

Model-Based Development with Distributed Cognition

Karan Singh Hundal

Master of Engineering

Department of Electrical and Computer Engineering
McGill University, Montreal

August 2019

A thesis submitted to McGill University in partial fulfillment of the requirements of
the degree of Master of Engineering

© Karan Singh Hundal 2019

ABSTRACT

Model Driven Engineering (MDE) aims to define a system using a series of models. It can refine more abstract models into more concrete models by using automatic model transformations, allowing the most appropriate modeling notation to express system properties at a given abstraction level. The capability of MDE techniques to generate automatic model transformations is particularly useful for Requirements Engineering since it can ensure consistency between different kinds of requirements models and can automatically construct architectural, design, and testing models from requirements, allowing tighter integration between these major phases of software development. A key prerequisite for incorporating a technique into MDE is the definition of a formal modeling language. It is recognized in the requirements modeling community that front-end activities in requirements engineering, including requirements elicitation techniques, have received little attention in terms of MDE. A major reason for this is the lack of a formal modeling language for some of these techniques. The application of MDE to elicitation techniques would allow the results of elicitation to be effectively connected to the downstream activities in requirements engineering, taking model-driven requirements engineering to the next abstraction level.

This thesis proposes a formal modeling language for Distributed Cognition (DC), a well-known requirements elicitation technique, using the key concepts present in the framework's literature. First, a metamodel for DC is defined using its theoretical constructs. Then, combined DC and User Requirements Notation (URN) models are constructed and analyzed and a combined DC/URN methodology is developed by performing model transformations between the modeling notations. A detailed case study is performed by applying the developed

methodology on an existing socio-technical system from DC literature, demonstrating the methodology's functionality and feasibility.

ABRÉGÉ

L'ingénierie dirigée par les modèles (IDM) vise à définir un système à l'aide d'une série de modèles. Elle permet de raffiner des modèles plus abstraits en modèles plus concrets en utilisant des transformations de modèles automatiques, permettant au langage de modélisation le plus approprié d'exprimer les propriétés du système à un niveau d'abstraction donné. La capacité des techniques IDM à générer des transformations automatiques de modèles est particulièrement utile pour l'ingénierie des exigences car elle permet d'assurer la cohérence entre différents types de modèles d'exigences et de construire automatiquement des modèles architecturaux, de conception et de test à partir d'exigences, permettant une intégration plus étroite entre ces phases majeures du développement de logiciel. La définition d'un langage de modélisation formel est une condition préalable essentielle à l'intégration d'une technique dans l'IDM. Les spécialistes de la modélisation des exigences reconnaissent que les activités initiales de l'ingénierie des exigences, incluant les techniques d'élicitation d'exigences, ont reçu peu d'attention en termes d'IDM. L'une des principales raisons concerne le manque de langages de modélisation formels pour certaines de ces techniques. L'application de l'IDM aux techniques d'élicitation permettrait de relier efficacement les résultats de l'élicitation aux activités en aval de l'ingénierie des exigences, amenant ainsi l'ingénierie des exigences dirigée par les modèles à un niveau d'abstraction supérieur.

Cette thèse propose un langage de modélisation formel pour la Cognition Répartie (CR), une technique d'élicitation d'exigences bien connue, en utilisant les concepts clés présents dans la littérature liée à CR. Tout d'abord, un métamodèle pour CR est défini à l'aide de ses concepts théoriques. Ensuite, des modèles combinant CR et la Notation des prescriptions utilisateur (User Requirements Notation – URN) sont construits et analysés et une méthodologie combinant

CR/URN est développée en effectuant des transformations de modèle entre les deux notations. Une étude de cas détaillée est réalisée en appliquant la méthodologie développée sur un système sociotechnique existant provenant de la littérature en CR, démontrant ainsi la fonctionnalité et la faisabilité de la méthodologie.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my gratitude towards the people who supported me throughout the journey of my Masters research.

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Gunter Mussbacher, whose continuous support and guidance, throughout my thesis, was instrumental in the completion of this research work. His enthusiasm and dedication towards the field of requirements engineering motivated me to pursue and diligently follow through my research. Without his encouragement and guidance, this thesis could not have been completed. He was always willing to answer my research related questions and provided continuous assistance throughout the research process. I could not have imagined having a better supervisor and mentor for my Masters study.

My sincere thanks to Prof. Jörg Kienzle and other members from the Software Engineering lab for providing valuable feedback during my thesis. Their inputs always helped me create the best solution and hence, steered my work into the right direction. I would especially like to thank Sahil Luthra (for helping me understand the concepts of TimedURN), Nadine Bou Khzam (for helping me understand the concepts of EMF), and Devyan Kapoor (for helping me understand the language constructs of the ATL transformation language).

I thank all my friends here at McGill University, who always challenged me to do better. They made my stay here in Montreal memorable and fun.

Finally, I must express my very profound gratitude towards my parents and my sister, who were a continuous source of support and motivation throughout the rigorous process of my

masters study, especially through the process of researching and completing this thesis. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

ABSTRACT.....	2
ABRÉGÉ.....	4
ACKNOWLEDGMENTS	6
TABLE OF CONTENTS.....	8
LIST OF TABLES.....	10
LIST OF FIGURES	11
CHAPTER 1: Introduction	13
1.1 Motivation.....	13
1.2 Contributions and Methodology.....	15
1.3 Thesis Organization.....	17
CHAPTER 2: Background Information.....	19
2.1 Background on Distributed Cognition	19
2.2 Distributed Cognition for Teamwork (DiCOT)	19
2.3 Background on User Requirements Notation (URN)	22
2.3.1 Goal-oriented Requirement Language (GRL).....	23
2.3.2 Use Case Map Notation (UCM)	26
2.3.3 TimedURN	29
2.4 URN Metamodel	31
2.5 Rationale for Choosing URN	39
2.6 Summary	40
CHAPTER 3: Metamodel for Distributed Cognition	41
3.1 Artefacts	41
3.2 Physical Layout.....	45
3.3 Information Flow.....	49
3.4 Evolution	52
3.5 Social Structure	55
3.6 Metamodel Completeness	55
3.7 Summary	56
CHAPTER 4: Transformation from DC to URN	57
4.1 DC Notation	57

4.2 Application of DC Notation to Software XP Team	59
4.3 Implementation.....	65
4.3.1 ATL (Atlas Transformation Language).....	65
4.3.2 Transformation Rules	68
4.3.3 Steps for DC Model Creation and Transformation to URN.....	75
4.4 Generated URN Models for Software XP Team.....	77
4.5 Benefits of Integrating DC and URN.....	83
4.6 Summary	85
CHAPTER 5: DC-Based Case Study	86
5.1 The Airplane Cockpit System	86
5.2 Domain Model for an Airplane Cockpit System.....	89
5.3 Application of DC Notation on the Airplane Cockpit System.....	90
5.4 Generated URN Models for Airplane Cockpit System.....	96
5.5 Testing the Generated URN Models	103
5.6 Summary	103
CHAPTER 6: Related Work.....	105
CHAPTER 7: Conclusion and Future Work.....	107
REFERENCES	109

LIST OF TABLES

Table 1 The principles of Distributed Cognition, identified by DiCOT	21
Table 2 DC Notation	58
Table 3 Transformation for DC Artefact	68
Table 4 Transformation for DC Workflow	69
Table 5 Transformation for DC Physical Layout	72
Table 6 Transformation for DC Information Flow	72
Table 7 Transformation for DC Evolution.....	74
Table 8 Transformation for DC MetaClasses	74

LIST OF FIGURES

Figure 1 An example of a GRL Model [17]	25
Figure 2. An example of UCMs [17]	29
Figure 3. An example of TimedGRL model for Energy Efficiency goals of a city [3]	30
Figure 4. URN Main [34]	32
Figure 5. URN Core [34]	33
Figure 6. GRL Core [34]	33
Figure 7. GRL Links [34]	34
Figure 8. GRL Overview [34]	35
Figure 9. UCM Map Links [34]	36
Figure 10. UCM PathNodes [34]	37
Figure 11. URN Concern [34]	38
Figure 12. Extension of URN for TimedURN [3]	39
Figure 13. DC Metamodel specification for the Artefact Theme	42
Figure 14. DC Metamodel specification for the workflow of a system	44
Figure 15. DC Metamodel specification for the Physical Layout Theme	46
Figure 16. Domain Model For XP-based Software Development Team	49
Figure 17. DC Metamodel specification for the Information Flow of a system	50
Figure 18. DC Metamodel specification for the Evolution of a system	53
Figure 19. DC Metamodel Root Classes	56
Figure 20. The Plan under execution in the Software XP team	60
Figure 21. Flow of Actions for Project Manager in a Plan	61
Figure 22. Flow of Actions for Senior Business Dev in a Plan	62
Figure 23. Flow of Actions for Senior Developer in a Plan	62
Figure 24. The Information contained inside the Software XP team System	63
Figure 25. The Information Hub used in the Software XP team system	64
Figure 26. Matched Rule Example	66
Figure 27. Example of distinct for-each and Helper function	67
Figure 28. Generated GRL Model for Senior Business Dev	78
Figure 29. Generated GRL Model for Project Manager	78
Figure 30. Generated GRL Model for Senior Developer	80
Figure 31. Generated UCM Model for Senior Business Dev	81
Figure 32. Generated UCM Model for Project Manager	81
Figure 33. Generated UCM Model for Senior Developer	82
Figure 34. The UCM Model for Iteration Panning in a Software XP team system	82
Figure 35. Domain Model for an Airplane Cockpit System	88
Figure 36. The Plan under execution in the Airplane Cockpit System (i)	90
Figure 37. The Plan under execution in the Airplane Cockpit System (ii)	91
Figure 38. The Information contained inside the Airplane Cockpit System	92
Figure 39. The Information Hub used in the Airplane Cockpit system	93
Figure 40. Flow of Actions for Captain in a Plan	94
Figure 41. Flow of Actions for First Officer in a Plan	94

Figure 42. Flow of Actions for Second Officer in a Plan	95
Figure 43. Flow of Actions for ATC and High-Altitude Controller in a Plan.....	96
Figure 44. Generated GRL Model for Captain	98
Figure 45. Generated GRL Model for First Officer.....	98
Figure 46. Generated GRL Model for Second Officer	99
Figure 47. Generated GRL Model for ATC and High-Altitude Controller	100
Figure 48. Generated UCM Model for Captain	101
Figure 49. Generated UCM Model for First Officer.....	101
Figure 50. Generated UCM Model for ATC and High-Altitude Controllers	102
Figure 51. Generated UCM Model for Second Officer	102
Figure 52. The UCM Model for Cockpit in Airplane Cockpit system	102

CHAPTER 1: Introduction

Model-Driven Engineering (MDE) promotes the idea of using well-defined models as principle components during software development [29]. The key goal of MDE is to define more abstract modeling notations that can increase the level of abstraction in the software development process. These models can then be transformed from a higher level of abstraction to a lower level of abstraction by using automated model transformations. This provides several advantages to the software development process. Firstly, it allows the specification of a system's property at a given level of abstraction by using the most appropriate modeling notation. Secondly, it automates parts of the software development process thereby allowing a software engineer to focus more on the software solution being developed.

Requirements Engineering (RE) can particularly benefit from MDE. In a real system, the requirements of a system and the systems themselves evolve over time. Therefore, it would be beneficial for the designers to have a consistent set of requirements models that can be easily evolved with the system's evolution and can be used to rationalize the design decisions of a system. Furthermore, since MDE can automatically construct (partial) architectural, design, and testing models from requirements models, it has the potential to strengthen the integration between requirements models and downstream models.

1.1 Motivation

As discussed in the previous section, MDE techniques are quite beneficial for RE. This is equally true for early requirements models, e.g., models for requirements elicitation activities, which may automatically be transformed to late requirements model. However, it is recognized within the RE community that many front-end activities like requirements elicitation have not

been sufficiently incorporated into the model driven development environment [12]. However, this is quite beneficial because the application of MDE to elicitation techniques would allow for the effective connection of the results of elicitation to downstream processes in RE and eventually to other phases in software development. Therefore, it has the potential to take model-driven requirements engineering to the next abstraction level. For a model to be used in MDE, its modeling notations must be defined formally [12]. Interestingly, an important requirements elicitation technique, Activity Theory (AT), has been successfully integrated into the MDE development paradigm [10]. This was done by specifying a metamodel for AT based on its concepts and then connecting its elements to the concepts of the User Requirements Notation (URN). URN is a modeling notation for requirements elicitation and analysis and provides the ability to specify the requirements of a system using goal and scenario-oriented concepts. Therefore, the development of the combined AT/URN methodology demonstrates that requirements elicitation techniques can be incorporated into MDE.

In this thesis, a formal modeling language for Distributed Cognition (DC) is presented, which is a well-known requirements elicitation technique. While both DC and AT are cognitive theories, there are significant differences between them. In AT, the cognitive processes of an individual are at the center of everything. Contrastingly, DC attempts to understand the significance of a situation by focusing on the entire socio-technical system. It emphasizes on the interactions taking place between the individuals and their environment for analyzing how information is propagated and transformed around the system. Furthermore, it provides different representations for the information flow occurring in a cognitive system. Another important difference between them is that while AT explicitly names its theoretical constructs, this is not the case with DC. Therefore, while we can use a similar process to the one used for the

integration of AT and MDE [10], including the transformation of DC concepts to URN, the integration of DC into MDE requires a few additional considerations (e.g., DC does not explicitly name its theoretical constructs). An important point to consider is that other requirements notations may cover concepts that cannot be captured using DC. This is particularly true for goal-oriented concepts, which, e.g., are captured by URN, and the transformation of DC to URN provides the modeler an ability to specify these missing details. Therefore, the development of a language for DC and the transformation of DC into URN allows DC system designers to capture a wide range of domains which can then be further transformed and analyzed using existing tooling for URN models, while undergoing the model-based design process. Furthermore, the transformation between the modeling notations also benefits URN since DC provides major emphasis on the flow of information in a system, which is not currently captured in URN models, thereby strengthening the requirements specification capabilities of URN.

1.2 Contributions and Methodology

As mentioned in the previous section, DC does not explicitly name its theoretical constructs. Therefore, to utilize the key concepts of the methodology, the Distributed Cognition for Teamwork (DiCOT) framework [8][9] is employed. DiCOT is a DC-based framework, which focuses on analyzing a socio-technical system using the concepts of DC. It identifies 22 principles that are based on DC and can be used as a way of reasoning about both the existing system design and possible future designs. By incorporating the principles of DiCOT, the theoretical constructs of DC become more explicit.

In theory, the precise definition of a formal modeling language requires the following concepts: Abstract Syntax, Concrete Syntax and Semantics of the language. The Abstract Syntax captures the main conceptual elements of the problem domain and how they can be connected, including relationships between concepts and constraints on these relationships. The Concrete Syntax defines the textual and/or graphical representation depicting the conceptual elements of the language. The Semantics of the language describes the meaning of the concepts of the language. It can be described using the following four ways: Denotational (associating the concepts of the language with mathematical concepts (e.g., sets)), Extensional (extending the semantics of an existing language (e.g., with UML profiling)), Translational (describing the concepts of the language by expressing them in another language with precise semantics (e.g., mapping to Python programming language)) and Operational (modeling the evolution (or behavior) of model instances of the language (e.g., state transition system)).

This research work focuses on the definition of a formal modeling language for DC. For this purpose, the main contributions of the thesis are the major components of the language defined as follows:

1. The **Abstract Syntax** of the DC language: The abstract syntax of the language is defined using the DC metamodel [14]. The metamodel specification allows the identification of the key DC concepts, necessary to analyze a system using the proposed language. A few elements in an earlier version of the metamodel [14] are modified in this thesis to include additional scenarios that were discovered later.
2. The **Concrete Syntax** of the DC language: In this thesis, a formal definition for the concrete syntax of the language is not provided. However, a DC notation is introduced –

out of necessity – that is useful in visualizing the concepts discussed in the case study. However, the focus of this thesis is not on the concrete syntax for DC.

3. The **Semantics** of the DC language: In this research, the translational semantics of the language is specified by using model transformations from DC to URN (User Requirements Language) [1][17]. The rationale for using URN is explained in more detail in subsequent sections.

This thesis explains the complete DC metamodel using the application of Distributed Cognition to an XP team [30]. This is useful for illustrating the concepts introduced in this work. Furthermore, the DC metamodel’s transformation to URN, and the rules used for the actual transformations between the language metamodels are discussed. A detailed case study is performed using the application of DC on an airline cockpit [16]. This paper conducted analyses of a team working and the design of systems in an aircraft cockpit and was one of the most influential papers from the DC literature. Therefore, a case study based on this system gives strong evidence for the methodology’s functionality and feasibility.

1.3 Thesis Organization

The current thesis is structured into seven main chapters (including this introduction) that contain the definition of the DC metamodel and a case study demonstrating the methodology’s functionality and feasibility.

- Chapter 2 introduces the key concepts of DC and the themes of DiCOT with the help of a Software XP team example. Furthermore, it introduces the concepts of the User Requirements Notation (URN).

- Chapter 3 presents the principles identified in DiCOT and defines the DC metamodel itself using these principles. It also utilizes the example of a Software XP team, to provide a detailed description of all the introduced concepts.
- Chapter 4 describes a runtime instance of the DC metamodel based on a Software XP team. It utilizes this instance to analyze combined DC/URN models and describes the transformation of DC concepts to URN.
- Chapter 5 describes a case study to demonstrate our methodology and its usefulness. First, it describes a runtime instance of the DC metamodel based on an airplane cockpit, which was analyzed using the DC framework in literature, and then transforms it to URN goal and scenario models for visualization and further analysis.
- Chapter 6 gives a brief overview of related work.
- Chapter 7 concludes the thesis and presents future work.

CHAPTER 2: Background Information

This chapter introduces the key concepts of DC and the DiCOT framework. It introduces a socio-technical system based on a Software XP team and utilizes it to elaborate the principles of DiCOT. Furthermore, it provides a brief introduction to the essential concepts of URN and describes the rationale behind choosing URN as the more detailed target models for the transformation of DC models.

2.1 Background on Distributed Cognition

Distributed Cognition (DC) [15] is a theoretical account of the distributed nature of cognitive phenomena across individuals, artefacts, and internal (i.e., cognitive) and external representations. In short, it views collaborative work as a single cognitive system [30]. It utilizes an event-driven description, emphasizing how information is transformed and propagated around the system as part of this interaction and distinguishes between the internal (in an individual's mind) and external representations of this information [15]. The framework has been used in many different contexts like interpreting qualitative data in field ship navigation [15], aircraft piloting [16], and call centres [11], and as an analytical tool in HCI [13]. This thesis uses the concepts of DC and a structured methodology, called DiCOT [8][9], to define the abstract syntax of the DC language.

2.2 Distributed Cognition for Teamwork (DiCOT)

This thesis adapts the DiCOT methodology to inform the metamodel of the proposed DC language. DiCOT (Distributed Cognition for Teamwork) [8] is a structured methodology, developed to analyze a sociotechnical system using DC. It comprises of five themes [8], i.e.,

three main themes and two additional aspects, developed by combining the ideas from Contextual Design and concepts of DC. The five themes are as follows:

1. The *artefact theme* focuses on the detail of the artefacts that are created and used in carrying out the activity under study. These are important in distributed cognition because they are an integral part of the cognitive system and how it operates.
2. The *physical layout* theme focuses on the physical environment within which the cognitive system operates. This is important from a distributed cognition perspective because the physical arrangement of the environment, including what an individual can see and hear, affects their cognitive space.
3. The *information flow* theme focuses on what information flows through the cognitive system, the media which facilitate that flow, and how the information is transformed in the process.
4. The *evolution theme* considers how the system under analysis has evolved over time to its current state. This can be helpful in analyzing situations like the adaptation of new artefacts to support system behavior.
5. The *social structure* theme considers the role played by different social groups of a system in coordinating activity and the segregation of responsibilities amongst these groups.

Furniss and Blandford [8][9] identify 22 principles from DC which can be loosely categorized according to these five themes. A brief introduction to these 22 principles is provided in Table 1 along with examples based on a Software XP team. A detailed discussion on the five themes and principles of DiCOT along with an introduction to the Software XP team is provided in subsequent sections. DiCOT has been applied to a variety of complex systems

including simulation-based team training [27] and mobile healthcare [22] for analyzing these systems using DC.

The example of a Software XP team has been taken from an observational field study on a mature XP team, conducted by Sharp and Robinson [30]. The study has been conducted on an XP team from a London based company that develops and maintains customer travel information webpages and alerts. The team has 23 developers, a project manager, and two business development staff who are acting as the customers. For the development process, the team follows one three-week iteration cycle with three one-week iterations within it. The team uses index cards for information sharing within the team. Index cards can be used as story or task cards and are kept at ‘The Wall’ which acts as a central focus of development activity for the current iteration. Therefore, each iteration has its own wall displaying different cards with a discrete set of stories and tasks assigned to the development team.

Table 1 The principles of Distributed Cognition, identified by DiCOT

Physical Layout	
Space and cognition	Considers the use of space available in a system to support activity, e.g., using the walls to display index cards and support code development
Perceptual	Considers how spatial representations aid computation, e.g., using different coloured index cards to represent tasks to be done
Naturalness	Considers how closely the properties of the representation reflect those of that which it represents, e.g., task cards are not a close representation of the development code to be completed
Subtle bodily supports	Considers the bodily actions used to support activity, e.g. nodding
Situation awareness	Considers how people are kept informed of what is going on, e.g., through what they can see, what they can hear, and what is accessible to the team
Horizon of observation	Considers what an individual can see or hear in a system (this influences situation awareness), e.g., certain developers are seated close to the wall and can therefore see their respective task cards relatively easily
Arrangement of equipment	Considers how the physical arrangement of the environment affects access to information, e.g., the arrangement of developers relative to each other
Information flow	
Information movement	Considers the mechanisms used to move information around the cognitive system, e.g., emails can move the information across the team
Information transformation	Considers when, how, and why information is transformed as it flows through the cognitive system, e.g., transforming requirements into code

Information flow (cont'd)	
Information hubs	Considers the points in a system where information flows meet and decisions are made, e.g., meeting rooms
Buffers	Considers units that hold up information until it can be processed without causing disruption to ongoing activity (these complement the Information Hub)
Communication bandwidth	Considers the ability of a communication channel to impart information, e.g., face-to-face communication imparts more information than email
Informal and formal communication	Considers the informal communications between individuals in a system, e.g., informal communication during a coffee break among the developers
Behavioural trigger factors	Considers the factors that cause activity to happen without an overall plan needing to be in place (this principle is beyond the scope of our current study), e.g., the story card on the wall causes developer to produce code
Artefacts	
Mediating artefacts	Considers the tools that are used to perform the activity, e.g., the wall and index cards
Creating scaffolding	Considers how people use their environment to support their tasks, e.g., creation of reminders by developers to reflect where they are in a task
Representation-goal parity	Considers how artefacts in the environment represent the relationship between the current state and goal state, e.g., completion of all story and task cards which represents completion of the development activity
Coordination of resources	Considers the resources of the system, like plans, goals, and history that are co-ordinated to aid action and cognition, e.g., utilizing the wall to reflect the planned work for a development iteration
Evolution over time	
Cultural heritage	Considers how the elements of the environment built up over time (these can include the inclusion of new elements like artefacts or actors to facilitate the activities in the system), e.g., the addition of new developers in the team to support its activities
Expert coupling	Considers the information processing cycles in the cognitive system (as an individual becomes an expert, these get faster), e.g., bringing new recruits up-to-speed by pairing them with experienced developers
Social structures	
Social structure and goal structure	Considers how the social structure within the team relates to the goal structure, e.g., the goals of a project manager differ from that of a developer
Socially distributed properties of cognition	Considers how the cognitive system is distributed within the team, e.g., project managers can have access to certain organizational information that is not available for developers

2.3 Background on User Requirements Notation (URN)

The User Requirements Notation (URN) [1][17] is an ITU-T standard that provides notations and techniques to visualize and analyze functional and non-functional requirements (such as performance, cost, security, and usability) [2]. It is a combination of two complementary languages. The first one, GRL (Goal-oriented Requirement Language), is used to model

stakeholders and their intentions. The second one, the UCM (Use Case Maps) notation, describes scenarios and high-level architectures and design. Additionally, the two languages can contain links amongst their elements. URN provides *URN links* (►) that can connect any URN model element. This can provide traceability between elements, e.g., from GRL to UCM model elements.

2.3.1 Goal-oriented Requirement Language (GRL)


GRL is a visual modeling notation used for capturing and analyzing system or business goals, intentions, and non-functional requirements (NFRs) of multiple stakeholders. Additionally, it can be used for specifying alternatives to be considered, decisions made, and rationales for making decisions. A GRL goal model is a connected graph that supports three main categories of concepts, namely *actors*, *intentional elements*, and *intentional relations*.

The *intentional elements* can be further categorized into five concepts:

1. **Goal:** A goal (◻) is a quantifiable (functional) objective that is often measurable based on an agreed-upon way.
2. **Softgoals:** A softgoal (◻) is a qualifiable but unquantifiable objective, which is essentially non-functional.
3. **Task:** A task (◻) represents a solution to (or operationalizations of) goals and softgoals.
4. **Resource:** A resource (◻) is an entity required by a goal, softgoal, or a task to be completed or achieved.
5. **Belief:** A belief (◯) is a rationale or argumentation for a contribution or an intentional relation and is associated with another intentional element.

The connections between these elements are created by using a variety of links in a goal model. These are known as *intentional relations* and can be categorized as follows:

1. **Decomposition Links** (+—): These links are comprised of AND, OR, and XOR relationships which allow an element to be decomposed into sub-elements (refinement).
2. **Contribution Links** (→): These links indicate the impact of one element on another. They have qualitative or quantitative contribution types.
3. **Dependency Links** (—■—): These links indicate a dependency between two intentional elements, typically residing in different actors.

In a goal model, the *intentional elements* normally (but not necessarily) reside within *actors*. An *actor* () is the stakeholder of a system, an active entity with intentions that executes actions to achieve its goals. Therefore, an *actor* is essentially the entity that performs tasks or use different available resources to achieve target goals and satisfy potential softgoals.

GRL provides the ability to reason about goals and requirements, since it has the capability to visualize the impact of conflicting goals and alternative solutions proposed to achieve a goal. This is particularly useful in reasoning about NFRs and quality attributes. The modeler can create a GRL *strategy*, which assigns an initial qualitative or quantitative satisfaction value to a set of intentional elements, to create alternative configurations of a GRL model. Furthermore, the modeler may assign an *importance* attribute for an intentional element inside an actor, describing the relative importance of this element for the actor. Both, the initial satisfaction values and the importance values, are used for the analysis of GRL models.

The analysis of GRL models is performed with the help of an *evaluation mechanism* that propagates the initial satisfaction values to high-level stakeholder goals and NFRs given a GRL

strategy. This eventually computes the satisfaction values at the actor level, also considering the importance of the attributes. Therefore, alternative strategies can be juxtaposed to reach the ideal trade-offs between conflicting stakeholder goals. However, for decisions leading to requirements for system and software engineers, it is essential to consider the evaluation results of a strategy relative to the results of other strategies rather than in isolation, since the result of a strategy does not yield an absolute assessment but rather a relative indicator of system performance of a strategy with respect to other strategies. A goal modeler can also integrate real-life measured values in the analysis of a goal model by using *Key Performance Indicators* (KPIs) to improve the accuracy of the analysis.

