Y, Spring M (2010) An adaptive ontology mapping approach with neural network based constraint satisfaction. Web Semant Sci Serv Agents World Wide Web 8(1):14–25. https://doi.org/10.1016/j.websem.2009.11.002
18. Kalyanpur A, Pastor DJ, Battle S, Padget JA (2004) Automatic mapping of OWL ontologies into java. In: Proceedings of the sixteenth international conference on software engineering and knowledge engineering (SEKE), pp 98–103

19. Arnold P, Rahm E (2014) Enriching ontology mappings with semantic relations. Data Knowl Eng 93:1–18. https://doi.org/10.1016/j.datak.2014.07.001
20. Khattak A, Pervez Z, Khan W, Khan A, Latif K, Lee S (2015) Mapping evolution of dynamic web ontologies. Inf Sci 303(C):101–119. https://doi.org/10.1016/j.ins.2014.12.040
21. Wang P, Xu B, Lu J, Kang D, Zhou J (2006) Mapping ontology relations: an approach based on best approximations. In: Proceedings of the 8th Asia-Pacific Web conference on Frontiers of WWW Research and Development, APWeb'06 (Lecture Notes in Computer Science 3841), pp 930–936. https://doi.org/10.1007/11610113_97
22. Kumar SK, Harding JA (2013) Ontology mapping using description logic and bridging axioms. Comput Ind 64(1):19–28. https://doi.org/10.1016/j.compind.2012.09.004
23. Arch-int N, Arch-int S (2013) Semantic ontology mapping for interoperability of learning resource systems using a rule-based reasoning approach. Expert Syst Appl 40(18):7428–7443. https://doi.org/10.1016/j.eswa.2013.07.027
24. Zawawy H, Mankovskii S, Kontogiannis K, Mylopoulos J (2015) Mining software logs for goal-driven root cause analysis. Art Sci Anal Softw Data Chap 18:519–554
25. Ghose A, Santiputri M, Saraswati A, Dam HK (2014) Data-driven requirements modeling: some initial results with i*. In: Tenth Asia-Pacific conference on conceptual modelling (APCCM), vol 154, pp 55–64. http://dl.acm.org/citation.cfm?id=2667691.2667698
26. Jin X, Donze A, Deshmukh JV, Seshia SA (2015) Mining requirements from closed-loop control models. IEEE Trans Comput Aided Des Integr Circ Syst 34(11):1704–1717. https://doi.org/10.1109/TCAD.2015.2421907
27. Qi J, Zhang Z, Jeon S, Zhou Y (2016) Mining customer requirements from online reviews: A product improvement perspective. Inf Manage Elsevier 53(8):951–963. https://doi.org/10.1016/j.im.2016.06.002
28. Garcia JE, Paiva AC (2016) Maintaining requirements using web usage data. Proc Comput Sci 100(Supplement C):626–633. https://doi.org/10.1016/j.procs.2016.09.204
29. Knauss A, Damian D, Franch X, Rook A, Müller HA, Thomo A (2016) ACon: a learning-based approach to deal with uncertainty in contextual requirements at run time. Inf Softw Technol Elsevier 70(Supplment C):85–99. https://doi.org/10.1016/j.infsof.2015.10.001
30. Hassani M, Siccha S, Richter F, Seidl T (2015) Efficient process discovery from event streams using sequential pattern mining. In: IEEE symposium series on computational intelligence, pp 1366–1373. https://doi.org/10.1109/SSCI.2015.195
31. Sohrabi MK, Ghods V (2016) CUSE: a novel cube-based approach for sequential pattern mining. In: 4th international symposium on computational and business intelligence (ISCBI) pp 186–190. https://doi.org/10.1109/ISCBI.2016.7743281
32. Hassani M, Lu Y, Wischnewsky J, Seidl T (2016) A geometric approach for mining sequential patterns in interval-based data streams. In: IEEE international conference on fuzzy systems (FUZZ-IEEE), pp 2128–2135. https://doi.org/10.1109/FUZZ-IEEE.2016.7737954
33. Chaudhari M, Mehta C (2016) Extension of prefix span approach with grc constraints for sequential pattern mining. In: International conference on electrical, electronics, and optimization techniques (ICEEOT), pp 2496–2498. https://doi.org/10.1109/ICEEOT.2016.7755142
34. Fei X, Zheng S, Li-jing Y, Chao F (2016) A improved sequential pattern mining algorithm based on prefixspan. World Autom Congr (WAC) 1–4. https://doi.org/10.1109/WAC.2016.7583059
35. Patel R, Chaudhari T (2016) A review on sequential pattern mining using pattern growth approach. In: International conference on wireless communications, signal processing and networking (WiSPNET), pp 1424–1427. https://doi.org/10.1109/WiSPNET.2016.7566371
36. Zhu X, Wu S, Zou G (2015) User behavior detection for online survey via sequential pattern mining. In: Fifth international conference on instrumentation and measurement, computer, communication and control (IMCCC), pp 493–497. https://doi.org/10.1109/IMCCC.2015.110
37. Uragaki K, Hosaka T, Arahori Y, Kushima M, Yamazaki T, Araki K, Yokota H (2016) Sequential pattern mining on electronic medical records with handling time intervals and the efficacy of medicines. In: IEEE symposium on computers and communication (ISCC), pp 20–25. https://doi.org/10.1109/ISCC.2016.7543708

38. Abbasghorbani S, Tavoli R (2015) Survey on sequential pattern mining algorithms. In: 2nd international conference on knowledge-based engineering and innovation (KBEI), pp 1153–1164. https://doi.org/10.1109/KBEI.2015.7436211
39. Ni Z, Wang S, Li H (2011) Mining organizational structure from workflow logs. In: Proceeding of the international conference on e-education, entertainment and e-management, pp 222–225. https://doi.org/10.1109/ICeEEM.2011.6137791
40. Schöniga S, Cabanillas C, Jablonski S, Mendling J (2016) A framework for efficiently mining the organisational perspective of business processes. Decis Support Syst Elsevier 89(Supplement C):87–97. https://doi.org/10.1016/j.dss.2016.06.012
41. Cleland-Huang J, Settimi R, Zou X, Solc P (2006) The detection and classification of non-functional requirements with application to early aspects In: 14th IEEE international conference requirements engineering, pp 39–48. https://doi.org/10.1109/RE.2006.65
42. Sendall S, Kozaczynski W (2003) Model transformation: The heart and soul of model-driven software development. IEEE Softw 20(5):42–45. https://doi.org/10.1109/MS.2003.1231150
43. Schönböck J et al (2009) Catch me if you can—debugging support for model transformations, MODELS, Workshops (Lecture notes in computer science 6002(2010)), pp 5–20. https://doi.org/10.1007/978-3-642-12261-3_2
44. Shah SMA, Anastasakis K, Bordbar B (2009) From uml to alloy and back again, MODELS, workshops (Lecture notes in computer science 6002(2010)), pp 158–171. https://doi.org/10.1007/978-3-642-12261-3_16
45. Kerzhner AA, Paredis CJJ (2010) Model-based system verification: a formal framework for relating analyses, requirements, and tests, MODELS, workshops (Lecture notes in computer science 6627(2011)), PP 279–292. https://doi.org/10.1007/978-3-642-21210-9_27
46. Mussbacher G et al (2011) Comparing six modelling approaches, MODELS, workshops (Lecture notes in computer science 7167(2012)), PP 217–243. https://doi.org/10.1007/978-3-642-29645-1_22
47. Mussbacher G (2010) Aspect-oriented user requirements notation. PhD thesis, School of Information Technology and Engineering, University of Ottawa, Canada
48. Georg G (2011) Activity theory and its applications in software engineering and technology. Technical Report CS-11-101, Colorado State University
49. Kathayat SB, Le HN, Bræk R (2011) A model-driven framework for component-based development. In: 15th International SDL Forum Toulouse Integrating System and Software Modeling 2011, France, pp 154–167. https://doi.org/10.1007/978-3-642-25264-8_13
50. Capozucca A, Cheng B, Guelfi N, Istoan P (2011) bcms-oom-spl, repository for model driven development. http://www.cs.colostate.edu/content/bcms-oom-spl
51. Klein J, Kienzle J (2007) Reusable aspect models, 11th workshop on aspect-oriented modelling. Nashville, TN, USA
52. Kienzle J et al (2010) Aspect-oriented design with reusable aspect models, transactions on aspect-oriented software development VII (Lecture notes in computer science 6210), pp 272–320. https://doi.org/10.1007/978-3-642-16086-8_8
53. Mussbacher G, Amyot D, Araújo J, Moreira A (2010) Requirements modelling with the aspect-oriented user requirements notation (AoURN): a case study. In; Transactions on aspect-oriented software development VII: a common case study for aspect-oriented modeling, pp 23–68. https://doi.org/10.1007/978-3-642-16086-8_2
54. User Requirements Notation (URN)—Language Definition (2008) ITU-T: Recommendation Z.151. Geneva, Switzerland. http://www.itu.int/rec/T-REC-Z.151/en
55. Allen R, Garlan D (1994) Formalizing architectural connection. In: Proceedings of the 16th international conference on software engineering, pp 71–80. http://dl.acm.org/citation.cfm?id=257734.257745
56. Cimatti A et al (1998) Formal verification of a railway interlocking system using model checking. J Formal Aspects Comput 10:361–380. https://doi.org/10.1007/s001650050022
57. Heitmeyer C, Jeffords R, Labaw B (1996) Automated consistency checking of requirements specifications. ACM Trans Softw Eng Methodol 5(3):231–261. https://doi.org/10.1145/234426.234431

58. Heninger K (1980) Specifying software requirements for complex system: new techniques and their application. IEEE Trans Softw Eng 6(1):2–13. https://doi.org/10.1109/TSE.1980.230208
59. Levenson N, Heimdahl M, Hildreth H (1994) Requirements specification for process control systems. IEEE Trans Softw Eng 20(9):684–706. https://doi.org/10.1109/32.317428
60. Giacomo GD, Lesperance Y, Levesque H (2000) Congolog, a concurrent programming language based on the situation calculus. J Artif Intell 121(1–2):109–169. https://doi.org/10.1016/S0004-3702(00)00031-X
61. McCarthy J, Hayes P (1969) Some philosophical problems from the standpoint of artificial intelligence. Mach Intell 4:463–504. https://doi.org/10.1016/B978-0-934613-03-3.50033-7
62. Mylopoulos J et al (1990) Telos: representing knowledge about information systems. ACM Trans Inf Syst 8(4):325–362. https://doi.org/10.1145/102675.102676
63. Castro J, Kolp M, Mylopoulos J (2002) Towards requirements-driven information systems engineering: the tropos project. Inf Syst 27(6):365–389. https://doi.org/10.1016/S0306-4379(02)00012-1
64. Fuxman A, Pistore M, Mylopoulos J, Traverso P (2001) Model checking early requirements specifications in tropos. In: Proceedings of the 5th international symposium on requirements engineering (RE), pp 174–181. https://doi.org/10.1109/ISRE.2001.948557
65. Liaskos S, Mylopoulos J (2010) On temporally annotating goal models. In: Proceedings of the 4th international i* workshop—iStar10, pp 62–66
66. Liaskos S, McIlraith SA, Mylopoulos J (2009) Towards augmenting requirements models with preferences. In: IEEE/ACM international conference on automated software engineering, pp 565–569. https://doi.org/10.1109/ASE.2009.91
67. Sebastiani R, Giorgini P, Mylopoulos J (2004) Simple and minimum-cost statisfiability for goal models. In: Proceedings of the 16th international conference on advanced information systems engineering (CAiSE'04), pp 20–35. https://doi.org/10.1007/978-3-540-25975-6_4
68. Koliadis G, Ghose A (2006) Relating business process models to goal-oriented requirements models in KAOS. In: Advances in knowledge acquisition and management, pacific rim knowledge acquisition workshop (PKAW), pp 25–39. https://doi.org/10.1007/11961239_3
69. Hinge K, Ghose A, Koliadis G (2009) Process SEER: a tool for semantic effect annotation of business process models. In: Proceedings of the 13th IEEE international conference on enterprise distributed object computing (EDOC), pp 54–63. https://doi.org/10.1109/EDOC.2009.24
70. Ghose A, Koliadis G (2008) PCTk: a toolkit for managing business process compliance. In: Proceedings of the 2nd international workshop on Juris-Informatics (JURISIN'08)
71. Kaiya H, Horai H, Saeki M (2002) AGORA: attributed goal-oriented requirements analysis method. In: Proceedings of the IEEE joint international conference on requirements engineering, pp 13–22. https://doi.org/10.1109/ICRE.2002.1048501
72. Yamamoto K, Saeki M (2007) Using attributed goal graphs for software component selection: an application of goal-oriented analysis to decision making. In: 26th international conference on conceptual modeling ER '07 tutorials, posters, panels and industrial contributions, vol 83, pp 215–220. http://dl.acm.org/citation.cfm?id=1386957.1386992

Chapter 3
i* and Enterprise Hierarchies

Distributed ontology integration deals with formal machinery that helps in establishing relations between concepts that belong to entirely different ontologies. Extensive use of multiple ontologies within the same enterprise often leads to inconsistencies, redundancies and anomalies. Thus, it becomes extremely important to accommodate these ontologies within the existing hierarchy. Defining correlations between model concepts belonging to different ontologies become mandatory for enabling such ontology integrations. This chapter presents research in this direction.

Suppose there exists an n-level hierarchy within the stakeholders of an enterprise and each level of the hierarchy has their own ontology for modelling the requirements of a deliverable. At each level, we obtain a model that utilizes the ontology of that particular level. This gives rise to a requirement refinement hierarchy consisting of n different models representing n different knowledge bases, one for each level of the hierarchy. Let the languages obtained from these n levels of the hierarchy be denoted by $L_1, L_2, \ldots, L_n$, respectively.

In such an environment setting, hierarchic ontology integration can be achieved with the help of a bridge language that is governed by the goal refinement conditions of the KAOS framework [1]—*entailment* and *consistency*. *Entailment* ensures that, for every bridging rule, higher level concepts are realizable by the lower level concepts and *Consistency* ensures that the lower level model concepts do *not* create a state that contradicts some high-level concept. *Minimality* is consciously ignored as two adjacent levels of the hierarchy are derived from completely different vocabularies. The lower levels of the hierarchy may intentionally model finer details that are not captured in the higher levels.

In order to address this problem, we try to propose a bridging mechanism or language that correlates concepts belonging to adjacent tiers of the n-level hierarchy. Such a set of bridging rules provide hierarchic ontology integration for the n-tier knowledge-based hierarchy. However, instead of having a formal language knowledge base for each tier of the refinement hierarchy, we can have knowledge bases written in conceptual modelling languages. This gives rise to a n-level conceptual

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_3

model hierarchy developed using multiple ontologies. In this chapter, we consider one particular case of conceptual modelling, known as *i** models, that is used for modelling requirement specifications. For greater understanding of i* models, readers can refer to [2]. In this work, we try to explore the nature and types of bridging rules or correlation constraints that may be defined for ensuring hierarchic ontology mapping of an n-tier i* hierarchy while maintaining consistency and entailment.

In many requirements engineering exercises, different stakeholder groups within an enterprise have different vocabularies which are utilized for modelling purposes. The main motivation behind this work lies in accommodating these distinct vocabularies of the different stakeholders with the help of bridging rules or correlation constraints, so that adjacent tiers of the n-tier ontological hierarchy are *relatively complete*. The notion of relative completeness has been further elaborated in Sect. 3.2. In order to illustrate the motivation behind our work, let us consider the case study of the University Admission System.

Case Study: University Admission System

Let us consider a university that offers Postgraduate degree courses. The courses may be offered from the school of Sciences, or Arts. The requirement specifications for the admission system vary across different administrative levels of the University. At the topmost levels of the hierarchy, the Vice Chancellor (VC) and the Pro-Vice-Chancellors (Pro-VCs) view the entire university as a single entity that is managed by different schools of faculty belonging to different departments. The higher levels of administration are agnostic to the department-level details of how students will be admitted into the various courses offered by each individual department. Such an abstract view of the University Admission System is presented in Fig. 3.1.

The perception of the University Admission System completely changes when we model the requirement specifications from the point of view of each individual department. Each department may offer more than one postgraduate course and the admission criteria for each of these courses may be different. For instance, the Computer Science and Engineering department may offer M.Sc., Postgraduate B.Tech., and M.Tech. courses. M.Sc. can be pursued at the postgraduate level by only those students who have a graduation in Computer Science (Hons.). The Postgraduate B.Tech. course can be applied by Higher Secondary students who have cleared some entrance tests. Even graduates in Computer Science (Hons.) can be admitted to the P.G. B.Tech course through lateral entry. The M.Tech. course can accommodate M.Sc. in Computer Science, Masters in Computer Applications, as well as B.Tech. graduates from other engineering colleges. This detailed requirement specification of the University Admission System has been modelled with the i* model shown in Fig. 3.2.

Figures 3.1 and 3.2 have different ontologies and capture two different i* models in the requirement refinement hierarchy. This is an overly simplified scenario where many of the constructs appearing in these two i* models have the same ontology.

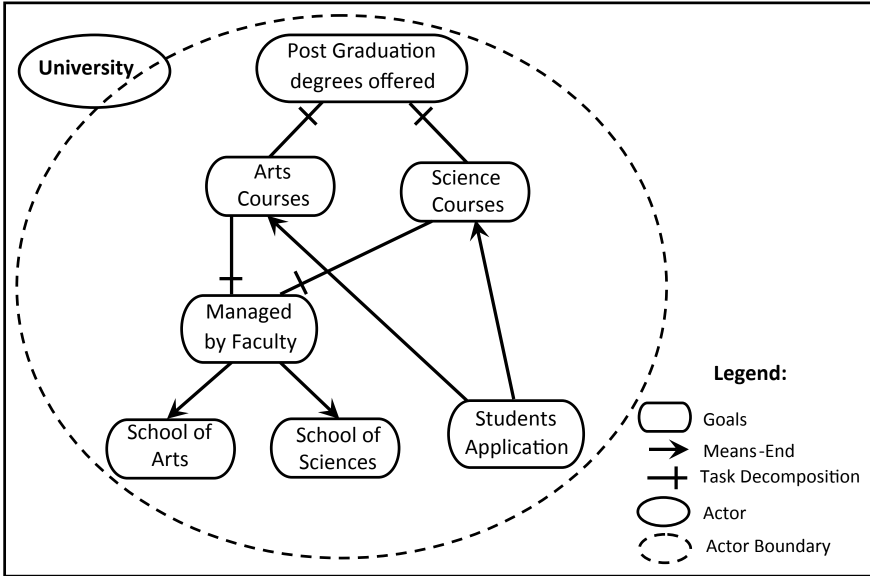


Fig. 3.1 Requirement specifications modelled by the university at abstract higher levels of the requirement refinement hierarchy

This may not be the case always. However, these two i* models have different actor sets, goals, subgoals, task decompositions and means-end decompositions.

The formal annotation of i* models defined by Guizzardi et. al. represents a means end link [3] using the proposition *ME(g,G)*, where *g* represents a means to deliberately achieve the goal *G*. We extend this formal annotation and say that if a goal *G* has *k* possible means $G_1, G_2, \ldots, G_k$, then it can be represented using the same binary predicate *ME* as $ME(\{G_1, G_2, \ldots, G_k\}, G)$. Similarly, task decompositions can also be defined by extending another binary predicate, say *TD*. Let MEL^i_{Univ} and TDL^i_{Univ}represent the set of means end links and task decomposition links for the *University* actor at level-i, respectively. The i* model shown in Fig. 3.1 can be formally described using this extended formal annotation as follows:

1. The University offers Post-graduation degrees in Arts and Sciences.

TD ({Offer *Arts Courses*, Offer *Science Courses*}, *Post-graduation Degrees Offered*) $\in TDL^i_{Univ}$

2. Students can apply to Arts courses or Science courses for Post-graduation.

ME ({Apply for *Arts Courses*, Apply for *Science Courses*}, *Students Application*) $\in MEL^i_{Univ}$

3. All courses offered by the University are managed by the respective faculty.

TD ({Manage *Arts Courses*, Manage *Science Courses*}, *Managed by Faculty*) $\in TDL^i_{Univ}$

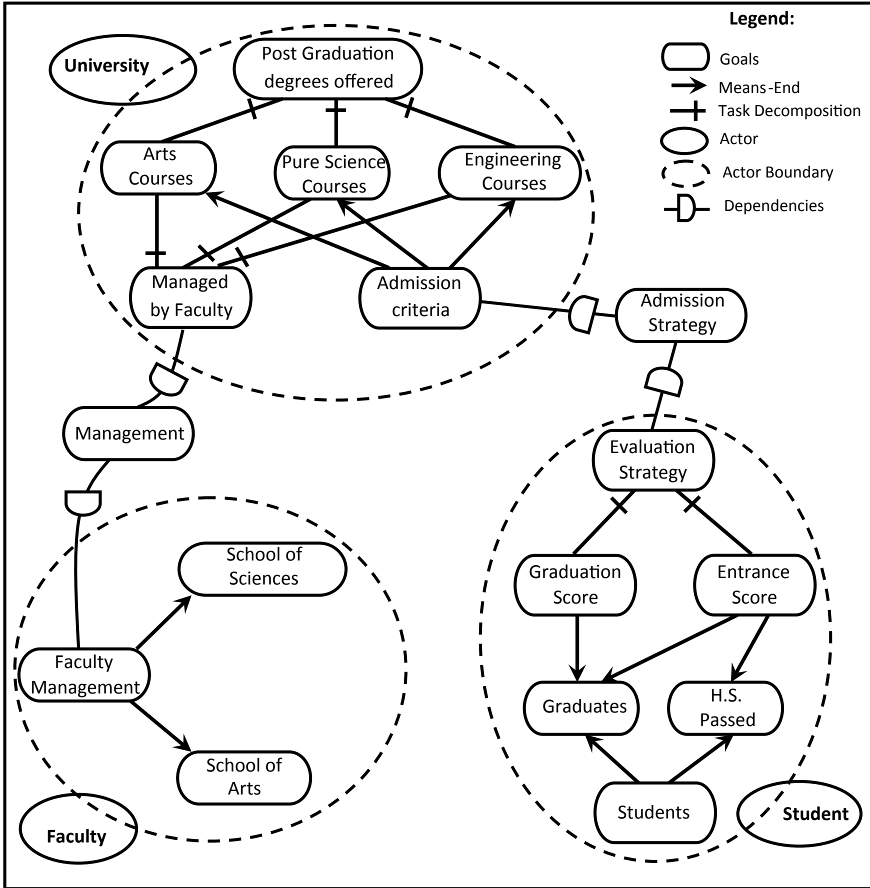


Fig. 3.2 Requirement specifications modelled by individual departments at the lower levels of the requirement refinement hierarchy

4. Faculty may belong to School of Sciences. These schools organize the Arts and Science courses that are offered by the University.

$$ME\ (\{\text{Belongs to } School\ of\ Arts, \text{Belongs to } School\ of\ Sciences\}, Faculty) \in MEL^{i}_{Univ}$$

$$TD\ (\{\text{Organize } Arts\ Courses\}, School\ of\ Arts) \in TDL^{i}_{Univ}$$

$$TD\ (\{\text{Organize } Science\ Courses\}, School\ of\ Sciences) \in TDL^{i}_{Univ}$$

Let MEL^{i+1}_{Univ} and TDL^{i+1}_{Univ} represent the set of means end links and task decomposition links for the *University* actor at level-($i + 1$), respectively. The ontologies describing the i* model in Fig. 3.2 can also be documented by extending the formal annotation of i* models [3], as before.

1. The University offers Post-graduation degrees in Arts, Pure Sciences and Engineering Sciences.

   TD ({Offer *Arts Courses*, Offer *Pure Science Courses*, Offer *Engineering Science Courses*}, *Post-graduation Degrees Offered*) $\in TDL_{Univ}^{i+1}$

2. All courses offered by the University are managed by the respective faculty.

   TD ({Manage *Arts Courses*, Manage *Pure Science Courses*, Manage *Engineering Science Courses*}, Managed by Faculty) $\in TDL_{Univ}^{i+1}$

3. The admission criteria for different courses are different.

   ME ({Criteria for *Arts Courses*, Criteria for *Pure Science Courses*, Criteria for *Engineering Science Courses}, Admission criteria)* $\in MEL_{Univ}^{i+1}$

4. Faculty may belong to either school. Respective faculties organize the corresponding courses offered by the University.

   ME ({Belongs to *School of Arts*, Belongs to *School of Sciences*}, *Faculty*) $\in MEL_{Univ}^{i+1}$
   TD ({Organize *Arts Courses*}, *School of Arts*) $\in TDL_{Univ}^{i+1}$
   TD ({Organize *Pure Science Courses*, Organize *Engineering Science Courses*}, *School of Sciences*) $\in TDL_{Univ}^{i+1}$

5. Student can be Graduates or HS passed students

   ME ({Qualified *Graduation*, Qualified *H.S.}, Students)* $\in MEL_{Univ}^{i+1}$

6. Evaluation strategy is calculated from both Graduation scores and Entrance exam scores

   TD ({Evaluate *Graduation Score*, Evaluate *Entrance Score*}, *Evaluation Strategy*) $\in TDL_{Univ}^{i+1}$

7. Graduates can provide with Graduation scores. Entrance exam may be given by both Graduates and HS passed students

   TD ({*Graduates*} Provide, *Graduation Score)* $\in TDL_{Univ}^{i+1}$
   TD ({*Graduates* Provide, *H.S. Passed* Provide}, *Entrance Score)* $\in TDL_{Univ}^{i+1}$

8. University depends on Faculty schools for management and organization of various courses and students depend on the University for Admission. The *DD(depender, dependee)* predicate is used for representing dependencies.

   DD (Faculty Management, Managed by Faculty) $\in DDL_{Univ}^{i+1}$
   DD (Admission criteria, Evaluation strategy) $\in DDL_{Univ}^{i+1}$

In such a situation, it becomes necessary to ensure that the ontologies used in the i* models of Figs. 3.1 and 3.2 are *relatively complete*. Let the set of requirements captured by the higher level (more abstract) i* model be denoted by $R[i^*(h)]$ and the

set of requirements captured by the lower level (more refined) i* model be denoted by $R[i^*(l)]$. The generalized notion of *relative completeness* can be represented as

$$R[i^*(h)] \subseteq R[i^*(l)] \tag{3.1}$$

The relative completeness relation of Eq. 3.1 can be ensured if we can define bridging rules of the form $p \leftrightarrow q$. These bridging rules represent statements that relate assertions made using the vocabulary of a higher level model (p) to assertions made using the vocabulary of the model at the level below (q). If the proposition q refers to a single element then these bridging rules are called *1-1 correlations* or *renaming rules*. If the proposition q refers to more than one element then such a rule defines a *1-Many correlation*. In the above case study, the following rules ensure that relative completeness of the hierarchic ontology is maintained.

(a) Science courses are partitioned into Pure Sciences and Engineering Sciences.

goal(Science Courses) ↔ *decomposes-to*(Pure Science Courses, Engineering Science Courses)

(b) Student Applications come from two types of students—Graduates and HS passed. Their selection criteria are also different. Students apply on the basis of the admission criteria that they satisfy.

goal(Students Application) ↔ *goal*(Admission criteria)

The rest of the chapter is organized as follows. Section 3.1 elaborates on the concept of hierarchic correlations of model elements and how they can be used to bridge the different i* models of the requirement refinement hierarchy. This is followed by the notion of relative completeness checking and their consequences in Sect. 3.2. The paper presents a possible heuristic mechanism for achieving bidirectional relative completeness in Sect. 3.3. Section 3.3.2 specifies how these heuristics may be used for relative completeness checking. The results from this discussion have been summarized as theorems in Sect. 3.3.3. The chapter concludes with Sect. 3.4.

3.1 Hierarchic Correlations

The higher levels of any requirement refinement hierarchy capture an abstract functioning of the enterprise. The i* models at the higher levels usually comprise of abstract Goals, Tasks and Resources, that require fine-tuning for requirement analyses. These requirements can be further refined and the lower levels of the requirement refinement hierarchy provide an elaborate understanding of the functioning of the enterprise. The i* perspective at these lower levels is far more detailed and consists of model elements having higher levels of granularity.

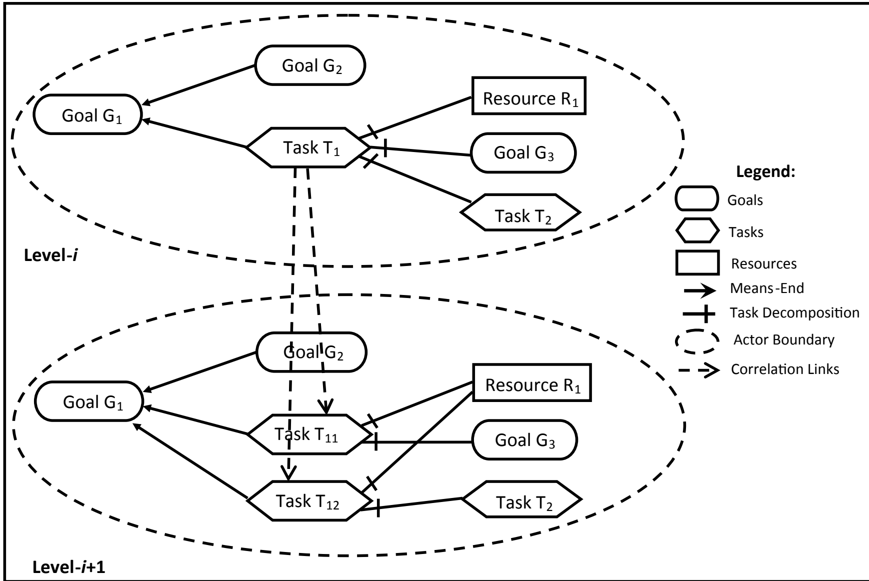


Fig. 3.3 Hierarchic correlation in an Actor Invariant refinement hierarchy

Hierarchic correlations try to bridge this gap in perspective. We assume that there are n levels in the requirement refinement hierarchy and the i* models at levels i and $(i + 1)$ can be defined as a hierarchic correlation of some model elements in level-i to a set of more refined model elements in level-$(i + 1)$. All the illustrations in this section model hierarchic correlations between adjacent levels of a goal model hierarchy. Although there may exist correlations between non-adjacent levels as well, we do not consider them in these examples. The formalisms proposed in these case studies can be extended to non-adjacent levels as well. Figure 3.3 illustrates hierarchic correlation of a task T_1 at level-i with tasks T_{11} and T_{12} at level-$(i + 1)$. The subtree of task T_1 (defined by all model elements that stem from T_1) in level-i is redistributed between tasks T_{11} and T_{12} in level-$(i + 1)$.

Model elements can be classified into four types based on their type of decomposition within a specific level. These are as follows:

1. *Only Child Links* (OCL). Model elements which are not the children of some other model element undergoing Task or Means-End Decomposition. However, the model element itself undergoes such a decomposition and is connected to other model elements using *child links*. In Fig. 3.4, goal G_1 at level-i is an OCL model element which has no parent and is connected to its children T_1 and G_2 using Means-End links.
2. *Only Parent Links* (OPL). Model elements which stem from some model element undergoing Task or Means-End Decomposition. However, the model element itself does not undergo any further decompositions. It is either independently

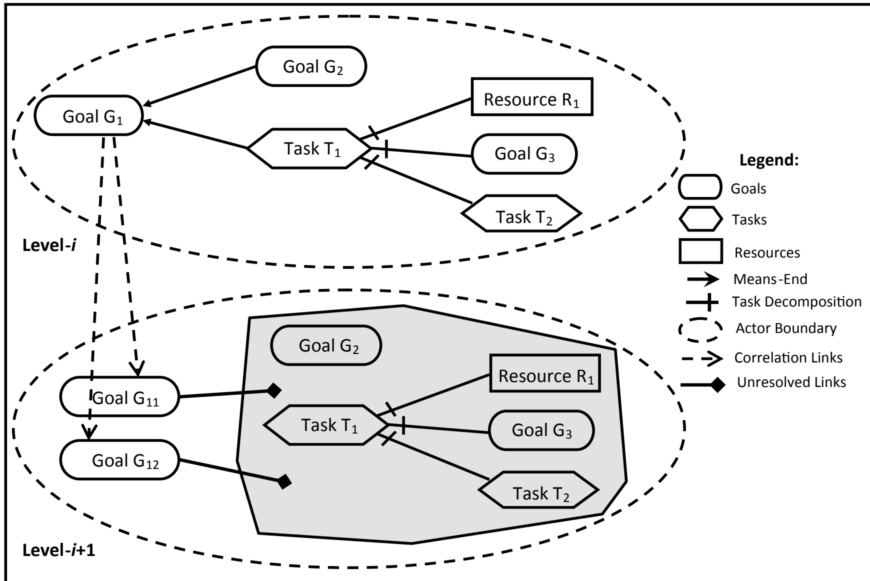


Fig. 3.4 Hierarchic OCL goal correlation in an Actor Invariant refinement hierarchy

achievable by the actor or dependent on some other actor for its fulfillment. It is connected to its parent only using *parent links*. In Fig. 3.5, goal G_2 at level-i is an OPL model element which is connected to its parent G_1 using a Means End Link and does not have any Child links.

3. *Parent and Child Links* (PCL). Model elements which are connected to their parent elements using *parent links* and also undergo Task or Means-End Decomposition to generate child elements, connected with *child links*. OPL and OCL model elements are special cases of PCL process elements that do not have parent links or child links. In Fig. 3.6, task T_1 at level-i is a PCL model element which is connected to its parent G_1 using a Means End Link and connects to its children G_3, T_2, and R_1 using Task Decomposition links.
4. *No Links* (NL). Stand-alone model elements that do not have associated *parent links* or *child links*.

The different types of goal correlation links suggested above are functionally complete and there cannot exist any other type of goal correlation. Hierarchic correlation rules need to be defined separately for each of these classes of model elements. Depending on the type of model element and the type of *parent* and *child links* (Task Decomposition links or Means End links), we can have different ways in which hierarchic correlations can be defined. So a PCL model element, having a Task Decomposition *parent link* and a Means-End *child link*, has a completely different hierarchic correlation as compared to another PCL model element that has Means End links as both its *parent link* and *child links*.

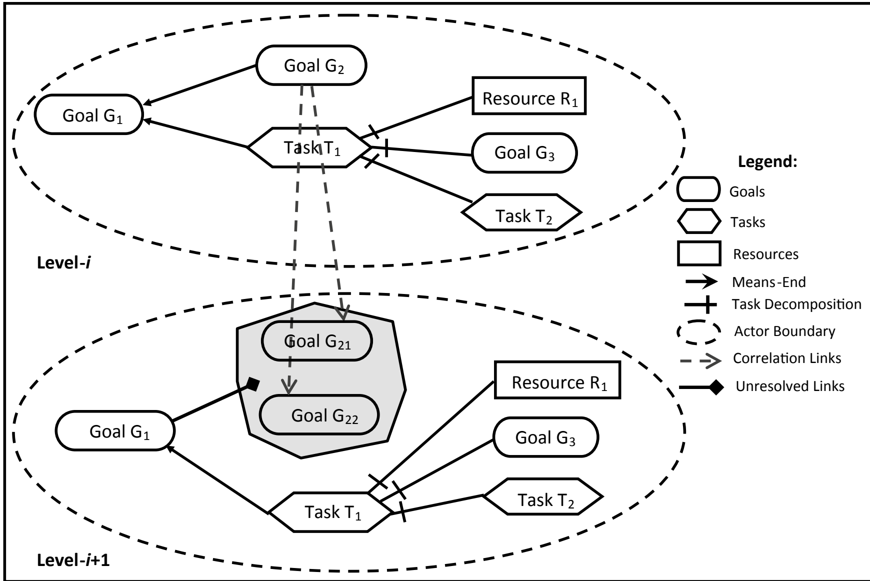


Fig. 3.5 Hierarchic OPL goal correlation in an Actor Invariant refinement hierarchy

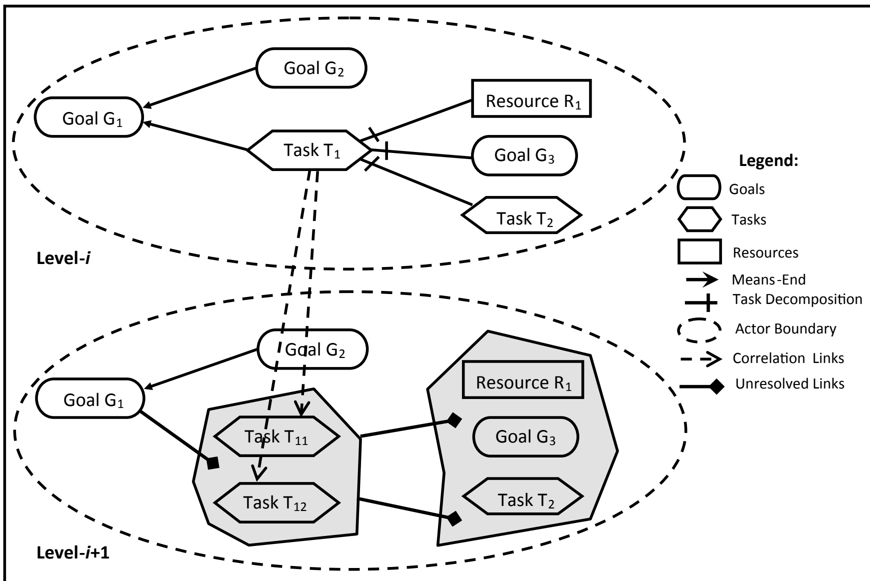


Fig. 3.6 Hierarchic PCL task correlation in an Actor Invariant refinement hierarchy

Hierarchic correlation of model elements need not be restricted within an actor. *Actor Invariant correlations* are characterized by the fact that the set of actors within the i* model remains the same and hierarchic correlations are established between high and low granularity model constructs within the same actor. These correlations try to establish whether lower level goals, tasks and resources collectively realize some higher level model construct while ensuring consistency. Figures 3.3, 3.4, 3.5 and 3.6 show hierarchic correlations in *Actor Invariant* settings.

An alternative to this is the idea of *Actor Variant correlations*. Such a refinement permits the existence of different sets of actors at different levels of the requirement refinement hierarchy. A goal, task, resource or dependency at a higher level may have a hierarchic correlation with lower level model constructs that are not restricted within the same actor. A correlation may establish the entailment of a high-level model construct from multiple low level model constructs that are distributed among multiple actors. Depending on how the hierarchically correlated model elements resolve their parent and child links, we may have new dependencies set up between this combined set of actors. Figures 3.1 and 3.2 show *Actor Variant hierarchic correlations* for the University Admission System case study.

Figure 3.7 illustrates another *Actor Variant hierarchic task correlation*. Task T_1 of actor A at level-i can be realized from tasks T_{11} and T_{12} belonging to actors A_1

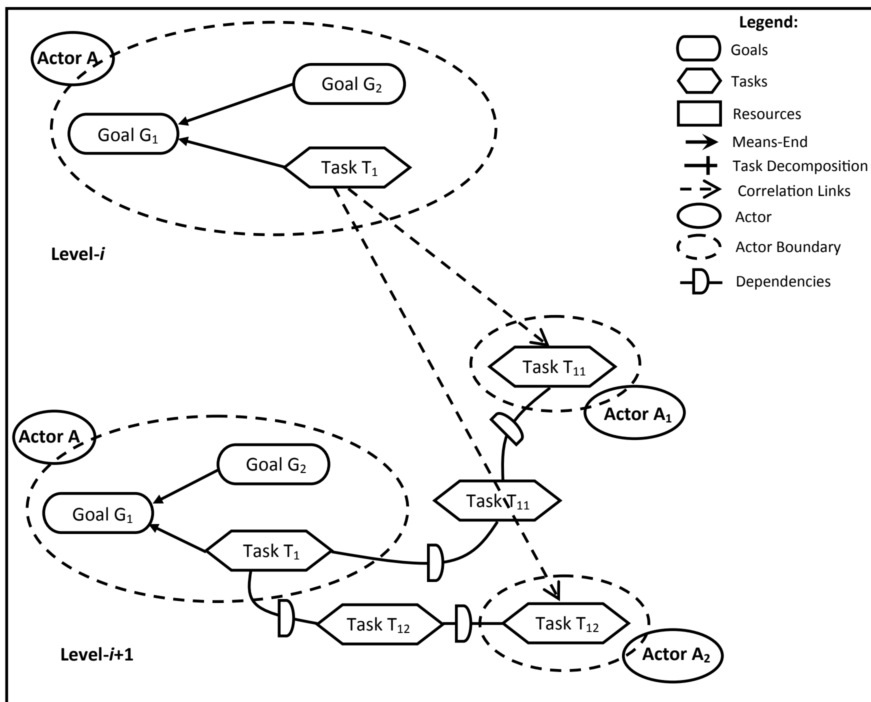


Fig. 3.7 Possible Hierarchic correlation in an Actor Variant refinement hierarchy

and A_2, respectively, at level-$(i + 1)$. The lower level requires dependencies to be set up between actors A, A_1 and A_2. Such a hierarchic correlation is valid, i.e., tasks T_{11} and T_{12} can exist independently, if and only if they are consistent with task T_1.

3.2 Relative Completeness Checking

Different levels of the requirement refinement hierarchy generate different i* perspectives of the features being delivered by the enterprise. The i* models generated from different levels of the hierarchy have completely different ontologies that are derived from the vocabulary of the enterprise architects. *Relative Completeness* of the requirement specifications are defined by the mapping of model elements between successive levels of the refinement hierarchy. Mapping between model constructs can be of two types:

(i) *1-1 Correlation*: Also called a *renaming rule*. A model element at some higher abstract level of a refinement hierarchy has a one-to-one mapping with some model element at the lower concretized level. The two different levels of the refinement hierarchy may have completely different ontologies and, hence, a one-to-one mapping across these levels represents a renaming of model constructs.

