Graph Transformation Games for Negotiating Features

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Mohammed A. Alabdullatif
Department of Informatics
University of Leicester

February 2017

Graph Transformation Games for Negotiating Features

Mohammed A. Alabdullatif

Abstract

The success of e-commerce applications and services depends on the outcomes of interactions between the provider of the products or services and its requestors. The flexibility of these agents to negotiate features of the products or services traded is an important characteristic of face-to-face business interactions, but is often missing in the online world. Flexibility is needed to discuss preferences and constraints in order to determine a solution that benefits both parties. Game theory is a natural framework in which to pose such problems. This thesis is concerned with a proposal-based negotiation: through which a service provider and requestor interact by exchanging proposals. In particular, we propose negotiation games based on feature models to design the flexible business interactions. Feature models are used to represent service configurations in order to support the variability of negotiated services, which increases the flexibility of the negotiators' interactions. We introduce graph transformation games to implement and analyse our negotiation games, modelling the negotiation of features by representing the state of the game by a graph and the moves of the players by graph transformation rules. We propose two analyses of our graph transformation games in order to explore different negotiation strategies. Firstly, we analyse our graph transformation games as extensive-form games, in which backward induction technique is used to solve the game and determine the optimal strategies for the negotiators at each state of the game. Secondly, we analyse our graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker. We define single-objective and multi-objective properties in order to generate optimal strategies for the players. To evaluate our approach, we applied it to a selection of feature models in order to test the scalability of the graph transformation games' generation and analysis time.

Acknowledgements

In the name of Allah, the Beneficent, the Merciful.

First and foremost, this work would not have been completed except by guidance of the Almighty Allah, who allowed my dreams to come true. I would like to thank Allah for giving me the power to believe in myself and pursue my dreams.

I would like to take this opportunity to extend my deepest gratitude to my academic supervisor, Professor Reiko Heckel, for constantly offering adequate supervision, and keeping me on the right path during the work on this thesis. He dedicated numerous hours to giving me countless guidance and valuable suggestions. I greatly appreciate his immeasurable efforts for bringing my dreams into reality.

I would like to thank my PhD co-supervisor, Professor Thomas Erlebach, and PhD tutor, Dr Fer-Jan de Vries for their guidance and support during my PhD journey. I would also like to thank the members of my thesis examining committee, Dr Artur Boronat, and Dr Radu Calinescu, for their valuable comments and suggestions.

Many friends and colleagues have shared time with me, and they helped to make my PhD enjoyable and memorable. I would like to thank my best friend Abdullah Alqahtani for being my true brother and always being there when I needed someone to talk to. I would also like to especially thank Dr Mohammad Kharabsheh, Dr Mohammad Alshira'H, Dr Ayman Bajnaid, Dr Abdullah Alshanqiti, Marwan Radwan, and Marco Hernandez. I would also like to take this opportunity to gratefully and sincerely thank my sponsor, King Faisal University, for granting me full scholarship to pursue my studies abroad.

I also extend my sincere gratitude to my lovely wife, Mona, who inspired me and provided constant encouragement during the entire process. I also thank my wonderful children: Abdulrahaman and Danah, for always making me smile and for understanding on those weekends when I was working on this research instead of playing games.

Last but not least, I would like to thank my parents for being there for the happy times and tough times and for their guidance through my life. It would have been harder without their warm prayers. I would also like to thank my sisters and my brother for their support and help.

Contents

List of Figures

List of Tables

This thesis is lovingly dedicated to my beloved family

Chapter 1

Introduction

This chapter introduces the research topic of this thesis. Section 1.1 discusses the context within which the research problems arise. Section 1.2 discusses the statement of the problem. Section 1.3 presents the proposed solution and overall aim and objectives. Finally, Section 1.4 presents the thesis outline.

1.1 Motivation

Web services and e-commerce technologies have dramatically changed the way requestors and providers conduct business. The trading environment is becoming more complex due to the explosive growth of the number of services and products that are supplied via electronic channels. This has led researchers in both academia and industry to adopt effective and efficient mechanisms in order to handle the interactions between the providers of these services or products and their requestors. However, one of the challenges facing online business is that interactions have to be standardised to be automated which, for complex products with a range of configurations, limits the ability of the provider to react in a flexible way to the diverse preferences

of their clients.

Moreover, in many scenarios, matching requestors' requirements to the available descriptions may result in a number of alternative, potentially partial, matches. For example, a service provider offers different types of transportation for tourists, such as train and airplane. In some cases, the transportation type depends on the holiday location. If a service requestor asks for a certain type of transportation for a location that is hard to reach by this type, they may get a limited number of exact matches as many travel agencies do not offer this transportation type for the preferred location. However, they may get partial matches which provide other types of transportation. Thus, the service requestor may change their original requirements according to the available matches. This is very common in today's business.

Furthermore, the relationship that the provider wants to have with its customer is not directly competitive because its ultimate goal is to do business and improve customer satisfaction and so have a long-term relationship with that customer. However, providers and requestors inevitably have conflicting interests, as providers will seek to maximise their profits by selling as many products as possible, preferably those with the highest profit margin, while requestors will be driven by their desire to obtain maximal value for minimal investment, usually on a limited budget.

To achieve business goals, service providers and service requestors need to interact and eventually reach an agreement on certain quantities of interest. One type of interaction that is gaining increasing interest in online business is *negotiation* [2]. There are several definitions of negotiation, of which we give three examples. Mayer in [3] define a negotiation as an "interaction in which people try to meet their needs or accomplish their goals by reaching an agreement with others who are trying to get their own needs met". Bichler et al. in [4] describe a negotiation as an "iterative communication and decision-making process between two or more parties, who cannot achieve their objectives unilaterally, exchange information, deal with interdependent tasks and search for a consensus". Robinson et al. in [5] view a negotiation as "a

method in which participants bring their goals to a bargaining table, strategically share information, and search for alternatives which are mutually beneficial".

Inspired by those definitions, we provide our definition of negotiation in web services and e-commerce as *an interaction between service providers and service requestors who have relations that are characterised by different preferences and competition but also shared interests, who are trying to come to a mutually acceptable agreement.*

Negotiation is often complicated, time-consuming and costly for participants to reach an agreement. According to [6], the complexity of negotiation is usually affected by three factors: parties have distinct interests, parties do not have full information about their counterparts, and parties are dependent on one another for agreement.

The behaviour and the outcomes of the negotiation are affected by the negotiation type. Negotiation theorists have identified two types of negotiation: distributive negotiation and integrative negotiation [7]. In distributive negotiations, the parties behave competitively, trying to minimise each other's gain rather than trying to collaborate for the benefit both. In game theory, this negotiation is called a zero-sum game [8] in which, if one party gains, the other loses. In integrative negotiation, parties view the negotiation as a non-zero-sum game. The parties do not behave competitively, and they do not intend to minimise each other's gain.

Negotiation has to follow a certain protocol. Protocol can be viewed as a set of rules governing the interaction among participants. This includes the participant types, the negotiation states and the permitted actions of the participants in particular states [9]. A negotiation strategy determines the actions of the participants at each state in order to achieve the business goal while using a particular negotiation protocol.

An effective negotiation framework including a simple protocol and a well-defined negotiation strategy are required to enable flexible interactions that allow the participants to discuss their preferences and reach an acceptable agreement.

1.2 Problem Statement

Negotiation has become increasingly important since automated interactions need flexibility. The flexibility of negotiation with complex configurations of services or products is rarely supported by the current frameworks, yet this flexibility is needed to enhance the ability of the negotiation participants to achieve the best outcomes. It is necessary for service providers to react more quickly to requestors' diverse preferences and take advantage of new opportunities. This requires a mechanism that facilitates a dynamic customisation of supplied services for any service requestor, which can also be used to offer alternative services or products to those requested. Moreover, a need exists for a formal model that captures the relationship among negotiation rules, constraints, strategies and goals in order to build an effective negotiation framework.

One of the key aspects of negotiation is the adoption of a negotiation protocol that the participants need to adhere to. The design of appropriate negotiation protocols is crucial and requires a careful consideration as it is closely connected to the domain in which the negotiators will be acting. The negotiation protocols should be simple and characterised by certain properties in order to provide the negotiators with a suitable interaction environment. Given a particular negotiation protocol, typical questions that arise are:

1. What type of negotiation should be considered? Is it integrative negotiation or distributive negotiation?

2. How do the negotiators choose their actions efficiently? In other words, what are their strategies while negotiating?

3. How much knowledge do the negotiators have about each other in order to anticipate the other's behaviour?

1.3 Solution

The goal of this research is to develop a new structured negotiation model that enables the negotiators to discuss their preferences and interact in a flexible and strategic way to reach an agreement that benefits both of them. This research proposes a game-theoretic approach to proposal-based negotiation where the information exchange between the participants is in the form of proposals which can be accepted or rejected. To reduce the efforts required in generating proposals in the negotiation process and increase the flexibility of participants' interactions, feature models are used to represent service configurations. The negotiation of features is implemented as graph transformation games to model and analyse the providers and requestors' strategic choices. It aims to provide a negotiation framework using game-theoretic techniques. A graph transformation game is a state-based game in which the states of the game are given by graphs. The rules of the game are defined by graph transformation rules which determine the available actions of the players. The rules are designed according to an introduced alternating-offer negotiation protocol in which the negotiators interact by taking turns in making proposals. We propose two analyses of our graph transformation games, one as extensive-form games and the other one as two-player turn-based stochastic games. The results that are obtained by these analyses determine the negotiators' optimal strategies. The main contributions achieved in this research are:

1. Graph transformation games.
2. Implementing graph transformation games:
   (a) Developing a metamodel to define the negotiation entities.
   (b) Designing negotiation rules to describe the actions of the negotiators.
3. Two different analyses of graph transformation games:

   (a) Extensive-form graph transformation games.

   (b) Stochastic graph transformation games.

4. Empirical scalability evaluations of both the implementation and the analysis of graph transformation games.

Part of our work was presented as abstract papers at:

- STAF 2014 Doctoral Symposium. Mohammed Alabdullatif and Reiko Heckel. A Game Theoretic Approach to Support Negotiation Based on Feature Models.
- Graphs as Models 2016. Mohammed Alabdullatif and Reiko Heckel. A Graph-based Game to Negotiate Features.

We also published a paper at the Seventh International Workshop on Graph Computation Models (GCM 2016) affiliated with the Conferences on Software Technologies: Applications and Foundations (STAF):

- Mohammed Alabdullatif and Reiko Heckel. Graph Transformation Games for Negotiating Features. In Proceedings of the Seventh International Workshop on Graph Computation Models (GCM 2016). July 2016.

1.4 Thesis Outline

The structure of the thesis is shown in Figure 1.1. The thesis is organised into seven chapters as follows.

Chapter 2 presents background information on the research topic, which includes:

- feature modelling, where we discuss feature model semantics, configuration techniques, modelling methods, supporting tools and the use of feature models in web services.
- game theory, including basic game-theoretic concepts, extensive-form games, stochastic games and the use of game theory in negotiation. We also provide an overview of the PRISM-games model checker, which we use in our analysis.
- graph transformation and its main components, which are graphs and rules. We also present basic concepts about the Henshin transformation language and tool environment.

Chapter 3 presents the proposed graph transformation games. It provides the detailed implementation steps for the graph transformation games, which include defining the game metamodel, designing the game rules and generating the transition system of the game. Experiments were conducted to measure the game state space generation time and these are also presented in the chapter.

Chapter 4 introduces our approach to analysing graph transformation games as extensive-form games. The graph transformation games were analysed using backward induction in order to determine the players' optimal moves, which represent the Nash equilibrium. The scalability of this analysis was assessed by conducting experiments to measure the time spent in applying backward induction and determining the payoff obtained by reasoning backward.

Chapter 5 presents the proposed analysis of graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker. The

graph transformation games were generated from Henshin to PRISM-games format by modifying Henshin source code. Single-objective and multi-objective properties were defined in order to generate optimal strategies for the players. This analysis was evaluated by conducting experiments to measure the time taken to construct the model in PRISM-games and the time spent in generating the strategies.

Chapter 6 highlights approaches related to ours, and discusses their differences to our approach in order to assess the research contribution and develop a clear direction for the proposed approach. We categorise the related work into the following: feature models in negotiation, feature models and graph transformation, feature models and game theory, game theory in web services and e-commerce negotiation, and game theory and graph transformations.

Chapter 7 provides the conclusions to the conducted research and describes further research issues to be considered as future work.

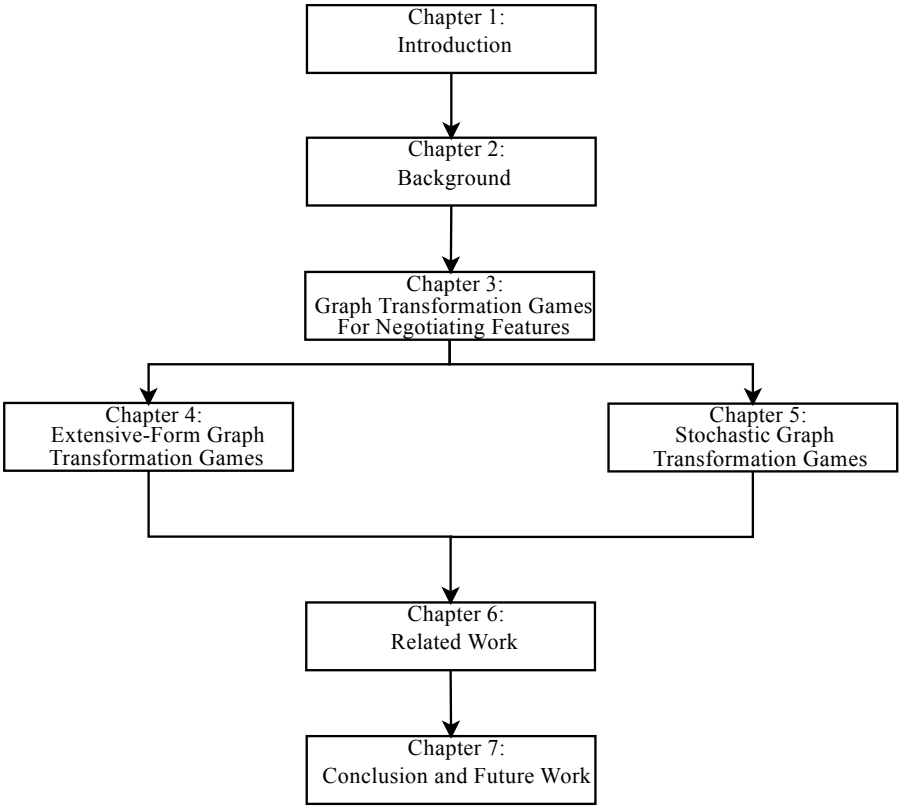


FIGURE 1.1: Thesis Structure

Chapter 2

Background

In this chapter, we provide the background information required to understand the technical contribution. In Section 2.1, we discuss feature models including their semantics, configuration techniques and methods in order to understand the required concepts throughout the thesis. We also discuss the use of feature models to model the variability of web services in Section 2.1.4. In Section 2.2, we discuss game theoretic concepts and game types required to understand our analysis in Chapter 4 and Chapter 5. Section 2.3 presents the basic notions of graph transformations including the definitions of graph and rules. We also discuss the Henshin tools that will be used in our implementation.

2.1 Feature Modelling

Feature modelling is a key technique for modelling the commonalities and variabilities of products in a Software Product Line (SPL) in terms of their features. It has generated a lot of interest since its introduction by Kang et al. in the FODA method [10]. We first describe the essential principles and semantic foundation of

feature models in Section 2.1.1. We then discuss the feature modelling configuration techniques, methods, applications and tools of feature modelling in the following sections.

2.1.1 Semantics

Features: each concept in a model represents a feature. There are several definitions of features [10–12]. For example, in [10], a feature has been defined as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems". It gives a part of the information of the complete model and it is the main component of a feature model. A feature model represents the relationships between different features involved in a model that give a global meaning to the model. For example, transportation and accommodation can be used as features in a Travel Agency feature model.

Feature Diagram: the features are organised into a tree which is called a feature diagram [13]. In the literature, the terms feature model and feature diagram are used interchangeably but feature diagrams are usually used to visually represent feature models. A feature configuration is an instance of a feature model that can be specified by selecting a set of features and it has to meet feature model constraints [14]. The feature diagram depicted in Figure 2.1 compactly describes a family of holiday services, where each member of the family corresponds to a combination of features.

Feature Hierarchy: feature diagrams are used to model the commonalities and variabilities of a software system in a hierarchical form [10]. The hierarchy is usually represented as a rooted tree; the root feature denotes the main concept of the feature

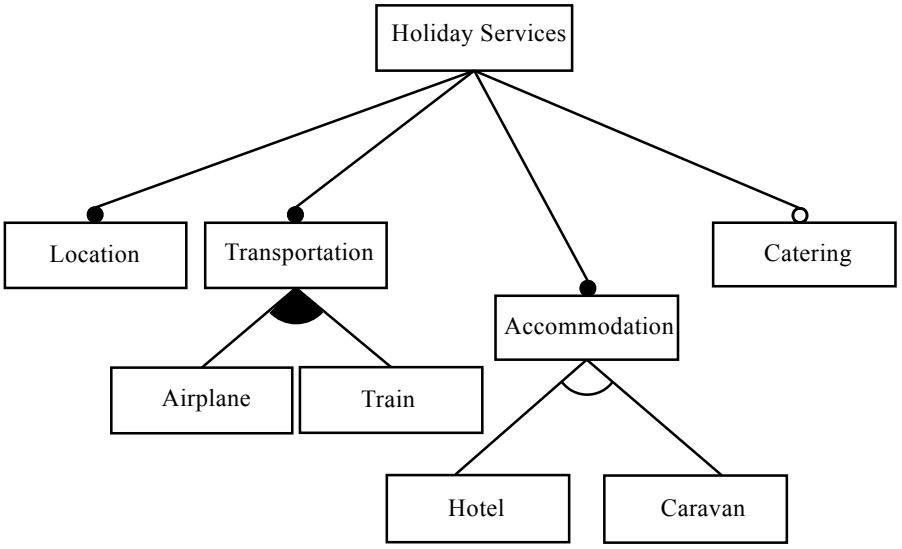


FIGURE 2.1: Holiday Services Feature Model

diagram. Edges in feature diagrams model parent-child relations. The primary purpose of a hierarchy is to organise a large number of features into multiple levels. The general purpose of the product can be obtained from the root. Then, more details can be explored after moving to the next level of nodes in the tree. Every feature has exactly one type, which is given by the relation between the feature and its parent. In our case, we have four types of features: mandatory, optional, alternative, and or.

Figure 2.2 gives a representation of the graphical notation for common relationships between a parent feature and its child features in a feature diagram. Throughout the thesis, we will rely on the same graphical notation, proposed by Czarnecki et al. in [11]. Following are the detailed explanations of the parent-child relationships.

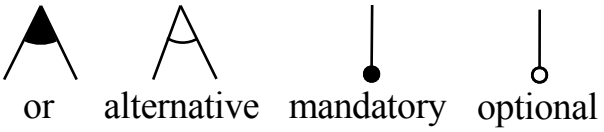


FIGURE 2.2: Graphical Notation of Relationships Between Features

Optional feature: a feature that may or may not be selected when its parent is selected. In the graphical representation, this type of feature is represented by a simple edge from the parent feature ending with an unfilled circle. In Figure 2.3, B may or may not be selected when A is selected. For example, the selection of Catering is optional in the feature diagram shown in Figure 2.1.

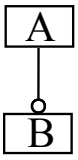


FIGURE 2.3: An Optional Feature

Mandatory feature: the feature must be selected if its parent is selected. In the graphical representation, this type of feature is represented by a simple edge from the parent feature ending with a filled circle. In Figure 2.4, B must be selected when A is selected. For example, Transportation is a mandatory feature in the feature diagram shown in Figure 2.1.

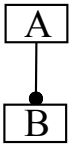


FIGURE 2.4: A Mandatory Feature

Alternative feature: exactly one of the features in this group must be selected if its parent is selected. In the graphical representation, each set of alternative features is represented by an arc, as shown in Figure 2.5, where exactly one of the features B or C must be selected if A is selected. In Figure 2.1, there are two types of Accommodation; exactly one of them must be chosen.

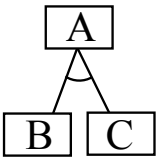


FIGURE 2.5: An Alternative Group

Or feature: at least one feature in this group must be selected if its parent is selected. In the graphical representation, each set of or features is represented by a black-filled arc, as shown in Figure 2.6, where at least one of the features B or C must be selected if A is selected. In Figure 2.1, there are two Transportation types, where at least one of them must be selected.
In addition to the hierarchy relations of the model, it is possible to add relations between features to express dependencies. For example:

- Requires: the selection of a source feature implies the selection of its target feature.
- Excludes: the selection of a feature implies the non-selection of another feature, which means that the two features cannot be part of the same configuration.

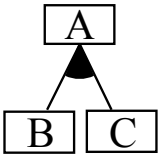


FIGURE 2.6: An Or Group

2.1.2 Configurations

Modelling variability of products is one of the most important aspects of feature modelling. Depending on the relation types (constraints) in a feature model, variability defines the valid combinations of features, which are called configurations.

The validity of a configuration is determined by selecting features in a manner that satisfies the variability constraints defined by the feature diagram. For example, in Figure 2.1, Hotel and Caravan are mutually exclusive and cannot be selected at the same time. The number of possible configurations grows as the variability increases. Dealing with feature model configurations has generated a lot of interests for many researchers. Various approaches have been proposed to deal with deriving feature model configurations [13–17]. Moreover, several tools have been developed to help reduce the complexity of the configuration process by automating the feature selection process [18–24].

In our work, we use feature models in negotiations to specify available configurations of negotiated services. Deriving feature model configurations is one of the main aspects of our research. We used the configuration techniques to support the variability of negotiated services in order to increase the flexibility of services negotiation.

2.1.3 Methods

In the literature, there are a number of feature modelling methods, such FODA [10], FeatuRSEB [25], FORM [12] and the method proposed by Czarnecki et al. in [11]. Feature-oriented domain analysis (FODA) is considered to be the foundation of all feature modelling methods [26]. In the following, we provide a brief description of each method.

FODA: FODA [10] defines three relationships between features. The basic relationship that can occur between features is the *consists-of relation*. This relation represents the mandatory features. *Optional features* can be used to represent that a feature may be selected but is not required. *Alternative features* can be used to indicate that exactly one of the features in the set of alternative features must be

selected. In addition, FODA defines two kind of composition rules: requires and incompatible (excludes) rules.

FeatuRSEB: Griss et al. in [25] presented an extension of FODA feature diagrams. They used similar semantics to the FODA diagrams with new graphical notations. The decomposition operator *or* is added to allow the selecting of one or more of the decomposed features.

FORM: The feature-oriented reuse method (FORM) [12] is an extension of FODA. One of the changes is the addition of two types of graphical relationships: generalisation/specialisation and implemented-by.

Czarnecki et al.: Czarnecki et al. [11] have studied feature diagrams in the context of Generative Programming. Their conceptualisation of features is based on FODA. Their feature diagram is a rooted tree with new graphical notation. They categorise features as mandatory, optional, alternative, and or-features. As we discussed in Section 2.1.1, the modelling method used here is the one proposed by Czarnecki et al., for its simplicity and clarity.

2.1.4 Feature Modelling and Web Services

Feature models have been used in a wide variety of applications. In this section, we focus on the use of feature models in web services to provide the flexibility in service specification and simplify customisation processes. Several approaches have been proposed to use feature models in web services specification and customisation. In [27], a feature-oriented approach for web services customisation has been proposed

to reduce complexity, automate validation and dynamic deployment. Service customisation is defined as activity performed by service consumers to customise service interfaces described by the Web Service Description Language (WSDL). Firstly, a service provider develops a customisable service using a feature model. Then, the feature model is published to service registries. Thirdly, a service consumer discovers the feature models of services it can customise. The approach was illustrated by a scenario of a news posting web service that content providers use to submit news entries to a Content Management System (CMS). A Content Provider can choose either "Direct" or "External Resource" features for posting news entry. He can optionally choose to update posting status with two alternative features: "Frequent Update" or "On Demand Query". He can also optionally request embedded format. Robak et al. [28] proposed the use of feature diagrams for modelling flexibility of web services. The knowledge contained in a feature diagram is then used to describe the commonality and variability of web services that can be dynamically customised for individual consumers.

In [29], the authors presented a methodology to model orchestration variability using a feature diagram. The feature diagram specifies a product line of orchestrations represented as configurations of invoked/rejected services. This methodology applied to the crisis management system case study. The Crisis Management System (CMS) feature diagram contains several features to represent the crisis types such as "Fire", "Car Accident" and "Theft". It also consists of features to represent the communications such as "GSM Telephony" and "GPS Location".

Naeem et al. in [30] proposed to use feature modelling techniques to specify the variability of provided and required services in order to increase the flexibility of the matching process. The feature models have been interpreted as linear logic formulas to provide the semantics for matching. They used an on-line travel agent service as a case study to illustrate their approach. The travel agent offers "Hotel" and "Flight" reservations. It also offers an optional "Transport" for hotel reservations.

Inspired by these works, in this thesis, we propose to use feature models to increase the flexibility of web services negotiation.

2.2 Game Theory

Game theory is a mathematical tool that analyses the strategic interactions among multiple decision-makers [31]. In [32], game theory is defined as "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers". Game theory can be applied whenever the actions of two or more parties are interdependent [33]. The mathematical theory of games was invented by John von Neumann and Oskar Morgenstern in [34], where they introduced the method of finding mutually consistent solutions for two-person zero-sum games. Nowadays, game theory has a wide range of applications, including economics, computer science, engineering, political science and biology [35]. We will explain some important concepts related to our work in the following sections.

2.2.1 Basic Concepts

In order to define a game in game theory, the type of interaction in the game should be considered. There are two main types of games that depend on the interaction types, strategic games and extensive-form games. In strategic games, players act simultaneously whereas, in extensive games, players choose their actions sequentially [31]. Simultaneous games are called static games whereas sequential games are called dynamic games.

A game in strategic form is usually represented as a matrix in which players choose their strategies at the same time. It has three elements: a set of players, a set of pure strategies for each player and payoff functions. Each player chooses a strategy

without any knowledge of what the other player chooses. A game in an extensive form is represented as a decision tree to model dynamic structure. It provides a complete description of the choices available for each player and when each can move. It has four elements: a set of players, a set of terminal histories (which is the set of all complete sequences in the game), a player function, which indicates who moves after each non-terminal history, and payoff functions to assign a payoff to players at each terminal history.

In our research, we are interested in games in the extensive form, where players choose their actions sequentially. We will discuss extensive-form games in more detail in Section 2.2.3.

In the following, we list some important concepts in game theory and provide a brief definition of each:

- **Player:** a player represents a decision-maker in the game.
- **Strategy:** in strategic-form games, a strategy is one of the possible actions of a player whereas in extensive-form games, a strategy is a complete plan of actions for a player in every possible state of the game.
- **Strategy profile:** this is a combination of strategies, one strategy for each player.
- **Payoff:** payoff or utility represents the outcome (a real number) for each player in the game. The payoff function is a function that indicates the outcome for each player in each strategy profile or combination.

2.2.2 Nash Equilibrium

This concept was developed in 1950 by John Nash [36], who showed that every finite non-cooperative (zero-sum) game has an equilibrium point. In [33], a Nash

Equilibrium is defined as "a list of strategies, one for each player, which has the property that no player can unilaterally change his strategy and get a better payoff". In equilibrium, each player is acting optimally with respect to the other players' behaviour. In other words, each player is playing a best response to the other players' strategies. Nash equilibrium provides us with a solution to analyse games. However, it is not always the best possible solution globally that could be achieved. In many cases, the Nash equilibrium is not Pareto optimal. Moreover, there may be more than one Nash equilibrium or there may be no Nash equilibrium at all.
In our research, we use the Nash equilibrium to determine the individually optimal actions for each negotiator at each stage of the negotiation game. This solution is not necessarily the best joint outcome.

2.2.3 Extensive-Form Games

Extensive-form games are applicable when decisions are sequential rather than simultaneous. An extensive-form game is represented as a rooted tree, which is called a game tree with nodes representing decision points and edges between nodes representing players' moves. The terminal nodes of the game tree hold the payoffs for each player at the end of every possible play. Figure 2.7 shows an example of extensive-form game.
We now provide a formal definition of extensive-form games with perfect information. This definition follows the one given by Osborne et al. in [37].

Definition 2.1. (**Extensive-Form Game** [38–40]) A finite extensive-form game G with perfect information is a quadruple $G = (N, H, P, (u_i))$ containing the following components:

- A finite set $N = \{1, 2, ..., n\}$ of **players**.