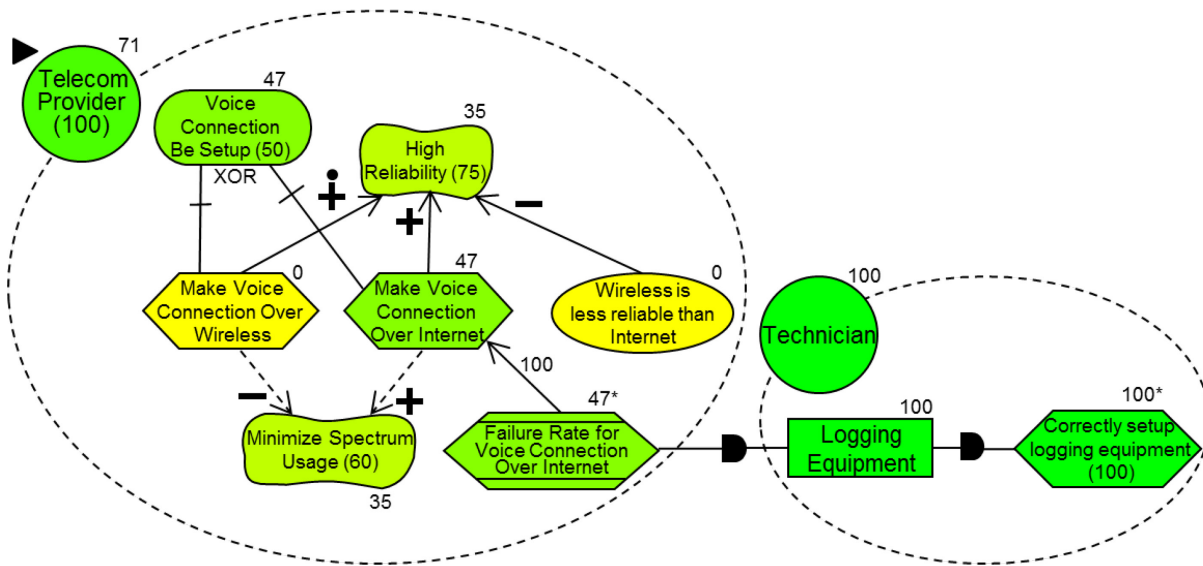


Figure 1 An example of a GRL Model [17]

Figure 1 provides an example of a typical GRL model. It contains the following *Actors*: “Telecom Provider” and “Technician”. These contain *Goals* like “Voice Connection Be Setup” and *Softgoals* like “High Reliability” and “Minimize Spectrum Usage”. “Make Voice Connection Over Internet” depicts a *Task* that contributes towards the achievement of different *goals* and *softgoals*. As explained before, rationale is captured by a *Belief* (for example,

“Wireless is less reliable than Internet”). “Logging Equipment” acts as a *Resource* required to achieve tasks or *goals*. The quantitative and qualitative contributions of different elements are depicted by using integer values like 60, 75 and symbols like +, −, respectively. Positive qualitative contributions differentiate \oplus (*make*, i.e., the contribution is positive and sufficient), \dagger (*help*, i.e., the contribution is positive but not sufficient), and + (*some positive* contribution). “Failure Rate for Voice Connection Over Internet” acts as a KPI, the value of which impacts the final goal model evaluation.

2.3.2 Use Case Map Notation (UCM)

The UCM visual scenario notation focuses on the causal flow of behavior. Modeling functional requirements of complex systems requires an emphasis on the behavioral aspects such as interactions between the system and its environment, on the cause to effect relationships among these interactions. UCM creates a behavior specification that is structured using components and provides an abstraction away from the message and data details. It supports the modeling and analysis of systems using scenarios, potentially combined with structural elements, at a high level of abstraction. UCM uses a *scenario* to partially describe a system usage that is defined as a set of partially ordered responsibilities performed by the system to transform inputs to outputs while satisfying preconditions and postconditions. A typical UCM contains four basic concepts:

1. **Start point** (●): A start point of a path captures triggering conditions.
2. **Responsibilities** (✕): A responsibility describes the required actions or steps to fulfill a scenario.
3. **End point** (■): An end point of a path represents resulting events and post-conditions.

4. **Paths:** A path expresses a causal sequence of responsibilities and may contain several other types of nodes.

Further, the notation contains many different types of nodes, used on the *path* to reflect different scenarios. The essential nodes are:

1. **OR-forks** (\neg) and **OR-joins** (\neg) are used to show alternatives. An OR-fork optionally contains a guarding condition for each of its branches. These nodes can also be used to model a loop.
2. **AND-forks** (\neg) and **AND-joins** (\neg) are used to depict concurrency. The notation does not impose nesting constraints on OR-forks, OR-joins, AND-forks, and AND-joins.
3. **Waiting places** (\bullet) and **timers** (\odot) denote locations where a scenario stops until a condition is satisfied or a timer expires.
4. **Stubs** (\diamond) are used to structure the UCM models hierarchically. These contain plug-in maps which are essentially reusable units of structure and behavior. A stub can be *static* or *dynamic*. A *static* stub contains only a single plug-in map while a *dynamic* stub can contain multiple plug-in maps which are selected based on the defined *selection policy* (i.e., essentially a guarding condition for each plug-in). The continuation of a path on a plug-in map is provided by *plug-in bindings*. These connect the in-paths and out-paths of a stub with the start and end points of a plug-in map, respectively.

In a UCM, *Components* (\square) specify the structural dimensions of a system which enables the specification of high-level architectures and design. Components can also contain sub-components and have various types like actor, team, agent, object, or process. A typical *map* can contain any number of *paths*, *nodes*, and *components*.

The analysis of UCM models is performed with the help of a *path traversal mechanism* which, given a UCM *scenario* with different pre- and post-conditions, start points, and expected end points, highlights the traversed scenario path to visualize the desired scenario in the UCM model.

Furthermore, GRL *strategies* and UCM *scenarios* are integrated with each other. This allows a scenario to be highlighted that belongs to an active strategy, provides scenarios the ability to influence the satisfaction values of GRL model elements by, e.g., changing them in responsibilities, and allows these satisfaction values to be used in guarding conditions. Finally, URN also includes *metadata* to tag any URN model element with additional information, hence allowing the definition of profiles for Domain Specific Languages (DSL)..

Figure 2 provides an example of typical UCMs. It shows different *components* like “OriginatingUser” and “TerminatingUser” which can be used to represent both software and non-software entities. The *start point* (“request”) triggers the events and *end points* like “busy”, “ringing” capture the resulting events. “forwardSignal” represents a *responsibility* performed during the execution of a scenario. Various concurrent and alternative elements exist along the paths of the UCMs. In UCM (v) “Teen Line (TL)”, the *condition* “[!TLactive]” is the guarding condition for the alternative paths in the map. “Originating” and “Terminating” ((i) “Simple Connection”) depict a *static stub* whereas “OrigFeatures” ((iii) “Originating Features”) is an example of a *dynamic stub*. The flow across these maps is ensured by defining different plug-in bindings between the *stubs* and the appropriate plug-in maps (e.g., the in-path of the “Originating” Stub is bound to the start point of the “Originating Features” plug-in map and the OUT1/OUT2 out-paths of the stubs are bound to the “success”/“fail” end points of the plug-in

map, respectively). Finally, “getPIN” in UCM (v) “Teen Line (TL)” is an example of a timer, implying that the scenario waits until the pin is entered by the user.

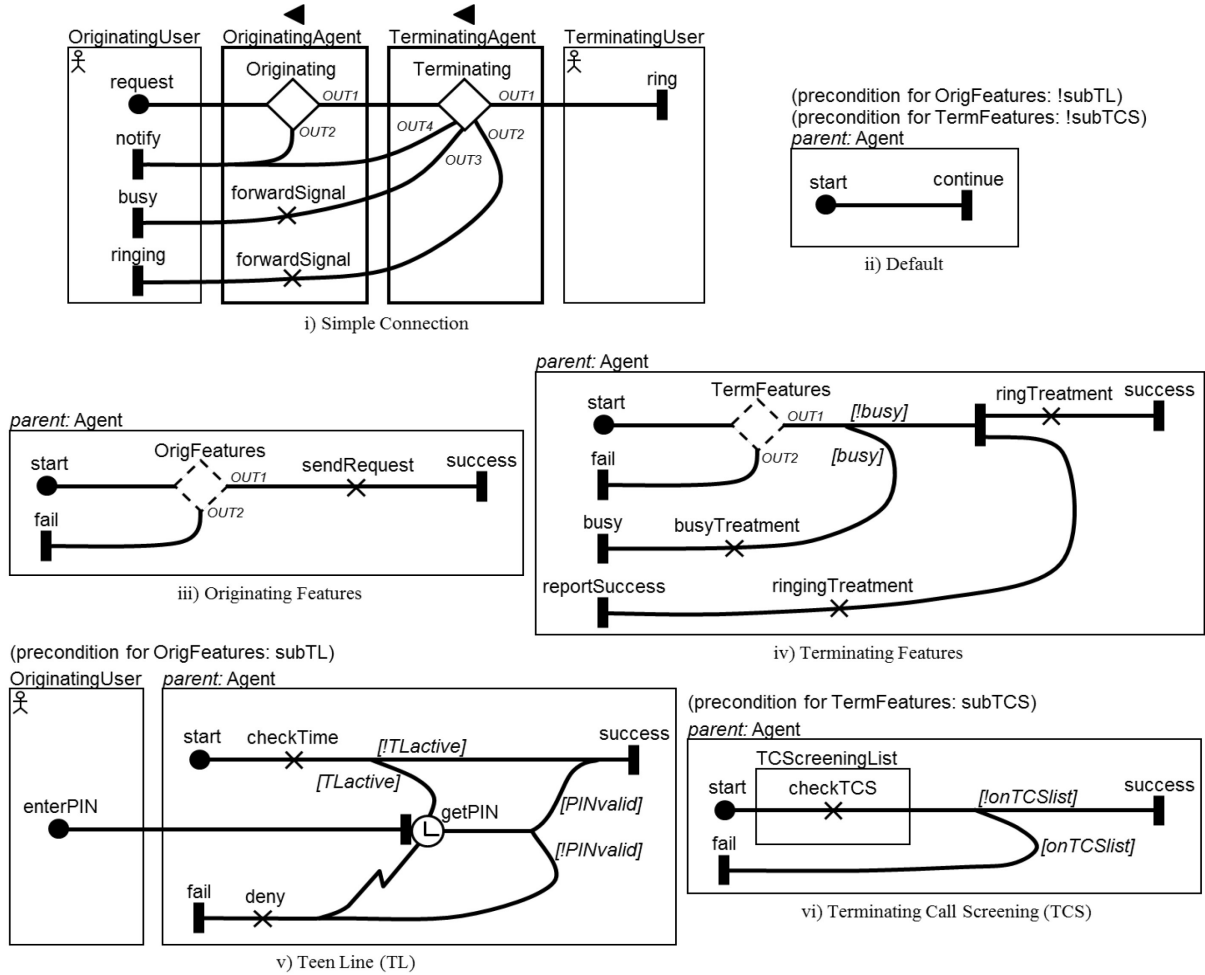


Figure 2. An example of UCMs [17]

2.3.3 TimedURN

TimedURN [3][4] is an extension of the URN standard, that provides the modeler the ability to model and analyze a comprehensive set of changes to a goal and scenario model within concrete time intervals. This is particularly useful since user requirements like stakeholder objectives, desired system qualities, potential solutions, and other model elements may evolve as time progresses. However, sometimes this evolution over time is predictable and can be

reasonably estimated. Therefore, it would be beneficial to explore several scenarios that might occur as the system model evolves. TimedURN provides a concise modeling environment for evolving systems. It specifies all changes in the same base model, which eases capturing all changes occurring on a system over time and maintaining a consistent model. In URN, TimedURN can be further classified into TimedGRL (Timed Goal-oriented Requirements Language) [4] and TimedUCM (Timed Use Case Maps) [3]. Combined, these allow changes to be modeled and analyzed for both goal and scenario models in a coordinated fashion.

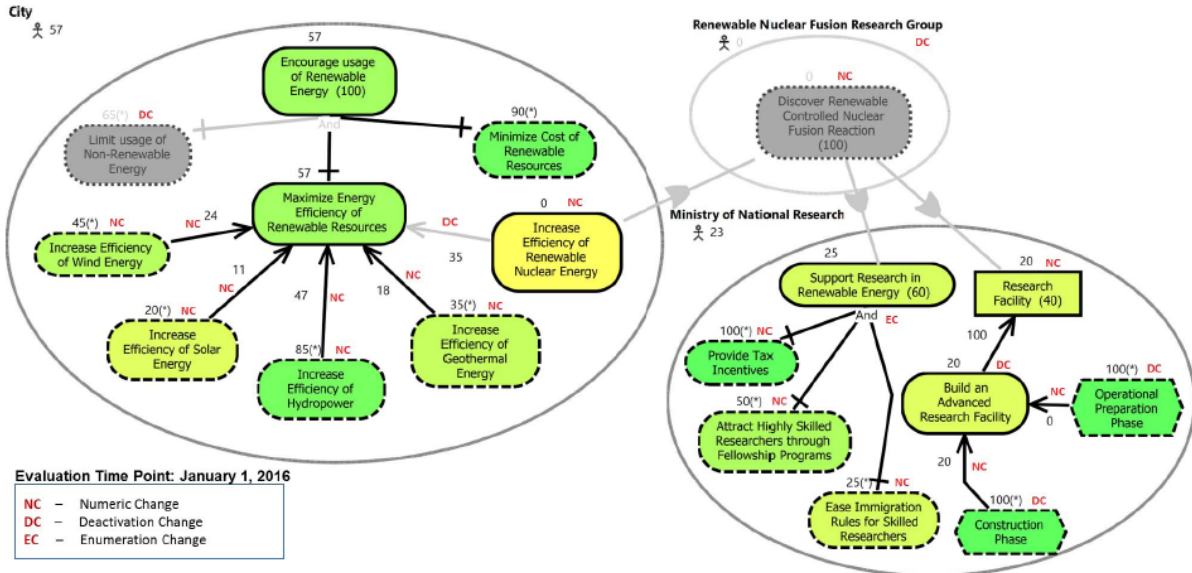


Figure 3. An example of TimedGRL model for Energy Efficiency goals of a city [3]

Figure 3 provides an example of a goal model that evaluates the evolving efficiency goals of a city. A *Change* is an expected behavior in a URN model over time. It contains a *start* and an *end* date to specify the time period for which the change element is applicable. It is further classified into *DeactivationChange* (marked as DC in Figure 3) and *PropertyChanges*. The latter like *NumericChange* (marked as NC in Figure 3) and *EnumerationChange* (marked as EC in Figure 3) are applied to an attribute (*affectedProperty*) of the model element.

In this example, the city wants to encourage usage of renewable energy by maximizing its overall renewable energy efficiency while minimizing the cost of renewable energy and limiting usage of non-renewable energy. *NumericChanges* can provide the ability to capture the improvements in efficiency for an energy source (e.g., the satisfaction value of “Wind Energy Efficiency” increases linearly from 45 in 2016 to 54 in 2031). *EnumerationChange* can specify the change in values over the course of time for any attribute with an enumeration type (e.g., “Support Research in Renewable Energy” changes from AND to OR in 2031). *DeactivationChange* is applied to the model element itself to define its actual existence timeline in the model. For example, a *DeactivationChange* is defined for the goal “Limit usage of Non-Renewable Energy”, because the city enforces this only in 2029, and is therefore currently grayed out.

TimedURN also specifies additional concepts, like *TimePoints*, which are discussed in further detail in Section 2.4. It should be noted that while the example provided in Figure 3 focuses on TimedGRL, the concepts discussed are equally applicable on TimedUCM.

2.4 URN Metamodel

This section provides an overview to the URN metamodel [34]. It needs to be noted that only the elements of the metamodel that are important for understanding the transformations from DC to URN are presented in this discussion. First, an introduction to the root classes of URN and its core elements is provided. This is followed by an elucidation of the elements present inside GRL and UCM, along with a discussion on URN *Concern* which groups these requirements specification elements together. Finally, the elements introduced inside URN to implement the concepts of TimedURN are presented.

The URN metamodel contains a root container called *URNspec*. It contains the following elements: *GRLspec*, *UCMspec*, *URNlink*, *URNdefinition*, and *Metadata*. Figure 4 presents these classes and the associations between them. The *GRLspec* acts as a container for the GRL specification elements. Similarly, UCM specification are contained inside *UCMspec*. *URNlink* is used to define a source-target connection between different *URNmodelElements*. Furthermore, a *Metadata* (which is essentially a name-value pair) is used to attach arbitrary information to *URNspec*, *URNlink*, or *URNmodelElement* in a URN diagram.

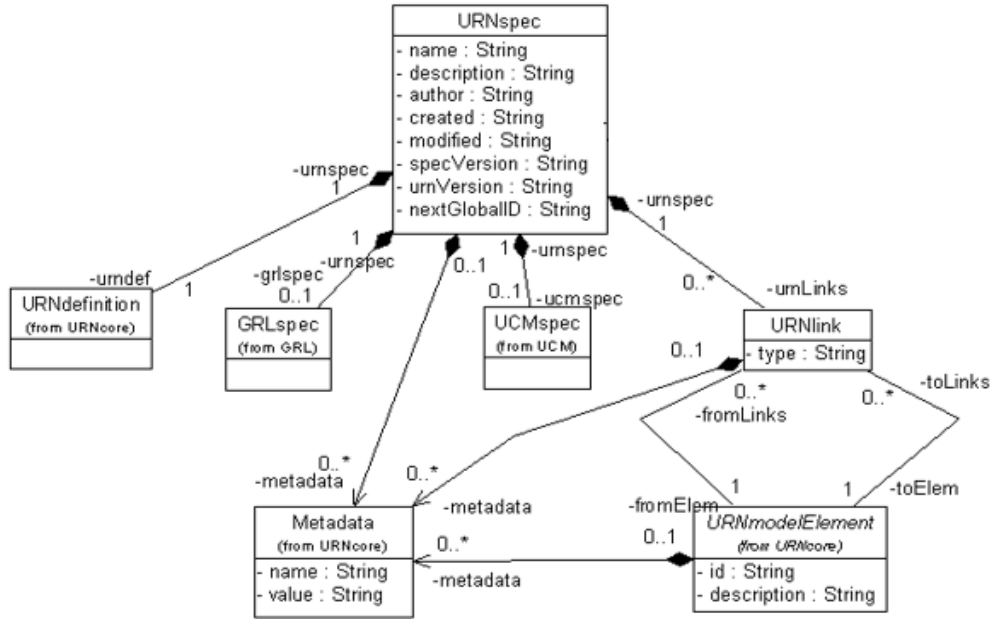


Figure 4. URN Main [34]

Figure 5 describes the core elements present inside URN. The *URNdefinition* (also contained inside *URNspec* (see Figure 4)) contains the following classes: *Responsibility*, *Component*, *ComponentType*. It also consists of specDiagrams (which could either be a *GRLGraph* for GRL elements or a *UCMmap* for UCM elements as abstracted by the *IURNDiagram* interface). A *Component* can be of different types like: *Team*, *Object*, *Process*, *Actor*, *Agent*, and *Other*. It can also contain multiple *Components*. Additionally, *ComponentType* provides the ability to specify a user-defined type for components.

Figure 6 describes the elements that belong to the core of GRL. A *GRLGraph* contains *GRLNodes* of the following types of subclasses: *CollapsedActorRefs*, *Beliefs*, and *IntentionalElementRefs*; and is associated with their respective connections, i.e., *BeliefLinks* and *LinkRefs*; and container references, i.e., *ActorRefs*. An *Actor* can include multiple *Actors* and has a relative importance defined with it. The *Actor* consists of different references namely, *ActorRef* and *CollapsedActorRef*. The *CollapsedActorRef* differs from an *ActorRef* in that it is used to highlight dependencies among actors without showing the internal goal hierarchy of the actor. The *ActorRef* provides the link between the *Actors* and its associated with *GRLNodes*. It can also be used to reflect a parent-child association between different *Actors*.

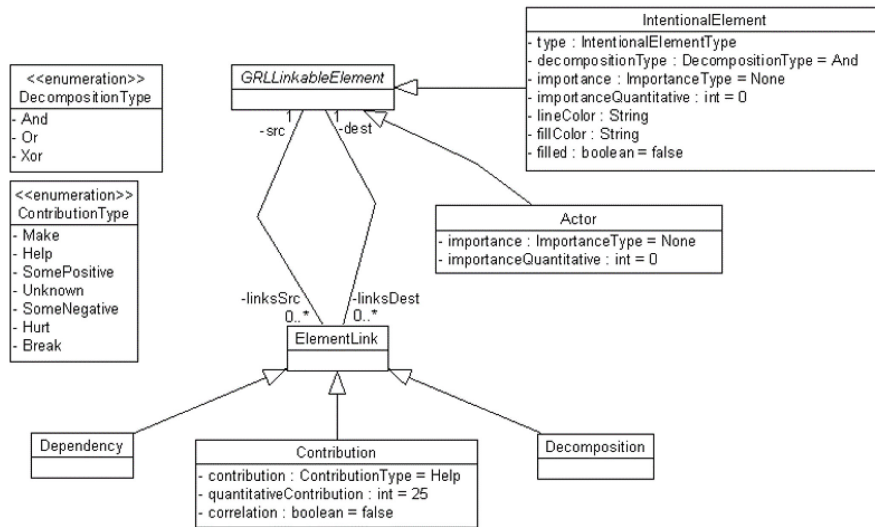


Figure 7. GRL Links [34]

Figure 7 describes the types of links that can exist between different elements of a *GRLGraph*. *Actors* and *IntentionalElements* are linkable elements in a *GRLGraph*. Therefore, they exist as subclasses of the *GRLLinkableElement* abstract class. The *ElementLink* class describes the types of links that can exist between GRL elements. It has three subtypes: *Dependency*, *Contribution*, and *Decomposition*. Each *ElementLink* is associated with a *LinkRef* (see Figure 6).

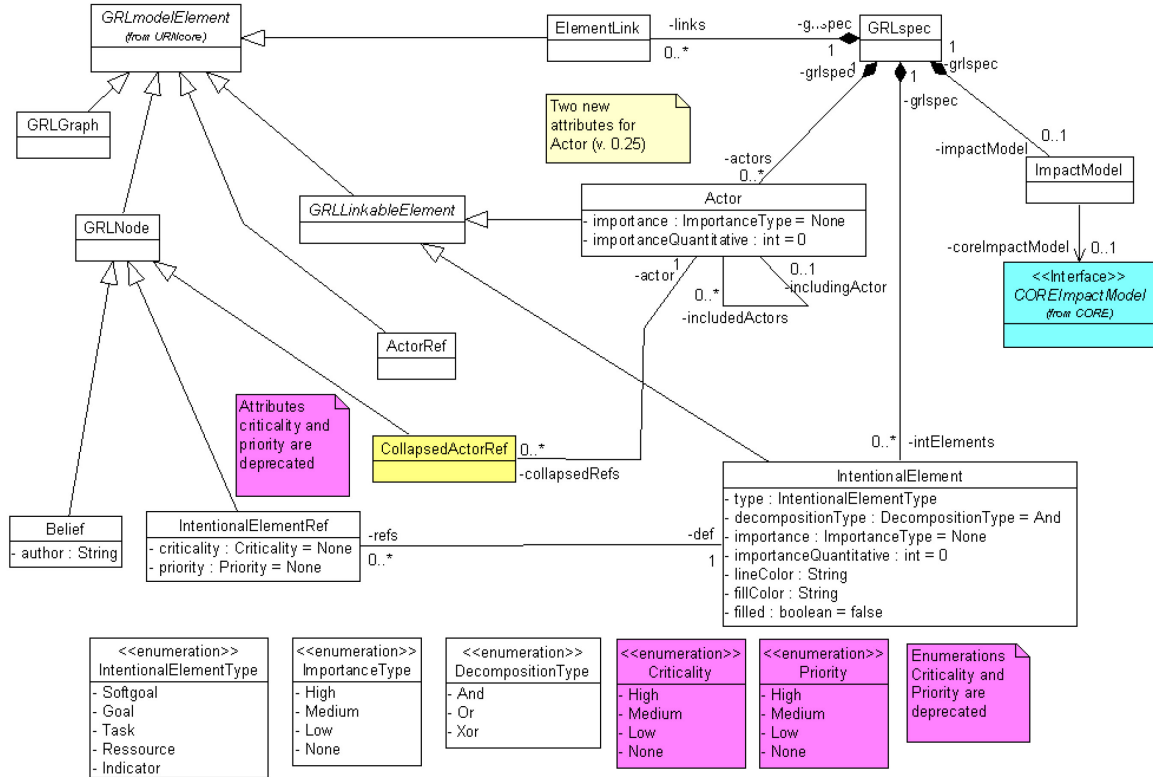


Figure 8. GRL Overview [34]

Figure 8 provides a complete overview of GRL elements and their relationships. The *GRLspec* acts as the root class and contains the *Actor*, *IntentionalElement*, and the respective *ElementLinks*. The *GRLGraph*, *GRLNodes*, *ElementLinks*, *GRLLinkableElements*, and their respective child classes exist as a specialization of the abstract class, *GRLmodelElement*, which is further generalized into *URNmodelElement* (see Figure 5).

Figure 9 describes the elements of a UCM Map. It consists of a *UCMmap* class that acts as a container for UCM specification elements. The *Component* and *Responsibility*, similar to *Actor* and *IntentionalElement*, have corresponding references called *RespRefs* and *ComponentRefs*, respectively. It also contains *PathNodes* which have associated *NodeConnections* that may contain a *Condition*.

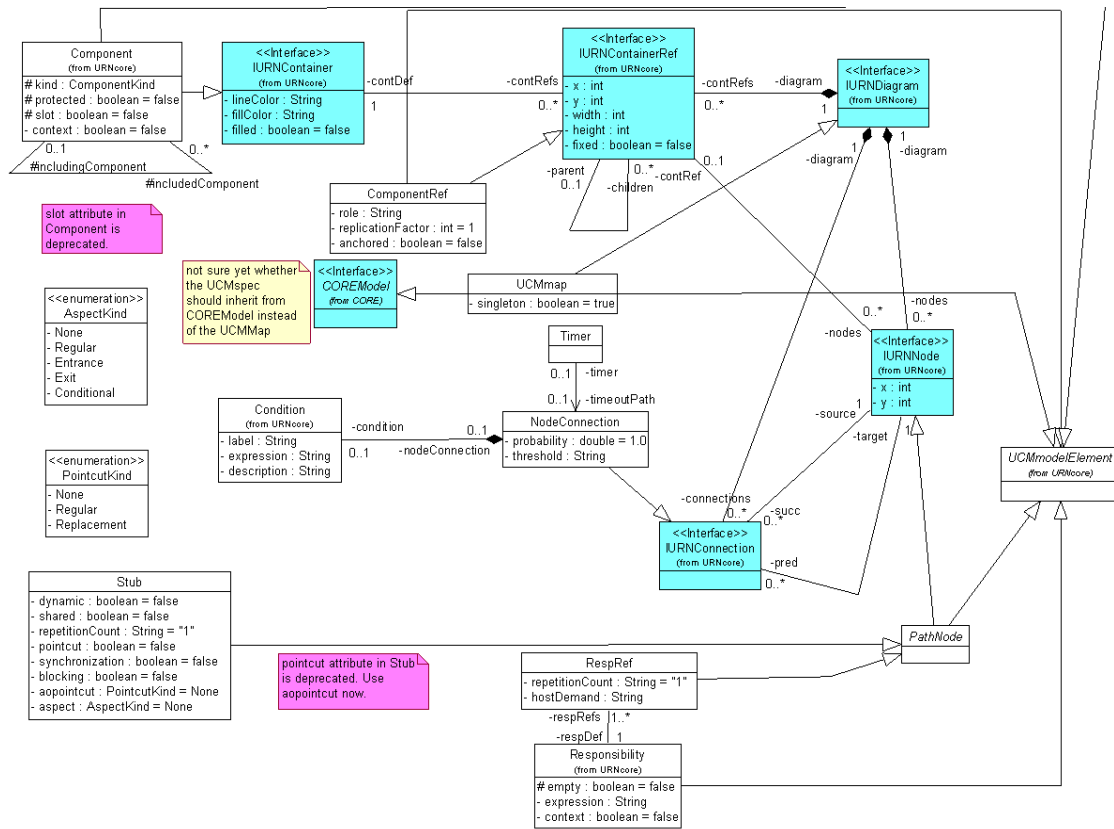


Figure 9. UCM Map Links [34]

Figure 10 shows the different types of PathNodes in a UCM Map. A *PathNode* can be of the following types: *EmptyPoint*, *Connect*, *WaitingPlace/Timer*, *Stub*, *OrFork*, *AndFork*, *OrJoin*, *AndJoin*, *StartPoint*, *EndPoint*, *RespRef*, and *FailurePoint*. Most of these elements are discussed in either this section or in Section 2.3.2. The *FailurePoint* indicates the location of a failure while the *DirectionArrow* indicates the flow of activity on a scenario path. An *EmptyPoint* is a path node that is useful to layout the path of a map. A *Connect* is used to connect two paths. This can be done either synchronously by connecting an *EndPoint* to another path (e.g., to a start point or a waiting place/timer that is triggered) or asynchronously by connecting an *EmptyPoint* to another path.