(ii) *1-Many Correlation*: A low granular, abstract model element at the higher levels of the refinement hierarchy has a one-to-many mapping with a subset of highly granular, concretized model elements participating in the lower level of the refinement hierarchy. 1-Many correlations have already been elaborated previously in Sect. 3.1.

(iii) *Many-Many Correlation*: A set of model elements at a higher level of the refinement hierarchy can be mapped to a subset of model elements at the lower level of the goal model hierarchy. In most cases, Many-Many correlations can be realized as a set of 1-1 and 1-Many correlations.

Based on these types of mapping that can exist between model elements at different levels of the refinement hierarchy, *Relative Completeness* can be of three types.

1. *Bidirectional Relative Completeness* (BRC). Adjacent levels of a requirement refinement hierarchy are said to satisfy Bidirectional Relative Completeness if and only if

   - Every model element at the higher level is either involved in a 1-1 correlation or a 1-Many correlation. This is represented by the following relation:

   $$R[i^*(h)] \subseteq R[i^*(l)]$$

   - AND, every model element at the lower level must be either involved in an *inverse* 1-1 correlation or an *inverse* 1-Many correlation. This is represented by the following relation:

$$R[i^*(l)] \subseteq R[i^*(h)]$$

These two relations can be combined to represent *BRC* using the following relation:

$$R[i^*(h)] \cong R[i^*(l)] \tag{3.2}$$

This is the ideal case where we can map all the model constructs appearing in some level of the requirement refinement hierarchy to all the model constructs appearing in the next adjacent level. We say that the relative completeness of the requirement refinement hierarchy holds in both directions.

2. *Unidirectional Relative Completeness* (URC). A two-level requirement refinement hierarchy is said to satisfy Unidirectional Relative Completeness if and only if

   - EITHER, all the model elements at the higher level can be mapped to a subset of lower level model elements using 1-1 or 1-Many correlations but not vice-versa. *URC* is said to exist from high to low and is represented by the following relation:

   $$R[i^*(h)] \subset R[i^*(l)] \tag{3.3}$$

   - OR, all the model elements at the lower level can be mapped to a subset of higher level model elements using *inverse* 1-1 or *inverse* 1-Many correlations but the reverse does not hold. *URC* is said to exist from low to high and is represented by the following relation:

   $$R[i^*(l)] \subset R[i^*(h)] \tag{3.4}$$

   The relative completeness of the requirements holds in only one direction of the requirement refinement hierarchy. Equations 3.3 and 3.4 define *URC* between adjacent levels of the hierarchy.

3. *Relative Incompleteness* (RI). A requirement refinement hierarchy that is neither *BRC* nor *URC* is said to be an *RI* requirement specification. A two-level requirement refinement hierarchy is said to satisfy Relative Incompleteness if and only if

   - Only a subset of the model elements appearing in the higher level of the refinement hierarchy can be mapped to a subset of model elements appearing in the lower level of the hierarchy. This is represented by the following relation:

   $$R[i^*(h)] \not\subseteq R[i^*(l)]$$

   - AND, only a subset of the model elements appearing in the lower level of the refinement hierarchy can be mapped to a subset of model elements appearing in the higher level of the hierarchy. This is represented by the following relation:

$$R[i^*(l)] \nsubseteq R[i^*(h)]$$

These two relations can be combined to represent *RI* within a requirement refinement hierarchy as

$$R[i^*(l)] \ncong R[i^*(h)] \tag{3.5}$$

3.2.1 Consequence of Relative Completeness

Relative Completeness Checking plays a pivotal role in establishing the *equivalence* and *traceability* of model elements residing within the i* models obtained from different levels of the requirement refinement hierarchy. *Bidirectional Relative Completeness* ensures that adjacent levels of the refinement hierarchy are in synchronization with each other. If *BRC* is ensured between all pairs of adjacent levels of the refinement hierarchy, then we can safely conclude that the entire requirement refinement hierarchy is in sync, i.e., the chances of errors of omission in the requirement specifications is mostly eliminated.

Unidirectional Relative Completeness ensures a one-way synchronization of requirement models in the requirement refinement hierarchy. If relation 3.3 is satisfied, then there exists some model elements in the lower level i* model which cannot be traced back to the previous level. Relation 3.4 represents the exact opposite condition. In either case, we can conclude that there is partial synchronization within the refinement hierarchy and the unmapped model elements need to be justified and scrutinized regarding their existence. URC is a likely scenario as lower levels of the hierarchy capture fine-grained requirements that are not perceived by the higher levels.

The ambiguity of checking relative completeness lies in the fact that *Relatively Incomplete* multilevel i* model specifications cannot be outright discarded as incorrect. It is possible that two i* models, representing the requirement specifications refined at two different levels of the refinement hierarchy, be *Relatively Incomplete* with respect to one another. The consequence of *Relative Incompleteness* lies in the fact that adjacent levels of the refinement hierarchy are not in sync with each other. This demands careful introspection by the enterprise architects to ensure the correctness of the requirement refinement hierarchy. Rigorous team meetings and discussions need to be conducted for revisiting the requirements and ensuring relative completeness manually.

3.3 Possible Heuristics

Relative Completeness Checking is not as trivial as it seems. We need to define heuristics for the hierarchic correlation of goals, tasks, resources and dependencies in both Actor Invariant and Actor Variant environment settings. These rules must

be individually spelled out for NL, OPL, OCL and PCL types of model elements as defined in Sect. 3.1. The correlation varies with the type of *parent* and *child links*. A hierarchic OPL goal correlation behaves in an entirely different manner as compared to a hierarchic PCL goal correlation. Again, a hierarchic OCL goal correlation with Means-End links as *child links* is quite different from a hierarchic OCL goal correlation with Task Decomposition links as *child links*. Once heuristics have been defined for all possible types of hierarchic correlations, we can proceed towards *Relative Completeness Checking*.

Let us look back into our case study that models the requirements of a University Admission System. Figure 3.1 captures an abstract i* model that occupies a position somewhere near the top of the requirement refinement hierarchy. *Science Courses* is a high-level goal that can be broken down to lower level goals *Pure Science Courses* and *Engineering Science Courses*. Such a 1-Many correlation is captured in Fig. 3.8. However, the trickier part is to accommodate such a 1-Many correlation within the lower level i* model by suitable adjustment of the decomposition links. We need to resolve how the higher granular goals can be connected to the existing model elements using Task Decomposition and Means End links.

Before proposing a possible heuristic for defining such a 1-Many correlation, we need to understand the semantic consequences of such a 1-Many correlation. We can make the following observations:

I. Previously the *University* was offering *Arts Courses* and *Science Courses*. The refined requirement suggests that the *University* now offers *Arts Courses, Pure Science Courses,* and *Engineering Science Courses*. Thus, both *Pure Science Courses* and *Engineering Science Courses* get connected to *Postgraduate Degrees Offered* using Task Decomposition links.
II. Previously *Science Courses* was *Managed by Faculty*. It follows logically that both *Pure Science Courses* and *Engineering Science Courses* will also be *Managed by Faculty*. Hence, again, both these higher granular goals get connected to *Managed by Faculty* using Task Decomposition links.
III. Previously *Student Applications* could be made to *Arts Courses* or *Science Courses*. After this 1-Many correlation, the lower level of the refinement hierarchy should capture *Student Applications* being made to *Arts Courses, Pure Science Courses,* or *Engineering Science Courses*. So both *Pure Science Courses,* and *Engineering Science Courses* get connected to *Student Applications* using Means End links.

These three semantic observations help us to obtain the i* model at the lower level of the refinement hierarchy and consisting of model elements having greater granularity. The corresponding i* model, thus, obtained is shown in Fig. 3.9. This i* model captures the requirements at some level of the requirement refinement hierarchy that lies between the granular levels captured by Figs. 3.1 and 3.2.

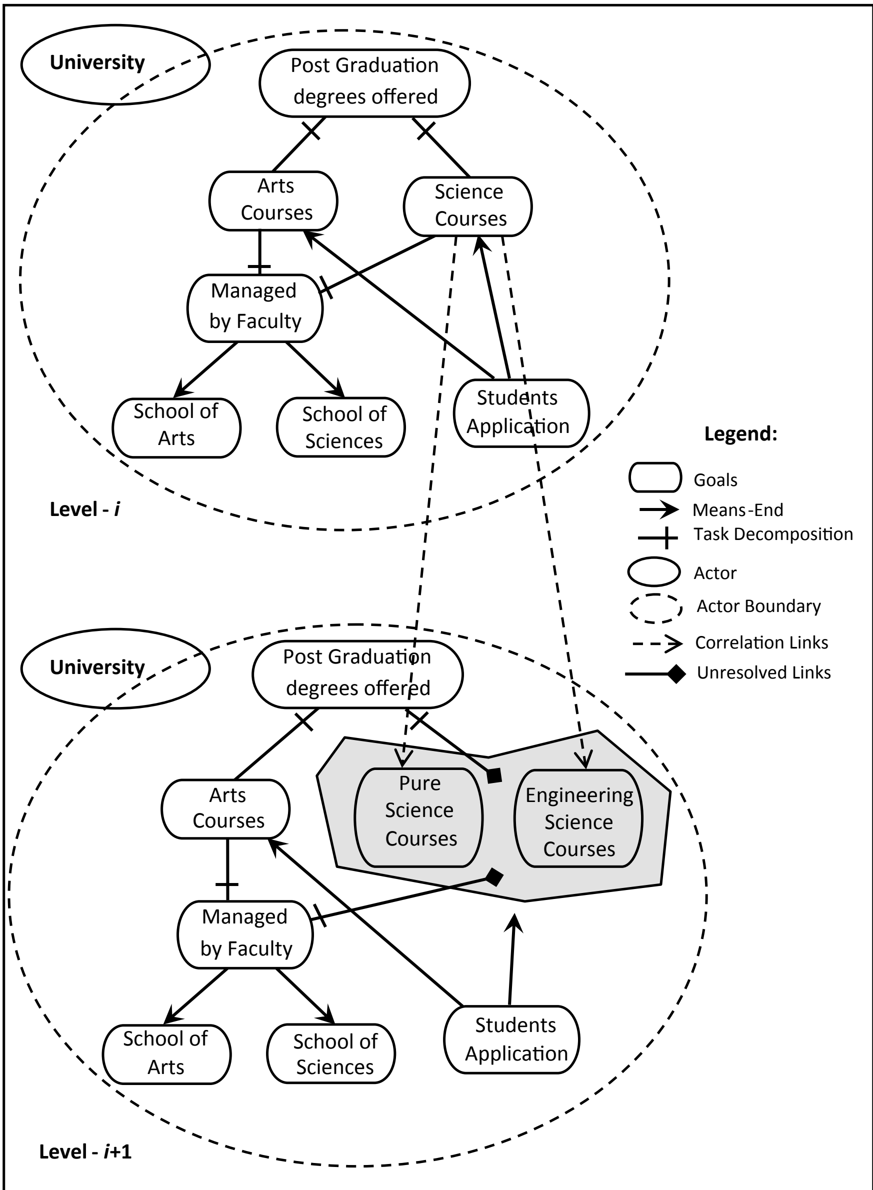


Fig. 3.8 1-Many correlation between lower granular goal *Science Courses* and higher granular goals *Pure Science Courses* and *Engineering Science Courses*

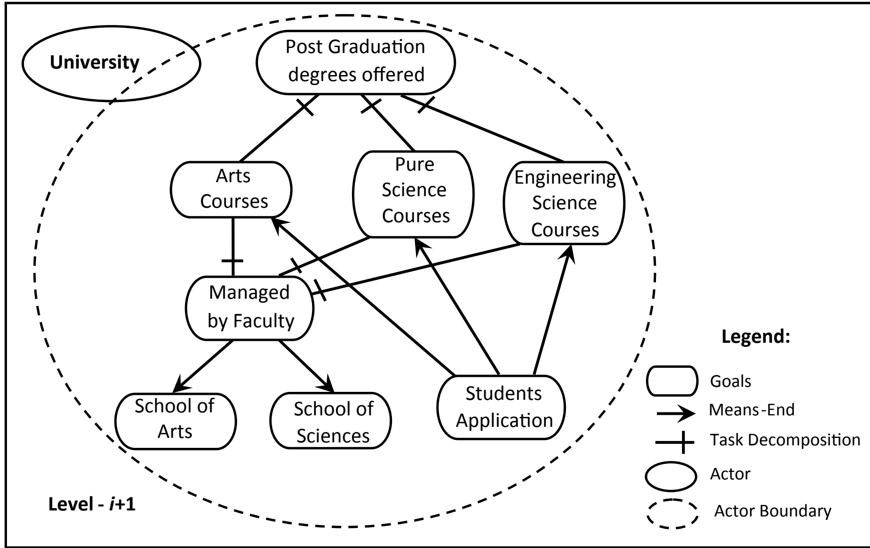


Fig. 3.9 The i* model obtained for the lower levels of the requirement refinement hierarchy after incorporating the 1-Many correlation

3.3.1 Formalizing the Heuristics

Considering the formal annotation of i* models suggested in [3], let MEL^i_{Univ} and TDL^i_{Univ} represent the set of Means End links and Task Decomposition links within the *University* actor at level-i, respectively, such that

- Every Means End link is represented by propositions of the form *ME(*$\{P_1, P_2, \ldots, P_{m-1}\}, P_m$*)* such that $\{P_1, \ldots, P_{m-1}\}$ form the means to achieve the end P_m.
- Every Task Decomposition link is represented by propositions of the form *TD(*$\{P_1, P_2, \ldots, P_{n-1}\}, P_n$*)* such that P_n is decomposed to $\{P_1, \ldots, P_{n-1}\}$.

According to Fig. 3.1, the *University* actor has the following Means End links and Task Decomposition links at level-i:

(i) MEL^i_{Univ} = {*ME({*Belongs to *School of Arts*, Belongs to *School of Sciences}, Faculty)*, *ME({*Apply for *Arts Courses*, Apply for *Science Courses}, Students Application)*}.
(ii) TDL^i_{Univ} = {*TD({*Offer *Arts Courses*, Offer *Science Courses}, Post-graduation Degrees Offered)*, *TD({*Manage *Arts Courses*, Manage *Science Courses}, Managed by Faculty)*, *TD({*Organize *Arts Courses}, School of Arts)*, *TD({*Organize *Science Courses}, School of Sciences)*}.

Now we have to incorporate the 1-Many correlation shown in Fig. 3.8 within our formal model. The 1-Many correlation show in Fig. 3.8 can be formally represented using the relation:

goal(Science Courses) ↔ *decomposes-to*(Pure Science Courses, Engineering Science Courses)

Assuming that except these newly introduced ontologies all existing ontologies remain unchanged, we can obtain the formal representation of the i* model shown in Fig. 3.9 as follows:

1. The Goal Set (GS) at level-($i + 1$) is obtained from the higher level Goal Set as follows:

   $\text{GS}^{i+1}_{Univ} = \text{GS}^{i}_{Univ} -$ {Science Courses} $\bigcup$ {Pure Science Courses, Engineering Science Courses}

2. The Means End links at level-($i + 1$) can be formalized as follows:

   $\text{MEL}^{i+1}_{Univ} = \text{MEL}^{i}_{Univ} -$ {*ME({*Apply for *Arts Courses*, Apply for *Science Courses}, Students Application)*} $\bigcup$ {*ME({*Apply for *Arts Courses*, Apply for *Pure Science Courses*, Apply for *Engineering Science Courses}, Students Application)*}

3. The Task Decomposition links at level-($i + 1$) can be formally defined as follows:

   $\text{TDL}^{i+1}_{Univ} = \text{TDL}^{i}_{Univ} -$ {*TD({*Offer *Arts Courses*, Offer *Science Courses}, Post-graduation Degrees Offered)*, *TD({*Manage *Arts Courses*, Manage *Science Courses}, Managed by Faculty)*, *TD({*Organize *Science Courses}, School of Sciences)*} $\bigcup$ {*TD({*Offer *Arts Courses*, Offer *Pure Science Courses*, Offer *Engineering Science Courses}, Post-graduation Degrees Offered)*, *TD({*Manage *Arts Courses*, Manage *Pure Science Courses*, Manage *Engineering Science Courses}, Managed by Faculty)*, *TD({*Organize *Pure Science Courses*, Organize *Engineering Science Courses}, School of Sciences)*}

In general, let MEL^i_j and TDL^i_j represent the set of Means End links and Task Decomposition links for the actor A_j at level-i, respectively. Let us suppose that there exists an OPL goal P_k that participates in a Means-End as well as Task decomposition. Then there exist propositions of the form

$$ME(\{P_1, P_2, \ldots, P_k, \ldots, P_{m-1}\}, P_m) \in MEL^i_j$$

AND

$$TD(\{P_A, P_B, \ldots, P_k, \ldots, P_R\}, P_{S}) \in TDL^i_j$$

The OPL Goal model element P_k participates as a child in both Task decompositions and Means-End decompositions. If P_k at level-i has a 1-Many correlation with model elements $P_{k1}, \ldots, P_{kn}$ at level-($i + 1$), then both the Means-End decomposition and Task decomposition sets can be refined and expanded to incorporate the model elements $P_{k1}, \ldots, P_{kn}$. However, the 1-Many correlation must satisfy the *Entailment* and *Consistency* conditions [1] of the KAOS framework. In such a situation, we can define the heuristics for 1-Many correlation of OPL model elements as follows:

Heuristic (HR#1): When an OPL model element, having both Means End links and Task Decomposition links as its *parent link*, establishes a 1-Many correlation with a set of model elements at a lower level of the requirement refinement hierarchy, then the formal representation of the i* model gets refined as

(i) $G_j^{i+1} = G_j^i - \{P_k\} \bigcup \{P_{k_1}, \ldots, P_{kn}\}$
(ii) $MEL_j^{i+1} = MEL_j^i\text{-}ME(\{P_1, P_2, \ldots, P_{k-1}, P_k, P_{k+1}, \ldots, P_{m-1}\}, P_m) \bigcup ME(\{P_1, P_2, \ldots, P_{k-1}, P_{k_1}, \ldots, P_{k_n}, P_{k+1}, \ldots, P_{m-1}\}, P_m)$
(iii) $TDL_j^{i+1} = TDL_j^i\text{-}TD(\{P_A, P_B, \ldots, P_J, P_k, P_L, \ldots, P_R\}P_S) \bigcup TD(\{P_A, P_B, \ldots, P_J, P_{k_1}, \ldots, P_{k_n}, P_L, \ldots, P_R\}, P_S)$

Similarly, we can define *1-Many correlation heuristics* for different types of model elements under varying combinations of *parent–child* link types.

3.3.2 Applying Heuristics for Relative Completeness Checking

Let the *1-Many correlation* heuristics be denoted by HR#1, HR#2, …, HR#N. For each such heuristic, HR#i^{-1} denotes the *inverse* 1-Many correlation heuristic. Along with this set, we also have a set of *1-1 correlation heuristics* denoted by OC#1, OC#2, …, OC#M, where each OC#j represents a renaming rule. As with 1-Many correlation heuristics, OC#j^{-1} denotes the *inverse* 1-1 correlation heuristic of OC#j. Let I_k^* denote the i* model at level-k and I_{k+1}^* denote that at level-($k+1$). *Relative Completeness Checking* can then be defined as follows:

Bidirectional Relative Completeness (BRC). I_k^* and I_{k+1}^* are said to be *BRC*-compliant iff

1. I_{k+1}^* can be derived from I_k^* by first applying all 1-1 correlation heuristics to model elements in I_k^*, in any random sequence, to obtain $I_k^{*\prime}$ and then applying all 1-Many correlation heuristics to $I_k^{*\prime}$ in any random sequence. This can be represented as follows:

$$\forall_{j=1}^{M}\ I_k^* \overset{OC\#j}{\Rightarrow} I_k^{*\prime}, \text{ and}$$
$$\forall_{i=1}^{N}\ I_k^{*\prime} \overset{HR\#i}{\Rightarrow} I_{k+1}^*$$

AND

2. I_k^* can be similarly derived from I_{k+1}^* by first applying the *inverse* 1-Many correlation heuristics to I_{k+1}^*, in any random sequence, to obtain $I_k^{*\prime}$ and then applying the *inverse* 1-1 correlation heuristics to $I_k^{*\prime}$ in any random sequence. This can be represented as follows:

$$\forall_{i=1}^{N}\ I_{k+1}^* \overset{HR\#i^{-1}}{\Rightarrow} I_k^{*\prime}, \text{ and}$$
$$\forall_{j=1}^{M}\ I_k^{*\prime} \overset{OC\#j^{-1}}{\Rightarrow} I_k^*$$

The keyword AND is of utmost importance for bidirectional completeness. Conditions (1) and (2) may be combined and we can say that both the i* models I_k^* and I_{k+1}^* are mutually derivable from each other. This is represented as follows:

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_k^* \underset{HR\#i^{-1} \wedge OC\#j^{-1}}{\overset{HR\#i \wedge OC\#j}{\Leftrightarrow}} I_{k+1}^*$$

Unidirectional Relative Completeness (URC). I_k^* and I_{k+1}^* are said to be *URC*-compliant iff

1. I_{k+1}^* can be derived from I_k^* by applying 1-1 correlation heuristics and then 1-Many correlation heuristics to a subset of the model elements in I_k^* in any random sequence. However, I_k^* cannot be derived back from I_{k+1}^* by applying the *inverse* 1-1 and *inverse* 1-Many correlations, successively. The following relations represent *URC*-compliance from I_{k+1}^* to I_k^*:

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_k^* \underset{OC\#j}{\overset{HR\#i}{\Rightarrow}} I_{k+1}^*, \text{ and}$$
$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_{k+1}^* \underset{OC\#j^{-1}}{\overset{HR\#i^{-1}}{\nRightarrow}} I_k^*$$

   OR
2. I_k^* can be derived from I_{k+1}^* by applying *inverse* 1-Many correlation heuristics and then *inverse* 1-1 correlation heuristics to a subset of model elements in I_{k+1}^* in any random sequence. However, I_{k+1}^* cannot be derived from I_k^* by applying the corresponding 1-1 and 1-Many correlations, successively. The following relations represent *URC*-compliance from I_k^* to I_{k+1}^*:

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_{k+1}^* \underset{OC\#j^{-1}}{\overset{HR\#i^{-1}}{\Rightarrow}} I_k^*, \text{ and}$$
$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_k^* \underset{OC\#j}{\overset{HR\#i}{\nRightarrow}} I_{k+1}^*$$

URC-compliance and *BRC*-compliance looks very similar. The difference lies in the keyword OR. Unidirectional relative completeness does not allow the mutual derivation of adjacent level i* models.

Relative Incompleteness (RI). I_k^* and I_{k+1}^* are said to be *RI*-compliant iff

1. I_k^* and I_{k+1}^* are not *BRC*-compliant.
2. I_k^* and I_{k+1}^* are not *URC*-compliant in either direction.

RI-compliance does not imply that there does not exist any 1-1 or 1-Many correlations between I_k^* and I_{k+1}^*. What Relative Incompleteness actually signifies is that only a subset of model elements in I_k^* has correlations with only a subset of model elements in I_{k+1}^*. Both I_k^* and I_{k+1}^* contain model elements that cannot be mapped through some 1-1 or 1-Many correlations. Thus, neither I_k^* nor I_{k+1}^* can derive the other with the successive random application of all 1-1 and 1-Many correlation heuristics. The following relations capture the notion of *RI*-compliance:

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_k^* \overset{HR\#i}{\underset{OC\#j}{\not\Rightarrow}} I_{k+1}^*, \text{ and}$$

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_{k+1}^* \overset{HR\#i^{-1}}{\underset{OC\#j^{-1}}{\not\Rightarrow}} I_k^*$$

The above conditions may be combined and we can say that neither of the i* models I_k^* and I_{k+1}^* are derivable from the other. This is represented as follows:

$$\forall_{i=1}^{N} \forall_{j=1}^{M} I_k^* \overset{HR\#i \wedge OC\#j}{\underset{HR\#i^{-1} \wedge OC\#j^{-1}}{\not\Leftrightarrow}} I_{k+1}^*$$

3.3.3 Results

Let $|I_k^*|$ and $|I_{k+1}^*|$ represent the number of model elements in the i* model at levels k and $k+1$, respectively. Let there be M 1-1 correlations and N 1-Many correlations that can be established between the i* models at these two adjacent levels. Each 1-1 correlation is a one-to-one mapping. 1-Many correlations, on the hand, are one-to-many mappings. Let $|HR\#i|$ denote the number of model elements in I_{k+1}^* that have a 1-Many correlation with some model element in I_k^*. Under these assumptions, we can have the following theorems.

Theorem 3.1 *I_k^* and I_{k+1}^* are BRC-compliant if and only if:*

(i) $|I_k^*| = (M + N)$, *and*
(ii) $|I_{k+1}^*| = (M + \sum_{i=1}^{N} |HR\#i|)$.

Proof *BRC*-compliance implies that all model elements in I_k^* must participate in either 1-1 correlations or 1-Many correlations. The total number of these correlations is given by $(M + N)$. This proves condition (i). Similarly, every model element in I_{k+1}^* must be a part of some *inverse* correlation. Since every 1-1 correlation maps single model elements and every 1-Many correlation maps $|HR\#i|$ model elements, the total number of elements participating in *inverse* correlations is given by $(M + \sum_{i=1}^{N} |HR\#i|)$. This proves condition (ii). ■

Theorem 3.2 *I_k^* and I_{k+1}^* are URC-compliant if and only if:*

(i) $|I_k^*| > (M + N)$ *and* $|I_{k+1}^*| = (M + \sum_{i=1}^{N} |HR\#i|)$
OR
(ii) $|I_k^*| = (M + N)$ *and* $|I_{k+1}^*| > (M + \sum_{i=1}^{N} |HR\#i|)$.

Proof *URC*-compliance from I_{k+1}^* to I_k^* demands that all model elements in I_{k+1}^* participate in *inverse* correlations but the same is not true for model elements in I_k^*. This implies that there exists excess model elements in I_k^* which do not participate in either type of correlation. This is established by condition (i).

URC-compliance from I_k^* to I_{k+1}^* demands that all model elements in I_k^* participate in either 1-1 correlations or 1-Many correlations but the same is not true for model

elements in I^*_{k+1}. This implies that there exists excess model elements in I^*_{k+1} which cannot be correlated back to model elements in I^*_k. Condition (ii) captures these criteria. ■

Theorem 3.3 *I^*_k and I^*_{k+1} are RI-compliant if and only if*

(i) $|I^*_k| > (M + N)$, *and*
(ii) $|I^*_{k+1}| > (M + \sum_{i=1}^{N} |HR\#i|)$.

Proof *RI*-compliance implies that both I^*_k and I^*_{k+1} have model elements that do not participate in either 1-1 correlations or 1-Many correlations. Thus, both the i* models have excess model elements that cannot be bridged and results in the inequalities of conditions (i) and (ii). ■

Once we can establish correlations between adjacent levels of a goal model hierarchy, we can apply the theorems as a formal basis for evaluating the degree of relative completeness that is being achieved. Based on the degree of relative completeness, enterprise architects can make decisions to increase the degree of correlation existing within the goal model hierarchy.

3.4 Conclusion

The research in this chapter contributes to the domain of requirements engineering by defining bridging constraints that enable hierarchic ontology integration of multilevel i* models. The novelty of this proposition lies in the fact that these bridging mechanisms help in stitching the i* models of the entire requirement refinement hierarchy. This chapter builds on this idea and provides some insights on how *relative completeness* can be ensured between adjacent tiers of multilevel i* refinement hierarchies. Rules have been defined for checking the degree of *relative completeness* as well. Bidirectional Relative Completeness (BRC) is suggested as the most ideal relative completeness criteria. The framework also proposes a *heuristic* for defining 1-Many correlations. We also state how such heuristics may be used to check relative completeness within refinement hierarchies and how the consequences may be utilized.

The ability to mine goal models also has important implications for requirements engineering, as well as a wide variety of other settings that benefit from goal modelling. The machinery that we present can, therefore, provide useful directions for future research and development. This machinery can also be used to mine know-how which can support enterprise innovation strategies in significant ways. The empirical evaluation presented in this chapter is preliminary in nature, but provides evidence that suggests that there is merit in pursuing this general approach.

References

1. van Lamsweerde A, Darimont R, Letier E (1998) Managing conflicts in goal-driven requirements engineering. Trans Softw Eng Spec Issue Inconsistency Manag Softw Dev 24(11):908–926. https://doi.org/10.1109/32.730542
2. Yu E. Modelling strategic relationships for process reengineering, Ph.D. thesis University of Toronto, Toronto, Canada
3. Guizzardi R, Franch X, Guizzardi G, Wieringa R (2013) Using a foundational ontology to investigate the semantics behind the concepts of the i* language. In: Proceedings of the 6th international i* workshop (iStar 2013), pp 13–18

Chapter 4
Model Checking with i*

Hierarchic correlation between adjacent levels of a goal model hierarchy only ensures the synchronization between different levels of the enterprise hierarchy. It does not ensure the compliance of goal models to business compliance rules. Model checkers or verifiers can do this type of analysis. Model checking is a method for formally verifying finite state concurrent systems represented by extended finite state models. Industry standard model checking tools—like SPIN [1], NuSMV [2]—accept these extended finite state models (E-FSM) as input. The input models are defined by a set of state transitions that characterize the possible execution traces that the system can generate. The model checking tools are also fed with specifications about the system, expressed using temporal logic. Efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Thus, either a positive acknowledgement is generated, if the specification is satisfied, or a counterexample is produced, if the specification is violated. Figure 4.1 illustrates the general working mechanism of model checkers.

Requirement models capture the requirement specifications of the system and have the same impact as design models have on the coding phase [3]. Requirement models are generated in the requirements analysis phase of software development. This is the first phase of developing a software or system, irrespective of the particular development life cycle model being followed—*Waterfall, Spiral, Prototype* or *Agile*. Requirement models can help enterprise architects and developers by allowing them to perform different kinds of analysis on the system being developed. Model checking against a given set of temporal properties (see Fig. 4.1) is also an important type of analysis that may be performed on goal-oriented requirement models.

Even after a system has been deployed, its environment keeps changing. This results in the user requirements to change as well. The system has to adapt to the ever-changing user needs during runtime. Before incorporating the newly evolved requirements into the existing framework, developers and architects need to ensure that the changing requirements do not result in conflicting/inconsistent states within the system. Thus, some kind of model checking needs to be done on the changed requirements to ensure that the system will remain consistent after the changes take

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_4

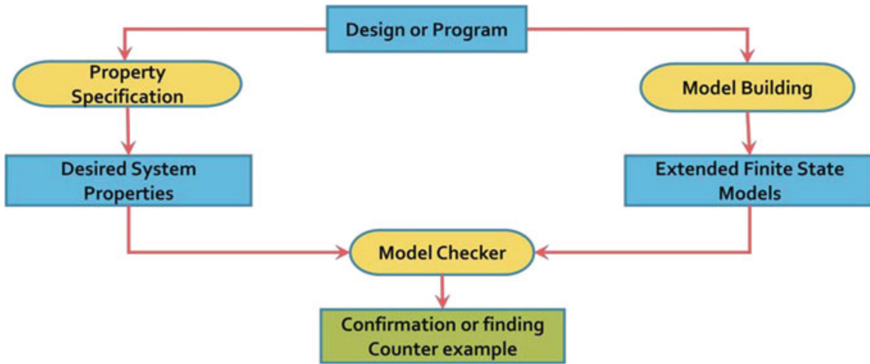


Fig. 4.1 Block diagram of standard model verifiers

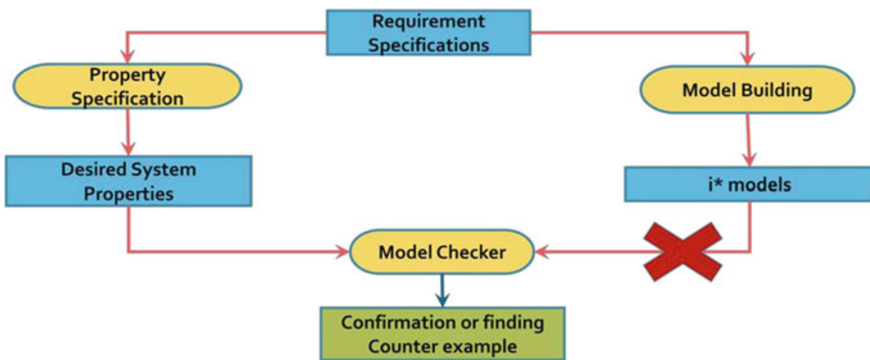


Fig. 4.2 Problem with i* model verification

effect. This may be done with industrial model checkers, if they are fed with the updated requirement models and the evolved requirement specifications as shown in Fig. 4.1.

i* is a goal-oriented requirements modelling notation that models requirements with the help of actors and their goals, tasks and resources. Inter-actor dependencies are also captured by the i* framework. An inherent attribute of the i* notation (and goal models in general) is that it is sequence agnostic and does not capture any sort of partial ordering between the goals and tasks. In the absence of sequencing information, standard industrial model checkers cannot verify i* models against temporal property specifications as shown in Fig. 4.2. Model checkers accept extended finite state models (E-FSM) as input for verifying temporal properties. Process models, sequence diagrams or activity diagrams, capture some ordering of states within the system and, hence, E-FSM(s) can be easily derived from these models. The process of extracting E-FSM(s) from i* models is far more complex as i* models do not capture state transitions within the system. This is the underlying research question being addressed in this chapter and is the main motivation behind this research.

Formal Tropos introduces the concept of actor instances and how dependencies, assertions, possibilities and invariants can exist in either of three states—*Not Created*, *Created Not Fulfilled* and *Fulfilled* [4, 5]. Formal Tropos associates the Tropos methodology to a formal specification language that allows the specification of constraints, invariants, pre- and post-conditions, thereby capturing the semantics of the i* graphical models. We extend this notion to i* model constructs in general and state that every goal, task or resource also exists in either of these three states. Every model construct makes two state transitions to reach the *Fulfilled* state from the *Not Created* state. The *Naïve Algorithm* uses a brute-force method to generate all possible finite state models that can be obtained by permuting the state transitions of individual model constructs. This results in an explosion within the finite state model space.

It is interesting to observe that, although an i* model is sequence agnostic, yet there exists some features or model constructs within the i* model that provide a temporal insight into the underlying requirements of the enterprise. For instance, every dependency has a *cause-effect property* in the sense that it is only when a dependee satisfies a requirement of the depender does the dependency become fulfilled. The *Semantic Implosion Algorithm* identifies these untapped temporal characteristics and tries to contain the rate of growth of the finite state model space corresponding to an i* model. Simulation results reveal that the *Semantic Implosion Algorithm* indeed outperforms the *Naïve algorithm* and provides a drastic improvement over the brute-force method.

The rest of the chapter is structured as follows. The next section (Sect. 4.1) details out the *Naïve Algorithm* and the *Semantic Implosion Algorithm*. The drawbacks of the *Naïve Algorithm* are identified and the *Semantic Implosion Algorithm* is proposed as a solution to these drawbacks. Section 4.2 describes a detailed simulation where both the algorithms are applied to the same classes of i* models and their performances are observed, compared and contrasted. Section 4.3 presents the `i*ToNuSMV` tool that we have developed for implementing the *Semantic Implosion Algorithm* and performing model checks on i* models. concludes the paper. This is followed by the version history and web links of the tool in Sects. 4.4 and 4.5, respectively. Finally, we conclude the chapter in Sect. 4.6.

4.1 Developing Finite State Models from an i* Model

The main research motivation is to analyse an i* model and derive a finite state model (FSM) that captures all the finite execution sequences that satisfy the given i* model. Without identifying a partial ordering of the operations within the enterprise, it becomes very difficult to check and verify temporal properties and compliance rules on the system. The underlying challenge of this work lies in the fact that i* models are sequence agnostic. Being complementary to the notion of FSMs, which define an ordering of states through which the system can go through, the conversion process cannot yield a unique execution trace corresponding to a given i* model. The idea here is to generate all the possible execution traces that satisfy the requirement

specifications captured in the given i* model. The algorithms presented in this article produce a finite state model space, as output, which defines this set of plausible finite state sequences. Once the finite state model space is obtained, we can apply model checking and generate a subset of this model space which satisfies all the compliance rules necessary for the operation of the enterprise. This final set of pruned finite state sequences can then be reverted back to the enterprise owner in order to verify the requirements.

In the following algorithms we are considering the more detailed strategic rationale (SR) diagram of an i* model. The SR-diagram is much more comprehensive than its strategic dependency (SD) counterpart and encompasses all the dependency information that are captured in the SD-diagram. In fact, an SD-diagram only represents the inter-actor dependencies but does not depict which particular model construct of the depender is dependent on which particular model construct of the dependee. The SR model is much more elaborate in this sense.

4.1.1 The Naïve Algorithm

The most intuitive solution that has been given by the GORE community uses the notion that every i* model element can exist in either of three possible states—*Not Created (NC), Created Not Fulfilled (CNF),* and *Fulfilled (F).* All the goals, tasks and resources that appear in the SR model are initially in the *Not Created (NC)* state and every model construct must make two transitions to reach the *Fulfilled (F)* state while transitioning through the intermediate *Created Not Fulfilled (CNF)* state. Unlike Formal Tropos [5], we do not consider multiple instances of goals, tasks or resources. We assume single instances and derive a finite state model corresponding to the given i* model. We obtain sequences of states by considering all possible permutations of the model elements and the states in which they exist.

Let us demonstrate the above concept with an example. Consider the simplest possible SR-diagram with one actor consisting of only one goal G. This is shown in Fig. 4.3a. The goal G can be in either of three states—*Not Created* denoted by $G_{(N)}$, *Created Not Fulfilled* denoted by $G_{(C)}$ and *Fulfilled* denoted by $G_{(F)}$. These three states give rise to 3! finite state sequences as shown in Fig. 4.3b–g. However, out all these six finite state models, only Fig. 4.3b is semantically correct. All the other finite state models are semantically inconsistent as a model element can go through its possible states in exactly one possible sequence—$G_{(N)} \rightarrow G_{(C)} \rightarrow G_{(F)}$. We call this sequence the *default sequence*, and must be satisfied by all model elements. Now, let us increase the complexity by incorporating one more model element in the SR-diagram, i.e., let there exist two model elements in the SR-diagram. These two model elements can belong to the same actor or to two different actors. In either case, the complexity analysis remains the same.

Let A_1 and A_2 be two different actors, each with a single goal node G_1 and G_2, respectively. Since each goal can be in either of three states, the total number of possible combined states is 3^2 (=9). However, since both G_1 and G_2 must

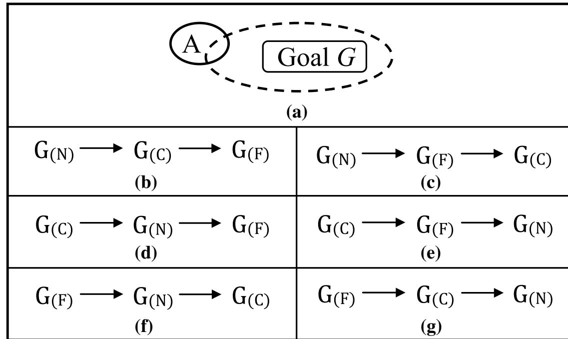


Fig. 4.3 **a** Actor A with a single goal G; **b** The only semantically correct finite state sequence; **c**–**g** Other possible finite state sequences that can be derived by permuting the state space but which are semantically incorrect

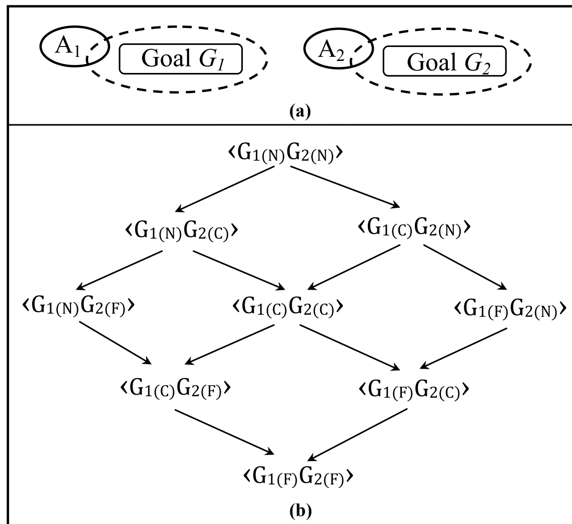


Fig. 4.4 **a** Actors A_1 and A_2 with goals G_1 and G_2, respectively; **b** The State Sequence Graph over the set of $3^2 = 9$ possible states

individually satisfy the *default sequence*, it becomes interesting to enumerate the valid state transition sequences that do not violate the *default sequence* of individual model elements. We draw a *State Sequence Graph* that captures all possible valid state transition sequences from the source node—denoted by $(G_{1(N)}\ G_{2(N)})$—to the destination node—denoted by $(G_{1(F)} G_{2(F)})$. Figure 4.4b illustrates the *State Sequence Graph* for these two goals.

The *State Sequence Graph* has all the nine possible combined state representations as vertices. These vertices are connected in the form of a lattice as all state transitions do not satisfy the *default sequence*. Each path in the *State Sequence Graph*, from the source node $(G_{1(N)}\mathrm{G}_{2(N)})$ to the destination node $(G_{1(F)}\ G_{2(F)})$, defines a semantically valid sequence of state transitions. In other words, each such path represents a finite state sequence corresponding to the given i* model. Thus, with two model elements, we obtain six different finite state sequences that satisfy the *default sequences* of the individual model elements.