- H is a set of sequences, the possible **histories** such that
  - $\varnothing \in H$.
  - For $h \in H$, we denote $A(h) = \{a \mid ha \in H\}$ to be the set of **actions** available at h.
  - $Z \subseteq H$ denotes the set of **terminal histories**, i.e., they are not subhistories of any other sequence.
- A **player function** P that assigns to each nonterminal history $h \in H \backslash Z$ a member $P(h) \in N$. $P(h)$ is the player who acts at the history h.
- For each player $i \in N$, a **utility function** $u_i : Z \to \mathbb{R}$ that denotes the payoff for player i at each possible terminal history.

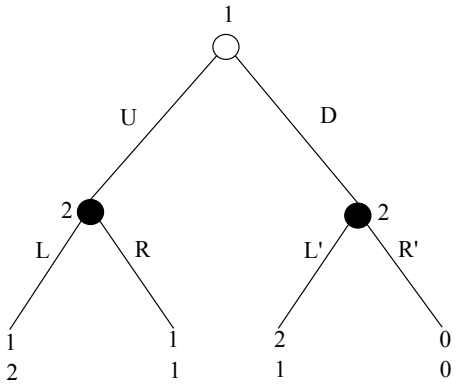


FIGURE 2.7: Extensive-form Game Example

We illustrate the above definition for the game shown in Figure 2.7.

$$N = \{1, 2\} \text{ (The set of } \textbf{players}\text{)}$$
$$H = \{\varnothing, U, D, (U, L), (U, R), (D, L'), (D, R')\} \text{ (The set of } \textbf{histories}\text{)}$$
$$A(\varnothing) = \{U, D\},\ A(U) = \{L, R\},\ A(D) = \{L', R'\}$$
$$Z = \{(U, L), (U, R), (D, L'), (D, R')\} \text{ (The set of } \textbf{terminal histories}\text{)}$$
$$H \backslash Z = \{\varnothing, U, D\}$$

$$P(\varnothing) = 1,\ P(U) = 2,\ P(D) = 2\ (\textbf{Player function})$$

$$u_1(U, L) = 1,\ u_1(U, R) = 1,\ u_1(D, L') = 2,\ u_1(D, R') = 0\ (\textbf{Payoff function}\text{ for player 1})$$

$$u_2(U, L) = 2,\ u_2(U, R) = 1,\ u_2(D, L') = 1,\ u_2(D, R') = 0\ (\textbf{Payoff function}\text{ for player 2})$$

In our research, we analyse our graph transformation games as extensive-form games.

2.2.3.1 Strategies

In game theory, a strategy is one of the most important concepts. In extensive-form games, a strategy is a complete contingent plan explaining what a player will do at every situation [41, 42]. We now provide a formal definition of a pure strategy in an extensive-form game.

Definition 2.2. (**Pure Strategy** [42]) A pure strategy for player i in an extensive-form game is a function $s_i : H_i \to A_i$ such that H_i is the set of histories at which player i takes an action, A_i is the set of actions available to player i and $s_i(h) \in A(h)$ for each $h \in H_i$.

Consider the game shown in Figure 2.7. Player 1 has two strategies: (U, D) and Player 2 has four strategies: $(L, L'), (L, R'), (R, L'), (R, R')$.
A strategy profile is a collection of strategies, one for each player. Let S_i denote the set of pure strategies for player i and let S denote the set of strategy profiles. Every strategy profile $s \in S$ defines a unique outcome path $O(s)$ showing how the game will proceed [43]. For example, in the game in Figure 2.7, the outcome paths are as follows:

$$O(U, (L, L')) = (U, L),$$

$$O(U,(L,R')) = (U,L),$$
$$O(U,(R,L')) = (U,R),$$
$$O(U,(R,R')) = (U,R),$$
$$O(D,(L,L')) = (D,L'),$$
$$O(D,(L,R')) = (D,R'),$$
$$O(D,(R,L')) = (D,L'),$$
$$O(D,(R,R')) = (D,R').$$

Once the strategies are obtained for every player, the next step is finding the payoffs. The payoffs can be obtained using the **outcome path** of the strategy profile. Thus, the payoff of the player i is $u_i(O(s))$ given a strategy profile s. For example, in the game in Figure 2.7, the payoff of player 1 for the outcome path $O(U,(L,L'))$ is: $u_1(O(U,(L,L')) = 1$.

Definition 2.3. (**Nash Equilibrium** [44]) A Nash equilibrium of an extensive-form game with perfect information $G = (N, H, P, (u_i))$ is a strategy profile s^* such that for each player $i \in N$, $u_i(O(s^*_{-i}, s^*_i)) \geq u_i(O(s^*_{-i}, s_i))$ for all $s_i \in S_i$.

2.2.3.2 Subgame Perfect Equilibrium

In game theory, a **subgame perfect equilibrium** is a refinement of a Nash equilibrium used in dynamic games that incorporate sequential rationality [45]. Sequential rationality means that it is common knowledge that each player will act rationally at each future state where he moves. Subgame perfect equilibrium concept was introduced by Reinhard Selten in 1965 [46].
First let us define a subgame. A subgame is a subset of an extensive-form game that constitutes a valid extensive-form game. Formally,

Definition 2.4. (**Subgame** [47]) The **subgame** of the extensive-form game with perfect information $G = (N, H, P, (u_i))$ **that follows the history** $h \in H \backslash Z$ is the

extensive-form game $G|_h = (N, H|_h, P|_h, (u_i|_h))$ that satisfies the following conditions:

- $h' \in H|_h \Leftrightarrow (h, h') \in H$.
- $P|_h(h') = P(h, h')$ for any $h' \in H|_h$.
- $u_i|_h(h') = u_i(h, h')$ for any terminal history $h' \in Z|_h \subset H|_h$.

Figure 2.8 shows a subgame $G|_U$ of the game shown in Figure 2.7 that follows a history U.

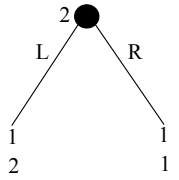


FIGURE 2.8: Subgame of the Extensive-form Game Example

We illustrate the above definition for the subgame shown in Figure 2.8.

$$N = \{1, 2\}$$
$$H|_U = \{\varnothing, L, R\}$$
$$Z|_U = \{L, R\}$$
$$H|_U \backslash Z|_U = \{\varnothing\}$$
$$P(\varnothing) = 2$$
$$u_1|_U(L) = 1, u_1|_U(R) = 1, u_2|_U(L) = 2, u_2|_U(R) = 1$$

A strategy profile is a subgame perfect equilibrium if it represents a Nash equilibrium of every subgame of the original game. Let $S_i|_h$ denote the set of strategies for player i in the game $G|_h$.

Definition 2.5. (**Subgame Perfect Equilibrium** [44]) A subgame perfect equilibrium of an extensive-form game with perfect information $G = (N, H, P, (u_i))$ is a strategy profile s^* such that for each player $i \in N$ and each $h \in H \backslash Z$ for which $P(h) = i$ we have, $u_i|_h(O_h(s^*_{-i}|_h, s^*_i|_h)) \geq u_i|_h(O_h(s^*_{-i}|_h, s_i))$ for all $s_i \in S_i|_h$.

In the game in Figure 2.7, $(D, (L, L'))$ is the unique subgame perfect equilibrium.

2.2.3.3 Backward Induction

Backward induction is a powerful technique that has been applied to many problems in computer science. It is known as Zermelo's algorithm, after Ernst Zermelo (1871-1953), who used it to analyse the game of chess [48]. In game theory, backward induction is a method used to solve a finite extensive-form game with perfect information by computing subgame perfect equilibria [33]. Zermelo's theorem states that "every finite game of perfect information has a pure strategy Nash equilibrium that can be derived through backwards induction" [49].
The procedure of backward induction is based on the idea that the players will start at the end of the game tree and determine the moves giving them the highest payoff, and work backward until reaching the beginning of the game tree [50].
In our work, we use backward induction to determine the optimal actions for the negotiators in our extensive-form graph transformation games, and to compute the subgame perfect equilibrium.

2.2.4 Stochastic Games

Stochastic games generalise Markov decision processes (MDPs) with multiple players and are a basic model in game theory [51]. In stochastic multi-player games (SMGs), the successor states are either chosen randomly or nondeterministically and the

choice of actions at the states may belong to different players of the game [52]. Turn-based stochastic multi-player games are a special case of SMGs where the choice of action at each state is under the control of exactly one player [53]. We now provide a formal definition of turn-based stochastic multi-player games, which we consider in analysing our graph transformation games.

Definition 2.6. (**Turn-based Stochastic Game** [54]) A (turn-based) stochastic multi-player game (SMG) is a tuple $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i\in\Pi}, \Delta, AP, \chi \rangle$, where:

- Π is a finite set of **players**.
- S is a finite, non-empty set of **states**.
- A is a finite, non-empty set of **actions**.
- $(S_i)_{i\in\Pi}$ is a partition of S.
- $\Delta : S \times A \to \mathcal{D}(\mathrm{S})$ is a (partial) **transition function**, where $\mathcal{D}(\mathrm{S})$ is a discrete probability distribution over states.
- AP is a finite set of **atomic propositions**.
- $\chi : S \to 2^{AP}$ is a **labelling function**.

In each state $s \in S$ of the SMG $\mathcal{G}$, the set of available actions is denoted by $A(s) \stackrel{\text{def}}{=} \{a \in A \mid \Delta(s,a) \neq \bot\}$. We assume that $A(s) \neq \varnothing$ for all s. The choice of action to take in s is under the control of exactly one player, namely the player $i \in \Pi$ for which $s \in S_i$. Once action $a \in A(s)$ is selected, the successor state is chosen according to the probability distribution $\Delta(s,a)$. A path of $\mathcal{G}$ is a possibly infinite sequence $\lambda = s_0 a_0 s_1 a_1 ...$ such that $a_j \in A(s_j)$ and $\Delta(s_j, a_j)(s_{j+1}) > 0$ for all j. A finite path is a finite such sequence.

Definition 2.7. (**Strategies** [54]) A strategy for player $i \in \Pi$ in $\mathcal{G}$ is a function $\sigma_i : (SA)^* S_i \to \mathcal{D}(\mathrm{A})$ which, for each path $\lambda \cdot s \in \Omega^+_{\mathcal{G}}$ where $s \in S_i$ and $\Omega^+_{\mathcal{G}}$ is the set of all finite paths, selects a probability distribution $\sigma_i(\lambda \cdot s)$ over $A(s)$.

Definition 2.8. (**Strategy Profile** [54]) A strategy profile $\sigma = \sigma_1, ..., \sigma_{|\Pi|}$ comprises a strategy for all players in the game.

Definition 2.9. (**Rewards** [55]) A reward function $r : S \to \mathbb{Q}^n$ assigns a reward to each state s of the game $\mathcal{G}$. Transition/action rewards are also possible in SMGs, which can easily be encoded by adding an auxiliary state per transition/action to the model.

2.2.5 PRISM-games Model Checker

PRISM-games [56] is an extension of the PRISM model checker [57]. It is the first tool to provide modelling for stochastic multi-player games (SMGs) [56]. It supports the verification of probabilistic systems as turn-based zero-sum stochastic games. The games are specified using an extension of the existing PRISM modelling language, which is a guarded-command-based language inspired by the Reactive Modules formalism [58]. It is built upon Markov Decision Processes (MDPs). A model in PRISM-games consists of modules that describe the behaviour of the players. The state is determined by a set of variables and the behaviour is specified by guarded commands.

PRISM-games specifies properties in the temporal logic rPATL [53], which combines features of the multi-agent logic ATL, the probabilistic logic PCTL and operators to reason about several different notions of reward measures, numerical properties and precise probability values [59].

rPATL: rPATL (Probabilistic Alternating-time Temporal Logic with Rewards) is a CTL-style branching-time temporal logic for expressing quantitative properties of SMGs [54]. In rPATL, state formulae (ϕ) and path formulae (ψ) are distinguishable. The coalition operator $\langle\langle C \rangle\rangle$ of ATL [60] has been adopted, combining it with the probabilistic operator $P_{\bowtie q}$ and path formulae from PCTL [61, 62] and a generalisation of the reward operator $R^{r}_{\bowtie x}$ from [63]. A typical rPATL property is [64]:

$$<< 1 >> \text{P}>=0.99 [\ \text{F}<=5\ \text{c}=2\ ]$$

which states that player 1 has a strategy to ensure that the probability of reaching a state satisfying c=2 within five time-steps is at least 0.99, regardless of the strategies of any other player. Another example of reward-based properties [64] is as follows:

$$<<\text{p1}>>\ \text{R}\{\text{"r"}\}\text{max}=?\ [\ \text{F}\ \text{"success"}]$$

which asks, "What is the maximum expected amount of reward "r" accumulated until reaching "success"?"

The semantics and operators of rPATL have been discussed extensively in [52–54]. Currently, PRISM-games supports turn-based, perfect-information SMGs. It extends the existing PRISM model checker by providing a modelling language for stochastic multi-player games. It provides a graphical user interface with model editor. It also provides a discrete-event simulation tool and graph-plotting functionality.

The core functionality of the tool comprises methods for verifying quantitative properties of stochastic games [53, 59] and support for synthesising optimal player strategies, exploring or exporting them, and verifying other properties under the specified strategy.

2.2.6 Game Theory in Negotiation

Game theory is concerned with the mathematical models of behaviour in strategic situations. It studies interactive decision-making in which self-interested agents interact with each other, taking into account each other's strategic decisions [65]. In negotiation, game theory offers a very powerful tool for the design of the negotiation process. Since the agents in the negotiation are self-interested, trying to optimise their own outcomes while taking into account the decisions that other agents may take, game theory gives a way of formalising and analysing these negotiation situations.

Using game theory in negotiation assumes that the negotiators are the players of the game who have individual and joint interests. These interests are measured by a payoff function. The agreement may benefit both negotiators but they have different preferences for different outcomes [66]. Game theoretic techniques can be applied to two key problems: the design of an appropriate protocol that models the interactions between the negotiators and the design of a strategy that negotiators can use while negotiating [67].

In our research, we use game theory to formalise and analyse our negotiation games.

2.3 Graph Transformation

Graphs and diagrams have been used to represent a variety of problems in computer science and software engineering [68]. They provide a simple and clear structure of systems and services. Graph Transformation Systems (GTS) have been used to model the dynamic behaviour of systems where graphs model the systems' states and their evolution is specified by graph transformation rules [69]. The conceptual (type) level of the system is represented by a type graph and its instance level is

represented by an instance graph. A type graph is usually visualised using a class diagram in Unified Modelling Language (UML) [70]. An instance graph is visualised by an object diagram. Graph transformation rules describe pre and post conditions of operations.

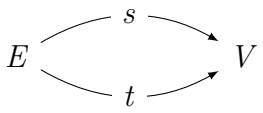
In our research, we use a type graph to describe the negotiation entities. The graph transformation rules are used to specify the changes to the negotiation state for each possible move.

In the following sections, we provide fundamental concepts of graphs and graph transformations.

2.3.1 Basic Concepts

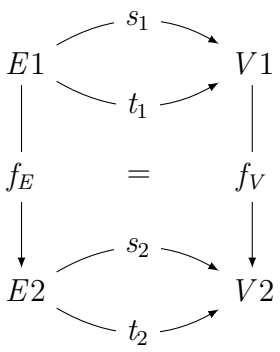
A (directed) graph consists of a set of vertices V and a set of edges E. Each edge has a source and a target vertex.

Definition 2.10. (**Graph** [71]) A graph $G = (V, E, s, t)$ consists of a set V of nodes (also called vertices), a set E of edges, and two functions $s, t : E \to V$, the source and target functions:



Graphs are related by graph morphisms, which preserve the source and target of each edge.

Definition 2.11. (**Graph Morphism** [71]) Given graphs G_1 , G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1,2$, a graph morphism $f : G_1 \to G_2$, $f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$:



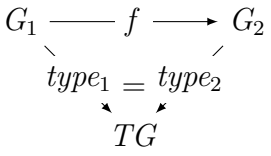
A graph morphism f is injective (or surjective) if both functions f_V , f_E are injective (or surjective, respectively); f is called isomorphic if it is bijective, which means both injective and surjective.

Fact 2.12. (**Composition of Graph Morphisms** [71]) Given two graph morphisms $f = (f_V, f_E) : G_1 \to G_2$ and $g = (g_V, g_E) : G_2 \to G_3$, the composition $g \circ f = (g_V \circ f_V, g_E \circ f_E) : G_1 \to G_3$ is again a graph morphism.

As discussed in Section 1.3, we developed a metamodel to define the negotiation entities. The metamodel can be conveniently expressed as a type graph, which defines a set of types for the nodes and edges of a graph.

Definition 2.13. (**Typed Graph** [71]) A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively.
A pair $(G, type)$ of a graph G together with a graph morphism type: $G \to TG$ is then called a typed graph.

Definition 2.14. (**Typed Graph Morphism** [71]) Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$:



Definition 2.15. (**Labelled Graph** [71]) A label alphabet $L = (L_V, L_E)$ consists of a set L_V of node labels and a set L_E of edge labels. A labelled graph $G = (V, E, s, t, l_V, l_E)$ consists of an underlying graph $G^0 = (V, E, s, t)$ together with label functions

$$l_V : V \to L_V \text{ and } l_E : E \to L_E.$$

In Figure 2.9 an example of a graph with node and edge labels is given.

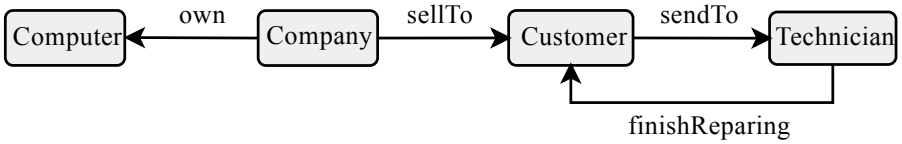


FIGURE 2.9: Labelled Graph Example

Definition 2.16. (**Labelled Graph Morphism** [71]) A labelled graph morphism $f : G_1 \to G_2$ is a graph morphism $f : G_1^0 \to G_2^0$ between the underlying graphs which is compatible with the label functions, i.e. $l_{2,V} \circ f_V = l_{1,V}$ and $l_{2,E} \circ f_E = l_{1,E}$.

A graph morphism from G_1 to G_2 is illustrated in Figure 2.10. The dashed vertical arrows represent the morphism's node and edge mapping components. This morphism is injective but not surjective.

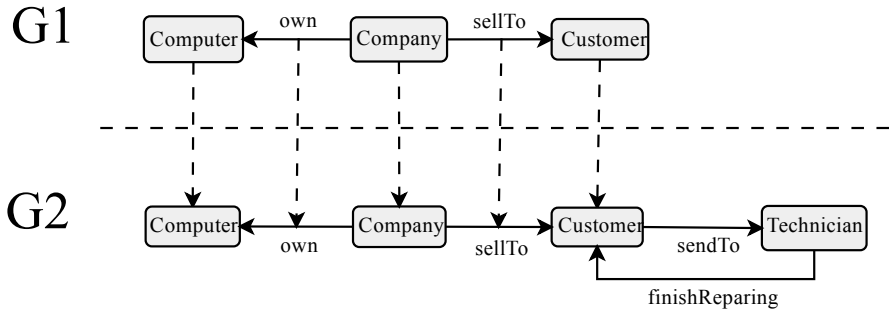


FIGURE 2.10: A Graph Morphism From G1 to G2

Graph transformation is most commonly defined in terms of category theory. It is important to show that graph structures lead to categories. For example [71]:

- The class of all graphs (as defined in Definition 2.10) as objects and of all graph morphisms (see Definition 2.11) forms the category **Graphs**, with the composition given in Fact 2.12, and the identities are the pairwise identities on nodes and edges.

- Given a type graph TG, typed graphs over TG and typed graph morphisms (see Definition 2.14) form the category $Graphs_{TG}$.

2.3.2 Algebraic Approach

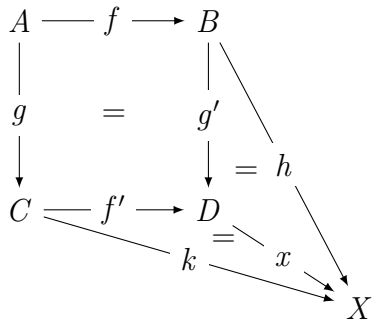
Various graph transformation approaches have been developed. A general approach is called the algebraic approach, where an entire subgraph can be replaced by a new subgraph. The algebraic approach is based on pushout constructions in the category Graphs of graphs. Pushouts are used to model the gluing of graphs, which is required to apply graph transformation rules to graphs.

Definition 2.17. (**Pushout** [71]) Given morphisms $f : A \to B$ and $g : A \to C$ in a category C, a pushout (D, f', g') over f and g is defined by:

- a pushout object D and

- morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$

such that the following universal property is fulfilled: for all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$:



We write $D = B +_A C$ for the pushout object D, where D is called the gluing of B and C via A.

2.3.2.1 Double Pushout Approach

We consider graph transformation based on the algebraic double-pushout (DPO) approach that covers the main ideas underlying the algebraic approach. Graph transformation is based on graph productions (rules), which describe a general way how to transform graphs. In the DPO approach, a production p consists of three graphs L, K and R. L and R are called the left-hand side and right-hand side respectively. K is referred to as the interface or gluing graph, which represents what the left and right hand sides have in common. The left-hand side L represents the preconditions of the rule, while the right-hand side R represents the postconditions. K represents a graph part that has to exist to enable the application of the rule.

Definition 2.18. (**Graph Production** [71]) A (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of (typed) graphs L, K, and R, called the left-hand side, gluing graph, and the right-hand side respectively, and two injective (typed) graph morphisms l and r. Given a (typed) graph production p, the inverse production is defined by $p^{-1} = (R \xleftarrow{r} K \xrightarrow{l} L)$.

Figure 2.11 gives an example of a graph transformation rule and a match for it in G. A graph transformation starts by finding a match m of L in the source graph

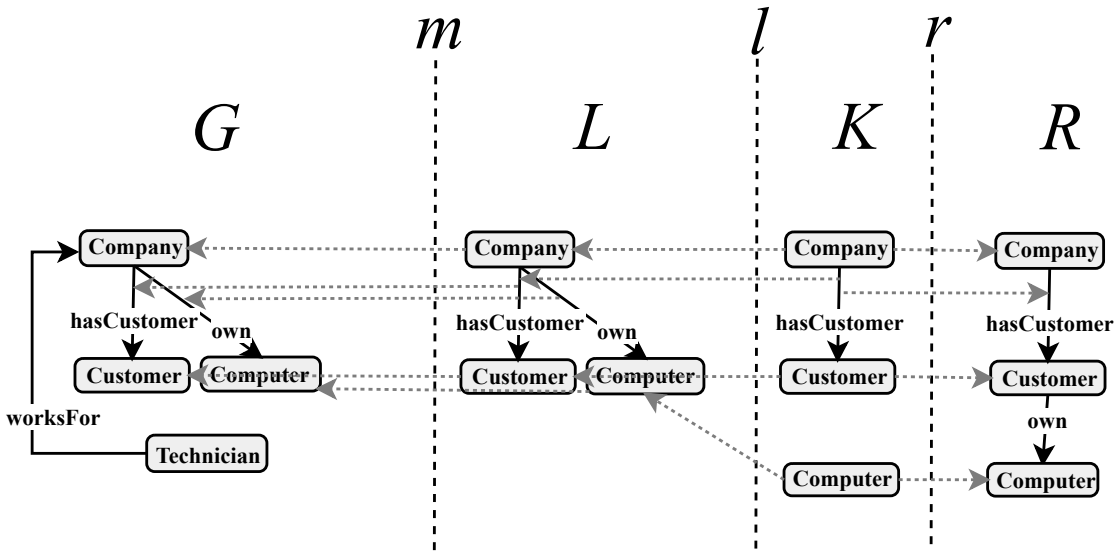


FIGURE 2.11: A Graph Transformation Rule *sell_to*

G. Then, $m(L \backslash l(K))$ are removed from G to create an intermediate graph D. The match m has to satisfy the gluing condition (see Definition 2.20). The graph $D = (G \backslash m(L)) \cup m(l(K))$ is obtained by removing the vertices and edges of L from G that are not in the image l. In the second step, a target graph H is produced by gluing $R \backslash l(K)$ and D.

Definition 2.19. (**Graph Transformation** [71]) Given a (typed) graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \rightarrow G$, called the match, a direct (typed) graph transformation $G \overset{p,m}{\Rightarrow} H$ from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushouts in the category *Graphs* (or $Graphs_{TG}$, respectively):

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ \downarrow m & (1) & \downarrow k & (2) & \downarrow n \\ G & \xleftarrow{\ f\ } & D & \xrightarrow{\ g\ } & H \end{array}$$

A sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ of direct (typed) graph transformations is called a (typed) graph transformation and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$.

Definition 2.20. (**Gluing Condition** [71]) Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, a (typed) graph G, and a match $m : L \to G$ with $X = (V_X, E_X, s_X, t_X)$ for all $X \in L, K, R, G$, we can state the following definitions:

- The gluing points GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- The identification points IP are those nodes and edges in L that are identified by m, i.e. $IP = \{v \in V_L \mid \exists\, w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$.
- The dangling points DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to $m(L)$, i.e. $DP = \{v \in V_L \mid \exists\, e \in E_G \backslash m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.

Now we define the (typed) graph transformation systems that we consider in our approach to implement our graph transformation games. A graph transformation system is defined by applying a set of productions on a graph.

Definition 2.21. (**Graph Transformation System** [71]) A typed graph transformation system $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P.

2.3.3 Typed Attributed Graphs

Graph transformation has been used as a meta-language to specify and implement visual modelling techniques, like the UML [72]. In most visual modelling techniques, (typed) attributed graphs are used as a representation mechanism [73]. An attributed graph can be seen as a graph where attributes are assigned for the nodes

and edges [74]. Several different concepts for typed and attributed graph transformation have been proposed (e.g. [72–74]). These approaches followed the algebraic approach to provide formal definitions of attributed graph transformation. In [72], the authors introduced a new concept, which is called, *E-graphs*, which allows both node and edge attributions.

Definition 2.22. (**E-graph and E-graph Morphism** [71]) An E-graph G with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$ consists of the sets:

- V_G and V_D, called the graph and data nodes (or vertices), respectively;
- E_G, E_{NA}, and E_{EA}, called the graph, node attribute, and edge attribute edges, respectively;

and the source and target functions:

- $source_G : E_G \to V_G$, $target_G : E_G \to V_G$ for graph edges;
- $source_{NA} : E_{NA} \to V_G$, $target_{NA} : E_{NA} \to V_D$ for node attribute edges; and
- $source_{EA} : E_{EA} \to E_G$, $target_{EA} : E_{EA} \to V_D$ for edge attribute edges

Consider the E-graphs:
G^1 and G^2 with $G^k = (V_G^k, V_G^k, E_G^k, E_G^k, E_G^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ for $k = 1, 2$. An E-graph morphism $f : G1 \to G2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_i^1 \to V_i^2$ and $f_{E_j} : E_j^1 \to E_j^2$ for $i \in \{G, D\}$, $j \in \{G, NA, EA\}$ such that f commutes with all source and target functions, for example $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$.

Definition 2.23. (**Attributed Graph and Attributed Graph Morphism** [71]) Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An

attributed graph $AG = (G, D)$ consists of an E-graph G together with a DSIG-algebra D such that $\dot{\cup}_{s \in S'_D} D_s = V_D$.

For two attributed graphs $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$, an attributed graph morphism $f : AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \to G^2$ and an algebra homomorphism $f_D : D^1 \to D^2$

Definition 2.24. (**Typed Attributed Graph and Typed Attributed Graph Morphism** [71]) Given a data signature $DSIG$, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final DSIG-algebra. A typed attributed graph (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.

A typed attributed graph morphism $f : (AG^1, t^1) \to (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \to AG^2$ such that $t^2 \circ f = t^1$.

In Figure 2.12, we give an example of an attributed type graph where we have graph nodes and data nodes. Attributes can be inscribed within the object vertex they belong to, as demonstrated in Figure 2.13a. In Figure 2.13b, we have an example of a typed attributed instance graph.