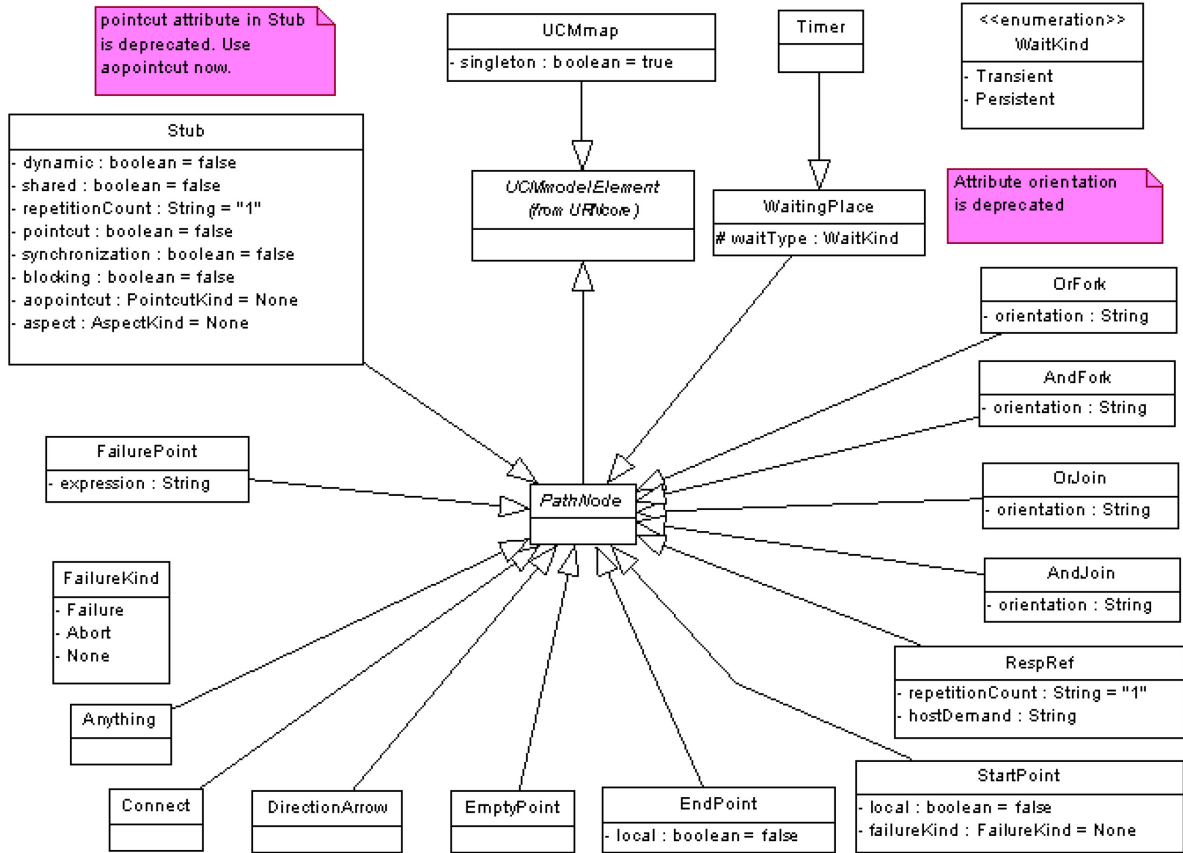


Figure 10. UCM PathNodes [34]

Similar to GRL, the abstract class *UCMmodelElement* is a specialization of the *URNmodelElement* and acts as a generalization of *UCMmap*, *Responsibility*, *Component*, *ComponentRef*, *PathNodes*, and their respective child classes (see Figure 10). The URN metamodel also contains elements that specify plugin bindings between elements but that is beyond the scope of this work. Furthermore, the URN metamodel also consists of elements that support various evaluation strategies, scenario definitions, as well as performance annotations. However, these are beyond the scope for this work and are not discussed in this section.

URN also groups related GRL and UCM diagrams (i.e., IURNDiagrams) and URN model elements into one unit of understanding. This unit is called a *Concern* (see Figure 11) and is guarded with a *Condition* for composition purposes.

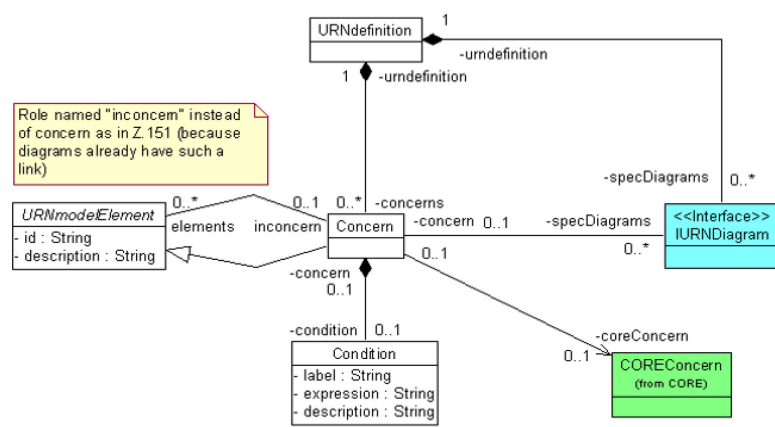


Figure 11. URN Concern [34]

Figure 12 describes the required extensions to the URN metamodel that have been introduced for implementing the concepts of TimedURN. The elements *Change*, *DeactivationChange*, *PropertyChange* along with its types *NumericChange* and *EnumerationChange* are discussed in Section 2.3.3. *BooleanChange* and *TextChange* can be used to define any new text or Boolean value for an attribute of a model element and are a kind of *PropertyChange*. NumericChanges can also be further categorized into different types of value changes like *QuadraticChange* or *FormulaChange*. All *Changes* are applied to *URNmodelElements* (*GRLmodelElement* or *UCMmodelElement*). *LinkDeactivationChange* impacts associations between URN model elements, allowing a link to be deactivated between elements in an URN model.

TimedURN also specifies additional concepts like *DynamicContext* and *Timepoints*. A *Timepoint* defines a specific date, while a *TimepointGroup* is a set of dates of interest for which the TimedURN model is to be evaluated taking the selected *DynamicContext* into account. A *DynamicContext* contains the grouping of *Changes* that act simultaneously. It allows the ability to observe the overall effect of these changes on the model and provides the option of having multiple groups to facilitate the exploration of alternatives and trade-offs. Both *DynamicContext*

and *Timepoint* are contained into their respective grouping elements, i.e., *DynamicContextGroup* and *TimepointGroup*, respectively. Like GRL and UCM specification elements, the *URNspec* acts as the root container for the extensions of TimedURN and contains the classes *DynamicContextGroup* and *TimepointGroup*.

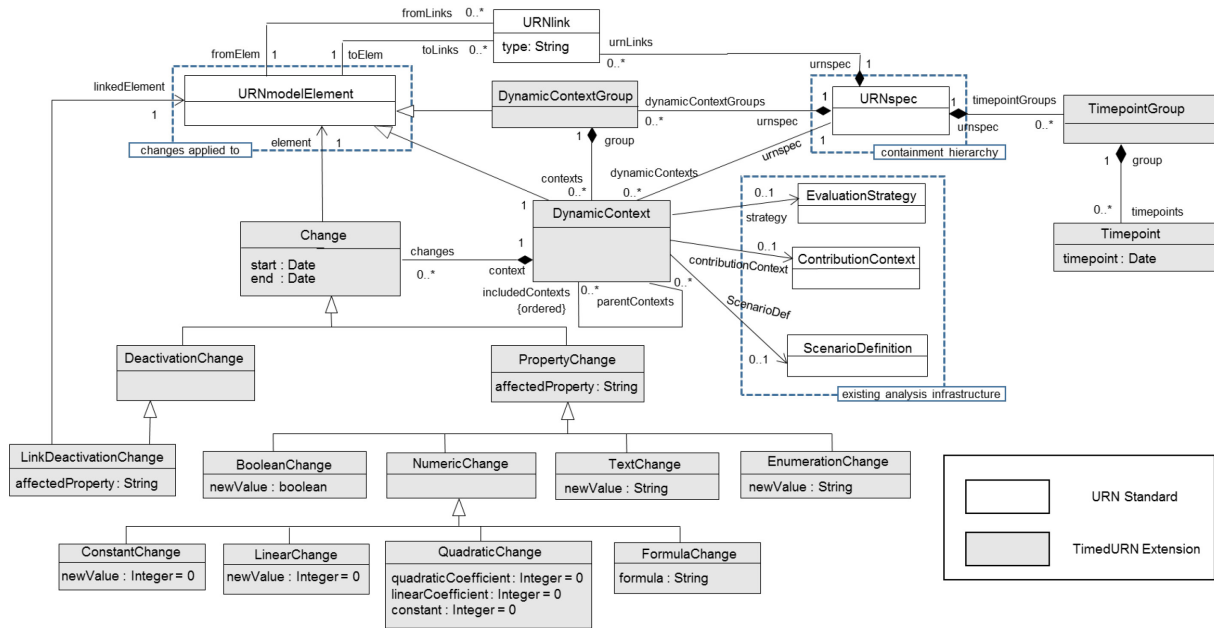


Figure 12. Extension of URN for TimedURN [3]

2.5 Rationale for Choosing URN

The main goal of this research work is to define a formal modeling language for Distributed Cognition. As discussed in Section 1.2, the semantics of a language can be defined by expressing its concepts in another language. However, the choice of the target language is quite important. It is because the principal motive behind defining a language for DC is incorporation of DC into MDE development paradigm. Therefore, the target language should contain precise semantics that adhere with other requirements engineering formalisms. URN is a standardized requirements specification language with a well-defined abstract syntax, concrete syntax, and semantics. It addresses explicitly, in a graphical way, goals and scenarios along with their relationships.

Furthermore, it provides additional capabilities like feature-oriented modeling [21], aspect-oriented concepts and evaluation [23], and support for combined modeling with Activity Theory (AT) and URN models [19]. The language and its associated tools like jUCMNav [18], developed at the University of Ottawa, and the TURN editor [20][28], developed at McGill University, are continuously evolving. Furthermore, jUCMNav provides traceability links between goal models and behavioral design models. Thus, integrating the DC methodology with goal and scenario modeling combined with the lower-level design capabilities of the jUCMNav tool provides us an ability to integrate DC with other requirements engineering formalisms and the Model-Driven Engineering (MDE) development paradigm.

2.6 Summary

This chapter discusses the essential concepts of DC and URN that are necessary to understand before discussing the concepts of the DC language in more detail. A brief introduction to DC and its existing applications is provided. Furthermore, DiCOT and its principles are introduced along with the example of a Software XP team to elaborate these principles. The principles of DiCOT are used in the next chapter to describe the DC metamodel. Finally, an overview to URN is provided and the rationale behind using URN for model-to-model transformations is discussed.

CHAPTER 3: Metamodel for Distributed Cognition

In this chapter, the metamodel for Distributed Cognition is defined. To do this, the principles identified in DiCOT are first listed based on its major themes and the classes that can be identified based on these principles are then defined. Furthermore, the example of a Software XP team (described in Chapter 2) is utilized to provide a detailed description of all the introduced concepts. It is important to note that this example is also used in subsequent chapters to define the concrete syntax of the DC notation. However, in this chapter, the focus is limited to utilizing the system for understanding the DC metamodel. An important advantage of utilizing this system is that the original field study on the Software XP team was conducted from a DC perspective. Therefore, it makes it easier to reason about the XP team using the DC metamodel since the identified metamodel instances can be validated by juxtaposing against the original field study.

3.1 Artefacts

In DiCOT, the artefact theme focuses on the artefacts that are used for carrying out the cognitive activity in the system under analysis. It contains the following four principles: (i) Mediating Artefact, (ii) Creating Scaffolding, (iii) Representation-goal parity, and (iv) Coordination of resources.

In order to define the metamodel classes using the Artefact theme, the concept of a *Resource* is introduced. A *Resource*, from a DC metamodel perspective, refers to (i) the tools (either physical or conceptual) used to perform an activity or (ii) the subjects engaged in an activity in the system. In Figure 13, the *Resource* is introduced as an abstract class. It is important to note that this is different from the resources discussed under the principle of Coordination of

resources. Finally, the key concepts of the Artefact theme exist as specializations of the *Resource* class.

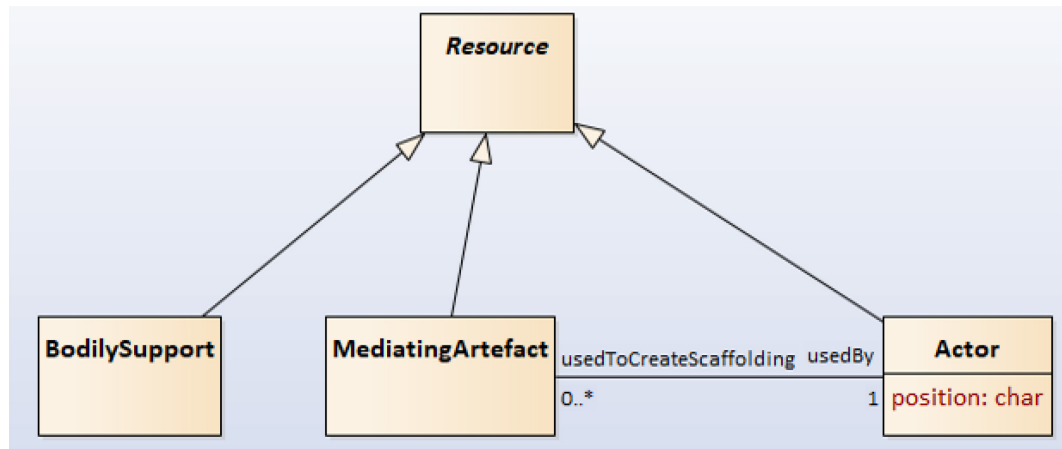


Figure 13. DC Metamodel specification for the Artefact Theme

Mediating artefact: Since DC considers the artefacts that are brought into coordination for the completion of an activity, a *MediatingArtefact* class represents the different artefacts that are used in the system to perform an activity. An *Actor* class is introduced to capture the entities performing an activity in the system. A many-to-many relationship exists between the *Actor* and the *MediatingArtefact*. The key *Artefacts* (short for *MediatingArtefact*) in the XP development team are the wall and the index cards. The runtime instance for this system will contain multiple walls (for different development iterations) and index cards. Other instances of the type *Artefact* would include email, books, developer machines, and the company wiki. The developers, project managers, and customers will exist as *Actors* of the system, each associated with appropriate *Artefacts*.

Creating Scaffolding: In this principle the focus is on how the actors of a system utilize the artefacts in their environment to simplify their tasks. An *Actor* uses one-or-more *Artefacts* to perform an activity or create scaffolding. For example, the *Actors* can use the objects for walls

and index cards for creating scaffolding since they can provide a clear idea of the progress made on the tasks.

Additionally, *BodilySupport* is considered as a subclass of the *Resource* abstract class. This class will be discussed in more detail in the physical layout theme. However, since *Actors* can use *BodilySupport* to support an activity, it is considered a part of the system's resources.

In the context of DC, Resources are abstract information structures that can be co-ordinated internally and externally to aid action and cognition [35]. DiCOT considers the six resources (i.e. plans, goals, possibilities, history, action-effect, and current state) that have been defined by Wright et al. [35] in their Resource Model. From an artefact model perspective, this helps in identifying the system in which the artefact operates. Therefore, in the metamodel specification, these abstract information structures are considered in relation to the resources present in the system. Additionally, during the design of the metamodel and analysis of the systems using DC, it became clear that most systems under analysis have a structured flow of activities which are performed by the different subjects present in the system. In Figure 14, the *DCWorkflow* class is introduced that captures the structured flow of activities in the system under analysis. An easier way to understand this workflow is to visualize an activity diagram with different activities executed in a sequential manner, where nodes represent states or control-flow structures and actions are performed on links between nodes. This workflow is then used to describe the co-ordination of the different abstract resources (*State*, *History*, *GoalState*, *Plan*, and *Action*) and utilizing these resources to achieve representation-goal parity.

Representation-goal parity and Coordination of resources: In DiCOT, the representation-goal parity captures how the artefacts in the environment aid cognition by providing an explicit representation of the relationship between the current state of the system

and the goal state. To accommodate this principal, the metamodel considers different *States* that are assigned to different stakeholders of a system. The *Plan* class represents the main plan that the stakeholders of a system are following to complete an objective. A DC system can be following multiple plans at a given time. The *Plan* class contains multiple *FlowLinks* and *FlowElements* and can be associated with one-or-more Actors at any given time. The *FlowLink* is associated with different Actions performed to achieve a task which further uses different *MediatingArtefacts* or *BodilySupport* and can possibly have a level of *Dependency* upon other elements (not explicitly shown in Figure 14). The *State* class represents the current state of an activity. The *GoalState* represents the final objective that the Actors are trying to accomplish. Both *ControlNode* (for control-flow structures) and *State* as well as *GoalState* are specializations of the abstract *FlowElement* class. This helps in defining a structured flow between the different states of the system. A *Plan* can be associated with multiple *History* states, which are essentially a combination of multiple *GoalStates* of previous *Plans*.

Figure 14. DC Metamodel specification for the workflow of a system

For example, the system can contain a *Plan* for completing the code development activity, involving *Actors* like the developers and project managers. They perform different *Actions* like communicating development tasks and actual code development, which creates different *States* in the system (e.g., the number of development tasks completed) and eventually results in their *GoalStates* (e.g., producing the code for all development tasks assigned to a developer). The number of development tasks completed in the previous development iteration acts as *History* for the current *Plan*.

3.2 Physical Layout

The Physical layout theme focuses on the physical environment under which the cognitive system operates. In DiCOT, the Physical layout theme considers the following seven principles: (i) Space and cognition, (ii) Arrangement of equipment, (iii) Perceptual, (iv) Naturalness, (v) Subtle bodily supports, (vi) Situational awareness, and (vii) Horizon of observation.

In Figure 15, the concepts defined in the Physical Layout theme are specified. Furthermore, an essential component of the metamodel is introduced, i.e., the *DomainModel*. The *DomainModel* of a system contains the elements present in the physical (or virtual) environment, for a system under analysis. It captures all relevant information and concepts of the system under analysis and contains a set of *DomainModelClasses*. The detailed description for each principle is provided below. A detailed discussion on the *DomainModel* for a Software XP-based team is provided in the subsequent section.

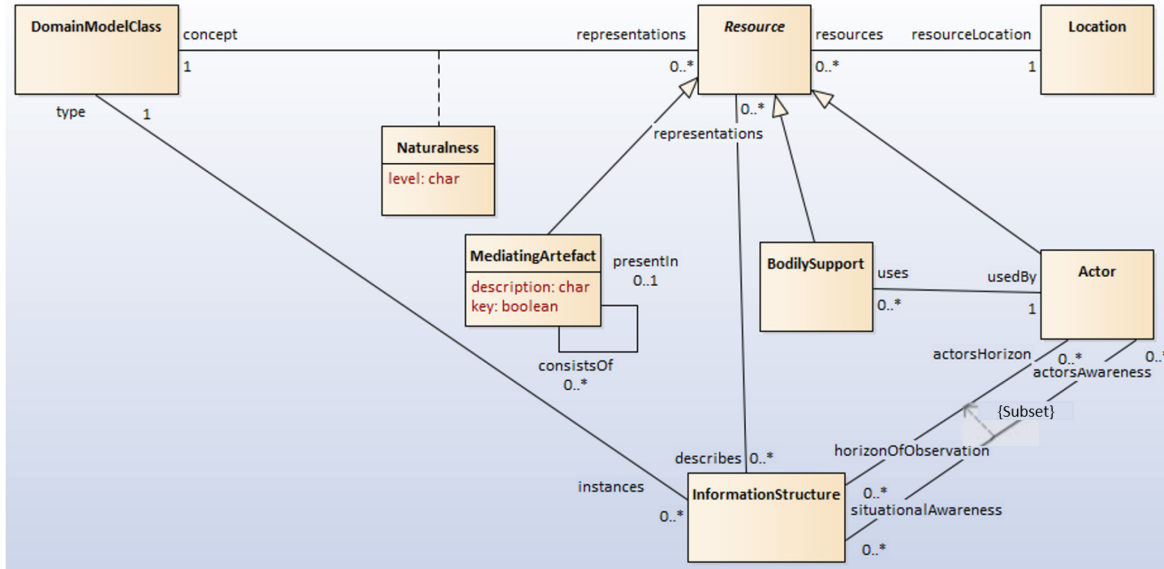


Figure 15. DC Metamodel specification for the Physical Layout Theme

Space and cognition and Arrangement of equipment: The metamodel considers the *Resource* class since the location of *MediatingArtefacts* and *Actors* is significantly important while considering the physical environment of a cognitive system. A *Location* class is introduced to specify the location of a *Resource* which facilitates the understanding of the arrangement of the equipment in the environment. Additionally, it is plausible that a *MediatingArtefact* consists of other *MediatingArtefacts* in the real world, reflected in the self-association for *MediatingArtefact*. E.g., a conference table consists of conference phones and speakers in an office environment. In the XP team, the runtime instances for key *Artefacts* (i.e., walls and cards) along with an object for the *Artefact* developer machine can be used to reflect space and cognition since the developers use these extensively to support their activities. The *Location* of the *Artefacts* and the *Actors* can be described using the *Location* class associated with the *Resource* class. This information can collectively describe how the equipment is arranged in the office environment. For example, two developers in the team might be located close to each other to facilitate pairing amongst developers.

Perceptual: This principle requires an understanding of the actual spatial representations used in the real environment. Therefore, it requires the specification of the *DomainModel* of the system under analysis. A *Resource* is related to a *DomainModelClass*, which can elucidate the perceptual principle since the pair describes the concrete, physical representation of an abstract entity in the *DomainModel* as a *Resource* in the system (for example, how the different *MediatingArtefacts* of the system provide the *Actors* with an idea of what needs to be done). For example, the *Artefact* objects for cards provide the *Actors* a clear idea of what needs to be done.

Naturalness: The *Naturalness* class is an association class between the information domain of the system and its concrete representation. It describes the closeness between the real world captured by a *DomainModelClass* and its associated representation (i.e., *Resource*) in the system under analysis. For example, one can assess how well the *Resource* card in the system represents the concept of requirement for a task in an iteration, as defined by *DomainModelClasses* in the domain model.

Subtle bodily support: The *BodilySupport* class specifies the use of bodily actions by the *Actors* to support their activities. Since an *Actor* can use multiple bodily actions, there exists a one-to-many association between the *Actor* and *BodilySupport*. An *Actor's* body action like pointing to a card can be assigned a *BodilySupport* object in the model, hence reflecting this principle.

Situational awareness and Horizon of observation: The principles of Situational Awareness and Horizon of observation in DiCOT are quite similar and influence the access of information to an *Actor* in a system. The *InformationStructure* class represents the information contained in a system and is described in more detail in Section 3.3. An *Actor's* access to information in a system reflects her *horizonOfObservation*. The Situational awareness of an

Actor is quite dependent upon her Horizon of observation. Therefore, *situationalAwareness* is modeled as a subset of the *horizonOfObservation* association between an *Actor* and *InformationStructure* in the DC metamodel. As *InformationStructure* describes actual information in the system, it is typed by a class in the *DomainModel*. These principles require a combination of the *Actor* object's *Location* and the *InformationStructure* objects for the information accessible to the *Actor*. The *Actor*'s *Location* is important since it influences her access to information like advices and questions. Again, the content of this information can be analyzed by creating instances of the *InformationStructure* class. The more information accessible to an *Actor* (i.e., the broader the *Actor*'s Horizon of observation), the potentially more situationally aware it becomes. For example, a team of developers located near each other can verbally communicate more easily, enabling team members to contribute insights to each other's conversations. Therefore, the Location of an *Actor* can impact her *situationalAwareness* and *horizonOfObservation*.

Domain Model for a XP-based Software development team: In Figure 16, a domain model is provided for the XP-based Software team. A typical development team contains three major types of employees: Developers, Business Dev, and Project Managers. They participate in different types of meetings (e.g., Daily Stand Up meeting, Iteration planning meeting, and Informal meetings). The main artefacts used for co-ordinating the development activities amongst the team are: Index Cards and the Wall. A Wall typically contains different types of index cards (story, task, and defects). Each type of card has a different color. Additionally, the team uses different tools like Email, Wiki, and development machines to complete their specific work items.

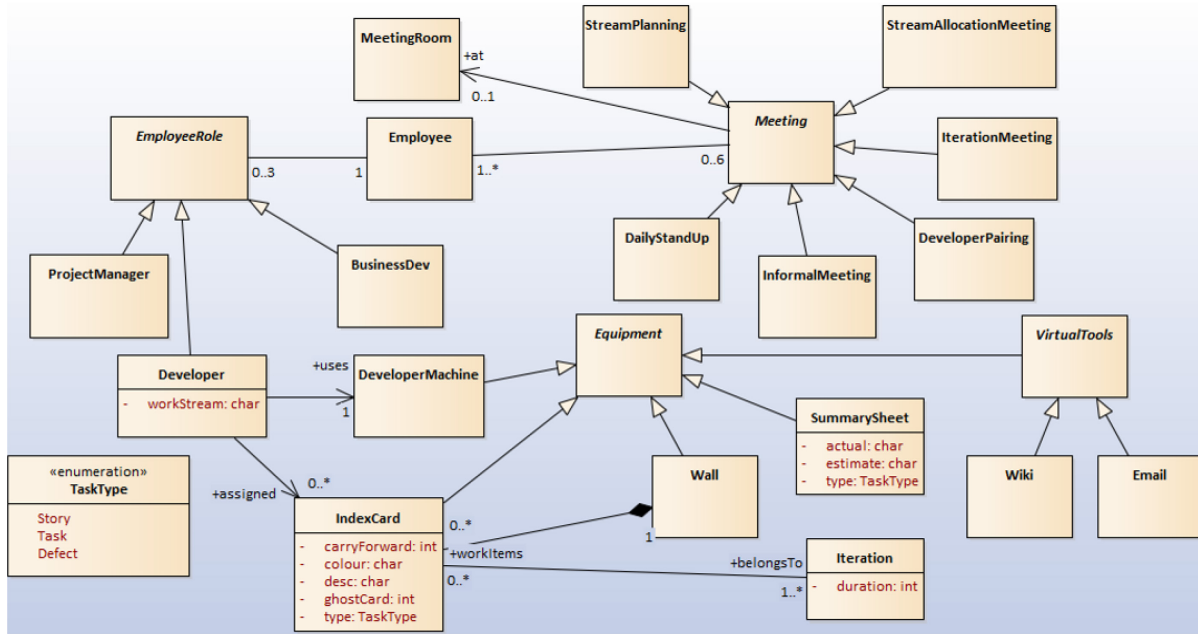


Figure 16. Domain Model For XP-based Software Development Team

3.3 Information Flow

The Information flow theme focuses on the movement and transformation of information in a cognitive system. In DiCOT, the Information Flow theme contains the following seven principles: (i) Information movement, (ii) Information transformation, (iii) Information hubs, (iv) Buffering, (v) Communication bandwidth, (vi) Informal communication, and (vii) Behavioural trigger factors (the last one is beyond the scope for this research work).

Figure 17 defines the concepts of the Information flow theme as applicable to the DC metamodel. A key component of the metamodel from an Information flow perspective is the *InformationStructure* class. This class describes the actual information that is contained in the system using different *Resources*. This information could be communicated in different forms like an Email or by an informal communication between actors and could be produced or transformed because of different actions occurring in a system.

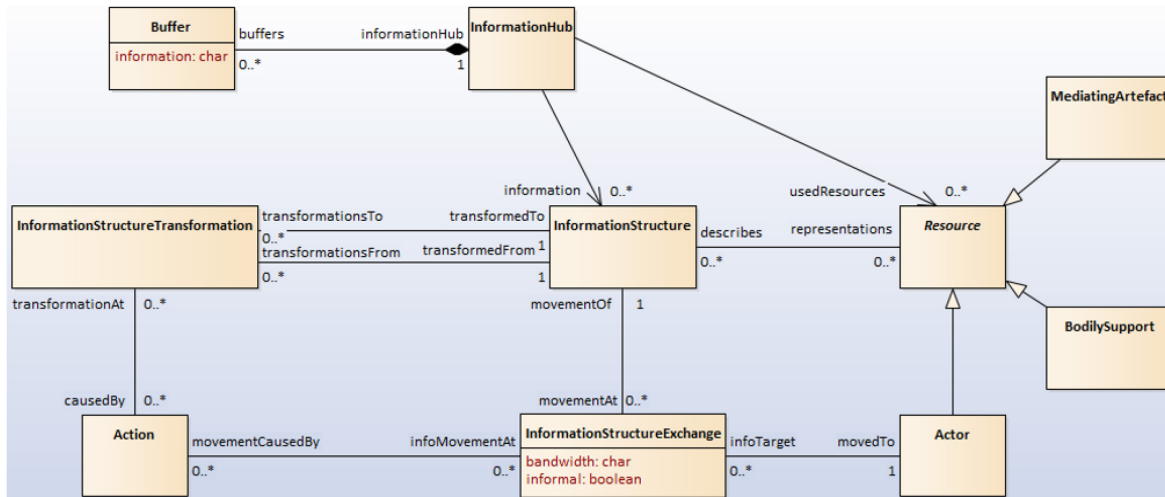


Figure 17. DC Metamodel specification for the Information Flow of a system

Information movement and Information transformation: The *Resource* class is quite essential to the Information flow theme, because the transformation and movement of information across the system is carried out by the Resources of the system, i.e., the *Actors* and *MediatingArtefacts*. However, since the focus is on the actual information on the artefacts rather than the artefacts themselves, the flow of information between actors is captured (since all information in the system is targeted towards different actors of the system). Furthermore, the actions are captured that cause this flow or transformation of information. An *Action* performed by an *Actor* (see Figure 14 in the Artefacts theme) may cause information movement (*movementCausedBy* association end) or transformation (*causedBy* association end).