4.1.1.1 State Sequence Graph

A *State Sequence Graph*, G_{SS}, can be defined as a 2-tuple $\langle V, E\rangle$ where V represents the set of vertices and E represents a set of directed edges such that

1. Each vertex $v_i \in$ V is an n-tuple $(\bar{G}_1, \bar{G}_2, \ldots, \bar{G}_n)$ that represents the state of each of the n model elements that appear in the SR model.
2. Each directed edge $e_{ij} \in$ E is directed from vertex v_i to vertex v_j such that $v_i \to v_j$ satisfies the *default sequence* for any one of the n model elements represented in every vertex. This implies that $v_i \to v_j$ represents either of the following:
   (a) Some goal G_i goes from the *NC* state to the *CNF* state, denoted by $(\bar{G}_1 \ldots G_{i(N)} \ldots \bar{G}_n) \to (\bar{G}_1 \ldots G_{i(C)} \ldots \bar{G}_n)$, or
   (b) Some goal G_i goes from the *CNF* state to the *F* state, denoted by $(\bar{G}_1 \ldots G_{i(C)} \ldots \bar{G}_n) \to (\bar{G}_1 \ldots G_{i(F)} \ldots \bar{G}_n)$.
3. The number of vertices in the vertex set V is 3^n, i.e., $|V| = 3^n$.
4. Each path from the source vertex $(G_{1(N)} \ldots G_{n(N)})$ to the sink vertex $(G_{1(F)} \ldots G_{n(F)})$ represents a valid ordering of state transitions that satisfies the *default sequence* of the individual model elements, i.e., every unique path $(G_{1(N)} \ldots G_{n(N)}) \to \ldots \to (G_{1(F)} \ldots G_{n(F)})$ represents a finite state model.

The next level of complexity involves three different model elements. The analysis remains the same irrespective of how these three model elements are distributed between actors. Let G_1, G_2 and G_3 be the three goals modelled in the SR-diagram. As mentioned above, since each goal can be in either of three states, this particular situation will result in a state space with $3^3(=27)$ combined states. The *State Sequence Graph* can be obtained as shown before. A detailed reachability analysis yields 90 different paths that exist between the source vertex $(G_{1(N)} G_{2(N)} G_{3(N)})$ and the sink vertex $(G_{1(F)} G_{2(F)} G_{3(F)})$. Each of these paths represents a sequence of valid state transitions such that none of the three goals G_1, G_2, and G_3 violate the *default sequence*. Thus, with three model elements in the SR-diagram we get 90 possible finite state sequences that correspond to the given i* model.

4.1.1.2 Counting Multidimensional Lattice Paths

In general, it is interesting to observe how the number of paths within a *State Sequence Graph* increases in accordance to the number of model elements (k) within an i* model. It is intuitive from the above case studies that the growth of thestate space size can be represented as an exponential function $f(k) = 3^k$. This is because each model element can exist in either of three states. On the other hand, the function representing the growth of the finite state model space is far more complex. Before going into the details of evaluating an upper bound for the finite state model space, we need to keep in mind that every model element is initially in the *Not Created* state and it needs two transitions to reach the *Fulfilled* state. Thus, the distance covered by each model element is always 2.

Consider the case where $k = 2$. Since each model element needs to cover a distance of 2, we can consider $P_{1(N)}P_{2(N)}$ and $P_{1(F)}P_{2(F)}$ as the *Least Upper Bound* and the *Greatest Lower Bound* of a 2×2 lattice. In general, the number of paths within a $n_1 \times n_2$ lattice is given by

$$L_P = \binom{n_1 + n_2}{n_1} = \frac{(n_1 + n_2)!}{n_1!n_2!} \tag{4.1}$$

So for a 2×2 lattice structure, we have

$$L_P = \binom{2+2}{2} = \frac{(2+2)!}{2!\,2!} = \frac{4!}{2!\,2!} = \frac{24}{4} = 6.$$

This is exactly what we obtain from our example of Fig. 4.4.

When $k = 3$, we can represent the set of all possible state sequences from $P_{1(N)}P_{2(N)}P_{3(N)}$ to $P_{1(F)}P_{2(F)}P_{3(F)}$ as a 3-dimensional cubic lattice with each dimension having distance 2. In general, the number of paths in a 3-dimensional cubic lattice with dimensions (n_1, n_2, n_3) is given by

$$L_P = \binom{n_1 + n_2 + n_3}{n_1, n_2, n_3} = \frac{(n_1 + n_2 + n_3)!}{n_1!n_2!n_3!} \tag{4.2}$$

So for a 3-dimensional cubic lattice with dimensions (2, 2, 2), we have-

$$L_P = \binom{2+2+2}{2, 2, 2} = \frac{(2+2+2)!}{2!\,2!\,2!} = \frac{6!}{2!\,2!\,2!} = \frac{720}{8} = 90.$$

Again, this is exactly what we obtain from our previous case study.

To generalize an upper bound on the growth function of the finite state model space, we need to realize that for k different model elements in the i* model we need a k-dimensional hypercube lattice. The number of paths in such a k-dimensional hypercube lattice with dimensions $(n_1, n_2, \ldots, n_k)$ is given by

$$L_P = \binom{n_1 + n_2 + \cdots + n_k}{n_1, n_2, \ldots, n_k} = \frac{(n_1 + n_2 + \cdots + n_k)!}{n_1!n_2!\ldots n_k!} = \frac{(\sum_{i=1}^{k} n_i)!}{\prod_{i=1}^{k}(n_i!)} \tag{4.3}$$

Irrespective of the number of model elements in the i* model, since each model element travels a distance of 2 to become fulfilled, we have the condition $\forall_{i=1}^{k}$, $n_i = 2$. The total number of paths is, thus, given by

$$L_P = \frac{(\sum_{i=1}^{k} 2)!}{\prod_{i=1}^{k}(2!)} = \frac{(2 * \sum_{i=1}^{k} 1)!}{\prod_{i=1}^{k}(2)} = \frac{(2k)!}{2^k}. \tag{4.4}$$

4.1.1.3 The Naïve Algorithm

Input: SR-diagram of the i* model of an enterprise
Output: The set of plausible finite state sequences that can be derived from the given i* model
Data Structure: A *List* that stores all the model elements appearing in the SR model

Step-1: Select actor A_j and populate *List* with all the model elements that appear within the actor boundary of A_j.
Step-2: Repeat Step-1 for all actors and proceed to create the *State Sequence Graph*.
Step-3: Initialize the vertex set V with the vertex $(P_{1(N)} \ldots P_{n(N)})$ representing all model elements in the `Not Created` state.
Step-4: Select a vertex v_i from the vertex set V.
Step-5: Create a new vertex v_i' at a distance of 1 from v_i such that

(a) Some model element P_k makes a transition from `Not Created` to `Created Not Fulfilled` state, i.e., $(\bar{P}_1 \ldots P_{k(N)} \ldots \bar{P}_n) \rightarrow (\bar{P}_1 \ldots P_{k(C)} \ldots \bar{P}_n)$, OR
(b) Some model element P_k makes a transition from `Created Not Fulfilled` to `Fulfilled` state, i.e., $(\bar{P}_1 \ldots P_{k(C)} \ldots \bar{P}_n) \rightarrow (\bar{P}_1 \ldots P_{k(F)} \ldots \bar{P}_n)$.

Step-6: If $v_i' \notin V$, then $V = V \cup v_i'$.
Step-7: Repeat Steps 4–6 till the vertex set V is not filled, i.e., while $|V| < 3^n$.
Step-8: Select any two vertices v_i,v_j from the vertex set V that are separated by a distance of 1.
Step-9: Set up a directed edge from v_i to v_j if and only if $v_i \rightarrow v_j$ satisfies the *default sequence* for any one of the n model elements.
Step-10: Repeat Steps 8–9 till we obtain the n-dimensional hypercube lattice structure (*State Sequence Graph*) for the SR model.
Step-11: Each path from the vertex $(P_{1(N)} \ldots P_{n(N)})$ to the vertex $(P_{1(F)} \ldots P_{n(F)})$ represents a finite state sequence that corresponds to the given SR model.
Step-12: Stop.

4.1.1.4 Simulation Results: The Hyperexponential Explosion

Equation 4.4 of Sect. 4.1.1.2 can be used to generate a data set and observe how the state space and the finite state model space grows with increasing number of model elements in the i* model. Table 4.1 represents such a data set with the number of model elements increasing from 5 to 85 in steps of 5. Data thus obtained have been plotted on a graph and the trends are observed. Figure 4.5 depicts the rate of growth for both the state space and the finite state model space with respect to the number of model elements appearing in the given i* model.

Interpretation of the graph is quite interesting. Both the growth curves plotted in Fig. 4.5 appear to be somewhat linear in nature, although they are not straight lines.

Table 4.1 Rate of growth of space w.r.t. the number of model elements

No. of process elements	State space	Finite state model space
5	243	113400
10	59049	2.37588E+15
15	14348907	8.09487E+27
20	3486784401	7.78117E+41
25	8.47289E+11	9.06411E+56
30	2.05891E+14	7.74952E+72
35	5.00315E+16	3.48622E+89
40	1.21577E+19	6.5092E+106
45	2.95431E+21	4.2227E+124
50	7.17898E+23	8.289E+142
55	1.74449E+26	4.4083E+161
60	4.23912E+28	5.8022E+180
65	1.03011E+31	1.7528E+200
70	2.50316E+33	1.1403E+220
75	6.08267E+35	1.5123E+240
80	1.47809E+38	3.8999E+260
85	3.59175E+40	1.876E+281

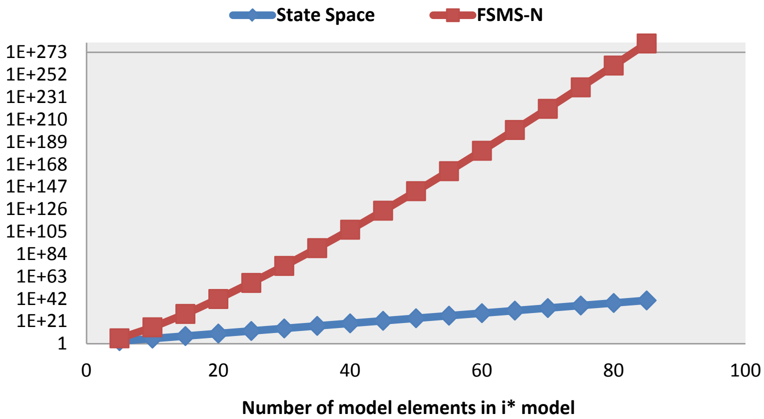


Fig. 4.5 Graph depicting the rate of growth of the state space and finite state model space with respect to the number of model elements in the i* model for the Naïve Algorithm

A careful analysis of the graph reveals that the vertical axis is a *logarithmic scale* where the values represent exponentially increasing integers. These values range from 1 to $1.876E + 281$. Thus, although the curves appear to be somewhat linear, they represent exponential growth functions on the logarithmic scale. In fact, the state space growth function, as represented by the blue curve, actually represents the growth function $f(k) = 3^k$. The growth function of the finite state model space, as represented by Eq. 4.4, is shown by the red curve.

The most significant inference that can be drawn from the graph is that the gradient of the blue curve is much less compared to that of the red curve. The gradient of a linear curve on a logarithmic scale signifies the rate of growth of the exponential function. This implies that although both the state space and the finite state model space grow exponentially, the rate of growth of the finite state model space is significantly large compared to that of the state space. In fact, the values in Table 4.1 reveal that, in every step, the state space grows by an approximate factor in the range $(10^2, 10^3)$, whereas the finite state model space grows by an approximate factor in the range $(10^{19}, 10^{20})$. This is really huge in terms of the rate of growth.

This extremely rapid growth in size of the finite state model space, caused by the *Naïve Algorithm*, results in a *hyperexponential explosion*. The growth curve of the finite state model space is so steep that it reaches infinitely large values for quite small number of model elements in the i* model. This implies that the finite state model space becomes quite unmanageable in real time when we are looking at the i* model of an entire enterprise. Thus, it becomes necessary to tackle this explosion in the finite state model space. One of the means to control this undesirable explosion is to extract partial sequence information that remains embedded within an i* model and perform some pruning activities while the finite state models are being generated. The *Semantic Implosion Algorithm* is proposed in the next section with this same intent. The proposed solution provides a significant improvement in terms of the rate of growth of the finite state model space.

4.1.2 The Semantic Implosion Algorithm (SIA)

The motive here is to prevent the *hyperexponential explosion* of the finite state model space that is caused by the *Naïve Algorithm*. Although the *Naïve Algorithm* generates all possible finite execution traces that can be derived from an i* model, some sort of pruning can be done on this model space. The simplest means of doing this is to feed the derived FSM into some standard model checker like *NuSMV* and check the model against user-defined temporal compliance rules, specified using CTL or LTL. However, since this needs to be done on the entire finite state model space, the time complexity of the entire process becomes unmanageable even when machine-automated.

It is desirable to prevent the *hyperexponential explosion* from occurring in the first place. We propose the *Semantic Implosion Algorithm*, or *SIA*, that tries to achieve this and proves to be successful to a good extent. *SIA* is based on the underlying hypothesis

that although an i* model is sequence agnostic, there exists some embedded temporal information that can be extracted and exploited to reduce the plausible space of finite state models. Temporal compliance rules may be further defined to reduce the size of the finite state model space.

Every model element P_i residing within the SR-diagram of an actor is uniquely identified using a system variable V_i. Every system variable V_i can have either of three values—0, 1, or 2—representing the `Not Created` ($P_{i(N)}$), `Created Not Fulfilled` ($P_{i(C)}$) and `Fulfilled` ($P_{i(F)}$) states, respectively. Every time a new model element P_j is encountered, a corresponding system variable V_j is created and initialized to 0 representing the Not Created state. This is reflected in the finite state model of the enterprise with a transition from the current state to a new state where the corresponding system variable V_j becomes a member of the state variable list.

The algorithm proceeds to explore the children of a chosen model element P_i. Before doing so, the corresponding system variable V_i is changed from 0 to 1 and pushed onto a *stack*. This is reflected in the finite state model with a state transition from the current state to a new state that reflects the fact that P_i has been created but not fulfilled. A model element is said to be `Fulfilled` when either it has no children (we have reached the actor boundary) or all its child model elements have been individually fulfilled. When this happens, the system variable V_i, corresponding to the model element P_i, is popped from the *stack* and updated with the value 2. A corresponding state transition is incorporated in the finite state model that reflects the fact that model element P_i has been fulfilled. Figure 4.6 illustrates the finite state model corresponding to a single model element and how the corresponding system variable is incorporated and updated along each transition.

However, it is interesting to note how the child model elements of a particular parent are processed. The processing differs for *task decompositions* and *means-end decompositions*. A *task decomposition* is an AND-decomposition and demands that all the child model elements be fulfilled in order to declare that the parent has also been fulfilled. A *means-end decomposition*, on the other hand, is an OR-decomposition and provides alternate strategies to fulfil the parent model element. Let us elaborate on the consequences of these two decompositions.

A *task decomposition* requires that all the child model elements be fulfilled before changing the state of the parent model element to the fulfilled state. However, since an i* model is sequence agnostic, the child model elements may be fulfilled in any random order. System variables associated with the child model elements should not defy the *default sequence* defined in Sect. 4.1.1. Let a model element P_j be decom-

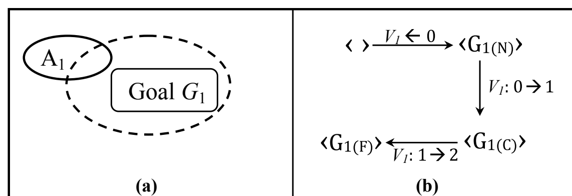


Fig. 4.6 **a** Actor A_1 with goal G_1; **b** The corresponding finite state model

posed by a *task decomposition* to a set of model elements $\langle P_1, P_2, \ldots, P_m \rangle$. The system variables associated with these model elements are V_1, V_2, …, V_m, respectively. We define a state transition from the current state with $V_j = 1$ to a new state with the state variables $V_j = 1, \forall_{r=1}^{m}, V_r = 0$. There exists several execution sequences of the decomposed model elements that finally results in a state with the state variables, $V_j = 1, \forall_{r=1}^{m}, V_r = 2$. The set of all possible execution sequences can be defined using a lattice structure, similar to the one shown in Fig. 4.4. Since all child model elements are fulfilled, we define another state transition in the finite state model that reflects the fact that the parent model element is also fulfilled, i.e., the new state has state variables V_j=2. The finite state model corresponding to such a *task decomposition* is shown in Fig. 4.7.

The interpretation of the figure is quite interesting. The lattice structure represents the set of all possible execution sequences that result in the successful fulfilment of the task decomposition. As seen in Sect. 4.1.1, the number of paths in a lattice structure for two model elements is 6. All of these 6 paths represent valid execution sequences or state transitions. Each path gives rise to a different finite execution sequence. This

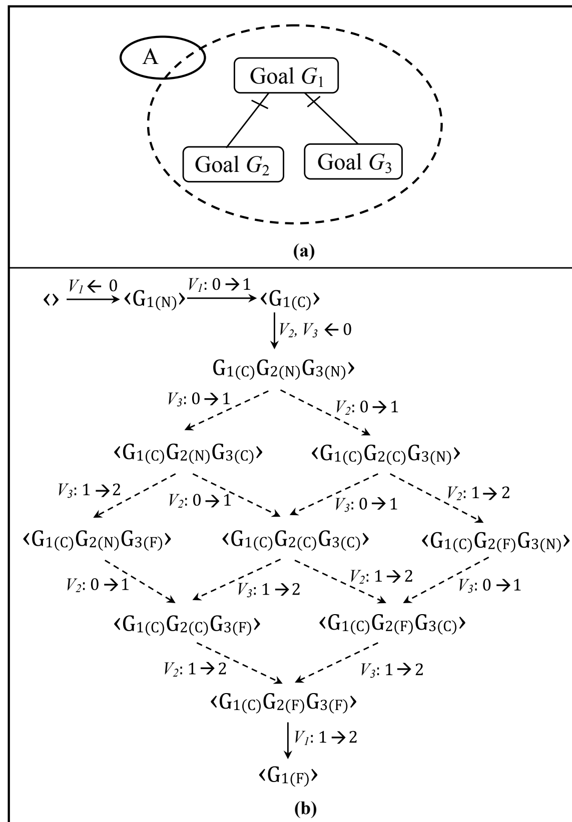


Fig. 4.7 **a** Actor A_1 with goals G_1, G_2 and G_3 connected through a task decomposition; **b** The corresponding set of all possible finite state models captured in a state sequence graph

implies that the task decomposition shown in Fig. 4.7 gives rise to 6 possible finite state sequences. The *Naïve Algorithm*, on the other hand, would generate a lattice structure with three model elements and the number of possible finite state sequences would become 90. This is a significant reduction in the finite state model space. In fact, the significant observation here is that a lattice structure will be generated only where task decompositions take place. In other words, only task decompositions will increase the size of the finite state model space.

A *means-end* decomposition is easier to handle. OR-decompositions, in general, do not increase the size of the finite state model space. Rather, if a particular model element P_j decomposes via a *means-end decomposition* into k model elements $\langle P_1, P_2, \ldots, P_k \rangle$, then we introduce k different transitions from the current state ($V_j = 1$) to k unique new states, each representing one of the k alternate means ($V_j = 1$, $V_p = 0$, $\forall_{p=1}^{k}$). An OR-decomposition is characterized by the fact that fulfilling any one of the alternate means implies fulfilling the parent model element. Thus, each of these k new states will make two transitions (labelled by V_p:$0 \rightarrow 1$ and V_p:$1 \rightarrow 2$) to reach their respective fulfilment states. Each alternate means will have a separate fulfilment state labelled by $V_j = 1$, $V_p = 2$, $\forall_{p=1}^{k}$. All the k fulfilment states will converge to a final state that represents the fulfilment of the parent model element P_j and is labelled by $V_j = 2$. The structure obtained is similar to the longitudinal lines on the globe of the earth. Figure 4.8 illustrates this further.

4.1.2.1 Some Interesting Features

1. Decompositions can be *nested*. This implies that decompositions can occur within other decompositions. One particular decomposition link may be further blown up with a second decomposition. For instance, *means-end decompositions* may be followed by a *task decomposition* along one means-end link and a *means-end decomposition* along some other means-end link. Figure 4.9 illustrates this scenario. This nesting of decompositions does not require any modifications on the algorithm. The corresponding finite state model is built accordingly where the state subsequences of the lower level decomposition is mereologically connected to the finite state model of the higher level decomposition.
2. It is interesting to note what happens if we reach a model element G_3, located at the actor boundary of actor A_1, that is dependent on some model element G_4 that is located at the actor boundary of actor A_2 (refer Fig. 4.10a).
   In this situation, we first proceed to complete the finite state models of the individual actors. We assume that the dependency between model elements G_3 and G_4 will be satisfied and pop out the system variable V_3 from the *stack* to set its value to 2. At the same time, we introduce a *temporary transition* in the corresponding finite state model that changes the state of G_3 from Created Not Fulfilled (*CNF*) to Fulfilled (*F*). This is shown in Fig. 4.10b. Such an assumption is necessary to proceed with the construction of the finite state model of individual actors.
   We need to maintain a list of all such dependencies. A *Global List* is maintained that stores 2-tuples of the form $\langle dependervariable, dependeevariable \rangle$.

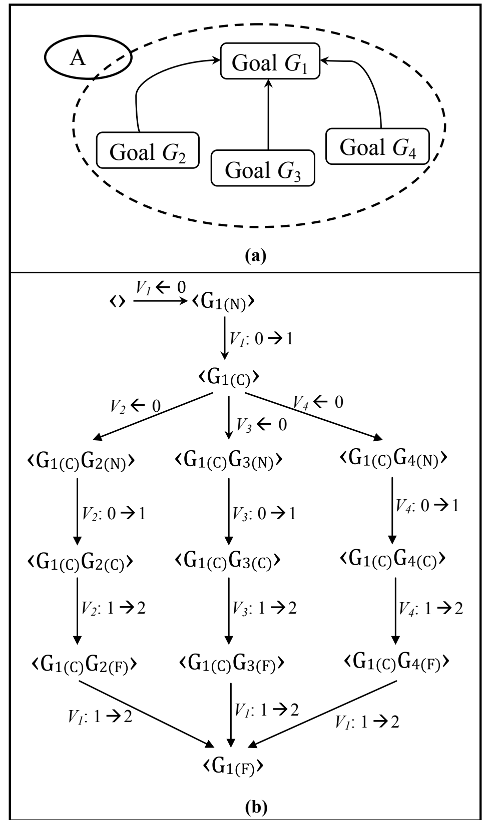


Fig. 4.8 **a** Actor A_1 with goals G_1, G_2, G_3 and G_4 connected through a means-end decomposition; **b** The corresponding finite state model

Once the finite state models of the individual actors have been built, the elements of the *Global List* are accessed. The above dependency has an entry of the form $\langle V_{13}, V_{24}\rangle$ and is interpreted as model element G_3 within actor A_1 depending on actor A_2 for model element G_4. The *temporary transition* in the finite state model of actor A_1 representing the change V_3: $1 \rightarrow 2$ is replaced by two new transitions that connect the finite state models of actors A_1 (FSM_1) and A_2 (FSM_2). The first transition is established from the state in FSM_1 having label $V_3 = 1$ to the state in FSM_2 having label $V_4 = 2$. The second transition is placed from the state in FSM_2 having label $V_4 = 2$ to the state in FSM_1 having label $V_3 = 2$. $\langle V_{13}, V_{24}\rangle$ is removed from the *Global List*. Figure 4.10c illustrates this process.

3. Dependency resolution causes state transitions to be set up between states belonging to the finite state models of the *depender* and the *dependee*. If the *depender* and *dependee* have M and N possible finite state sequences in their models, respectively, then we get $M \times N$ combination of sequences for interlinking the finite state models of the *depender* and *dependee*. The dependency resolution is reflected in all the $M \times N$ combinations.

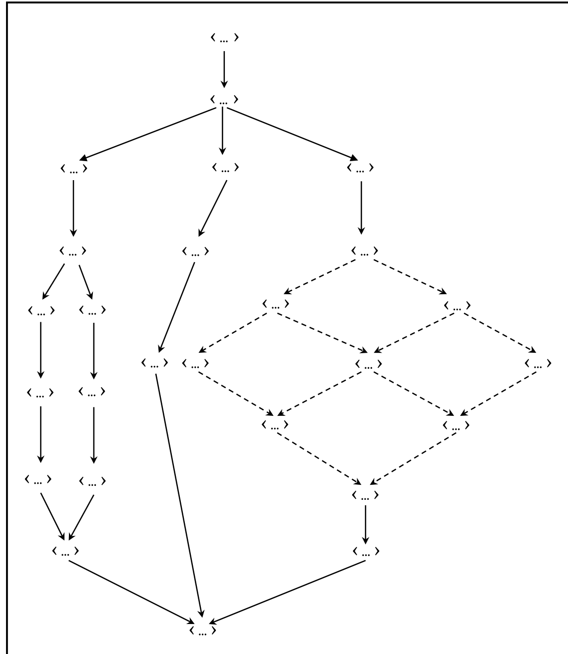


Fig. 4.9 The state sequence graph corresponding to a nested decomposition. A higher level means-end decomposition contains another means-end decomposition along the leftmost link and a task decomposition along the rightmost link

Let n be the total number of model elements occurring in the SR-diagram of the enterprise. The terminating condition of the *Semantic Implosion Algorithm* is given by the constraint, $\forall_{j=1}^{n}, V_j = 2$, the *stack* is empty and the *Global Dependency List* is empty. The algorithm initiates with the root model elements at the actor boundaries. State transitions are defined in the corresponding finite state model as and when model elements are discovered, explored and fulfilled. Let us look into the *Semantic Implosion Algorithm* now.

4.1.2.2 The Semantic Implosion Algorithm

Input: SR-diagram of the i* model of an enterprise.
Output: The finite state model that can be derived from the given i* model containing the set of plausible finite state sequences.
Data Structure: A *Local Stack* for each actor that stores model elements of the actor and a *Global List* to keep track of dependencies between actors.

Step-1: For every model element P_i that is not at the end of a *task decomposition* or *means-end* link, assign a system variable $V_i = 0$. Perform a *Depth-First Scan* of the SR-diagram of each actor starting at these boundary model elements.

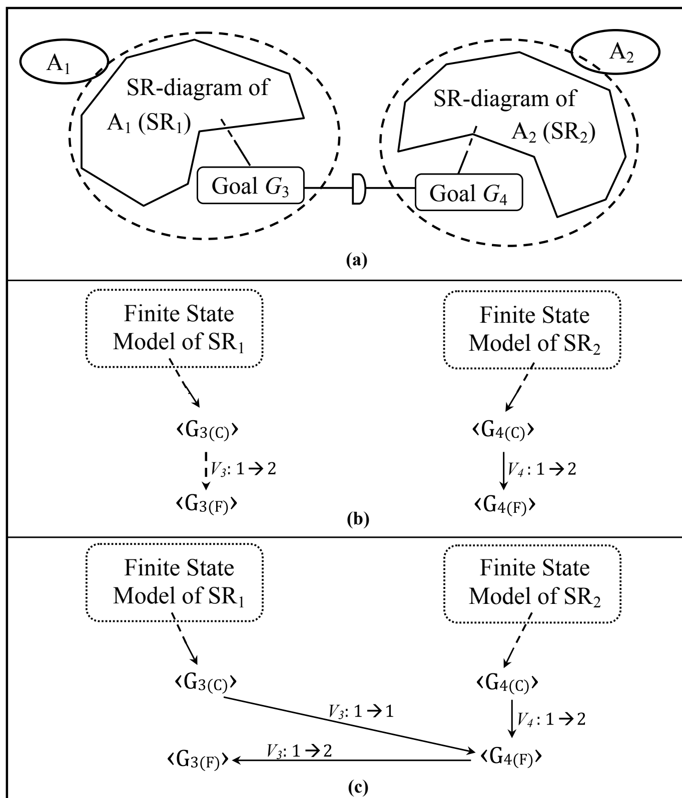


Fig. 4.10 **a** Goal G_3 of actor A_1 dependant on Goal G_4 of actor A_2; **b** Temporary transition from $G_{3(C)}$ to $G_{3(F)}$ introduced; **c** Resolution of the dependency by replacing the temporary transition with two permanent transitions

Step-2: For any model element P_j with $V_j = 0$, set $V_j = 1$ and push it onto the *Local Stack*. Reflect this transition in the finite state model by plotting a transition from the `Not Created` state to the `Created Not Fulfilled` state. Label this transition V_j:0→1.

Step-3: Discover all model elements $\langle P_1, P_2, \ldots, P_q \rangle$ that stem from the element P_j and are connected to P_j with *task decomposition* or *means-end* links. For each such element P_k, initialize a system variable V_k such that $\forall_{k=1}^{q} V_k = 0$.

(a) If P_j is at an actor boundary with no elements stemming from it and with no dependencies to other actors, pop V_j from the *Stack* and set $V_j = 2$. Set up a corresponding transition in the finite state model from the `Created Not Fulfilled` state to the `Fulfilled` state. Label this transition V_j:1→2.

(b) If P_j is dependent on some other actor for fulfilment, then pop V_j and insert it into the *Global List* with value $V_j = 2$. Insert a *tem-*

porary transition between states `Created Not Fulfilled` and `Fulfilled` for element P_j. No need to label this transition as it is a *temporary transition*.

(c) If P_j undergoes a *task decomposition* then we obtain several different finite state sequences for the *task decomposition* by permuting the order of execution of the child model elements. Each such permutation can be considered to be a valid execution trace and can be attached to the overall finite state model to obtain the unique finite state model for that actor.

(d) If P_j undergoes a *means-end decomposition* then we obtain multiple transitions from the current node in the same finite state model. Each transition represents an alternate strategy and is triggered by the corresponding guard condition. All the alternate state transitions emanating from the parent model element must converge at a state that represents that the parent model element has been fulfilled.

Step-4: Repeat *Steps 2–3* for all siblings of P_j in all the finite state models generated for actor A_i.
Step-5: Repeat *Steps 1–4* until the *Local Stack* is empty. This leaves us with the set of plausible finite state models of an actor A_i.
Step-6: Repeat *Steps 1–5* to extract all the possible finite state models of all the actors in the i* model.
Step-7: Remove elements of the form $\langle V_{ik}, V_{jl} \rangle$ from the *Global List*.
Step-8: Remove the *temporary transitions* corresponding to V_{ik} from the finite state model of actor A_i.
Step-9: Insert transitions from the P_k-`Created Not Fulfilled` state in the finite state model of actor A_i to the P_l-`Fulfilled` state in the finite state model of actor A_j. Label these transitions V_k:1→1.
Step-10: Insert another set of transitions from the P_l-`Fulfilled` state to the P_k-`Fulfil- ed` state between the finite state models of actors A_i and A_j. Label these transitions V_k:1→2.
Step-11: Repeat *Steps 7–10* until the *Global List* is empty and all the dependencies have been resolved.
Step-12: Stop.

4.1.3 Soundness and Completeness

Both the *Naïve Algorithm* and the *Semantic Implosion Algorithm* are *complete* because given a goal model both the algorithms are capable of generating a finite state model which include all the possible state transitions for valid execution traces. However, it is not wise to say that the *Naïve Algorithm* is *sound* because the generated finite state model also contains invalid state transitions. The *Semantic Implosion Algorithm*, on the hand, is *sound* because it contains all the valid transitions within the finite state model.

4.2 Complexity Analysis

Let us perform some analytics on comparing and contrasting the heuristics of the *Naïve Algorithm* and the *Semantic Implosion Algorithm*. The two metrics that are used for this analysis are the *State Space* (SS) and the *Finite State Model Space* (FSMS). However, since both algorithms share the concept of every model element going through 3 states, the SS metric will be the same for both algorithms and is defined by the function $f(k) = 3^k$, where k represents the number of model elements in the given SR-model. The FSMS metric is far more crucial in contrasting the heuristics that underline the two algorithms.

Figure 4.5 of Sect. 4.1.1.4 clearly illustrates the *hyperexponential explosion* caused by the *Naïve Algorithm* in the finite state model space. This is mainly due to the fact that the *Naïve Algorithm* considers all possible orderings of the model elements while ensuring the *default sequence* of each individual model element. A careful understanding of the *Semantic Implosion Algorithm* reveals that, while the finite state models of individual actors are being built, the finite state model space increases only when the following conditions hold:

1. Whenever a *nested Task Decomposition* is encountered. Suppose a goal/task is decomposed to k different model elements. Since an i* model is sequence agnostic, these k model elements can be executed in any order. The set of all possible execution traces is given by a k-dimensional hypercube lattice with each dimension having distance 2. As discussed in Sect. 4.1.1.2, the finite state model space increases by a factor of $\frac{(2k)!}{2^k}$ as given by Eq. 4.4. This implies that if the finite state model space already has p execution traces, a Task Decomposition into q model elements causes the size of the finite state model space to become $p.\frac{(2q)!}{2^q}$. In general, if the SR-diagram of an actor within the i* model has D_T task decompositions, and the number of possible alternate execution sequences generated by each of these task decompositions be given by $\#Seq_1, \#Seq_2, \ldots, \#Seq_{D_T}$, then the finite state model space size is given by the following relation:

$$S = \prod_{i=1}^{D_T} \#Seq_i \tag{4.5}$$

2. Whenever a dependency is being resolved. *Dependency resolution* results in merging the finite state model space of the *depender* and the *dependee*. If the finite state model spaces of actors A_i and A_j contain M and N finite state sequences, respectively, and there exists at least one dependency between these actors, then irrespective of the number of dependencies between A_i and A_j, the size of the finite state model space changes from $M + N$ to $M \times N$. Again, if actor A_j requires dependency resolution with actor A_k, and actor A_k has L finite state models, then the combined finite state model space has size $L \times M \times N$.
3. Let there be n actors participating in an i* model. Let the size of the finite state model spaces of the individual actors be given by $S_1, S_2, \ldots, S_n$, respectively.

Assuming that all the actors are interconnected with dependencies, the finite state model space (*FSMS*) for the entire enterprise is given by the following equation:

$$FSMS = \prod_{i=1}^{n} S_i \tag{4.6}$$

Both *Dependency Resolution* and *nested Task Decomposition* conditions are represented using the cartesian product relation. So, performance analysis of the two heuristics boils down to two basic steps. The first step involves observing the growth of the finite state model space for each individual actor. The second step is to observe the growth of the finite state model space for the entire enterprise.

4.2.1 Actor Internal Analytics

It is very difficult to predict the distribution of model elements within the SR-diagrams of individual actors. Since this is the first step of behaviour analysis, we are concerned with the growth of the finite state model space for individual actors within an i* model. In order to generate a consistent data set, we assume a uniform distribution of model elements. We increase the number of model elements occurring within the SR-diagram of an actor in the i* model, in steps of 5. Without loss of *uniformity*, we assume that for every 5 model element within an actor, there exists a task decomposition of 4 elements. This assumption is necessary as we want to estimate an upper bound on the growth function and the number of finite state sequences grows significantly with Task Decompositions as opposed to Means-End Decompositions.

We know that the *Naïve Algorithm* causes the finite state model space to grow according to Eq. 4.4, i.e., FSMS-N = $\frac{(2k_1)!}{2^{k_1}}$, where k_1 is the number of model elements in the i* model. The *Semantic Implosion Algorithm* grows only on the basis of task decompositions. The number of possible execution sequences generated by a 4-element task decomposition is obtained by substituting $k = 4$ in Eq. 4.4, i.e., $\frac{(2\times4)!}{2^4} = \frac{8!}{16} = 2520$. Since every 4-element task decomposition increases the finite state model space size by a factor of 2520, applying the cartesian product relation, we obtain the growth function of the *Semantic Implosion Algorithm* to be given by FSMS-S = 2520^{k_2}, where k_2 is the number of 4-element task decompositions occurring within the SR-diagram of an actor. Table 4.2 reflects such a data set.

The performance ratio parameter in Table 4.2 represents the reduction in finite state model space, obtained by the *Semantic Implosion Algorithm*, with respect to the *Naïve Algorithm*. The smaller the ratio the greater is the reduction in finite state model space achieved by the *Semantic Implosion Algorithm*. As the values in this column reflect, the reduction rate is not constant and increases from $\Theta(10^{-9})$ to $\Theta(10^{-17})$. This is also evident from the graph plotted for this data.

The graph plotted on the basis of this data is shown in Fig. 4.11. It is interesting to analyse the graph. The vertical axis is again a logarithmic scale of integers. The almost

Table 4.2 Actor internal analytics

No. of process elements (k_1)	No. of task decompositions (k_2)	Naïve algorithm	SI algorithm	Performance ratio
		FSMS-N = $\frac{(2k_1)!}{2^{k_1}}$	FSMS-S = 2520^{k_2}	$(\frac{FSMS\text{-}S}{FSMS\text{-}N})$
5	1	113400	2520	0.0222
10	2	2.37588E+15	6350400	2.67286E−9
15	3	8.09487E+27	1.6E+10	1.97656E−18
20	4	7.78117E+41	4.03E+13	5.17917E−29
25	5	9.06411E+56	1.02E+17	1.12532E−40
30	6	7.74952E+72	2.56E+20	3.30343E−53
35	7	3.48622E+89	6.45E+23	1.85041E−66
40	8	6.5092E+106	1.63E+27	2.50415E−80
45	9	4.2227E+124	4.1E+30	9.70943E−95
50	10	8.289E+142	1.03E+34	1.24261E−109
55	11	4.4083E+161	2.6E+37	5.89796E−125
60	12	5.8022E+180	6.56E+40	1.1306E−140
65	13	1.7528E+200	1.65E+44	9.41351E−157
70	14	1.1403E+220	4.16E+47	3.64816E−173
75	15	1.5123E+240	1.05E+51	6.94306E−190
80	16	3.8999E+260	2.64E+54	6.7694E−207
85	17	1.876E+281	6.67E+57	3.55544E−224

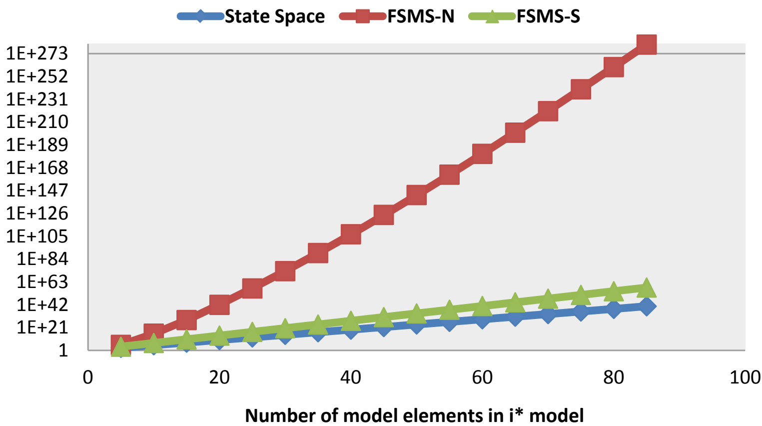


Fig. 4.11 Behaviour analysis with respect to the finite state model space of individual actors for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the number of model elements in the i* model varies

linear curves plotted on this scale represent exponential functions. The gradient of these approximately linear curves represent the rate of growth of the corresponding exponential function. The following observations can be concluded from the graph:

1. The *blue curve* depicts the growth of the state space and is consistent for both scenarios, given by 3^k. As both algorithms have the same underlying basis that every model element goes through three states, the state space growth remains the same for both the algorithms.
2. The *green curve* represent the behaviour of the *Semantic Implosion Algorithm*. The two lines with triangle and diamond annotations are very close to each other and have almost similar gradients. This implies that the rate of growth of the finite state model space, as observed from the *Semantic Implosion Algorithm*, is almost similar to the rate of growth of the state space.
3. The *red curve* depicts the finite state model space growth of the *Naïve Algorithm*. The slope of this line is much greater than those of the green and blue lines. This represents the *hyperexponential explosion* that is a characteristic of the *Naïve Algorithm*.
4. A closer look at the FSMS values in Table 4.2 reveals the fact that the FSMS metric increases by a factor in the range of $(10^{19}, 10^{20})$, for the *Naïve Algorithm*, whereas, for the *Semantic Implosion Algorithm*, the FSMS metric increases by a factor of 10^3.

From the above data set—Table 4.2 and Fig. 4.11—it is evident that the *Semantic Implosion Algorithm* provides a huge improvement with respect to the rate of growth of the finite state model space for individual actors in comparison to the *Naïve Algorithm*. This is the significant contribution of the heuristic proposed in the *Semantic Implosion Algorithm*.

4.2.2 Inter-Actor Analytics

These analytics provide an insight into how *Actor Internal Analytics* scales up and impacts the growth rate of the finite state model space with respect to the entire i* model representing an enterprise. There are two events that impact *Inter-Actor Analytics* as follows:

1. *Density of Actors* participating in the i* model, and
2. *Distribution of Model Elements* within the SR-diagram of the actors.

Let us individually analyse how these two parameters effect the growth rate of the finite state model space.

4.2.2.1 Variation of Actor Density

In order to simulate a data set, we assume a uniform density of five model elements within individual actors and evaluate the rate of growth of the finite state model space. Similar to the data in Table 4.1, we assume that every actor has a 4-element task decomposition. The *Naïve Algorithm* does not take the semantics of the model elements into consideration and, thus, the finite state model space size can be evaluated by replacing $k = 5$ in Eq. 4.4. The finite state model space size of every actor is obtained as

$$\forall i, S_i = \frac{(2 \times 5)!}{2^5} = \frac{10!}{32} = 113400.$$

Replacing this value of S_i in Eq. 4.6, we get the finite state model space for the entire enterprise (FSMS-N) as

$$FSMS\text{-}N = (113400)^n \tag{4.7}$$

The *Semantic Implosion Algorithm*, on the other hand, causes the finite state model space of individual actors to grow only when task decompositions are encountered. Since we assume a 4-element task decomposition to exist in each actor, the finite state model space(S_i) of all the actors remains constant and is given by replacing $k = 4$ in Eq. 4.4. Thus,

$$\forall i, S_i = \frac{(2 \times 4)!}{2^4} = \frac{8!}{16} = 2520.$$

Since uniform distribution of model elements has been assumed, replacing this value of S_i in Eq. 4.6 gives the finite state model space for the entire enterprise (FSMS-S) as generated by the *Semantic Implosion Algorithm*. Thus,

$$FSMS\text{-}S = (2520)^n \tag{4.8}$$

In order to generate a simulated data set, we restrict the number of model elements in each actor to 5 and increase the density of actors (n) within the i* model of the enterprise from 5 to 55 in steps of 5. The data set is obtained by replacing these values of n in Eqs. 4.7 and 4.8. Table 4.3 represents such a data set. The performance ratio column represents the relative decrease in the finite state model space that is obtained by the *Semantic Implosion Algorithm*. Figure 4.12 shows the corresponding graph structure that is obtained by plotting this data.