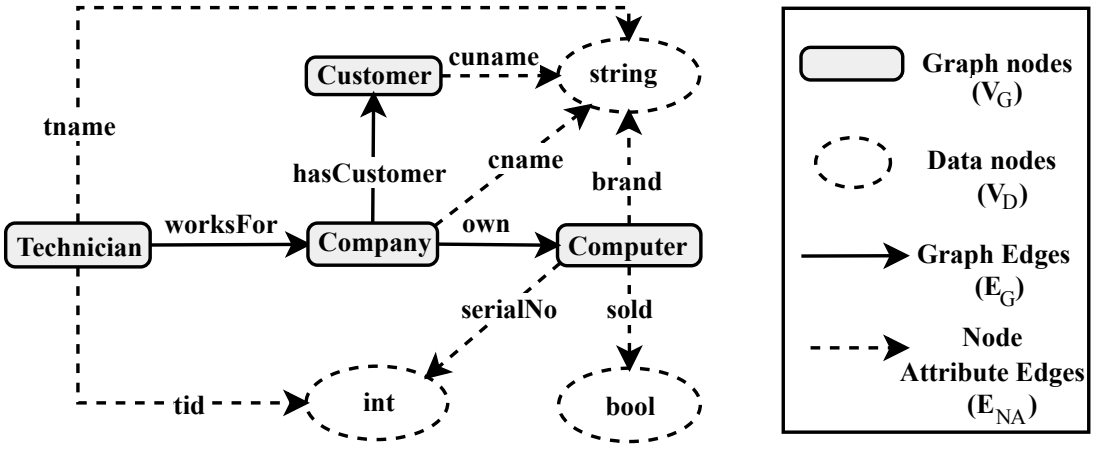


FIGURE 2.12: Attributed Graph Example

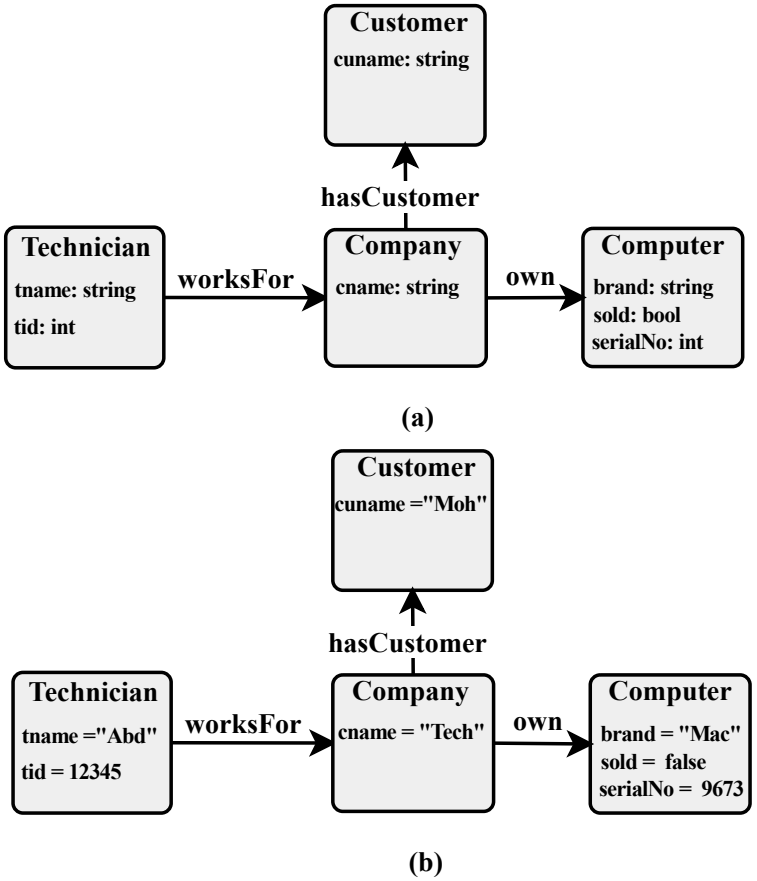


FIGURE 2.13: Attributed Graph (a), and Its Instance (b)

In our approach, we use the Henshin transformation tool [75], which has its roots in attributed graph transformations. It offers a formal foundation for validation of EMF model transformations. The EMF model can be seen as a type graph with attribution, inheritance and multiplicities and its instance model can be seen as a typed attributed graph [76].

2.3.4 Henshin Language and Tools

In this section, we introduce the Henshin transformation language and tool environment that we use in our implementation of graph transformation games. Henshin [75] is an Eclipse plug-in that supports visual modelling and execution of rule-based

EMF model transformations. Henshin extends the transformation language of EMF Tiger [77]. Its transformation rules are supported by powerful application conditions and flexible attribute computations. They can be structured by means of transformation units that can control the order of rule applications. Currently, Henshin comprises three modules:

1. a tree-based and a graphical editor for defining transformation systems.
2. a runtime component, currently consisting of an interpreter engine.
3. a state space generator and an extension point for analysis tools.

Before defining rules in Henshin, a metamodel should be created using the EMF Eclipse plug-in. The rules can be applied to an instance model of the metamodel, which can also be created using EMF tools.
Henshin offers a visual syntax, and sophisticated editing functionalities, execution and analysis tools. There are two editors to define model transformations in Henshin: i) a tree-based editor, generated by EMF itself, and ii) a graphical editor, implemented using GMF.
The graphical editor shows rules in an integrated manner with the pattern to find (left-hand side, LHS), the resulting pattern (right-hand side, RHS) and application conditions. At the top of every rule, its name and parameters are specified. Inside a rule, we create Nodes, Edges and Attributes. The nodes represent the classes of the metamodel and the edges are used to specify the link between nodes. Nodes and edges are annotated with stereotypes (actions). There are a number of actions:

- preserve: the node/edge is preserved during the rule application.
- delete: delete an existing node/edge after the rule application.
- create: create a new node/edge after the rule application.

- forbid: forbid the existence of a node/edge during the rule application.

Figure 2.14 illustrates how the rule *sellTo* would be represented in Henshin. In this example, we show how the attribute *sold* value can be changed during the rule transformation.

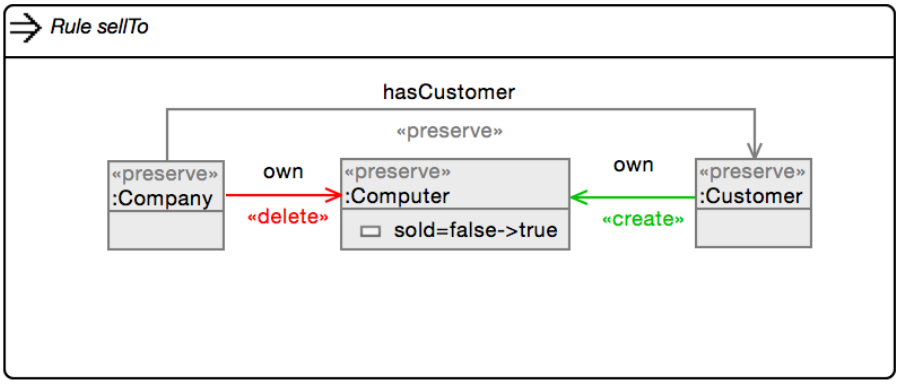


FIGURE 2.14: *sellTo* Transformation Rule

Henshin also provides tools for generating and analysing the state spaces of model transformation. It starts from some initial states and executes the transformation rules until reaching the terminal states, where no rules can be applied. Figure 2.15 illustrates the transformation rules of a graph transformation system called *Computer*. In Figure 2.16, we show the generated state space for the *Computer* graph transformation system. Here, we use the instance graph in Figure 2.13b as an initial graph.

2.4 Summary

In this chapter, we provided detailed information about feature modelling. We explained the semantics of feature models, the configuration techniques, the feature modelling methods and the tools that support the creation of feature diagrams. We

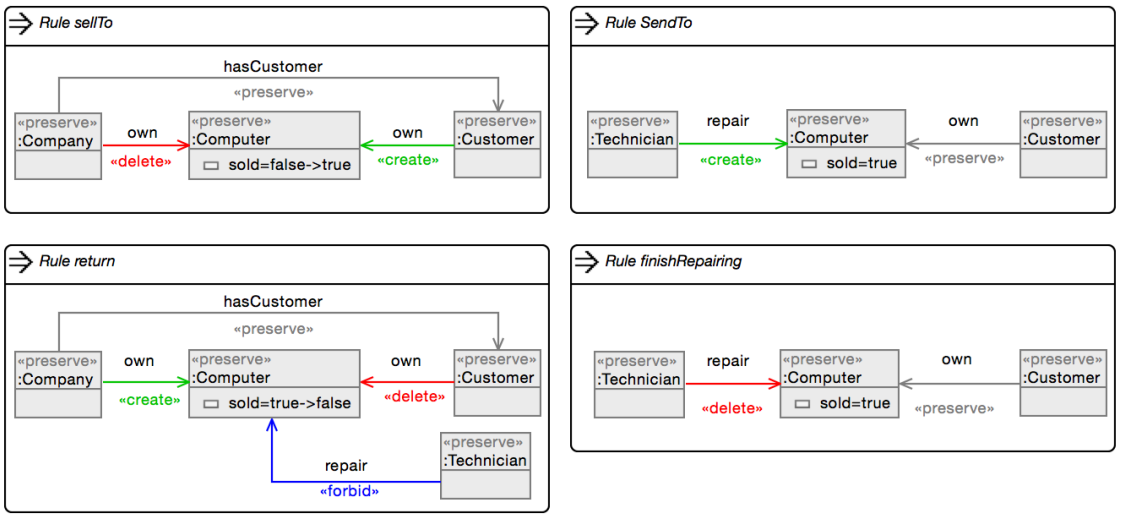


FIGURE 2.15: *Computer* Graph Transformation System Rules

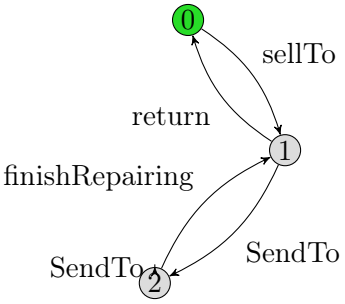


FIGURE 2.16: *Computer* State Space

also discussed the use of feature models in web services to support service customisation.

We moved on by giving basic game theoretic concepts related to our work, including the definition of extensive-form games and stochastic games. We discussed the PRISM-games model checker that we use in our analysis of our graph transformation games. We also presented the use of game theory in negotiation.

At the end, we provided information about graph transformation and the algebraic approach, including the formal definitions of graphs, graph morphism, graph production and graph transformation system. We discussed typed attributed graphs and their usage in our approach. Finally, we presented an overview about the Henshin

transformation language and tool environment that we use in our implementation.

Chapter 3

Graph Transformation Games for Negotiating Features

The negotiation games proposed in this chapter model the interaction between a provider and a requestor who use feature models to represent service configurations. The approach seeks to enable the service provider and requestor to discuss their preferences in order to reach an agreement that benefits both of them. The negotiation depends on selecting and deselecting features which represent the services and their characteristics.

The negotiation process may be represented graphically to show how a negotiation state may actually change and evolve. For dynamic graph-like structures, graph transformation provides a formal specification technique which supports visual and rule-based transformation for graph structures. Therefore, graph transformation games are proposed in order to model our negotiations and to analyse the providers and requestors' strategic choices. We aim to provide a negotiation framework using game-theoretic techniques. A graph transformation game is a state-based game in which the states of the game are given by graphs. The rules of the game are defined by graph transformation rules which determine the players' available actions.

In this chapter, we start with a scenario to motivate and explain the proposed approach in Section 3.1. In Section 3.2, we discuss the feature negotiation games. Section 3.3 presents the definition of graph transformation games. Section 3.4 provides a detailed explanation of the implementation of the graph transformation games. Section 3.5 concludes the chapter.

3.1 Motivating Example

To illustrate the approach, a small example is presented in this section and is used as a running example throughout this thesis. It is concerned with the negotiation between a travel agency (service provider) and a service requestor aiming to reach an agreement and establish a contract. The travel agency offers different packages of holiday services with specific variability of Location, Accommodation and Transportation described in the feature model shown in Figure 3.1. The root feature Holiday Services denotes the main concept of the feature model. Subsumed under the root feature are four child features: Location, Transportation, Accommodation and Catering. Location, Transportation and Accommodation are mandatory features whereas Catering is an optional feature. Location is a required feature as it indicates the location of the preferred holiday. Transportation has two features in an Or group: Airplane and Train. The requestor of this service is required to choose at least one transportation type if they select Transportation. Accommodation has two features in an Alternative group: Caravan and Hotel. The travel agency offers two types of accommodation; only one of them can be selected in a holiday service. Catering can be offered as an Optional feature.

Let us consider that the service requestor is interested in booking a holiday with specific requirements. For example, the requestor wishes to hire a private car for transportation and a hotel for accommodation for a specific location.

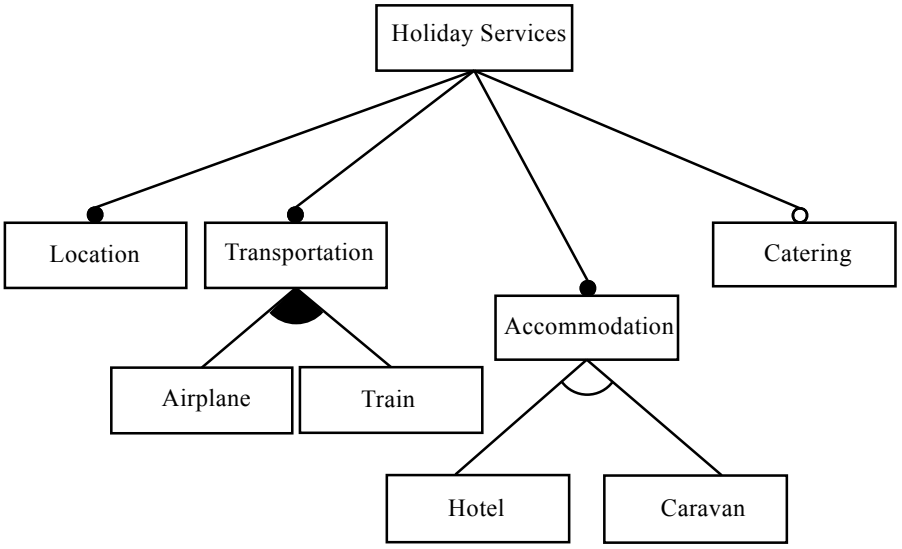


FIGURE 3.1: Travel Agency Feature Model

If we consider these requirements as a set of features, it is obvious that the travel agency cannot provide these exact requirements as they do not offer a car for transportation.

Therefore, in an inflexible scenario in which the requestor is not willing to change their original requirements according to the available alternative offers, no business will be conducted. Losing the deal may affect both the service provider, who wants to gain more profits, and the service requestor, who is interested in making a deal with a certain provider for some reasons, such as reliability, price, etc.

For the service requestor, in today's business, it is necessary for them to be more flexible and able to change their original requirements according to the available offers. Therefore, a negotiation allows them to explore the possible solutions and reach an agreement.

In this example, the provider initially offers airplane as an alternative feature to the car requested. Then, the requestor can start a negotiation with the provider to add or remove existing features. The provider can also start suggesting changes to the original requirements which may be of interest to the requestor.

3.2 Feature Negotiation Games

In our negotiation, there are two negotiators, service provider and service requestor, a feature model describing the supplied services, a configuration representing a valid combination of services, and a set of negotiation actions such as propose, accept proposal and reject proposal. The negotiators can make proposals to add and withdraw features to/from the configuration. They can respond to proposals by either accepting or rejecting them.
In our running example, there are four available proposals which can be accepted or rejected according to the negotiators' preferences, as follows:

1. Add Catering feature: the negotiators can propose adding Catering to the holiday as it is an optional feature.

2. Add Train feature: the negotiators can propose adding Train as another type of transportation. This is correct according to the travel agency feature model (Or group).

3. Withdraw Airplane feature: the negotiators can propose withdrawing Airplane after accepting the addition of Train so at least one of them exists in the configuration.

4. Add Caravan feature and withdraw Hotel feature: the negotiators can propose adding Caravan and withdrawing Hotel as at most only one of them can be selected.

In order to enable us to formulate a negotiation as a two-player negotiation game, three assumptions are made:

1. Information is complete. This assumption implies that each player has full knowledge of the other's preferences. Although games with incomplete information appear more realistic than games with complete information, the proposed approach is an important step in solving incomplete information games.

2. The rules of the game (protocol) are known, as these tell what actions are permitted.

3. The feature model which describes the services is publicly known so the requestor can add and remove features based on the given feature model.

The negotiators can achieve a gain by reaching a satisfactory agreement. The service provider can determine the gain of each feature by calculating the difference between its cost and its price. Therefore, given the price P and the cost Co of a feature f, the valuation function of the provider can be defined as:

$$W_{Pro}(f) = P(f) - Co(f) \tag{3.1}$$

This function returns a real value for a feature f to the service provider. Consequently, the utility function of the provider returns the total value of all features in the configuration C, which can be defined as:

$$U_{Pro}(C) = \sum_{f \in C} W_{Pro}(f) \tag{3.2}$$

Similarly, the service requestor can determine their gain for each feature by calculating the difference between its price and its value. The value of a feature for the service requestor cannot be measured directly but it can be measured indirectly. Certain factors can be used to measure and determine these values. For example, the service requestor may prefer Airplane to Train because the airplane is more

comfortable and the trip duration is shorter, although the train is cheaper in price. Thus, the values that the service requestor obtains from a feature may overcome its price, although the price is high. Therefore, given the price P and the value V of a feature f, the valuation function of the requestor can be defined as:

$$W_{Req}(f) = V(f) - P(f) \tag{3.3}$$

The utility function of the requestor returns the total value of all features in the configuration C, which can be defined as:

$$U_{Req}(C) = \sum_{f \in C} W_{Req}(f) \tag{3.4}$$

In Table 3.1, we show the cost, price and value of each feature in our running example.

Feature	Cost	Price	Value
Transportation- Train	2	4	6
Transportation- Airplane	5	9	12
Accommodation- Caravan	3	5	9
Accommodation- Hotel	5	10	17
Catering	3	8	11

TABLE 3.1: The Cost, Price and Value of Each Feature

To play a game, we assume as given:

- Two players Pro, Req representing the provider and the requestor.
- A feature model FM describing the available services. It consists of a set of features, F.
- A feature configuration C representing the configuration under discussion.

- Two payoff functions $U_{Pro}(C)$, $U_{Req}(C)$. Each negotiator has its own payoff (utility) function, as shown in (3.2) and (3.4). These functions return real values for every configuration C for the provider and the requestor.

In order to implement the feature negotiation games, the feature model, configuration, negotiators and types of proposals must be defined. Moreover, the rules of the games must be designed carefully to define the interaction of the negotiators, which details what and when actions are permitted.

3.3 Graph Transformation Games

As an implementation of our feature negotiation games, we propose graph transformation games which combine both graph transformation and game theoretical concepts. We aim to use graph transformations to model the negotiation games as state-based transformations while game theory is used to analyse the interactions between states.

Using game theory in our negotiation assumes that the negotiators are the players of the game, who have individual and joint interests. These interests are measured by payoff functions.

In graph transformation, there are two main components: graphs and rules that can be applied to these graphs. In the following, we describe the use of these components in the implementation of our feature negotiation games:

Graphs: A graph transformation game is a game whose states are given by graphs. Such a graph consists of:

1. A feature model describing the set of all possible configurations by listing the existing features and their dependencies.

2. The current configuration under discussion consisting of all features agreed thus far.

3. A negotiation state, e.g., proposals to add or remove features, both current and past (in order to avoid repetition of proposals), as well as information on whose turn it is to accept or reject a feature or make a new proposal.

In each graph, we calculate the payoffs (utilities) of both players for the current configuration under discussion. This means that, after accepting the addition of any feature, the value of that feature will be added to the total payoffs of the current configuration. Similarly, when removing a feature from the configuration, the value of that feature will be deducted from the total.

Graph Transformation Rules: In graph transformation games, the players' moves are defined by graph transformation rules. These moves are defined as operations in which the rules describe their pre and post conditions.

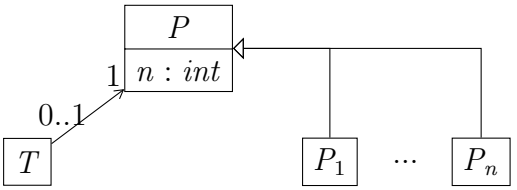
Graph Transformation Game:

The definition of a *graph transformation game* therefore consists of:

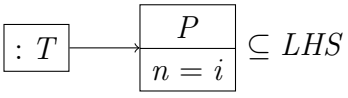
- A type graph TG to define the set of possible states $\mathcal{G}(TG)$.
- A set of players P representing the provider and the requestor.
- A set of rules R where $R(p) \subseteq R$ is the set of rules for player p. The rules describe the available moves of the players in the negotiation.
- A start graph G_0 as initial state.
- For each player p a payoff function $f_p(G) = y$ that defines a real-valued evaluation for each graph $G \in \mathcal{G}(TG)$. The value of each graph represents the value of the configuration under discussion.

As we will see in Section 3.4, our games are turn-based, where players take turns when playing. Each state is under the control of exactly one player. The players who control the states are determined by the rules. The implementation of the turn-based interaction in our graph transformation games is illustrated as follows:

- The type graph TG contains a superclass P with a set of subclasses $\{P_1, ..., P_n\}$ to represent the *players* and a class T to represent the *turn*. The *turn* T can only be associated with exactly one player.



- For each $r \in R(i)$, the *turn* T is associated with the player i in the left-hand side of the rule.



3.4 Implementation

In this section, we introduce the implementation of our graph transformation games. We have developed a metamodel to define the negotiation entities and describe the relationships between them in Section 3.4.1. Using this metamodel, the negotiation of features can be precisely specified as graph transformation rules, as described in Section 3.4.2.

3.4.1 Game Metamodel

In Figure 3.2, we present a metamodel to define the negotiation entities and describe the relationships between them. It will also be used as a type graph to implement our transformation rules. It consists of three representations. The first representation describes the feature model including the types of features and the relationships between them. The second representation describes the configuration and its selected features. The third representation describes the negotiation states including the negotiators and the types of proposals. A **Container** class contains all other classes and it will be used to create the dynamic instance graph. A **Count** is a class that is used to indicate the time of each proposal. A **Start** class is used to specify who starts the negotiation. A **Make** class indicates the time of making and responding to proposals. A **Move** class is used to design taking turns and a **Pass** class is used to allow passing turns. In the following, we describe each representation with its concerned classes.

The Feature Model Representation: a **FeatureModel** class represents the feature model that is used to describe the services. It has exactly one **Root** feature and a set of **SubFeature(s)**. **Root** and **SubFeature(s)** are specialisations of a **Feature** class, so they inherit its functionality. A **Feature** has a **name** attribute of type String to assign a name to every feature and an **order** attribute of type Integer in case the features have to be proposed in a specific order. The **SubFeature** has four subclasses, **Mandatory**, **Alternative**, **Optional** and **Or** classes. These are the types of feature in every feature model. Every **SubFeature** has two attributes: **Rpayoff** and **Ppayoff**, of type Integer. **Rpayoff** represents the value of the feature to the requestor and **Ppayoff** represents the value of the feature to the provider. Each **SubFeature** must have exactly one parent of type **Feature** and each **Feature**

has zero or more children of type **SubFeature**. Also, the **SubFeature** can include or exclude any other **Subfeature**.

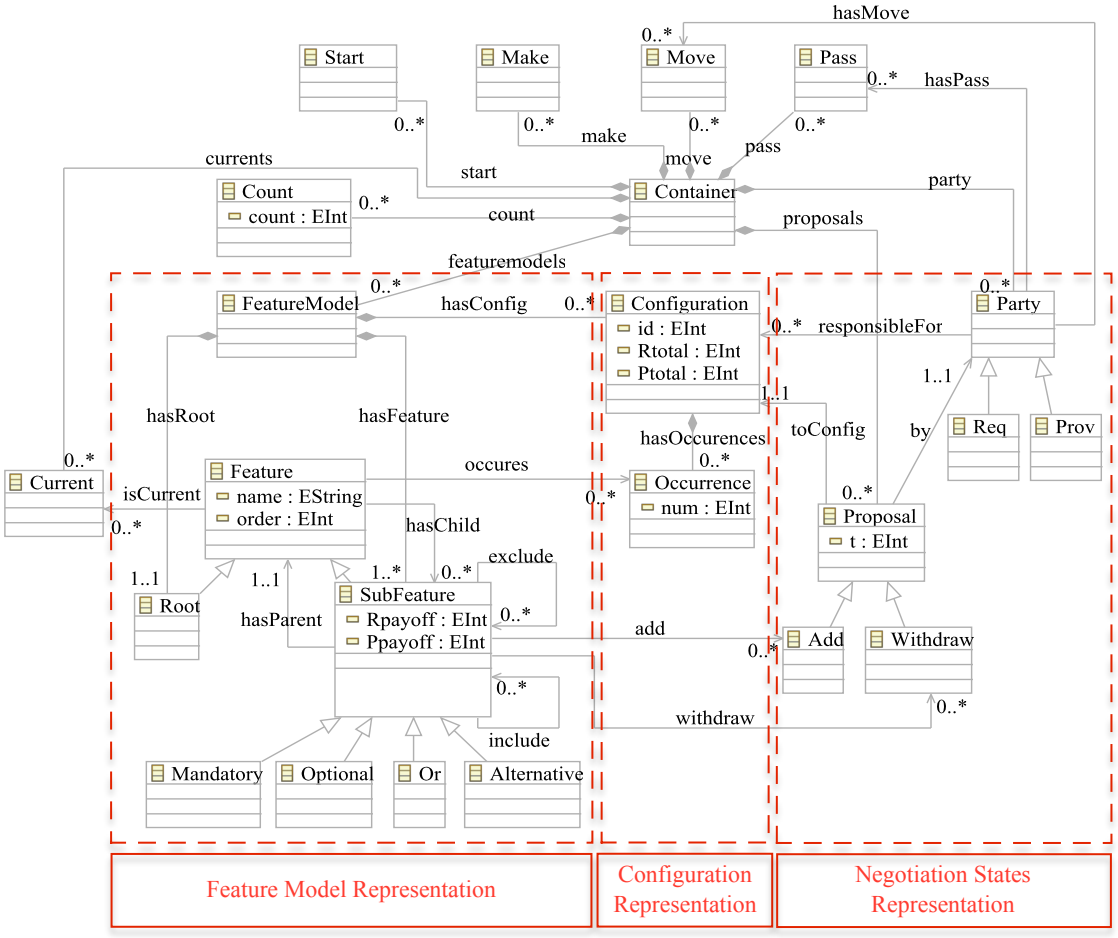


FIGURE 3.2: Metamodel for Negotiation

The Configuration Representation: the **FeatureModel** has a set of **Configuration(s)**. Each **Configuration** has a set of **Occurrence(s)** of **Feature(s)**. This means that these features have been selected in the configuration. The **Configuration** has three attributes: a unique **ID** of type Integer, **Rtotal** of type Integer to represent the value of all selected features in the configuration to the requestor,

and **Ptotal** of type Integer to represent the value of all selected features in the configuration to the provider. These values are calculated according to utility functions (3.2) and (3.4).

The Negotiation States Representation: a **Prov** (provider) and a **Req** (requestor) are subclasses of the **Party** class that represents the negotiators. The **Party** is linked to the Configuration to represent who is responsible for making changes to the resulting configuration. The **Party** can make any number of **Proposal(s)**. The **Proposal** has two subclasses, **Add** and **Withdraw**. The **Party** can propose to **Add** or **Withdraw** any type of features specified by the rules. The **Proposal** is linked to the **Configuration** to keep track of what has been proposed.

3.4.2 Game Rules

Based on the metamodel in Figure 3.2, we can define the rules that govern the players' moves as graph transformation rules. The design of an appropriate negotiation protocol that governs the interactions between the negotiators is one of the key problems that must be considered. For that reason, we outline the properties which have been taken into account while designing our negotiation protocol as follows:

1. **Flexibility:** this property is concerned with the level of flexibility in terms of what is allowed/not allowed to the negotiators at each state of the negotiation.

2. **Simplicity:** a protocol is simple if it is not complicated in its specifications and implementations.

3. **Applicability with feature modelling:** our protocol is designed to define the rules that can be applied to the feature modelling context. For example,

the protocol should define the rules of adding and withdrawing features to the feature model configuration.

4. **Equality:** this property means that both negotiators have the same rights and equal access to all the negotiation aspects. Also, no negotiator has higher power than others to force them to accept their proposals or terminate the negotiation.

Before defining our negotiation protocol, the following are the actions for each player based on the types of feature:

1. Actions for making proposals: the negotiators have five actions to propose adding and withdrawing features from the feature Configuration depending on the types of feature, as follows:
   (a) Propose adding Optional feature: the negotiator proposes adding a new Optional feature to the Configuration.
   (b) Propose adding Or feature: the negotiator proposes adding a new Or feature to the Configuration.
   (c) Propose withdrawing Optional feature: the negotiator proposes withdrawing an existing Optional feature from the Configuration.
   (d) Propose withdrawing Or feature: the negotiator proposes withdrawing an existing Or feature from the Configuration.
   (e) Propose substituting Alternative feature: the negotiator proposes substituting an existing Alternative feature in the Configuration with another Alternative feature.
2. Actions for responding to proposals: the negotiators have 10 actions to respond to the proposals by either accepting and rejecting, as follows:

(a) Accept adding Optional feature: the negotiator accepts the other negotiator's proposal to add a new Optional feature.

(b) Reject adding Optional feature: the negotiator rejects the other negotiator's proposal to add a new Optional feature.

(c) Accept adding Or feature: the negotiator accepts the other negotiator's proposal to add a new Or feature.

(d) Reject adding Or feature: the negotiator rejects the other negotiator's proposal to add a new Or feature.

(e) Accept withdrawing Optional feature: the negotiator accepts the other negotiator's proposal to withdraw an existing Optional feature.