In addition, another *Actor* is associated with an *InformationStructureExchange* to describe the target of the flow of information. Movement or transformation may also be a result of multiple actions (reflected by the multiplicities of *movementCausedBy* and *causedBy*). These actions can occur anytime during the execution of a *Plan* in the system (see Figure 14 in Section 3.1) and may be caused by different actors of a system. In the case of *InformationStructureTransformation*, the source *InformationStructure* (*transformedFrom*) and

the target *InformationStructure* (transformedTo) are specified. The flow of information in a system (*InformationStructureExchange*) can be described by using the instances of the *InformationStructure* class (to describe the information being moved), the instances for *Actor* (to describe the source and target of the information movement), and the *Action* (to describe what causing the movement). Therefore, in the XP team, an example would be an email from a project manager to the developers to describe the changes to a web interface. In this case, the *Actors* would be objects for project manager and developers, the instance for *Action* would be sending an email, and the object for *InformationStructure* would describe the actual changes to the web interface. Furthermore, consider the scenario of actual code development by the developers. This reflects a transformation of information (*InformationStructureTransformation*) from requirements to actual programming code. Therefore, this scenario can be described by creating instances for code development (*Action*), requirements, and developed code (*InformationStructure*). The *Actor* involved in this action is the developer.

Information hubs and Buffering: The *InformationHub* class represents the meeting point of different information channels in the system, where different sources of information are processed together (e.g., a meeting room). It is quite plausible that these hubs are busy depending on the information being considered in the real environment. Therefore, the *InformationHub* class contains multiple *Buffers* that ensure the efficient working of the hubs and the smooth movement of information. There exists a directed association between the *InformationHub* and *InformationStructure*. This allows the information to exist independent of the knowledge of the hub through which it flows. An *InformationHub* may be associated with multiple *Resources* (like meeting rooms may contain multiple team members, telephones, and whiteboards). For example,

the objects for meeting rooms and pairing between *Actors* need to be defined for an *InformationHub* object along with any *Buffers*, if necessary.

Informal communication and Communication bandwidth: The informal attribute of the *InformationStructureExchange* class describes whether an exchange of information is of an informal nature or not. The bandwidth attribute of the *InformationStructureExchange* class describes the richness of an information exchange between the actors. Consider an informal exchange of information occurring between two developers paired for a development task. This exchange can be captured by creating instances for *Actor*, *InformationStructure*, *InformationStructureExchange* (with the informal attribute set to true), and *Action* (and an instance for *BodilySupport*, if the exchange of information is verbal). Additionally, if the exchange of information is verbal, then its communication bandwidth is quite high. This can be described using the bandwidth attribute of the instance for *InformationStructureExchange*.

3.4 Evolution

The Evolution theme constitutes how the system has evolved over time to its current state. In DiCOT, the Evolution theme contains the following two principles: (i) Cultural heritage and (ii) Expert coupling. It is important to note that the elements in DC metamodel for the evolution theme are highly influenced by the concepts of TimedURN [3][4]. TimedURN provides the ability to specify evolving requirements over-time. Therefore, by specifying similar elements in the DC metamodel, the evolving nature of model elements over-time can be effectively captured.

Figure 18 covers the concepts of the Evolution theme by considering an abstract class *Changeable*. All the elements in the specified DC metamodel, that are changeable over time, are considered as specializations of this *Changeable* class.

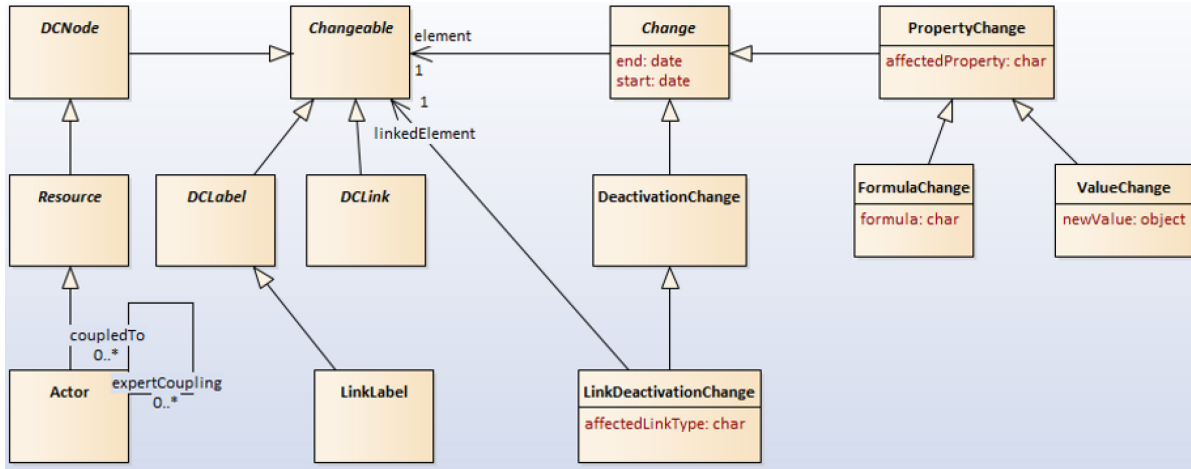


Figure 18. DC Metamodel specification for the Evolution of a system

Cultural heritage: By introducing the *Changeable* class as a parent class to all evolvable elements of the DC system (*DCNodes* and *DCLinks*), different aspects are covered in the evolution of a system like the inclusion of a new *MediatingArtefact*, *Actor*, or *Plan* in the cognitive system. For example, consider the workforce of an organization with multiple employees as the unit of analysis for a DC system. Each employee is modeled as an instance of the *Actor* class. Changes in the size of the workforce with time are captured by using *DeactivationChanges* for the affected employees of this organization. Therefore, the instances of the *Actor* class would change accordingly, and *Actor* is considered as a changeable element in our metamodel. As changes are captured explicitly, it is possible to reason about the evolving system.

An important aspect of the evolution part of the DC metamodel is the *Change* class, which is further refined into three key types of changes (*PropertyChange*, *DeactivationChange*, and *LinkDeactivationChange*), allowing all possible changes to a system over time to be captured [3]. The inclusion of the start and end attributes in the *Change* class allow the ability to visualize the validity of an object in the system. Consider a UML object diagram [25] that represents the runtime instances of the system under analysis. The *PropertyChange* can be used

to apply changes to the attribute values of objects (attribute is identified by *affectedProperty*), while the *DeactivationChange* and the *LinkDeactivationChange* can be used to add/remove objects or links between the objects, respectively (type of link is identified by *affectedLink*). The *PropertyChange* is further divided into *FormulaChange* and *ValueChange* to accommodate the different types of property changes possible. The *FormulaChange* expresses a change to a numerical value with the help of a formula based on time. The *ValueChange* essentially acts as a catch for all types of attribute values (e.g., Strings, Boolean, and Enumeration literals) that can undergo changes. It contains an attribute, called *newValue*, which is of the type *Object* (since an *Object* is the superclass for values like String, Boolean etc.) and can be used to specify the different types of changed values. No other types of changes are needed to manipulate the runtime instances of a system. As discussed above, the *Resource* class is a changeable element in the proposed DC metamodel. The XP team evolved from using index cards to summary sheets when they changed their office. This can be easily reflected using the changeable objects of *MediatingArtefact* and deactivating the objects that are impacted by the change. Additionally, scenarios like addition of new developers to the team can be easily accounted for by changing *Actor* objects.

Expert coupling: As an example of this principle, consider how the members of a team collaborate to bring new recruits up-to-speed. A self-association of the *Actor* class can reflect the principle of Expert coupling in the Evolution theme. This is logically justified since collaborations in a real system for knowledge transfer take place among individual *Actors*. The association identifies some of those actors as experts. For example, the addition of a recruit requires pairing with existing members of the team to bring the recruit up-to-speed. This can be described by using the *expertCoupling* association of the *Actor*.

3.5 Social Structure

The Social structure theme focuses on the role played by different social groups in a system. In DiCOT, the Social structure theme consists of the following two principles: (i) Social structure and goal structure and (ii) Socially distributed properties of cognition. In the proposed DC metamodel, the definition of a workflow introduces an element of goal structure to different actors in the system (see Figure 14) and the presence of expert coupling (see Figure 18) distributes the cognition amongst multiple subjects in the system. The concepts of horizon of observation and situational awareness (see Figure 15) further describe how cognition may be distributed among multiple subjects. Finally, multiple subjects may already be involved in information transformation and movement (see Figure 17). Therefore, no additional classes are introduced to specify the Social structure theme.

3.6 Metamodel Completeness

The elements of the DC metamodel that have been covered so far describe the different themes and their respective principles defined in DiCOT. In this section, additional concepts are introduced for the completeness of the proposed metamodel in terms of inheritance structure and containment. In Figure 19, the classes *DCNode*, *DCLink*, and *DCSpec* form the root classes of the DC metamodel. All concepts like *Resource*, *InformationHub*, and *Plan* are a kind of *DCNode* in the DC metamodel. Similarly, the concepts *FlowLink* and *Dependency* are a kind of *DCLink*. The subclasses of *DCNode* and *DCLink* are contained inside *DCSpec*, if they are not yet contained in any other class as shown in the previous figures of the DC metamodel. In addition, the *Change* class is also contained in *DCSpec* but is not a subclass of *DCNode* or *DCLink*.

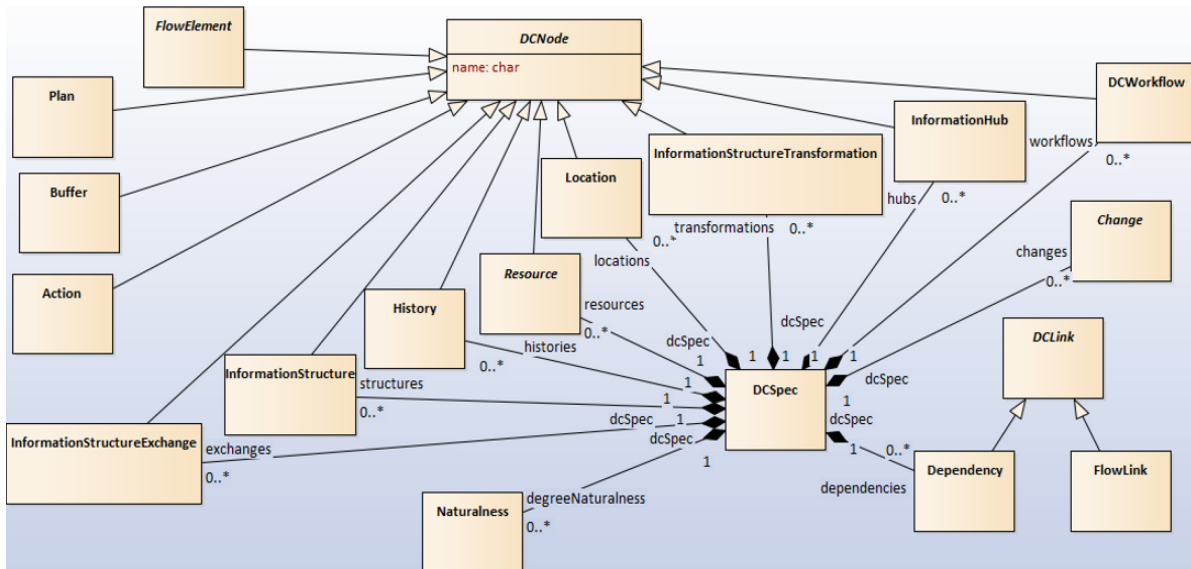


Figure 19. DC Metamodel Root Classes

3.7 Summary

This chapter elaborates the different principles of DiCOT, based on its five major themes. These principles are then used to formally define the metamodel for Distributed Cognition. A detailed explanation is provided for introducing the different classes in the metamodel. Furthermore, an existing system from the DiCOT literature is used to elucidate the concepts of the metamodel. Essentially, in this chapter, we formulate the abstract syntax for the language specification of DC. In the next chapter, we will use the same example of a software XP team to explain the proposed DC notation, i.e., the concrete syntax. Finally, we will explain the transformations from DC to URN and visualize them using the runtime instances of the software system.



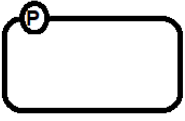
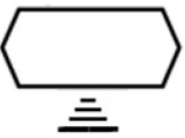








CHAPTER 4: Transformation from DC to URN












This chapter describes the transformation process for transforming DC elements into corresponding URN elements. First, a DC notation is introduced that is useful in visualizing the concepts discussed in the transformations. The notation is then applied to the Software XP team (described in Chapter 2) to visualize an instance of this system, analyzed using DC by applying the concepts of the DC metamodel. This is followed by a discussion of the ATL transformation tool and the actual transformation rules. The resulting URN models are then presented followed by a discussion on the benefits of constructing and analyzing combined DC/URN models.

4.1 DC Notation

Table 2 describes a DC notation which is described using the concepts defined in Chapter 3. The main motivation behind introducing the notation is its ability to facilitate the visualization of instances generated by analyzing a system based on the proposed DC metamodel. Most elements of the notation are congruent with the existing elements in the URN modeling notation, making it easier to juxtapose the initial DC model and the transformed URN model. While this is not meant to be the definitive concrete syntax for the DC modeling language, it provides the much-needed ability to discuss the transformations between DC and URN clearly. The notation primarily focuses on elements that are directly impacted by the transformations between the modeling notations. However, additional symbols are provided to describe most elements that would be defined in a typical instance model generated post-analysis of a system based on the DC metamodel. For all symbols defined in the notation, the name of the instance is provided in black colour next to the symbol in the visualized diagrams.

Table 2 DC Notation

DC Notation Symbol	Significance and Usage
	<ol style="list-style-type: none"> 1. Describes the Actor in the system. 2. All Actors of the system are visualized using separate diagrams to visualize individual flow of activities during the execution of a Plan.
	<ol style="list-style-type: none"> 1. Describes the MediatingArtefact present in the system under analysis. 2. A MediatingArtefact can be used to perform actions, carry information. It also has an associated Location in the notation.
	<ol style="list-style-type: none"> 1. Describes the Plan executed in a Workflow in the system under analysis. 2. All elements inside the Plan, in the visualized diagrams, describe different elements like Actors and MediatingArtefacts involved in the Plan. 3. The actual execution of steps in the Plan is provided by using separate diagrams for each Actor.
	<ol style="list-style-type: none"> 1. The hexagon describes the Action performed by an Actor to achieve their respective Goals inside a Plan. 2. All Actions require different MediatingArtefacts. 3. The stack of horizontal lines links the Action with the FlowLink to describe its execution during the flow of activities in a Plan.
	<ol style="list-style-type: none"> 1. Describes the InformationHub that uses different Resources of the system to facilitate the flow of information.
	<ol style="list-style-type: none"> 1. Describes the State of a system during the execution of a Plan.
	<ol style="list-style-type: none"> 1. Describes the Sequence elements introduced to link the different FlowLinks in the instance model.
	<ol style="list-style-type: none"> 1. Describes the Location of an Actor or MediatingArtefact in the system. 2. All Location elements are visualized inside the Plan. The associated elements are linked using (———).
	<ol style="list-style-type: none"> 1. Describes the LinkLabel to identify a condition defined on a FlowLink.
	<ol style="list-style-type: none"> 1. Describes the WorkFlow present in the system under analysis. 2. It is linked to the corresponding Plan by using (———).
	<ol style="list-style-type: none"> 1. Describes the History of the Plan under execution in the system. 2. It is linked to the corresponding Plan by using (———).
	<ol style="list-style-type: none"> 1. Describes the Goal of an Actor during the execution of a Plan in the system.

DC Notation Symbol	Significance and Usage
	<ol style="list-style-type: none"> 1. Describes the InformationStructure created as a result of activities involving different MediatingArtefacts and Actors in the system. 2. All the elements associated with the InformationStructure are linked using ().
	<ol style="list-style-type: none"> 1. Describes the BodilySupport used by an Actor in the system. 2. The Actor using BodilySupport is linked to it using ().
	<ol style="list-style-type: none"> 1. Describes the AndFork (i.e., concurrent branches in a workflow).
	<ol style="list-style-type: none"> 1. Describes the AndJoin (i.e., synchronization of branches).
	<ol style="list-style-type: none"> 1. Describes the OrFork (i.e., alternative branches in a workflow).
	<ol style="list-style-type: none"> 1. Describes the OrJoin (i.e., merging of branches without synchronization).
	<ol style="list-style-type: none"> 1. Provides a link between the Action and the corresponding MediatingArtefact. 2. The arrow points from the MediatingArtefact to the Action using it.
	<ol style="list-style-type: none"> 1. Describes the FlowLink.
	<ol style="list-style-type: none"> 1. All elements that are associated with a Change exist as the appropriate symbol from the ones specified above except that they have a different color (e.g., an Action with a DeactivationChange appears as the symbol on the left). 2. The duration of validity for elements linked with a Change is provided next to their representation symbol along with the type of change (e.g., DChange for DeactivationChange).

4.2 Application of DC Notation to Software XP Team

The main *Actors* in the XP team are the Senior Developer, Senior Business Dev, and the Project Manager. Figure 20 describes the *Plan* under execution in the Software XP Team to complete the development of a software project. It belongs to “WorkflowA” and contains the three *Actors* and their corresponding *Locations* in the system. The *MediatingArtefacts* used by the *Actors* to implement their *Actions* include the following: Wall, Story Card, Task Card, Bug

Card, Developer Machine A, Developer Machine B and the Summary Sheet. Like the system *Actors*, *MediatingArtefacts* have been linked to their respective *Locations* in the system. An important point to consider is that the definition of instances for the *Location* of an element considers the Wall as the reference point. Therefore, while all elements are present inside the XP team office, they have been linked to their *Locations* with respect to the Wall. Only the Wall and the Summary sheet have been linked to “Location Inside XP team office”. This is done because the Wall needs a primary *Location* and the Summary Sheet does not have a fixed *Location*. The *Plan* also has an associated *History* which identifies the development items completed before the execution of the *Plan*.

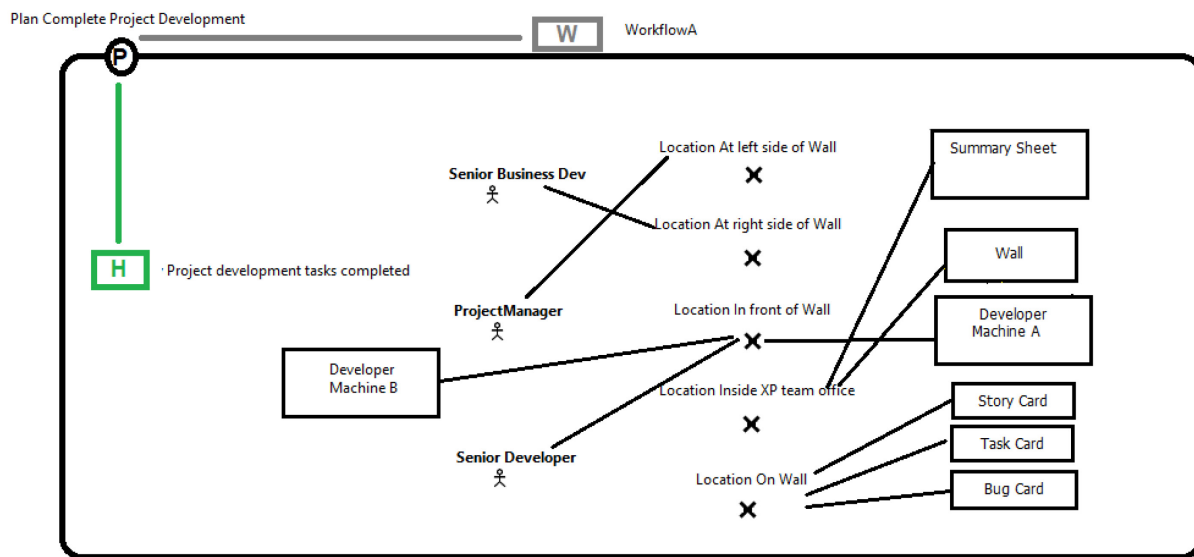


Figure 20. The Plan under execution in the Software XP team

Figure 21 provides the flow of activities for a Project Manager during the execution of the *Plan*. It begins with the *State*, i.e., “Current Project Development tasks” and proceeds towards a series of *Actions* linked together by using different *FlowLinks*, *And/Or Forks* and *Joins*, and *Sequence* elements. The links between these *MediatingArtefacts* and the *Actions* they contribute towards are also provided using the arrowed links. Initially, the Project Manager plans the work

for an iteration, followed by communicating the development tasks to the developers of the team, which contributes towards its *GoalState* to “Assign tasks to developers”. This is followed by management of the development activity by understanding the challenges faced by the developers and reassigning the development tasks if necessary. The development activity is continuously monitored. Based on the development activity, the *State* “Current iteration tasks remaining” is updated, which contributes to the final *GoalState* to “Ensure completion of dev tasks”. The Task and Bug cards contribute towards all *Actions* while the Wall provides a platform to communicate, manage, and monitor development tasks. Although the original XP team system does not discuss the use of *BodilySupport* by the *Actors* of the system, an instance for it, i.e., “Point At Card” is specified to ensure a more complete analysis of the system and visualization using the introduced notation. Hence, the ProjectManager uses “Point At Card” to communicate the development tasks to the developers of the team.

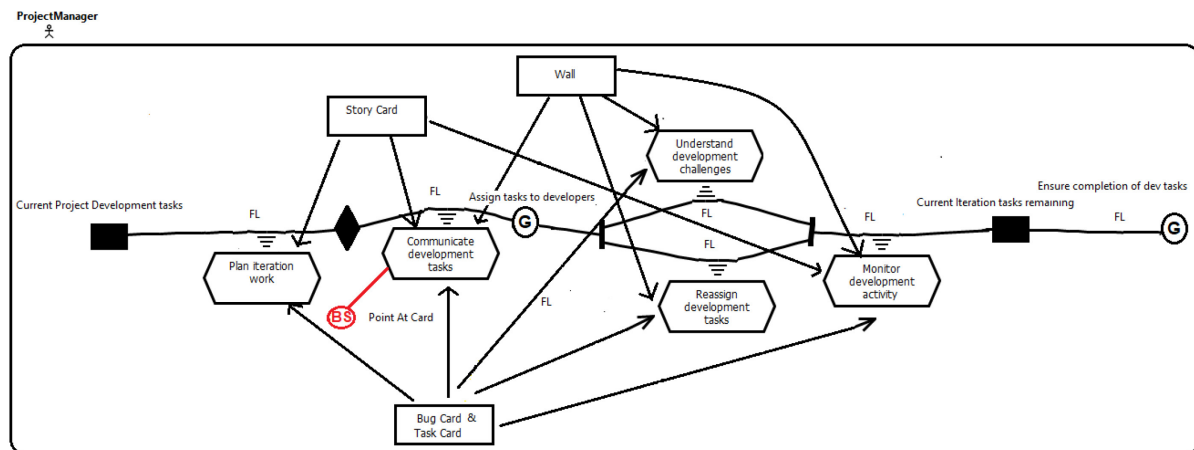


Figure 21. Flow of Actions for Project Manager in a Plan

Figure 22 provides the flow of activities for a Senior Business Dev (referred to as Business Dev from here on) during the execution of the *Plan*. The Business Dev begins by receiving the client’s project requirements that act as the initial *State* for it. This is followed by a sequence of *Actions* performed by the *Actor*. Firstly, the Actor understands the specified requirements, which

are then prioritized and provided to the Project Manager. These series of steps facilitate the *Actor* in achieving its *GoalState*, i.e., “Communicate high priority tasks”. For all Actions, the Business Dev requires the Summary Sheet.

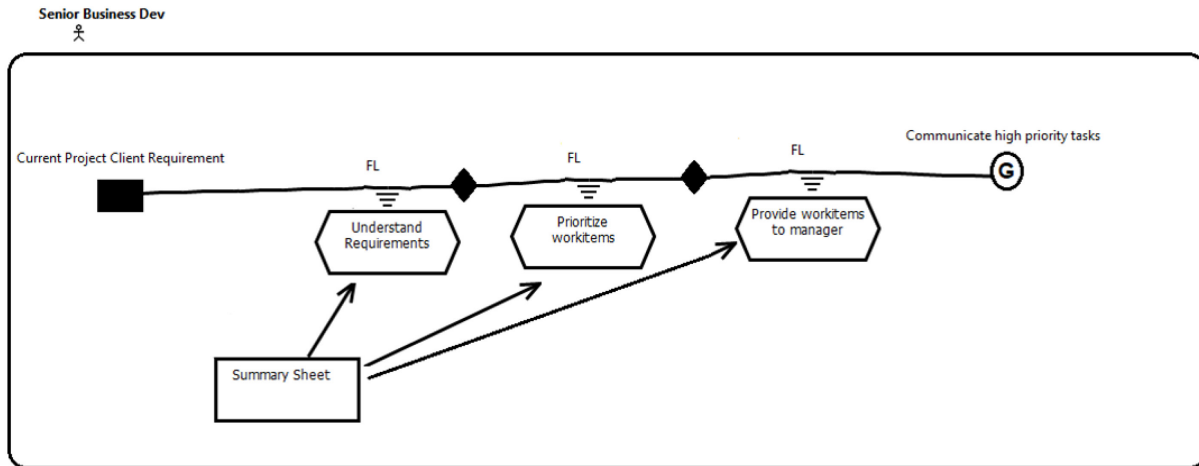


Figure 22. Flow of Actions for Senior Business Dev in a Plan

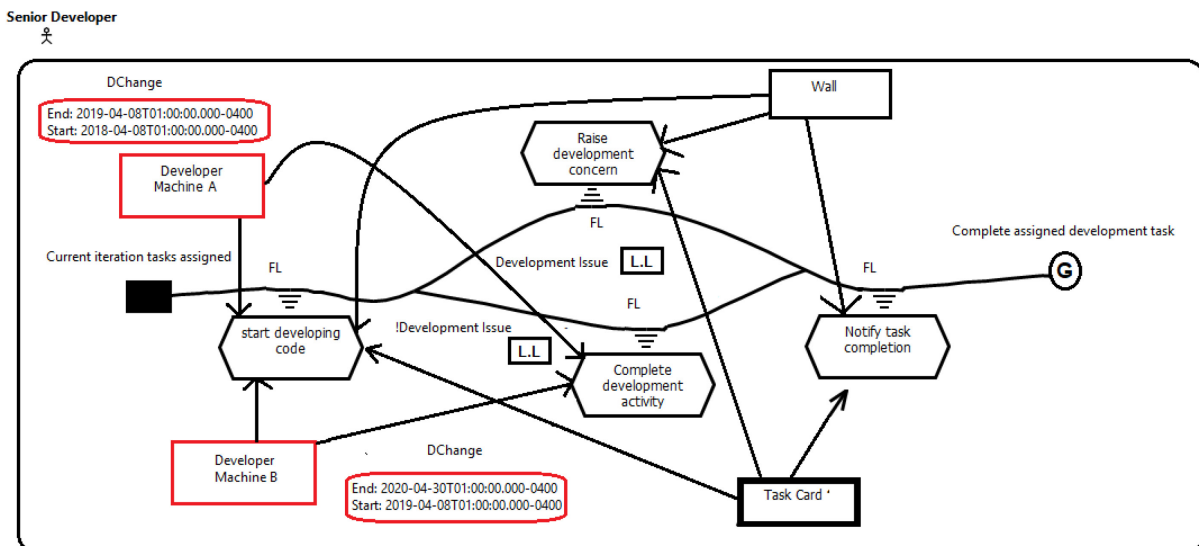


Figure 23. Flow of Actions for Senior Developer in a Plan

Figure 23 provides the flow of activities for a Senior Developer (referred to as Dev from here on) during the execution of the *Plan*. It begins with the *State*, i.e., “Current iteration tasks assigned” following which the *Action* “start developing code” is performed. The Dev might then encounter a development issue that needs to be addressed. Therefore, the *OrFork* has an

associated *LinkLabel*, i.e., “Development Issue”, providing it the ability to raise this concern. Otherwise, the Dev proceeds on the alternative path to complete the development task and notify its completion to the Project manager, achieving her *GoalState*, i.e., “Complete assigned development task”. The Dev uses her Developer Machine to complete her assigned development tasks, while the Wall provides the ability to clearly communicate her development concerns and notify the completion of her tasks. An important concept provided in Figure 23 is the *DeactivationChange*, which is not part of the original description of the Software XP team but is introduced here for completeness. The Dev use a particular developer machine (“Developer Machine A”) for completing her development activity but decides to replace her machine with a faster developer machine (“Developer Machine B”). The *Start* and *End* dates describe that duration and the variance in color of the elements differentiates them from regular elements.