Interpretation of the graph is quite intuitive. The *blue curve* represents the growth function of the *Naïve Algorithm*. In this data set, it represents the exponential function $(113400)^n$. The *red curve* plots the growth function of the *Semantic Implosion Algorithm* and represents the exponential $(2520)^n$. With the vertical axis representing a logarithmic scale of integers, the two functions are mapped as nearly linear curves with different gradients. The gradient of the blue curve is greater than the gradient

Table 4.3 Inter-actor analytics obtained by varying actor density

No. of actors (n)	Naïve algorithm	SI algorithm	Performance ratio
	FSMS-N = $(113400)^n$	FSMS-S = $(2520)^n$	$(\frac{FSMS\text{-}S}{FSMS\text{-}N})$
5	1.87528E+25	1.01626E+17	5.41924E−9
10	3.51666E+50	1.03277E+34	2.93679E−17
15	6.59471E+75	1.04956E+51	1.59152E−25
20	1.23669E+101	1.06662E+68	8.62479E−34
25	2.31914E+126	1.08396E+85	4.67397E−42
30	4.34902E+151	1.10158E+102	2.53294E−50
35	8.15562E+176	1.11949E+119	1.37266E−58
40	1.52940E+202	1.13768E+136	7.43872E−67
45	2.86805E+227	1.15618E+153	4.03124E−75
50	5.37840E+252	1.17497E+170	2.18461E−83
55	1.00860E+278	1.19407E+187	1.18388E−91

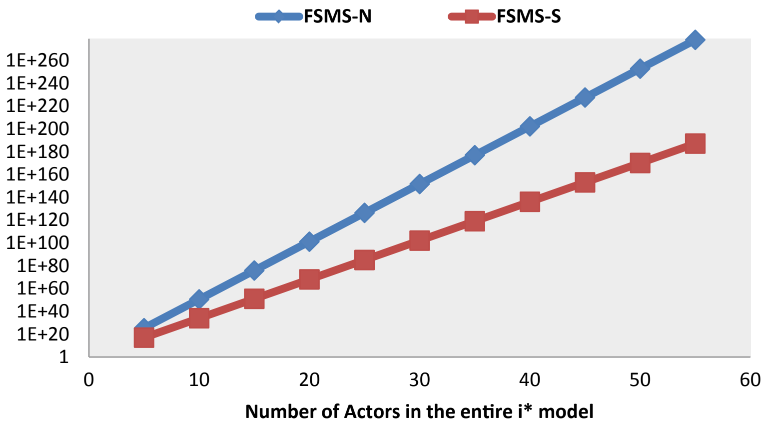


Fig. 4.12 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the density of actors in the i* model varies

of the red curve. This implies that the *Naïve Algorithm* increases the size of the finite state model space more rapidly as compared to the *Semantic Implosion Algorithm*. This is evident from the growth functions Eqs. 4.7 and 4.8 itself. However, this is an overly simplified data set with uniform distribution and semantics.

4.2.2.2 Variation of the Distribution of Model Elements

In this particular simulation, we fix the number of actors involved in the enterprise model to five. Keeping the number of actors fixed, the distribution of model elements peractor is increased from 5 to 25 in steps of 5. Assuming uniform distribution across all the actors in the i* model, every actor generates it's finite state model space with the same size. The space size changes with varying model element distribution. Let the size of the finite state model spaces of the individual actors be given by S_1, $S_2, \ldots, S_5$, respectively, for some model element distribution k.

The *Naïve Algorithm* combines Eqs. 4.4 and 4.6 to give a function representing the growth of the finite state model space as follows:

$$FSMS\text{-}N = \left(\frac{(2k_1)!}{2^{k_1}}\right)^5, \forall k_1, k_1 \in \{5, 10, 15, 20, 25\}. \tag{4.9}$$

The *Semantic Implosion Algorithm* expands the finite state model space for Task Decompositions only. Our underlying assumption that there exists a 4-element Task Decomposition for every group of 5 elements dictates the growth function of the finite state model space as follows:

$$FSMS\text{-}S = \left(\frac{(2k_2)!}{2^{k_2}}\right)^5, k_2 = k_1 \div 5, \forall k_1, k_1 \in \{5, 10, 15, 20, 25\}. \tag{4.10}$$

The data generated from Eqs. (4.9) and 4.10 is shown in Table 4.4. The number of actors has been fixed to be 5. The performance ratio values represent the improvement in finite state model space that is achieved by the *Semantic Implosion Algorithm*. The smaller the value, the greater is the gain in performance achieved by the SIA heuristics. The rapid rate of increase in performance reflects the benefits of using the improved heuristics of the *Semantic Implosion Algorithm*. Figure 4.13 represents the graph corresponding to this data.

Table 4.4 Inter-actor analytics obtained by varying the distribution of goals

No. of process elements (k_1)	Naïve algorithm	SI algorithm	Performance ratio
	FSMS-N $= (\frac{(2k_1)!}{2^{k_1}})^5$	FSMS-S $= (\frac{(2k_2)!}{2^{k_2}})^5$, $k_2 = k_1 \div 5$	$(\frac{FSMS\text{-}S}{FSMS\text{-}N})$
5	1.87528E+25	1.01626E+17	5.41924E−9
10	7.57046E+76	1.03277E+34	1.36421E−43
15	3.47576E+139	1.04956E+51	3.01966E−89
20	2.85249E+209	1.06663E+68	3.73929E−142
25	6.11823E+284	1.08399E+85	1.77174E−200

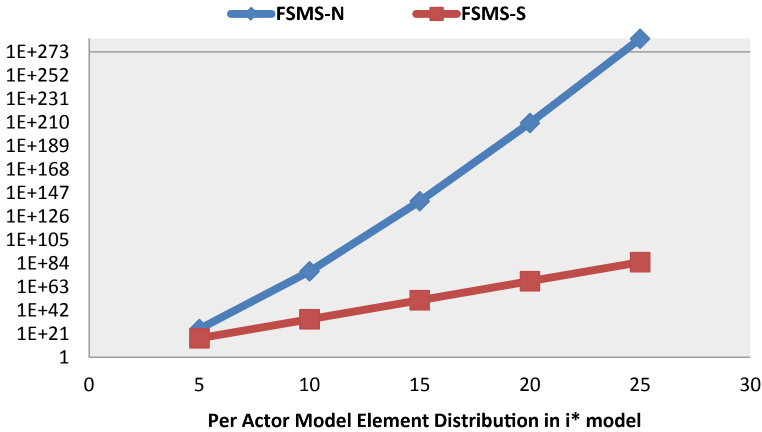


Fig. 4.13 Behaviour analysis with respect to the finite state model space of the entire enterprise for the *Naïve Algorithm* (FSMS-N) and the *Semantic Implosion Algorithm* (FSMS-S) as the distribution of model elements within actors in the i* model varies

The interpretation of the graph is quite similar to the previous graphs. The vertical axis represents a logarithmic scale of integers. Both the exponential functions, given by Eqs. 4.9 and 4.10, appear as straight lines. However, the gradients of the two lines are widely different. This implies that the rate of growth of FSMS-N (represented by the *blue curve*) is much greater than that of FSMS-S (represented by the *red curve*).

4.2.3 SIA Analytics

The analytics provided in Tables 4.2, 4.3, and 4.4, and the corresponding graphs shown in Figs. 4.11, 4.12, and 4.13, all point in the same direction. The obvious conclusion from these data sets is that the *Semantic Implosion Algorithm* provides a huge improvement over the more simple *Naïve Algorithm*. This improvement is in the context of the finite state model space and clearly establishes the superiority of the SI-heuristics in comparison to the Naïve-heuristics.

The above conclusion triggers an urge to take an insight into the behaviour of the *Semantic Implosion Algorithm* when both the parameters—*Actor Density* and *Model Element Distribution*—are varied simultaneously. Table 4.5 presents such a data set. The data is generated by varying the distribution of model elements in individual actors from 5 to 25 peractor, in steps of 5. The finite state model space size is obtained using the following equation:

$$FSMS\text{-}A = \left(\frac{(2k_2)!}{2^{k_2}}\right)^A, k_2 = k_1 \div 5 \tag{4.11}$$

Table 4.5 Inter-actor analytics obtained by varying both actor density and distribution of goals for the *Semantic Implosion Algorithm*

No. of process elements (k_1)	SI algorithm		
	FSMS-5	FSMS-10	FSMS-15
5	1.01626E+17	1.03277E+34	1.04956E+81
10	1.03277E+34	1.06662E+68	1.10158E+102
15	1.04956E+51	1.10157E+102	1.15617E+153
20	1.06663E+68	1.13769E+136	1.21349E+204
25	1.08399E+85	1.17503E+170	1.38069E+255

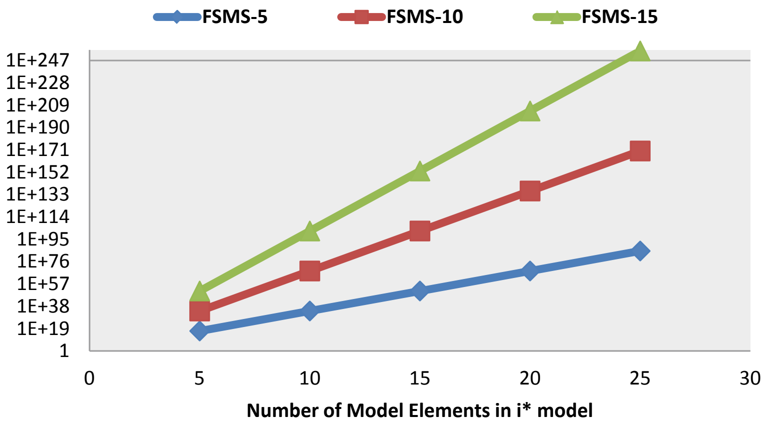


Fig. 4.14 Behaviour analysis of the *Semantic Implosion Algorithm* (w.r.t. the finite state model space) as the distribution of model elements within actors and the actor density in the i* model are both varied

Here, A represents the number of actors in the i* model of the enterprise. k_2 is obtained from k_1 based on the assumption that we have a 4-element task decomposition for every group of 5 model elements. Maintaining the uniformity of model element distribution across all the actors, we obtain the data set for 5, 10 and 15 actors, represented by FSMS-5, FSMS-10 and FSMS-15, respectively. The graph obtained from the data set in Table 4.5 is shown in Fig. 4.14.

The graph is fairly simple to analyse and interpret. The vertical axis is again a logarithmic scale. Each of the individual curves (*green, red,* and *blue*) appears to be linear but represent exponential growth functions. The fact that the finite state model space size will increase with greater number of actors has already been observed in Fig. 4.12. Hence, as the number of actors increase, the curves are positioned higher. It can also be concluded from Fig. 4.13 that for a fixed actor density, the finite state model space size increases with increasing density of model elements. Hence, the positive gradient in each of the three approximately linear curves.

The more important observation here is that the nearly linear curves are not parallel to each other. The gradient of the lines increase with increasing actor density, i.e., the green curve is steeper than the red curve which, in turn, is steeper than the blue plot. The gradient of these approximately linear curves represent the rate of growth of the exponential functions that capture the growth of the respective finite state model spaces. This means that as the actor density increases, the finite state model space increases even more rapidly.

4.3 The i*ToNuSMV Tool

Unlike dataflow and workflow models, goal models do not capture sequences of activities within the system or enterprise being designed. This makes it difficult for analysts to check the correctness of these models in the requirements phase itself. Since goal models have their own motivation, quite distinct from those of process models or workflow models, researchers have come up with completely different analysis techniques that provide new insights into the system or enterprise being developed.

Horkoff and Yu [6] have documented an exhaustive survey of the existing goal model analysis techniques and how requirement analysts can select from these alternatives based on different criteria and attributes. Applying model checking techniques to goal models (like i*) has been considered by researchers from the community. The main research problem here is that model checkers accept extended finite state models as input. Finite state models capture some sort of sequential information that represents the possible state transitions that a system can go through. Since goal models are sequence agnostic they cannot be adapted and fed into model checkers directly. We aim to provide a significant contribution in this direction by proposing the *i*ToNuSMV* tool.

The *i**ToNuSMV tool addresses this issue and performs model transformation of the given i* model. Figure 4.15 illustrates the architecture of the proposed solution. The *FSM Building* module generates a finite state model corresponding to the given i* model and the *NUSMV Mapper* module maps the generated finite state model to the NuSMV input language. The output of the *i*ToNuSMV* tool can be fed directly into the NuSMV model verifier and can be checked against temporal properties, behavioural characteristics or compliance rules written using LTL, or CTL.

*4.3.1 i*ToNuSMV Input*

The *i*ToNuSMV* prototype does not provide a graphical interface for drawing i* models. Rather, it takes a textual representation of the i*-SR-diagram as input. We use the tGRL notation as our input language. This may help in the integration of tool with the jUCMNav framework. For the i* model shown in Fig. 4.16, the corresponding tGRL representation is as follows:

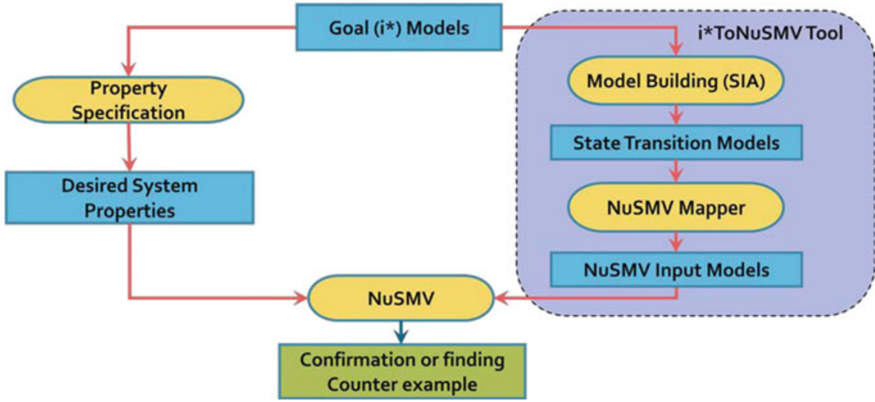


Fig. 4.15 The *i*ToNuSMV* tool

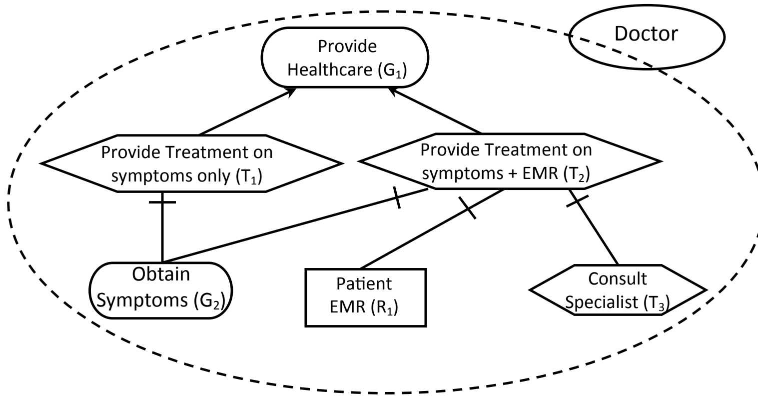


Fig. 4.16 An i* model of a single actor *Doctor*

grl *test_model*
{ **actor** *Doctor*{
goal *ProvideHealthcare*{**decompositionType=or**;}
task *Symptoms_Treat*{**decompositionType=and**;}
task *Symptoms_EMR_Treat*{**decompositionType=and**;}
goal *ObtainSymptoms*{}
resource *PatientEMR*{}
task *ConsultSpecialist*{}
Symptoms_Treat **decomposedBy** *ObtainSymptoms*;
Symptoms_EMR_Treat **decomposedBy** *ObtainSymptoms,PatientEMR, ConsultSpecialist*;
ProvideHealthcare **decomposedBy** *Symptoms_Treat,Symptoms_EMR_Treat*;
}
}

4.3.2 The Preprocessing Module

We perform a simplified lexical analysis of this textual input by tokenizing the text (using *filtokn.exe*) and then identifying keywords, operators and user-defined model artefacts (using *recognit.exe*). We proceed to identify the tree structure of the model artefacts (using *modlroot.exe*) and begin our model transformation process from the root of this tree structure.

4.3.3 The Model Transformation Module

After obtaining the desired tree structure, we proceed to generate the finite state model corresponding to the given SR-diagram. We use the *Semantic Implosion Algorithm* (SIA) [7] for converting the given i* model to a finite state model. The algorithm, as proposed by the authors, proposes a methodology for exploiting the semantics of SR-diagrams and creating a finite state model with minimum number of state transitions.

SIA uses the notion of each model artefact going through three states—*Not_Created* (NC), *Created_Not_Fulfilled* (CNF), and *Fulfilled* (F)—as proposed by Fuxman in [5]. SIA controls an explosion of the state transition space by mapping k-element means-end decompositions to k-conditional branch structures and k-element task decompositions to k-dimensional hypercube lattices. A detailed illustration of the algorithm and the significant improvement achieved w.r.t. the state transitional space complexity, has been documented in the original article [7]. This model transformation is achieved in the *i*ToNuSM V* tool by executing the *extract.exe* binary.

4.3.4 The Mapper Module

The mapper module takes the extended finite state model produced by the *Semantic Implosion Algorithm* and maps it to the input language of the NuSMV model verifier. We assign identifiers with all goals, tasks and resources that appear in the given SR-diagram. Each such identifier can have three possible values—`NC, CNF, FU`—corresponding to the three states mentioned in the previous section. All identifiers are initialized to the `NC` value which marks the initial state of our finite state model. The final state of the state model is denoted by any state where the root node has the value `FU`. The state transitions of the finite state model are captured using `next()` value assignments in the NuSMV input language. The *mapper.exe* binary does this mapping and generates an NuSMV input model as the final output.

4.3.5 *i*ToNuSMV Output*

The particular example shown in Fig. 4.16 generates a finite state model *STT.opm* and the corresponding NuSMV input model *NUSMV_input.smv*. *Var.opm* is a text file that contains the list of state variables that have been assigned to all the goals, tasks and resources. For the above example, *Var.opm* gets populated as shown below.

Also, according to the *Semantic Implosion Algorithm*, every state variable is initialized to zero, which represents the *Not Created* state of the entity. As the finite state model is built, every state variable makes two transitions. A 0→1 transition implies that the entity represented by that state variable goes from the *Not Created* state to the *Created Not Fulfilled* state. A 1→2 transition, on the other hand, implies that the represented entity goes from the *Created Not Fulfilled* state to the *Fulfilled* state. A sample finite state model for the healthcare example is shown in Table 4.7. The state variables used in the finite state model are in accordance with Table 4.6.

The same set of state variables are used to build the NuSMV input models. The NuSMV model corresponding to the finite state model shown in Table 4.7 is as follows:

```
MODULE main
VAR
V101 : NC, CNF, FU;
V102 : NC, CNF, FU; ....
```

All state variables are declared with enumerations NC (*Not Created*), CNF (*Created Not Fulfilled*), and FU (*Fulfilled*). Once all variables are declared, all the state variables are initialized to NC.

```
ASSIGN
init(V101) := NC;
init(V102) := NC; ....
```

After the declaration and initialization of state variables, we proceed to define the transition of state variables as captured in the finite state model of Table 4.7. For instance, rows 1, 6 and 15 of the finite state model represent transitions for state variable `V101`. We read these three lines of the finite state model and create the next state value for the *NuSMV* model as follows:

Table 4.6 State variable listing of entities

Variable name	Entity
V101	ProvideHealthcare
V102	Symptoms_Treat
V103	Symptoms_EMR_Treat
V105	ObtainSymptoms
V108	PatientEMR
V109	ConsultSpecialist

Table 4.7 Finite state model recorded in *STT.opm*

Present state	Event	Next state
V101 = 0	V101:0→1	V101 = 1,V102 = 0,V103 = 0
V101 = 1,V102 = 0,V103 = 0	V102:0→1	V101 = 1,V102 = 1,V103 = 0,V105 = 0
V101 = 1,V102 = 1,V103 = 0,V105 = 0	V105:0→1	V101 = 1,V102 = 1,V103 = 0,V105 = 1
V101 = 1,V102 = = 1,V103 = = 0, V105 = 1	V105:1→2	V101 = 1,V102 = 1,V103 = 0,V105 = 2
V101 = 1,V102 = 1,V103 = 0,V105 = 2	V102:1→2	V101 = 1,V102 = 2,V103 = 0
V101 = 1,V102 = 2,V103 = 0	V101:1→2	V101 = 2
V101 = 1,V102 = 0,V103 = 0	V103:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 0,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 0,V108 = 0,V109 = 0	V105:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 1,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 1,V108 = 0,V109 = 0	V105:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 0,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 0,V109 = 0	V108:0→1	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 1,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 1,V109 = 0	V108:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 0
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 0	V109:0→1	V101 = 1,V102 = 0,V103 = = 1,V105 = 2,V108 = 2,V109 = 1
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 1	V109:1→2	V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 2
V101 = 1,V102 = 0,V103 = 1,V105 = 2,V108 = 2,V109 = 2	V103:1→2	V101 = 1,V102 = 0,V103 = 2
V101 = 1,V102 = 0,V103 = 2	V101:1→2	V101 = 2

```
next(V101) :=
case
V101=CNF & V102=NC & V103=FU : FU;
V101=CNF & V102=FU & V103=NC : FU;
V101=NC : CNF;
TRUE: V101;
esac;
```

This is done for all the state variables appearing in Table 4.6. The complete NuSMV model corresponding to the finite state model of Table 4.7 is obtained in *NUSMV_input.smv*.

4.3.6 The i*ToNuSMV Algorithm

An algorithm for the entire process may be specified as follows:
Input: Textual representation of an i* model SR-diagram.
Output: An extended finite state model and the corresponding NuSMV input model.

Algorithm:

Step-1: Tokenize the input text file using *filtokn.exe* and separate all tokens.
Step-2: Identify all the tokens and distinguish the keywords, user-defined variables and operators, separately, using *recognit.exe*.
Step-3: Identify the root model element from which the Semantic Implosion Algorithm will begin execution by using *modlroot.exe*. Associate state variables/identifiers with each goal, task and resource appearing in the i* model.
Step-4: Run the Semantic Implosion Algorithm by executing the *extract.exe* binary. This code generates the extended finite state model that can be derived from the i* model.
Step-5: Finally this finite state model is mapped to an NuSMV input model with the *mapper.exe* executable.
Step-6: *cleanup.exe* is used to clear the working directory before loading and converting the next i* model.

4.3.7 Platforms Used

The front end of the tool has been developed in the Microsoft Visual Basic environment. The 64-bit binaries have been generated using the Eclipse and Pelles C platforms.

4.3.8 Application Scenario

Let us consider the remote healthcare example illustrated in Fig. 4.16. A remote healthcare enterprise may want to comply to a temporal constraint that a *Doctor* will provide long term treatment only after it receives a *Symptoms Message* from the patient through the *ObtainSymptoms* goal. This implies that in the task decomposition of *Symptoms_EMR_Treat*, *ObtainSymptoms* must be Fulfilled before resource *PatientEMR* is acquired and task *ConsultSpecialist* is performed. This can be captured as a system property specified in CTL. The state variables listed in Table 4.6 can be used to define this property as follows:

$$AG((V103 = CNF \wedge V105 = FU \wedge \neg(V108 = FU \vee V109 = FU)) \\ \rightarrow F(V105 = FU \wedge V108 = FU \wedge V109 = FU))$$

This property can be fed into the *NuSMV* model checker and verified against the *NUSMV_input.smv* input model generated by the *i*ToNuSMV* prototype. The NuSMV input model passes the model verification test if and only if all execution paths in the corresponding finite state model satisfy the condition that $V105$ is fulfilled before $V108$ and $V109$ are fulfilled. Otherwise, the CTL property is violated and *NuSMV* generates counterexamples.

4.4 i*ToNuSMV Version Manager

The tool has evolved through several versions as described below:

- **i*ToNuSMV Version 1.01**: Beta prototype that supported only 3-level goal models. Multi-actor scenarios were also not supported.
- **i*ToNuSMV Version 1.02**: This version accepts an i* model in non-standardized textual format and only converts it to the corresponding finite state machine and NuSMV input model. Model checking is not supported.
- **i*ToNuSMV Version 1.03**: The NuSMV model verifier is integrated into the tool. Users can now verify CTL specifications on the NuSMV model being generated.
- **i*ToNuSMV Version 1.04**: Bug fix. Previous version was path dependent. User was compelled to instal the tool in path C:Äù\istarToNuSMV. Path dependency removed.
- **i*ToNuSMV Version 2.01**: MAJOR UPGRADE. The input language of the tool has been changed from the previous non-standardized textual input to `tGRL` [8]. Supports multi-actor scenarios nut not inter-actor dependencies.
- **i*ToNuSMV Version 2.02:**: MAJOR UPGRADE. Inter-actor dependencies have been implemented.One finite state machine for the entire goal model is generated rather than peractor finite state machines as in the previous versions.

4.5 Contact and URL

The *i*ToNuSMV* tool can be freely downloaded from the following URL—http://cucse.org/faculty/tools/. The URL also contains a user manual of the *i*ToNuSMV* prototype in pdf format and can be downloaded from the link provided at the end of the page. For any further queries, please mail the authors at *novarun.db@gmail.com*.

4.6 Conclusion

Model checking tools, typically check a model against certain temporal properties. The need to bridge the gap between i* models and any other model with partial ordering is evident. Although model transformations have existed in the industry for quite some time, no work has been done to derive finite state models from i* models. This paper first illustrates and presents a *Naïve Algorithm* for extracting sequences from i* model constructs. Simulation results demonstrate how this causes a *hyperexponential explosion* in the finite state model space. The *Semantic Implosion Algorithm* provides an improvement to counter this explosion.

Detailed simulations have been done by applying both the algorithms to similar types of i* models and the results show that the *Semantic Implosion Algorithm* provides a significant improvement over the *Naïve Algorithm*. Typically, the finite state model space grows in the order of 10^{20} for the *Naïve Algorithm*, whereas, for the *Semantic Implosion Algorithm*, the growth rate is restricted to the order of 10^3. Although this may not be the best approach to extract a minimal set of plausible finite execution sequences, it definitely provides a significant improvement over the *Naïve Algorithm*.

The set of possible finite execution traces, that correspond to a given i* model, can be further pruned by feeding them into a model checking tool like NuSMV and checking them against certain enterprise-specific temporal properties or compliance rules. All models that generate counter-examples may be discarded. This is one of the biggest advantages of having a model that captures ordering of states. Also, once the set of valid finite state models have been obtained, we can map them to BPMN models, Petri Nets, or even UML models. This helps enterprise architects by allowing the automated generation of code snippets, thereby, reducing the efforts required to build the enterprise. Thus, once the requirements have been finalized and modelled by the architects, the development of the enterprise becomes fully automated. This ensures greater consistency and correctness and reduces the risks of failure.

The *i**ToNuSMV tool is a research prototype that takes a tGRL representation of an SR-model as input. The tool can also be extended to any goal modelling framework due to the generic nature of the model transforming Semantic Implosion Algorithm. A detailed working of the i*ToNuSMV tool with a multi-actor scenario having inter-actor dependencies is illustrated in the User Manual and Tutorial Video on the tool page.[1] Appropriate screenshots of the tool interface have also been provided.

[1] http://cucse.org/faculty/tools/.

References

1. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295. https://doi.org/10.1109/32.588521
2. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NUSMV: a new symbolic model verifier. In: Proceedings of the 11th international conference on computer aided verification (CAV), pp 495–499. http://dl.acm.org/citation.cfm?id=647768.733923
3. Lapouchnian A (2005) Goal-oriented requirements engineering: an overview of the current research, Depth Report. University of Toronto. Canada, Toronto
4. Fuxman A, Pistore M, Mylopoulos J, Traverso P (2001) Model checking early requirements specifications in tropos. In: Proceedings of the 5th international symposium on requirements engineering (RE), pp 174–181. https://doi.org/10.1109/ISRE.2001.948557
5. Fuxman AD (2001) Formal analysis of early requirements specifications, MS thesis. Department of Computer Science. University of Toronto, Canada
6. Horkoff J, Yu E (2011) Analyzing goal models—different approaches and how to choose among them. In: Proceedings of the 2011 ACM symposium on applied computing (SAC), pp 75–682. https://doi.org/10.1145/1982185.1982334
7. Deb N, Chaki N, Ghose A (2016) Extracting finite state models from i* models. J Syst Softw, SI: COMPSAC, Elsevier 121:265–280. https://doi.org/10.1016/j.jss.2016.03.038
8. Abdelzad V, Amyot D, Alwidian SA, Lethbridge T (978) A textual syntax with tool support for the goal-oriented requirement language. In: iStar, vol 978, pp 61–66. http://ceur-ws.org/Vol-1402/paper6.pdf

Chapter 5
Goal Model Maintenance

Goal models, and in particular, those specified in the form of AND-OR graphs, are often specified without regard for the context in which these goal models are used. In this chapter, we offer techniques for making goal models context-sensitive. We do this in two ways. First, we provide a means for representing and analysing the collateral effects of achieving a goal. A collateral effect is a collection of state changes (in the world in which we wish to pursue the achievement of the goals specified in the goal model) which are not desired but are nonetheless necessary to achieve the goal(s) of interest. If the goal is to achieve a condition `p` and the only two options available to an organization to achieve (these are thus OR-refined subgoals) is via the subgoal `p AND q` or the subgoal `p AND r`, then `q` and `r` are collateral effects (or collateral post-conditions). Second, we define a machinery to ensure that the collateral effects associated with goals in a goal model do not violate critical safety and liveness conditions for the domain. The machinery supports both the checking of such violations, and, in the event that violations are detected, the resolution of these violations (via a novel reconciliation operator).

The development and maintenance of goal models is a critical element that determines the success of an enterprise. There exists a vast literature on strategic management that argues this point. Goal model maintenance mechanisms must be in place as stakeholder motivations and domain assumptions keep evolving over time. Previously compliant goals can become infeasible or non-compliant with the system regulations over time. Goal models need to be updated for a variety of reasons:

1. Changes in the business context—changes in market conditions, the appearance of new competitors, etc.
2. Changes at the operational level—availability of machinery/manufacturing capability, availability of suppliers, availability of various links in a supply chain, etc.
3. Changes in an organization's strategic direction—typically decided at the board level of an organization.

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_5

At a representational level, changes to a goal model might be necessitated by:

- The appearance of new constraints, such one that requires that certain goals should be neither explicitly nor implicitly be accepted as a driver for organizational behaviour (this can happen due to changes in compliance requirements, for instance). Changes of this kind have been investigated by Hoesch-Klohe and Ghose [1]
- The appearance of domain constraint violations within the goal model.

This chapter takes a closer look at the latter kind of changes. Inconsistencies may appear because of a variety of reasons. There exist multiple perspectives of employees at different levels in an organization. Senior management might interpret a goal in a certain way, while middle managers might adopt a different interpretation of more refined goals that they are tasked to achieve, while operational staff might have even more divergent interpretations of the further refined subgoals that they operationalize. Changes manifest more explicitly at different levels in an organization. As the discussion above suggests, some changes manifest at the board level (changes in strategic direction), others at middle management level (tactical changes in market conditions, for instance) while others at the operational level. All of these entail that different levels of a goal model might be interpreted differently, leading to inconsistencies.

The resolution of these inconsistencies sits at the heart of the problem of strategic alignment in enterprise management. Manual alignment of goal models can prove to be costly and error-prone. Analysts may fail to identify inconsistencies or derive suboptimal solutions since they are limited by their power to visualize the entire space of goal modification alternatives that govern the strategic alignment process. Optimality can have multiple interpretations. In our setting, the interpretations of a goal are represented by its semantic annotation. We position our contribution within a new and general framework for assessing goal model proximity. The intent is that when a goal model needs to change, the degree of those changes must be minimized to the extent possible. This general setting is similar to that of Hoesch-Klohe and Ghose.

We offer a range of intuitions for assessing goal model proximity—like structural proximity, semantic proximity, hierarchy-sensitive proximity, etc. Based on this notion of proximity, we try to assist requirement analysts by providing a framework that allows her to explore the entire space of goal model configurations that satisfy a given change request and identify the configuration that is closest to the original goal model. The framework presented here does not prescribe any specific formal language as there exist multiple languages (with varying degrees of expressibility) that can be used to formalize goals. Our goal model maintenance framework tries to address the problems of non-entailment and inconsistency. Although there exist proposals for strategic alignment of goal models, researchers have not explored the benefits of leveraging the structure and semantics of goal models simultaneously. Going beyond the AND/OR graph structure of goal models and considering the goal model semantics simultaneously, helps analysts to better understand the intentions

being captured by the enterprise. The AFSR framework [J.2] proposed in this chapter improves on the existing work by:

- providing an innovative goal model semantic annotation formalization that has advantages during strategic alignment, but is non-prescriptive to represent existing goal models.
- providing a semantic reconciliation mechanism that allows efficient identification of entailment and consistency conflicts within goal model configurations.
- assisting analysts by providing conflict-free configurations that incorporate bare minimal changes.
- providing a framework that maps the goal model maintenance exercise to a state-space search problem having a heuristic path cost function that is admissible and consistent.
- illustrating how an admissible and consistent heuristic guarantees an optimal solution (that deviates minimally) by using A* search.

The AFSR framework consists of three different algorithms—the *S*emantic *R*econcilia-tion *A*lgorithm (`SRA`), the *E*ntailment *R*esolution *A*lgorithm (`ERA`), and the *C*onsistency *R*esolution *A*lgorithm (`CRA`). `SRA` reconciles the context-free *immediate* satisfaction conditions of goals into context-sensitive *cumulative* satisfaction conditions. The *cumulative* satisfaction conditions associated with any goal represents the semantics of satisfying the system requirements captured by the goal model *sub*tree rooted at that goal. Once the *cumulative* satisfaction conditions of a goal are derived, `SRA` compares them with it's *immediate* satisfaction conditions and flags *domain constraint violations*, if they are detected. `ERA` and `CRA` are then used by the AFSR framework to resolve these violations. Both these algorithms provide requirement analysts with possible "*conflict—free*" alternatives, obtained by refactoring the original goal model.

The rest of the chapter is organized as follows. Section 5.1 elaborates on how `SRA` works and raises flags on detecting domain constraint violations. Section 5.2 elaborates on the `ERA` and `CRA` algorithms and how they provide conflict-free alternatives to requirement analysts. The chapter also provides a roadmap to implement the AFSR framework in Sect. 5.3. We present a case study of how the framework can be extended to i* models in Sect. 5.4. Section 5.5 illustrates how we can realize this roadmap by building a prototype and performing some simulations. Simulation results are also discussed in this section. Section 5.6 concludes the chapter.

5.1 Semantic Reconciliation

The main objective of this chapter is to ensure consistency in the specification of goals and, thus, consistency in the shared understanding of organizational intent amongst stakeholders. We define an annotated goal model as one in which every goal has been annotated with their intended satisfaction conditions. These annotations can be represented as 2-pairs of the form ⟨*immediate-func, cumulative-func*⟩. The term

'*func*' in the above pair refers to functional semantic annotations as we intend to propose the framework with respect to functional requirements only and ignore the non-functional requirements for the time being. Thus, "*immediate-func*" refers to the satisfaction conditions for the functional requirements captured by that goal. Requirement analysts are required to provide the *immediate-func* annotations for each goal. *Cumulative-func* annotations of a goal represent the set of satisfaction conditions that are derived by accumulating the *immediate-func* semantics of the goal tree that is rooted at that goal.

Use Case: Healthcare

We take a real-life use case of a healthcare enterprise. We first demonstrate how a goal model can be annotated in real-world business settings. We also demonstrate how we can apply the proposed AFSR framework for identification of violations and their resolution in an evolving environment with changing business demands. Let us consider the healthcare example shown in Fig. 5.1 with goal, task and resource labels.

The goal model in Fig. 5.1 can be annotated as follows:

- $\mathtt{IE(G_1)} = \{Received_Patient, Provided_Relief\}$
- $\mathtt{IE(G_2)} = \{Emergency_Treatment_Provided\}$
- $\mathtt{IE(G_3)} = \{Normal_Treatment_Provided\}$
- $\mathtt{IE(G_4)} = \{\{Received_Text\}, \{Received_Voice\}\}$
- $\mathtt{IE(R_1)} = \{PreExisting_Disease_Searched, Allergies_Checked\}$
- $\mathtt{IE(T_1)} = \{\{\{Sample_Taken\}, \{Performed_Procedure\}\}, Test_Result_Known\}$

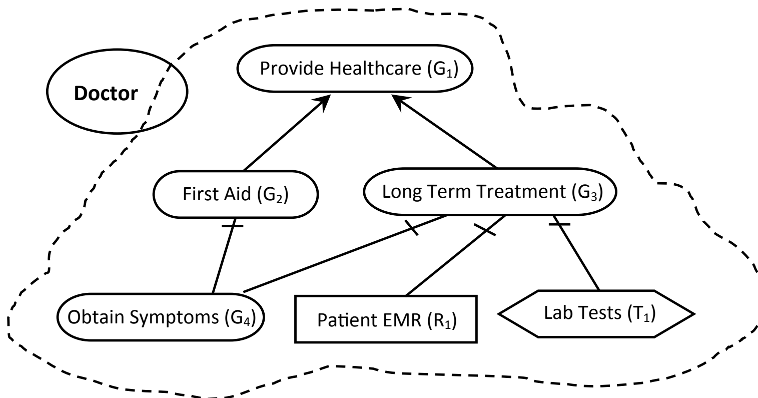


Fig. 5.1 Goal model of an existing healthcare enterprise

Annotation of individual goals, tasks and resources within the goal model with immediate satisfaction conditions is the only phase of the AFSR framework that requires human intervention. These annotations are context-free and the requirement analyst lists them for each goal model artefact in a stand-alone perspective. Thus, the immediate satisfaction conditions need to be associated with a knowledge base of semantic rules that correlate these immediate satisfaction condition. Thus, for the above set of immediate satisfaction conditions, we have the following correlation rules:

KB1: $Emergency_Treatment_Provided \rightarrow Received_Text \vee Received_Voice.$
KB2: $Normal_Treatment_Provided \rightarrow (Received_Text \vee Received_Voice) \wedge PreExisting_Disease_Searched \wedge Test_Result_Known.$
KB3: $Received_Patient \rightarrow Received_Text \vee Received_Voice$
KB4: $Provided_Relief \rightarrow Emergency_Treatment_Provided \vee Normal_Treatment - _Provided.$

Using the proposed AFSR framework, we want to compute the cumulative satisfaction conditions for each goal model artefact as follows:

- CE(G_4) = $\{\{Received_Text\}, \{Received_Voice\}\}$
- CE(R_1) = $\{PreExisting_Disease_Searched, Allergies_Checked\}$
- CE(T_1) = $\{\{\{Sample_Taken\}, \{Performed_Procedure\}\}, Test_Result_Known\}$
- CE(G_2) = $\{Emergency_Treatment_Provided, \{\{Received_Text\}, \{Received_Voice\}\}\}$
  Applying **KB1**, the CE set for G_2 reduces to:
  CE(G_2) = $\{\{Received_Text\}, \{Received_Voice\}\}$
- CE(G_3) = $\{Normal_Treatment_Provided, \{\{Received_Text\}, \{Received_Voice\}\}, PreExisting_Disease_Searched, Allergies_Checked, \{\{Sample_Taken\}, \{Performed_Proced - ure\}\}, Test_Result_Known\}$
  Applying **KB2**, the CE set for G_3 reduces to:
  CE(G_3) = $\{\{\{Received_Text\}, \{Received_Voice\}\}, PreExisting_Disease_Searched, Allergies_Checked, \{\{Sample_Taken\}, \{Performed_Procedure\}\}, Test_Result_Known\}$
- CE(G_1) = $\{\{Received_Patient, Provided_Relief, \{\{Received_Text\}, \{Received_Voice\}\}\}, \{Received_Patient, Provided_Relief, \{\{Received_Text\}, \{Received_Voice\}\}, PreExisting_Disease_Searched, Allergies_Checked, \{\{Sample_Taken\}, \{Performed_Procedure\}\}, Test_Result_Known\}\}$
  Applying rules **KB3** and **KB4**, the CE set of G_1 reduces to:
  CE(G_1) = $\{\{\{Received_Text\}, \{Received_Voice\}\}, \{\{\{Received_Text\}, \{Received_Voice\}\}, PreExisting_Disease_Searched, Allergies_Checked, \{\{Sample_Taken\}, \{Performed_Pr - ocedure\}\}, Test_Result_Known\}\}$

The notion of accumulating lower level semantics stems from the idea that subgoals lower in a goal tree provide more detailed accounts of ways in which higher level goals might be satisfied. Thus, there is value in propagating these semantics up the goal tree to obtain annotations (attached to each goal) that provide more detailed (and complete) formal accounts of alternative ways in which a goal is actually being satisfied. A formal annotation of the goals empowers requirement analysts to use automated reasoners for consistency and compliance checking, thereby releasing the analysts from laborious and complex manual analysis and evaluations. The semantic reconciliation machinery can be viewed as a black box that takes context-independent immediate satisfaction conditions as input and produces context-sensitive cumulative satisfaction conditions as output. The implications of the *cumulative* satisfaction conditions, for a given goal model, depend on the precision with which analysts specify the *immediate* satisfaction conditions [2, 3].