(f) Reject withdrawing Optional feature: the negotiator rejects the other negotiator's proposal to withdraw an existing Optional feature.

(g) Accept withdrawing Or feature: the negotiator accepts the other negotiator's proposal to withdraw an existing Or feature.

(h) Reject withdrawing Or feature: the negotiator rejects the other negotiator's proposal to withdraw an existing Or feature.

(i) Accept substituting Alternative feature: the negotiator accepts the other negotiator's proposal to substitute an Alternative feature.

(j) Reject substituting Alternative feature: the negotiator rejects the other negotiator's proposal to substitute an Alternative feature.

3.4.2.1 Alternating-offer Negotiation Protocol

In this section, we introduce our negotiation protocol as an Alternating-offer Negotiation Protocol in which the negotiators interact by taking turns in making proposals. Alternating-offer protocols are widely used in negotiation to provide certain goals

and objectives. In our negotiation, the rules in the Alternating-offer protocol are used to guide the negotiators in proposing and responding to proposals at specific times.

The negotiators have the following possible actions:

1. Propose: the negotiators make proposals.
2. Respond: the negotiators accept or reject the proposals.
3. Pass turn: a negotiator passes its turn to the other negotiator if it does not want to make a proposal at a certain time.

The design of the Alternating-offer Negotiation protocol depends on the idea that the negotiation takes place over a sequence of rounds. Figure 3.3 describes the Alternating-offer Protocol using a state-chart diagram. It shows how the negotiators interact by taking turns to construct the negotiation game in a tree structure. The protocol refers to the negotiators as **Negotiator 1** and **Negotiator 2**. As discussed before, the assumption is that we have two negotiators, the service provider and the service requestor (Negotiator 1 and Negotiator 2), who negotiate over a feature model configuration by exchanging proposals for adding, withdrawing and substituting features.

Given that domain, the Alternating-offer Negotiation Protocol is described in Figure 3.3 as follows: negotiation takes place by the negotiators taking turns, in which each negotiator proposes and responds to proposals at a specific time.

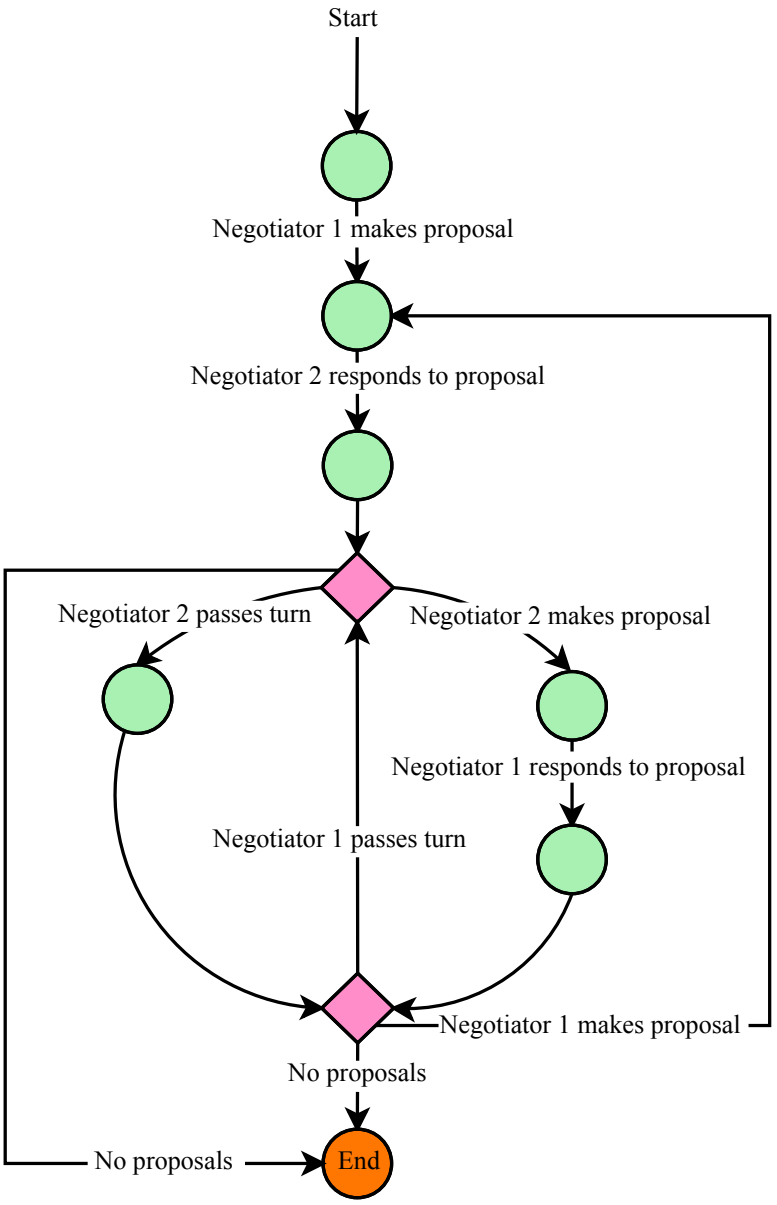


FIGURE 3.3: Alternating-offer Negotiation Protocol State-chart Diagram

Negotiator 1 begins at round 0 by **making a proposal**, to which **Negotiator 2** can **respond** by either accepting or rejecting it. After responding to the proposal, **Negotiator 2** can either **make a new proposal** or **pass turn** to **Negotiator 1**. If **Negotiator 2 makes a new proposal**, **Negotiator 1** can **respond** to it by either accepting or rejecting it. Then, **Negotiator 1** can either **make a new proposal** or **pass turn** to **Negotiator 2**. If **Negotiator 2 passes turn** to **Negotiator**

1, again the same process is followed by allowing **Negotiator 1** to **make a new proposal** or **pass turn** to **Negotiator 2**. This process continues until there are no proposals left to be made or if neither negotiator wants to make a new proposal, and this can be achieved if both pass turns one after the other.

3.4.2.2 Application to Running Example

The initial configuration in our running example contains: Airplane for Transportation and Hotel for Accommodation. Figure 3.4 shows part of the negotiation game tree, which is constructed based on the Alternating-offer Negotiation Protocol. It provides an example of a sequence of rounds in which the negotiators take turns in exchanging the proposals. In this example, it assumes that the requestor starts the negotiation by making one of three available proposals at round 0. After that, the provider can accept or reject the proposal and then make a new proposal or pass turn and so on. In this figure, blue nodes represent the requestor's decision nodes, while red nodes represent the provider's decision nodes. The terminal nodes of the tree can be reached if there are no available proposals left or two pass actions have been made one after the other.

3.4.2.3 The Graph Transformation Games in Henshin

In order to implement the available actions for the negotiators according to the given interactions in Figure 3.3 in which the parties take turns, we divided our rules into three categories. First, rules for starting the negotiation. Second, rules for describing the moves of the requestor. Third, rules for describing the moves of the provider. The names of the requestor rules start with "Req" prefix and the names of the provider rules start with "Prov" prefix. We created two versions of our negotiation rules. The first version allows the negotiators to make any available

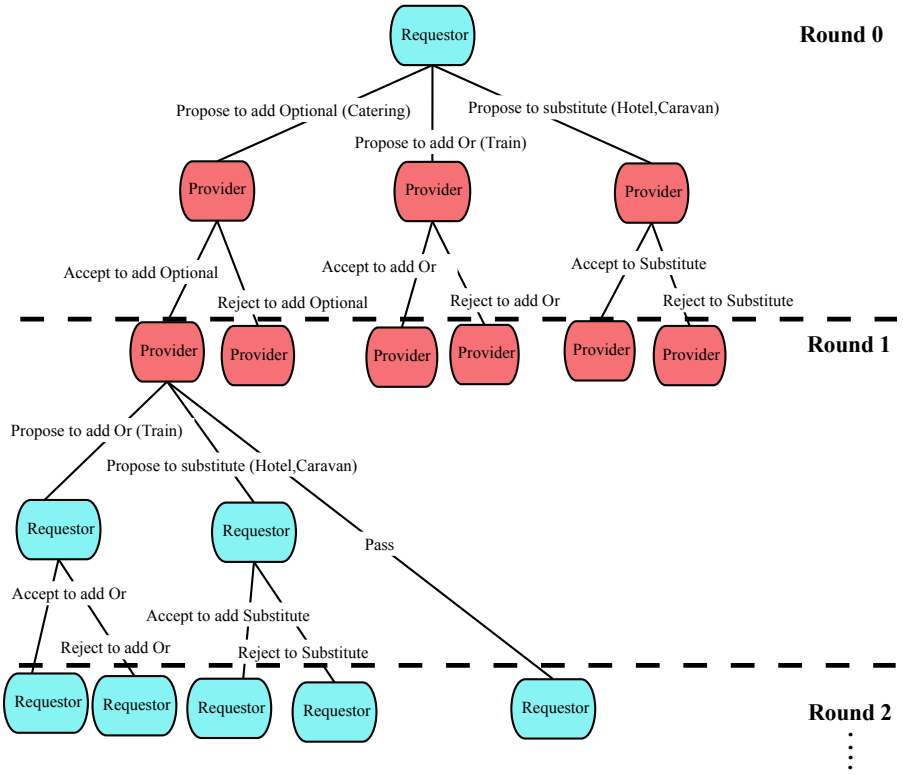


FIGURE 3.4: Part of the Negotiation Game Tree

proposal at each proposing time. The second version explores the feature model in a specific order in which only one feature can be proposed at each proposing time. In this section, we firstly provide a detailed description of the first version of our negotiation rules. Then, we present an example rule of the second version. In all rules, the explanation of Count, Start, Make, Move and Pass instances are omitted as they are only used to design the behaviour of the negotiators but not the actual negotiation of features.

1. Five rules for making proposals:

   (a) **propose_to_addOpt:** the negotiator proposes adding a new Optional feature to the Configuration.

In Figure 3.5, the *Req_propose_to_addOpt* rule is shown. It has four parameters, **featureName**, **time**, **Pt**, and **Rt**. According to its preconditions, there should be an **Optional** feature, a **Req**, a **Configuration**, a parent **Feature** of the Optional feature that has an **Occurrence** in the Configuration, and a **Container**. The parent Feature is needed to maintain the consistency of the Configuration because the child features cannot be added without their parents. The Optional feature has an attribute **name** which takes the value of the parameter featureName. The Configuration has two attributes: **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.
This rule has four negative application conditions. The Optional feature has not been added before. The Optional feature has not been withdrawn before. The Optional feature is not currently occurring in the Configuration. The Optional feature is not excluded by a feature that is occurring in the Configuration. The negative application conditions are expressed by **forbid** actions.
In this case, the rule *Req_propose_to_addOpt* can be applied with the result of preserving objects expressed by **preserve** actions and creating one **Add** proposal instance in the current time **(time+1)**. The **Add** object has a link to the Optional feature that is proposed, a link to the Configuration and a link the Req. These are expressed by **create** actions.

(b) **propose_to_addOr:** the negotiator proposes adding a new Or feature to the Configuration. In Figure 3.6, the *Req_propose_to_addOr* rule is shown. It has four parameters, **featureName**, **time**, **Pt**, and **Rt**. According to its preconditions, there should be an **Or** feature, a **Req**, a **Configuration**, a parent **Feature** of the Or feature that has an **Occurrence** in the Configuration, a sibling **Or** feature that also has an **Occurrence** in

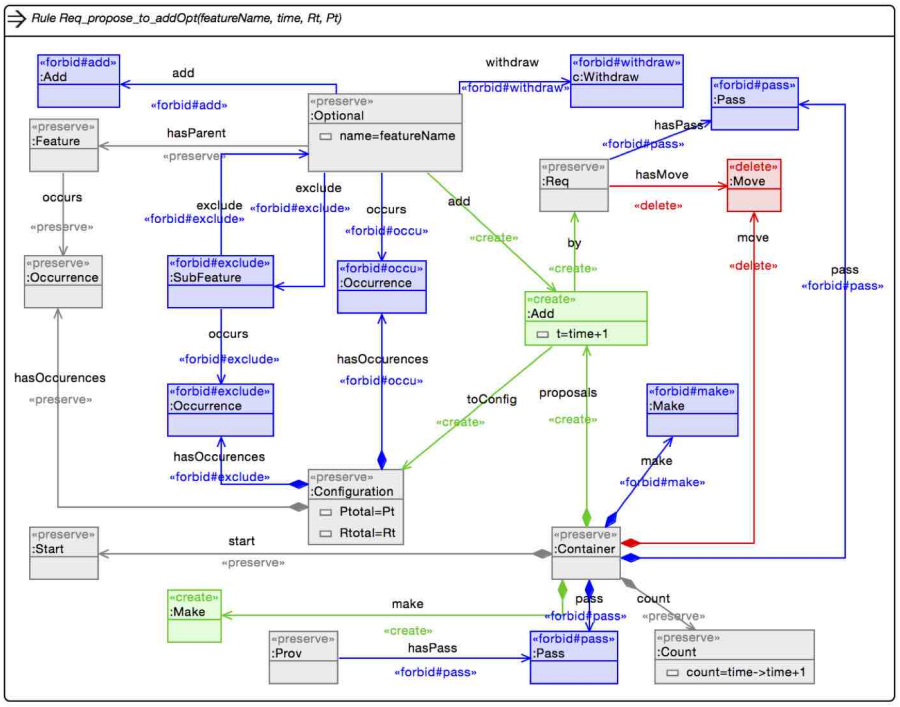


FIGURE 3.5: Transformation Rule *Req_propose_to_addOpt*

the Configuration, and a **Container**. The parent Feature is needed to maintain the consistency of the Configuration. The sibling Or feature is used to show that the existence of the parent requires the existence of at least one Or child feature. The Or feature has an attribute **name** which takes the value of the parameter featureName. The Configuration has two attributes: **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has four negative application conditions. The Or feature has not been added before. The Or feature has not been withdrawn before. The Or feature is not currently occurring in the Configuration. The Or feature is not excluded by a feature that is occurring in the Configuration. The negative application conditions are expressed by **forbid** actions.

After applying the *Req_propose_to_addOr* rule, objects expressed by **preserve** actions are preserved and one **Add** proposal instance is created in

the current time **(time+1)**. The **Add** object has a link to the Or feature that is proposed, a link to the Configuration and a link to the Req. These are expressed by **create** actions.

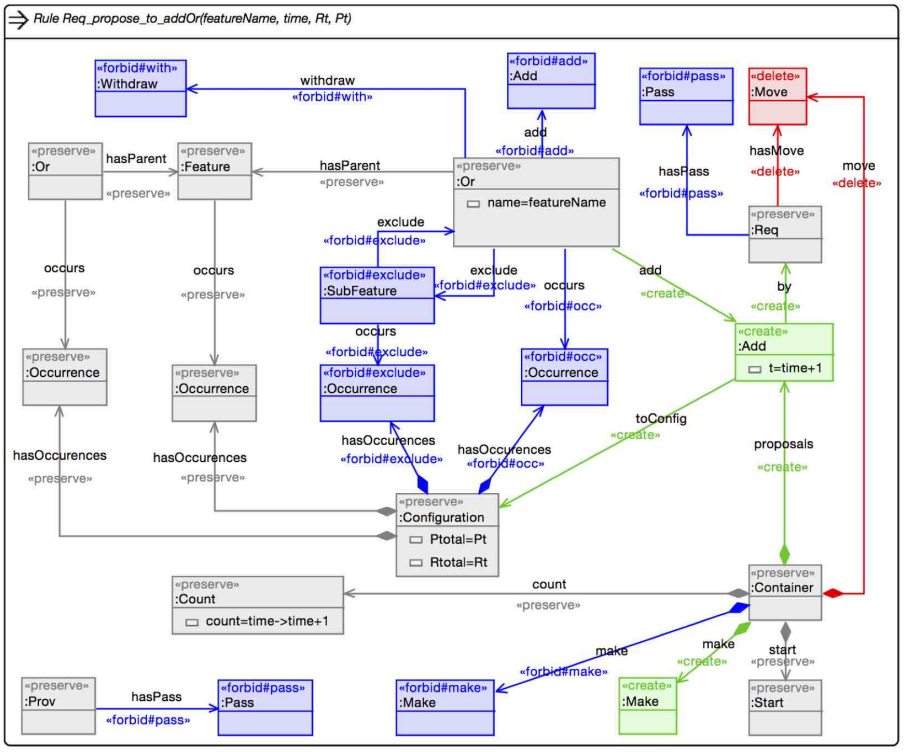


FIGURE 3.6: Transformation Rule *Req_propose_to_addOr*

(c) **propose_to_withdrawOpt:** the negotiator proposes withdrawing an existing Optional feature from the Configuration.

In Figure 3.7, the *Req_propose_to_withdrawOpt* rule is shown. It has four parameters, **featureName**, **time**, **Pt**, and **Rt**. According to its preconditions, there should be an **Optional** feature that has an **Occurrence** in the Configuration, a **Req**, a **Configuration**, and a **Container**. The **Optional** feature has an attribute **name** which takes the value of the parameter featureName. The Configuration has two attributes: **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has three negative application conditions. The Optional feature has not been added before. The Optional feature has not been withdrawn before. The Optional feature is not included by a feature that is occurring in the Configuration, because withdrawing this feature results in an inconsistency of the Configuration. The negative application conditions are expressed by **forbid** actions.

In this case, the rule *Req_propose_to_withdrawOpt* can be applied with the result of preserving objects expressed by **preserve** actions and creating one **Withdraw** proposal instance in the current time **(time+1)**. The **Withdraw** object has a link to the Optional feature that is proposed, a link to the Configuration and a link the Req. These are expressed by **create** actions.

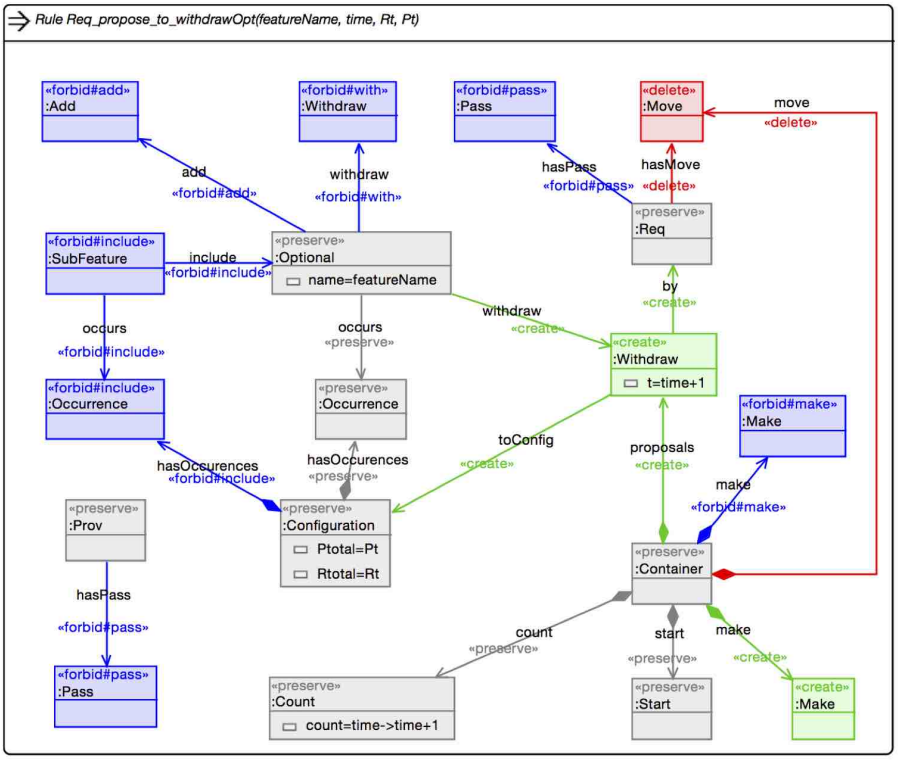


FIGURE 3.7: Transformation Rule *Req_propose_to_withdrawOpt*

(d) **propose_to_withdrawOr:** the negotiator proposes withdrawing an existing Or feature from the Configuration.

In Figure 3.8, the *Req_propose_to_withdrawOr* rule is shown. It has four parameters, **featureName**, **time**, **Pt**, and **Rt**. According to its preconditions, there should be an **Or** feature that has an **Occurrence** in the Configuration, a **Req**, a **Configuration**, a parent **Feature** of the Or feature that has an **Occurrence** in the Configuration, a sibling **Or** feature that also has an **Occurrence** in the Configuration, and a **Container**. The parent Feature and the Or sibling feature are needed to maintain the consistency of the Configuration because the parent of the Or group requires the existence of at least one Or child feature. The Or feature has an attribute **name** which takes the value of the parameter featureName. The Configuration has two attributes: **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has three negative application conditions. The Or feature has not been added before. The Or feature has not been withdrawn before. The Or feature is not included by a feature that is occurring in the Configuration because withdrawing this feature results in an inconsistency of the Configuration. The negative application conditions are expressed by **forbid** actions.

After applying the *Req_propose_to_withdrawOr* rule, objects expressed by **preserve** actions are preserved and one **Withdraw** proposal instance is created in the current time **(time+1)**. The **Withdraw** object has a link to the Or feature that is proposed, a link to the Configuration and a link to the Req. These are expressed by **create** actions.

(e) **propose_to_substitute:** the negotiator proposes substituting an existing Alternative feature in the Configuration with another Alternative

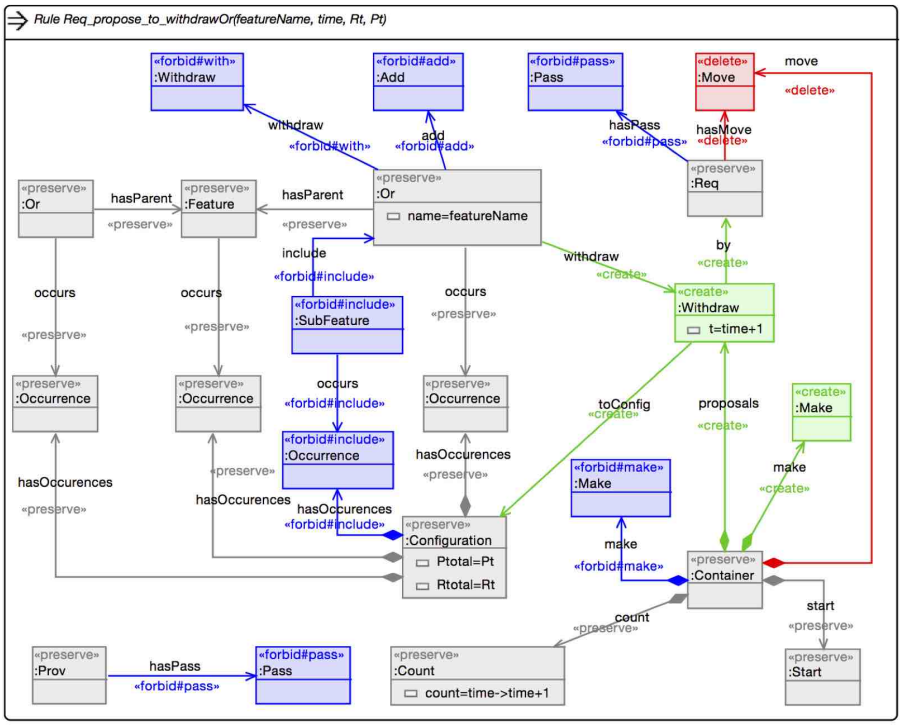


FIGURE 3.8: Transformation Rule *Req_propose_to_withdrawOr*

feature.

In Figure 3.9, *the Req_propose_to_substitute* rule is shown. It has five parameters, **featureName**, **featureName1**, **time**, **Pt**, and **Rt**. According to its preconditions, there should be an **Alternative** feature (the feature to be withdrawn) that has an **Occurrence** in the Configuration, a **Req**, a **Configuration**, a parent **Feature** of the Alternative feature that has an **Occurrence** in the Configuration, a sibling **Alternative** feature (the feature to be added) and a **Container**. The parent Feature and the Alternative sibling feature are needed to maintain the consistency of the Configuration because the parent of the Alternative group requires exactly one Alternative child feature. The Alternative feature has an attribute **name** which takes the value of the parameter featureName. The sibling Alternative feature has an attribute **name** which takes the value of the parameter featureName1. The Configuration has two attributes:

Rtotal, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has five negative application conditions. The Alternative feature has not been added or withdrawn before. The sibling Alternative feature has not been added or withdrawn before. The Alternative feature is not included by a feature that is occurring in the Configuration because withdrawing this feature results in an inconsistency of the Configuration. The sibling Alternative feature is not excluded by a feature that is occurring in the Configuration. The sibling Alternative feature is not currently occurring in the Configuration. The negative application conditions are expressed by **forbid** actions.

After applying *Req_propose_to_substitute* rule, objects expressed by **preserve** actions are preserved and two proposal instances, **Add** and **Withdraw**, are created in the current time **(time+1)**. The **Withdraw** object has a link to the Alternative feature, a link to the Configuration and a link to the Req. Similarly, the **Add** object has a link to the sibling Alternative feature, a link to the Configuration and a link to the Req. These are expressed by **create** actions.

2. Ten rules for responding to proposals:

   (a) **accept_to_addOpt:** the negotiator accepts the other negotiator's proposal to add a new Optional feature.

   In Figure 3.10, the *Prov_accept_to_addOpt* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Optional** feature, a **Prov**, a **Configuration**, an **Add** proposal object that is linked to the Optional feature and the Configuration, and a **Container**. The Optional feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**,

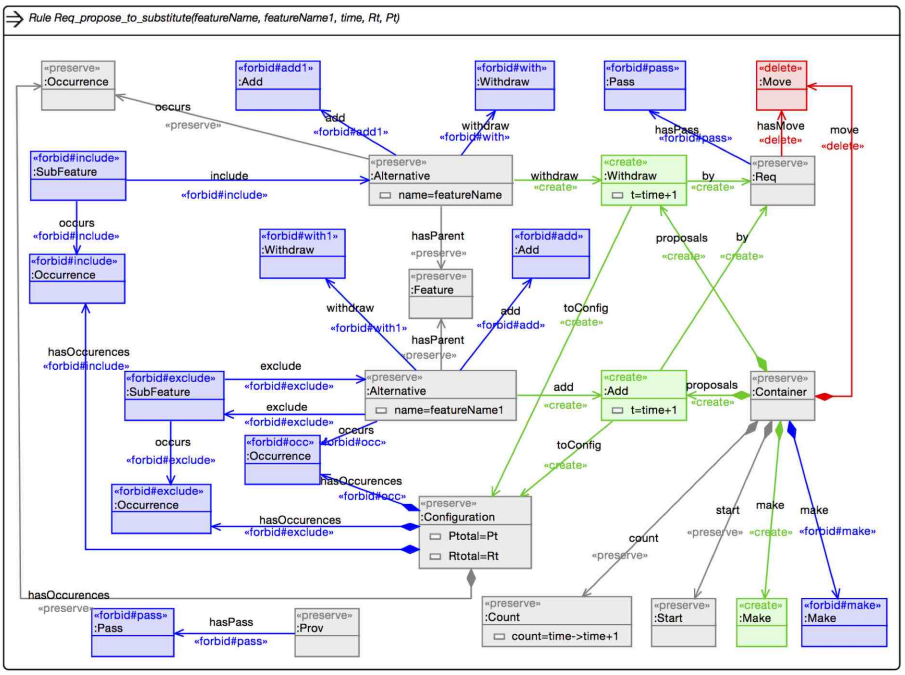


FIGURE 3.9: Transformation Rule *Req_propose_to_substitute*

which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has three negative application conditions. The Optional feature has not been withdrawn before. The Optional feature is not currently occurring in the Configuration. The proposal Add has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions. After applying the *Prov_accept_to_addOpt* rule, objects expressed by **preserve** actions are preserved and a new **Occurrence** object is created and linked to both the Optional feature and the Configuration. Also, the value of the attribute Rtotal in the Configuration will be changed from Rt to Rt+R and the value of the attribute Ptotal in the Configuration will be changed from Pt to Pt+P. It means that we add

the value of the feature to the total of the requestor and provider.