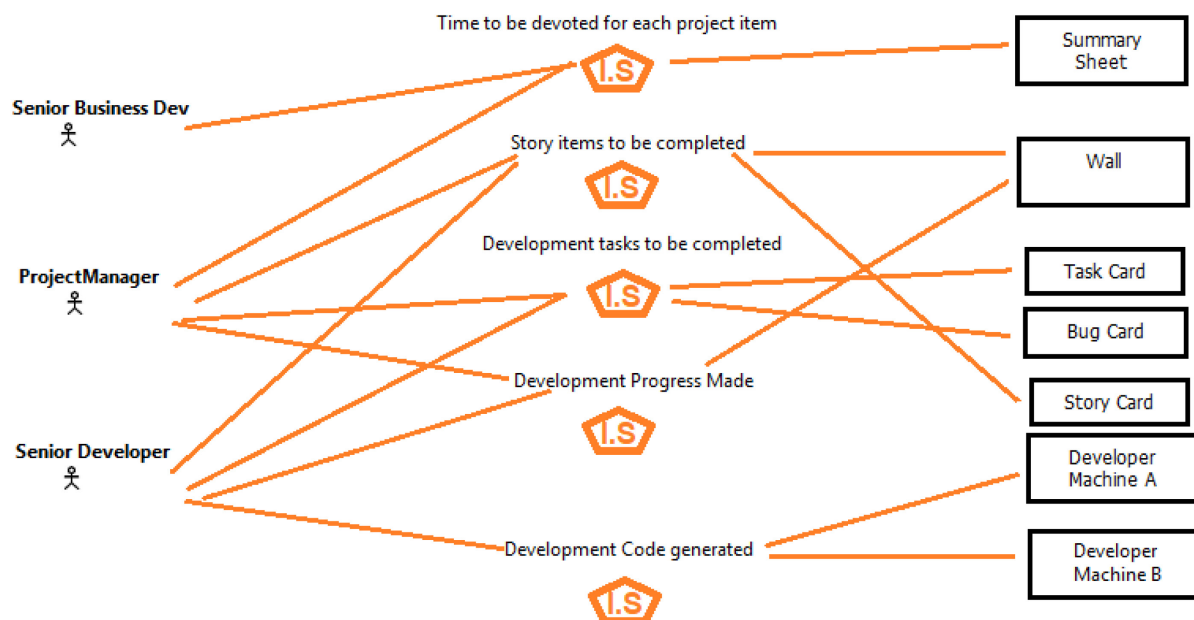


Figure 24. The Information contained inside the Software XP team System

Figure 24 describes the information that is contained inside the Software XP team system using the *Actors* and the *MediatingArtefacts*. The links between the *InformationStructure*

(referred to as information from here on) and the elements that contain them are also provided (links between the *Actor* and *InformationStructure* contribute towards the *SituationalAwareness* of the *Actor*). The information “Time to be devoted for each project item” is provided to the Business Devs and the Project Manager using the Summary Sheet. The Project Manager and the Dev have access to the information regarding “Story items to be completed”, “Development tasks to be completed”, and “Development Progress Made”, discussed using the Wall and Index cards (Story, Task, and Bug card). Finally, the Dev generates the information “Development Code generated” using her Developer Machines. It needs to be noted that the development of code, from the information of tasks to be completed, can act as an instance for *InformationStructureTransformation* caused by the *Action* of completing development activity by the Dev. Additionally, the communication between the Project Manager and the Dev regarding the progress of development can act as *InformationStructureExchange*. However, they are not used in the transformations, because the information structure is not covered by URN models. Consequently, these concepts do not have an associated symbol in this notation.

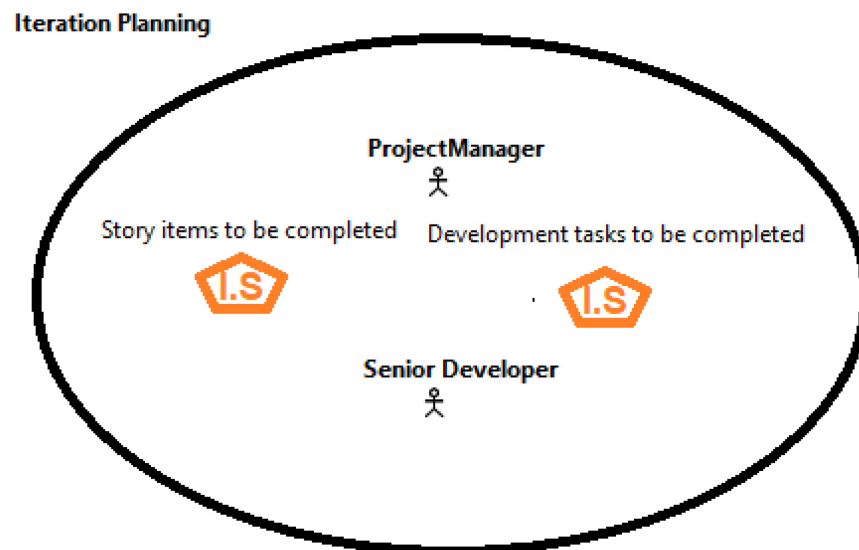


Figure 25. The Information Hub used in the Software XP team system

The Software XP team system also has several types of meetings amongst its team members to discuss various team requirements. These meetings can be considered as an *InformationHub* since they facilitate the flow of information in the system. Figure 25 describes the Iteration Planning meeting that involves the ProjectManager and Dev. It acts as an *InformationHub* for the Software XP team system and allows the ProjectManager and Devs to discuss the Story Items and development tasks that need to be completed in the current development iteration.

4.3 Implementation

This section provides an overview of the Atlas Transformation Language Tool, which is used to implement the transformations between DC and URN. It is followed by the actual transformation rules between the modeling notations. The section ends with a discussion on the steps that a modeler needs to follow to create an instance of a system using the DC metamodel and execute these transformations.

4.3.1 ATL (Atlas Transformation Language)

ATL [5] is a Model-to-Model transformation language that is used to transform a set of source models into target models. It is developed and maintained by AtlanMod [31] and OBEO [32]. ATL is a declarative-imperative hybrid, composed of a set of transformation rules that identify the matching of source model elements, followed by the creation and initialization of target model elements. Three types of transformation rules exist in ATL:

1. *Matched Rules (Declarative Programming)*: These are used to create elements of the target model by utilizing the information of elements of the source model. Each matched rule executes once for every matched source element and the elements are accessed using the Object Constraint Language (OCL) [24]. All rules written for implementing

DC-URN transformations are matched rules. A typical matched rule contains an input and output pattern. The input pattern consists of the “from” keyword, declaration of input variable, and an OCL expression that returns the input element to be transformed. The pattern can also contain a filter to identify a set of constraints. The output pattern contains one or more elements of the target model to which the matching source elements, from the input pattern, will be transformed. It consists of the keyword “to”, variable declaration, and initialization of target model elements. Figure 26 provides an example of a matched rule from the transformations between DC and URN, transforming the *DeactivationChange* class from the DC source model to the one from the URN target model. The initializations for the target elements is provided by using `<-` inside “to” between the target and source attributes (e.g., the *start* attribute from the DC source model is transformed to the *start* attribute of the URN target model).

```
-- Matched Rule

rule DCDeactChange2URNDeactChange
{
  from
  dchg:DC!DeactivationChange
  to
  udchg:urn!DeactivationChange(
    element <- dchg.element,
    start <- dchg.start,
    end <- dchg.end
  )
}
```

Figure 26. Matched Rule Example

2. *Called Rules (Imperative Programming)*: These types of rules need to be called explicitly in order to be executed. They can generate target model elements and need to be called from either a matched rule or another called rule. As a rule, the result of a

called rule is the last statement executed in the block. Called rules are not used in DC-URN transformations.

3. *Lazy Rules (Imperative Programming)*: These rules have a source model element as a parameter and require a source pattern to be matched while creating the target model element. Lazy rules are not used in DC-URN transformations.

An additional language construct that has been used extensively in specifying the transformations between DC and URN is the *distinct for-each* target pattern. This pattern can be used inside a matched rule to create a distinct target element for each element linked to a source model element. These distinct target elements can all be referenced by a single output variable. Although, lazy rules can also be used instead of the *distinct for-each* pattern for creating new elements, the usage of the pattern is extremely useful and convenient in DC-URN transformations. The top of Figure 27 provides an example of the *distinct for-each* pattern. It creates a distinct *Contribution* link for each input element present inside the collection “artefacts”. All elements can be referenced by using the single variable “c1”. Last but not least, ATL also has helpers which are either variables or functions used to implement code that can be reused. In DC-URN transformations, the helper functions are used to either create id increments or extract names of DC model elements. For example, the bottom of Figure 27 shows a helper method (to increment an Integer by one) used in specifying the DC-URN transformations.

```
-- Example of a distinct-foreach pattern
c1:distinct urn! Contribution foreach(e in as.artefacts)()
-- This helper increments the id for an element
helper context Integer def : inc() : Integer = self + 1;
```

Figure 27. Example of distinct for-each and Helper function

4.3.2 Transformation Rules

The tables provided in this section describe the transformations between DC and URN. Like the previous chapter, the transformation rules are described based on the five themes of DiCOT. In all tables, the first column provides the source DC element belonging to a theme, the second column list the target elements in URN (see Section 2.4), the third column describes the type of rule applied for implementing the transformation in ATL, and the final column provides a description of the transformation rules that are applied to these elements. While this section only provides a description of the transformation rules, the actual implementation, written in the ATL transformation language, is available at <https://github.com/JUCMNAV/DCtoURN>.

Table 3 describes the transformation between DC and URN for the elements *Actor*, *MediatingArtefact*, and *BodilySupport* of the DC Artefact theme (see Figure 13).

Table 3 Transformation for DC Artefact

DC (See Figure 13)	URN	Rule	DC to URN
Actor	Actor, ActorRef, Component, ComponentRef, URNLink, Metadata	Matched Rule	<ol style="list-style-type: none"> 1. A new Actor is created along with an ActorRef in GRL. The Intentional-Elements linked to an Actor are created using its associated Actions, MediatingArtefacts, and BodilySupport elements. 2. A new Component (Type – Actor) along with its ComponentRef is created in UCM. The path nodes bound to the Component are created using its associated Action and GoalState elements. 3. A URNLink is defined between the GRL Actor and the UCM Component. 4. The Metadata containing the Actor’s Location and Naturalness is also added to the Actor and Component.
MediatingArtefact	IntentionalElement, IntentionalElementRef, Contribution, LinkRef, Metadata	Matched Rule	<ol style="list-style-type: none"> 1. A new IntentionalElement (Type – Resource) along with its Intentional-ElementRef is created in GRL. 2. A Contribution link, and its associated Linkref, is created from the Intentional-

DC (See Figure 13)	URN	Rule	DC to URN
			<p>Eelement to the Action that uses it.</p> <p>3. The Metadata containing the MediatingArtefact's Location and Naturalness is also added to the IntentionalElement.</p>
BodilySupport	IntentionalElement, IntentionalElementRef, Contribution, LinkRef, Metadata	Matched Rule	<p>1. A new IntentionalElement (Type – Task) along with its Intentional-ElementRef is created in GRL.</p> <p>2. A Contribution link, and its associated Linkref, is created from the Intentional-Element to the Action that uses it.</p> <p>3. The Metadata containing its Location and Naturalness is also added to the IntentionalElement.</p>

For all elements transformed from DC to URN for the Artefact theme, the id element is specified for both main elements and their corresponding references. The internal path nodes, contribution links, and node connections for the transformed GRL Actors and UCM Components depend on their Actions, States, and GoalStates. Table 4 explains the transformation between DC and URN for the elements *Workflow*, *Plan*, *History*, *GoalState*, *State*, *FlowLink*, *AndFork/Join*, *ORFork/Join*, *LinkLabel*, and *Sequence* of the DC Workflow theme (see Figure 14).

Table 4 Transformation for DC Workflow

DC (See Figure 14)	URN	Rule	DC to URN
Workflow	UCMspec	Matched Rule	1. A new UCMspec is created for every Workflow element in DC model.
Plan	UCMmap	Matched Rule	<p>1. A new UCMmap is created for every Plan element in DC model.</p> <p>2. The GoalState, State, AndFork/Join, OrFork/Join form the path nodes of the UCMmap.</p> <p>3. The FlowLinks are mapped to the NodeConnections in the target UCMmap.</p>
History	URNLink, Metadata	Matched Rule	<p>1. A new URNLink (Typed – History) is created between the GRLGraph (containing all GRL Actors) and the StartPoints (for all DC Actors) on the UCMmap.</p> <p>2. A History element is added to the</p>

DC (See Figure 14)	URN	Rule	DC to URN
			Metadata of the UCM map.
GoalState	IntentionalElement, IntentionalElementRef, EndPoint, StartPoint- EndPoint pair, Connect, Contribution, LinkRef, URNLink	Matched Rule and <i>distinct for-each pattern</i>	<ol style="list-style-type: none"> 1. A new IntentionalElement (Type – Goal) along with its IntentionalElementRef is created in GRL. 2. The Contribution links and their respective references for the IntentionalElements are generated in GRL based on the Actions performed by individual Actors (linked with GoalStates) and the FlowLinks connecting the GoalStates. 3. An Action contributes towards either the intermediate or the ending GoalState based on the order of occurrence of the GoalState during the traversal of a path created using the FlowLinks. An Action occurring after a GoalState contributes to the next GoalState. 4. The inter-goal Contribution links and their respective references are also generated to reflect the contribution of intermediate goals towards the final GoalState. 5. An EndPoint is created in the UCMmap, for every GoalState inside the component created for its associated Actor. 6. An intermediate GoalState creates a Startpoint-Endpoint pair plus a Connect on the UCM. 7. The created EndPoints and Startpoint-Endpoint pairs are linked to the IntentionalElements of the GoalStates in GRL using the URNLinks.
State	IntentionalElement, IntentionalElementRef, StartPoint, StartPoint- EndPoint pair, Connect, Contribution, LinkRef, URNLink	Matched Rule	<ol style="list-style-type: none"> 1. A new IntentionalElement (Type – Indicator) along with its IntentionalElementRef is created in GRL. 2. A Contribution link, and its associated Linkref, is created for States in GRL based on the Actions performed by individual Actors and their GoalStates connected to the State using the FlowLinks. 3. A State contributes towards all the Actions and the GoalStates occurring between the current and the next State during the traversal of a path created by connecting the FlowLinks for a particular Actor. 4. A State without a preceding FlowLink is transformed into a StartPoint, while an

DC (See Figure 14)	URN	Rule	DC to URN
			<p>intermediate State create a Startpoint-Endpoint pair plus a Connect on the UCM.</p> <p>5. The StartPoints and the StartPoint-Endpoint pairs in UCM are linked to IntentionalElements in GRL using URNLinks.</p>
Action	IntentionalElement, IntentionalElementRef, Contribution, LinkRef, Responsibility, RespRef	Matched Rule and <i>distinct for-each</i> pattern	<p>1. A new IntentionalElement (Type – Task) along with its IntentionalElementRef is created in GRL.</p> <p>2. The MediatingArtefacts and BodilySupport used to perform an action are transformed into the Contribution links and their respective references for the IntentionalElements in GRL.</p> <p>3. An Action element is transformed into a Responsibility element along with its respective reference on the UCMmap.</p> <p>4. A Responsibility element is linked to the NodeConnections created using the FlowLinks that are associated with an Action.</p>
FlowLink	NodeConnection, URNLink	Matched Rule	<p>1. A FlowLink is transformed into NodeConnections on the UCMmap.</p> <p>2. The “from” and “to” elements act as the source and target elements for the NodeConnection.</p> <p>3. URNLinks are created between Responsibilities (in UCM) and Tasks (GRL) for FlowLinks with an associated Action.</p>
ORJoin	OrJoin	Matched Rule	1. An ORJoin is transformed to an OrJoin on UCMmap.
ORFork	OrFork	Matched Rule	1. An ORFork is transformed to an OrFork on UCMmap.
AndJoin	AndJoin	Matched Rule	1. An AndJoin is transformed to an AndJoin on UCMmap.
AndFork	AndFork	Matched Rule	1. An AndFork is transformed to an AndFork on UCMmap.
LinkLabel	Condition	Matched Rule	1. A LinkLabel is transformed into a Condition on the UCMmap. This Condition is defined on the NodeConnection created from the FlowLink linked with the LinkLabel.
Sequence	DirectionArrow	Matched Rule	1. A Sequence element is transformed into a DirectionArrow on the UCMmap.

The main benefit of using a *distinct for-each* for the transformations of *Action* and *GoalState* elements is to create unique contribution links for each element to which they are linked. For

example, an Action can be performed by using different *MediatingArtefacts* and *BodilySupport* elements. Therefore, these elements are contributing towards the fulfillment of an Action and have an associated contribution link towards it in the transformed GRL. The *distinct for-each* links the tasks to the respective references for these elements, which avoids the creation of duplicate elements in the GRL diagram.

Table 5 describes the transformation between DC and URN for only the elements *Location* and *Naturalness* of the DC Physical Layout theme (see Figure 15). The other elements belonging to the DC Physical Layout theme are covered in other transformation rules tables.

Table 5 Transformation for DC Physical Layout

DC (see Figure 15)	URN	Rule	DC to URN
Location	Metadata	Matched Rule	1. A Location element is added to the Metadata of associated Actor, MediatingArtefacts, and BodilySupport elements.
Naturalness	Metadata	Matched Rule	1. A Naturalness element is added to the Metadata of associated Actor, MediatingArtefacts, and BodilySupport elements.

Table 6 describes the transformation between DC and URN for the elements *InformationHub* and *Buffer* of the DC Information Flow theme (see Figure 17). Although the elements *InformationStructure*, *InformationStructureTransformation*, and *InformationStructure-Exchange* are also specified in the table, they are not transformed to any element in URN due to the lack of an appropriate element for them in the target notation.

Table 6 Transformation for DC Information Flow

DC (see Figure 17)	URN	Rule	DC to URN
InformationStructure	None	None	None
InformationStructure-Exchange	None	None	None

DC (see Figure 17)	URN	Rule	DC to URN
InformationStructure-Transformation	None	None	None
InformationHub	UCM Component, ComponentRef, UCMmap	Matched Rule	<ol style="list-style-type: none"> 1. An InformationHub element is transformed into a Component (Type – Team), and its respective references, on a separate UCMmap. 2. A Resource used by the InformationHub is mapped to a Component inside the Component created from the InformationHub.
Buffer	Metadata	Matched Rule	<ol style="list-style-type: none"> 1. A Buffer element is added to the Metadata of its associated InformationHub.

Some DC elements are mapped to Metadata in URN (e.g., Buffer), while other elements are not mapped at all (e.g., InformationStructure). The major reason behind transforming a few DC elements to Metadata or not providing a transformation at all is that some DC elements do not have an appropriate target element in URN. Hence, there is no place in the URN model to attach Metadata that could capture the DC element. This is a major benefit for specifying transformations between DC and URN in that it provides the ability to strengthen both notations by combining capabilities that might be missing in either one of them. This is discussed in greater detail in Section 4.5.

Table 7 describes the transformation between DC and URN for the elements *DeactivationChange*, *LinkDeactivationChange*, *PropertyChange*, *FormulaChange*, and *ValueChange* of the DC Evolution theme (see Figure 18). All transformations (except for *ValueChange*) between DC and URN elements to capture the evolution of a system are essentially one-to-one mappings between the notations. The major reason behind this is that the elements in the DC metamodel for evolution were transcribed from TimedURN concepts [3][4] as explained in Section 3.4.

Table 7 Transformation for DC Evolution

DC (see Figure 18)	URN	Rule	DC to URN
DeactivationChange	DeactivationChange	Matched Rule	1. All the attributes of DeactivationChange in DC are mapped to the corresponding attributes of DeactivationChange in URN.
LinkDeactivation-Change	LinkDeactivation-Change	Matched Rule	1. All the attributes of LinkDeactivationChange in DC are mapped to the corresponding attributes of LinkDeactivationChange in URN.
PropertyChange	PropertyChange	Matched Rule	1. All the attributes of PropertyChange in DC are mapped to the corresponding attributes of PropertyChange in URN.
FormulaChange	FormulaChange	Matched Rule	1. All the attributes of FormulaChange in DC are mapped to the corresponding attributes of FormulaChange in URN.
ValueChange	BooleanChange, TextChange, EnumerationChange	Matched Rule	1. All the attributes of ValueChange in DC are mapped to the corresponding attributes of BooleanChange, StringChange, or EnumerationChange in URN based on the type of value specified for the newValue attribute of ValueChange.

Table 8 describes the transformations occurring between the DC and URN for element DCSpec, which is essentially the root class of the DC metamodel (see Figure 19).

Table 8 Transformation for DC MetaClasses

DC (see Figure 19)	URN	Rule	DC to URN
DCSpec	URNspec, URNdefinition, GRLspec, GRLGraph, DynamicContext, DynamicContextGroup	Matched Rule	<ol style="list-style-type: none"> 1. A new URNdefinition is created along with URNspec, GRLspec, GRLGraph, DynamicContext, and DynamicContextGroup. 2. The GRLGraph and UCMmaps (for Plan and InformationHub) created in URN are added to URNdefinition. 3. Actors, Intentional Elements, and Element Links created in URN are added to the GRLspec. 4. The path nodes and NodeConnections created in URN for IntentionalElementRefs, Beliefs, ActorRefs, LinkRefs, and BeliefLinks

DC (see Figure 19)	URN	Rule	DC to URN
			<p>are added to the GRLGraph.</p> <p>5. The DeactivationChange, LinkDeactivationChange, PropertyChange, FormulaChange, BooleanChange, TextChange and EnumerationChange elements are added to DynamicContext. This is further contained inside a DynamicContextGroup.</p> <p>6. The URNdefinition, GRLspec, UCMspec, DynamicContext, and DynamicContextGroup, along with the URNLinks created between GRL and UCM elements are added to the URNspec.</p>

4.3.3 Steps for DC Model Creation and Transformation to URN

This section describes the steps that need to be followed by a DC modeler to analyze a system using DC and generate its corresponding .jucm file (URN model). The system used for implementing these steps is the Eclipse IDE. The explanation of the implementation steps is provided based on the folder structure used for creating the original transformations (<https://github.com/JUCMNAV/DCtoURN>) and assumes that the necessary ATL and EMF plugins are installed in Eclipse. Also, since the latest jUCMNav plugin for Eclipse does not contain the TimedURN concepts, the modeler will need to download and execute the jUCMNav project which can be found at <https://github.com/JUCMNAV/seg.jUCMNav>. The implementation steps are as follows:

1. Create the Domain Model for the target system. For the Software XP team, the Domain Model is provided in Figure 16. A modeler may use any class diagram modeling tool based on her preference. The domain model for the Software XP team is modeled with the TouchCORE [33] tool.

2. Create a new General project in Eclipse. Import the ECORE file specifying the metamodel for DC (<https://github.com/JUCMNAV/DCtoURN>). Additionally, import the file that contains the metamodel specification for the output extension of the Domain Model. For example, TouchCORE generates a .RAM file for the models specified in it. Therefore, for this system, the RAM ECORE file specifying the metamodel for RAM (Reusable Aspects Models) and the CORE ECORE file (to satisfy RAM metamodel dependency) needs to be imported in the project.
3. Generate a generator model for this project by navigating to the menu and clicking File→New→Other and select EMF Generator Model and select the project from the workspace. Provide an appropriate name for the model followed by a .genmodel extension. Once the .genmodel file is created for the system, open the file and generate the edit, editor, and test code by selecting Generator→Generate All in the menu options.
4. Once the editor files are generated, right click the .edit folder and Run As→Eclipse Application. This opens an editor in the Runtime Eclipse Environment that can be used to specify the instance model for the target DC system. It generates the corresponding instance using a tree-based editor.
5. To create a new instance for the system with this editor, create a new file (Type – DC) and specify the main containment element as Spec. Once the new file with the Spec element is created, right click on the file and select import resource. This can be used to import the domain model file generated for the target system. Finally, create the instance of the target system and save it. The created instance is saved as a .dc file.
6. To run the transformations, create a new project folder inside the main Eclipse application (not in the Runtime Eclipse Environment) and import the folders provided on

the Github link (preferably using the same folder hierarchy as used in the original transformations). It is important to import all the folders since they are arranged in a manner that satisfies the necessary inter-metamodel dependencies. Now, import the generated .dc file into the input folder of the project. Open the ATL folder and right-click the transformations file (TransformDC2URN.atl). Select ATL transformations which opens the configuration specifications for executing the transformations (these need to be specified only once and are reused for later executions). Provide the path of the DC.ecore file for the DC metamodel as the source and the urn.ecore for the URN metamodel as the target. Also provide the source model as the .dc file generated for the target system. The target model can be appropriately named with a .jucm extension. Click Run to execute the transformations. The output .jucm file is generated in the target location.

7. To view the output .jucm file, import the jUCMNav project and run it as an Eclipse application. This opens a jUCMNav editor where the generated .jucm file can be viewed and additional information can be added to the URN models.

4.4 Generated URN Models for Software XP Team

Figure 28 describes the GRL model generated for the Business Dev of a Software XP team. It corresponds to Figure 22 and details the goals of the Business Dev, i.e., “Communicate high priority tasks” along with the actions contributing towards achieving that goal like “Understand Requirements”.

The “Summary Sheet” acts as a useful *Resource* that the Business Dev utilizes to execute its actions while “Current Project Client Requirement” are the number of project deliverables that

can act as a KPI value that contributes towards achieving the Business Dev's goals and tasks.

URN links are defined for the goals and tasks in the GRL models from the appropriate elements (*Endpoints* for goals and *Responsibilities* for tasks) present in the corresponding UCM map for the Business Dev.