Given a goal model configuration, we identify mainly two different types of domain constraint violations that may exist within the model—*entailment* and *consistency*. The solution presented in this paper tries to answer the question—"*Given a goal model maintenance exercise, how can we remove all conflicts existing within a goal model and generate a conflict-free configuration (version) that deviates minimally from the original goal model?*". Given a goal model configuration, there exists a vast space of modified configurations that address this issue and generates conflict-free versions. It becomes quite infeasible for the analysts to enumerate the complete search space which, in turn, may result in analysts coming up with suboptimal solutions to the goal model maintenance problem.

Goal models have AND-decompositions as well as OR-decompositions. AND-decompositions capture the lower level subgoals that must be fulfilled in order to satisfy a higher level goal. OR-decompositions, on the other hand, capture alternatives for fulfilling a given goal. Each OR-decomposition link shows one possible means for fulfilling the parent goal. This gives rise to the notion of goal subgraphs that define unique solutions for fulfilling high-level goals. We call these subgraphs "*OR-refined goal models*". Let us first define what we mean by OR-refined goal models.

Definition *OR-refined Goal Models* (ORGMod*s*). An OR-refined goal model for a given high-level goal G is one with no OR-alternatives. It is obtained by committing to a specific OR-alternative wherever OR-alternatives exist in the goal model.

*ORGMod*s can be derived by performing a modified depth-first-search (DFS) of the goal model subtree rooted at G such that

i. Whenever we encounter an AND-decomposition, we include all the children in the decomposition, and
ii. Whenever we encounter an OR-decomposition, we commit to only one of the possible alternatives.

For instance, let us consider the high-level goal G_1 in Fig. 5.2. A possible *ORGMod* for achieving this goal is marked by a dashed polygon. The *ORGMod* so identified can be written using a top-down approach as $\langle G_1, G_3, \{G_6, \langle G_7, G_{10}, G_{12}\rangle\}\rangle$. This notation is quite easy to follow. We start with the root goal G_1 and

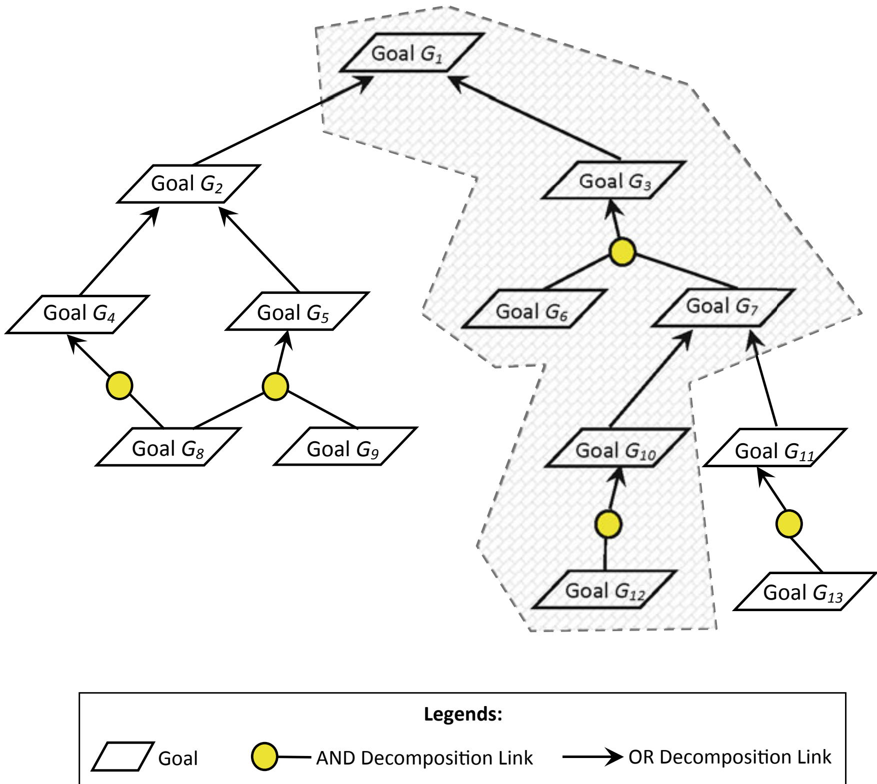


Fig. 5.2 An OR-refined Goal Model highlighted within the goal model

keep traversing the goal model until we reach leaf-level goals. Sequences of goals within angle brackets '$\langle\rangle$' represent successive levels in the goal model whereas '$\{\}$' are used to capture siblings within AND-decompositions. So $\langle G_7, G_{10}, G_{12}\rangle$ represents three successive levels of the *ORGMod* including goal G_7 whereas $G_3, \{G_6, G_7\}$ represents an *AND-decomposition* of goal G_3 into goal G_6 and goal G_7. Both angle brackets and braces can be nested within one another.

The semantic reconciliation machinery processes the immediate satisfaction conditions of individual goals and builds their corresponding cumulative satisfaction conditions. We deploy this machinery between adjacent levels of an *ORGMod*. The cumulative satisfaction condition of goal G_{12} is combined with the immediate satisfaction condition of goal G_{10} to obtain the cumulative satisfaction condition of goal G_{10}. The cumulative satisfaction condition of goal G_{10} is then combined with the immediate satisfaction condition of goal G_7 to obtain the cumulative satisfaction condition of goal G_7 and so on until we reach the root goal G_1.

5.1.1 *ORGMod* Extraction

As mentioned previously, it is cumbersome to derive the cumulative satisfaction condition for an entire goal model. Instead, the semantic reconciliation process can be restricted to one or more *ORGMod*s. This is practically more useful as requirement analysts may wish to see the cumulative satisfaction conditions of some desired goals. *ORGMod*s represent goal sub-models that are derived from the original goal model and play a decisive role in the cumulative semantic reconciliation process. *ORGMod*s help in pruning alternatives that can be excluded from the process.

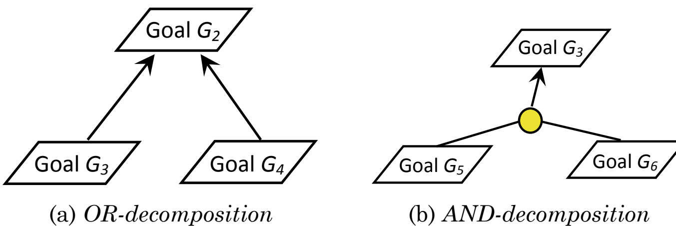
The pruning process also helps in assigning unique labels to *ORGMod*s. An *ORGMod* label identifies the order in which satisfaction conditions can be reconciled for a chosen high-level goal. Consider the *OR-decomposition* shown in Fig. 5.3a. It captures two different alternates to achieve the high-level goal G_2. This results in two different labels for the two different *ORGMod*s that can be obtained

Label 1: $\langle\langle G_2, G_3\rangle, [G_2, G_4]\rangle$
Label 2: $\langle\langle G_2, G_4\rangle, [G_2, G_3]\rangle$

Each label identifies an *ORGMod* for satisfying goal G_2 and an *exclusion set* (as defined in [2]). *Exclusion sets* are used to list those alternate paths that were not selected at an *OR-decomposition* for the given *ORGMod* label. So for the Label 1, goal G_2 is fulfilled by performing goal G_3 and, hence, $[G_2, G_4]$ is in the exclusion set. $\langle G_2, G_3\rangle$ specifies that semantic reconciliation must occur from G_3 to G_2. Similarly, we obtain Label 2 if we choose goal G_4 over goal G_3.

Consider the *AND-decomposition* shown in Fig. 5.3b. *AND-decompositions* denote independent objectives that can be executed in any order or even in parallel. Sibling goals are represented using { } in the *ORGMod* sequence. Figure 5.3b has the following label:

Label 3: $\langle\langle G_3, \{\langle G_5\rangle, \langle G_6\rangle\}\rangle, [\emptyset]\rangle$



(a) *OR-decomposition* (b) *AND-decomposition*

Fig. 5.3 Goal models illustrating the two different types of decompositions or splits that artefacts can undergo

$\{\langle G_5 \rangle, \langle G_6 \rangle\}$ represents independent goals contained within a set. This set is considered as a separate element in the outer sequence of goals. Also, the exclude set is null as *AND-decompositions* do not provide choices to requirement analysts.

We intend to automate the process of extracting all possible *ORGMod*s that have a particular goal as root. This goal may be randomly chosen by the requirement analysts for cumulative semantic reconciliation. There are three subprocesses involved with the *ORGMod* label extraction process—*path traversal*, *extracting decomposition sequences*, and *deriving ORGMod labels*. The following sections elaborate on these subprocesses.

5.1.1.1 Path Traversal

Any goal model can be viewed as a goal graph that already has an embedded tree structure. All the goals can be considered as generic nodes in a tree with all the decomposition links serving as edges. We ignore softgoals and contribution links for the time being. We perform a depth-first search on the goal graph. The path traversal process returns a list of paths, each of which is a sequence of goals from the root to the leaves and with no parallel edges. Applying the path traversal procedure on the goal model shown in Fig. 5.2 with goal G_1 as the chosen locus of semantic reconciliation, we get the following list of paths:

Path List:
Path 1: $\langle G_{1(X)}, G_{2(X)}, G_4, G_8 \rangle$
Path 2: $\langle G_{1(X)}, G_{2(X)}, G_{5(A)}, G_9 \rangle$
Path 3: $\langle G_{1(X)}, G_{2(X)}, G_{5(A)}, G_8 \rangle$
Path 4: $\langle G_{1(X)}, G_{3(A)}, G_6 \rangle$
Path 5: $\langle G_{1(X)}, G_{3(A)}, G_{7(X)}, G_{10}, G_{12} \rangle$
Path 6: $\langle G_{1(X)}, G_{3(A)}, G_{7(X)}, G_{11}, G_{13} \rangle$

The symbols *(X)* and *(A)* are used to mark goals that undergo *OR-decompositions* and *AND-decompositions*, respectively. *OR-decompositions* are analogous to exclusive gateways whereas *AND-decompositions* are analogous to parallel gateways. These paths are collected and segmented into groups based on decomposition sequences.

5.1.1.2 Extracting Decomposition Sequences

The path list derived in the previous section can be segmented into collections of goals as shown in Fig. 5.4. Each of these segments captures the decomposition of a goal into subgoals and are referred to as decomposition sequences. These sequences can be captured using a generic format that represents objects of the decomposition class. Figure 5.5 shows the decomposition sequence objects (DSO(s)). Each decomposition sequence begins with a goal that undergoes an *AND-decomposition* or an

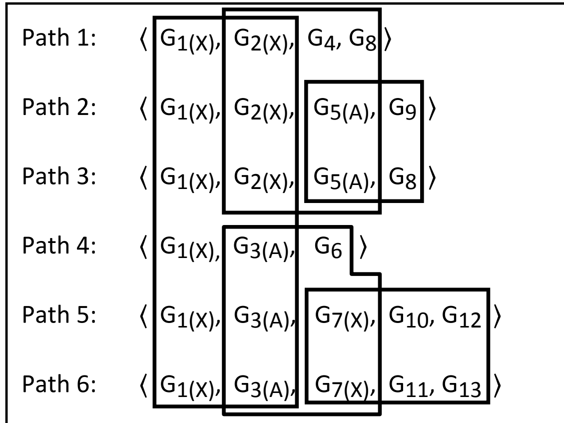


Fig. 5.4 Decomposition sequence segmentation of the path list for the goal model in Fig. 5.2

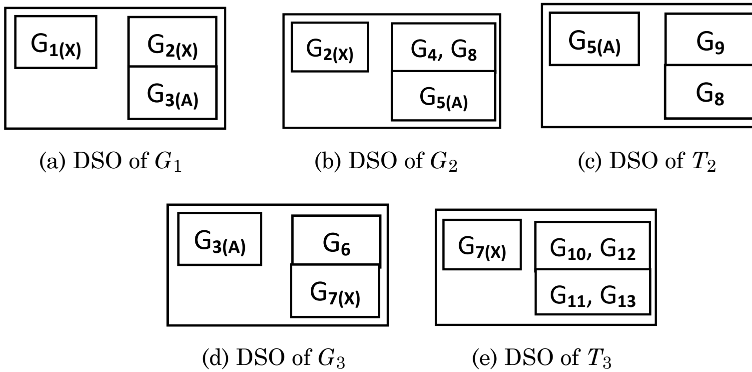


Fig. 5.5 Decomposition Sequence Objects (DSO) derived from the decomposition sequences in Fig. 5.4

OR decomposition and its lower level goals. Decomposition sequences either end at leaf-level goals or at the beginning of the next decomposition sequence.

$G_{1(X)}$ marks the beginning of the first decomposition sequence that ends in either $G_{2(X)}$ or $G_{3(A)}$. Both these decomposition goals mark the beginning of the next decomposition sequence. $G_{2(X)}$ can either end in the leaf-level goal G_8 or in the decomposition goal $G_{5(A)}$. Similarly, $G_{3(A)}$ either end in the goal G_6 or the AND-decomposition node $G_{7(X)}$. This process is repeated to obtain a decomposition sequence for $G_{5(A)}$ and $G_{7(X)}$, respectively.

Each box in Fig. 5.4 represents a decomposition sequence segment. Each segment represents a subsequence starting at some goal that either undergoes an *OR-decomposition* or an *AND-decomposition*. We can use these decomposition segments to derive Decomposition Sequence Objects (DSOs) as shown in Fig. 5.5. Each decom-

position sequence begins with a different goal and can be mapped to a unique DSO for that goal. For example, Fig. 5.5a shows the DSO for goal G_1. For each DSO, the label on the left represents the root goal and the list of labels on the right represents the decomposition subsequences beginning at that goal and ending at either a leaf-level goal or at another decomposition goal. The leftmost decomposition sequence for goal G_1 (leftmost segment in Fig. 5.4) has two subsequences—one ending in $G_{2(X)}$ and the other ending in $G_{3(A)}$. The DSO for $G_{2(X)}$ represents two subsequences—one ending in G_8 and the other in $G_{5(A)}$.

Since goal models have an inherent tree structure, the parent–child relationship existing between the goals is preserved in the decomposition sequence objects as well. While developing *ORGMod*s, this relationship plays a vital role. For example, sequences of the goal $G_{7(X)}$ (in Fig. 5.5e) must be resolved before creating the sequences of $G_{3(A)}$ (Fig. 5.5d). In general, decomposition sequences of the child goals must be incorporated within the parents. DSOs create a unique model that is independent of the original path list. This enables the creation of completely independent *ORGMod*s. Each DSO must have a method that accumulates the decomposition subsequences of the child goals and assembles them with the parent subsequence.

5.1.1.3 Deriving ORGMod Labels

In order to derive all possible *ORGMod*s that stem from a particular goal (as desired by requirement analysts), we need to provide the first decomposition sequence only. Every decomposition sequence passes its own sequence to its child decomposition sequence. The child decomposition sequence uses recursion to build all the possible subsequences and merges them with the parent sequence. The merging process is repeated backwards till we reach the first decomposition sequence. The final merge operation produces the set of all possible *ORGMod* labels that can be derived for the given goal from the goal model.

For the goal model of Fig. 5.2, we begin with the decomposition sequence object of goal G_1. G_1 passes its subsequence to its child decomposition sequences G_2 and G_3. G_2 and G_3 build their own sequences separately and merge them with the parent sequence of G_1. The following sequence of operations illustrates the *ORGMod* label derivation process. The symbol $\rightarrow$ is used to indicate the passing of sequences from a parent sequence to a child sequence. The symbol $\leftarrow$ is used to denote the passing of all merged sequences from the child to the parent. The beginning subsequence before the first decomposition in our example is $\emptyset$ which calls the first decomposition sequence.

$$\boxed{\emptyset \rightarrow G_{1(X)}}$$

In order to generate the *ORGMod* labels, we look at the DSO of $G_{1(X)}$ (Fig. 5.5a). $G_{1(X)}$ must process its child decomposition sequences before returning the final list of *ORGMod* labels.

$\langle G_{1(X)}, \langle G_{2(X)} \rangle \rangle$

$\langle G_{1(X)}, \langle G_{3(A)} \rangle \rangle$

G_1 passes its subsequence (∅) to $G_{2(X)}$ and $G_{3(A)}$.

$\emptyset \rightarrow G_{2(X)}$

$\emptyset \rightarrow G_{3(A)}$

$G_{2(X)}$ processes its child subsequences before returning the final list of sequences to $G_{1(X)}$. Child subsequences are obtained from the DSO of $G_{2(X)}$, shown in Fig. 5.5b.

$\langle G_{2(X)}, \langle G_4, G_8 \rangle \rangle$

$\langle G_{2(X)}, \langle G_{5(A)} \rangle \rangle$

$G_{2(X)}$ passes its subsequence (∅) to $G_{5(A)}$ which subsequently processes its child sequences as listed in it's DSO (Fig. 5.5c).

$\emptyset \rightarrow G_{5(A)}$

$\langle G_{5(A)}, \{\langle G_9 \rangle, \langle G_8 \rangle\} \rangle$

All the child subsequences of $G_{2(X)}$ are sent back to $G_{2(X)}$ and combined with its own sebsequence.

$G_{2(X)} \leftarrow \langle G_4, G_8 \rangle$

$G_{2(X)} \leftarrow \langle G_{5(A)}, \{\langle G_9 \rangle, \langle G_8 \rangle\} \rangle$

Similarly, $G_{3(A)}$ also processes its child subsequences before returning the final list of sequences to $G_{1(X)}$. Figure 5.5d shows the list of child subsequences that must be processed

$\langle G_{3(A)}, \{\langle G_6 \rangle, \langle G_{7(X)} \rangle\} \rangle$

We proceed in a similar manner and process the child sequences of $G_{7(X)}$ (shown in Fig. 5.5e). These subsequences are returned to their parent $G_{3(A)}$ as follows:

$G_{3(A)} \leftarrow \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{10}, G_{12} \rangle \rangle\}$

$G_{3(A)} \leftarrow \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{11}, G_{13} \rangle \rangle\}$

Both $G_{2(X)}$ and $G_{3(A)}$ merge their child sequences with their own subsequence and return the resulting sequences to their parent $G_{1(X)}$.

$G_{1(X)} \leftarrow \langle G_{2(X)}, \langle G_4, G_8 \rangle\rangle$

$G_{1(X)} \leftarrow \langle G_{2(X)}, \langle G_{5(A)}, \{\langle G_9 \rangle, \langle G_8 \rangle\}\rangle\rangle$

$G_{1(X)} \leftarrow \langle G_{3(A)}, \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{10}, G_{12} \rangle\rangle\}\rangle$

$G_{1(X)} \leftarrow \langle G_{3(A)}, \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{11}, G_{13} \rangle\rangle\}\rangle$

$G_{1(X)}$ receives the child decomposition sequences and combines them with its own subsequence and generates the set of all *ORGMod* labels that can be derived for goal $G_{1(X)}$.

$\langle G_{1(X)}, \langle G_{2(X)}, \langle G_4, G_8 \rangle\rangle\rangle$

$\langle G_{1(X)}, \langle G_{2(X)}, \langle G_{5(A)}, \{\langle G_9 \rangle, \langle G_8 \rangle\}\rangle\rangle\rangle$

$\langle G_{1(X)}, \langle G_{3(A)}, \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{10}, G_{12} \rangle\rangle\}\rangle\rangle$

$\langle G_{1(X)}, \langle G_{3(A)}, \{\langle G_6 \rangle, \langle G_{7(X)}, \langle G_{11}, G_{13} \rangle\rangle\}\rangle\rangle$

Result

When the process concludes, we obtain the list of all possible *ORGMod* labels from the decomposition sequence of $G_{1(X)}$. In the previous illustrations, we have omitted *exclusion sets* for the sake of simplicity. Exclusion sets are obtained when child sequences are assembled by the parent sequence. The final list of *ORGMod* labels, including exclusion sets, are as follows:

*ORGMod*1: $\langle G_1, \langle\langle G_2, \langle\langle G_4, G_8 \rangle, [G_2, G_5]\rangle\rangle, [G_1, G_3]\rangle\rangle$

*ORGMod*2: $\langle G_1, \langle\langle G_2, \langle\langle G_5, \{\langle G_9 \rangle, \langle G_8 \rangle\}\rangle, [G_2, G_4]\rangle\rangle, [G_1, G_3]\rangle\rangle$

*ORGMod*3: $\langle G_1, \langle\langle G_3, \{\langle G_6 \rangle, \langle G_7, \langle\langle G_{10}, G_{12} \rangle, [G_7, G_{11}]\rangle\rangle\}\rangle, [G_1, G_2]\rangle\rangle$

*ORGMod*4: $\langle G_1, \langle\langle G_3, \{\langle G_6 \rangle, \langle G_7, \langle\langle G_{11}, G_{13} \rangle, [G_7, G_{10}]\rangle\rangle\}\rangle, [G_1, G_2]\rangle\rangle$

Figure 5.6 highlights the four different *ORGMod*s that have been derived using the process described above. Consider the first *ORGMod*. At G_1 we choose the path G_1, G_2 and, hence, $[G_1, G_3]$ becomes the exclusion set. Similarly, at G_2, we choose G_4 over G_5 and, hence, $[G_2, G_5]$ becomes the exclusion set for this point.

The process described above is exhaustive and generates all possible *ORGMod* labels that can be derived for any desired goal. Each such *ORGMod* label consists of a list of goals that exist in sequences—exclusion sets (*OR-decompositions*) or parallel sets (*AND-decompositions*). Once we obtain the *ORGMod* labels corresponding to some goal, we can extract the *immediate-func* annotations and reconcile them to derive the *cumulative-func* annotations for individual goals. The process of semantic reconciliation has been elaborated in the next section.

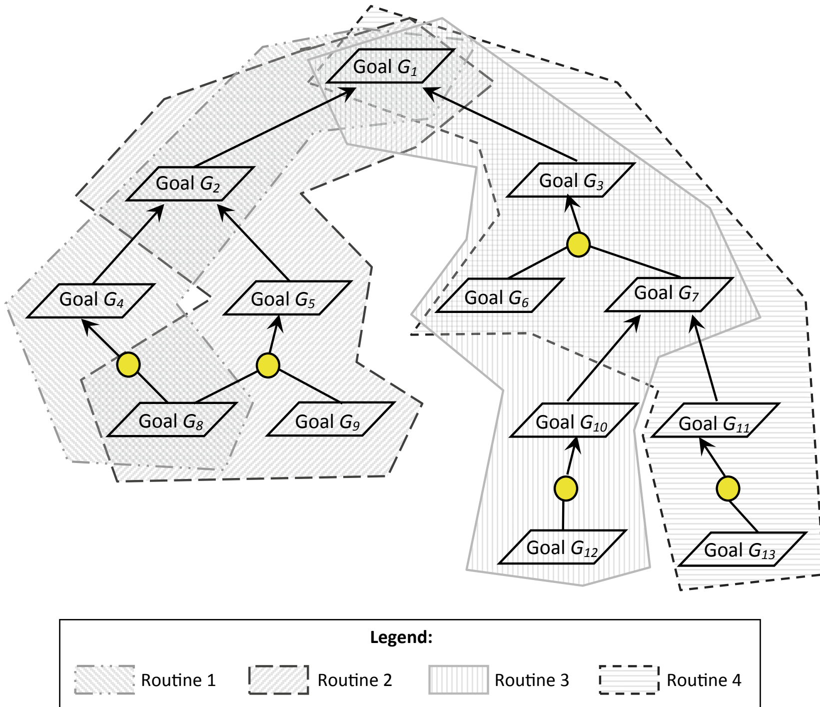


Fig. 5.6 The four possible *ORGMods* that have been derived for goal G_1, highlighted within the goal model of actor A_i

5.1.2 Semantic Reconciliation Operators

The *semantic reconciliation* operation takes the immediate satisfaction conditions of the higher level goal and cumulative satisfaction conditions of the lower level goals as input and determine the cumulative satisfaction conditions of the higher level goal as output. This definition implies that, for leaf-level goals, the immediate and cumulative satisfaction conditions are identical, provided they are not dependant on leaf-level goals residing in other actor boundaries. Consider the following simple example, where we have the higher level goal *Make_Payment* which decomposes to the lower level goals *Check_Balance* and *Transfer_Funds*. Assuming informal annotations of satisfaction conditions, we may have an semantic annotation scenario as follows:

1. *Check_Balance*: `Insufficient`
2. *Transfer_Funds*: `Cheque_bounced`
3. *Make_Payment*: `Payment_done`

Clearly, this results in an inconsistent satisfaction conditions scenario. Background rules may be defined on the underlying knowledge base to prevent such inconsistent satisfaction conditions from occurring in the same satisfaction scenario. We assume that every goal model has an underlying knowledge base (KB) that provides the basis for preventing such inconsistencies.

Assuming that immediate satisfaction conditions are represented as sets of canonical, non-redundant clauses in conjunctive normal form (CNF) [2–4], we represent the immediate satisfaction condition of a goal A as the set $IE(A) = \{ie_1, ie_2, \ldots, ie_n\}$ and its derived cumulative satisfaction condition as the set $CE(A) = \{ce_1, ce_2, \ldots, ce_m\}$, for a given *ORGMod*. Let A_i and A_j be two adjacent level goals in the goal model. We define $rec(A_i, A_j)$ as the semantic reconciliation operator that derives the cumulative satisfaction condition of A_i by combining its immediate satisfaction conditions with the cumulative satisfaction conditions of A_j, i.e., $CE(A_i) = rec(A_i, A_j)$. While deriving the cumulative satisfaction conditions of the goal A_i, we consider two components:

- All immediate satisfaction conditions of A_i that are derivable from the cumulative satisfaction conditions of A_j, given by $\{IE(A_i) \cap CE(A_j)\}$, and
- All additional cumulative satisfaction conditions of A_j whose negations are not listed in the immediate satisfaction conditions of A_i, given by $\{CE(A_j) \setminus \neg IE(A_i)\}$.

Thus, we define the semantic reconciliation operation as follows:

$$CE(A_i) = rec(A_i, A_j) = \left\{ IE(A_i) \cap CE(A_j) \right\} \bigcup \left\{ CE(A_j) \setminus \neg IE(A_i) \right\}, \quad (5.1)$$

such that $\neg IE(A_i) = \{\neg ie_1, \neg ie_2, \ldots, \neg ie_n\}$.

If consistency is not satisfied for all members of $CE(A_j)$, then we proceed to include as many cumulative satisfaction conditions of A_j as possible while maintaining consistency with the knowledge base KB. The following example provides a better understanding of the reconciliation operation.

Example: Let A and B be adjacent level goals in an goal model where B is the next lower level below A. Let $IE(A) = \{a, b, c\}$ and $CE(B) = \{\neg b, c, d, e\}$. The cumulative satisfaction condition of A, denoted by $CE(A)$ can be evaluated as follows:

$$\begin{aligned}
IE(A) &= \{a, b, c\} \\
CE(B) &= \{\neg b, c, d, e\} \\
CE(A) &= rec(A, B) \\
&= \{IE(A) \cap CE(B)\} \cup \\
&\{CE(B) \setminus \neg IE(A)\} \\
IE(A) \cap CE(B) &= \{c\}, \\
\neg IE(A) &= \{\neg a, \neg b, \neg c\}, \\
CE(B) \setminus \neg IE(A) &= \{\neg b, c, d, e\} \setminus \{\neg a, \neg b, \neg c\} \\
&= \{c, d, e\} \\
\therefore CE(A) &= \{c\} \cup \{c, d, e\}\} = \{c, d, e\}.
\end{aligned}$$

Theorem 5.1 *For finding out the cumulative satisfaction condition of any two adjacent level goals A_i and A_j (given by Eq. 5.1), the intersection operation is redundant, i.e., the semantic reconciliation operation $rec(A_i, A_j)$ can be represented as*

$$CE(A_i) = rec(A_i, A_j) = \{CE(A_j) \setminus \neg IE(A_i)\} \tag{5.2}$$

Proof We prove the above theorem by establishing that $IE(A_i) \cap CE(A_j) \subseteq CE(A_j) \setminus \neg IE(A_i)$, and, hence, redundant.

Before proceeding to the proof we would like to clarify the notations. For any two satisfaction condition sets X and Y the universe of satisfaction conditions for their semantic reconciliation operation is given by $U = X \cup Y \cup \neg X \cup \neg Y$ where $\neg X$ and $\neg Y$ can be derived by taking the negation of each satisfaction condition within the set. Also, we know that *set difference* between two sets X and Y can be expressed using *set intersection* as $X \setminus Y = X \cap Y^C$ where Y^C represents the compliment set of Y with respect to the universe U.

$$\begin{aligned}
&\because CE(A_j) \setminus \neg IE(A_i) = CE(A_j) \cap (\neg IE(A_i))^C, \\
&\text{we have to prove: } CE(A_j) \cap IE(A_i) \subseteq CE(A_j) \cap (\neg IE(A_i))^C \\
&\Rightarrow \text{we have to prove: } IE(A_i) \subseteq (\neg IE(A_i))^C.
\end{aligned}$$

For any two satisfaction condition sets X and Y along with their universe U, the set $(\neg X)^C$ consists of the union of three different subsets

1. All elements of the set X.
2. All elements in Y that are not in $\neg X$, i.e., $Y \setminus \neg X$.
3. All elements in $\neg Y$ that are not in $\neg X$, i.e., $\neg Y \setminus \neg X$.

Thus, we have the following derivation:

$$
\begin{aligned}
(\neg X)^C &= X \cup (Y \setminus \neg X) \cup (\neg Y \setminus \neg X) \\
&= X \cup (Y \cap \neg X^C) \cup (\neg Y \cap \neg X^C) \\
&= X \cup \{(Y \cup \neg Y) \cap \neg X^C\} \qquad \text{[Distributive Law]}
\end{aligned}
$$

If the set $\{(Y \cup \neg Y) \cap \neg X^C\} = \emptyset$, then $(\neg X)^C = X$; else $X \subset (\neg X)^C$. Thus, we can conclude that $X \subseteq (\neg X)^C$. Replacing the set X with $IE(A_i)$ and the set Y with $CE(A_j)$, we can conclude that

$$
\begin{aligned}
IE(A_i) &\subseteq (\neg IE(A_i))^C \\
\Rightarrow CE(A_j) \cap IE(A_i) &\subseteq CE(A_j) \cap (\neg IE(A_i))^C \\
\Rightarrow CE(A_j) \cap IE(A_i) &\subseteq CE(A_j) \setminus \neg IE(A_i).
\end{aligned}
$$

$$\therefore \text{ from Eq. 5.1, } CE(A_i) = rec(A_i, A_j) = \{CE(A_j) \setminus \neg IE(A_i)\}.$$

■

Goal decompositions have multiple children in most cases, and this requires the semantic reconciliation machinery to be more generic. We need to explicitly define the mechanisms when there are multiple subgoals resulting from a given goal decomposition. This becomes mandatory for goals that undergo *AND-decomposition*. However, for a goal undergoing *OR-decomposition*, the interpretation changes. As already stated, we perform semantic reconciliation on a *per ORGMod* basis. This implies that we select only one alternative whenever we encounter an OR-decomposition. However, requirement analysts may desire to answer questions like—"*Do I have a strategy that works properly?*" In order to answer such questions, and for completeness of the framework, we need to specify the machinery for semantic reconciliation over OR-decompositions as well.

5.1.2.1 AND-Reconciliation Operator

AND-decompositions can occur within an *ORGMod* itself. The example shown in Fig. 5.2 illustrates an *AND-decomposition* of goal G_3 into goals G_6 and G_7. Using our bottom-up approach, the cumulative satisfaction conditions of G_6 and G_7 will be evaluated first. Since G_6 is a leaf-level goal, $CE(G_6) = IE(G_6)$. The cumulative satisfaction condition of goal G_7 is obtained from the next level goal G_{10} as $CE(G_7) = rec(G_7, G_{10})$. The *AND-reconciliation* operation will be required when we try to evaluate the cumulative satisfaction conditions of the goal G_3 using $CE(G_6)$ and $CE(G_7)$.

Let A_j and A_k be two goals that result from the *AND-decomposition* of the higher level goal A_i. Let $CE(A_j) = \{ce_{j1}, ce_{j2}, \ldots, ce_{jm}\}$ and $CE(A_k) = \{ce_{k1}, ce_{k2}, \ldots, ce_{kn}\}$, respectively, where ce_{xy} represents the y-th cumulative satisfaction condition

of goal x. Let the immediate satisfaction conditions of A_i be denoted as $IE(A_i)$. In that case, we define $CE(A_i)$ using the *ANDrec*() semantic reconciliation operation as follows:

$$ANDrec(A_i, A_j, A_k) = \{rec(A_i, A_j) \cup rec(A_i, A_k)\}. \tag{5.3}$$

In general, if a goal A_N undergoes an *AND-decomposition* to generate the set of goals $A_1, A_2, \ldots, A_K$, then we can define $CE(A_N)$ using the *ANDrec*() operation as follows:

$$ANDrec(A_N, A_1, A_2, \ldots, A_K) = \left\{ \bigcup_{P=1}^{K} rec(A_N, A_P) \right\}. \tag{5.4}$$

Guard condition. Since goal models are sequence agnostic, the ordering of sibling events that stem from an AND-decomposition is abstracted from the model description. Thus, a correct goal model design demands that, during runtime, the order of executing events should not impact the state of the system. For example, with respect to Eq. 5.3, the system should reach the same state of affairs if A_j is performed before A_k or *vice versa*. This property is known as "`Commutativity of State Updation`". Let *State_Updt*() denote the state updation operator such that *State_Updt*(A_p) results in changing the current state of the system by incorporating the satisfaction conditions of performing event A_p and obtaining a new state. Thus, with respect to Eq. 5.3, the commutativity property demands that

$$State_Updt(A_j, State_Updt(A_k)) =$$

$$State_Updt(A_k, State_Updt(A_j)) \tag{5.5}$$

In general, if a goal A_N undergoes an *AND-decomposition* to generate the set of goals $A_1, A_2, \ldots, A_K$, then commutativity is satisfied if applying the state updation operator on any random ordering of these K events, results in the same final state. Commutativity of State Updation must be satisfied for AND-decompositions. For OR-decompositions, we need not worry about commutativity as they represent alternate strategies and we perform analysis on a per *ORGMod* basis.

5.1.2.2 OR-Reconciliation Operator

An *OR-decomposition* provides alternate strategies for achieving the same goal. Since an *ORGMod* chooses one particular alternative whenever it encounters an *OR-decompositi-on*, we need an *OR-reconciliation* whenever we want to combine the cumulative satisfaction conditions of two or more *ORGMod*s (or subroutines) at the point of an *OR-decomposition*. Figure 5.7 illustrates an example where we have two alternate *ORGMod*s for satisfying the goal G_1. The two *ORGMod*s are denoted as $\langle G_1, \langle G_3, \{\langle G_6\rangle, \langle G_7, \langle G_{10}, G_{12}\rangle\rangle\}\rangle\rangle$ and $\langle G_1, \langle G_2, \langle G_5, \{\langle G_9\rangle, \langle G_8\rangle\}\rangle\rangle\rangle$.

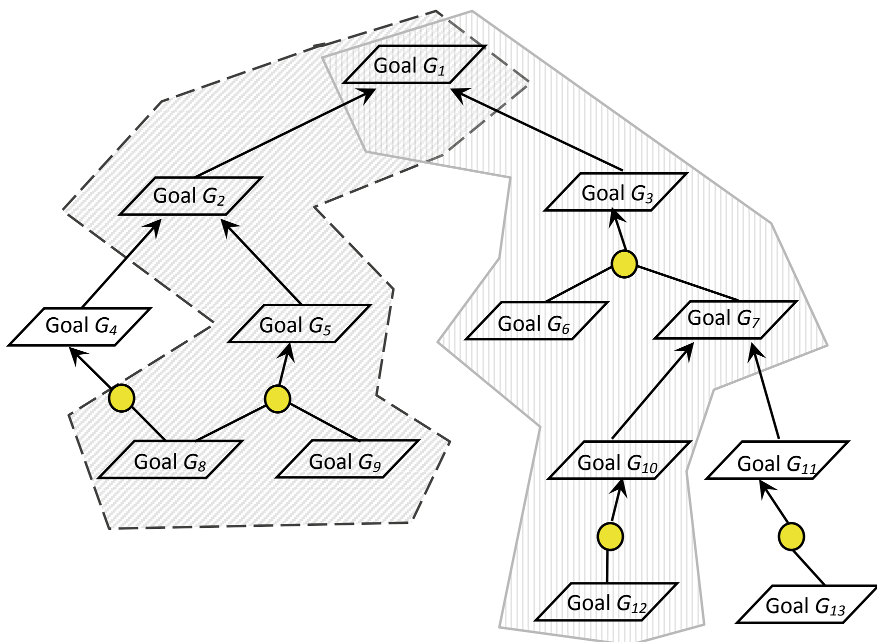


Fig. 5.7 Two *ORGMod*s highlighted for goal G_1 which undergoes a *OR-decomposition* within the goal model of actor A_i

Let A_j and A_k be two goals that result from the *OR-decomposition* of the higher level goal A_i and represent two different strategies for satisfying A_i. Let $CE(A_j)$ = $\{ce_{j1}, ce_{j2}, \ldots, ce_{jm}\}$ and $CE(A_k)$ = $\{ce_{k1}, ce_{k2}, \ldots, ce_{kn}\}$, respectively. Let the immediate satisfaction conditions of A_i be denoted as $IE(A_i)$. In that case, we define $CE(\mathrm{A}_i)$ using the *ORrec*() semantic reconciliation operation as follows:

$$ORrec(A_i, A_j, A_k) = \{\{rec(A_i, A_j)\}, \{rec(A_i, A_k)\}\}. \tag{5.6}$$

In general, if a goal A_N undergoes an *OR-decomposition* to generate the set of goals $A_1, A_2, \ldots, A_K$, then we can define $CE(A_N)$ using the *ORrec*() operation as follows:

$$ORrec(A_N, A_1, \ldots, A_K) = \{rec(A_N, A_x) | \forall x, A_x \in \{A_1, \ldots, A_K\}\}. \tag{5.7}$$

5.1.3 Illustrative Examples

Let us illustrate the working of the semantic reconciliation formalism with the help of some illustrative examples. We consider three different types of functional checks on an annotated goal model—namely, *entailment, consistency,* and *minimality*. Of these,

satisfying *entailment* and *consistency* is mandatory as they ensure the correctness of a goal model. *Minimality* is an optional check that does not result in incorrect system states. We illustrate four different examples that demonstrate different degrees of correctness for goal models.

Case 1: Entailment not satisfied but consistency satisfied Consider the goal model shown in Fig. 5.8. It consists of a primary goal G that undergoes an OR-decomposition into goals G_1 and G_2, which further undergo AND-decompositions.

Every goal has been labelled with their immediate satisfaction conditions, as specified by the requirement analysts. We now perform a semantic reconciliation over this goal model using Eqs. 5.4 and 5.7, defined in previous sections. The cumulative satisfaction conditions of goal G_1 is obtained using Eq. 5.4 as

$$\begin{aligned} CE(G_1) &= ANDrec(G_1, G_3, G_4, G_5) \\ &= \{rec(G_1, G_3) \cup rec(G_1, G_4) \cup rec(G_1, G_5)\} \\ rec(G_1, G_3) &= \{\{p, s\} \cup \emptyset\} = \{p, s\}, \\ &\qquad\qquad \text{and } [CE(G_3) \cap \neg IE(G_1) = \emptyset] \\ rec(G_1, G_4) &= \{\{\neg q\} \cup \emptyset\} = \{\neg q\}, \\ &\qquad\qquad \text{and } [CE(G_4) \cap \neg IE(G_1) = \emptyset] \\ rec(G_1, G_5) &= \{\emptyset \cup \{t\} = \{t\}, \\ &\qquad\qquad \text{and } [CE(G_5) \cap \neg IE(G_1) = \emptyset] \end{aligned}$$

$$\therefore CE(G_1) = \{\{p, s\} \cup \{\neg q\} \cup \{t\}\} = \{p, \neg q, s, t\}$$

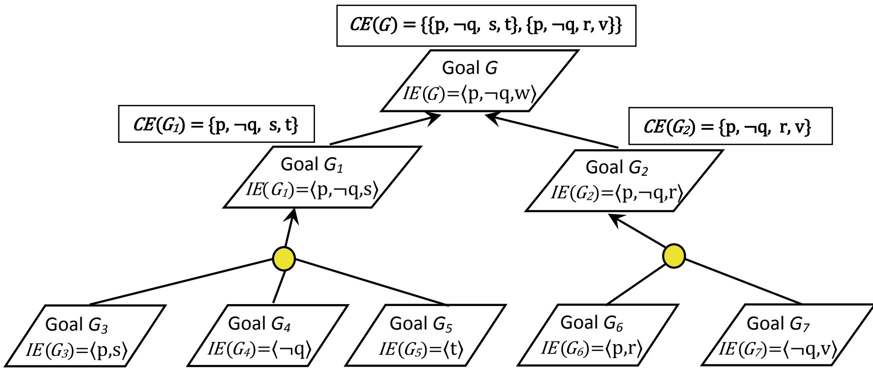


Fig. 5.8 An illustrative example showing how semantic reconciliation can be used to detect problems in *entailment* although *consistency* is ensured

This has been shown in the figure as a label outside goal G_1. Since $IE(G_1) \subseteq CE(G_1)$, this *AND*-decomposition satisfies both *entailment* and *consistency*. Similarly, we proceed to evaluate the cumulative satisfaction conditions of goal G_2 as

$$\begin{aligned} CE(G_2) &= ANDrec(G_2, G_6, G_7) \\ &= \{p, \neg q, r, v\} \end{aligned}$$

This cumulative satisfaction condition has also been shown in the figure outside goal G_2. Again, since $IE(G_2) \subseteq CE(G_2)$, this *AND*-decomposition also satisfies both *entailment* and *consistency*. Following the bottom-up approach, we now proceed to evaluate the cumulative satisfaction conditions of goal G using Eq. 5.7 as follows

$$\begin{aligned} CE(G) &= ORrec(G, G_1, G_2) \\ &= \{\{p, \neg q, s, t\}, \{p, \neg q, r, v\}\} \end{aligned}$$

The cumulative satisfaction conditions of goal G has been labelled in the figure. Since neither of the members in $CE(G_1)$ or $CE(G_2)$ have any mutually conflicting satisfaction conditions with $IE(G)$, hence, we conclude that the semantic reconciliation is *consistent*. However, since both $IE(G) \nsubseteq CE(G_1)$ and $IE(G) \nsubseteq CE(G_2)$, we conclude that none of the strategies ensure *entailment*.