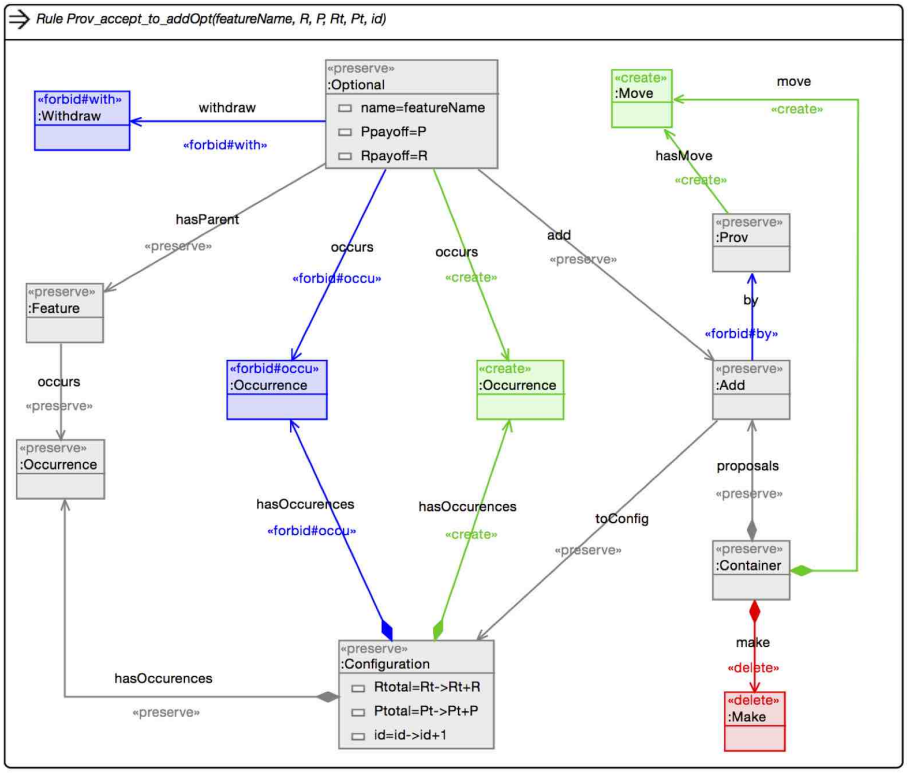


FIGURE 3.10: Transformation Rule *Prov_accept_to_addOpt*

(b) **reject_to_addOpt:** the negotiator rejects the other negotiator's proposal to add a new Optional feature.

In Figure 3.11, the *Prov_reject_to_addOpt* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Optional** feature, a **Prov**, a **Configuration**, an **Add** proposal object that is linked to the Optional feature and the Configuration, and a **Container**. The Optional feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter

Pt.

This rule has three negative application conditions. The Optional feature has not been withdrawn before. The Optional feature is not currently occurring in the Configuration. The proposal Add has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_reject_to_addOpt* rule, objects expressed by preserve actions are preserved and the link **toConfig** between the Add object and the Configuration is deleted. The deletion is expressed by **delete** actions.

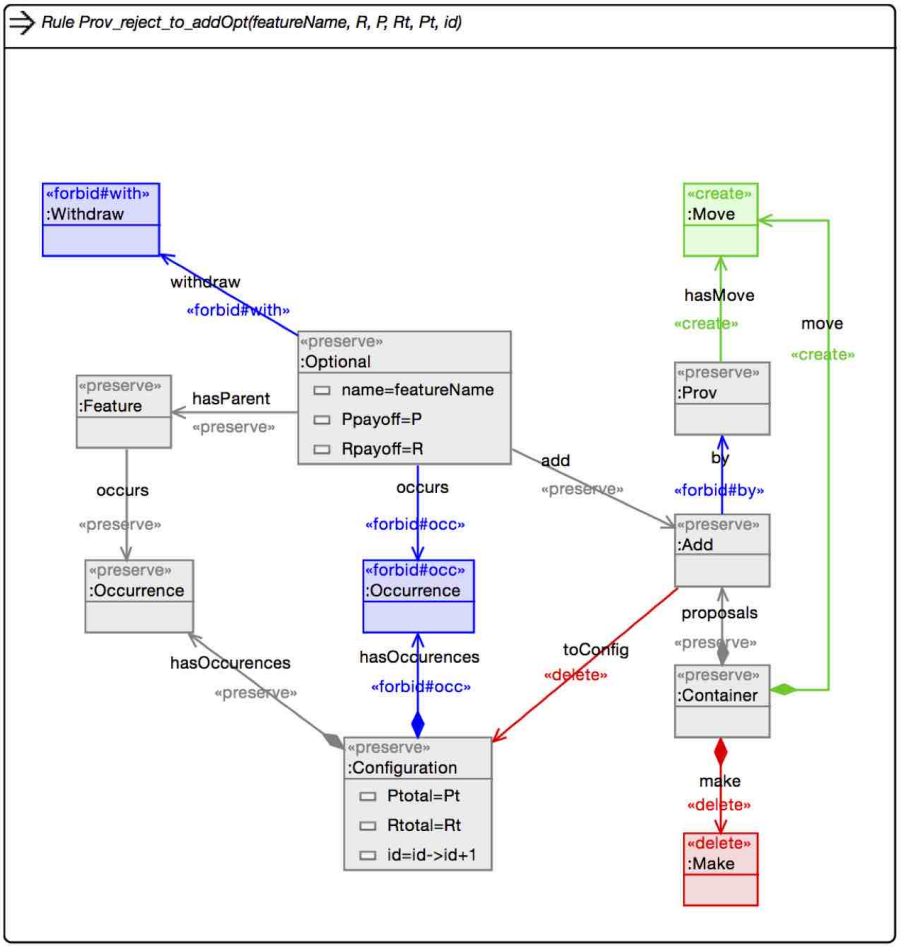


FIGURE 3.11: Transformation Rule *Prov_reject_to_addOpt*

(c) **accept_to_addOr:** the negotiator accepts the other negotiator's proposal to add a new Or feature.

In Figure 3.12, the *Prov_accept_to_addOr* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Or** feature, a parent **Feature** of the Or feature that has an occurrence in the Configuration, a **Prov**, a **Configuration**, an **Add** proposal object that is linked to the Or feature and the Configuration, and a **Container**. The Or feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has three negative application conditions. The Or feature has not been withdrawn before. The Or feature is not currently occurring in the Configuration. The proposal Add has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_accept_to_addOr* rule, objects expressed by **preserve** actions are preserved and an **Occurrence** object is created and linked to both the Or feature and the Configuration. Also, the value of the attribute Rtotal in the Configuration will be changed from Rt to Rt+R and the value of the attribute Ptotal in the Configuration will be changed from Pt to Pt+P. It means that we add the value of the feature to the total of the requestor and the provider.

(d) **reject_to_addOr:** the negotiator rejects the other negotiator's proposal to add a new Or feature.

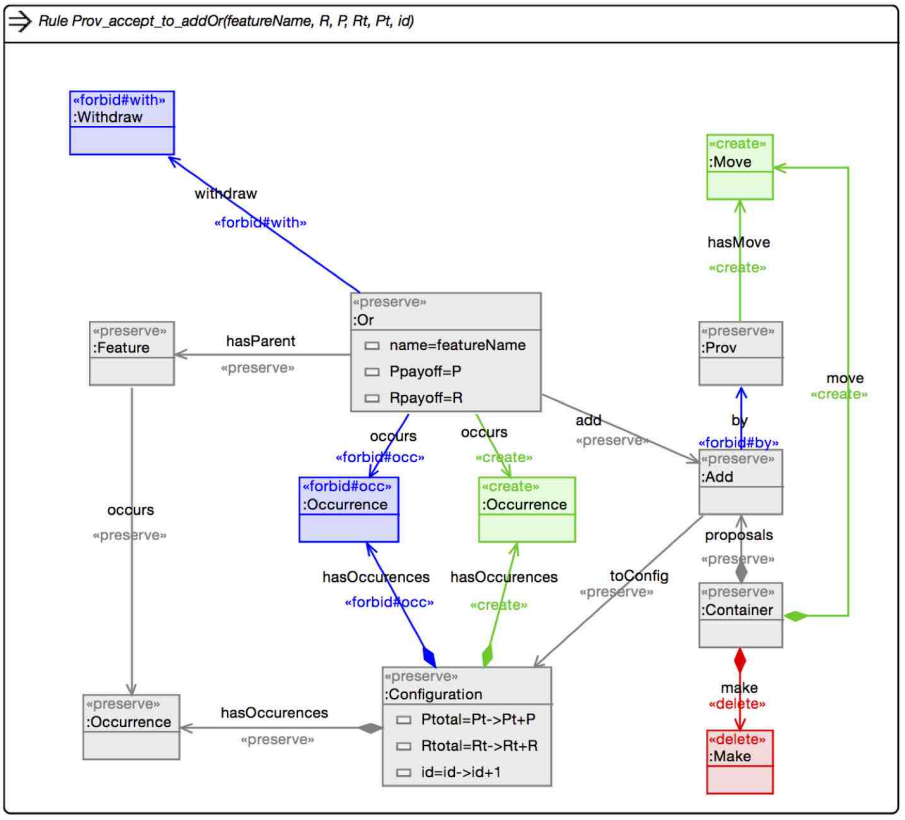


FIGURE 3.12: Transformation Rule *Prov_accept_to_addOr*

In Figure 3.13, the *Prov_reject_to_addOr* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Or** feature, a parent **Feature** of the Or feature that has an occurrence in the Configuration, a **Prov**, a **Configuration**, an **Add** proposal object that is linked to the Or feature and the Configuration, and a **Container**. The Or feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has three negative application conditions. The Or feature has

not been withdrawn before. The Or feature is not currently occurring in the Configuration. The proposal Add has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_reject_to_addOr* rule, objects expressed by **preserve** actions are preserved and the link **toConfig** between the Add object and the Configuration is deleted. The deletion is expressed by **delete** actions.

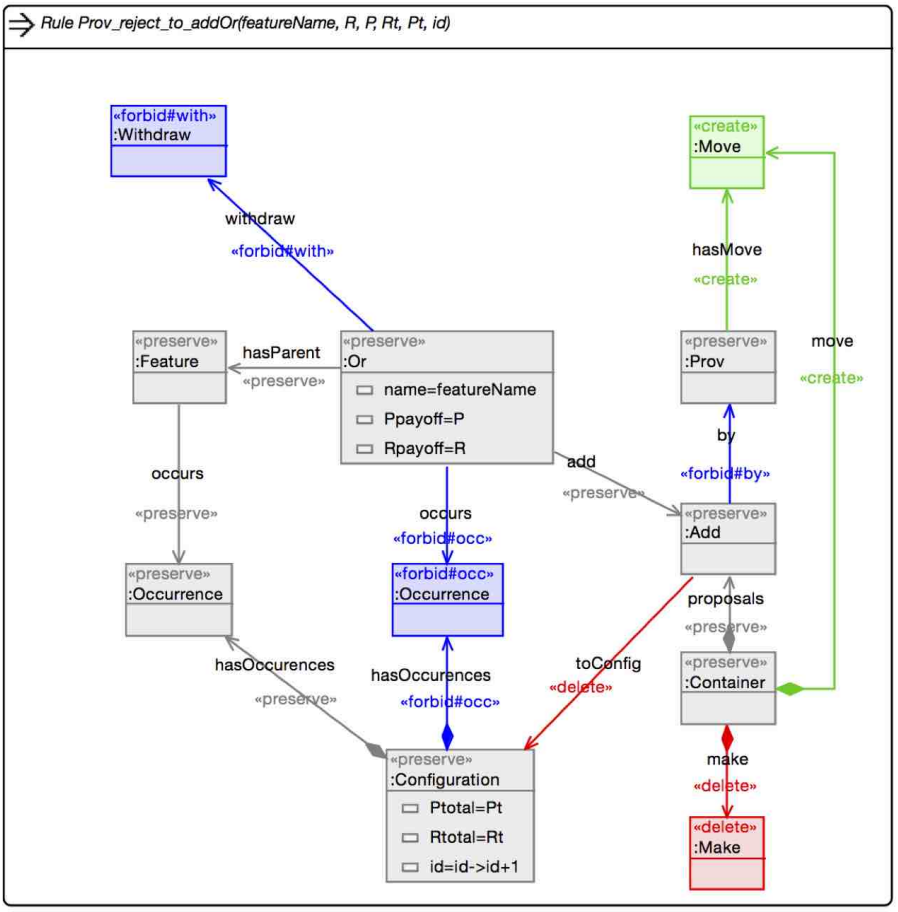


FIGURE 3.13: Transformation Rule *Prov_reject_to_addOr*

(e) **accept_to_withdrawOpt:** the negotiator accepts the other negotiator's proposal to withdraw an existing Optional feature.

In Figure 3.14, the *Prov_accept_to_withdrawOpt* rule is shown. It has six

parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Optional** feature that has an **Occurrence** in the Configuration, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Optional feature and the Configuration, and a **Container**. The Optional feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has two negative application conditions. The Optional feature has not been added before. The proposal Withdraw has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_accept_to_withdrawOpt* rule, objects expressed by **preserve** actions are preserved and the **Occurrence** object with its links are deleted. Also, the value of the attribute Rtotal in the Configuration will be changed from Rt to Rt-R and the value of the attribute Ptotal in the Configuration will be changed from Pt to Pt-P. It means that we subtract the value of the feature from the total of the requestor and the provider.

(f) **reject_to_withdrawOpt:** the negotiator rejects the other negotiator's proposal to withdraw an existing Optional feature.

In Figure 3.15, the *Prov_reject_to_withdrawOpt* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Optional** feature that has an **Occurrence** in the Configuration, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Optional feature and the Configuration, and

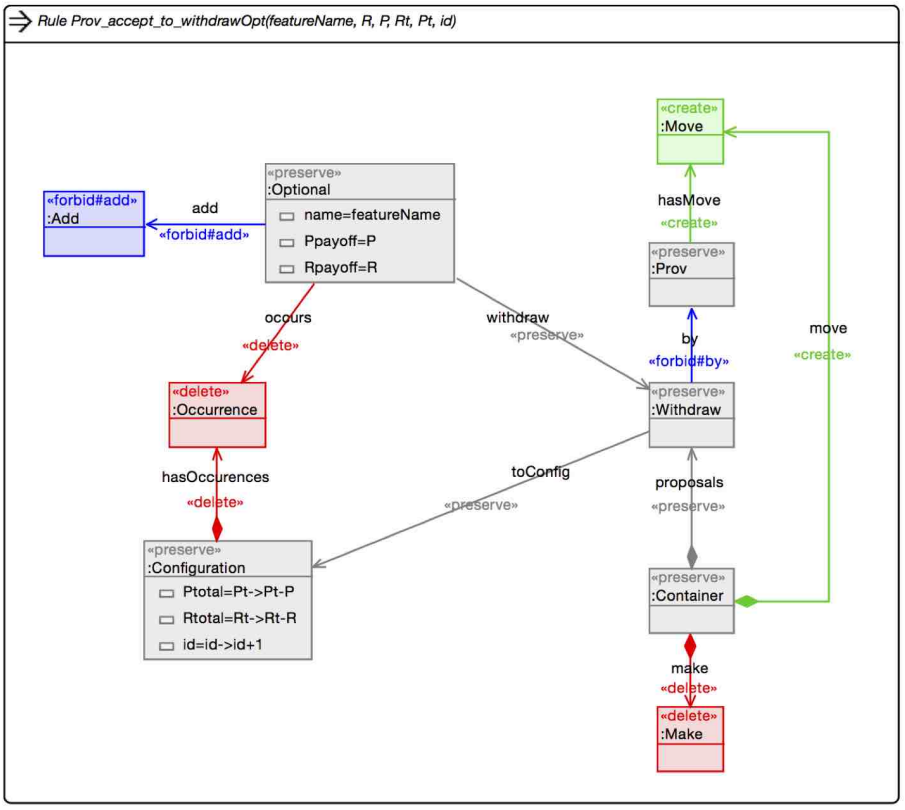


FIGURE 3.14: Transformation Rule *Prov_accept_to_withdrawOpt*

a **Container**. The Optional feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has two negative application conditions. The Optional feature has not been added before. The proposal Withdraw has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_reject_to_withdrawOpt* rule, objects expressed by **preserve** actions are preserved and the link **toConfig** between the Withdraw object and the Configuration is deleted. The deletion is expressed

by **delete** actions.

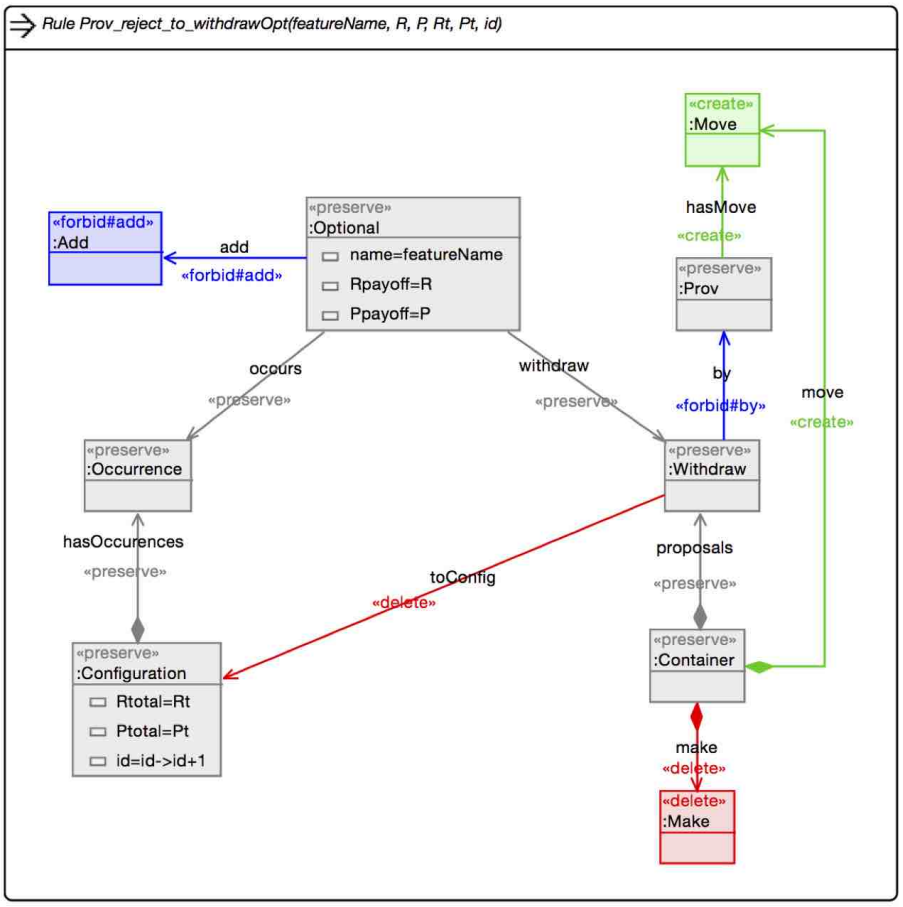


FIGURE 3.15: Transformation Rule *Prov_reject_to_withdrawOpt*

(g) **accept_to_withdrawOr:** the negotiator accepts the other negotiator's proposal to withdraw an existing Or feature. In Figure 3.16, the *Prov_acce pt_to_withdrawOr* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Or** feature that has an **Occurrence** in the Configuration, a parent **Feature** with another child **Or** feature who have occurrences in the Configuration, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Or feature and the Configuration, and a **Container**. The Or feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter

R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has two negative application conditions. The Or feature has not been added before. The proposal Add has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_accept_to_withdrawOr* rule, objects expressed by **preserve** actions are preserved and the **Occurrence** object with its links are deleted. Also, the value of the attribute Rtotal in the Configuration will be changed from Rt to Rt-R and the value of the attribute Ptotal in the Configuration will be changed from Pt to Pt-P. It means that we subtract the value of the feature from the total of the requestor and the provider.

(h) **reject_to_withdrawOr:** the negotiator rejects the other negotiator's proposal to withdraw an existing Or feature.

In Figure 3.17, the *Prov_reject_to_withdrawOr* rule is shown. It has six parameters, **featureName**, **R**, **P**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Or** feature that has an **Occurrence** in the Configuration, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Or feature and the Configuration, and a **Container**. The Or feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

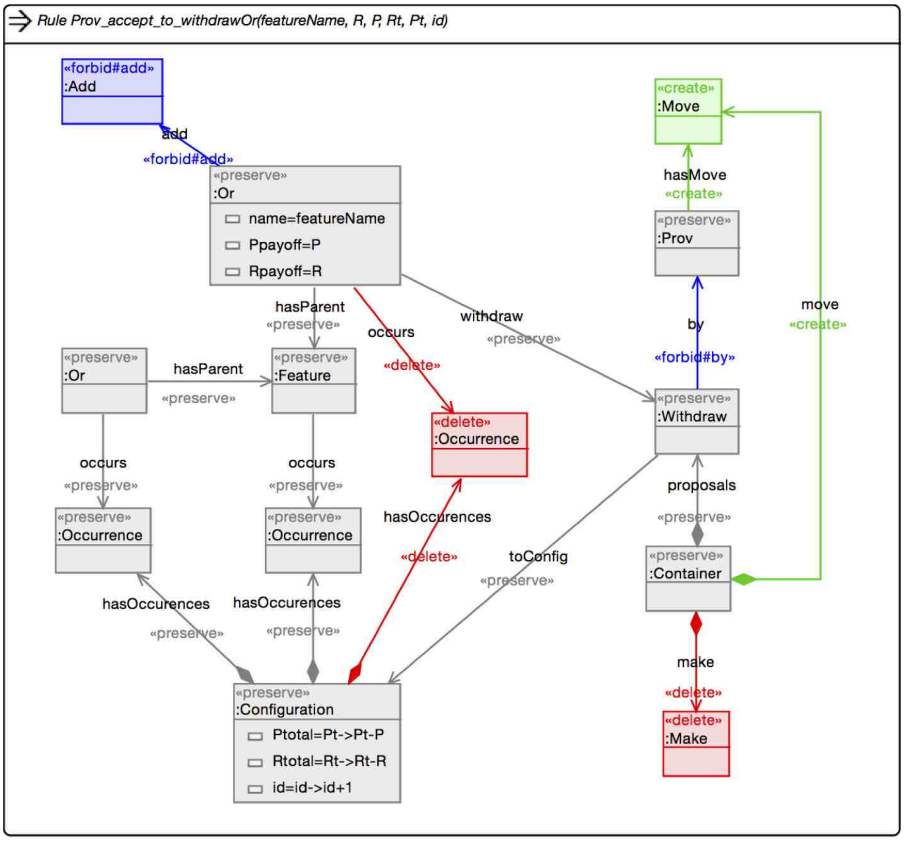


FIGURE 3.16: Transformation Rule *Prov_accept_to_withdrawOr*

This rule has two negative application conditions. The Or feature has not been added before. The proposal Withdraw has not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions. After applying the *Prov_reject_to_withdrawOr* rule, objects expressed by **preserve** actions are preserved and the link **toConfig** between the Withdraw object and the Configuration is deleted. The deletion is expressed by **delete** actions.

(i) **accept_to_substitute:** the negotiator accepts the other negotiator's proposal to substitute an Alternative feature.

In Figure 3.18, the *Prov_accept_to_substitute* rule is shown. It has nine parameters, **featureName**, **featureName1**, **R**, **R1**, **P**, **P1**, **Rt**, **Pt** and

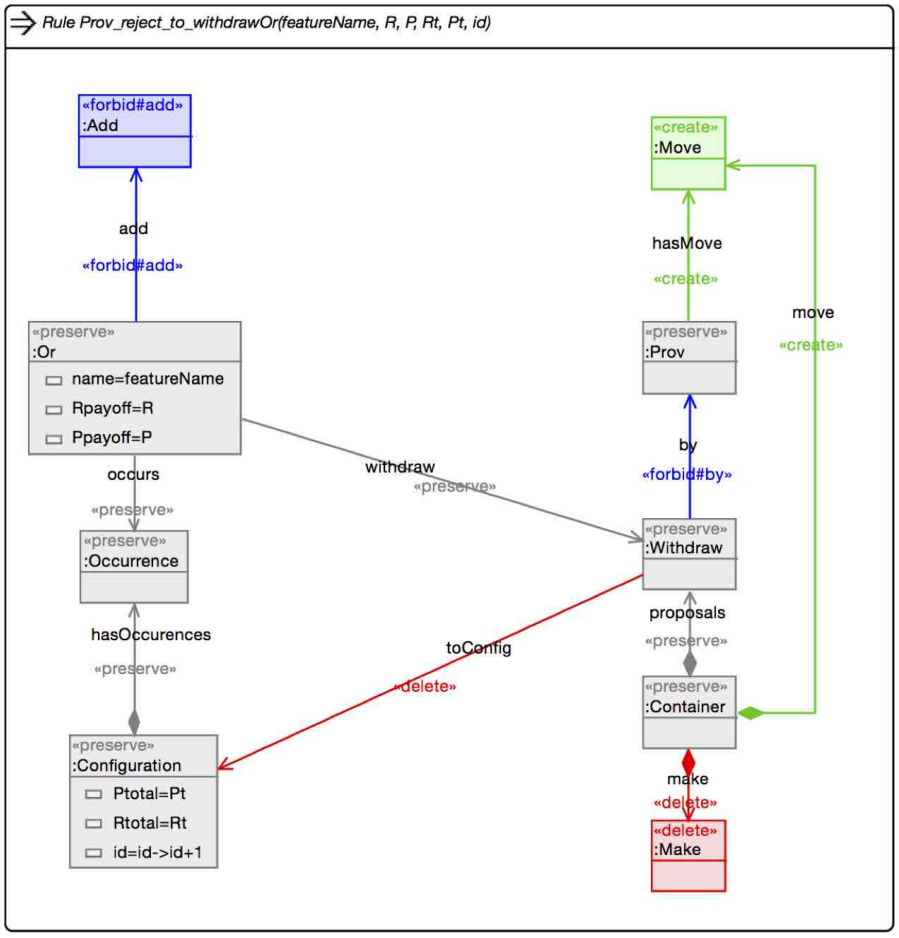


FIGURE 3.17: Transformation Rule *Prov_reject_to_withdrawOr*

id. According to its preconditions, there should be an **Alternative** feature that has an **Occurrence** in the Configuration, a sibling **Alternative** feature, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Alternative feature and the Configuration, an **Add** proposal object that is linked to the sibling Alternative feature and the Configuration, and a **Container**. The Alternative feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**, which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The sibling **Alternative** feature has three attributes: **name**, which takes the value of the parameter featureName1, **Rpayoff**, which takes the value of the parameter R1, and **Ppayoff**, which

takes the value of the parameter P1. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has one negative application condition. The proposals Withdraw and Add (Substitute) have not been made by the Prov itself. The negative application condition is expressed by a **forbid** action.

After applying the *Prov_accept_to_substitute*, objects expressed by **preserve** actions are preserved. The **Occurrence** object and its links are deleted. The **Add** object with its links are deleted. A new **Occurrence** object is created and linked to the sibling Alternative and the Configuration. Also, the value of the attribute Rtotal in the Configuration will be changed from Rt to Rt-R+R1 and the value of the attribute Ptotal in the Configuration will be changed from Pt to Pt-P+P1. It means that we subtract the value of the Alternative feature from the total and add the value of the sibling Alternative feature to the total of the requestor and the provider.

(j) **reject_to_substitute:** the negotiator rejects the other negotiator's proposal to substitute an Alternative feature.

In Figure 3.19, the *Prov_rejects_to_substitute* rule is shown. It has nine parameters, **featureName**, **featureName1**, **R**, **R1**, **P**, **P1**, **Rt**, **Pt** and **id**. According to its preconditions, there should be an **Alternative** feature that has an **Occurrence** in the Configuration, a sibling **Alternative** feature, a **Prov**, a **Configuration**, a **Withdraw** proposal object that is linked to the Alternative feature and the Configuration, an **Add** proposal object that is linked to the sibling Alternative feature and the Configuration, and a **Container**. The Alternative feature has three attributes: **name**, which takes the value of the parameter featureName, **Rpayoff**,

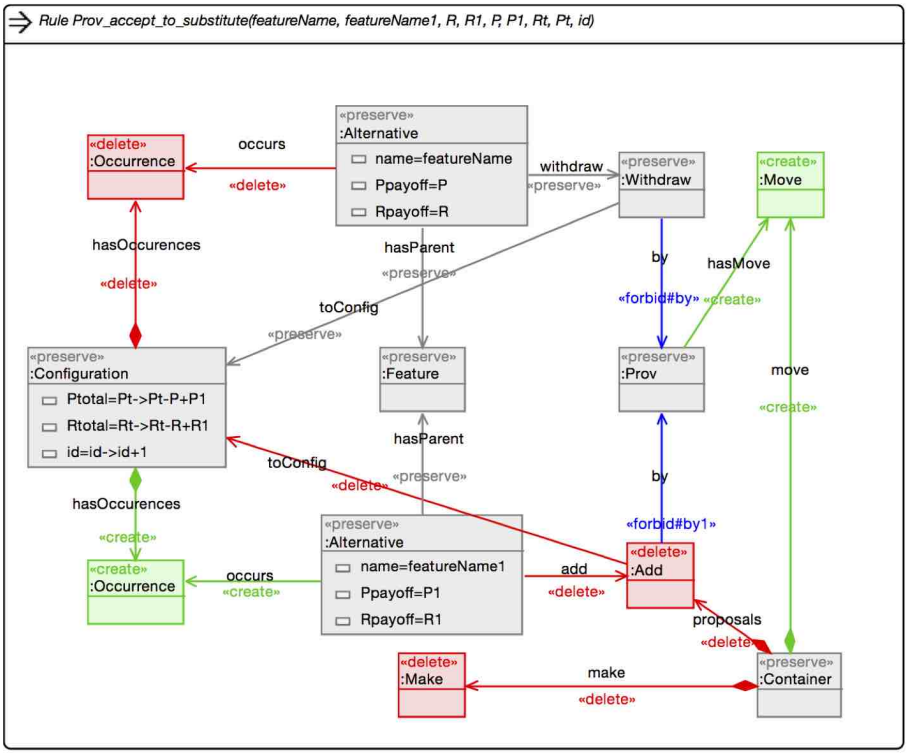


FIGURE 3.18: Transformation Rule *Prov_accept_to_substitute*

which takes the value of the parameter R, and **Ppayoff**, which takes the value of the parameter P. The sibling Alternative feature has three attributes: **name**, which takes the value of the parameter featureName1, **Rpayoff**, which takes the value of the parameter R1, and **Ppayoff**, which takes the value of the parameter P1. The Configuration has three attributes: **ID**, which takes the value of the parameter id, **Rtotal**, which takes the value of the parameter Rt, and **Ptotal**, which takes the value of the parameter Pt.