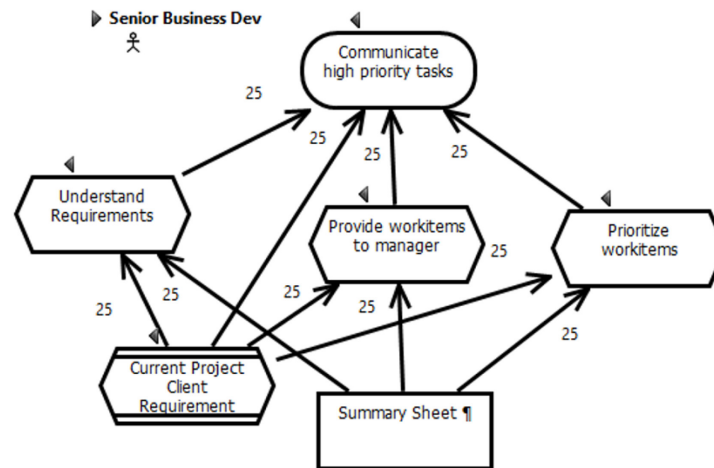


Figure 28. Generated GRL Model for Senior Business Dev

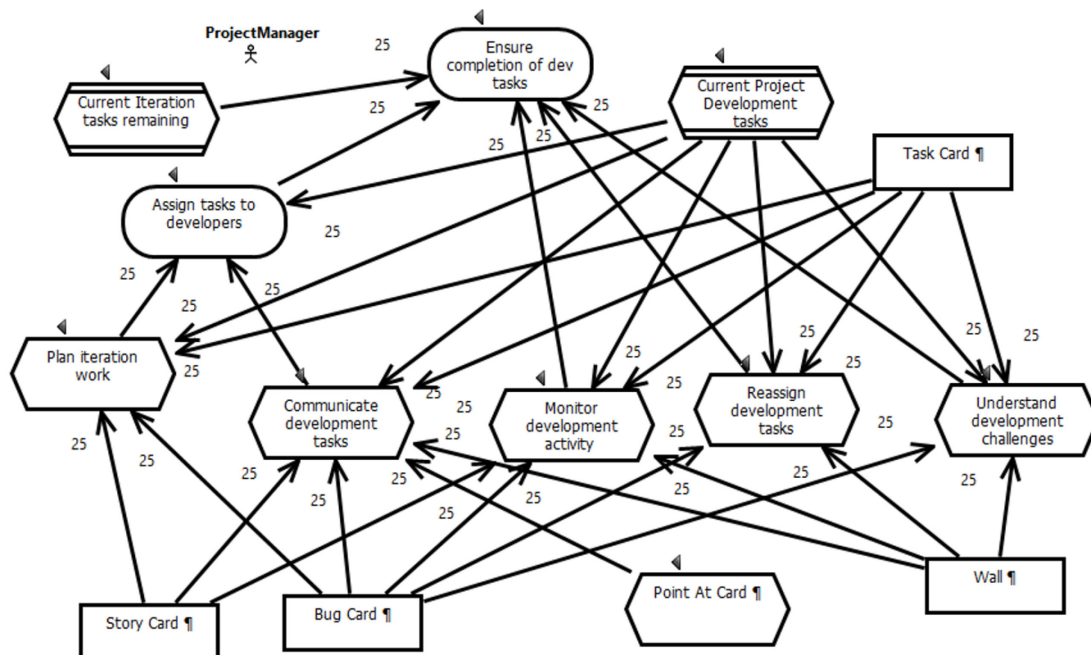


Figure 29. Generated GRL Model for Project Manager

Figure 29 describes the GRL model generated for the Project Manager of a Software XP team. It corresponds to the DC instance described in Figure 21. The Project Manager has the followings goals: “Ensure completion of dev tasks” and “Assign tasks to developers”. To implement these goals, the Project Manager performs different actions like “Plan iteration work” and “Monitor development activity”. The index cards (Story, Task and Bug cards) and the Wall help the Project Manager achieve its goals. Since “Assign tasks to developers” is an intermediate goal occurring before the final goal (i.e., “Ensure completion of dev tasks”) (see Figure 21), actions like “Plan iteration work” and “Communicate development tasks” only contribute to it, while the actions that occur after it (e.g., “Monitor development activity”) contribute only to the final goal. Additionally, “Assign tasks to developers” contributes towards the achievement of “Ensure completion of dev tasks”. The project manager can also point at the index cards to communicate the development tasks to the developers and support its activities. KPI values like “Current Iteration tasks remaining” and “Current Project Development tasks” provide the number of tasks remaining and number of tasks originally planned, respectively, and contribute towards the Project Manager’s goals and tasks directly or indirectly. “Current Project Development tasks” essentially contributes towards all the Project Manager’s actions (like “Plan iteration work” and “Monitor development activity”) and goal (“Assign tasks to developers”) until the intermediate state “Current Iteration tasks remaining” (see Figure 21). “Current Iteration tasks remaining” follows a similar trend until the final goal of the Project Manager is achieved.

Figure 30 describes the GRL model generated for the Dev of a Software XP team. It corresponds to the DC instance described in Figure 23. The Dev primary goal is to “Complete assigned development task”. The Dev performs actions like “Start developing code” and “Raise development concern” to achieve them. “Current iteration tasks assigned” acts as the KPI value

that helps the Dev achieve its goal and tasks. Different system resources like “Wall” and “Developer Machine” are used by the Dev to implement its actions.

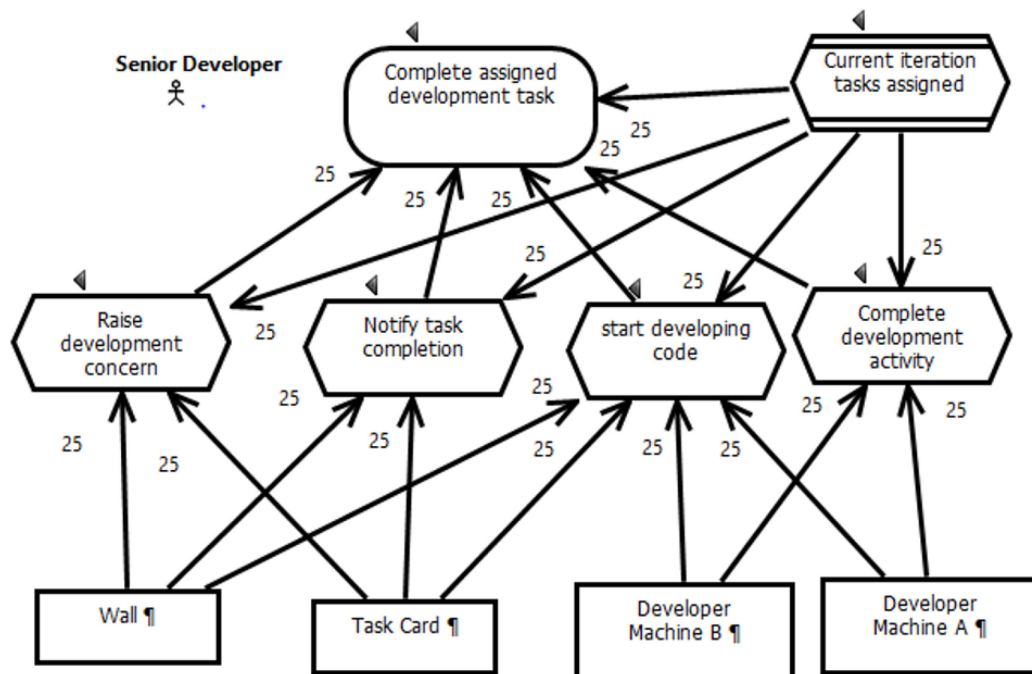


Figure 30. Generated GRL Model for Senior Developer

Figure 31, Figure 32, and Figure 33 describe the UCM models for the Senior Business Dev, Project Manager, and Senior Developer of the Software XP team system, respectively, that correspond to their DC instances described in Figure 22, Figure 21, and Figure 23, respectively. The models are collectively contained inside the final UCM model called “Complete Project Development”. The flow of activities in the generated UCM models are almost like the DC instances described previously, with DC elements replaced with their corresponding target URN element (excluding elements like *MediatingArtefacts* since they are only transformed to GRL elements).

The States like “Current Project Client Requirement”, “Current Project Development tasks” and “Current iteration tasks assigned” belonging to the Senior Business Dev, Project Manager

and Devs are transformed into the starting points for the UCM models for each of these DC *Actors* (transformed into UCM *Components*). The Actions performed by these *Actors* like “Start developing code” and “Plan iteration work” are transformed into UCM *Responsibility* linked to the UCM *NodeConnections* generated from the *Flowlinks* in the source runtime instance. The final *GoalStates* (based on the path generated by connecting the *FlowLinks*) like “Complete assigned development task” and “Ensure completion of dev tasks” are transformed into the end points for the UCM models. *URNLinks* are defined to connect the *Responsibilities* and *EndPoints* in the UCM map to the *Tasks* and *Goals* in the corresponding GRL model.

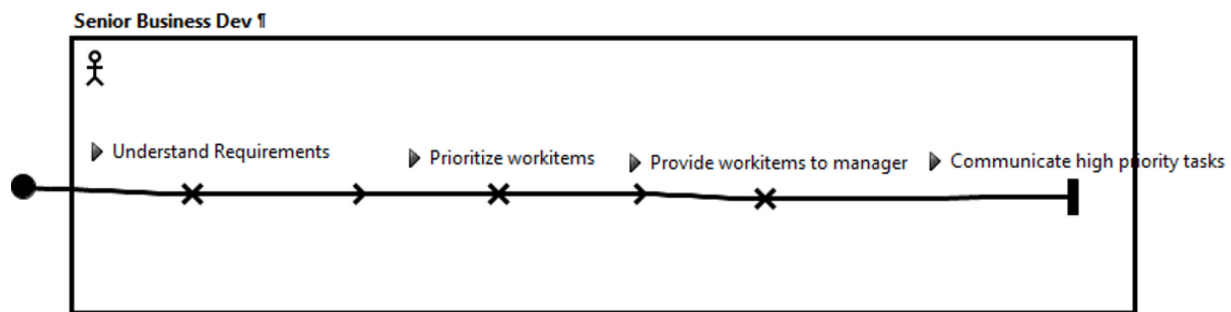


Figure 31. Generated UCM Model for Senior Business Dev

In Figure 32, “Current Iteration tasks remaining” and “Assign tasks to developers” are StartPoint-EndPoint pairs that have been transformed from an existing intermediate *State* and *GoalState* in the original DC instance, respectively.

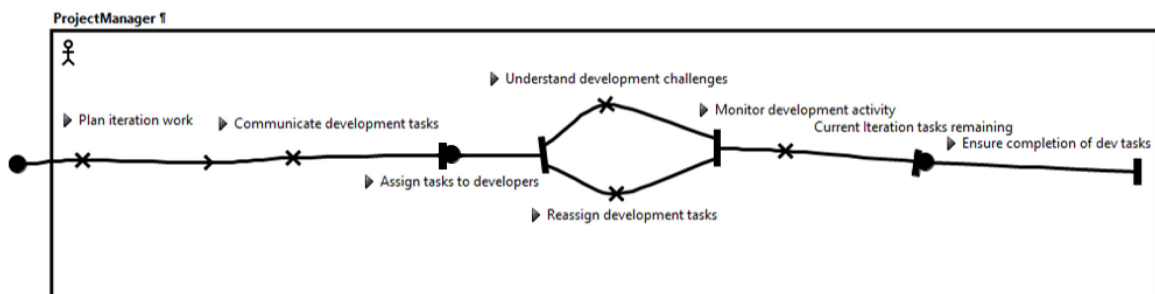


Figure 32. Generated UCM Model for Project Manager

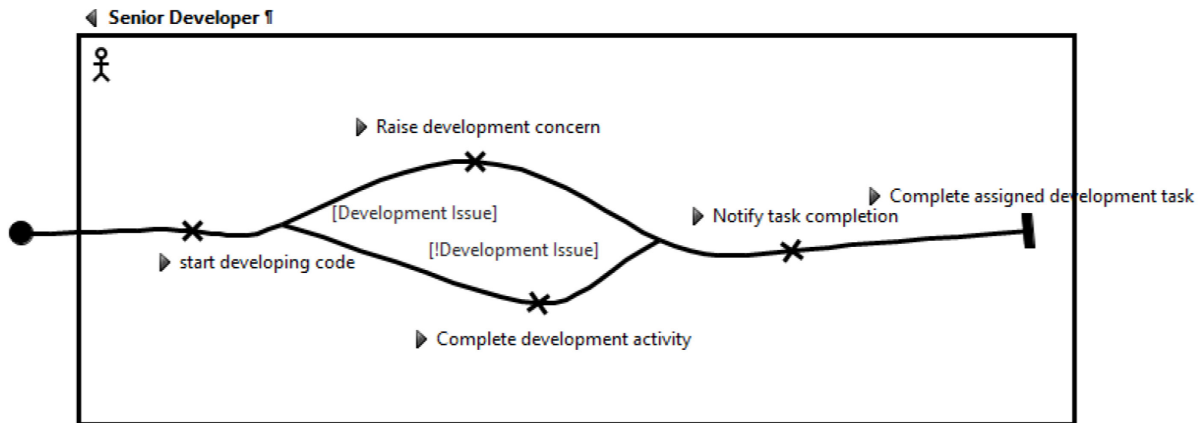


Figure 33. Generated UCM Model for Senior Developer

Figure 34 describes another UCM map that is generated for “Iteration Planning” that acts as an *InformationHub* in the original DC instance (see Figure 25). It contains the components for Project Manager and Dev that participate in the meeting and facilitate the flow of information.

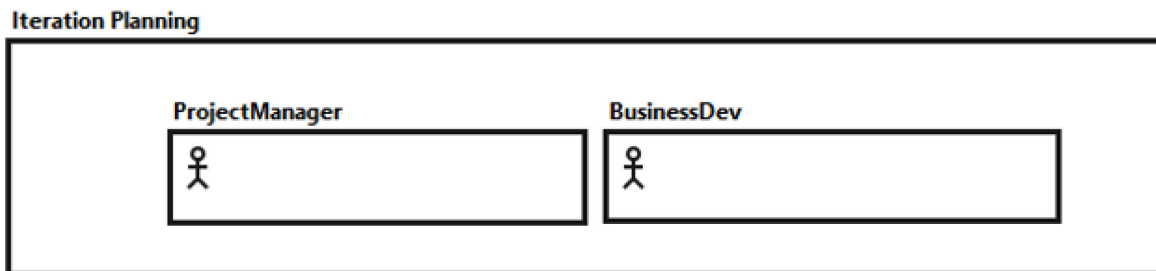


Figure 34. The UCM Model for Iteration Panning in a Software XP team system

In Figure 23, the Dev uses different developer machines (based on the specified duration) for completing her development activity. Since this change is specified using a *DeactivationChange*, it is transformed to a URN *DeactivationChange* and added to the *DynamicContext* and *DynamicContextGroup* present inside the specification of the transformed URN models. Similarly, other elements (except *ValueChange*) that describe the evolution of a DC instance have a one-to-mapping to TimedURN (see Figure 12) elements. The jUCMNav tool does not visualize these concepts in a graphical way, hence they are not visible in the transformed URN models.

URNLinks (of type – History) are defined between the GRLGraph (containing all GRL Actors) and the start points for each DC Actor on the UCM map to describe the *History* “Project development tasks completed” of the Software XP team system. Additionally, the value “Project development tasks completed” is added to the “_history” metadata of the UCM map. Finally, elements like Project Managers and the index cards (essentially *Resources* in the source DC instance) contain metadata to describe their respective locations in the system and the element’s naturalness. For example, the UCM Component for the Project Manager contains the metadata “_location” and “_naturalness” with the values “At left side of Wall” and “repManager/High”, respectively. The first value is essentially the *Location* of the Project Manager described in the DC instance (see Figure 20), while the second value implies that the closeness between the Project Manager in the real world, described using the domain model (see Figure 16), and its representation in the instance model described using the DC notation is “High”.

4.5 Benefits of Integrating DC and URN

Besides the obvious advantages of providing the semantics for the DC language and the evaluation of DC models with URN evaluation and traversal mechanisms, the integration of DC and URN has many other benefits for both notations. A major contribution by URN to this integrated approach is its ability to provide more fine-grained contribution values for contributing elements which can then be compared through trade-off analysis to determine how well design alternatives meet important stakeholder goals. For example in Figure 29, all actions and used resources contribute equally towards the completion of a Project Manager’s goals, because they all have a contribution of 25 towards their target elements. This happens because the source DC instance does not provide the ability to specify detailed contributions of respective actions towards the completion of an *Actor’s* goals. Therefore, its transformed GRL models do

not capture this information. However, using DC in conjunction with URN allows a modeler to use DC to analyze its target system and then transform it into URN goal and scenario model. The modeler can then add the more detailed information regarding contribution values of goal model elements. Additionally, the modeler can also specify importance values of intentional elements and GRL actors as well as the KPI values in the transformed GRL models. This added information can then be used to analyze different design alternatives for their impact on stakeholders using URN analysis techniques such as the GRL evaluation mechanism by specifying a GRL strategy (see Section 2.3.1). Furthermore, a modeler can also add Timepoints to the generated URN models to perform a comprehensive analysis of a set of changes to a system within concrete time intervals. Thus, it allows a modeler to make a more informed decision regarding their system design. In addition, the resulting UCM scenario models from the DC transformation can be analyzed with the UCM traversal mechanism by specifying scenario definitions, hence creating a test suite for the UCM model which is not available in DC. Furthermore, scenario definitions allow key scenarios to be highlighted for improved understanding which is also not supported by DC.

A major contribution of DC towards the integrated approach is its ability to analyze the flow of information in the system. In DC, the significance of system information is quite high since it is central to the distribution of cognition in the system. Therefore, the DC language provides specific elements that can capture various aspects of information like what information exists in the system, the *Actors* and *MediatingArtefacts* sharing this information, and the type of information transformations occurring as a result of *Actions* performed by different *Actors*. However, URN does not provide the ability to explicitly capture the information flowing in the

system. Therefore, essentially a combined DC/URN methodology provides the modeler the ability to perform a more complete analysis of the target system.

4.6 Summary

This chapter discusses the transformation rules that are used for implementing the transformation between DC and URN. First, a DC notation is introduced that provides the ability to visualize a DC instance model. Then, the DC notation is applied to the Software XP team system to describe a typical instance model defined using the DC language. The transformation rules between DC and URN are then provided along with an overview of the ATL tool (used to implement the transformations) and a discussion on the steps that a modeler needs to analyze a system using the DC language. This is followed with a description of the transformed URN models for the DC instance. Finally, a discussion on the benefits of integrating DC and URN is provided. In the next chapter, the proposed DC language is utilized to analyze an Airplane Cockpit system to demonstrate the proposed methodology and its feasibility.

CHAPTER 5: DC-Based Case Study

This chapter describes a case study that is performed to assess the adequacy and feasibility of the proposed DC language in analyzing real world socio-technical systems. The case study is performed on an Airplane Cockpit system, which has been analyzed using DC [16]. As explained in the Chapter 1, the Airplane Cockpit system is taken from one of the most influential papers from the DC literature. First, an overview of the Airplane Cockpit system is provided. It is followed by the specification of a domain model for this system. A DC instance model for the Airplane Cockpit system is then created by using the concepts of the DC metamodel and described using the DC notation from Chapter 4. Finally, the URN models generated by transforming the DC instance to URN goal and scenario models along with the steps performed to validate these models are discussed.

5.1 The Airplane Cockpit System

The Airplane Cockpit system [16] considered for this case study is based on the data from an actual airline flight crew performing a full mission simulation of a flight from Sacramento to Los Angeles. The simulated aircraft is about 8 minutes out of Sacramento and is climbing through 19,000 feet toward their cruise altitude of 33,000 feet (provided in its flight plan). For this simulation, the crew consists of the following three: Captain, First Officer, and Second Officer. Each crew member has a specific duty assigned to them. The Captain (referred to as Capt from here on) performs several activities like communicating with the traffic controllers, flying the plane, monitoring the activities of the crew members, and checking the flight plan to decide the course of action etc. The First Officer (referred to as FO from here on) primarily assists the Capt in her activities and performs activities like flying the plane and setting the Altitude alert system

etc. The Second Officer (referred to as SO from here on) handles the engine thrust settings and performs additional housekeeping tasks. All crew members monitor the Air Traffic Control (ATC) frequency.

The objective of the crew, as mentioned in the airplane's flight plan, is getting the required cruising clearances and to ascend from its current altitude of 19,000 feet towards a cruise altitude of 33,000 feet. The ATC Controller and High-Altitude Controller are responsible for providing the required clearances for the aircraft to cruise to a higher altitude. In this flight simulation, the members are assumed to be communicating with the Oakland, California Air Route Traffic Control Center. The airplane has a currently cleared altitude of 23,000 feet and requires the clearances from the controllers to achieve its target cruise altitude.

To achieve their objectives, the crew members use several different tools. This impacts both the type and flow of information present in the Airplane Cockpit system. All crew members have separate workstations. Since the functionalities of the Capt and FO are almost similar, there was a trend in civil aircrafts to provide duplicate flying instruments that are mechanically linked (e.g., mechanically linked control yokes) to permit either pilot to fly the airplane and provide a measure of redundancy in the event that the instruments on one side fail. However, the cockpit system under analysis provides separate workstations, with completely separate sidestick controllers and interfaces for the Flight Management Computer System (FMCS). The airplane's altimeter specifies the current altitude of the aircraft while the altitude-alert system gives the information regarding the current altitude to which the plane is cleared to fly. An important tool used by the SO is the engine performance table. This table provides the engine performance ratio (EPR) values of the aircraft that it uses to achieve its goal of achieving maximum engine thrust. All crew members can use paper slips provided on their workstations to complete their

housekeeping tasks. Furthermore, they all utilize the radio to either communicate or listen to the instructions provided by the ATC. Finally, the controllers communicate with the crew using the radio. Other tools like the navigation chart do not directly impact the crew's objectives.

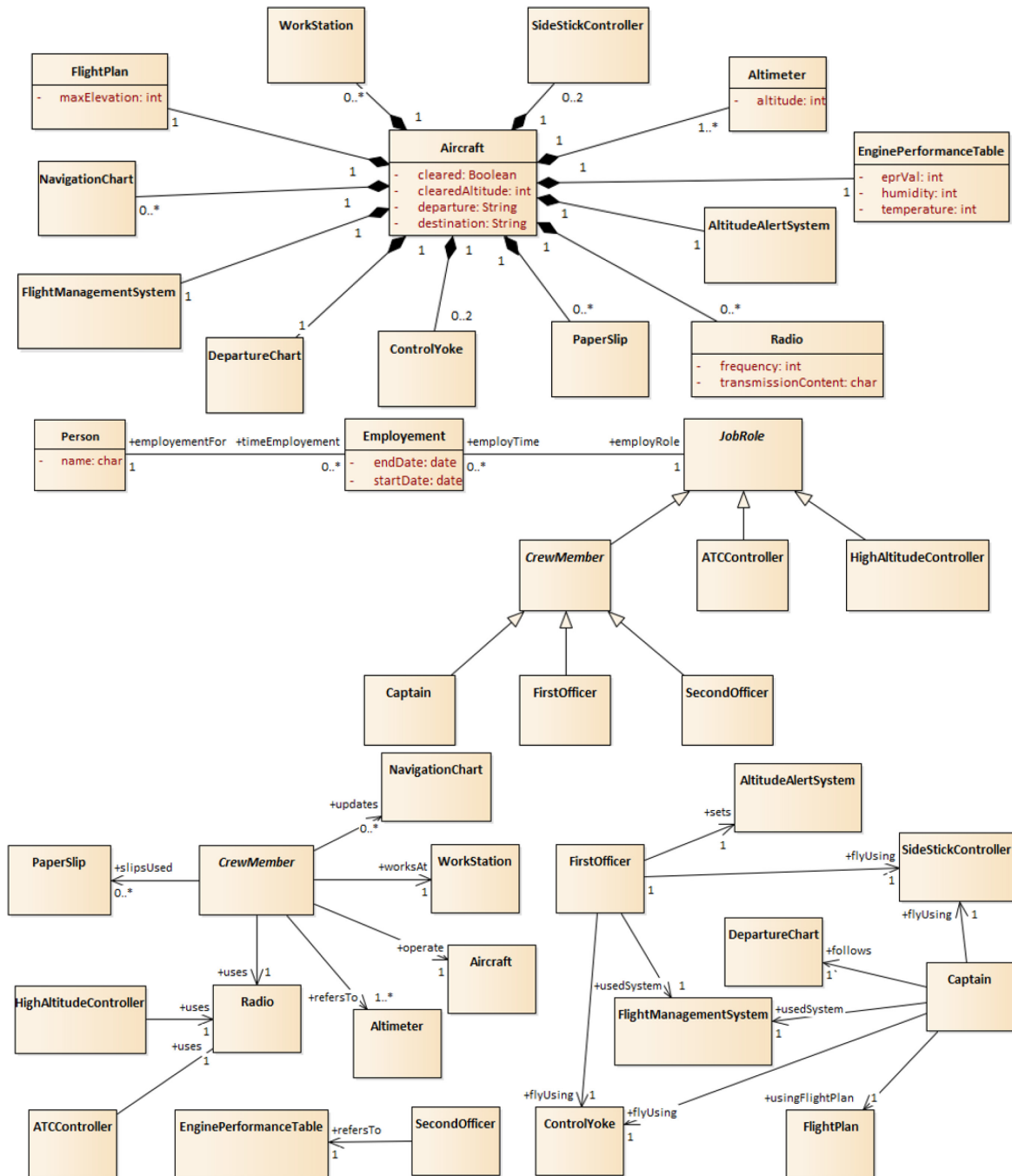


Figure 35. Domain Model for an Airplane Cockpit System

5.2 Domain Model for an Airplane Cockpit System

Figure 35 describes the domain model for The Airplane Cockpit System. It contains the elements for the persons playing the role of Crew Members of the airplane (Capt, FO, and SO) which are generalized into an abstract class called *CrewMember* along with a *Person* class detailing these persons. Since the members are effectively an employee of the system, they have an associated *Employment* class that specifies the details of their employment tenure and is additionally linked to the abstract class *JobRole*. The other persons in the system are the ATC Controller and the High-Altitude Controller, and all exist as a specialization of the *JobRole* class.

The system has different tools that are utilized to accomplish several different goals and objectives. All the necessary tools, used by the main actors of the system, are represented by a class in the domain model along with the necessary attributes. For example, the *EnginePerformanceTable* contains the information regarding the aircraft's EPR values along with the temperature and humidity corresponding to these values. All tools are contained inside the *Aircraft*, which has an associated cleared and cruising altitude along with a designated departure and destination location.

As mentioned in the previous section, the accomplishment of several different objectives by the crew members and the controllers is done by utilizing several different tools. All members can use *PaperSlips* and the *Radio*, which is also used by the controllers. The crew members have designated *Workstations*, update the *NavigationCharts*, and can refer to the *Altimeter* inside the *Aircraft* whenever required. The SO refers to the *EnginePerformanceTable* for setting the engine thrust force. The FO sets the *AltitudeAlertSystem*. The Capt utilizes the *FlightPlan* and follows the *DepartureChart* to decide the set of activities required to fly the plane and reach the

destination. Both Capt and FO use the FlightManagementSystem and the SideStickController or the ControlYoke depending on the aircraft model.

5.3 Application of DC Notation on the Airplane Cockpit System

The main *Actors* in the Airplane Cockpit System are the Captain, First Officer, Second Officer, ATC Controller, and the High-Altitude Controller. Figure 36 and Figure 37 describes the *Plan* under execution in the Airplane Cockpit to increase the altitude of the flight. It belongs to “WorkflowA” and contains the five *Actors* and their corresponding *Locations* in the system. The *MediatingArtefacts* used by the *Actors* to implement their *Actions* include the following: Flight Plan, Altimeter, Altitude Alert System, Navigation Chart, Departure Chart, Paper Slips, Engine Performance table and the Aircraft. Besides these, the FO and Capt have their dedicated FMCS (FMCS_B and FMCS_A), sidestick controllers (SideStickB and SideStickA), and control yokes (ControlYokeB and ControlYokeA). All crew members have dedicated instances for their assigned WorkStations. All *Actors* have elements to visualize their respective Radios.