Case 2: Entailment satisfied but consistency is not Consider the goal model shown in Fig. 5.9. It consists of a primary goal G that undergoes an *OR*-decomposition into goals G_1 and G_2, which further undergo *AND*-decompositions.

Every goal has been labelled with their immediate satisfaction conditions, as specified by the requirement analysts. We now perform a semantic reconciliation over

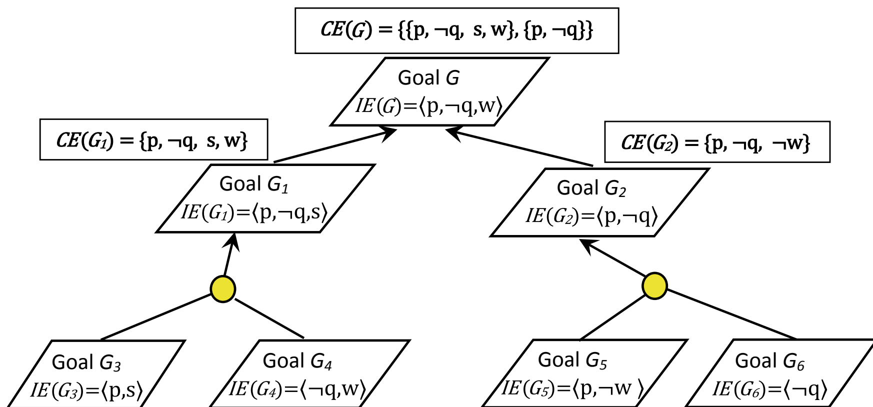


Fig. 5.9 An example showing a *consistency* conflict between the immediate annotations of parent goal G and the cumulative annotations of child goal G_2

this goal model using Eqs. 5.4 and 5.7, defined in previous sections. The cumulative satisfaction conditions of goal G_1 is obtained using Eq. 5.4 as

$$CE(G_1) = ANDrec(G_1, G_3, G_4) = \{p, \neg q, s, w\}$$

Since $IE(G_1) \subseteq CE(G_1)$, this *AND*-decomposition satisfies both *entailment* and *consistency*. Similarly, we proceed to evaluate the cumulative satisfaction condition of goal G_2 as

$$CE(G_2) = ANDrec(G_2, G_5, G_6) = \{p, \neg q, \neg w\}$$

Again, since $IE(G_2) \subseteq CE(G_2)$, this *AND*-decomposition also satisfies both *entailment* and *consistency*. Following the bottom-up approach, we now proceed to evaluate the cumulative satisfaction condition of goal G using Eq. 5.7 as follows

$$\begin{aligned}
CE(G) &= ORrec(G, G_1, G_2) \\
&= \{rec(G, G_1), rec(G, G_2)\} \\
rec(G, G_1) &= \{\{p, \neg q, w\} \cup \{s\}\} = \{p, \neg q, w, s\}, \\
&\qquad \text{and } [CE(G_1) \cap \neg IE(G) = \emptyset] \\
rec(G, G_2) &= \{\{p, \neg q\} \cup \emptyset\} = \{p, \neg q\}, \\
&\qquad \text{and } [CE(G_2) \cap \neg IE(G) = \{\neg w\}]
\end{aligned}$$

$$\therefore CE(G) = \{\{p, \neg q, w, s\}, \{p, \neg q\}\}$$

The cumulative satisfaction conditions of goal G has been labelled in the figure. Since $IE(G) \subseteq \{p, \neg q, w, s\}$, we can say that there is at least one strategy that fulfils *entailment*. However, since $CE(G_2) \cap \neg IE(G)$ is not null, we conclude that *consistency* is not ensured in the second strategy and the conflicting satisfaction conditions are the members of the set $CE(G_2) \cap \neg IE(G)$.

Case 3: Both entailment and consistency are satisfied Consider the goal model shown in Fig. 5.10. This figure is exactly similar to Fig. 5.9 with one minor change. The immediate satisfaction condition of goal G_5 is changed from $\{p, \neg w\}$ to $\{p, w\}$.

This is the best possible outcome that requirement analysts and the client would like to achieve in the requirements phase. The reconciliation of satisfaction conditions is done in the same way as shown in the previous two examples. Assuming that the reader has understood the working principle, we skip the cumulative satisfaction condition evaluation process. However, there are three interesting observations in this example that needs to be highlighted.

1. Unlike the previous example shown in Fig. 5.9, $CE(G_2) \cap \neg IE(G) = \emptyset$. This implies that the inconsistency issue existing in the previous example, does not persist in this scenario. Also, since $IE(G) \subseteq CE(G)$, *entailment* is satisfied.

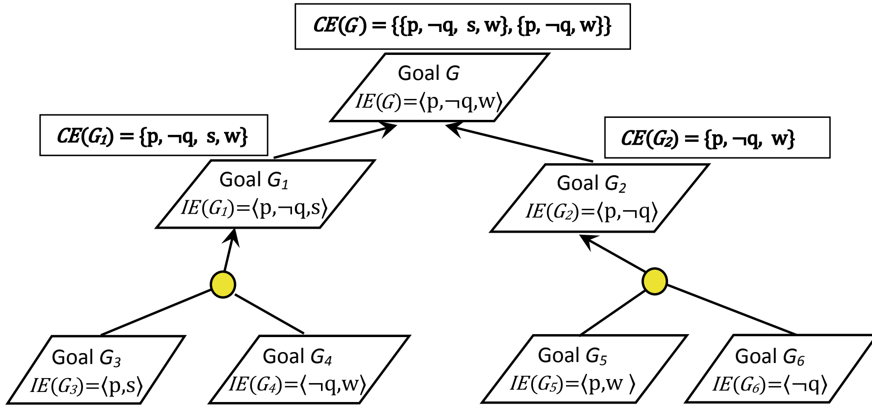


Fig. 5.10 An example showing how *entailment* and *consistency* are both satisfied

2. This example also justifies our bottom-up approach. None of the subgoals G_1 or G_2 have $w \in IE(G_1)$ or $w \in IE(G_2)$. However, semantic reconciliation ensures that the effect w gets propagated from goals G_4 and G_5 upwards such that all members of the cumulative satisfaction condition set $CE(G)$ satisfy the immediate satisfaction conditions $IE(G)$.
3. This is the kind of situation where *minimality* can play some role for system designers. Both members of the set $CE(G)$ satisfy the *entailment* and *consistency* conditions. In such a situation, designers may choose a particular strategy that produces a minimal set of additional satisfaction conditions. In this example, $\{p, \neg q, w\}$ is a more minimal solution for satisfying goal G as compared to $\{p, \neg q, w, s\}$ as the latter produces an additional satisfaction condition of $\{s\}$.

Case 4: Neither entailment nor consistency are satisfied From the previous examples, one can easily visualize a scenario where neither *entailment* nor *consistency* is satisfied. This scenario is not at all desirable from the client's as well as designer's perspective. Requirement engineers may have to revisit the client and perform refinements of the previously elicited requirement specifications. The analysts can also help the client by highlighting erroneous and conflicting requirement specifications.

Semantic Reconciliation Algorithm (SRA)

`Input`: A goal model whose model elements have been annotated with immediate satisfaction conditions

`Output`: Annotation of the model elements with cumulative satisfaction conditions derived from the semantic reconciliation process

`Algorithm_SRA:`

Step-1: Start.
Step-2: Identify a *ORGMod* using the *ORGMod* Extraction Algorithm.
Step-3: For each *ORGMod* repeat the following steps.

(*a*) Begin at the leaf-level goals and evaluate the cumulative satisfaction condition $CE(L_a)$ of each leaf-level goal L_a.
(*b*) Go to the previous level and check if the goals in this level undergo an AND-decomposition or an OR-decomposition. Depending on the type of merge operation required, evaluate the respective cumulative satisfaction conditions using Eqs. 5.4 or 5.7, respectively.
(*c*) Repeat the previous two steps till we reach the root of the extracted *ORGMod*.

Step-4: Check if the immediate satisfaction condition set $IE()$ and the cumulative satisfaction condition set $CE()$ of the root satisfies *entailment* and *consistency*. If not, then raise a flag to the requirement analysts.
Step-5: Repeat Steps 3–4 for all possible *ORGMod*s.
Step-6: Stop.

5.2 Resolving Conflicts Using Model Refactoring

The SRA algorithm helps in identifying *entailment* and *consistency* issues during the bottom-up semantic reconciliation process. However, the AFSR framework does not merely identify these issues. It also makes an attempt to resolve these issues by refactoring the given goal model. Requirement analysts are provided with possible solutions and necessary changes that need to be incorporated into the requirements model in order to satisfy *entailment* and *consistency*. In the following sections, we first illustrate how we attempt to resolve these issues using test cases. We then propose a formal algorithm for doing the same.

5.2.1 Entailment Issues

The SRA algorithm raises an *entailment* issue at any point in the semantic reconciliation process when the immediate satisfaction conditions $IE(M)$ of a goal M are not achieved or satisfied by the subtree rooted at M. The cumulative satisfaction conditions of the subtree are captured in $CE(M)$. Depending on whether M undergoes an AND-decomposition or an OR-decomposition, $CE(M)$ contains only one member or multiple members, respectively. The cumulative satisfaction condition for AND-decompositions is one single set of satisfaction conditions obtained using Eq. 5.4 defined in Sect. 5.1.2.1. On the other hand, the cumulative satisfaction

condition set for OR-decompositions contains as many members as the number of alternative strategies in the OR-decomposition. Each member is again a set of satisfaction conditions reconciled over that particular *ORGMod*, obtained using Eq. 5.7 defined in Sect. 5.1.2.2.

It is easy to raise an *entailment* issue for AND-decompositions as we only need to check if $IE(M) \nsubseteq CE(M)$. However, for OR-decompositions, we need to check this condition for each individual member of the $CE(M)$ set. In general, we can formally define the condition for raising an *entailment* issue as

$$\exists CE_i \in CE(M), \text{ s.t. } IE(M) \nsubseteq CE_i$$

Once an *entailment* issue is flagged by the `SRA` algorithm, we proceed to derive two data sets for resolving the issue—`deficiency-lists` and `availability-tuples`. The `deficiency-list` $\mathbb{D}$ is used to identify all those immediate satisfaction conditions in $IE(M)$ that are not present in $CE(M)$. This is obtained individually for all members CE_i of $CE(M)$. The set is evaluated as follows:

$$\mathbb{D} = \{IE(M) \setminus CE_i | \forall CE_i \in CE(M)\} \tag{5.8}$$

Once we obtain the `deficiency-list` $\mathbb{D}$ we proceed to explore whether these satisfaction conditions are fulfilled or achieved by goals that lie in other solution paths. We consider a one-to-one mapping, called the *Availability Function*, that maps each member $d_i \in \mathbb{D}$ to a tuple of integers $(n_1, n_2, \ldots, n_k)$. The *Availability* mapping tries to capture the information whether any particular satisfaction condition $d_{ij} \in d_i$ can be fulfilled along other solution paths. It is defined as follows:

$$\mathbb{A} : \mathbb{D} \rightarrow \mathbb{N}^{\Bbbk}$$

such that $\mathbb{A}(d_i) = (n_1, n_2, \ldots, n_k)$ where $k = |d_i|$ and $\forall d_{ij} \in d_i$,

$$n_j = \begin{cases} r, & \text{if } \exists CE_r \in \text{CE(M) s.t.} d_{ij} \in CE_r. \\ 0, & \text{otherwise.} \end{cases} \tag{5.9}$$

Each such tuple corresponding to a `deficiency-list` is called an `availability tuple`. The set of all `availability-tuples` forms the range of the *Availability Function*. In the next two sections, we demonstrate the model refactoring strategies that can be used to resolve *entailment* issues for OR-decompositions and AND-decompositions.

5.2.1.1 Entailment Resolution for OR-Decompositions

Consider the goal model shown in Fig. 5.11. There are two alternate means G_1 and G_2 for fulfilling the high-level goal G. Neither of the members in $CE(G)$ contains all the

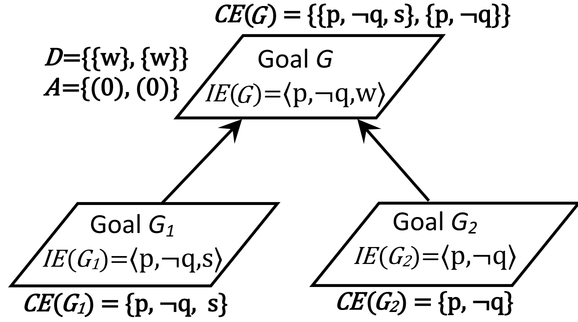


Fig. 5.11 A sample goal model showing failure of *entailment* at goal G that undergoes OR-decomposition

immediate satisfaction conditions in $IE(G)$. We proceed to resolve this *entailment* issue by first listing the `deficiency-list` for each path and then evaluating the `availability-tuple` for each path.
The `deficiency-list` for goal G_1 is given by

$$d_1 = IE(G) \setminus CE_1 = \{p, \neg q, w\} \setminus \{p, \neg q, s\} = \{w\}$$

The `availability-tuple` for goal G_1 is given by $\mathbb{A}(d_1) = (0)$ since w is not contained in CE_2. Similarly, the `deficiency-list` and `availability-tuple` for goal G_2 is obtained as $d_2 = \{w\}$ and $\mathbb{A}(d_2) = (0)$.

This implies that the satisfaction condition$\{w\}$ is not derived from either of the strategies. The intuition behind providing a solution to the requirement analysts is that "*we need to incorporate a goal, say G', which brings about the state of affairs* '*w*' *on the world in which the actor resides*". Thus, we introduce a temporary goal CT_1 with immediate satisfaction condition $IE(CT_1) = \{w\}$ and merge it with goals G_1 and G_2 to achieve the satisfaction condition 'w'in the cumulative satisfaction condition of G. The solution is shown in Fig. 5.12.

5.2.1.2 Entailment Resolution for AND-Decompositions

For AND-decompositions, the solution is not so complex. Consider the *entailment* issue being addressed in Fig. 5.13. Since goal G undergoes an AND-decomposition, its cumulative satisfaction condition set $CE(G)$ contains only one member which is the set of satisfaction conditions derived from all the individual AND-decomposition links.

We proceed to evaluate the `deficiency-list` as follows:

$$d = IE(G) \setminus CE(G) = \{p, \neg q, t, v, w\} \setminus \{p, \neg q, s, \neg r, w\} = \{t, v\}$$

The `availability-tuple` for goal G is given by $\mathbb{A}(d) = (0, 0)$. In fact, for AND-decompositions, the `deficiency-list` will always contain only one set

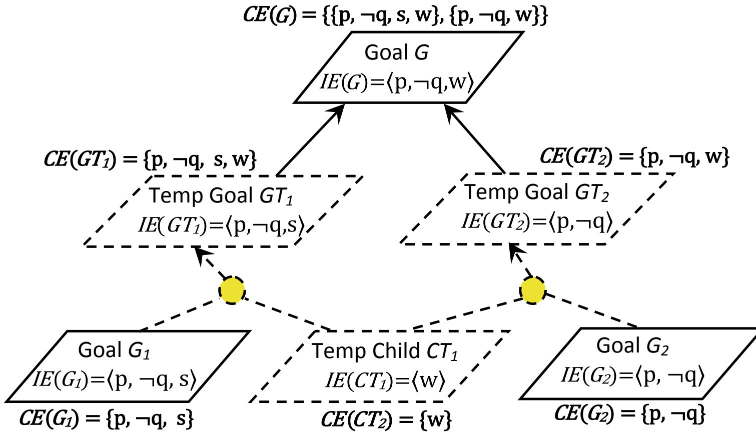


Fig. 5.12 Temporary high-level goals GT_1 and GT_2 are used to merge goals G_1 and G_2 with the temporary goal CT_1

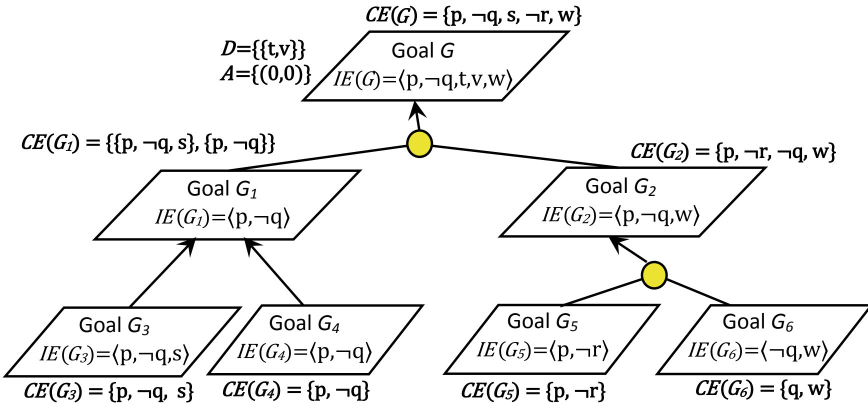


Fig. 5.13 A sample goal model showing failure of *entailment* at goal G that undergoes AND-decomposition

member and it's `availability-tuple` will always be of the form (0, 0,. . .,0) depending on the number of satisfaction conditions in the `deficiency-list` set.

The reason is quite intuitive. The very semantics of an AND-reconciliation necessitate that all distinct satisfaction conditions appearing in all individual paths be reconciled in one set. This, in turn, makes the solution very simple- "*we need to incorporate a goal, say G', which brings about these unaccounted state of affairs (as obtained in the* `deficiency-list`) *on the world in which the actor resides*". We introduce a temporary goal CT_1 with immediate satisfaction condition $IE(CT_1) = \{t, v\}$ and merge it with goals G_1 and G_2 to incorporate these satisfaction conditions in the cumulative satisfaction condition of G. The solution is shown in Fig. 5.14.

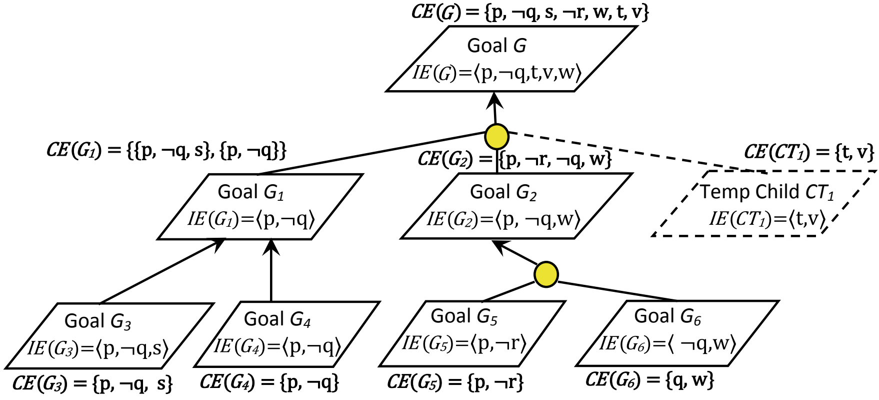


Fig. 5.14 Temporary goal CT_1 is merged with goals G_1 and G_2 to obtain the cumulative semantic annotation of G

Entailment Resolution Algorithm (`ERA`)

`Input`**:** Identify a goal M as a point of *entailment* failure if the following condition is satisfied

$$\exists CE_i \in CE(M), \text{ s.t. } IE(M) \not\subseteq CE_i$$

`Output`**:** A possible solution for entailment resolution using model refactoring

`Algorithm_ERA:`

Step-1: Start.

Step-2: Evaluate the *Deficiency* set $\mathbb{D}$ using Eq. 5.8.

Step-3: Define the *Availability* mapping function $\mathbb{A}$ according to Eq. 5.9.

Step-4: If the point of failure M undergoes OR-Decomposition and links nodes $M_1, M_2, \ldots, M_p$, then propagate the respective `deficiency-list` and the corresponding `availability-tuple` along each of the p paths. For each pa th, do the following:

(*a*) At the next level, create a temporary high-level goal GT_i having the same immediate satisfaction conditions as M_i, i.e., $IE(GT_i) = IE(M_i)$.

(*b*) Create an AND-decomposition of GT_i with its leftmost child being M_i.

(*c*) Add another child CT_i to GT_i whose immediate satisfaction conditions are obtained by concatenating those members in the `deficiency-list` d_i whose corresponding `availability-tuple` values n_j are zero (0).

(*d*) For every other member d_{ij} in the `deficiency-list` that has a non-zero `availability-tuple` value n_j, set up an AND-decomposition link (if it does not exist already) between GT_i and the goal M_i' residing in path n_j such that $d_{ij} \in IE(M_i')$.

Step-5: If the point of failure M undergoes AND-decomposition and produces nodes $M_1, M_2, \ldots, M_q$, then add a temporary child goal CT_i under M and set up an AND-decomposition link between M and CT_i. Annotate CT_i as $CE(CT_i) = IE(CT_i) = \mathbb{D}$.

Step-6: Repeat Steps 2–5 for all goals, in a bottom-up manner, during the semantic reconciliation procedure.

Step-7: Stop.

5.2.2 *Consistency Issues*

Consistency is defined as a condition where the cumulative satisfaction condition of any goal M does not contain mutually conflicting satisfaction conditions. Otherwise, the goal is said to be inconsistent. Inconsistency of goals during the semantic reconciliation process can be classified into two different types—*hierarchic inconsistency* and *sibling inconsistency*.

5.2.2.1 Hierarchic Inconsistency

Hierarchic inconsistency occurs when some immediate satisfaction condition(s) of a parent goal is in conflict with some cumulative satisfaction condition of a child goal. This type of inconsistency can occur for both AND-decompositions and OR-decompositions. In case of OR-decompositions, we need to be worried with only those alternate means that are inconsistent. The system does not fail as long as there exists one alternative that is consistent with respect to satisfaction conditions. For AND-decompositions, the consequences are much more critical and can result in an inconsistent system. Figure 5.15 illustrates an AND-decomposition that results in hierarchic inconsistency.

The SRA algorithm takes care of *hierarchic* inconsistency in the semantic reconciliation process itself. Equation 5.2 of Theorem 5.1 removes all those cumulative satisfaction conditions of the child goal that are inconsistent with immediate

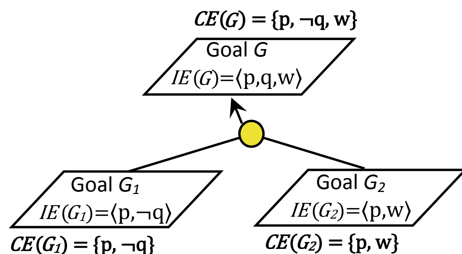


Fig. 5.15 *Hierarchic* inconsistency at goal G arising out of the immediate satisfaction condition q of G and the cumulative satisfaction condition $\neg q$ of goal G_1

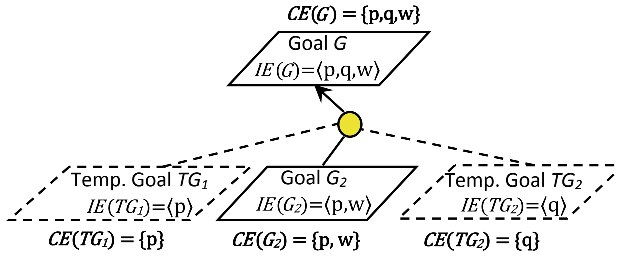


Fig. 5.16 Eliminating inconsistencies in the semantic reconciliation process governed by Eq. 5.2 in Theorem 5.1

satisfaction conditions of the parent goal while evaluating the cumulative satisfaction conditions of the parent goal.

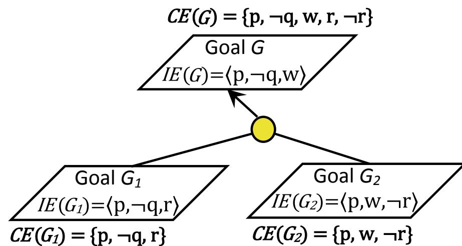
Figure 5.16 shows how hierarchic inconsistency is tackled in the SRA algorithm. We also replace the victim child with a temporary goal TG_1 not containing the inconsistent satisfaction conditions. Now, the problem basically reduces to that of *Entailment Resolution* and the ERA algorithm can be used to resolve it.

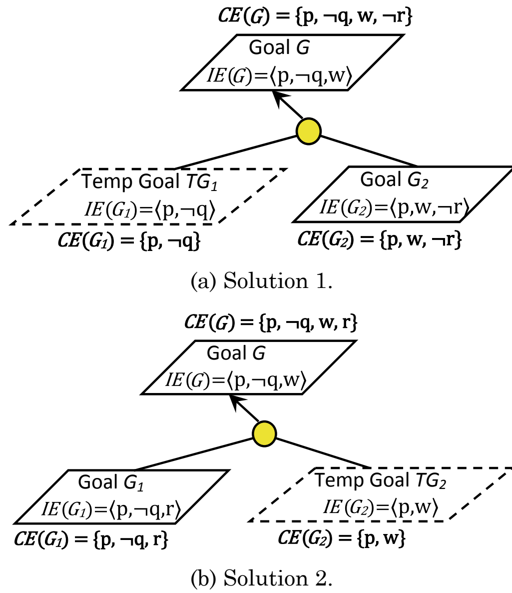
5.2.2.2 Sibling Inconsistency

The other type of inconsistency is *sibling* inconsistency which arises during semantic reconciliation of mutually conflicting satisfaction conditions from child nodes of the same parent node. Figure 5.17 illustrates one such scenario where the cumulative satisfaction conditions of goal G contain both r and $\neg r$, reconciled from child goals G_1 and G_2, respectively.

Unlike resolution of hierarchic inconsistencies, resolving sibling inconsistencies do not result in entailment issues. This implies that removing any one of the conflicting satisfaction conditions from the child goals is sufficient to resolve this type of inconsistency. With respect to the scenario shown in Fig. 5.17, we can highlight to the analysts that either effect r of goal G_1 or effect $\neg r$ of goal G_2, needs to be dealt with. These two solutions are shown in Fig. 5.18. The requirement analysts can then decide

Fig. 5.17 *Sibling* inconsistency at goal G arising out of the immediate satisfaction condition r of goal G_1 and the immediate satisfaction condition $\neg r$ of goal G_2





(a) Solution 1.

(b) Solution 2.

Fig. 5.18 `Solution 1`: eliminates the effect annotation r of goal G_1. `Solution 2`: eliminates the effect annotation $\neg r$ of goal G_2

which particular solution best suits the requirements of the enterprise depending on the consequences and significance of the satisfaction conditions r and $\neg r$.

Consistency Resolution Algorithm (`CRA`)

`Input`: Identify a goal M as a point of *consistency* failure.

`Output`: A possible solution for consistency resolution using model refactoring

`Algorithm_CRA:`

Step-1: Start.
Step-2: Identify the type of inconsistency as *hierarchic* if-

$$\exists ce_i \in CE(M), ie_j \in IE(M) \text{ s.t. } ce_i = \neg ie_j$$

(a) Identify the child M_k which contributes the satisfaction condition ce_i in $CE(M)$, i.e., $ce_i \in CE(M_k)$.
(b) Remove the satisfaction condition ce_i from the cumulative satisfaction condition set as well as its immediate satisfaction condition set, i.e.,

$$CE'(M_k) = CE(M_k) \setminus ce_i$$
$$IE'(M_k) = IE(M_k) \setminus ce_i$$

(c) The goal model is now consistent but the immediate satisfaction condition ie_j of M does not appear in its cumulative satisfaction condition set. This can be resolved by calling the ERA algorithm.

Step-3: Identify the type of inconsistency as *sibling* if-

$$\exists ce_i, ce_j \in CE(M), ce_i, ce_j \notin IE(M) \text{ s.t. } (ce_i = \neg ce_j)$$

(a) Identify the siblings M_{k1} and M_{k2} which contribute the satisfaction conditions ce_i and ce_j in $CE(M)$, respectively. That is,

$$ce_i \in CE(M_{k1}) \text{ and } ce_j \in CE(M_{k2}).$$

(b) Remove the satisfaction condition ce_i from the cumulative and immediate satisfaction condition sets of M_{k1}, i.e.,

$$CE'(M_{k1}) = CE(M_{k1}) \setminus ce_i$$
$$IE'(M_{k1}) = IE(M_{k1}) \setminus ce_i$$

OR, remove the satisfaction condition ce_j from the cumulative and immediate satisfaction condition sets of M_{k2}, i.e.,

$$CE'(M_{k2}) = CE(M_{k2}) \setminus ce_j$$
$$IE'(M_{k2}) = IE(M_{k2}) \setminus ce_j$$

(c) Present both the above alternatives to the requirement analysts as consistent solutions.

Step-4: Repeat Steps 2–3 for all goals, in a bottom-up manner, during the semantic reconciliation procedure.

Step-5: Stop.

Resolving an inconsistency using the proposed Consistency Resolution Algorithm can refactor the goal model in a way that gives rise to new inconsistencies. However, the algorithm can be applied repeatedly to get rid of the newly introduced inconsistencies.

Use Case: Healthcare

Let us continue with the use case discussed in Sect. 5.1. Now let us suppose a couple of changes in the business environment setting:

`Change-1`: The healthcare enterprise passes a regulation that *Long Term Treatment* cannot be provided without consulting a specialist.

`Change-2`: If it is not an emergency, then it must be ensured that patient is not allergic to any chemicals before performing a test. This is to prevent situations

such as an MRI scan (with contrast) becomes the reason of death for a patient who is allergic to contrast fluids like iodine.

These changes in the business environment can be reflected in the goal model of Fig. 5.1 by updating the immediate satisfaction conditions of goal G_3 (for `Change-1`) and task T_1 (for `Change-2`). The modified immediate satisfaction sets of these two goal model artefacts become as follows:

- Modified **KB2**: $Normal_Treatment_Provided \rightarrow (Received_Text \vee Received_Voice) \wedge PreExisting_Disease_Searched \wedge Test_Result_Known \wedge Consulted_Specialist$.
  $IE'(G_3) = \{Normal_Treatment_Provided\} \equiv \{\{\{Received_Text\}, \{Received_Voice\}\}, PreExisting_Disease_Searched, Test_Result_Known, Consulted_Specialist\}$.
- $IE'(T_1) = \{Allergies_Checked, \{\{Sample_Taken\}, \{Performed_Procedure\}\}, Test_Result_Known\}$

If we now use the AFSR framework to compare the cumulative satisfaction conditions (already computed for the previous business setting) with the modified immediate satisfaction conditions (in the current setting), we will notice *entailment* conflicts for both G_3 and T_1. The AFSR framework handles these two conflicts with the Entailment Resolution Algorithm separately as follows:

`ERA-1`: The entailment conflict for G_3 is due to the newly introduced satisfaction condition $\langle Consulted_Specialist \rangle$. Since this condition is not fulfilled by any of the child nodes and G_3 undergoes AND-decomposition, we add another temporary goal called "Consult Specialist" as a child of G_3. We label this goal TG_1.

`ERA-2`: For the entailment conflict of T_1, the AFSR framework finds that the newly added satisfaction condition $\langle Allergies_Checked \rangle$ can be fulfilled by it's sibling R_1. So the AFSR framework sets up a parent–child link between T_1 and R_1.

The modified goal model that incorporates these changes is shown in Fig. 5.19.

5.3 An Implementation Roadmap

5.3.1 The Generalized Framework

Given a finite (or infinite) state space, if a path exists between the start state and the goal state, then A* search is guaranteed to find the optimal path if the heuristic cost at each node is *admissible* and *consistent*. In order to prove that our goal maintenance framework resolves conflicts in a goal model to generate a conflict-free version that also deviates minimally from the original model, we first map the goal maintenance problem to a state-space search problem and then show that the optimal solution

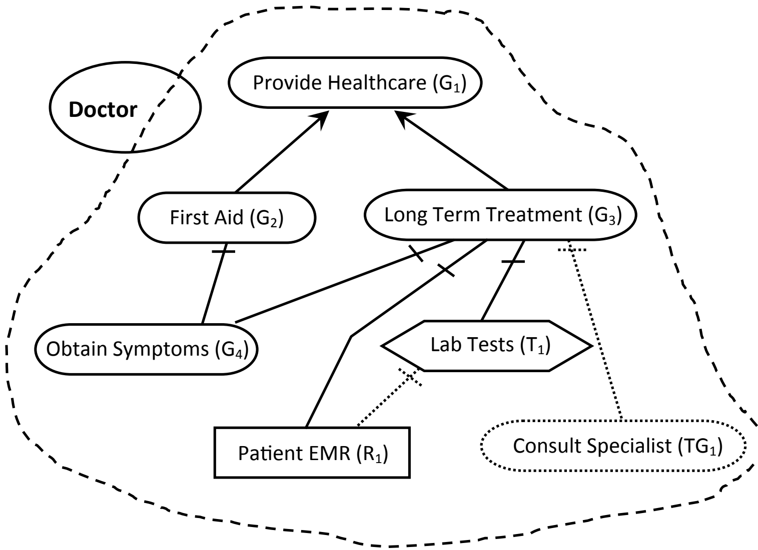


Fig. 5.19 Modified goal model incorporating the two business environment changes `Change-1` and `Change-2`

obtained using A* search is in fact the result that we are interested in. The mapping of the goal modification problem to the state space search problem that we will define requires the following concepts:

1. *α-satisfiability*: Let α represent a set of *change constraints*. A change constraint represents the change that we desire in a goal model, and therefore, the driver of a given goal maintenance exercise. A change constraint defines what must be true of the goal model obtained as output of a goal maintenance exercise. We may, for instance, require that a goal model be updated to include a new goal (at a specific point in a goal model, such as in the form of an AND-refined child sub-goal of a specified parent goal that exists in the prior goal model), to remove an existing goal (i.e., go from a state where the model might entail a given goal to a state where the goal model is guaranteed not to), or to restore consistency of the goal model (in the sense of the *reconciliation* operator discussed earlier). We will view α as a set of constraints of these kinds, and will deem a goal model to be α-`satisfiable` if it satisfies all the constraints listed in the set α.
2. *Goal Modification Operators*: Given a goal model G and a set of change constraints α, we can define goal modification operators that make changes to the given goal model in order to make some non-satisfiable goal (or set of goals) α-satisfiable. Let the modified goal model obtained after the application of some goal modification operator(s) be denoted as G'. The goal modification process can then be represented as the transition $G \rightarrow G'$. Table 5.1 lists the primitive goal modification operators.

Table 5.1 List of primitive goal modification operators

Primitive operators	Function	Illustration
`leaf_rmv(g_l)`	Removing a leaf goal `g_l`, without any dependencies, along with its parent link	
`nleaf_rmv(g_p)`	Removing a non-leaf goal `g_nl`, along with its parent and child links, if they exist	
`add(g)`	Adding a new stand alone goal `g`	
`AND_add(g_p,g_l)`	Adding a new leaf-level goal `g_l` to a parent goal g_p using AND-decomposition link	
`OR_add(g_p,g_l)`	Adding a new leaf-level goal `g_l` to a parent goal g_p using OR decomposition link	
`join(g_p,g,A/O)`	Joining an existing goal `g` with another existing goal `g_p` using AND(`A`)/OR(`O`) decomposition links	
`break(g_p,g)`	Separates an existing goal `g` from another its parent goal `g_p` but does not remove either `g` or `g_p` from the model	

3. *Goal Model Reachability*: Given two different versions G_i and G_j of a goal model G, we say that G_j is `reachable` from G_i if there exists a sequence of goal modification operations from G_i that results in G_j. This is represented as $G_i \rightarrow \cdots \rightarrow G_k \rightarrow \cdots \rightarrow G_j$. In short, this can be represented as $G_i \rightsquigarrow G_j$.

   The state space for G is said to be `completely reachable` if such sequences of goal modification operations exist for every pair of states in the state space. If the state space for G is not completely reachable, we conclude that some of the goal modification operators are overly `coarse-grained`. Again, if the operators are too `fine-grained`, even if there exists a sequence of operator applications that computes an optimal solution, it may be that some prefixes of these sequences will lead to non-optimal evaluations of the cost function.

With these prerequisites we can now proceed to define the state space search problem associated with a goal modification problem.

Definition 5.1 (*State-Space Search Problem*) For a given goal model modification problem, the corresponding state-space search problem can be represented as a 6-tuple $\langle S_{stt}, s_{strt}, \Sigma_{mod}, \Delta_{succ}, \Omega_{heur}, \Gamma_{test}\rangle$ where

$i.$ S_{stt} denotes the set of states that define the state space. Each state $s_i \in S_{stt}$ represents a version of the original goal model, derived by some sequence of operations.

$ii.$ $s_{strt} \in S_{stt}$, denotes the initial state and represents the original goal model G.

$iii.$ Σ_{mod} denotes the set of goal modification operators (listed in Table 5.1) that can be applied to any state to reach a new state in the state space.

$vi.$ Δ_{succ} is the transition function that captures all the state transitions that can occur within the state space S_{stt} due to the application of goal modification operators in Σ_{mod}, i.e., Δ_{succ}: $S_{stt} \times \Sigma_{mod} \rightarrow S_{stt}$.

$iv.$ Ω_{heur} is the heuristic path cost function that has two components—the actual minimum cost for reaching state s_i from the start state (Ω_{act}), and the estimated cost of reaching a goal state from state s_i (Ω_{est}). Thus,

$$\Omega_{heur}(s_i) = \Omega_{act}(s_{strt}, s_i) + \Omega_{est}(s_i, s_{fin}),$$
$$\text{where } s_{fin} \text{ is a final state.}$$

$iv.$ Γ_{test} is the goal test function which identifies a goal state as one where the goal model is *completely* α*-satisfiable*.

Given a goal model G, the optimization challenge is to find the minimally different goal model G' that is completely α-satisfiable. In order to prove that applying A* search to the above state space search problem gives an optimal solution, we only need to prove that the heuristic cost function Δ_{heur} is *admissible* and *consistent*. We define a goal model proximity relation $<_G$ that is used to identify the optimal solution.

Definition 5.2 (*Goal Model Proximity*) Each goal model G is associated with a proximity relation $<_G$ such that for any two variations G_i and G_j of the original goal model, $G_i <_G G_j$ implies that G_i is closer to G than G_j. $<_G$ is defined by a triple $\langle <^V_G, <^E_G, <^{EFF}_G\rangle$ for measuring the proximity of vertices, edges and cumulative satisfaction conditions, respectively. $G_i <_G G_j$ iff each of $G_i <^V_G G_j$, $G_i <^E_G G_j$ and $G_i <^{EFF}_G G_j$ holds.

Definition 5.3 ($\alpha - minimality$) A goal model variation G' is an α-minimal solution with respect to a given goal model G and a set of change constraints α iff each of the following hold:

- G is not *completely* α*-satisfiable*.
- G' is *completely* α*-satisfiable*.
- There exists no goal model G'' such that $G'' <_G G'$ and G'' is *completely* α*-satisfiable*.

5.3.2 *Taxonomy of Goal Model Proximity Measures*

We offer a range of intuitions for assessing goal model proximity:

1. Structural proximity: This is often realized via graph edit distance.
2. Semantic proximity: This can be realized by using measures of distances between theories (such as the Dalal distance).
3. Hierarchy-sensitive proximity measures: Here, we can implement multiple distinct intuitions. We might argue that a higher level goal is a more stable, reliable and ultimately more important indicator of an organization's motivational stance, and thus accord higher priority to goals higher in a goal tree. Alternatively, we might argue that goals lower in a goal tree more accurately reflect investments in infrastructure, and that being more willing to modify these entails that we are willing to modify actual goal realization infrastructure without heed to cost. In such situations, we might find it useful to accord higher priority to lower level goals than higher level ones.
4. Compliance-based proximity: decided by the degree of conformance to a given rule set.
5. Softgoal-based proximity: decided by the degree of contribution to a given set of softgoals.
6. Component/Service-based proximity: decided by the number of off-the-shelf components/services that may be assigned to the lower level goals and the number of new components/services that have to be developed from scratch.
7. Mining-based proximity: decided by the greater number of goals that can be mined from event logs.

5.3.3 *Evaluating Goal Model Proximity*

In this section, we create a specific instantiation of the generalized framework and demonstrate how the framework can be used to address the goal modification problem. Before we instantiate the six tuples associated with the state space search formulation, let us first elaborate on the goal model proximity relation $<_G$ that will be used for distinguishing the optimal solution from other non-optimal solutions. In this section, we concentrate on structural and semantic proximity measures only for measuring the distance between two goal models; other proximity measures may also be incorporated to evaluate this distance.