This rule has two negative application conditions. The sibling Alternative feature is not currently occurring in the Configuration. The proposals Withdraw and Add have not been made by the Prov itself. The negative application conditions are expressed by **forbid** actions.

After applying the *Prov_rejects_to_substitute* rule, objects expressed by

preserve actions are preserved and the link **toConfig** between Add object and the Configuration is deleted, and the **Withdraw** object with its links are deleted. The deletion is expressed by **delete** actions.

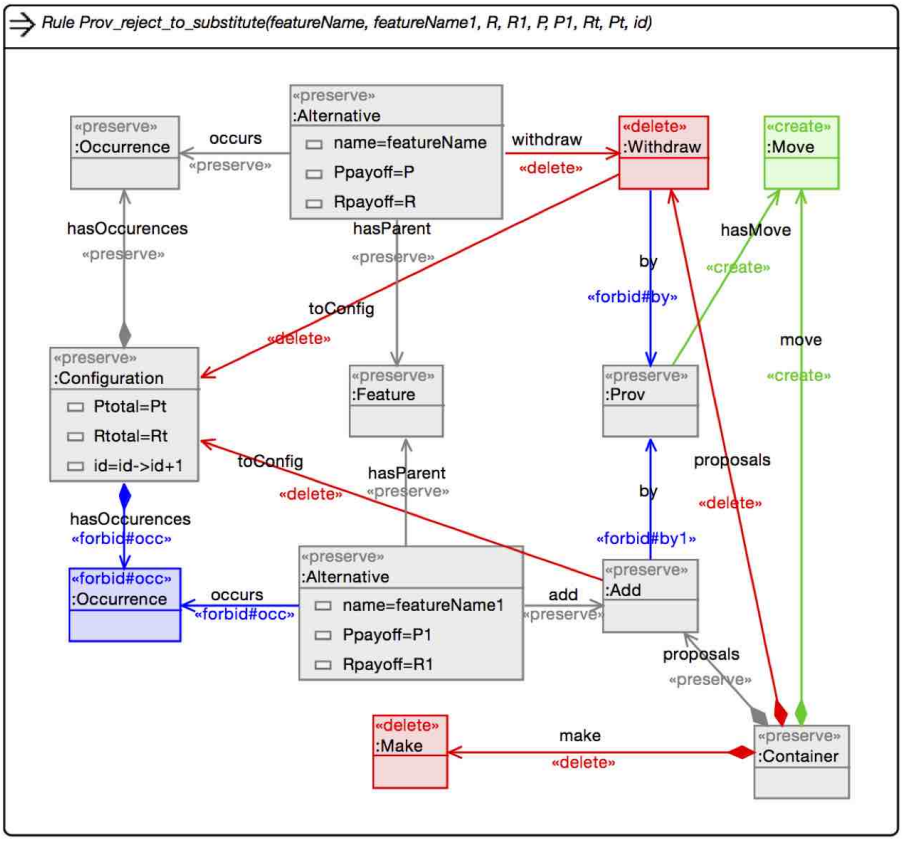


FIGURE 3.19: Transformation Rule *Prov_reject_to_substitute*

In the second version of our rules, we assign a unique number to each feature in ascending order starting from 0. These numbers are used to represent the order in which the features can be proposed. In Figure 3.20, the *Req_propose_to_addOpt* rule is shown. It is similar to the rule shown in Figure 3.5 except that the Optional feature has an attribute **Order**, which takes the value of the parameter **time**. It means that the feature cannot be proposed unless its order is equal to the value of the parameter **time**, which increments with every proposal.

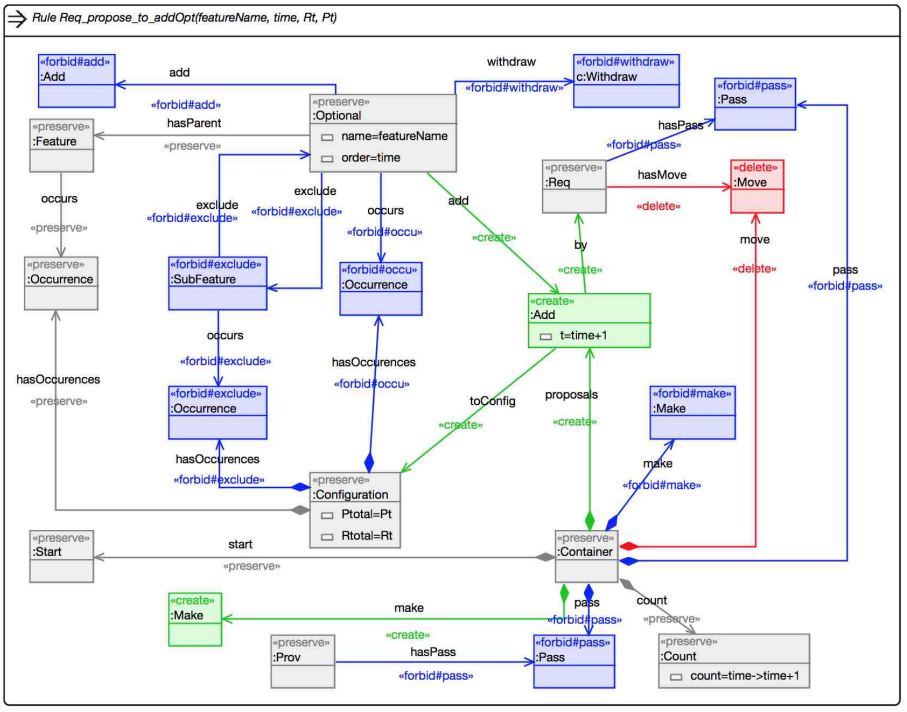


FIGURE 3.20: Transformation Rule *Req_propose_to_addOpt*

3.4.3 Generating the Transition System of the Game

We generate the graph transformation game in the form of a labelled transition system using Henshin State Space tools. It starts from some initial states and executes the transformation rules until reaching the terminal states where no rules can be applied. The initial state in our game should include an initial configuration. According to the design of our rules, the labelled transition system is generated as a tree because of the following:

1. Only starting rules can be executed at the initial state, which creates a rooted tree with only one player who can move.

2. The rules are designed to allow the taking of turns, which ensures that in any state only one player can move.

3. Making and responding to proposals are specified by the current time of proposing, which means that making the same proposal at a different time will lead to a different state. This ensures that each state has only one incoming edge (a child can only have one parent).

4. The time parameter in the Count class also ensures that there is no transition cycling as the time increments with every proposal. Thus, there is no possibility of going back.

The initial configuration in our running example contains: Airplane for Transportation and Hotel for Accommodation. In this case, our graph transformation game state space is generated containing 3300 states and 3299 transitions.

Returning to the properties discussed in Section 3.4.2, in the following, we will discuss how these properties have been considered in the implementation of our negotiation rules.

Firstly, our negotiation protocol is *flexible* in terms of allowing the negotiators to choose among all available proposals according to their preferences. It also allows them to pass turn if they do not want to make a proposal at a certain time. However, there is one restriction that limits its flexibility. Our negotiation rules disallow the negotiators from re-proposing the addition/withdrawing of features that were rejected once in order to generate a finite transition system. This may affect the applicability of our approach because some features may become relevant due to other changes in a new configuration. For example, in our running example, a **Train** rejected at a stage may become relevant again when an alternative **Accommodation** type is accepted.

When it comes to *simplicity*, our negotiation protocol is very simple in terms of the communication language that is required to propose and respond to proposals.

Our negotiation rules are clearly defined and implemented to be *applicable with feature modelling*. The implementation of our rules is based on the types of feature for both making proposals and responding to them.
When it comes to *equality*, both negotiators have similar rules that allow them to propose and respond to proposals and no one has higher power than others.

3.4.4 Scalability

We conducted experiments to evaluate the scalability of generating the graph transformation game state space. We apply our transformation rules to different feature models for both versions of our rules. In all experiments, we show the number of features in the feature models, the number of proposed features (proposals), the number of generated states, the number of transitions and the generation time. As we discussed in Section 3.4.2, the number of proposals is based on the types of features in each feature model example. Moreover, we use feature model examples with similar structure so that all proposals can be made from the initial state in order to generate larger state space. These experiments were conducted on 2.5 GHz Intel Core i7 with 16 GB of main memory using Henshin 1.0.0. In Table 3.2, we show the generation results after applying the first version of our negotiation rules. We applied the transformation rules to different feature model examples. We started with examples containing one to five proposals. We stopped at the fifth proposal because the expected state space for six proposals will be huge. This can be observed by looking at the difference between the number of generated states in the fourth and fifth rows.

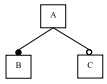
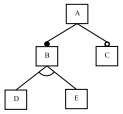
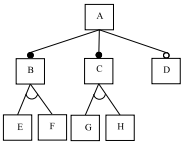
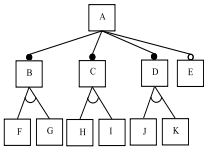
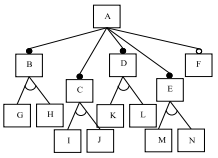
Feature Model	No of features	No of proposals	No of states	No of transitions	Generation time (Seconds)
	3	1	8	7	0.606
	5	2	71	70	0.684
	8	3	814	813	1.563
	11	4	11501	11500	10.767
	14	5	195956	195955	148.346

TABLE 3.2: Generation Results Using Alternating-offer Negotiation Protocol (1)

The generation results from applying the second version of our negotiation rules are shown in Table 3.3. We applied the transformation rules to examples containing one to 10 proposals. The results also show some improvements in the generation time compared with the first version.

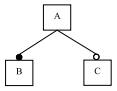
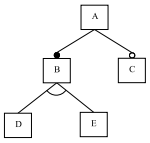
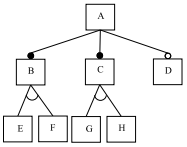
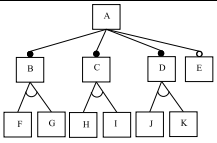
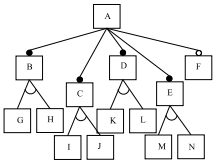
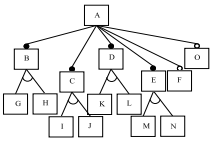
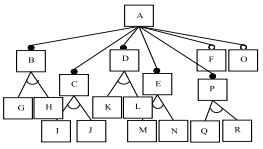
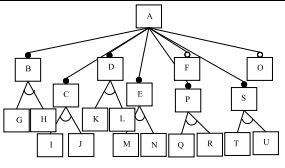
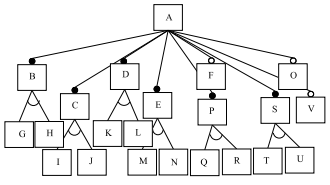
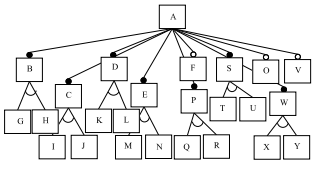
Feature Model	No of features	No of proposals	No of states	No of transitions	Generation time (Seconds)
	3	1	8	7	0.546
	5	2	36	35	0.648
	8	3	140	139	0.775
	11	4	500	499	1.159
	14	5	1716	1715	2.679
	15	6	5748	5747	6.631
	18	7	18996	18995	20.857
	21	8	62260	62259	47.488
	22	9	203060	203059	1566.107
	25	10	660276	660275	3174.711 (≈ 53 min)

Table 3.3: Generation Results Using Alternating-offer Negotiation Protocol (2)

3.5 Summary

This chapter proposed graph transformation games that model the negotiation of features between the provider and the requestor. The aim was to implement our negotiation games, modelling the negotiation of features by representing the state of the game by a graph and the moves of the players by graph transformation rules. A type graph has been developed to represent the negotiation entities and to implement our transformation rules. The rules have been implemented to model the negotiation interactions defined by our negotiation protocol. Henshin transformation tools have been used to implement the transformation rules and generate the game state space. We conducted different experiments to evaluate our proposed approach. The evaluation results show that the size of the game state space is affected by the number of proposed features, which might cause problems with large feature models. In the next chapters, we will present different types of games which will be used to analyse our graph transformation games.

Chapter 4

Extensive-Form Graph Transformation Games

In this chapter, we analyse our graph transformation games as two-player extensive-form games. We will discuss how our graph transformation games can be analysed using backward induction to determine the optimal action at each stage of the game for each player. These extensive-form games are non-zero-sum games, which means that the players do not play competitively and they do not intend to minimise each other's payoffs. However, each player will try to maximise its individual outcome regardless of what the other player gets.

Section 4.1 introduces the idea of analysing our graph transformation games as extensive-form games. Section 4.2 discusses how our graph transformation games can be analysed using backward induction. In Section 4.3, we provide a detailed explanation of the implementation of the backward induction algorithm. Section 4.4 summarises the chapter.

4.1 Introduction

The scenario of the graph transformation game illustrates that it is a dynamic and multi-stage game. It represents the structure of interaction between players, their possible moves and their choices at every state. In game theory, this scenario can be seen as a typical example of an extensive-form game in which the players move sequentially by exchanging proposals. Thus, we propose to analyse our graph transformation games as two-player non-zero-sum extensive-form games with complete information. Figure 4.1 shows an overview of the proposed approach. Henshin is used to generate our turn-based graph transformation games as extensive-form games.

Extensive-form games are defined as games in a tree structure with payoffs at the terminal nodes. Our graph transformation games, constructed based on our Alternating-offer Negotiation Protocol, are generated in the form of a tree-like, labelled transition systems. They have similar characteristics to the extensive-form games tree, as follows:

1. Only one player can start the game at the root, which is the requestor in our graph transformation games.

2. Only one player can move at each state so the moves of the players are distinguishable.

3. The payoffs at the terminal states are determined according to every possible play of actions.

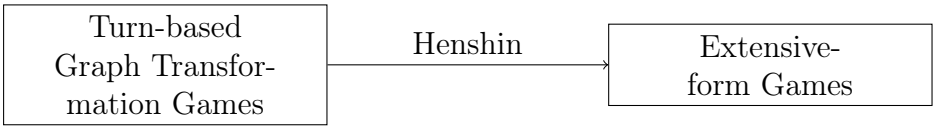


FIGURE 4.1: An Overview of the Proposed Approach

In extensive-form games, each player's payoff/utility function is defined on terminal histories. A terminal history is a sequence of actions for which no actions follow. In our graph transformation games, the payoff functions return real values for each player at each state of the game. Here, we are only interested in the payoffs at the terminal states, which indicate the players' outcomes at terminal sequences. A terminal sequence is a transformation sequence that starts from the initial state and terminates at a terminal state where no rule can be applied.
Given a set of rules R and start graph G_0, a **transformation sequence** $G_0 \overset{r_1}{\Rightarrow} G_1 \overset{r_2}{\Rightarrow} ... \overset{r_n}{\Rightarrow} G_n$ is a sequence of steps starting in the start graph G_0. A sequence is **terminal** if in the last graph G_n no rule $r \in R$ that can be applied.

Extensive-form Graph Transformation Game Definition:

The definition of an extensive-form graph transformation game therefore consists of:

- A **type graph** TG to define the set of possible states $\mathcal{G}(TG)$.
- A finite set $N = \{1, 2, ..., n\}$ of **players**.
- A set of **rules** R where $R(i) \subseteq R$ is the set of rules for player $i \in N$ and for $i \neq j \in N : R(i) \cap R(j) = \phi$.
- A **start graph** G_0 as initial state.
- For $i \in N$ a **payoff function** $payoff_i : TS \to \mathbb{R}$, defined for each terminal sequence.
- For $i \in N$ a **strategy** $s_i(G) = (r, m)$ that gives for each $G \in \mathcal{G}(TG)$ a rule $r \in R(i)$ and a match m for r in G.
- A **player function** $P : \mathcal{G}(TG) \to N$ that assigns a player $i \in N$ to each $G \in \mathcal{G}(TG)$, $P(G) = i$.

4.2 Overview of Game Analysis Method

Extensive-form games with complete information can be analysed using backward induction to determine a sequence of optimal actions. Backward induction assumes that each player will act rationally at each future state in the game, which is called sequential rationality. In our graph transformation games, backward induction is used to analyse and solve the games. Backward induction is a classic and powerful analytical tool for decision-making in settings that can be modelled as finite extensive-form games and, as will see, is a fundamental analytical tool in negotiation settings [65]. We analyse the graph transformation state space by reasoning backward, starting from the terminal states until reaching the initial state. We select the optimal transition at each state and eliminate non-optimal transitions. The result of backward induction is a strategy profile containing a strategy for each player, which is the Nash Equilibrium of the game. The strategies should tell the negotiators how to act during the negotiation process, what to propose and how to respond to the proposals. Solving the game by backward induction provides a Subgame Perfect Nash Equilibrium which represents a Nash equilibrium of every subgame of the original game.

As we discussed above, the optimality in this solution refers to the individually optimal actions for each negotiator at each stage. This is not neccesarily the best joint outcome. Moreover, there may be more than one Nash equilibrium as discussed in Section 2.2.2.

4.3 Implementing Backward Induction

In this section, we introduce the implementation of backward induction. Based on the given state space metamodel in Section 4.3.1, we define graph transformation

rules that are used to apply the backward induction algorithm in Section 4.3.2. In Section 4.3.3, we discuss the generation of the state space instance. In Section 4.3.4, we show the results of applying backward induction rules to our running example graph transformation game state space. In Section 4.3.5, we conduct some experiments to evaluate the proposed approach.

4.3.1 State Space Metamodel

The state space metamodel is given by Henshin as an Ecore model and available at [78]. This metamodel will be used as a type graph to design backward induction rules which will be applied to the graph transformation game state space instance. In Figure 4.2, we present the state space metamodel to define the state space elements. We are interested in the classes that define StateSpace, State and Transition. We modified the original metamodel by adding some attributes that will be used in the analysis of the game. The following is the list of modifications:

1. We added **ptotal** attribute of type Integer to the State class, which will be used to store the value of **Ptotal** attribute in the Configuration class in the state graph. This attribute represents the payoff of the provider.

2. We added **rtotal** attribute of type Integer to the State class, which will be used to store the value of **Rtotal** attribute in the Configuration class in the state graph. This attribute represents the payoff of the requestor.

3. We added **transitionLabel** attribute of type String to the Transition class to store the transition labels.

4. We added **mover** attribute of type String to the Transition class to specify the movers at each state.

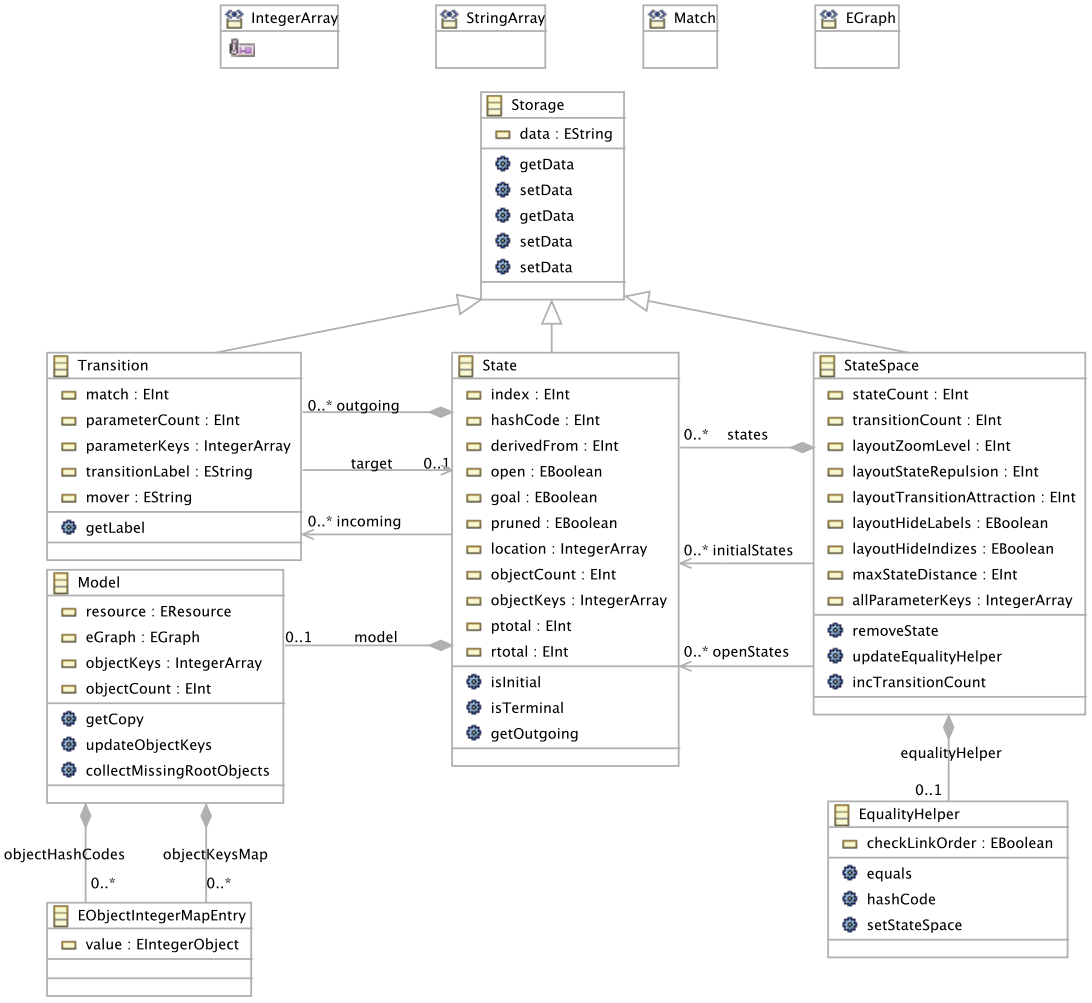


FIGURE 4.2: State Space Metamodel

4.3.2 Backward Induction Rules

Based on the metamodel in Figure 4.2, we can define the rules that are used to reason through the state space backward starting from terminal states to the initial state. Before defining our rules, we show the backward induction algorithm as described in [1]. Then, we show how our rules will work to apply the backward induction algorithm.

The backward induction procedure, shown in Figure 4.3, is as follows:

- **Step 1**: select any pen-terminal node, i.e., nodes preceding terminal nodes.

- **Step 2**: select one move that gives the mover the highest payoff.
- **Step 3**: assign the payoff vector for both players to the node at hand.
- **Step 4**: eliminate all moves and terminal nodes following this node. We will have a shorter game where this node will be a terminal node.

We repeat these steps until we only have the origin (initial node). The moves picked are the outcome of the game and the result is a strategy profile.

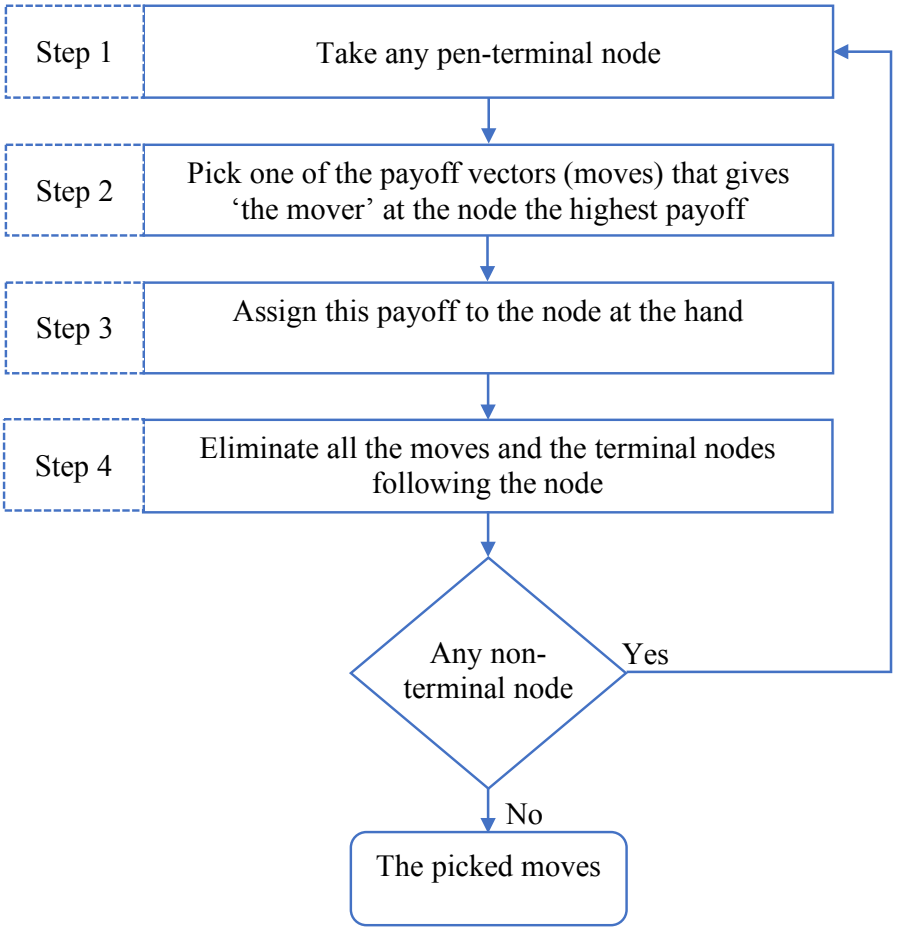


FIGURE 4.3: Backward Induction Algorithm [1]

In Henshin, we created different rules to apply the backward induction algorithm to our graph transformation state space instance. The only difference is that, in

Step 4, we do not eliminate all transitions and states but we eliminate non-optimal transitions and keep the optimal transitions to show the strategy profile later. For that reason, we created some rules to be applied first by selecting the pen-terminal and terminal states to ensure that we started from the end of the tree. Then, we created other rules to follow the same procedure with internal states but we have to ensure that we have visited all successor states of the states at hand before eliminating non-optimal transitions. We divided these rules into two categories according to the mover, which can be either the requestor or the provider. The rules in both categories are almost the same except for the mover of the transitions.
For brevity, we show the design of three rules as examples and how they follow the procedure of the backward induction algorithm described above:

1. LeavesReq rule: this rule is used to select the highest payoff of the requestor in pen-terminal states. In Figure 4.4, the *LeavesReq* rule is shown. It has seven parameters, **x**, **rt**, **pt**, **rt1**, **pt1**, **pt2** and **rt2**. According to its preconditions, three **State(s)** have to be found.

   The first State represents a pen-terminal state and it has two outgoing **Transition(s)** to the other two states. It has three attributes: **data**, which takes the value of the parameter x, **ptotal**, which takes the value of parameter pt, and **rtotal**, which takes the value of the paramter rt.

   The second State has two attributes: **ptotal**, which takes the value of the parameter pt1, and **rtotal**, which takes the value of the parameter rt1.

   The third State has two attributes: **ptotal**, which takes the value of the parameter pt2, and **rtotal**, which takes the value of the parameter rt2.

   Each Transition has an attribute **mover**, which takes a String "Req" to specify that the mover is the requestor.

   The rule has two negative application conditions. The second and third states have no outgoing transitions, which means that they are terminal states. **(Step**

1)

The rule has an attribute condition, which is that the value of rt1 is greater than or equal to the value of rt2. This means that we pick up the highest payoff of the requestor. **(Step 2)**

After applying the *LeavesReq* rule, objects expressed by <<preserve>> actions are preserved. The values of both ptotal and rtotal in the first state are changed from pt and rt to pt1 to rt1 respectively. This means that we assign the highest payoff to the pen-terminal state. **(Step 3)**

The Transition object between the first and third states with its links are removed **(Step 4)**. Here, we only remove the non-optimal transitions as we keep the picked ones.

In the first state, the value of attribute data is changed from x to a String "Explored" to keep tracking of visited states.