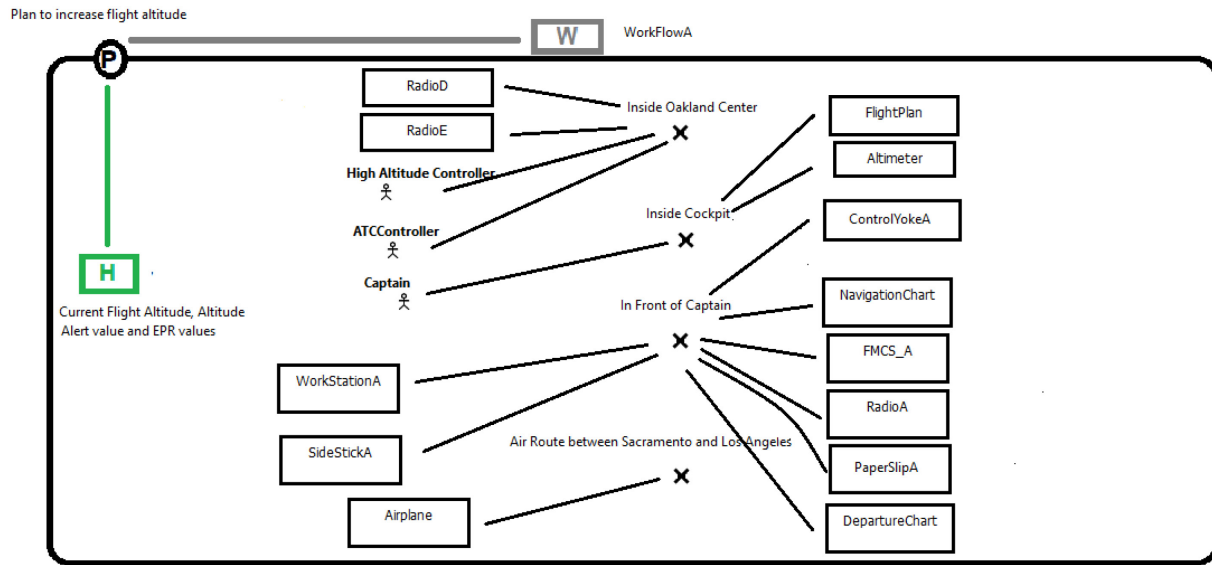


Figure 36. The Plan under execution in the Airplane Cockpit System (i)

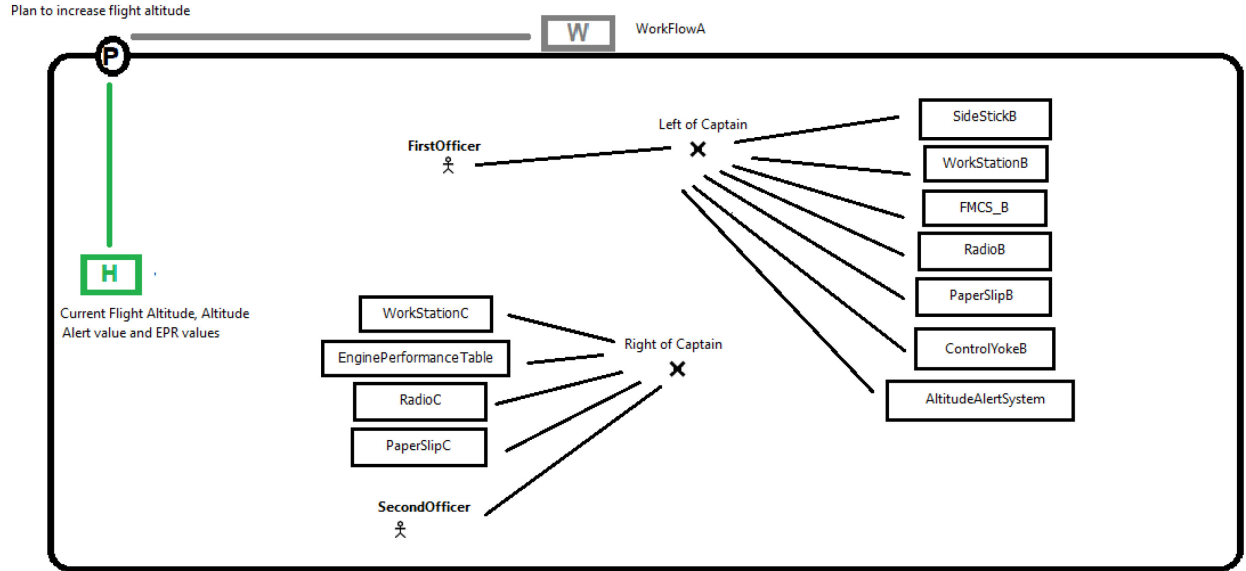


Figure 37. The Plan under execution in the Airplane Cockpit System (ii)

Like the system *Actors*, *MediatingArtefacts* have been linked to their respective *Locations* in the system. An important point to consider is that the definition of instances for the *Location* of an element is done by considering the *Location* of the Captain as the reference point. The controllers are located inside the Oakland Center while the airplane is on route between Sacramento and Los Angeles. The *Plan* also has an associated *History* which identifies the current flight altitude, altitude alert value, and EPR values before the execution of the *Plan*.

Figure 38 specifies the information that is contained inside the Airplane Cockpit system using the *Actors* and the *MediatingArtefacts*. The information “Flight Altitude” is accessible to all crew members by referring to the Altimeter. The FO can refer to the Altitude Alert System containing the information regarding the “Flight Cleared Altitude”. However, the Capt can sometimes request the same information from the FO. The Capt has the information regarding the “Max elevation to be achieved”, which is provided in the Flight Plan that is accessible to the Capt. However, the FO can also access this information through informal communication between the Capt and FO, if required. The SO utilizes the Engine Performance Table for the

“EPR values”; therefore, this information is only accessible to the SO. The Capt and the controllers communicate using the radios to discuss “ATC Clearances” (the FO can sometimes participate in this communication, if necessary). The ATC controller can also request the information regarding “Flight Cleared Altitude” and “Flight Altitude” using the radio before re-directing the Capt to the High-Altitude controller. Clearly, the system under analysis has an extensive amount of information. Although, the original analysis of this system [16] elucidates the information present in the system, there has been no mention of an *InformationHub* that facilitates the flow of this information in the system. For completeness of the case study, the Cockpit is considered as the *InformationHub* of the system. This makes logical sense since the Cockpit allows the collection and flow of important system information by utilizing various system resources. Figure 39 describes the Cockpit as an *InformationHub* with the crew members co-ordinating amongst each other to facilitate the flow of information in the system.

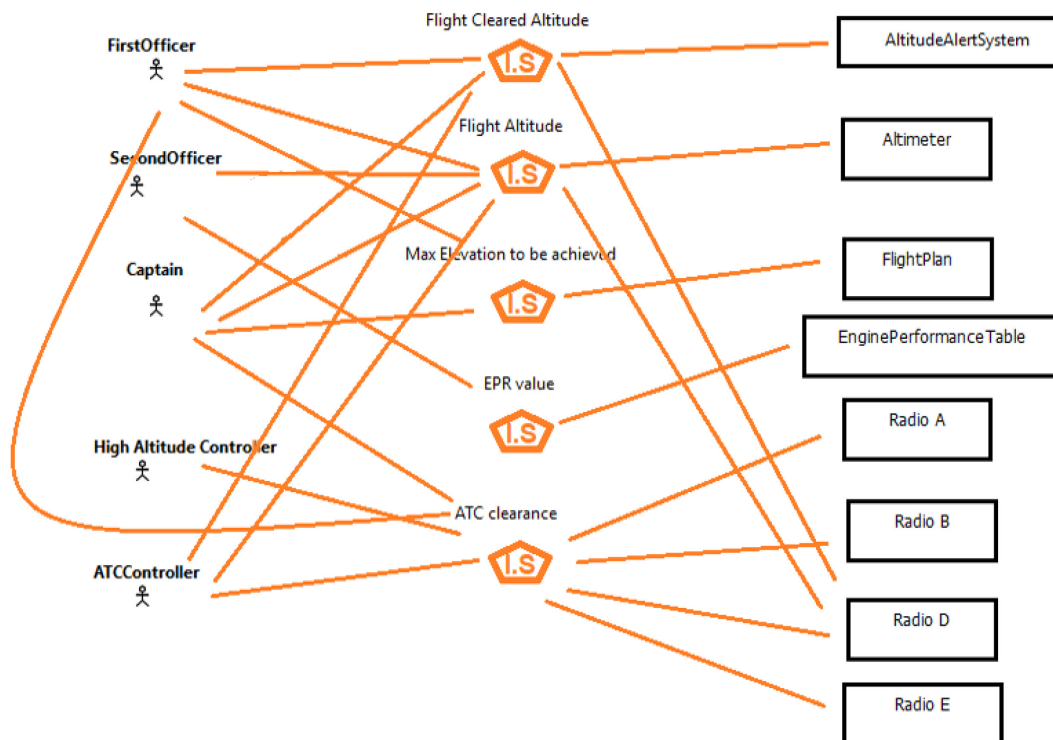


Figure 38. The Information contained inside the Airplane Cockpit System

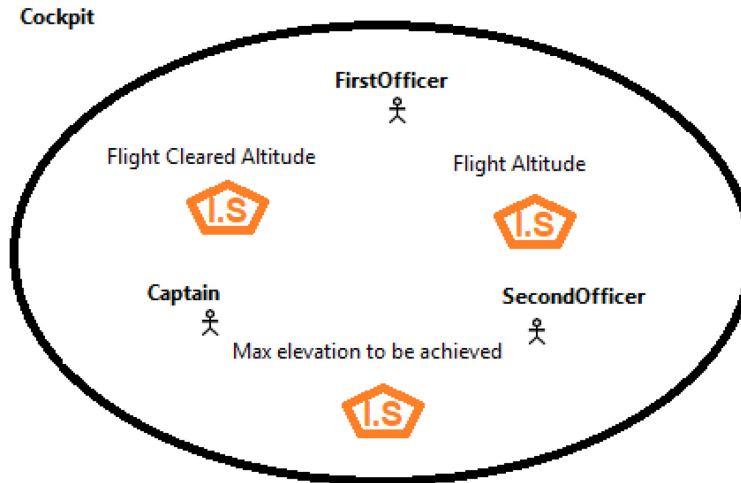


Figure 39. The Information Hub used in the Airplane Cockpit system

Figure 40 provides the flow of activities for the Capt during the execution of the *Plan*. It begins with the *State*, i.e., “Current Flight Altitude” that is followed by the Capt checking the flight plan and deciding the series of steps that need to be executed to achieve the target cruise altitude. This is followed by the execution of several *Actions*: the Capt can use her radio to communicate with the ATC depending on whether the flight altitude needs to be increased or not. Alternatively, in case the flight altitude is satisfactory, the Capt can utilize *MediatingArtefacts* like Paper and the Departure Chart to “Perform housekeeping task” and “Perform redundant error checking” while continuously monitoring the ATC frequency. Throughout the execution of these *Actions* the Capt monitors the aircraft and crew performance. This is achieved by utilizing *MediatingArtefacts* like the flight Altimeter and the FMCS. When the Capt decides to communicate with the controllers for altitude clearances, they provide an appropriate response after understanding the current flight status. If the controllers do not provide the necessary altitude clearance, the Capt ascends to an intermediate altitude (*GoalState*). Otherwise, the Capt flies the plane and increases its altitude (from the intermediate altitude like 23,000 ft) to its filed cruise altitude thereby achieving its intermediate *GoalState* of “Getting required altitude clearances” and the final *GoalState* of “Climbing to filed cruise

altitude”. Although the original Airplane Cockpit system does not discuss the use of *BodilySupport* by the *Actors* of the system, an instance for it, i.e., “Using finger to point at Flight Plan” has been specified to ensure the completion of analyzing the system and visualizing it using the introduced notation. The Capt can use this to check the flight plan. The ControlYoke and the SideStick controller have been linked with instances of type *DeactivationChange*, since their use can vary with the aircraft model used by the members of the crew.

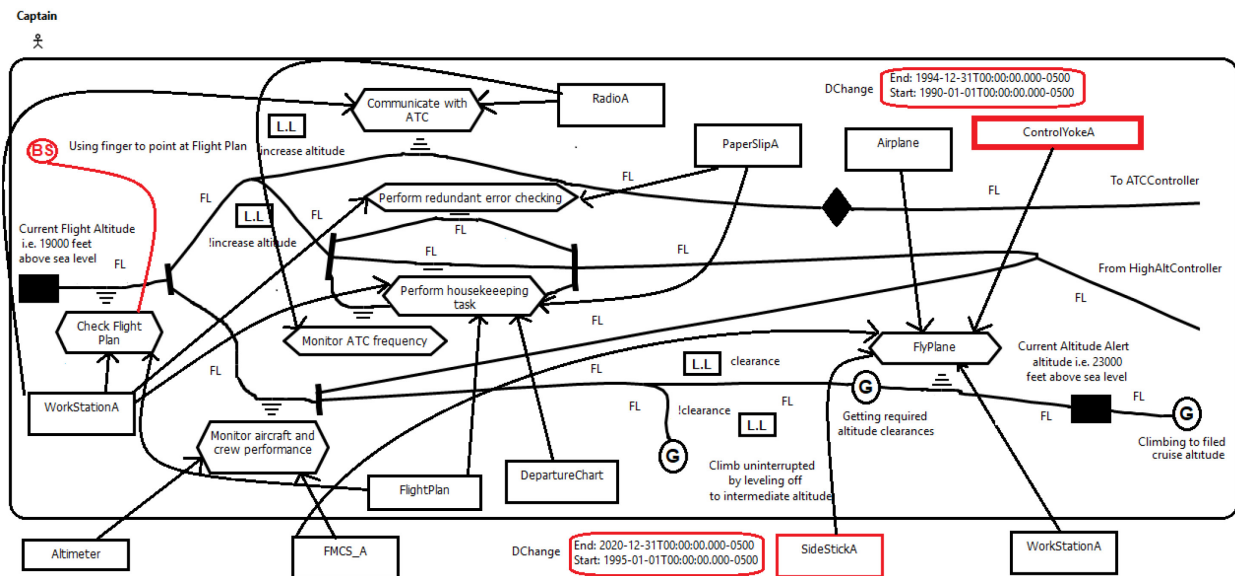


Figure 40. Flow of Actions for Captain in a Plan

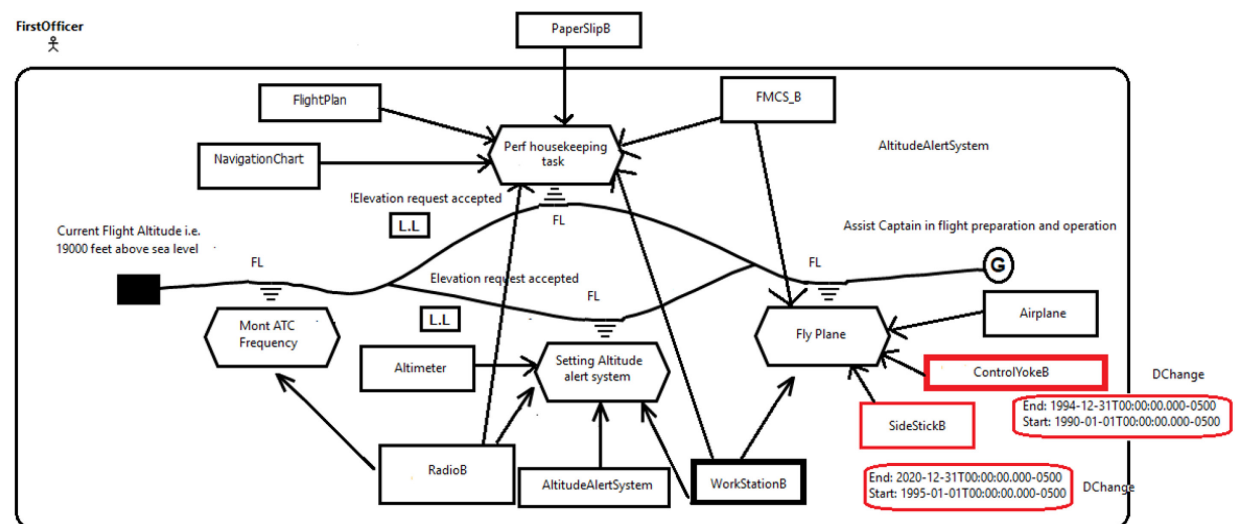


Figure 41. Flow of Actions for First Officer in a Plan

Figure 41 provides the flow of activities for the FO during the execution of the *Plan*. It begins with the *State*, i.e., “Current Flight Altitude” followed by the *Action* to monitor ATC frequency by using its radio. Since the FO essentially assists the Capt during flight operation, many of their *Actions* like performing housekeeping tasks and flying the plane are similar. The FO can also set the altitude alert system by accessing the Altitude Alert System, in case an elevation request has been accepted by the controllers. Like the Capt, the ControlYoke and the SideStick controller have been linked with instances of type *DeactivationChange* along with the duration of their usage. The FO implements these *Actions* to achieve its final *GoalState* of “Assist Captain in flight preparation and operation”.

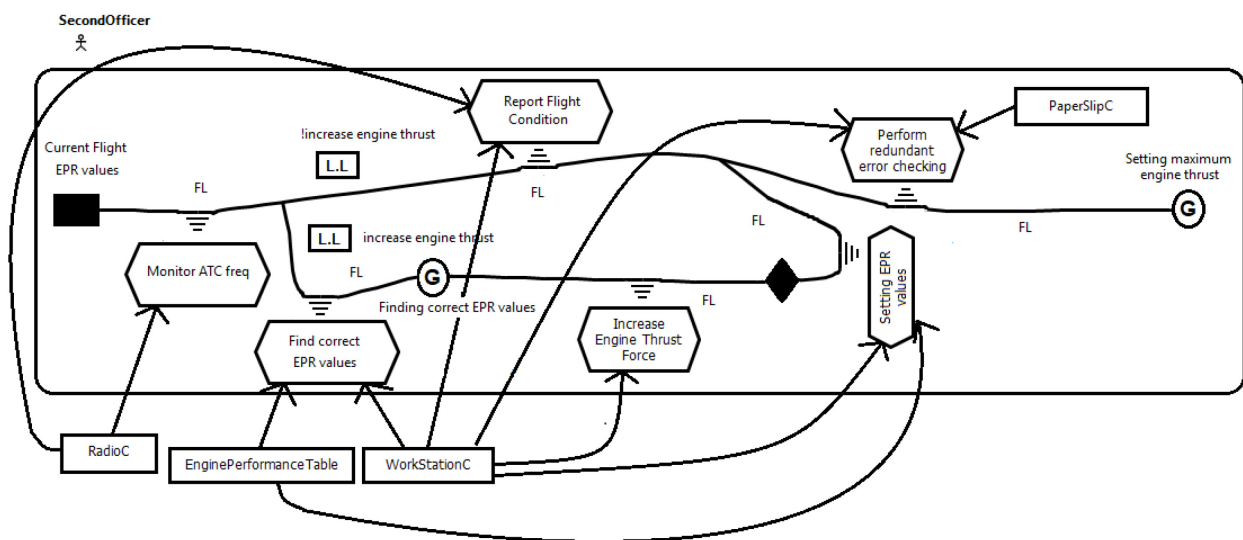


Figure 42. Flow of Actions for Second Officer in a Plan

Figure 42 provides the flow of activities for the SO during the execution of the *Plan*. It begins with the *State*, i.e., “Current Flight EPR values”. The major interesting feature of this flow is the series of steps followed by the SO provided the engine thrust needs to be increased. The SO first finds the correct EPR values from the Engine Performance table on its workstation and increases the engine thrust force based on these values and updates these values in the table. This series of *Actions* also assists the SO in achieving her sub-goal of “Finding Correct EPR

values” while the complete series of *Actions* assists the SO in achieving her main *GoalState* of “Setting maximum engine thrust”.

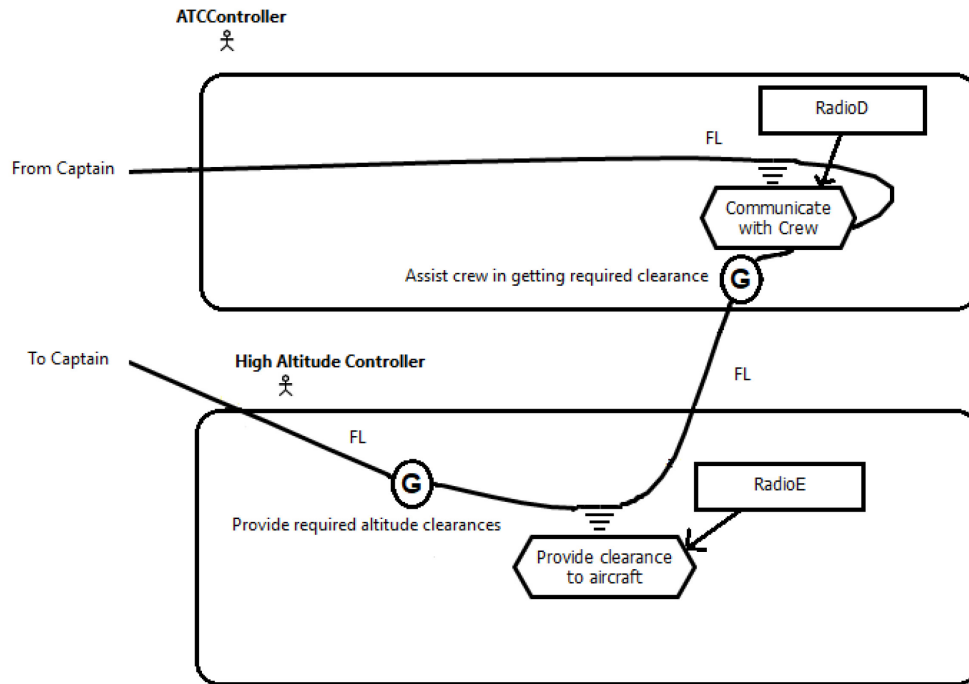


Figure 43. Flow of Actions for ATC and High-Altitude Controller in a Plan

Figure 43 provides the flow of activities for the ATC and High-Altitude Controller during the execution of the *Plan*. The ATC Controller communicates with the crew using her radio, in case the Capt requests an increase altitude clearance (see Figure 40). The ATC Controller can then re-direct the crew towards the High-Altitude controller to get the necessary clearances. The main *GoalState* for an ATC Controller is “Assist crew in getting required clearance” and for a High-Altitude controller “Provide required altitude clearance”. However, these goals are intermediate *GoalStates* in the overall plan and are visualized accordingly in the DC notation.

5.4 Generated URN Models for Airplane Cockpit System

This section provides an explanation for the URN models that have been generated by implementing the transformations between DC and URN. Since most concepts in the generated

URN models have already been discussed in Section 4.4, this section does not repeat this discussion for the generated URN models of the Airplane Cockpit system's DC instance.

Figure 44 describes the GRL model generated for the Capt of an Airplane Cockpit system. It corresponds to the sequence of *Actions* followed by a Capt (see Figure 40) and details the goals of the Capt like “Climbing to filed cruise altitude”, “Getting required altitude clearances” and “Climb uninterrupted by leveling off to intermediate altitude”. It also contains tasks that are used to describe the actions like “Communicate with ATC” and “Check Flight Plan” that are contributing towards the achievement of the Capt's goals. Since “Getting required altitude clearances” is an intermediate goal occurring before the final goal (i.e., “Climbing to filed cruise altitude”) (see Figure 40), actions like “Check Flight Plan” and “Communicate with ATC” only contribute to it, while the actions that occur after it (e.g., “FlyPlane”) contribute only to the final goal. Additionally, “Getting required altitude clearances” contributes towards the achievement of “Climbing to filed cruise altitude”. A key point to consider is that because “Getting required altitude clearances” and “Climb uninterrupted by leveling off to intermediate altitude” are alternatives, both goals have similar contributing elements. However, since “Climb uninterrupted by leveling off to intermediate altitude” is a final goal, it does not contribute towards “Climbing to filed cruise altitude”.

A feature of this GRL model is the use of “Using finger to point at Flight Plan” (*BodilySupport* element in the original DC instance) by the Capt to achieve the task of “Check Flight Plan”. “Current Flight Altitude” and “Current Altitude Alert Altitude” act as the KPI elements that contribute to all goals and tasks occurring between their corresponding locations (see Figure 40) and a DC modeler can specify different values for them in the transformed GRL models.

Figure 45 describes the GRL model generated for the FO of an Airplane Cockpit system. It corresponds to the sequence of *Actions* followed by a FO (see Figure 41) and details the goal of the FO, i.e., “Assist Captain in flight preparation and operation”, along with the actions contributing towards achieving these goals like “Fly Plane” and “Setting Altitude Alert System”.

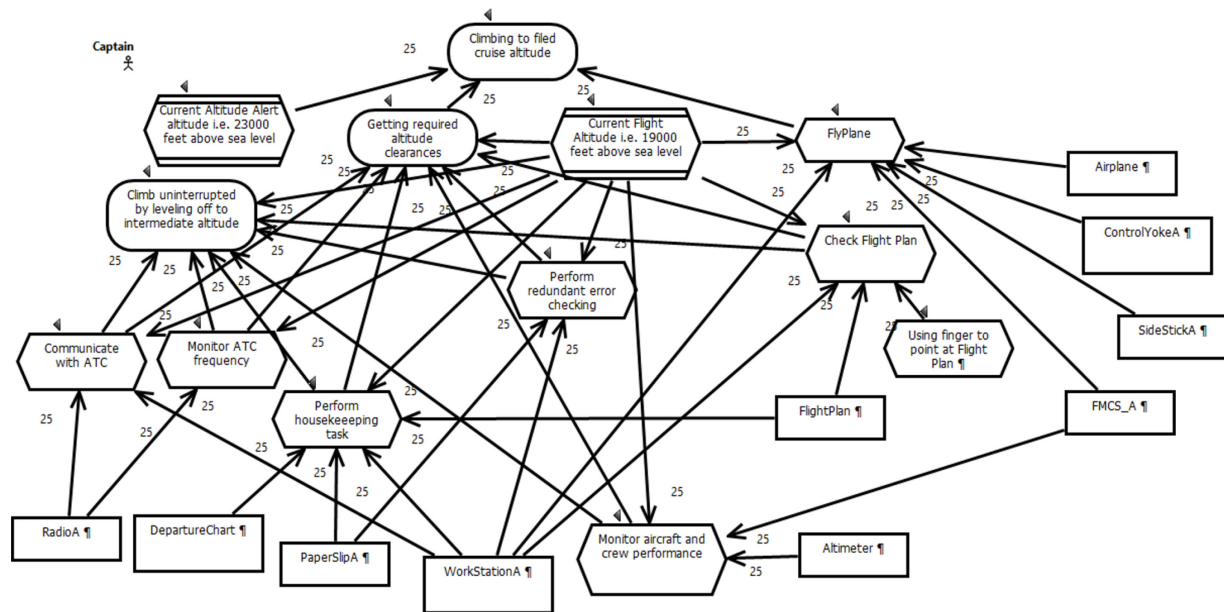


Figure 44. Generated GRL Model for Captain

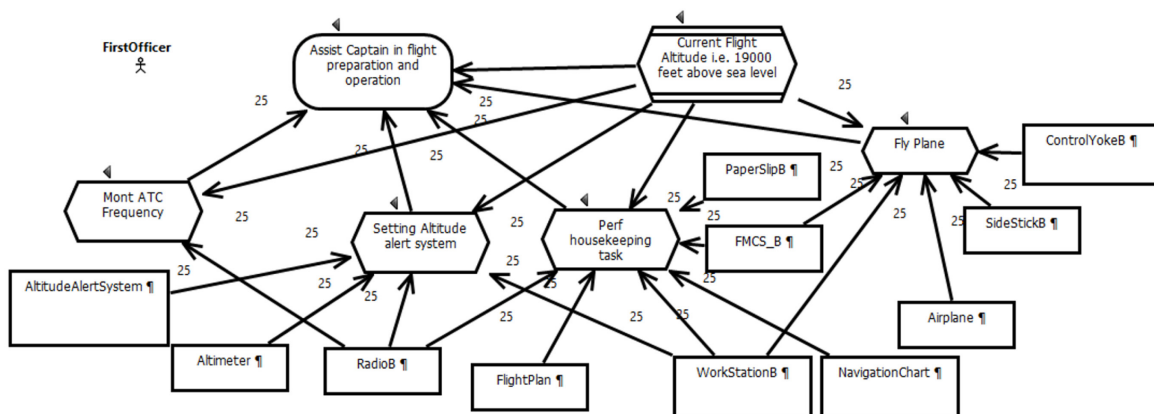


Figure 45. Generated GRL Model for First Officer

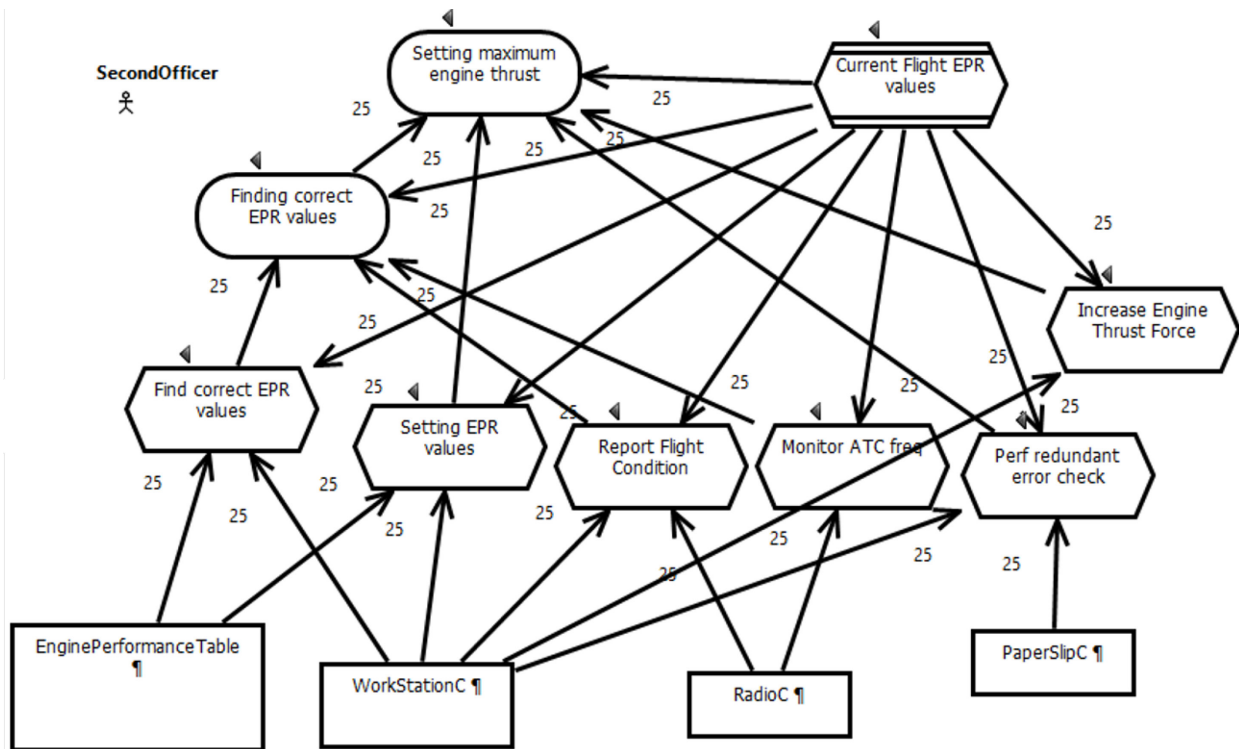


Figure 46. Generated GRL Model for Second Officer

Figure 46 describes the GRL model generated for the SO of an Airplane Cockpit system. It corresponds to the series of Actions performed by the SO (see Figure 42) and details the goals of the SO like “Finding correct EPR values” and “Setting maximum engine thrust” along with the actions contributing towards achieving these goals like “Setting EPR values” and “Increase Engine Thrust Force”. All tasks (like “Monitor ATC freq” and “Increase Engine Thrust Force”) and KPI elements (“Current Flight EPR values”) contribute towards appropriate goals and tasks based on their position on the path (see Figure 42). Similarly, Figure 47 specifies the GRL model generated for the ATC and High-Altitude Controllers of an Airplane Cockpit system. It corresponds to the sequence of Actions followed by the controllers (see Figure 43) and details the goals of the controllers: “Assist crew in getting required clearance” and “Provide required altitude clearance”.