Let ∇_G be the distance measurement operator such that, for any two variations G_1 and G_2 of a given goal model, $G_1 \nabla_G G_2$ represents the relative distance between variations G_1 and G_2 with respect to the original goal model G. This is evaluated as $(G_1 \nabla G) - (G_2 \nabla G)$. We say that $G_1 <_G G_2$ iff $(G_2 \nabla_G G_1) > 0$. Structural or syntactic proximity requires comparison of vertices and edges for which we use the measurement operators ∇_G^V and ∇_G^E, respectively. Semantic proximity uses cumulative satisfaction conditions and measures it using the operator ∇_G^{EFF}. We base the

proximity calculations on set cardinality, as defined in [5]. Let us look into the Entailment Resolution Algorithm (ERA) and see how these distances can be measured.

Let $G (= \langle V, E\rangle)$ be the original goal model whose current modified version $G_1 (= \langle V_1, E_1\rangle)$ has an entailment conflict. ERA is applied to resolve this conflict and reach the state $G_2 (= \langle V_2, E_2\rangle)$. The distance measurement operators are then calculated as follows:

1. $\nabla_G^V = |V_1 \setminus V| - |V_2 \setminus V|$
2. $\nabla_G^E = |E_1 \setminus E| - |E_2 \setminus E|$
3. $\nabla_G^{EFF} = |CE_1 \setminus CE| - |CE_2 \setminus CE|$,
   where $CE = \bigcup_g^{\forall g \in V} CE(g)$; CE_1 and CE_2 follow accordingly.

Case-I AND-decompositions.

Let G_1 be the current goal model version (refer Fig. 5.13) having an entailment conflict and G_2 be the version that is generated after applying ERA (refer Fig. 5.14). Using set cardinality, we measure the structural and semantic proximity as

1. $G_2 \nabla_G^V G_1 = 1$, the newly added subgoal.
2. $G_2 \nabla_G^E G_1 = 1$, the AND-decomposition link connecting this new subgoal.
3. $G_2 \nabla_G^{EFF} G_1 \geq 1$, the satisfaction conditions that have been added for resolving entailment conflicts.

Case-II OR-decompositions.

Let G_1 be the current goal model version (refer Fig. 5.11) having an entailment conflict and G_2 be the version that is generated by ERA (refer Fig. 5.12). We again compute the structural and semantic proximities, based on set cardinality, as follows

1. $G_2 \nabla_G^V G_1 = 2$, the newly added subgoal and a higher level goal that fuses these two subgoals
2. $G_2 \nabla_G^E G_1 = 2$, the AND-decomposition links connecting the high-level goal to the two subgoals
3. $G_2 \nabla_G^{EFF} G_1 \geq 1$, the satisfaction conditions that have been added for resolving entailment conflicts.

We can similarly evaluate the distances for the Consistency Resolution Algorithm as well. Once we evaluate ∇_G^V, ∇_G^E, and ∇_G^{EFF}, we can compute the net distance using a normalized weighted average as follows:

$$G_1 \nabla_G G_2 = \left(\omega_1 \cdot \nabla_G^V + \omega_2 \cdot \nabla_G^E + \omega_3 \cdot \nabla_G^{EFF}\right), \tag{5.10}$$

$$\text{where } \forall i, 0 \leq \omega_i \leq 1 \text{ and } \sum_{i=1}^{3} \omega_i = 1$$

It is now easy to visualize why $G_2 \nabla_G G_1$ will be positive whenever ERA or CRA is invoked by the AFSR framework and that G_1 will always remain more proximal to the original goal model than G_2, i.e., $G_1 <_G G_2$.

The distance measurement operators are defined based on the cardinality of the vertex sets, edge sets, and the semantic annotation sets. The weights for evaluating goal model proximity can also be adjusted to give priority to structural or semantic definitions. However, a greater distance need not necessarily imply a more undesired solution. A greedy approach like the one discussed here may result in obtaining a local optimum. Simulated annealing approaches or heuristic approaches may be deployed for finding out the global optimum.

5.4 Using AFSR on the i* Framework

The AFSR framework, proposed in the previous sections, has been developed for performing goal model maintenance exercises. The goal modelling constructs used for developing the AFSR framework are common to most goal modelling frameworks like KAOS, GBRAM and i*. It is a generic framework and can be easily adapted to fit any specific goal modelling framework. In this section, we illustrate how the AFSR framework can be adapted for maintenance of i* models.

The i* framework is one of the most complex State-of-the-Art goal modelling frameworks. In terms of modelling constructs, the i* framework supports inter-actor dependencies that need to be handled separately. Apart from that, task decompositions and means-end decompositions are analogous to AND-decompositions and OR-decompositions, respectively. The mechanism for semantic reconciliation also remains the same. Another characteristic of the i* framework is that it supports both hard-goals (functional requirements) and softgoals (non-functional requirements) within the same model. However, since the AFSR framework works with functional satisfaction conditions only, we ignore softgoal constructs for the time being. The following sections elaborate on how the AFSR framework can be extended to incorporate semantic reconciliation over inter-actor dependencies and how we can deploy the implementation roadmap for i* models in particular.

5.4.1 Dependency Reconciliation Operator

The AFSR framework described earlier has an underlying assumption that all leaf-level goals are primitively satisfiable. This means that the leaf-level goals do not depend on other actors for satisfying them and, hence, the *ORGMods* shown in these examples are restricted to within the actor boundary. However, this may not be the case always.

Consider the multi-actor goal model shown in Fig. 5.20. The figure illustrates an *ORGMod* that spans beyond the actor boundary of A_i and forays into the boundary

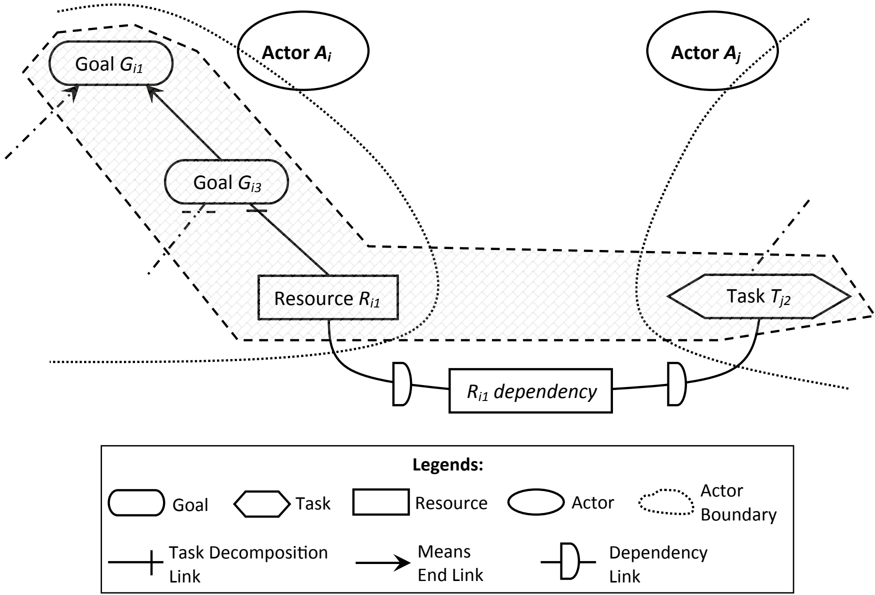


Fig. 5.20 An *ORGMod* that extends beyond the boundary of actor A_i as the resource R_{i1} depends on task T_{j2} of actor A_j

of actor A_j. The R_{i1} *dependency* captures the requirement that actor A_i needs to acquire the resource R_{i1} and depends on actor A_j to perform task T_{j2} to provide the required resource. Unlike the previous examples, $CE(R_{i1}) \neq IE(R_{i1})$. In general, we can redefine the formula for evaluating the cumulative satisfaction conditions of leaf-level goal L_a as follows:

$$CE(L_a) = DEPrec(L_a)$$

$$= \begin{cases} IE(L_a), & \text{if } L_a \text{ is independent.} \\ rec(L_a, L_b), & \text{using Eq. (5.1), if } L_a \text{ depends on } L_b. \end{cases} \tag{5.11}$$

Assuming that for any dependency, both the depender and the dependee are leaf-level goals, and do not undergo any further decompositions, we proceed to derive the cumulative satisfaction condition $CE(R_{i1})$ using the dependency semantic reconciliation operation $DEPrec()$ as follows:

$$CE(R_{i1}) = DEPrec(R_{i1}) = rec(R_{i1}, T_{j2}).$$

such that $CE(T_{j2}) \cap \neg IE(R_{i1}) = \emptyset$, i.e., $IE(R_{i1})$ and $CE(T_{j2})$ are mutually consistent. If the intersection results in a non-empty set, then the corresponding dependency gives rise to inconsistencies within the model.

In general, we may have a chain of dependencies (not a cycle or loop) for satisfying a leaf-level goal. Assuming that there are no cycles, let $\langle A_1, M_1\rangle \rightarrow \langle A_2, M_2\rangle \rightarrow \cdots \rightarrow \langle A_k, M_k\rangle$ represent a chain of transitive dependencies where $\langle A_i, M_i\rangle \rightarrow \langle A_j, M_j\rangle$ implies that a leaf-level goal M_i in actor A_i is dependent on another leaf-level goal M_j in actor A_j. The cumulative satisfaction condition for the first goal in the dependency chain, $CE(M_1)$, can be derived from the following recurrence relation, using Eq. 5.11:

$$CE(M_i) = DEPrec(M_i)$$

$$= \begin{cases} IE(M_i), & \text{if i=k.} \\ rec(M_i, DEPrec(M_{i+1})), & \forall i, 1 \leq i \leq \quad (k-1). \end{cases} \tag{5.12}$$

5.4.2 Implementation Roadmap for i*

We have already provided an implementation roadmap for the AFSR framework that maps the goal maintenance problem to a state-space search problem with the help of the six tuples $\langle S_{stt}, s_{strt}, \Sigma_{mod}, \Delta_{succ}, \Omega_{heur}, \Gamma_{test}\rangle$. We now elaborate on how these six tuples can be realized for the purpose of maintaining i* models:

i. The language we will use to populate the set α will consist of the following predicates:

  a. *consistent*(G) which states that a goal model G is consistent in the sense of the reconciliation operator.

  b. *entails*(G, k, γ) which states that goal model G k-entails goal γ.

  c. *not_entails*(G, k, γ) which states that the goal γ is strongly non-derivable from goal model G, level k onwards.

  d. *cond_entails*(G, k, γ, β) which states that a goal model G k-entails γ if and only if it entails β at level $k-1$.

  e. *cond_not_entails*(G, k, γ, β) which states that γ is strictly non-derivable from goal model G, level k onwards, whenever G entails β at level $k-1$.

ii. Each state $s_i \in S_{stt}$ represents a modified version of the original goal model such that some subset of goals within the model are guaranteed to be α-satisfiable.

iii. The start state s_{strt} represents the original goal model G.

iv. The list of primitive goal modification operators (Table 5.1) is appended with the following primitive operators for handling dependencies:

  a. `add_dep(`g_1^A`, `g_1^B`)` which adds a dependency between actors A and B making g_l^A depend on g_l^B.

  b. `rem_dep(`g_1^A`, `g_1^B`)` which removes an existing dependency between goals g_l^A(in actor A) and g_l^B(in actor B).

Table 5.2 List of compound goal modification operators

No.	Compound operator	Functionality	Sequence of primitive operations
1	ER_OR	Resolves entailment conflicts arising in OR-decompositions (as shown in Figs. 5.12 and 5.13)	break(G, G_1) break(G, G_1) OR_add(G, GT_1) OR_add(G, GT_2) join(GT_1, G_1, A) AND_add(GT_1, CT_1) join(GT_2, G_2, A) AND_add(GT_2, CT_1)
2	ER_AND	Resolves entailment conflicts arising in AND-decompositions (as shown in Figs. 5.14 and 5.15)	AND_add(G, CT_1)
3	CR_Hier	Resolves hierarchic consistency conflicts (as shown in Figs. 5.16 and 5.17)	leaf_rmv(G_1) AND_add(G, TG_1) AND_add(G, TG_2)
4	CR_Sibl	Resolves sibling consistency conflicts (as shown in Figs. 5.18 and 5.19)	leaf_rmv(G_1) AND_add(G, TG_1) or leaf_rmv(G_1) AND_add(G, TG_1)

c. leaf_dep_rmv(g_1) which removes a leaf-level goal g_l that is dependant on some other goal by removing the associated dependency as well.
Σ_{mod} comprises of the set of compound goal modification operators—ERA_AND, ERA_OR, CRA_Hier, and CRA_Sibl—that can be applied to any state to reach a new state in the state space. We can use the set of primitive operators (including the newly added dependency primitives) to represent these compound goal modification operations as sequences of primitive operations. Table 5.2 shows this in detail.

v. Δ_{succ} is the transition function that captures all the state transitions that can occur within the state space S_{stt} due to the application of compound goal modification operators in Σ_{mod}, i.e., Δ_{succ}: $S_{stt} \times \Sigma_{mod} \rightarrow S_{stt}$.

vi. For the heuristic cost estimate function Ω_{heur}, we define a weighting scheme based on structural and semantic proximity, where we assign *positive* weights to goal model refinements. These weights are proportional to the levels at which they occur such that higher level refinements have greater impact on the cost estimate. This implies that-

a. $\Omega_{act}(start,s)$ is the sum of the weights of all goal refinements that have been included so far.

b. $\Omega_{est}(s,goal)$ is an optimistic guess of the net sum of the weights that can be further added to the goal model in state s to obtain an α-satisfiable solution.

We have defined specific instants of the six tuples that are used to define the state space search problem. Since the state space search has been properly defined, we

now need to show that the heuristic cost estimate function Ω_{heur} is both *admissible* and *consistent*. This ensures that applying A* search to the state space search will find the optimal path to the goal model having minimum deviation.

Theorem 5.2 *The heuristic cost function Ω_{heur} is admissible and consistent, i.e., for any intermediate state s,*

1. $\Omega_{est}(s, s_{fin}) \leq \Omega_{act}(s, s_{fin})$ *(admissibility), and*
2. $|\ \Omega_{est}(s_1, s_{fin}) - \Omega_{est}(s_2, s_{fin})| \leq \Omega_{act}(s_1, s_2)$ *(consistency)*

Proof The proof takes the same approach as in Konstantin's thesis [1]. We consider a relaxed version of the problem where conflicts are resolved as and when they are encountered during the bottom-up reconciliation process. We can compute the estimated cost of reaching a final state from the current state by identifying all conflicts. This estimated cost is the sum of the weights assigned to all goal model refinements that must be applied to obtain an α-satisfiable goal model.

We then observe that for the non-relaxed version of the problem, requirement analysts can choose to resolve any particular conflict within the goal model subtree. This might not be at the lowest level and can, thus, give rise to new conflicts. Again, the cost estimates are based on the solutions provided by ERA and CRA—which make bare minimal changes to a given goal model configuration for resolving conflicts. Analysts can choose to deploy their own solutions for a conflict for which the actual cost can only increase. Thus, the estimated cost of reaching the final state can never be larger than the actual cost and the heuristic estimate is never underestimated. This implies that $\Omega_{est}(s, goal) \leq \Omega_{act}(s, goal)$ holds for all states. Hence, the heuristic is admissible.

The proof of consistency follows from the proof of admissibility. Let s_1 and s_2 be two intermediate states in the state space. Since admissibility holds for both these states, we have

$$\begin{aligned} \Omega_{est}(s_1, s_{fin}) &\leq \Omega_{act}(s_1, s_{fin}) \\ \Omega_{est}(s_2, s_{fin}) &\leq \Omega_{act}(s_2, s_{fin}) \end{aligned} \tag{5.13}$$

If we take an absolute difference on both sides we have

$$|\Omega_{est}(s_1, s_{fin}) - \Omega_{est}(s_2, s_{fin})| \leq |\Omega_{act}(s_1, s_{fin}) - \Omega_{act}(s_2, s_{fin})| \tag{5.14}$$

But since $\Omega_{act}(s, goal)$ is a monotonically increasing function, we can conclude that

$$|\Omega_{act}(s_1, goal) - \Omega_{act}(s_2, goal)| = \Omega_{act}(s_1, s_2) \tag{5.15}$$

Replacing Eq. 5.15 in Eq. 5.14, we get the consistency criteria

$$|\Omega_{est}(s_1, goal) - \Omega_{est}(s_2, goal)| \leq \Omega_{act}(s_1, s_2) \tag{5.16}$$

Hence, proved.

5.5 Experimental Evaluation

In this section, we perform an experimental evaluation of our approach to show that our proposed methodology can be deployed in practical settings for maintenance and support. We achieve this by showing that a prototype of the AFSR framework can be scaled up to large and complex goal models occurring in real-life scenarios.

We first elaborate on the performance indicators and drivers that will serve as parameters for measuring scalability in Sect. 5.5.1. This is followed by elaboration of experimental preliminaries (Sect. 5.5.2) including the process for random generation of semantically annotated goal models. We also provide a brief description of an uninformed version of A* search. The performance metrics of this uninformed search will serve as the baseline for measuring the performance of A* search. Finally, we present all the experimental results with suitable graphs in Sect. 5.5.3.

5.5.1 Indicators and Drivers

Indicators. The most important indicator is *execution time* which signifies the time it takes to compute a valid outcome of the SRA algorithm for a given annotated goal model. The execution time is directly influenced by the number of calls to the ERA and (or) CRA operators to create an α-satisfiable goal model. Another important indicator is the number of states that are created (or visited) during the A* search. For uninformed search, this indicator will face an exponential growth.

Drivers. The most significant driver is the *size of the goal model* that is being fed as input to the SRA algorithm. The size is measured by the number of levels in the goal model (i.e., the depth). As the depth of a goal model increases, the cost of applying the ERA and CRA operators at the higher levels also increases. We denote this driver as $level$.

Another important driver is the number of goals in the goal model and the corresponding number of satisfaction conditions being fed into the SRA algorithm. However, there is no strict correspondence between the two as goals are annotated arbitrarily. This dilemma is resolved by considering the maximum number of satisfaction conditions that can be used to annotate the goals. We denote this driver as *eff*.

5.5.2 Experimental Preliminaries

In this section, we explain random generation of goal models, random generation of satisfaction conditions for goal models, and the formulation of an uninformed search as a modification of the A* search.

5.5.2.1 Random Generation of Goal Models

Goal models were randomly generated as we did not have access to complex real-life goal models. This is reasonable for the purpose of evaluating scalability as it allows us to create a large number of goal models of a chosen size and complexity. All parameters of the random generation procedure are denoted by n_x, where x identifies the parameter.

We first generate n_r number of root goals where each root goal represents the root of a goal decomposition tree. For each root goal, the random generation procedure creates a refinement consisting of at least one- and at most n_s number of subgoals. The type of refinement (OR-/AND-decomposition) is also decided randomly. The process is performed recursively for the newly created subgoals until the goal model being generated reaches a depth of n_{levels}.

5.5.2.2 Random Generation of Satisfaction Conditions

A goal model is annotated with satisfaction conditions by random generation of the set of *immediate satisfaction conditions* (IE) for each goal within the model. A set *Eff* of n_{eff} satisfaction conditions is generated. An immediate satisfaction condition (*IE*) is defined as a set of satisfaction conditions—either in their original or negated form. An immediate satisfaction condition set *IE* of size $n_{effsize}$ is generated by randomly selecting a satisfaction condition from *Eff*, $n_{effsize}$ times, and adding it with an equal probability as a negative- or positive variable to *IE*. This step is repeated for all the goals in the generated goal model.

5.5.2.3 Modification of A* Search

In order to provide a baseline for comparing the performance of A* search, we create a modified version of A* that does not use any domain-specific knowledge that is encoded within the heuristic function. In the semantic reconciliation process using A*, we only flag conflicts (entailment or consistency) and create a cost matrix that provides an estimate of resolving these conflicts that exist at different levels of the given goal model. Based on this cost matrix, we choose to perform the most cost-efficient semantic reconciliation using the `ERA` or `CRA` operators. For performing an uninformed search, we invoke the `ERA` or `CRA` operators by selecting a conflict randomly from the cost matrix.

5.5.3 Process and Results

The experimental results are presented and discussed in this section, but first the experimental process and set-up are described in more detail.

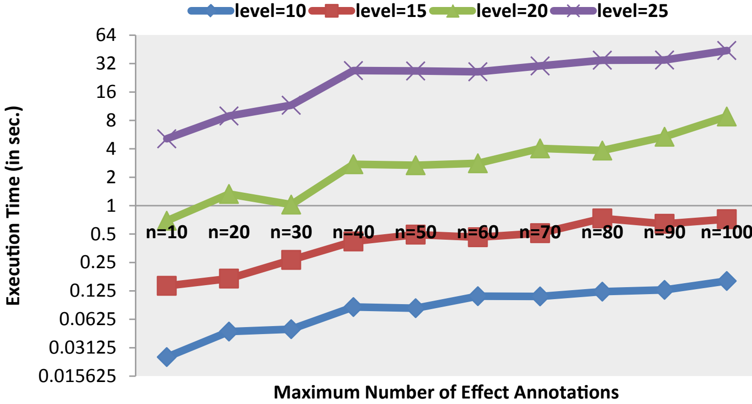


Fig. 5.21 Variation of execution time for random generation of annotated goal models

5.5.3.1 Experimental Process and Set-Up

We have performed the experiments in two parts.

In the first part, we have tried to evaluate the variation of execution time for random generation of semantically annotated goal models. This was done by fixing the values of $n_r = 1$ and $n_s = 2$. Random goal models were generated having depth $n_{levels} \in \{10, 15, 20, 25\}$. For satisfaction condition of the goals, we chose $n_{eff} \in \{10, 20, \ldots, 100\}$ and $n_{effsize}$ was randomly chosen to be bounded by n_{eff}, i.e., $n_{effsize} \leq n_{eff}$. To obtain an average behaviour of the random generation process, we generated 50 sets of data for each instantiation of the drivers.

In the second part of the experiment, we perform semantic reconciliation for randomly generated annotated goal models. As in the previous experiment, n_r and n_s are set to values 1 and 2 respectively. Random goal models are generated having depth $n_{levels} \in \{5, 6, 7, 8\}$ and $n_{eff} \in \{10, 20, 30\}$. For each driver instantiation, we performed 50 runs of our experiment to obtain mean values. In this experiment, we measured the number of times the ERA and CRA operators were invoked to find an α-consistent goal model as indicator. For each generated model, we ran the A* as well as uninformed search. In order to complete the experiments in feasible time, we aborted each execution of uninformed search that lasted longer than 10 min. The experiments were conducted on an Intel Core i5-3330 CPU, operating at 3.00 GHz and having 4GB of RAM.

5.5.3.2 Experimental Results

We first present the results for random generation of semantically annotated goal models. Figure 5.21 plots the execution time (in seconds) against the maximum number of satisfaction conditions for each goal. Data sets have been obtained for

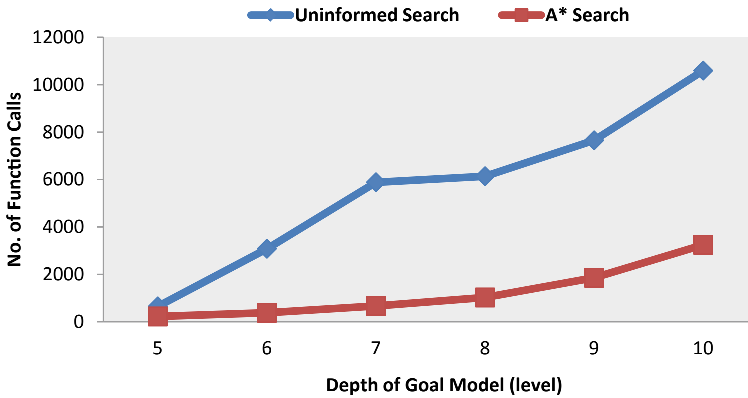


Fig. 5.22 Number of calls to the semantic reconciliation operators (ERA and CRA) for randomly generated goal models with $n_{eff} = 10$

goal models having depths 10, 15, 20 and 25. The y-axis has been converted to the logarithmic scale $\log_2$. The four curves are almost linear in nature. Linear curves on a logarithmic scale represent exponential functions. Thus, for all the four different data sets, the execution time increases exponentially as the maximum number of satisfaction conditions increases. However, the exponential growth for $level = 10$ (represented by the blue curve) is much less as compared to the exponential growth for $level = 25$ (represented by the purple curve); although they appear to be somewhat parallel on the logarithmic plot. The execution time for $level = 10$ varies from 0.0249 seconds to 0.1592 seconds, whereas, for $level = 25$, it varies from 5.1093 seconds to 44.04 seconds. Thus, the higher the position of the linear curve on the y-axis, the more rapid exponential growth of the execution time occurs.

Next we present the results for our semantic reconciliation framework AFSR. Figure 5.22 shows the average number of calls to the resolution operators ERA and CRA when the depth of the goal models varies from 5 to 10 (for both A* and Uninformed search). The maximum number of satisfaction conditions is kept fixed at $n_{eff} = 10$. As shown in the graph, A* drastically reduces the number of function calls as compared to uninformed search. This is quite logical. In uninformed search, we perform semantic reconciliation randomly which may give rise to new conflicts in the lower levels. Thus, the lower level goals get modified multiple times, repeatedly, as the SRA process progresses to the higher levels. On the other hand, A* uses the heuristic cost function and targets to reconcile the least cost conflict at the lowest level of the goal model. This eliminates the repeated modification of goals in the lower levels of the goal model. Thus, the number of calls to the reconciliation operators is also greatly reduced.

Finally, we present our results on performance evaluation when all drivers are varied simultaneously. Figure 5.23a, b show three-dimensional graphs where we measure the number of times the semantic reconciliation operators (ERA and CRA) are

invoked when we vary both the depth of the goal models (*level*) as well as the maximum number of satisfaction condition (n_{eff}).

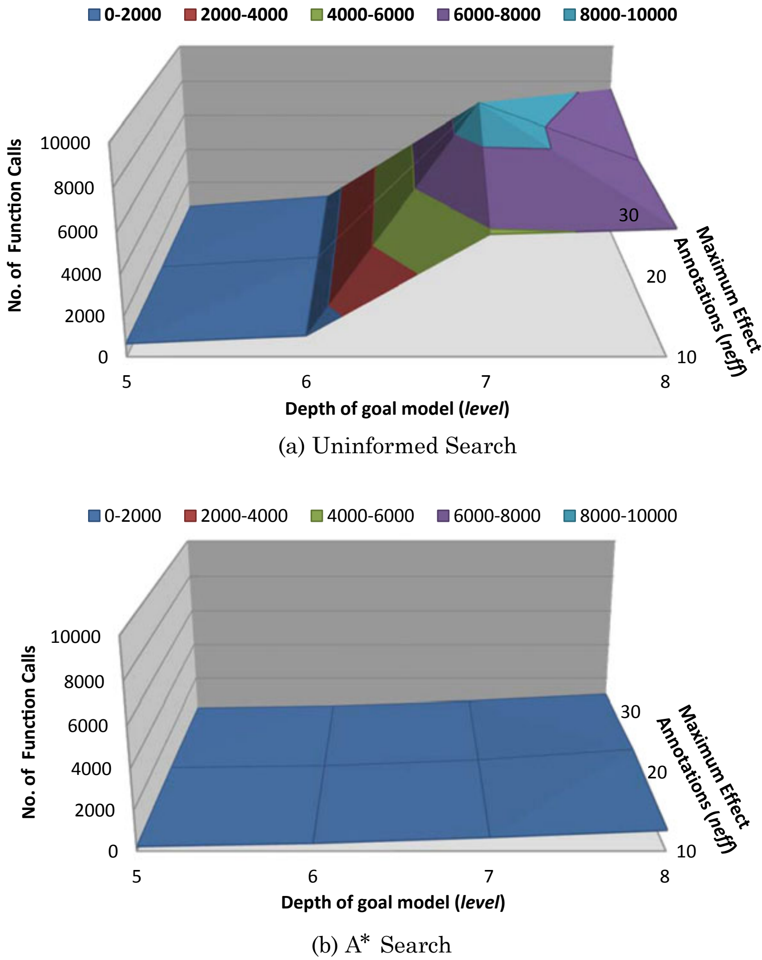
As observed in Fig. 5.23a, variations in *level* and n_{eff} have a significant impact on the number of operator invocations (and, hence, the execution time) during Uninformed search. A more careful observation shows that an increase in *level* has a much greater impact on the number of calls than an increase in n_{eff}. This can be explained as follows. First, if we increase n_{eff} for given goal model depth, it only requires the AFSR framework to handle greater number of satisfaction conditions simultaneously. This may demand higher space complexity, but the number of calls to ERA and CRA does not change significantly. On the other hand, if we increase the number of levels in the goal model, then the number of goals increases exponentially and so does the number of required semantic reconciliation (assuming even distribution of entailment and consistency conflicts during the random generation process). Thus, there is an exponential growth in the number of invocations of the semantic reconciliation operators as the number of levels in the goal model is increased.

Figure 5.23b shows that variations in *level* and n_{eff} only slightly impact the number of calls to the semantic reconciliation operators during A* search. The number of calls increases only slightly with increase in the goal model depth. This is due to the greater number of subgoals that need to be reconciled when higher level conflicts are resolved with the help of the heuristic cost matrix. The graphs shown in Fig. 5.23 are encouraging as they suggest relatively stable execution times for randomly generated goal models, regardless of the number of conflicts. However, execution time is not guaranteed to be constant for goal models of a particular size as it depends on the number of entailment or consistency conflicts that need to be reconciled.

5.6 Conclusion

In this chapter, we propose AFSR—a semi-automated framework for reconciliation of satisfaction conditions over annotated goal models. The satisfaction condition annotation and reconciliation process help requirements analysts by offering semantic descriptions of goal models. The proposed AFSR framework has two major components—a *semantic reconciliation* module and a *conflict resolution* module. The *semantic reconciliation* module reconciles the individual satisfaction conditions of goals in the context of the entire goal model. It also raises entailment and consistency conflicts that are detected during the semantic reconciliation process. The *conflict resolution* module resolves these conflicts by using model refactoring and presents analysts with possible *conflict-free* alternatives. In this chapter, we have developed the framework using generic goal model constructs and can be extended to any framework like KAOS, GBRAM, or i* (as already shown).

The proposed work may be extended to make it compatible across multiple goal modelling environments. We have also provided an implementation roadmap where we can map the goal model maintenance problem to the state space search problem and shown how A* search can be used to find a conflict-free solution having minimal



(a) Uninformed Search

(b) A* Search

Fig. 5.23 Number of calls to the effect reconciliation operators (ERA and CRA) for varying values of the drivers *level* and n_{eff}

deviation from the original goal model. We have also developed a prototype for the AFSR framework and measured how its performance scales with the size and complexity of the annotated goal models. Applying A* search over uninformed search provides significant benefits in terms of execution time and, hence, makes the AFSR framework scalable for real-life goal models having greater complexity.

References

1. Hoesch-Klohe K, A framework to support the maintenance of formal goal models, PhD thesis University of Wollongong, Wollongong, Australia
2. Hinge K, Ghose A, Koliadis G (2009) Process SEER: a tool for semantic effect annotation of business process models. In: Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing (EDOC), pp 54–63. https://doi.org/10.1109/EDOC.2009.24
3. Ghose A, Koliadis G (2008) PCTk: a toolkit for managing business process compliance. In: Proceedings of the 2nd international workshop on juris-informatics (JURISIN'08)
4. Raut M, Singh A (2004) Prime implicates of first order formulas. Int J Comput Sci Appl 1(1):1–11
5. Ghose A, Koliadis G (2007) Auditing business process compliance. In: Proceedings of the 5th international conference on service-oriented computing (ICSOC), pp 169–180. https://doi.org/10.1007/978-3-540-74974-5_14

Chapter 6
Conclusion and Future Work

The works presented as part of this book are novel and yet limited in terms of their applicability to real-world enterprises. There is ample scope for further exploration of the problems identified as part of this research. The algorithms and results presented in this book can be further modified and adapted for better application to enterprises. This chapter is divided into two sections. Section 6.1 presents a brief summary of our research contributions in this book. Section 6.2 presents some exciting future research directions that can be explored to benefit the GORE community.

6.1 Summary of the Work

In this book, we present a collection of novel solutions that aims to improve the state-of-the-art as far as enterprise modelling and requirements analysis is considered in goal-oriented requirements engineering. We address an enterprise modelling scenario that had not been considered by the community previously. We highlight the importance of goal modelling in enterprise hierarchies and particularly underline the importance of an ontology integration framework for such goal model hierarchies (also referred to as requirement refinement hierarchies). We present one such framework for integrating the ontologies between adjacent level goal models and measuring the degree of correlation that exists between them. We also establish the fact that goal model hierarchies are not merely a hypothetical concept and they really manifest themselves in real world event logs. The relevance of this research stems from the fact that we could mine adjacent and non-adjacent hierarchic structures from real-world data.

Apart from enterprise modelling, we have also contributed to the GORE community by enhancing the existing state-of-the-art in terms of requirements analysis. We have identified that very limited research had been done to perform model checking on goal models. We have proposed a new heuristic called the *Semantic Implosion Algorithm* and simulated it to compare its performance with an existing heuristic that

© Springer Nature Singapore Pte Ltd. 2020

N. Deb and N. Chaki, *Business Standard Compliance and Requirements Validation Using Goal Models*, Services and Business Process Reengineering,
https://doi.org/10.1007/978-981-15-2501-8_6

was proposed by Fuxman [1]. The new heuristic has been shown to outperform the existing heuristic by a factor of almost 10^{17}. Thus, our proposed solution is much more efficient and scalable when it comes to performing model checks. We have also implemented our proposed algorithm by developing a tool, called i*ToNuSMV. The tool accepts goal models in the textual GRL (tGRL) notation and temporal properties in CTL. The underlying model checker that has been integrated into our tool is NuSMV. Analysts can now feed goal models into the i*ToNuSMV tool and check them against temporal compliance rules. NuSMV generates counter-examples whenever a CTL property is not satisfied by the goal model.

Finally, we underline the importance of going beyond the structural features and orderings, and analysing the semantics of goal model configurations. The AFSR framework proposed therein enables the modeller to annotate goals with their associated semantics. The framework has a semantic reconciliation machinery that can evaluate the semantics of any goal as obtained from its subgoals. These derived and intended semantics can then be compared to perform semantic analyses like entailment and consistency checks. AFSR does not stop here; it goes beyond conflict detection by suggesting re-factored solutions that are conflict-free. This framework can be deployed for real-world goal model maintenance and their adoption in evolving requirement settings. However, exploring the entire space of goal model configurations for identifying the optimal solution manually seems to be quite impractical and erroneous. Human effort often leads to suboptimal solutions. We have shown how this situation can be tackled by mapping the goal model maintenance problem to the state space search problem. Establishing the admissibility and consistency of our heuristic path cost function has allowed us to deploy A* search over the space of goal model configurations, thereby, guaranteeing the optimal solution.

6.2 Future Research Directions

In this section, we try to shed some light on the future research directions emanating from the works presented in this book. A greater insight into the impact of enterprise hierarchies on goal modelling techniques can be derived from the work on goal model hierarchies. We observe from the data mining exercise on real-world data that employees within an organization need not necessarily follow the structure of the hierarchy. We have mined non-adjacent hierarchic correlations from the data as well. The proposed framework for requirement refinement hierarchies works with adjacent level hierarchies only. This framework can be extended to non-adjacent levels as well, thereby, developing a system to measure the correlation of the entire requirement refinement hierarchy. The works on requirements analysis can be extended as discussed in the following sections.

6.2.1 Extracting Business Compliant Finite State Models

The i*ToNuSMV tool is evolving quite rapidly. Version 2.02 of the i*ToNuSMV tool supports multi-actor goal models having inter-actor dependencies. It also supports model checking with CTL constraints. However, we have been working on a major release that will derive constrained finite state machines from a goal model. This implies that instead of feeding a goal model as input and then checking a temporal constraint on the derived FSM, we will provide the constraint along with the goal model as input and the derived FSM will already satisfy the given constraint.

6.2.1.1 Assumptions

The different types of CTL constraints have been studied in detail and this paper works with a finite subset of such constraints in the framework. The primary goal is to generate a compliant finite state model by pruning transitions from the finite state model generated by i*ToNuSMV ver2.02. The proposed guidelines have the following four assumptions:

A-1 Since FSMs are derived for fulfilment of goals, the framework works with only `AG` and `EG` temporal operators for the violation of goal fulfilment. Example:`AG(V109!=FU).`

A-2 Two CTL predicates can be connected through Boolean connectives like AND and OR. This framework allows the user to define only two predicates at a time and connect them by the`AND` or `OR` operator. Example: `AG(V109!=FU AND V102!=FU).`

A-3 Another type of CTL constraint that is addressed is implication ($\rightarrow$). Any two constraint can have implication between them. The implication operator has been restricted to only single level of nesting. Example: `AG(V101=CNF` $\rightarrow$ `AF(V102=FU AND V103!=FU)).`

A-4 The goal tree level for an actor has been assumed to be 3 to reduce the problem complexity.

6.2.1.2 CTL Properties Handled

This section briefly explains each of the CTL constraints that were addressed in [2] and how the corresponding finite state models are derived.

1. **EG(V#! = FU) for AND-decomposition**. These types of properties are safety properties that prevent something bad from happening. Ensuring this property on a goal with AND-decomposition requires the pruning of `CNF` $\rightarrow$ `FU` transitions for some subset of the child nodes.

2. **EG(V#! = FU) for OR − decomposition**. The same type of safety property on a goal with OR-decomposition has different consequences. Ensuring such a property requires the pruning of `CNF` → `FU` transitions for all the child nodes.
3. **EG(V#! = FU AND V#! = FU)**. CTL properties which have multiple CTL predicates connected with boolean AND connectives can be ensured by satisfying each predicate separately. The solution space is a Cartesian product of models generated from each CTL predicate—denoted by $M \times N$.
4. **EG(V#! = FU OR V#! = FU)**. CTL properties having multiple CTL predicates connected with boolean OR connectives can be ensured by satisfying either of the predicates or both. The solution space is much larger and denoted by $M + N + (M \times N)$.
5. **AG(V#! = FU → V#! = FU)**. These type of CTL properties (defined with the implication operator →) specify an ordering over the fulfilment of goals. Thus, all those invalid states need to be pruned from the FSM that violate this property. State transitions to or from these invalid states are correspondingly removed.
6. **EG(V#! = FU) for AND-OR-decompositions**. Ensuring such safety properties for multilevel goal models with OR-decompositions nested under an AND-decomposition requires the pruning of `CNF` → `FU` transitions for all OR-children. This needs to be done for any subset of the AND-children of the root node.
7. **EG(V#! = FU) for OR-AND-decomposition**. If the root goal is OR-decomposed followed by each OR-child undergoing an AND-decomposition, then these types of CTL properties can be ensured by pruning `CNF` → `FU` transitions for any subset of AND-children for each of the OR-child of the root goal.

The above seven types of CTL properties have been addressed in the newly proposed version 3.0 of the $i^*ToNuSMV$ framework. For a more detailed understanding of how each of these CTL property classes is ensured within a goal model, readers can refer to [2]. Figure 6.1 demonstrates the workflow of the $i^*ToNuSMV$3.0 framework.

6.2.1.3 Demonstration with Case Study

In this section, the working of the $i^*ToNuSMV$3.0 deployment interface is demonstrated with the help of a simple real-life case study. Figure 6.2 shows a simple goal model that captures the requirements for `Access Locker`. It requires two tasks to be performed—`VerifyCodeTrue` verifies whether the user access code entered is true and `GiveAccess` finally gives the access of the locker to the user provided the code entered is true. An intuitive CTL property associated with this goal model is also shown in the figure. `AG(V103!=FU → V104!=FU)` implies that the task `GiveAccess` cannot be performed until the task `VerifyCodeTrue` is successfully completed.

The $i^*ToNuSMV$2.02 tool, which implements the Semantic Implosion Algorithm (SIA), generates a finite state model irrespective of the CTL property associ-

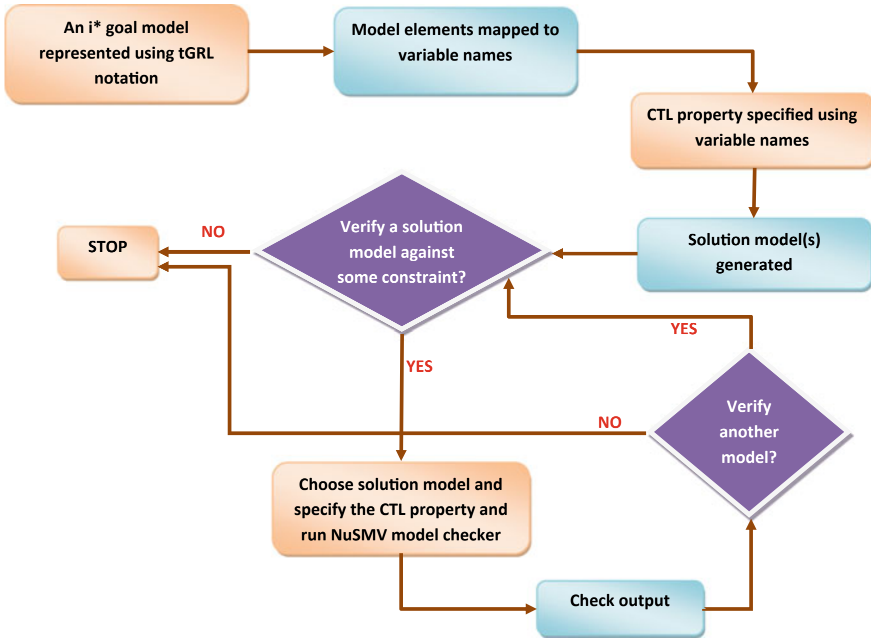
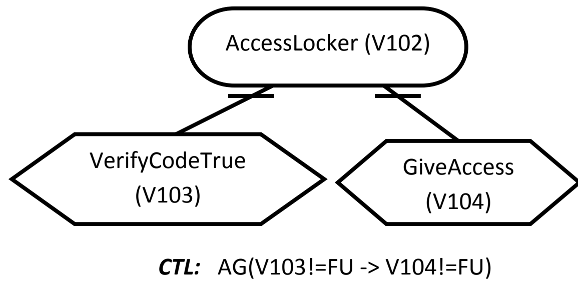


Fig. 6.1 The workflow of the *i*ToNuSMV3.0* deployment framework

Fig. 6.2 A simple goal model for accessing a locker



ated with the goal model. Since the goal model in Fig. 6.2 has a two child AND-decomposition, the corresponding FSM has a 2-dimensional lattice structure for capturing all possible execution sequences to fulfil the root goal. The derived FSM is shown in Fig. 6.3.