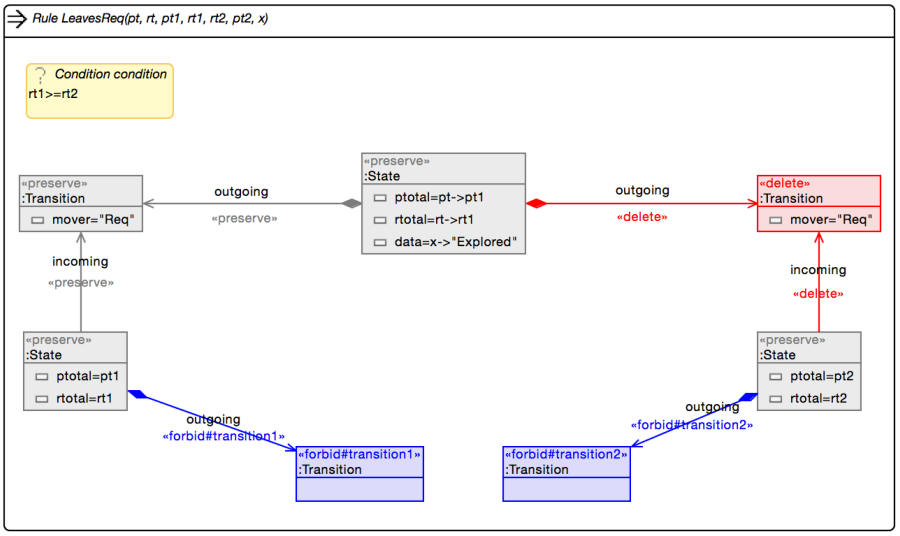


FIGURE 4.4: Transformation Rule *LeavesReq*

2. CompareReq rule: this rule is used to select the highest payoff of the requestor in intermediate states.

   In Figure 4.5, the *CompareReq* rule is shown. It is similar to the *LeavesReq* rule except that the second and third states have no negative application conditions,

which means that they are not terminal states. The second and third states must be visited (explored). That means that their optimal transitions have been picked up. This rule is only applied when the first state has only two outgoing transitions. Thus, after applying this rule, only one optimal transition is remaining and all non-optimal transitions have been eliminated.

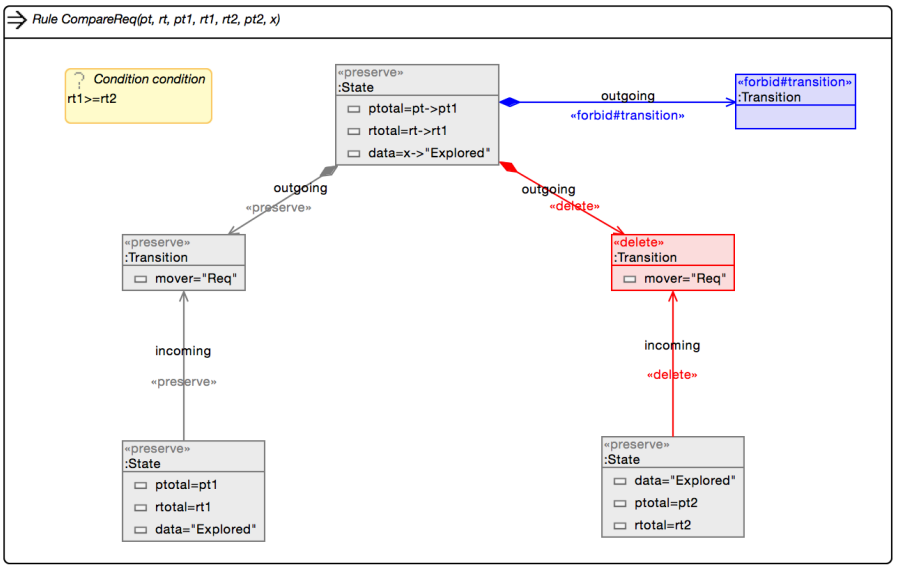


FIGURE 4.5: Transformation Rule *CompareReq*

3. CompareReq1 rule: in Figure 4.6, we show the *CompareReq1* rule. This rule is similar to the *CompareReq* rule but here the rule can be applied if the first state has more than two outgoing transitions. After applying the rule, the attribute data in the first state will not be changed, which indicates that the optimal transition has not been picked up yet.

In the following, we provide a brief description of how our backward induction rules work to obtain a strategy profile which is a subgame perfect equilibrium.

- **Determining the optimal transitions in pen-terminal states**: we started by determining the optimal transitions in the pen-terminal states as follows:

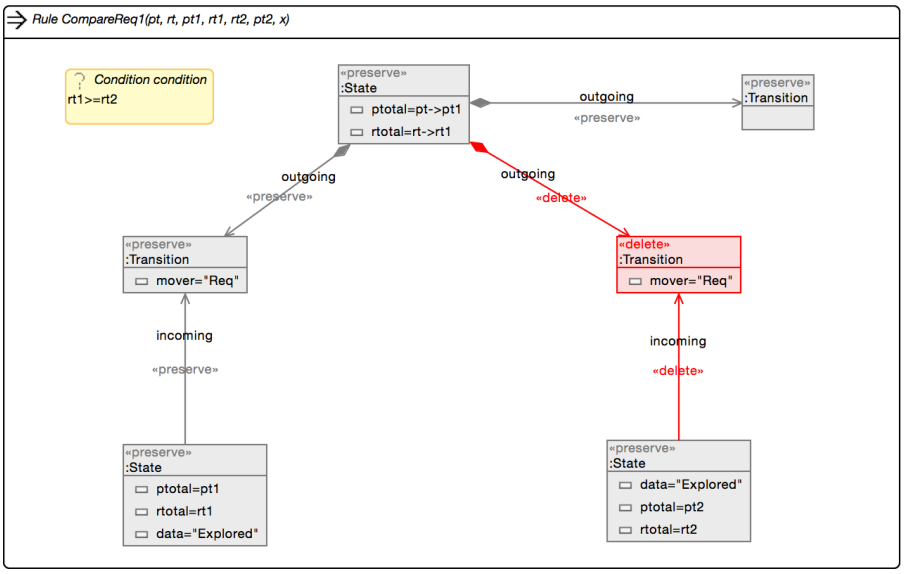


FIGURE 4.6: Transformation Rule *CompareReq1*

- Select any pen-terminal state.
- Pick up the transition that gives the mover the highest payoff.
- Assign this payoff to the state at the hand.
- Eliminate non-optimal transitions.
- Change the value of data attribute in the state at the hand to "Explored".

- **Determining the optimal transitions in intermediate states**: after determining the optimal transitions in all pen-terminal states, we move on to determine the optimal transitions in the intermediate states. We follow the same steps as in the pen-terminal states except that we need to check that the value of data attribute of all successor states of the state at the hand is equal to "Explored" which means that they have been visited and their optimal transitions have been picked up. This ensures that all states have been visited and their optimal transitions have been picked up before reaching the initial state.

The obtained optimal transitions constitute the strategy profile which is a subgame perfect equilibrium.

The application of our rules can be described by the pseudo code in Algorithm 1:

Algorithm 1 Backward Induction

Input: Graph transformation state space instance with
set of states S,
and set of transitions T

Output: New graph transformation state space instance after eliminating non-optimal transitions

while *there exists a pen-terminal state* $p \in S$ *with more than one outgoing transition* **do**
  Pick up optimal outgoing transition for the mover according to the payoffs in the sucessor states,
  Assign the selected payoffs to p,
  Set p as visited state,
  Eliminate non-optimal outgoing transition
end
while *there exists an intermediate state* $n \in S$ *with more than one outgoing transition and all its sucessor states are visited* **do**
  Pick up optimal outgoing transition for the mover according to the payoffs in the sucessor states,
  Assign the selected payoffs to n,
  Set n as visited state,
  Eliminate non-optimal outgoing transition
end

4.3.3 Generating the State Space Instance

As we discussed, we modified the original state space metamodel by adding new attributes to the classes. In this phase, we encode the values of these attributes to the graph transformation game state space instance. We extract the values from each state graph and encode them to the state instance in our state space instance.

4.3.4 Application to Running Example

We applied backward induction rules to our running example graph transformation game state space. The result assigns 15 as the highest payoff for the requestor, which also gives 16 to the provider for the configuration: Airplane and Train for Transportation, Hotel for Accommodation, and Catering. This gives better results for both negotiators than the initial configuration (Airplane for Transportation and Hotel for Accommodation), in which the payoff of the requestor was 10 and the provider was 9. The optimal transition for each player at each state represents the strategy profile, which is the Nash Equilibrium of the game. In our running example graph transformation game state space, we have 3300 states. In Table 4.1, we show the optimal transitions for the players at the first 20 states.

State	Transition	State	Transition
0	Req_start_to_addOr	10	Req_accept_to_addOr
1	Prov_accept_to_addOpt	11	Req_reject_to_substitute
2	Prov_accept_to_addOr	12	Req_propose_to_substitute
3	Prov_reject_to_substitute	13	Req_accept_to_addOr
4	Prov_propose_to_addOr	14	Req_reject_to_substitute
5	Prov_propose_to_addOr	15	Req_propose_to_substitute
6	Prov_propose_to_withdrawOr	16	Req_accept_to_addOpt
7	Prov_propose_to_substitute	17	Req_reject_to_withdrawOr
8	Prov_propose_to_addOr	18	Req_reject_to_substitute
9	Prov_propose_to_addOr	19	Req_propose_to_withdrawOr

TABLE 4.1: The Optimal Transitions for the Players in Our Running Example

4.3.5 Scalability

We conducted experiments to explore the scalability of our analysis. We apply backward induction rules to the graph transformation games generated in Section 3.4.4. We measure the speed in applying the rules, which represents the required analysis time for each graph transformation input. In all experiments, we show the number of states and transitions in the graph transformation game, the time to generate the

state space instance, the number of rule applications and the total application time. In Table 4.2, we show the generation results from applying our backward induction rules to graph transformation games generated in Table 3.2. In Table 4.3, we show the results from applying our backward induction rules to graph transformation games generated in Table 3.3, in which the negotiators propose in a specific order.

In all experiments, the results show that the generation of backward induction

Graph Transformation Game		**Instance generation time (Seconds)**	**Backward induction rules**	
No of states	**No of transitions**		**No of rules applications**	**Application time (Seconds)**
8	7	3.435	1	0.986
71	70	5.528	19	1.192
814	813	6.203	245	15.100

TABLE 4.2: Alternating-offer Negotiation Protocol (1) Backward Induction Results

Graph Transformation Game		**Instance generation time (Seconds)**	**Backward induction rules**	
No of states	**No of transitions**		**No of rules applications**	**Application time (Seconds)**
8	7	3.358	1	0.991
36	35	5.554	9	1.091
140	139	5.643	41	1.460
500	499	5.964	153	5.980
1716	1715	7.407	537	173.153

TABLE 4.3: Alternating-offer Negotiation Protocol (2) Backward Induction Results

analysis is affected by the size of the graph transformation game. For that reason, in Table 4.2, we stop at the third example because the generation of the fourth example takes up too much time ($\approx$ 4 hours) without returning any results. The version in Table 4.3 provides better results because it produces smaller games.

4.4 Summary

In this chapter, we proposed analysing our graph transformation games as two-player non-zero-sum games in an extensive form. We showed how the graph transformation games can be analysed using backward induction to obtain the game results, which is considered as a Nash equilibrium of the game. We provided a detailed explanation of the implementation of the backward induction algorithm. We conducted different experiments to evaluate the proposed analysis. The evaluation results show that the proposed analysis does not scale with large examples.

Chapter 5

Stochastic Graph Transformation Games

In this chapter, we analyse our graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker. We will discuss how to export the graph transformation games into the PRISM-games format, which includes defining the players of the game, defining the modules to describe the possible states and the ways in which the states change over time, and defining the players' rewards. We will also define the properties to be checked in PRISM-games. The specification of the properties is based on the temporal logic rPATL. After checking the properties, PRISM-games also supports strategy synthesis to generate the optimal strategies for the players.

In this chapter, we introduce the idea of analysing our graph transformation games as two-player turn-based stochastic games in Section 5.1. Section 5.2 presents the generation of the graph transformation games into PRISM-games format. In Section 5.3, we provide a detailed explanation of the analysis of our graph transformation games including defining properties and generating strategies. Section 5.4 presents some experiments for evaluation. Section 5.5 concludes the chapter.

5.1 Introduction

In our negotiation, the negotiators interact by taking turns in making and responding to proposals. Each negotiator tries to maximise their gain by requesting features with the highest gain. This negotiation situation can be modelled as a stochastic game where the uncertainty in this problem comes from the negotiators' unpredictable reaction to proposals. The negotiators can behave erratically, either deliberately in order to be less predictable or because their individual preferences differ from the average. In such a situation, it is important to show how uncertainty affects the negotiation outcome. Thus, we propose to analyse our graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker by exploring different strategies for players.

In a turn-based multi-player stochastic game (SMG), there is a finite number of players, a finite number of states and a finite set of actions. At each state, only one player can choose from a set of available actions. In our graph transformation games, the transition system of the game is finite because the number of configurations in a feature model is finite. We have two negotiators who interact by taking turns. The graph transformation rules are designed so that only one player can make a move at any state. Therefore, our graph transformation games can be modelled as turn-based multi-player stochastic games.

In Figure 5.1, we show two possible ways of generating our turn-based graph transformation games to turn-based multi-stochastic games in PRISM-games. The first way is by generating the labelled transition systems of our graph transformation games from Henshin to PRISM-games format. The second way is by implementing our negotiation of features using the PRISM-games model checker directly. In our work, we consider the first way while we leave the second way for future work as discussed in Section 7.4.4.

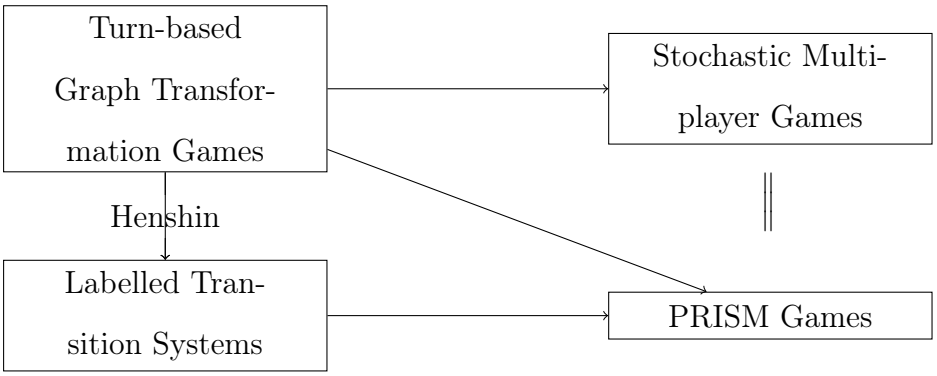


FIGURE 5.1: An Overview of the Proposed Approach

In SMGs, reward structures assign a real value to each state, which the players receive as payoffs. It is also possible in SMGs to assign the rewards to transitions. The total payoff over a path is the sum of the payoffs over each state in the path [79]. Thus, in PRISM-games, the payoffs are calculated in a different way from the original calculation in our graph transformation games. Here, we need to differentiate between the payoff (reward) at each state and the total payoff over a path to each state. Instead of giving the total value of selected features in each state, we give a value to each state according to the changes to the configuration, except for the initial state, which should have the total values of selected features in the initial configuration. So, after adding a feature to the configuration, we assign its value as a positive payoff to the current state. Similarly, if we withdraw a feature from the configuration, we assign its value as a negative payoff to the current state. In case there are no changes, we assign a zero payoff to the current state. However, the current version of PRISM-games allows us to use mixed-sign reward structures although the developers assume that the reward is either non-negative or non-positive for all states in order to express minimisation problems via maximisation [52, 55, 80].
The total payoff at a state is the sum of the payoffs over each state in the path to that state. This should give the total value of selected features in the configuration at each state. In order to apply these changes to our graph transformation games,

we only modified the rules, in particular the Ptotal and Rtotal attributes' values in the Configuration instance to obtain the values as described here. For example, in Figure 5.2, we show the *Req_accept_to_addOpt* rule after modifying the values of these attributes. In this rule, the values of both Ptotal and Rtotal attributes in the Configuration will be replaced with the values of Ppayoff and Rpayoff attributes of the Optional feature, respectively, to indicate that the feature has been added to the Configuration. In the original rules, the values of the added optional feature were added to the total in the configuration.

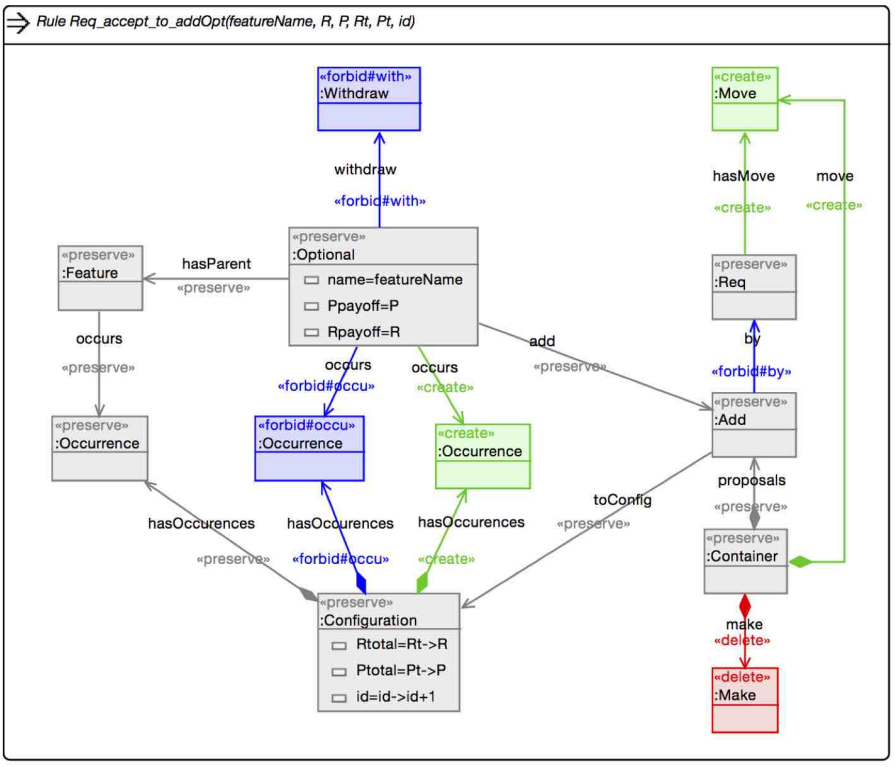


FIGURE 5.2: Transformation Rule *Req_accept_to_addOpt*

Stochastic Graph Transformation Game Definition:

The definition of a stochastic graph transformation game consists of:

- A **type graph** TG to define the set of possible states $\mathcal{G}(TG)$.

- A finite set $N = \{1, 2, ..., n\}$ of **players**.
- A set of **rules** R where $R(i) \subseteq R$ is a set of rules for player $i \in N$ and for $i \neq j \in N : R(i) \cap R(j) = \phi$.
- A **start graph** G_0 as initial state.
- For $i \in N$ a **payoff function** $payoff_i : STS \to \mathbb{R}$, where STS is the set of transformation sequences.
- For $i \in N$ a **strategy** $s_i(G) = (r, m)$ that gives for each $G \in \mathcal{G}(TG)$ a rule $r \in R(i)$ and a match m for r in G.
- A **player function** $P : \mathcal{G}(TG) \to N$ that assigns a player $i \in N$ to each $G \in \mathcal{G}(TG)$, $P(G) = i$.
- A **labelling function** $label : R \to L$ that assigns labels to the rules.
- A **rating function** $rate : R \to \mathbb{R}$ that assigns a rate to each rule.

As we discussed, the uncertainty in our negotiation comes from the unpredictable behaviour of the negotiators, who have different levels of rationality. Thus, stochastic concepts are required to model these negotiation situations. We define a stochastic element in our stochastic graph transformation games by assigning rates to the transitions (rules). The rates are used to specify the probability distribution over the outgoing transitions with the same label from the same state.
The following function returns the set of outgoing transitions with label l from state s:

$$T_l(s) = \{s \xrightarrow{r} s' \mid label(r) = l\}$$

Stochastic games use MDP structures, where the total probability of transitions of the same label outgoing from each state is one. Thus, we divide the rate of each

outgoing transition over the total rates of outgoing transitions with the same label. The probability is computed by the following function:

$$prob(s \xrightarrow{r} t) = \frac{rate(r)}{\sum\limits_{s \xrightarrow{r'} t' \in T_{label(r)}(s)} rate(r')}$$

5.2 Generating the PRISM Game

We model our stochastic graph transformation games in PRISM-games as follows:

Players: we have two players, the provider and the requestor, who play by making proposals to add and withdraw features from the configuration and responding to them. The players and the distribution under their control are specified by **player ... endplayer** constructs.

Modules: a model in PRISM-games consists of modules that describe the behaviour of the players, the state is determined by a set of variables and the behaviour is specified by guarded commands. We require one module whose state is defined by a variable with value 0 to the maximum number of states in our graph. The behaviour of the players is specified by a guarded command as follows:

$$[action]\ guard -> update;$$

The $action(s)$ are the labels of the transitions (rules) in our graph transformation games. Each action belongs to only one player. If the *guard* is satisfied, the module updates its variable according to the *update*. For example, in our model, a guarded command could be the following:

$$[Req\_propose\_to\_addOpt]\ s = 1 -> (s' = 2);$$

Rewards: the values of the features are represented as rewards associated with states in PRISM-games. A positive reward indicates that a feature has been added and a negative reward indicates that a feature has been withdrawn. A zero reward in a state indicates that no feature has been added or withdrawn.

From Henshin to PRISM-games:

Henshin supports generating the state space to several formats such as continuous-time Markov chain (CTMC) and Markov decision process (MDP) to be analysed by the PRISM model checker. In our work, we modified Henshin source code, in particular the one used to generate MDPs, to generate our state space in PRISM-games format (SMGs) (see Appendix B.3.2) as follows:

- We used the **smg** keyword which indicates the SMG model instead of the **mdp** keyword.
- We created two **player ... endplayer** constructs, one for the provider and one for the requestor. We used the transition prefixes "Prov" and "Req" in our graph transformation games to map the transitions under each player's construct. The pseudo code in Algorithm 2 describes how we create players' constructs and transitions under each player's construct.
- We created one module M that contains the guarded commands to specify the behaviour of the players as discussed above. In order to model the stochastic element in our games, we relabelled both 'accept' and 'reject' transitions as 'respond' and assigned a probability distribution to them according to their rates[1]. For example, if the rate of Req_accept_to _addOpt is 3 and Req_reject_to_addOpt is 1, a 'respond' transition could be the following:

$$[Req\_respond\_to\_addOpt]\ s = 1 -> 0.75 : (s' = 1) + 0.25 : (s' = 2);$$

[1]Knowledge of past behaviour of the players could be used to estimate these rates.

Algorithm 2 Players' Constructs Creation

Input: Graph transformation state space instance,
S is the set of states,
T is the set of transitions,
stateCount is the number of states in the state space instance
Output: Players' constructs for SMGs in PRISM-games

```
Write:("smg") \\ smg keyword
Write:("player requestor") \\ starting requestor's construct
for (i=0; i< stateCount; i++) do
    for (j=0; j< the number of outgoing transitions from S(i); j++) do
        if (outgoing transition T(j) has a label starts with "Req") then
            \\ we write the transition labels that start with prefix "Req"
            Write:("["+ label of T(j) + "],")
        end
    end
end
Write:("endplayer") \\ closing requestor's construct
Write:("player provider") \\ starting provider's construct
for (i=0; i< stateCount; i++) do
    for (j=0; j< the number of outgoing transitions from S(i); j++) do
        if (outgoing transition T(j) has a label starts with "Prov") then
            \\ we write the transition labels that start with prefix "Prov"
            Write:("["+ label of T(j) + "],")
        end
    end
end
Write:("endplayer") \\ closing provider's construct
```

This command means that the requestor accepts adding the optional feature with a probability of 0.75 and rejects with a probability of 0.25. The pseudo code in Algorithm 3 shows the implementation of the module with the guarded commands.

- We created two reward structures, "prov" and "req", where "prov" assigns the rewards for the provider at each state and "req" assigns the rewards for the requestor at each state. In Algorithm 4, we show a pseudo code to describe how reward structures are created.

Algorithm 3 Module Creation

Input: Graph transformation state space instance,
S is the set of states,
T is the set of transitions,
stateCount is the number of states in the state space instance
acceptRate is the rate for acceptance
rejectRate is the rate for rejection
Output: Module for SMGs in PRISM-games
Write:("module M") \\ starting module
Write:("s : [0.." + stateCount-1 + "] init 0") \\ state variable
for *(i=0; i< the number of transition in T; i++)* **do**
  if *(transition T(i) label does not contain "accept" or "reject")* **then**
    \\ we write the guarded commands that have no probabilities
    Write:("[" + label of T(i) + "] " + source state of T(i) + " -> " + target state of T(i) +";")
  end
  \\ Only accept and reject transitions have probabilities here
  if *(transition T(i) label contains "accept")* **then**
    Write:("[" + label of T(i) + "] " + source state of T(i) + " -> " + acceptRate/(acceptRate+rejectRate) + ":" + target state of T(i))
  end
  if *(transition T(i) label contains "reject")* **then**
    Write:(" + "+ rejectRate/(acceptRate+rejectRate) + ": " + target state of T(i) + ";")
  end
end
Write:("endmodule") \\ closing module

5.3 Analysing the Game

PRISM-games supports strategy synthesis to obtain optimal strategies for the players. The strategy determines for each player at each state what action should be taken. Each strategy can be analysed manually in the simulator view or exported to a file. The exported file contains a matrix with two columns (see Appendix B.3.3). The first column shows the list of states and the second column shows the choice taken in each state. In this section, we explore different strategies and generate a verified strategy satisfying certain minimal requirements.

Algorithm 4 Reward Structures Creation

Input: Graph transformation state space instance,
S is the set of states,
T is the set of transitions,
stateCount is the number of states in the state space instance
Output: Reward structures for SMGs in PRISM-games
Write:("rewards "req"") \\ starting "req" reward structure
for *(i=0; i<stateCount; i++)* **do**
  int rTotal= the value of Rtotal attribute from the configuration in the state graph of S(i)
  Write:("s="+ i + " : " + rTotal + ";")
end
Write:("endrewards") \\ closing "req" reward structure
Write:("rewards "prov"") \\ starting "prov" reward structure
for *(i=0; i<stateCount; i++)* **do**
  int pTotal= the value of Ptotal attribute from the configuration in the state graph of S(i)
  Write:("s="+ i + " : " + pTotal + ";")
end
Write:("endrewards") \\ closing "prov" reward structure

5.3.1 Single-objective Strategy

We define single-objective rPATL reward-based properties to explore the best individual outcomes. These reward-based properties have the forms:

<<provider>> R{"prov"}max=? [F "deadlock"]

<<requestor>> R{"req"}max=? [F "deadlock"]

The first property asks PRISM-games to generate an optimal strategy for the provider. It returns the maximum expected accumulated value of reward "prov" until reaching deadlock states, which represent the terminal states in our state space. At the same time, PRISM-games also generates the optimal strategy for a requestor seeking to minimise the value of reward "prov" of the provider. That means the game is considered a zero-sum game and the optimal strategies generated by PRISM-games

represent a Nash Equilibrium [79]. Similarly, the second property generates the optimal strategy for the requestor and returns the maximum expected accumulated value of reward "req" until reaching deadlock states.

Clearly, this is too limited a point of view for a negotiation, where a joint optimum needs to be found, but it helps to understand which rewards can be expected.

In our running example, we assigned equal rates to accept and reject transitions so each negotiator accepts with a probability of 0.5 and rejects with a probability of 0.5. PRISM-games returns 10 as the maximum expected reward that the provider can guarantee and 10.25 as the maximum expected reward that the requestor can guarantee. In Table 5.1, we show a possible plan of actions generated by the provider's strategy. We manually added the proposed features.

The results show that, due to the competitive nature of the game and the probability distribution over transitions, there is only a very small chance of achieving more than what was present in the initial configuration. However, the results are affected by the probability distribution over transitions. For example, if the negotiators accept with a probability of 0.25 and reject with a probability of 0.75, PRISM-games returns 9.75 as the maximum expected reward for the provider, which is worse than the previous results, but it returns 10.3 as the maximum expected reward for the requestor, which is a slightly better than the previous results.

State	Feature	Action	Probability	Reward "prov"
				9
1	Catering	[Req_start_to_addOpt]	1	0
4		[Prov_respond_to_addOpt]	0.5	5
10	Train	[Prov_propose_to_addOr]	1	0
45		[Req_respond_to_addOr]	0.5	2
110	Hotel	[Req_propose_to_substitute]	1	0
239		[Prov_respond_to_substitute]	0.5	-3
476	Airplane	[Prov_propose_to_withdrawOr]	1	0
860		[Req_respond_to_withdrawOr]	0.5	-4

TABLE 5.1: Example of Generated Provider's Strategy

5.3.2 Multi-objective Strategy

We define a multi-objective reward-based property to achieve a more collaborative negotiation as follows.