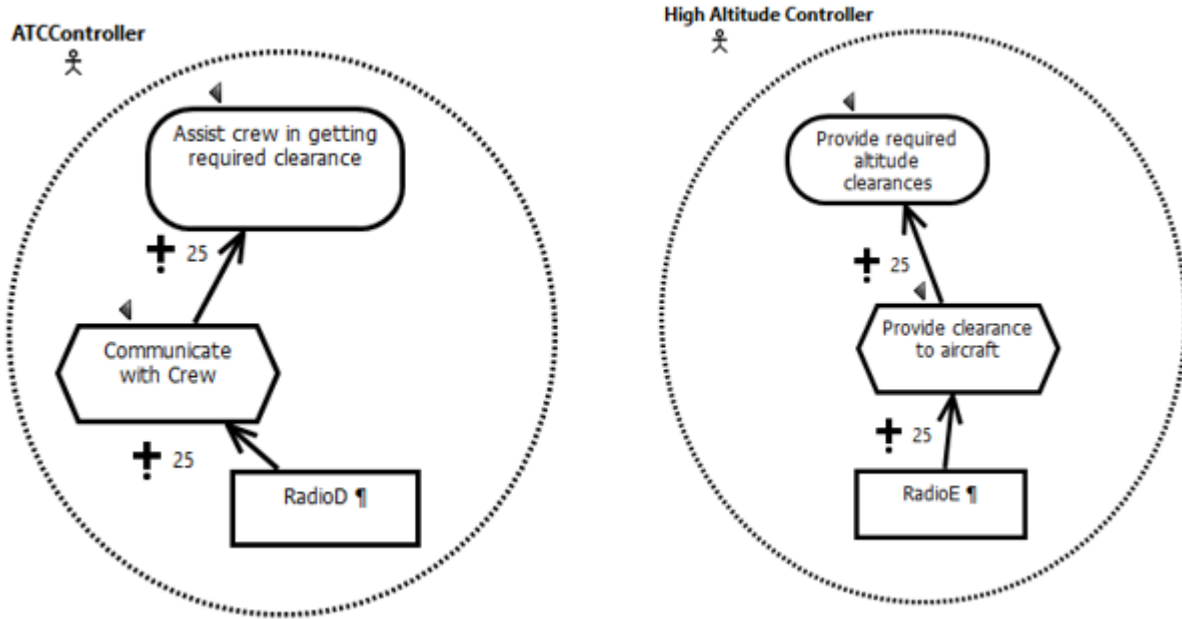


Figure 47. Generated GRL Model for ATC and High-Altitude Controller

Figure 48, Figure 49, Figure 50, and Figure 51 describe the UCM models for the Capt, FO, ATC and High-Altitude controllers, and SO of the Airplane Cockpit system that correspond to their DC instances described in Figure 40, Figure 41, Figure 43, and Figure 42, respectively. The models are collectively contained inside the final UCM model called “Increase Flight Altitude”. The flow of activities in the generated UCM models are almost like the DC instances described previously, with DC elements replaced with their corresponding target URN element.

The States like “Current Flight EPR values” and “Current Flight Altitude” belonging to the SO, Capt, and FO are transformed into the starting points for the UCM models for each of these DC *Actors* (transformed into UCM *Components*). The *Actors* actions like “Increase Engine Thrust Force” and “Check Flight Plan” are transformed into UCM *Responsibilities* linked to the UCM *NodeConnections* generated from the *Flowlinks* in the source runtime instance. The final *GoalStates* (based on the path generated by connecting the *FlowLinks*) like “Climbing to filed cruise altitude”, “Climb uninterrupted by leveling off to intermediate altitude”, and “Setting maximum engine thrust” are transformed into the end points for the UCM models. *URNLinks* are

defined to connect the *Responsibilities* and *EndPoints* in the UCM map to the *Tasks* and *Goals* in the corresponding GRL model.

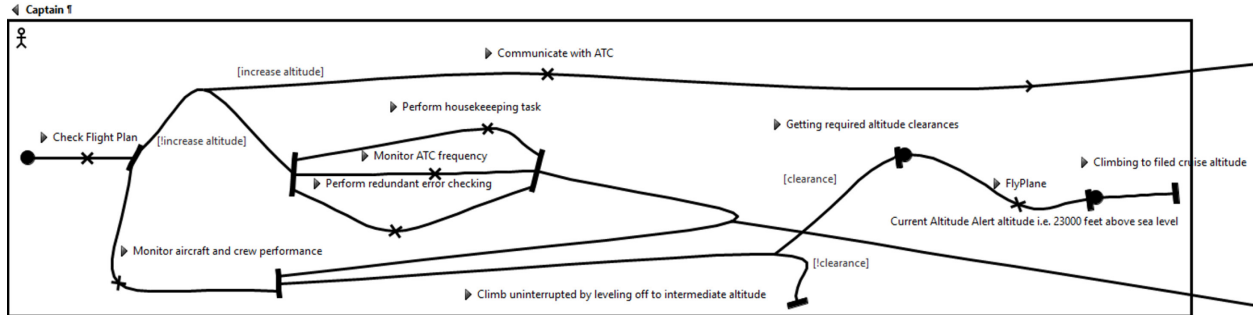


Figure 48. Generated UCM Model for Captain

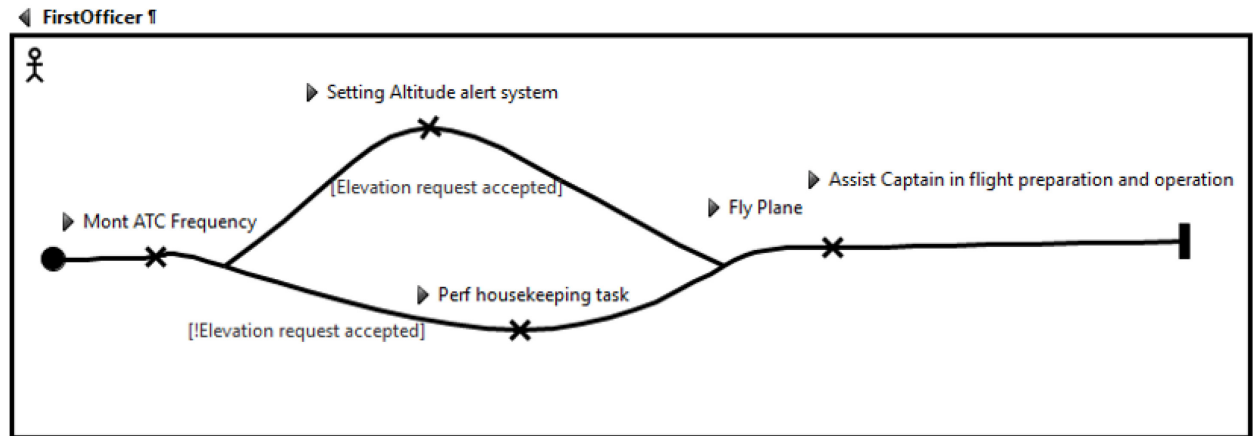


Figure 49. Generated UCM Model for First Officer

In Figure 48, “Current Altitude-Alert Altitude” and “Getting required altitude clearances” are represented as StartPoint-EndPoint pairs that have been transformed from an existing intermediate *State* and *GoalState* in the original DC instance. Similar StartPoint-EndPoint pairs have been generated in Figure 50 and Figure 51 to reflect existing intermediate *GoalStates* in the source DC instance.

Figure 52 describes another UCM map that is generated for the “Cockpit” that acts as an *InformationHub* in the original DC instance (see Figure 39). It contains the components for Capt,

FO, and the SO that co-ordinate their activities in the Cockpit and facilitate the flow of information in the system.

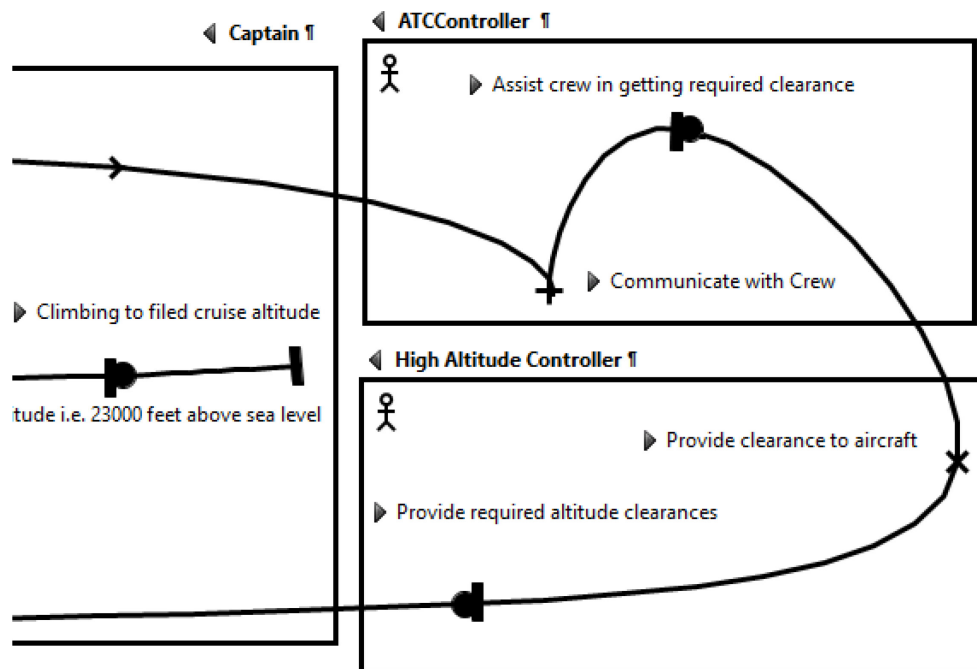


Figure 50. Generated UCM Model for ATC and High-Altitude Controllers

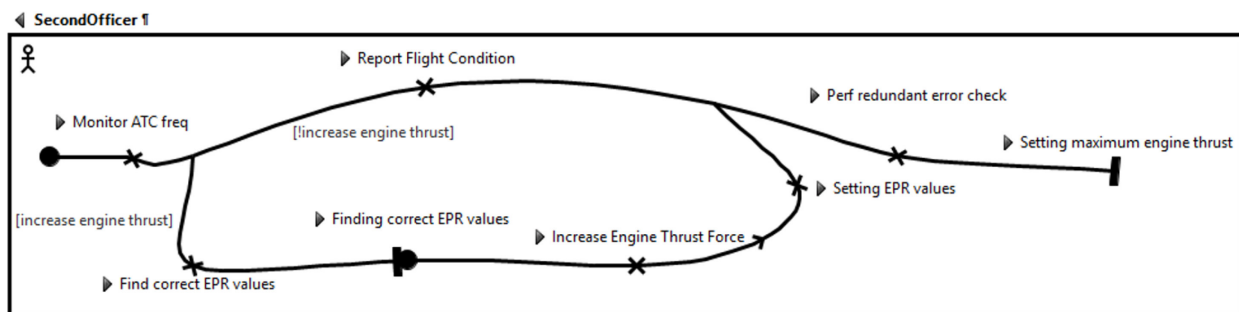


Figure 51. Generated UCM Model for Second Officer

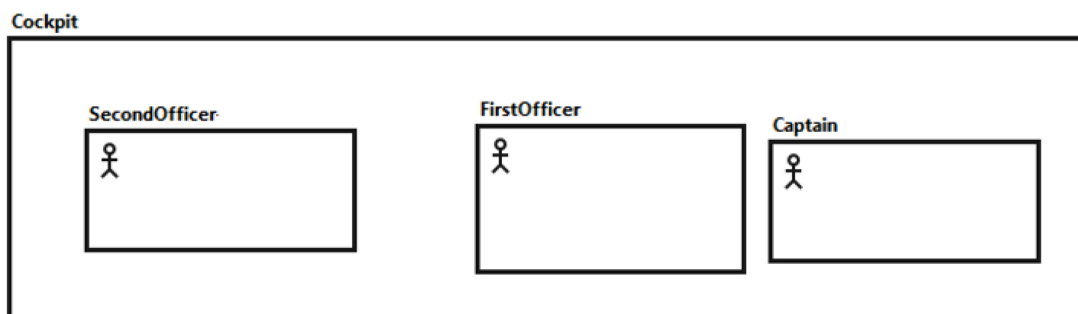


Figure 52. The UCM Model for Cockpit in Airplane Cockpit system

5.5 Testing the Generated URN Models

In addition to thoroughly testing the transformation of the DC models for the Software XP team and the Airline Cockpit system into URN goal and scenario models, a few other test transformations are performed in an ad-hoc fashion. Although a detailed end-to-end testing of the transformed URN models for potential errors is beyond the scope of this research work, a few steps leveraging the existing capabilities of the jUCMNav tool are applied to validate the correctness of the transformed URN models. Firstly, the generated models are analyzed in jUCMNav using a combination of the default GRL evaluation strategy and UCM scenario definitions created post-transformations. This provides the ability to both inspect the generated models and validate the traceability links amongst the GRL and UCM model elements. The second step involves the modification of existing contribution values of a few elements in GRL (from the default value of 25) along with providing numerical values to the KPI elements to analyze their impact on the system. As expected, the generated GRL model reacts in accordance to the values provided to the different elements. Finally, the elements of TimedURN are validated by creating different Timepoints and analyzing their impact on the generated models. Again, the model behaves in accordance with the evaluation strategy of TimedURN, deactivating changes that occur in a specified Timepoint. In addition to these validation steps, all elements present in the models' metadata are manually analyzed to validate their correctness.

5.6 Summary

This chapter describes a case study that is based on a real-life Airplane Cockpit System that exists in the DC literature. First, the Airplane Cockpit system is described. Then, a domain model capturing the elements present in the actual physical environment for the system is

described. This is followed by the definition of a DC-based instance model for the Airplane Cockpit system using the concepts of the proposed DC language along with visualization of these instances using the introduced DC notation. Finally, the DC elements are transformed into the URN goal and scenario models. Essentially, the analysis performed for implementing the case study outlines the steps that a DC modeler needs to execute to use the proposed DC language to analyze a system using DC. The case study demonstrates the feasibility of the proposed DC language concepts for analyzing a real-life socio-technical system using DC.

CHAPTER 6: Related Work

To the best of our knowledge, there has been no prior attempt to specify a formal modeling language for Distributed Cognition. However, previous works exist for the specification of modeling languages for other domains by implementing Model-To-Model (M2M) transformations between the domain and the User Requirements Notation (URN).

Work done by Georg and Mussbacher [10] is most closely related to this research work. It is where Activity Theory (AT) models are first formally defined by specifying a metamodel derived from AT concepts. These models are then subsequently connected to goals and scenario models specified with URN. Based on this work, the metamodel for URN and the capabilities of jUCMNav are extended to provide support for models based on AT concepts [19], thereby effectively integrating AT in URN and the MDE development paradigm.

The work by Kumar and Mussbacher [20] focuses on defining a textual syntax for URN, called Textual URN (TURN). The authors specify the abstract syntax for the notation by defining a metamodel for TURN. An interesting feature in this work is the specification of a concrete syntax for the notation that is defined using the Xtext framework [36]. Xtext automatically generates an editor for specifying an instance model using TURN. The semantics for TURN are provided by implementing the transformations between TURN and URN metamodels using the ATL tool.

While this research work is unique in the sense that it provides the specification of a metamodel for DC in the context of the Model Driven Engineering (MDE) development paradigm [29], there exist other works that use the concepts defined by DC to establish metamodels for other domains.

The work by Belkadi et al. [6] focuses on defining a situation model to support awareness in collaborative design. These collaborative situations include basic entities like human and material resources, interactional resources like activities, and roles like actors. The authors define a situational metamodel for collaborative situations, which utilizes the concepts of DC. However, they do not specify a metamodel for DC.

The work by Omicini, Ricci, and Viroli [26] defines a metamodel for multi-agent systems (MAS) focusing on agents and artifacts [26]. They provide an informal description of their metamodel rather than a formal specification with a metamodel. While they do not define a metamodel for DC, they utilize its concepts, particularly the concept of artifact.

Other work in metamodeling by Bhupatiraju [7] determines a coherence network to describe a theoretical understanding of medication management activities. Like the previous work, the metamodel is not defined formally and is used as a systematic tool to avoid or eliminate medical errors. This work uses the concepts of DC for establishing the metamodel for coherence networks but does not specify a metamodel for DC.

CHAPTER 7: Conclusion and Future Work

This research work proposes a formal modeling language for Distributed Cognition (DC) [15]. First, the abstract syntax for the language is defined by specifying a metamodel for DC utilizing the themes and principles defined in Distributed Cognition for Teamwork (DiCOT) [8]. The specification of a metamodel allows users to analyze a system consisting of artifacts, actors, goals, and workflow elements (among other concepts) from a DC perspective. This is followed by the specification of a DC notation which can be used to visualize the runtime instances of the DC models that are defined based on the concepts of the proposed metamodel. The semantics of the language are then provided by implementing the transformations between DC and the User Requirements Notation (URN) [1][17] notations using the ATL tool [5]. A runtime instance for the Software XP team [30] using the concepts of the DC metamodel is created and visualized using the proposed DC notation followed by the generation of corresponding URN models for this instance, demonstrating the initial adequacy of the proposed language concepts. Finally, a case study is performed by applying the concepts of the developed DC modeling language on an Airplane Cockpit System [16], which is an existing socio-technical system from DC literature. Overall, it is observed that all aspects of the case study as described by the authors [16] can be covered with the proposed DC modeling language.

The main objective of the DC language is to integrate DC with other requirements engineering formalisms and the Model-Driven Engineering (MDE) development paradigm. This provides modelers the ability to analyze a system using the concepts of DC and then utilize existing MDE techniques, specifically requirements specification techniques, to refine their analysis. At the very least, a runtime instance model of their problem system may potentially be enumerated, thereby facilitating the analysis of their system with the help of a software model.

The implementation of transformations between the DC and URN models has significant advantages for both modeling notations. There exist several concepts in DC, like the location of an entity and type of information flowing in the system under analysis that cannot be captured by URN. Conversely, URN provides capabilities like assigning detailed contribution values, importance values, and KPI values to intentional elements and performing GRL trade-off analysis to determine how well design alternatives meet important system goals. Furthermore, the transformed DC model can be analyzed with a scenario test suite based on UCM scenario definitions and key scenarios can be highlighted by the UCM traversal mechanism. Therefore, combined DC/URN models allow a modeler to make a more informed decision regarding system design.

In future work, the focus of this research will be on (a) providing a simple editor for models conforming to the proposed DC metamodel, (b) investigating a more definitive, concrete syntax for the proposed DC language, (c) defining OCL constraints [24] on the elements of the DC metamodel for completeness, and (d) specifying the reverse transformations from URN to DC to allow proposed changes in URN to be propagated back to DC models. Finally, empirical studies need to be conducted to further assess the sufficiency of the proposed DC modeling language and the intuitiveness and usability of models created using it.

REFERENCES

- [1] Amyot, D. and Mussbacher, G., "User Requirements Notation: The First Ten Years, The Next Ten Years", *Journal of Software (JSW)*, Academy Publisher 6(5):747–768, 2011. DOI: 10.4304/jsw.6.5.747-768.
- [2] Amyot, D. and Mussbacher, G., "URN: Towards a New Standard for the Visual Description of Requirements", Sherratt E. (eds) *Telecommunications and beyond: The Broader Applicability of SDL and MSC. SAM 2002*, LNCS, vol 2599, pp. 21–37. Springer, Berlin, Heidelberg, 2003. DOI: 10.1007/3-540-36573-7_2.
- [3] Aprajita, Luthra, S., and Mussbacher, G., "Specifying Evolving Requirements Models with TimedURN", *IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, Buenos Aires, Argentina, pp. 26–32, 2017. DOI: 10.1109/MiSE.2017.10.
- [4] Aprajita and Mussbacher, G., "TimedGRL: Specifying Goal Models Over Time", 6th *International Model-Driven Requirements Engineering Workshop (MoDRE 2016)*, IEEE CS, pp. 125–134, 2016. DOI: 10.1109/REW.2016.035.
- [5] Atlas Transformation Tool - <https://www.eclipse.org/atl/>
- [6] Belkadi, F., Bonjour, E., Camargo, M., Troussier, N., and Eynard, B., "A situation model to support awareness in collaborative design", *International Journal of Human-Computer Studies*, 71(1):110–129, 2013. DOI: 10.1016/j.ijhcs.2012.03.002.
- [7] Bhupatiraju, R.T., "Modeling medication management practices: the coherence theory of medication activities", PhD thesis, Oregon Health and Science University, USA, 2011. DOI: 10.6083/M43T9F6H.
- [8] Blandford, A. and Furniss, D., "DiCoT: A Methodology for Applying Distributed Cognition to the Design of Teamworking Systems", Gilroy S.W., Harrison M.D. (eds) *Interactive Systems. Design, Specification, and Verification. DSV-IS 2005*. LNCS, vol 3941, pp. 26–38. Springer, Berlin, Heidelberg, 2006. DOI: 10.1007/11752707_3.
- [9] Furniss, D. and Blandford, A., "Understanding emergency medical dispatch in terms of distributed cognition: a case study", *Ergonomics*, 49(12 –13):1174 –1203, 2006. DOI:10.1080/00140130600612663.
- [10] Georg, G., Mussbacher, G., Amyot, D., Petriu, D., Troup, L., Lozano-Fuentes, S., and France, R., "Synergy between Activity Theory and Goal/Scenario Modeling for Requirements Elicitation, Analysis, and Evolution", *Information and Software Technology (INFISOFT)*, Elsevier 59:109–135, 2015. DOI: 10.1016/j.infsof.2014.11.003.
- [11] Halverson, C.A., "Activity theory and distributed cognition: Or what does CSCW need to DO with theories?", *Computer Supported Cooperative Work (CSCW)*, 11(1–2): 243–267, 2002. DOI: 10.1023/A:1015298005381.

- [12] He, C. and Mussbacher, G., “Model-Driven Engineering and Elicitation Techniques: A Systematic Literature Review”, 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), Beijing, pp. 180–189, 2016. DOI:10.1109/REW.2016.041.
- [13] Hollan, J., Hutchins, E., and Kirsch, D., “Distributed Cognition: Toward a new foundation for human-computer interaction research”, ACM Transactions on Computer-Human Interaction, 7(2):174 –196, 2000. DOI: 10.1145/353485.353487.
- [14] Hundal, K.S. and Mussbacher, G., "Model-Based Development with Distributed Cognition", 8th International Model-Driven Requirements Engineering Workshop (MoDRE 2018), IEEE CS, pp. 26–35, 2018. DOI: 10.1109/MoDRE.2018.00010.
- [15] Hutchins, E., "Cognition in the Wild", Cambridge MA: MIT Press, 1995.
- [16] Hutchins, E and Klausen, T., "Distributed cognition in an airline cockpit", Cognition and communication at work, Cambridge University Press, pp. 15–34, 1996.
- [17] International Telecommunication Union, "Recommendation Z.151 (10/18), User Requirements Notation (URN) – Language definition", 2018. <http://www.itu.int/rec/T-REC-Z.151/en>
- [18] jUCMNav website, version v7.0.0, University of Ottawa - <http://softwareengineering.ca/jucmnav>.
- [19] Kapoor, D. and Mussbacher, G., "Support for Activity Theory in the jUCMNav Requirements Engineering Tool", M.Eng. (non-thesis) Project Report, Department of Electrical and Computer Engineering, McGill University, Montreal, Canada, August 2018.
- [20] Kumar, R. and Mussbacher, G., "Textual User Requirements Notation", 10th System Analysis and Modeling Conference (SAM 2018), Copenhagen, Denmark, October 2018. Khendek, F. and Gotzhein, R. (Eds.), System Analysis and Modeling: Languages, Methods, and Tools for Systems Engineering, Springer, LNCS 11150:163–182, 2018. DOI: 10.1007/978-3-030-01042-3_10.
- [21] Liu, Y., Su, Y., Xinshang, Y., and Mussbacher, G., "Combined goal and feature model reasoning with the User Requirements Notation and jUCMNav", 4th International Model-Driven Requirements Engineering Workshop (MoDRE 2014), Karlskrona, IEEE CS, pp. 321–322, 2014. DOI: 10.1109/RE.2014.6912277.
- [22] McKnight, J. and Doherty, G., “Distributed cognition and mobile healthcare work”, 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction – Volume 2, Liverpool, UK, pp. 35–38, 2008.
- [23] Mussbacher, G., Araújo, J., Moreira, A., and Amyot, D., “AoURN-based Modeling and Analysis of Software Product Lines”, Software Quality Journal 20(3–4):645–687, 2012. DOI: 10.1007/s11219-011-9153-8.
- [24] Object Management Group, "UML 2.0 Object Constraint Language specification", 2006. <https://www.omg.org/spec/OCL/About-OCL/>

- [25] Object Management Group, “Unified Modeling Language (UML) 2.5.1”, 2017.
<http://www.omg.org/spec/UML/2.5.1>
- [26] Omicini, A., Ricci, A., and Viroli, M., “Artifacts in the A&A metamodel for multi-agent systems”, *Autonomous Agents and Multi-Agent Systems* 17(3):432–456, 2008. DOI: 10.1007/s10458-008-9053-x.
- [27] Rybing, J., Nilsson, H., Jonson, C.-O., and Bang, M., “Studying distributed cognition of simulation-based team training with DiCoT”, *Ergonomics*, 59(3):423–434, 2015. DOI: 10.1080/00140139.2015.1074290.
- [28] Saini, R., Bali, S., and Mussbacher, G., "Towards Web Collaborative Modelling for the User Requirements Notation Using Eclipse Che and Theia IDE", 11th Workshop on Modelling in Software Engineering (MiSE 2019), IEEE CS, Montreal, Canada, May 2019.
- [29] Schmidt, D.C., “Model-Driven Engineering”, *IEEE Computer* 39:41–47, 2006.
- [30] Sharp, H., Robinson, H., Segal, J., and Furniss, D., “The role of story cards and the wall in XP teams: a distributed cognition perspective”, *AGILE 2006 (AGILE'06)*, Minneapolis, MN, pp. 11–75, 2006. DOI: 10.1109/AGILE.2006.56.
- [31] The AtlanMod team (Ecole des Mines de Nantes & INRIA) – <http://www.emn.fr/z-info/atlanmod>
- [32] The Obeo company – <http://www.obeo.fr>
- [33] TouchCORE website, version v7.0.2, McGill University, May 2019 - <http://touchcore.cs.mcgill.ca/>
- [34] URN Metamodel - <http://jucmnav.softwareengineering.ca/foswiki/ProjetSEG/URNMetaModel>
- [35] Wright, P.C., Fields, R.E., and Harrison, M.D., “Analysing Human–Computer Interaction as Distributed Cognition: The Resources Model”, *HCI Journal*, 15(1):1 –41, 2000. DOI: 10.1207/S15327051HCI1501_01.
- [36] Xtext Framework – <https://www.eclipse.org/Xtext/>