The research guidelines proposed in [2] have been implemented in *i*ToNuSMV 3.0*. It is an extension of the Semantic Implosion Algorithm that takes the finite state model generated by SIA and prunes those transitions which violate the given CTL property. The pruned finite state model for the given goal model and CTL property (refer to Fig. 6.2) is shown in Fig. 6.4.

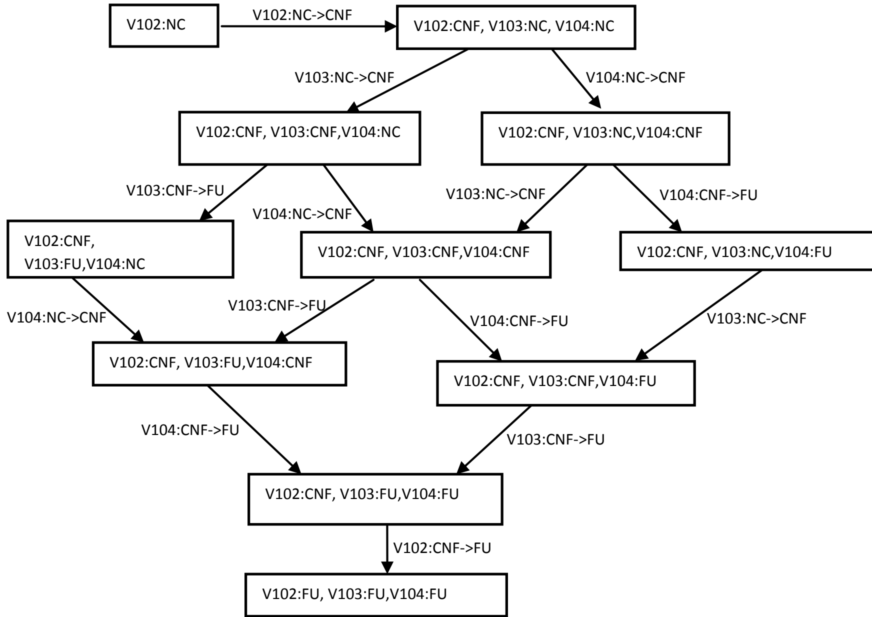


Fig. 6.3 FSM generated by *i**ToNuSMV 2.02*

6.2.1.4 URL

The *i**ToNuSMV 3.0* framework can be downloaded from the following link: https://github.com/istarToNuSMV/i-ToNuSMV3.0. The User Manual and use case examples have been shown on the webpage.

6.2.1.5 Experimental Results

In this section, some experimental results have been documented that were obtained after performing extensive simulations with the existing (version 2.02) and newly proposed (version 3.0) versions of the $i^*ToNuSMV$ framework. Arbitrary goal models were designed with varying complexity in terms of the number of actors, the number of goals, the number of AND/OR-decompositions and the complexity of the associated CTL constraints. The simulations did not bring out any anomalous behaviour. Data were collected with respect to the number of transitions in the final output FSM and the execution time.

The bar chart of Fig. 6.5 shows a comparative analysis between SIA (implemented in version 2.02) and Complaint-SIA (implemented in version 3.0). *i**ToNuSMV 2.02* does not generate a compliant FSM like *i**ToNuSMV 3.0*. Thus, the FSM generated by version 2.02 includes all possible execution sequences between sets of states. The complaint-FSM generated by version 3.0 will have fewer number of transitions as

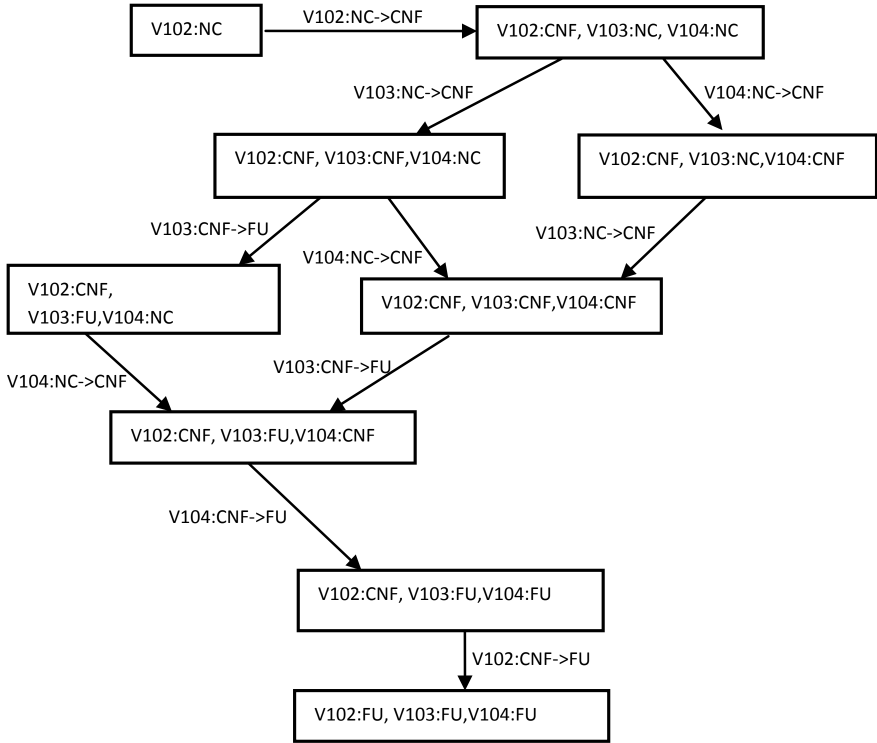


Fig. 6.4 FSM generated by *i*ToNuSMV 3.0*

all CTL properties used in these simulations, impose some sort of ordering between events. This results in the final FSM having only a subset of the transitions included by SIA. The degree (or %) of reduction in state space is dependent on several factors rather than only one.

The line plot shown in Fig. 6.6 compares the execution time of SIA and Complaint-SIA—both measured in milliseconds. With the same set of simulation parameters, it is observed that *i*ToNuSMV 3.0* takes much more time than *i*ToNuSMV 2.02* to generate the finite state models. This is also quite logical as version 3.0 implements some additional checks and tasks after SIA is executed (as in version 2.02). Basically, version 3.0 takes the FSM generated by SIA and individually scans and prunes transitions to satisfy the given CTL property. Also, as discussed in [2], there may be multiple strategies for pruning different subsets of transitions in order to satisfy the CTL property. *i*ToNuSMV 3.0* executes each such strategy and generates a unique finite state model (pruned and compliant) for each of these strategies. This is the reason why version 3.0 takes much longer to reach completion.

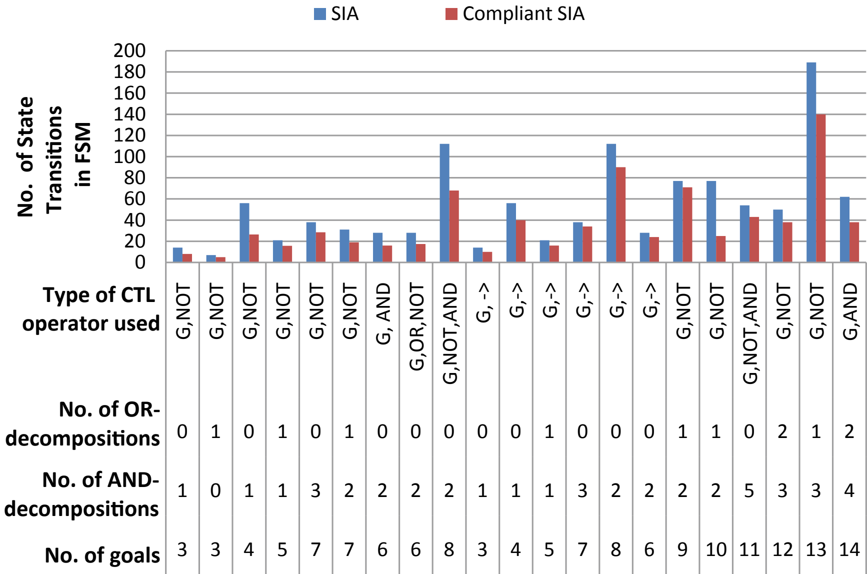


Fig. 6.5 Number of state transitions in the final FSM

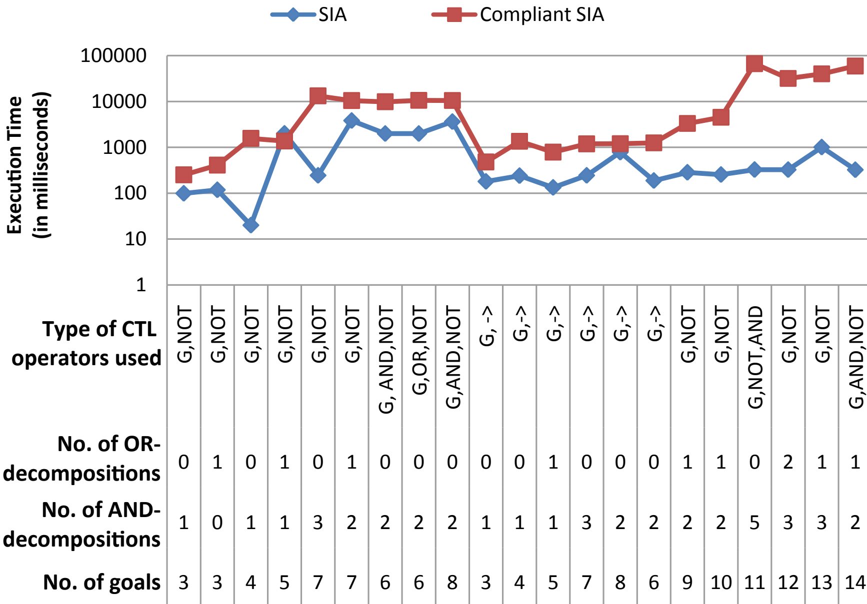


Fig. 6.6 Execution time for deriving the final FSMs

6.2.1.6 Conclusion and Future Work

In [2] the authors only presented some use cases to demonstrate how CTL compliance can be ensured in finite state models. They also documented an algorithm for the process. However, a proper deployment framework implementing the guidelines was missing. This paper builds on the guidelines proposed in [2] and presents a proper deployment interface for i*ToNuSMV 3.0. It provides the URL for downloading and installing the framework and the different features supported by the interface (with the help of a case study). It also measures and compares the performance of the newly built version with the existing version of the i*ToNuSMV tool (also see Table 6.1).

This work presents a tool to generate finite state models from goal models that already satisfy some given CTL constraint. Finite state models can be more readily transformed into code. Thus, this research takes an important step towards the development of business compliant applications directly from goal models. Most business compliance rules have some sort of temporal ordering over events and can be represented with temporal logics efficiently. However, the proposed solution has several assumptions which needs to be relaxed for making the framework more complete.

One of the more important limitations of the proposed solution is that only one temporal property (in CTL) can be specified along with the goal model specification. Future versions of the $i^*ToNuSMV$ framework will aim to allow users to specify multiple CTL properties over a single goal model specification. Another limitation of the new version is the extra processing time that is required. The additional pruning mechanism requires extensive checking of the finite state model generated by SIA. Currently, research efforts are being channelized to develop an efficient version of the Semantic Implosion Algorithm that will generate compliant FSMs in a more efficient manner.

Table 6.1 Feature comparison between verions 2.02 and 3.0

Features	i^*ToNuSMV 2.02	i^*ToNuSMV 3.0
Input specification	i^*goal model defined using tGRL	i^*goal model defined using tGRL and a CTL property
Number of FSM generated	1	1 or more than one
Compliant FSM	FSM may or may not be compliant to any temporal property	FSM compliant with a given CTL property
Solution space	Comparatively large	Reduced solution space
Number of NuSMV input	1	One for each of the FSM generated
Verification	NuSMV model checker verifies property on single finite state model	NuSMV model checker can separately verify each of the solution models

6.2.2 *The CARGo Tool*

The AFSR framework has been presented along with an implementation roadmap that uses A* search. This research direction has several avenues that can be further explored to make it more applicable to enterprises. For instance, we have worked with functional semantic annotations only. Research can be directed to incorporate non-functional semantics associated with softgoals. Non-functional semantic analyses can enrich the mechanism for choosing between multiple strategies of goal satisfaction. Also, the most imminent research scope is to build a proper tool interface that implements the AFSR framework.

The *CARGo* prototype [3] is built on the AFSR framework. It makes the use of a data structure, as illustrated in the following section, for representing and modifying goal models. Algorithm 1 shows the main procedure of the prototype and how this data structure is used.

6.2.2.1 Semantically Annotated i* Networks (SAi* Nets)

SAi* Nets are a non-linear data structure representation of goal models that have been developed for the *CARGo* tool prototype. It is similar to an adjacency list (a list of linked lists) where each list captures the strategic rational model of a specific actor. Every list is headed by an *actor_node* which specifies the particular actor. Each goal model element within the actor's goal tree is represented using *tree_nodes* that have the node structure shown in Fig. 6.7. A sample abstracted SAi* Net representation is shown in Fig. 6.8. All computations and modifications proposed by the AFSR framework, for identification and removal of annotation conflicts, is implemented on the SAi* Net. The final conflict-free SAi* Nets are translated to textual goal model descriptions for end-user readability. Each *tree_node* has the following fields:

- *val*: An integer used to identify each goal model element uniquely.
- *str*: Name of the element.
- *type*: Integer values are used to identify decomposition type of the *tree_node*—0 for OR-decompositions, 1 for AND-decompositions and 2 for leaf nodes.

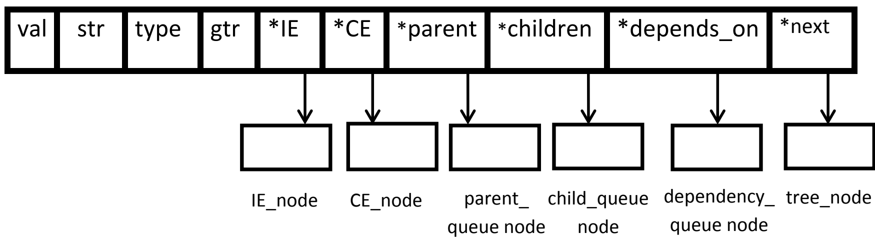


Fig. 6.7 *tree_node* structure

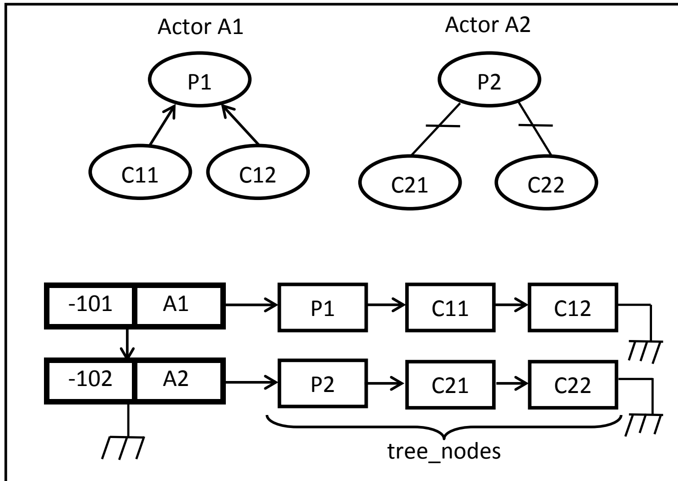


Fig. 6.8 Abstracted view of an example SAi* Net

- *gtr*: Integers used to identify goal, task or resource.
- **IE*: List of immediate annotations as first-order logic predicates.
- **CE*: List of cumulative annotations as evaluated by SRA.
- **parent*: A pointer to its parent queue. For root nodes, this field is NULL.
- **children*: A pointer to its child list. For leaf nodes, this list is empty.
- **depends_on*: A pointer to the list of dependencies associated with that *tree_node*. For an independent node, this list is empty.
- **next*: A pointer to the next *tree_node* in the actor's goal model.

6.2.2.2 Platform Used

The back end code is generated in the C language. The front end design is developed in JAVA.

6.2.2.3 URL of the CARGo Prototype

The *CARGo* tool can be freely downloaded from the following URL: https://github.com/CARGoTool/CARGoV1.0.

Algorithm 1 CARGo_tool

Input: i* model with immediate annotations in textual format.
Output: Conflict-free i* model variant in textual format.
Data Structure: SAi* network and a *conflict_list*.

```
1: procedure MAIN
2:     Build SAi* network from given input i* model
3:     Compute CE of each node by traversing SAi* network
        in bottom-up approach
4:     Traverse SAi* network either in top-down or in
        bottom-up approach according to given user choice
        and generate the conflict_list.
5:     do
6:         Extract node from the conflict_list.
7:         Perform ERA or CRA as per the type of conflict
8:         Apply SRA to update the SAi* network and con-
            flict list.
9:     while conflict_list is not empty
10:    Generate i* model representation of the conflict free
        SAi* network in textual format
11: end procedure
```

6.2.2.4 Benefits of the CARGo Tool

The annotation of goal model artefacts within a goal model is not one of the major contributions of this tool. It is somewhat similar to the annotation mechanism supported by jUCMNav for GRL. In fact, we work with tGRL goal models. The main benefit of the tool is in the domain of goal model maintenance in changing business environments. The tool helps with the adaptation of goal models when business requirements change. Changing requirements cause a change in the relative contexts of the goal model artefacts. The *CARGo* tool identifies the conflicts arising out of these changes in contexts. Conflict resolution is performed by refactoring the goal model and creating a goal model variant that is conflict-free. Existing goal modelling analysis techniques can be applied to all goal model variants as well. Thus, the *CARGo* tool helps in the evolution of goal models in changing business environments.

6.2.2.5 Conclusion

The *CARGo* tool is *sound* as the output is always a conflict-free goal model. It is also partially *complete* with the exception of softgoals and softgoal contexts. The number of iterations for conflict resolution is nondeterministic as it depends on the number and type of conflicts observed in the initial goal model.

6.2.3 Building Mobile Applications from Goal Model Specifications

This section elaborates the generalized framework of the GRL2APK tool. Figure 6.9 illustrates the framework and Sect. 6.2.3.1 explains the role of the individual components. The framework is generic and does not depend on any specific technology; it is, however, limited to only structural NFRs. The following sections elaborate on the workflow of the GRL2APK tool and a real-life use case illustrating how the tool can be used to generate Android applications.

6.2.3.1 Architecture of the New Framework

The overall architecture of the GRL2APK framework is depicted in Fig. 6.9. In this framework, we propose an integration of different components and services that allows enterprise architects to build applications while having the flexibility to choose the operationalizations of the specified NFRs. This framework has four underlying assumptions:

1. The code modules of the functional requirements are stored in a cloud repository.
2. The desired NFRs have to be specified by the enterprise architects within the goal model description. System developers only need to choose the "*operationalization*" of the desired NFRs.
3. NFR catalogs for high-level NFRs have been developed by specialized requirement engineers and stored in a cloud repository for reuse.
4. The functional codes for implementing the operationalizations of *structural* NFRs have also been stored in the code repository.

The proposed GRL2APK framework has the following components:

- **Goal Model Specifier**: The first module provides an interface to the enterprise architects to design the goal model based on the end-user requirements. Enterprise architects can use any goal requirements language to capture such models. We suggest the `Extended tGRL` (or, XtGRL) language.
- **Component Extractor**: The goal model is then passed through this module to go through the `XtGRL` grammar artefacts as specified by the enterprise architects. The module scans through the input model and extracts all the high-level NFRs and functional goal decompositions that have been specified.
- **NFR Fetcher**: After extraction of the specified high-level NFRs, the *NFR Fetcher* module is invoked. This module fetches the NFR catalogs of all those high-level NFRs that have been identified in the previous phase. It is based on assumption (3). It is a 1-to-1 mapping that allows this module to fetch the required NFR catalogs.
- **NFR Repository**: This repository stores two types of information—the *NFR catalogs* for high-level NFRs and a *NFR conflict database* that identifies conflicts across NFRs and their operationalizations. Such a repository can be stored on a local server or in some cloud repository like Google Firebase or Amazon AWS. NFR

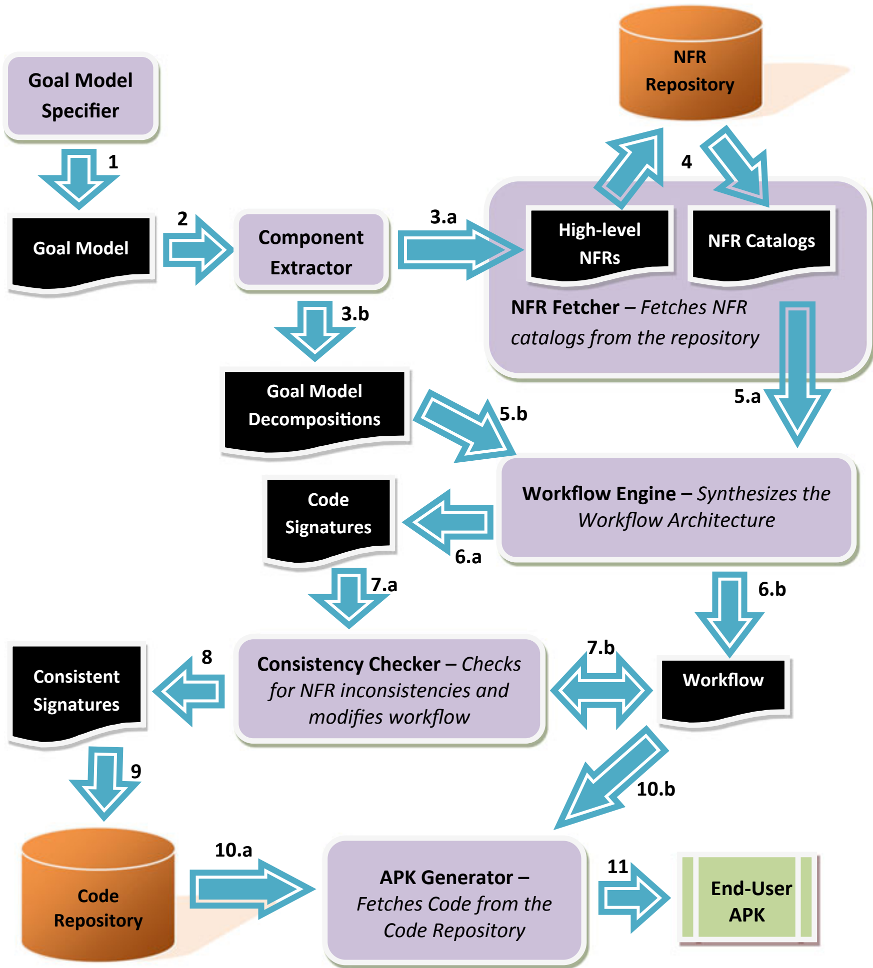


Fig. 6.9 The proposed framework for app orchestration from goal models using selective composition of NFRs

catalogs are static in nature as they only capture the decomposition of high-level NFRs into low-level operationalizations. The NFR conflict database is dynamic and needs to be updated based on available NFR operationalizations and also on the particular application vertical where the framework is being deployed.

- **Workflow Engine**: The goal decompositions (from Step-2) and the NFR Catalogs (from Step-3) are fed into the *Workflow Engine* that provides an interface where the system developer has to choose between different operationalizations and code signatures for both functional and non-functional requirements.
- **Consistency Checker**: The developer may choose operationalization strategies that conflict with other high-level NFRs. This module alerts the developer of the

existence of such conflicts. The developer, however, has the choice to prioritize a particular operationalization, thereby, ascertaining the satisfaction of the NFR. The Workflow file is correspondingly updated and the consistent set of Code Signatures are also identified.

- **APK Generator**: This is the final phase of the framework that takes the consistent set of Code Signatures and the Workflow to generate an APK file for end-users. By the time the framework reaches this phase, all operationalizations of specified high-level NFRs have been decided and all conflicts (if any) have been resolved with the help of developer prioritization.
- **Code Repository**: This repository stores two types of codes—*Functional Requirement (FR) Codes* and *NFR Codes*. FR Codes are used to implement specific functionalities represented by goals and tasks. NFR codes are used to capture operationalizations of structural high-level NFRs.

The FR codes within the Code repository may be developed by software developers (who may or may not specialize in Requirements Engineering). The NFR Repository and the NFR codes are typically created, updated and managed by requirement engineers who are well-trained and experts in NFR management. The Code repository may be built incrementally—the greater the availability of FR and NFR code components, the richer is the quality of the App generated by the GRL2APK framework. The GRL2APK approach is aimed at driving towards the automation of app generation based on the availability of integrable code components within a code repository. However, validation of the app being generated with respect to the requirements captured in the original goal model still remains a necessity.

The GRL2APK tool is built on the newly proposed framework with the help of mainly four technologies: *Acceleo* (a platform for code generation), *Google Firebase* (cloud storage for NFR catalogue repository), *Amazon AWS S3* (cloud storage for functional code repository) and *Java Services* (used at the back end for consistency checking and app generation). We will elaborate on each of these technologies and how they have been used for building the GRL2APK tool (Fig. 6.10).

6.2.3.2 Components of the GRL2APK Tool

The GRL2APK tool provides a guideline (only) as to how the different components of the GRL2APK framework (shown in Fig. 6.9) can be realized using state-of-the-art technologies. System developers can choose among alternate available technologies for realizing any of the components.

- **Acceleo** [4]: We provide as input a goal model written in `XtGRL` to the Acceleo platform of the Eclipse tool. Acceleo is an Eclipse-based product created and developed by the Eclipse Strategic Member Obeo. Acceleo uses Model to Text language (MTL) to extract the component of a model. It supports Java services behind the scene to process these kinds of domain-specific languages. Acceleo extracts the

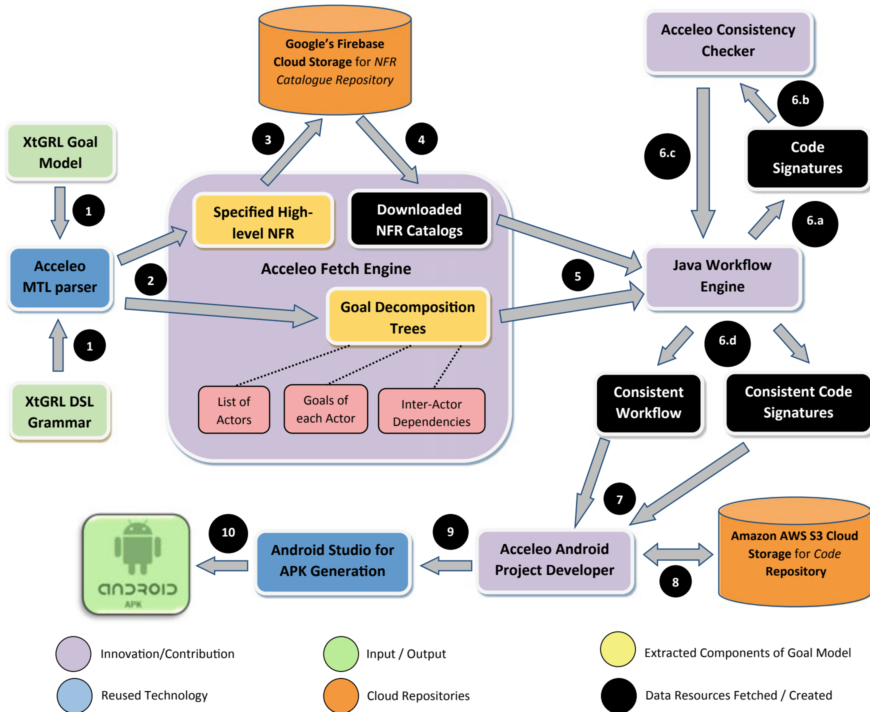


Fig. 6.10 The implementation framework with the process steps numbered in black circles

necessary information from the input requirements model. It also gives the provision to write Java services to accomplish specific tasks.

`Acceleo Modules`: The *Component Extractor* and *NFR Fetcher* modules in Fig. 6.9 are implemented using this technology. We call it the `Acceleo MTL Parser` and `Acceleo Fetch Engine`, respectively, in Fig. 6.10.

- **Google Firebase** [5, 6]: One of the vital assumptions of the proposed framework is to access NFR catalogs from a cloud repository as per the specified high-level NFRs. We use Google Firebase cloud service where we can store any kind of files. A dozen of Google Firebase cloud storage APIs provide flexibility to access specific NFR catalogs as specified in the requirements model. Another important aspect of the Google Firebase cloud storage is that we can authenticate every user of the application with ease.

  `Firebase Module`: The *NFR Fetcher* (shown in Fig. 6.9) uses Google Firebase APIs to download NFR catalogs from the NFR Repository. This component is called `Acceleo Fetch Engine` in Fig. 6.10.

- **Amazon AWS S3** [7]: AWS S3 is one of the leading object storage cloud services that allows accessing, storing and analysing any amount of data securely from anywhere. We have chosen S3 as the functional code repository. Amazon claims the durability of AWS S3 is about 99.99%. Several APIs are available to access functional codes stored in AWS S3 using Java services in Acceleo platform and integrate them with the other modules. Alternatively, *Google Cloud*, *Microsoft AZURE* or other cloud services could also be used for these repositories.

  `AWS Module`: The *APK Generator* (Fig. 6.9) uses AWS S3 APIs to integrate the functional code corresponding to the given goal model and chosen NFR operationalizations. We call it the `Acceleo Android Project Developer` in Fig. 6.10.

- **Java Services**: Finally we use several Java services to generate Android source codes from extracted components of the requirements model and NFR components of the NFR catalogs. We also integrate necessary files and dependencies of Android libraries and generate .APK file (Android Application Package) with Java services.

  `Java Modules`: The *Consistency Checker, Workflow Engine* and *APK Generator* heavily use Java Services. The corresponding modules in Fig. 6.10 are called `Acceleo Consistency Checker, Java Workflow Engine` and `Acceleo Android Project Developer`, respectively.

6.2.3.3 Workflow of the GRL2APK Tool

Input: Goal model specification capturing functional and high-level non-functional requirements, a cloud repository for NFR catalogs and another cloud repository storing functional code.

Output: An Android .APK file implementing the operationalizations of structural NFRs as selected by the developer.

Process Steps: (see Fig. 6.10)

*Step*1 Goal model specification and the `XtGRL` CFG is fed into the `AcceleoMTL Parser`.

*Step*2 MTL modules and Java services in the `Acceleo MTL Parser` process the goal model and extract the necessary components—high-level NFRs and goal decomposition trees.

*Step*3 According to the "demands"of the softgoals within the goal model, specific NFR catalogs (stored in the Google Firebase cloud storage) are accessed by the `Acceleo Fetch Engine`.

*Step*4 Google Firebase APIs are used to download those catalogs.

*Step*5 NFR Catalogs and Goal Decompositions are fed into the `Java Workflow Engine`.

*Step*6 The framework iteratively derives a set of Code Signatures that are conflict-free as follows:

(a) The developer selects his desired Code Signatures for FRs and NFR operationalizations.
(b) The NFR operationalizations are checked for conflicts in the `Acceleo Consistency Checker`.
(c) In case of an NFR conflict, the developer is prompted to choose another set of operationalizations. The corresponding Code Signatures are collected and the process is repeated.
(d) If there is no conflict, then the `Java Workflow Engine` generates the Consistent Workflow and Code Signatures for generating the APK.

*Step*7 The Consistent Workflow and Code Signatures are fed into the `Acceleo Android Project Developer`.

*Step*8 The `Acceleo Android Project Developer` uses Amazon AWS S3 APIs to access the code repository and download the actual code components.

*Step*9 The `Acceleo Android Project Developer` creates the Android Studio project for APK generation while making necessary changes to the root *AndroidManifest.xml* file.

*Step*10 The Android project created in `Step-9` is fed into Android Studio for compilation and building of the APK file.

6.2.3.4 Generating a Remote Healthcare Android App

We have considered remote healthcare system for our case study. It refers to the ongoing healthcare project "*A Framework for Healthcare Services using Mobile and Sensor cloud Technologies*" under the Information Technology Research Academy *ITRA*.[1] The project coordinators agreed to share their code repositories that would help in the generation of Android applications using our proposed framework. We modelled a part of the system consisting of some functional goals and some NFRs like `Security` and `Data-space Performance`. In this section, we present in detail how the framework is executed in a real-life scenario. The necessary screenshots for every phase has been provided for better visualization.

The XtGRL Goal Model

In this section, we create a scenario where an actor Patient wants to submit his medical details. The corresponding goal *ProvideMedicalDetails* "demands" the high-level NFRs *Security* and *Data-space Performance*. The input XtGRL goal model is as follows:

[1]Project URL: https://itra.medialabasia.in/?p=632.

```
grl Health Care{
    actor Patient{
        goal SeekHealth care{
            decompositionType ='and';
            decomposedBy ProvideMedicalDetails,
            SendReports,GetMedicine;
        }
        goal ProvideMedicalDetails{
            demands Security;
            demands Data-space Performance;
        }
        softGoal Security;
        softGoal Data-space Performance;
    }}
```

NFR Catalogs

The more important functional requirement in the goal model, with respect to the proposed framework, is the `ProvideMedicalDetails` goal. This goal "demands" two different high-level NFRs—`Security` and `Data-space Performance`. The NFR catalogs corresponding to these two high-level NFRs are as follows:

```
nfrl catalog{
    nfr__SGoal Security{
        decompositionType=or;
        decomposedBy AES_Encryption, DES_Encryption;
    }
    op_SGoal AES_Encryption;
    op_SGoal DES_Encryption;
}
nfrl catalog{
    nfr_SGoal Data-space Performance{
        decompositionType=or;
        decomposedBy PPM, LZ, CM;
    }
    op_SGoal PPM;
    op_SGoal LZ;
    op_SGoal CM;
}
```

Workflow Engine Interface

Once the NFR catalogs are downloaded, the Workflow Engine allows the developer to decide the control flow between different goals and tasks as well as the implementation code signatures for both functional and non-functional requirements.

Consistency Checker

Once the Code Signatures are selected by the developer, the Consistency Checker module checks the chosen signatures against a NFR conflict database (that is stored in the NFR repository). In case of conflicts, the module shows a prompt as seen in Fig. 6.11. The value of 30% is derived from the conflict database. If there are no conflicts between the chosen operationalizations (for example, in our case study, Lempel Ziv (LZ) for Data-space Performance and AES_Encryption for Security), then the Workflow Engine creates the workflow file as shown in Fig. 6.12.

APK Generator

The APK Generator can now download the code modules based on the code signatures that are mentioned in the workflow file (Fig. 6.12). The workflow file captures the order of execution of goals (or tasks), which in this case study turns out to be provideMedicalDetails(), followed by send_reports() and get_medicine(). The work-

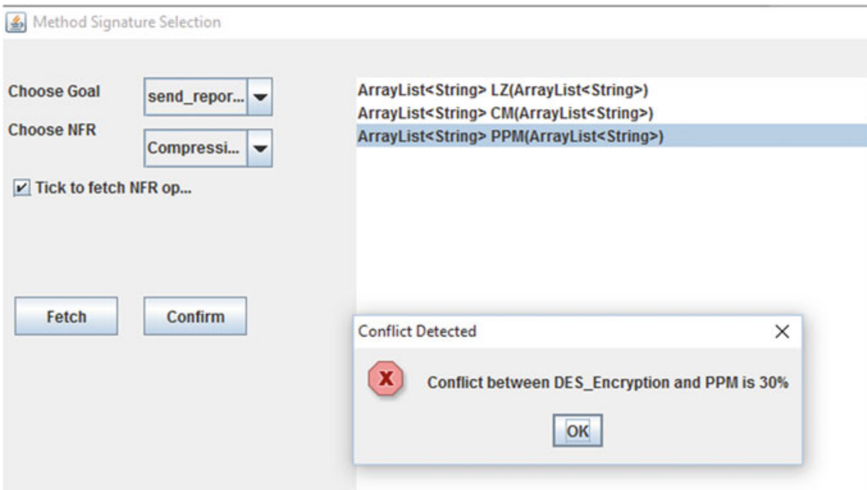


Fig. 6.11 Conflict identified between operationalization PPM (for Data-space performance) and DES_Encryption (for security)

```
void provideMedicalDetails(ArrayList<String>)
>> ArrayList<String> LZ(ArrayList<String>)
>> ArrayList<String> AES_Encryption(ArrayList<String>)
>> ArrayList<String> send_reports()
>> void get_medicine(ArrayList<String>)
>> void SeekHealthcare()
>> stop()
```

Fig. 6.12 Workflow generated for LZ (for Data-space performance) and AES_Encryption (for security)

flow also captures the order in which the NFR operationalizations have to be applied. The provideMedicalDetails() module passes the patient data (accepted as argument) to the LZ() code module for compression. The compressed data is then passed to AES_Encryption() module for encrypting before storage.

Figure 6.13 shows a screenshot of the app that is generated with the help of Android Studio for the above case study. Figure 6.14 shows how the data are stored in the Patient database. For illustration purposes, we included two dummy operationalizations—No_Encryption() and No_Compression()—to show the proper functioning of the GRL2APK framework based on developer's choice. A careful inspection of Fig. 6.14 shows a medical record of patient "ajit pal" which is being submitted from the app interface shown in Fig. 6.13. The selection of No_Encryption() and No_Compression() by the developer resulted in storing this data in the database as-is. Another patient data on the lower side of Fig. 6.14 shows how the data has been stored after applying LZ() compression followed by AES_Encryption(). Thus, depending on the developer's choice of NFR operationalizations, the generated app behaves differently.

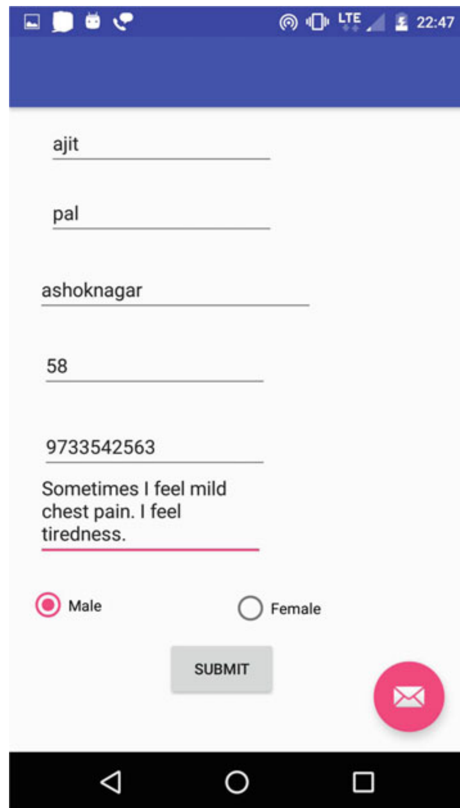


Fig. 6.13 Screenshot of the app

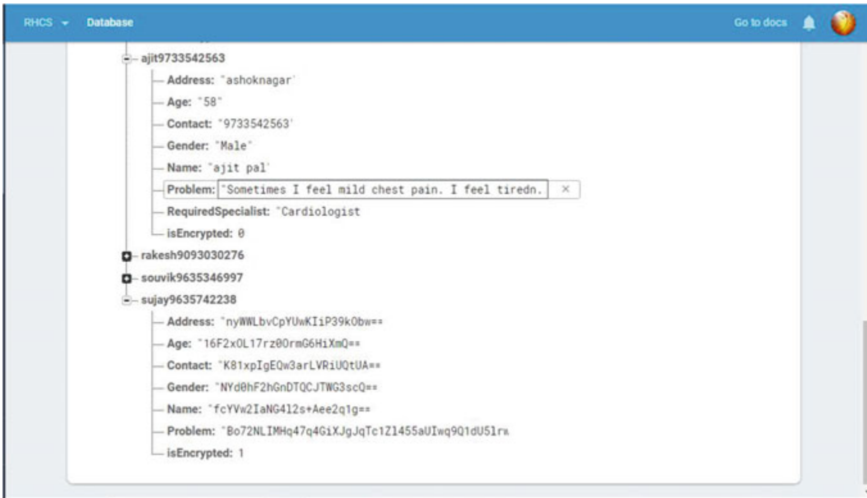


Fig. 6.14 Screenshot of patient database

References

1. Fuxman AD (2001) Formal analysis of early requirements specifications. MS thesis, Department of Computer Science. University of Toronto, Canada
2. Deb N, Chaki N, Roy M, Bhaumik A., Pal S Extracting business compliant finite state models from i* models. In: Advanced computing and systems for security (ACSS), advances in intelligent systems and computing, vol. 995. Springer, Singapore. ISBN: 978-981-13-8962-7
3. Deb N, Mallik M, Roychowdhury A, Chaki N Cargo: a prototype for contextual annotation and reconciliation of goal models. In: Accepted in the 27th international IEEE requirements engineering conference (RE)
4. Musset J, Juliot É, Lacrampe S, Piers W, Brun C, Goubet L, Lussaud Y, Allilaire F (2006) Acceleo user guide, vol. 2
5. Moroney L (2017) Moroney, Anglin, definitive guide to firebase. Springer. https://doi.org/10.1007/978-1-4842-2943-9
6. Stonehem B (2016) Google android firebase: learning the basics, vol. 1. First Rank Publishing
7. AWS (2016) Amazon simple storage service developer's guide. https://s3.cn-north-1.amazonaws.com.cn/aws-dam-prod/china/pdf/s3-dg.pdf