<<provider, requestor>> ((R{"prov"}>=pmax [C] & R{"req"}>=rmax [C]))

Here, *pmax* is the maximum expected value of reward "prov" and *rmax* is the maximum expected value of reward "req" generated by the previous strategies. This property asks PRISM-games to generate a collaborative strategy for both players which guarantees that the expected total reward values for reward structures "prov" and "req" are at least *pmax* and *rmax*, respectively. In Table 5.2, we show a plan of actions generated by the collaborative strategy. When collaborating, the maximum reward that the provider can guarantee is 12.5 and the maximum reward that the requestor can guarantee is 12.5. This indicates a better result than playing competitively.

Returning to our running example, the negotiation game allows the requestor to explore the possible alternatives to its original requirements, in particular the car hire. Its collaborative strategy with the provider gives better total value than playing competitively.

State	Feature	Action	Probability	Rewards	
				"req"	"prov"
				10	9
1	Catering	[Req_start_to_addOpt]	1	0	0
4		[Prov_respond_to_addOpt]	0.5	3	5
10	Train	[Prov_propose_to_addOr]	1	0	0
45		[Req_respond_to_addOr]	0.5	2	2
111		[Req_pass]	1	0	0
245		[Prov_pass]	1	0	0

TABLE 5.2: Example of Generated Collaborative Strategy

5.4 Scalability

We conducted experiments to investigate the scalability of our analysis. We use the graph transformation games generated in Section 3.4.4. In all experiments, we measure the time taken to export our state space transition system to PRISM-games format, the time taken to construct the SMG model in PRISM-games and the time taken to generate the strategy. These experiments were conducted on a 2.5 GHz Intel Core i7 with 16 GB of main memory using Henshin 1.0.0. In Table 5.3, we show the results of analysing our graph transformation games generated in Table 3.2. We use the provider's strategy in these experiments.

Graph Transformation Game		**PRISM-games**		
No of states	**No of transitions**	**Exporting time to PRISM-games (Seconds)**	**Construction time in PRISM-games (Seconds)**	**Strategy generation time (Seconds)**
8	7	0.205	0.03	0.001
71	70	0.273	0.032	0.003
814	813	0.835	0.062	0.015
11501	11500	6.383	10.981	2.769
195956	195955	90.588	7088.807	2462.566

TABLE 5.3: The Results of Analysing Our Graph Transformation Games (1) Using Provider's Strategy

In Table 5.4, we show the generation results from analysing our graph transformation games generated in Table 3.3. We also use the provider's strategy in these experiments. In Tables 5.5 and 5.6, we conducted the same experiments but here we measure the time to generate the multi-objective strategy.

Graph Transformation Game		PRISM-games		
No of states	**No of transitions**	**Exporting time to PRISM-games (Seconds)**	**Construction time in PRISM-games (Seconds)**	**Strategy generation time (Seconds)**
8	7	0.189	0.035	0.001
36	35	0.22	0.033	0.003
140	139	0.371	0.038	0.004
500	499	0.645	0.048	0.01
1716	1715	1.558	0.198	0.057
5748	5747	2.706	1.965	0.696
18996	18995	7.97	33.229	7.21

TABLE 5.4: The Results of Analysing Our Graph Transformation Games (2) Using Provider's Strategy

In all experiments, constructing the models in PRISM-games and generating strategies took up most of our time, especially with larger models. This is because we constructed a flat labelled transition system in PRISM-games rather than its specification. Also, we observe that generating multi-objective strategies took more time than generating single strategies. For example, in Table 5.4, the strategy generation time in the last row is 7.21 seconds, while in Table 5.6, the strategy generation time for the same example is 19.583 seconds.

Graph Transformation Game		PRISM-games		
No of states	**No of transitions**	**Exporting time to PRISM-games (Seconds)**	**Construction time in PRISM-games (Seconds)**	**Strategy generation time (Seconds)**
8	7	0.205	0.031	0.048
71	70	0.273	0.041	0.066
814	813	0.835	0.087	0.289
11501	11500	6.383	16.187	7.669
195956	195955	90.588	8181.228	6457.284

TABLE 5.5: The Results of Analysing Our Graph Transformation Games (1) Using Collaborative Strategy

Graph Transformation Game		**PRISM-games**		
No of states	**No of transitions**	**Exporting time to PRISM-games (Seconds)**	**Construction time in PRISM-games (Seconds)**	**Strategy generation time (Seconds)**
8	7	0.189	0.018	0.005
36	35	0.22	0.02	0.013
140	139	0.371	0.029	0.043
500	499	0.645	0.038	0.145
1716	1715	1.558	0.254	0.607
5748	5747	2.706	3.177	2.561
18996	18995	7.97	48.229	19.583

TABLE 5.6: The Results of Analysing Our Graph Transformation Games (2) Using Collaborative Strategy

5.5 Summary

In this chapter, we proposed analysing our graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker. We discussed how our graph transformation games can be modelled as turn-based stochastic games. We presented the requirements of mapping our graph transformation games from Henshin to PRISM-games format. We analysed the games by exploring different strategies to optimise the negotiators' rewards considering our running example presented in Chapter 3. To evaluate the proposed analysis, we conducted some experiments on different graph transformation games to measure the construction time and strategy generation time in PRISM-games.

Chapter 6

Related Work

In this chapter, we discuss techniques addressing similar or related problems. To the best of our knowledge, the use of graph transformation and game theory to implement and analyse the negotiation of features has not been explored elsewhere, one of the reasons being that using feature models to increase the flexibility of e-commerce negotiation has not attracted much attention from researchers in the literature. Also, very few works have focused on the combination between graph transformations and game theoretic techniques.

We start with the approaches available for using feature models in electronic negotiation in Section 6.1. In Section 6.2, we present different approaches that are proposed to deal with feature diagrams using graph transformations. Section 6.3 presents the approaches available for using game theory to deal with configuration techniques for feature modelling. Section 6.4 discusses the approaches related to the use of game theory in e-commerce and web services negotiation. In Section 6.5, we discuss related approaches that use a combination of game theory and graph transformation. Section 6.6 concludes the chapter.

6.1 Feature Models in Negotiation

In this section, we discuss the works related to the use of feature models to decrease the complexity of e-commerce and web services negotiation processes.

In [81], the authors proposed a software engineering approach for e-contract enactment. It is based on software product lines and feature modelling, which allows the representation of e-services by features. They developed a contract meta-model based on feature modelling to offer contract templates to optimise the e-contract establishment process. The negotiation is performed according to configuration techniques for feature modelling, in which mandatory features are kept whereas optional and alternative features are chosen according to the negotiation between the involved parties.

Fantinato et al. in [82] proposed a feature-based approach in order to decrease the complexity in the establishment of web service e-contracts. It is similar to the approach presented in [81] but this approach is concerned with the specific web service context besides other new extensions. The e-contract establishment activities, including negotiation, are controlled by the feature model and configuration.

In [83], the same authors extended the two previous works with the new WS-contract metamodel based on WS-BPEL and WS-Agreement. They also emphasised QoS attributes. A prototype FeatureContract toolkit was developed to automatically support the proposed approach. Two feature models are elaborated to represent services and QoS attributes. The QoS attributes give multiple options and levels for negotiation.

These works are related to our work in the sense that we also use feature models to represent e-services and configuration techniques to support the variability of negotiated services. However, their approaches do not discuss how the involved parties should interact, whereas we focus on the design of an appropriate negotiation protocol. Moreover, they do not discuss the definition of the gain and how an acceptable

agreement can be reached, while our approach aims to provide an effective way to reach a mutually acceptable agreement.

In [84], the authors presented an approach based on feature modelling for price definition in the e-contract establishment of web services, extending the approach presented in [83]. During the negotiation, the prices are already associated with the e-services and the QoS levels in the feature models. The consumers are aware of the prices and can negotiate based on them, with no need to query other sources.

Vecchiato et al. in [85] proposed an extension to the work presented in [83] to include control operations to be performed in case of e-contract violation and to support negotiation and renegotiation. The control-operations feature is a sub-tree of a QoS attribute to specify the operations to be executed when the QoS attribute levels are not met.

In [86], the authors discussed the e-contract life cycle from negotiation, establishment and enactment to renegotiation within the context of a feature-based BPM infrastructure. They extended the FeatureContract toolkit presented in [83].

Our work is different in several aspects. As in the previous works, they did not discuss how the negotiation parties can achieve an acceptable agreement. Furthermore, they used QoS attributes and control operations to control the negotiation process, while in our approach we use game theory to analyse the negotiators' strategic interactions.

Silva et al. in [87] proposed an integrated web services negotiation process that considers the human interaction and the use of different protocols. They focused on the application of feature modelling to describe the negotiated services. Their contributions include the definition of the negotiation process and the definition of a conceptual model to support the negotiation of web services. Their negotiation process can support the most common negotiation styles, such as bargain and auction. This work is related to our work in two aspects. First, as in all previous works, feature models have been used to describe the negotiated services. Second, this work

focused on the actual negotiation process, including the role of the negotiators and the negotiation protocols. However, the definition of the negotiation strategy and the negotiation approach, e.g. cooperative, competitive etc., were not discussed in this work, while it is one of our main contributions.

6.2 Feature Models and Graph Transformations

Some approaches have been proposed in different areas to use graph transformation to deal with feature diagrams considering features' relationships.

In [88], the authors presented a first proposal for automated support for feature model refactoring based on graph transformation. They used the Attributed Graph Grammar System (AGG) to implement their approach. They mapped the LHS and RHS of the patterns of the feature model refactoring to the LHS and RHS of the AGG transformation rule. They planned to integrate the automatic support for feature model refactoring into the FAMA plug-in [19].

Segura et al. in [89] proposed using graph transformations to automate the merging of feature models. They proposed a catalogue of 30 rules to merge feature models to be implemented using the AGG system. They extended their previous work presented in [88] by showing their first results. They used a simplified version of the metamodel for attributed feature models presented in [90] as a type graph to implement graph transformation rules.

In [91], the authors proposed a rule-based approach to structural feature model differencing, which is based on a graph representation of feature diagrams. They developed a metamodel for feature diagrams to specify feature diagram edit operations as model transformation rules. They used the model transformation language Henshin [75] for specifying edit operations.

Deckwerth et al. [92] proposed a conflict-detection approach based on symbolic

graph transformation to facilitate concurrent edits on extended feature models. They defined edit operations on extended feature models by means of symbolic graph transformation rules. A metamodel for extended feature models including features' attributes has been developed. They applied their conflict-detection notion presented in [93] to analyse potential conflicts among concurrent edits. The approach is implemented by combining eMoflon with an SMT solver.

These works overlap with ours in the sense that we also use graph transformations to deal with feature models. However, they do not deal with feature model configurations, while we use graph transformations to deal with configurations of feature models in the negotiation context.

6.3 Feature Models and Game Theory

The closest work to ours was proposed by García-Galán et al. in [94], who suggested an interpretation of multi-user configuration as a game theoretic problem. They modelled variability-intensive systems as feature models, and user decisions as feature model configurations. Their approach focused on the conflicts that may arise when different users make decisions on the same configuration concurrently. Thus, they proposed an automated bargaining process, inspired by cooperative game theory, to achieve conflict-free and satisfactory configurations. A set of trade-offs for each user has been introduced to specify alternative decisions in case of conflicts, and the impact over their satisfaction. These trade-offs are defined as a compensation or compromise in the exchange of something. To automate bargaining, they defined an arbitrator who should deal fairly and efficiently with the users. The arbitrator considers a simultaneous and complete information cooperative game of N players.

As in their work, we use feature models to support variabilities of negotiated products and services using feature model configurations. We also use game theoretic techniques to solve the conflicts that may arise in terms of negotiation games. However, their work differs from ours in certain aspects. In our approach, the negotiators make decisions sequentially by making and responding to proposals according to their preferences, and not simultaneously. Moreover, our negotiation takes place between the negotiators directly and does not rely on any third party, such as an arbitrator or a negotiation broker, which is more suitable for web services negotiation in practice.

6.4 Game Theory in Web Services and E-commerce Negotiation

Game-theoretic techniques have been previously applied to web services negotiation to support e-commerce applications. In the following, we provide some of the available approaches that have used game theoretic techniques to support web services and e-commerce negotiation.

Zheng et al. in [95] proposed the use of two-player bargaining games to represent 1-to-1 web services negotiation. They focused on the 1-to-1 web services negotiation between a single service provider and a single service consumer. In the bargaining game, one player makes an offer to the other player, who can accept or refuse it. If it accepts the offer, the game is over. If it refuses the offer, it needs to make a counteroffer. The process repeats until one of them accepts an offer, or no trade occurs before a finite deadline. They introduced two reservation values for the players where a reservation value is a point beyond which a negotiator will walk away. They also determined a Nash equilibrium that can be regarded as a fair solution. This work is related to our work as we also apply game theoretic techniques to web

services negotiation. However, our work is different as, in theirs, bargaining games can only be used to bargain over how to divide the gains by making offers and counter offers, while, in our negotiation, the players exchange proposals to add and withdraw features which can be either accepted or rejected.

In [96], the authors proposed a game theoretic model for negotiations between providers and requestors. They formulated the negotiation as a game theoretic model to analyse their strategic choices in one-time negotiation and repeated negotiation. They assumed that it is a static and complete information game. This work is related to ours as we also formulate the negotiation between a provider and a requestor using game theory. However, in our approach, we use dynamic games while in their approach they used a static game which is not suitable for our scenario where the negotiators interact by exchanging proposals.

Boella et al. in [97] developed a formal game-theoretic model to negotiate a decision between agents about which behaviour to choose. They illustrated how agents use the game theory within contract negotiation. They defined violation games between an agent and the normative system in which the agent predicts the behaviour of the normative system. However, this approach does not address the problem of equilibrium analysis or the negotiation protocol to obtain an agreement, unlike in our approach.

In [98], the authors discussed the theoretical difficulties and opportunities involved in applying game theory and mechanism design to automated agents. They proposed a two-player bargaining model within the ADEPT (Advance Decision Environment for Process Tasks) format. However, as we discussed earlier in this section, bargaining games cannot be used in our case.

Yan et al. in [99] proposed a framework in which the service consumer is represented by a set of agents who negotiate quality of service constraints with the service providers for various services in the composition. A utility-function-based decision-making model is proposed based on which agents can proactively decide

on the course of further actions. This work is related to ours as we also use utility functions to decide the negotiators' outcomes. However, this work focused on the negotiation of the quality of services, while in our approach we focus on the negotiation of the actual services.

In [100], the authors proposed a system named AutONA (Automated One-to-one Negotiation Agent) to automate multiple 1-to-1 negotiation over the price for quantities of a substitutable good subject to the organisation's procurement constraints of target quantity, price ceiling and deadline. The negotiation process was modelled as a round-based multiple 1-1 negotiation game. This approach overlaps with ours in the sense that we also use a negotiation game to model our negotiation process. However, as in [99], this approach focused on the negotiation of quality of services, in particular price and quantities.

Huang et al. in [101] presented a formal model for autonomous agents to negotiate on the internet. The negotiation process is driven by the internal beliefs of participating agents. In every negotiation iteration, an agent checks the history of the process, updates its beliefs about its opponents and then tries to maximise its own expected payoff based on its own subjective beliefs. The players choose their actions simultaneously at each time period. Moreover, the authors conducted a series of experiments to examine the impact of different beliefs on the outcomes of a basic on-line negotiation scenario. Apart from the fact that their approach focused on players' beliefs and the use of iterated games for negotiation, the definition of their formal model is closely related to the negotiation game definition.

In [102], the authors proposed a framework for negotiation processes that provides a consistent model for supporting a comprehensive range of negotiations in a dynamic eBusiness environment. The framework provides the foundation for constructing dynamic negotiation processes including negotiation protocol and negotiation strategy. Once a protocol is selected and agreed, the negotiation becomes a game between the selected negotiation partners where the rules are the negotiation protocol. However,

they did not focus on the type of the game and the strategies but on the message-exchange activities in the negotiation protocol.

Shang et al. in [103] proposed a bilateral business negotiation model. They modelled the negotiation as an incomplete information dynamic game. In the negotiation, there is a bidder and an accepter in turn who can accept, refuse and bargain. If the accepter chooses bargain, the role of the two sides will exchange. Whenever a reaction of accept or refuse is chosen by one of the negotiators, the negotiation process will come to an end. This approach is related to ours because we also use game theory to model the negotiation but their approach cannot be used in our case as they used bidding, which is suitable for competitive but not for cooperative negotiation.

In [104], the authors proposed an automated negotiation mechanism that includes a co-evolutionary mechanism to search complex and large spaces and a degree of satisfaction. It allows the negotiating agents to express different levels of cooperation in the negotiation and the degree of satisfaction without revealing the utility function. However, they did not focus on the type of the game and the definition of negotiation strategies.

Preibusch in [105] examined how service providers may resolve the trade-off between their personalization efforts and users' individual privacy concerns through negotiations. They modelled the negotiation process as a Bayesian game where the service provider faces different types of users. The framework for the negotiation process is a dynamic game where the service provider has high bargaining power. In this approach, the users are of certain types, which affects the players' payoffs. In our approach, we do not consider the player types but their strategies. Moreover, this approach used a bargaining style of negotiation, which is not suitable in our case.

In [106], the authors proposed a game theory model for automatic SLA negotiation between service customer and service provider where a service broker provides optimal value in price and quality to both of the parties. They considered the case

where both service provider and service requestor submit their SLA template to the service broker and in turn the service broker provides them with the optimal negotiated value for their SLA. The negotiation is represented as a static game in which the players make their choices simultaneously. In our approach, we use dynamic games to represent our negotiation while static games are not suitable for our scenario. Furthermore, we focus on a direct negotiation between the provider and requestor and do not rely on a negotiation broker.

Table 6.1 presents a summary comparison of the approaches reviewed in this section including our approach. In the (Game Type) column, we present the type of the game, which can be either static or dynamic. The (Information) column specifies whether the game is of complete or incomplete information. In the (Negotiation Strategies) column, the 'check' mark indicates that the approach focused on the definition of the negotiation strategies while negotiating. In the (Nash Equilibrium) column, the 'check' mark indicates that the Nash equilibrium has been defined to determine the optimal solution to the game.

6.5 Game Theory and Graph Transformations

In this section, we discuss the works related to the combination of game theory and graph transformations.

Hindriks in [107] proposed generating game strategies using graph transformations and is to our knowledge the closest work to ours in terms of the implementation. He presented a system that allows the Minimax algorithm to be applied to the game. The output of the algorithm is a strategy that provides a choice between possible moves for any state of the game. The states of the games were modelled as graphs and the moves as graph transformation rules. He used the GROOVE simulator tool [108] to explore the game state space and calculate the Minimax value of all states.

Approach	Game Type	Information	Negotiation Strategies	Nash Equilibrium
Our Approach	Dynamic/Negotiation	Complete	✓	✓
Zheng et al. [95]	Dynamic/Bargaining	Complete/Incomplete	✓	✓
Sun et al. [96]	Static	Complete	✓	✓
Boella et al. [97]	Dynamic/Negotiation	-	×	×
Binmore et al. [98]	Dynamic/Bargaining	Incomplete	×	×
Yan et al. [99]	Dynamic/Bargaining	-	✓	×
Byde et al. [100]	Dynamic/Bargaining	-	✓	×
Huang et al. [101]	Dynamic/Bargaining	Incomplete	✓	×
Kim et al. [102]	Dynamic/Negotiation	-	×	×
Shang et al. [103]	Dynamic/Bargaining	Incomplete	✓	×
Chao et al. [104]	Dynamic/Negotiation	Incomplete	×	×
Preibusch [105]	Dynamic/Bargaining	Incomplete	✓	×
Ray et al. [106]	Static	-	✓	✓

TABLE 6.1: Summary Comparison of the Reviewed Approaches

A heuristic function was used to calculate the heuristic value of a given game state. The approach was evaluated by generating strategies for a range of games.

The main difference between our approach and this approach is that this approach focused on analysing zero-sum games using the Minimax algorithm, in which if one player gains the other loses, while in our approach we explore both zero-sum and non-zero-sum games for negotiation.

In [109], the authors proposed a generative model that combines graph transformations and game theory. They represented a complex network as a sequence of

node-based transformations determined by the interactions of nodes present in the network. They used graph transformation to model the node-based transformation while game theory is used to abstract the interaction between nodes. They proposed a model called the dynamic spatial game and applied it to two biological examples. They considered two-player symmetric games with two strategies, cooperate and defect. Each node has an associated label that denotes its strategy and obtained value according to its strategy. Each node executes an action according to the obtained value. The action consists of the replacement of the node by means of an appropriate production. This work is related to ours in the sense that we also use game theory to analyse the interactions between states in state-based transformations. However, in our approach, we use graph transformations to model dynamic games whose states are modelled as graphs and the moves of the players as graph transformation rules, while in this approach they modelled a symmetric game where its nodes represent the strategies of the game and the actions are used to replace the nodes according to a given production. Thus, it differs from ours in the type of game and the definitions of the game states and moves.

In [110], the author proposed two-player zero-sum structure rewriting games in the course of which a structure is manipulated by the players using rewriting rules. The author provided a formal definition including the definition of the winning strategy. Łukasz et al. in [111] introduced a general games model in which states are represented by relational structures and actions by structure rewriting rules. They developed an algorithm that computes rational strategies for the players. They used an evaluation game which is a statistical model used by the players to assess the state after each move and to choose the next action. Our concern is with analysing the negotiators' strategic choices in order to determine a solution that benefits both of them, beyond zero-sum games.

6.6 Summary

In this chapter, we reviewed the literature by providing a compilation of the most relevant works in the areas of feature modelling, game theory and graph transformations. We classified these works into five categories: Feature Models in Negotiation, Feature Models and Graph Transformations, Feature Models and Game Theory, Game Theory in Web Services and E-commerce Negotiation and Game Theory and Graph Transformations. We also discussed how our work is related to these works and how it differs from them.

Chapter 7

Conclusion and Future Work

This chapter presents the conclusions to the thesis and suggestions for future research work. Section 7.1 presents a summary of the research and its outcomes. In Section 7.2, we discuss the main contributions of the thesis. Section 7.3 discusses the conclusions of the conducted research. Finally, Section 7.4 lists suggestions for further research directions.

7.1 Overall Summary

The work in this thesis highlights the important role that negotiation plays in the success of e-commerce applications and services. It argues that the flexibility of a provider and a requestor to negotiate is needed to discuss preferences and constraints in order to determine a solution that benefits both of them. In order to achieve this, feature models were used to increase the flexibility of agents' interactions. Negotiation games were proposed to model the interaction between a provider and a requestor who use feature models to represent service configurations. They were implemented as graph transformation games in which the states of the game are

given by graphs and the moves of the players by graph transformation rules. Such a graph consists of a feature diagram, the current configuration under discussion and a negotiation state. The graph transformation games were analysed to find optimal strategies for the negotiators. Different experiments were conducted in order to evaluate the approach.

7.2 Contributions

In this thesis, we proposed negotiation games, implemented as graph transformation games. We categorise the contributions of the thesis into:

1. Graph Transformation Games.
2. Implementing Graph Transformation Games:
    2.1. Defining game metamodel.
    2.2. Designing game rules.
3. Analysing Graph Transformation Games:
    3.1. Extensive-form graph transformation games.
    3.2. Two-player turn-based stochastic games.

In the following subsections, we provide a summary of the above-mentioned contributions.

7.2.1 Graph Transformation Games

We introduced graph transformation games that combine both graph transformation and game theoretical concepts to implement and analyse our negotiation games.

Graph transformation was used to model the negotiation games as state-based transformations while game theory was used to analyse the interactions between states. The states of the games are given by graphs and the rules of the games are defined by graph transformation rules.

7.2.2 Implementing Graph Transformation Games

The implementation of our graph transformation games passed through two main steps: defining the game metamodel and designing game rules.

Defining the Game Metamodel: a metamodel was developed to define the negotiation entities. It was used as a type graph to define graph transformation rules. It contains three representations. First, the representation of the feature model, which defines the relationships of features in the feature diagram. Second, the configuration representation to represent the selected features. Third, the negotiation state representation to negotiators and their proposals. Moreover, the metamodel defines classes to represent the behaviour of the negotiators in which they interact by taking turns.

Designing Game Rules: based on the game metamodel, the moves of the players were defined as graph transformation rules. The players can make proposals and respond to them by accepting or rejecting. The proposals were designed depending on the types of features in the feature model, such as adding optional and substituting alternative features. The rules were created using the Henshin transformation language and tool environment. We also used Henshin state space tools to generate the game state space. We conducted different experiments to measure the speed in generating the state space and test the scalability.

7.2.3 Analysing Graph Transformation Games

We proposed two different analyses of our graph transformation games, one in which we analysed them as extensive-form games and the other in which we analysed them as two-player turn-based stochastic games

Extensive-form Graph Transformation Games: we proposed to analyse our graph transformation games as extensive-form games. The backward induction technique was adopted to reason our game state space backward in order to determine the optimal moves of the players and therefore a Nash equilibrium of the game. The result obtained by backward induction is the subgame perfect equilibrium, which represents a Nash equilibrium of every subgame. The analysis was evaluated by conducting different experiments on different graph transformation games to test the scalability of the analysis.

Two-player Turn-based Stochastic Games: due to the uncertainty arising from the unpredictability of negotiators' reaction to proposals, we proposed analysing our graph transformation games as two-player turn-based stochastic games using the PRISM-games model checker. Firstly, we generated our graph transformation games from Henshin to PRISM-games format by modifying the Henshin source code. Then, we analysed the generated games by defining different reward-based properties in order to generate optimal strategies for the players. We defined single-objective properties to explore best individual outcomes. We also defined multi-objective properties to achieve a more collaborative negotiation. Single-objective properties are fixed and can be used in any example whereas multi-objective properties change according to each example. Different experiments were conducted to measure the construction time of the model in PRISM-games and the time spent in generating the strategies.

Discussion: we applied both analysis methods to the same domain of feature negotiation games in order to discover different analysis results. These methods were presented as alternative to each other in analysing graph transformation games. However, they have different requirements and provide different results. Thus, in the following, we provide a brief comparison between these analysis methods, looking in particular at the types of negotiation, the suitability and the scalability of each method.

- Types of negotiation: each analysis method presents a different type of negotiation. In extensive-form graph transformation games, the negotiators do not behave competitively and they do not intend to minimise each other's payoffs. However, each negotiator will try to maximise its individual outcome regardless of what the other negotiator gets. Thus, the result of this analysis is not necessarily the best joint outcome although it is better than behaving competitively. In stochastic graph transformation games, we explored two different types of negotiation, competitive and cooperative negotiation. In competitive negotiation, the negotiators have completely opposite interests and try to minimise each other's gain. The results of playing competitively show that there is only a very small chance of achieving more than what was present before starting the negotiation due to the competitive nature of the game and the probability distribution over transitions. When cooperating, the negotiators have joint strategy in order to achieve a particular goal. In our negotiation games, the results of collaborating are better than playing competitively.

- Suitability: each analysis method is suitable for different scenario. Some real-life negotiations can be described by extensive-form games with perfect information, where the negotiators make choices sequentially and each negotiator is perfectly informed of all previous actions. Analysing extensive-form games with backward induction is suitable for scenarios, where each negotiator wants

to maximise its own individual outcome without any concern for what the other negotiator gets. However, in many real-life negotiations, negotiators are often concerned about reaching a higher joint outcome that is acceptable by both parties rather than maximising their own individual outcomes. Moreover, this analysis requires sequential rationality, where players must play optimally at every point in the game. This assumption may not be possible in real-life scenarios.

When uncertainty exists in the negotiation situations, stochastic games is suitable to analyse such situations. In some scenarios, the negotiators can behave erratically and may have different levels of rationality that cause unpredictable reaction to proposals. Thus, with stochastic games, the uncertainty is represented by assigning probabilities to negotiation actions. Analysing stochastic games using PRISM-games allows synthesising optimal strategies for both competitive and collaborative negotiations. However, it is very difficult in real-life negotiation to obtain accurate probabilities for negotiation actions. This may cause the negotiation to fail as it has high impact on the obtained results.

- Scalability: as we discussed in Chapter 4 and Chapter 5 both analysis methods have scalability issues with large graph transformation games. However, synthesising strategies with PRISM-games is more scalable than computing strategies in extensive-form games. Computing strategies in large extensive-form games using backward induction demands an extraordinary amount of computer memory and in some large examples does not return any results.

7.3 Conclusion

The aim of this thesis is to provide a flexible and structured negotiation process that enables the negotiators to discuss their preferences and interact in a strategic

way to reach an agreement that benefits both of them. It has proposed graph transformation games that combine game theoretic and graph transformation concepts. The approach shows the usefulness of using game theory to provide a solution to the negotiation by finding the optimal decision-making strategies. Graph transformation was used to provide a formal specification technique, which supports the visual representation of the game.

We are interested in exploiting game theory as a software engineering tool, to analyse requirements and design autonomous systems that interact in a flexible and cooperative way to maximise non-functional requirements expressed by payoffs. Synthesising a strategy for such a system corresponds to the step from requirements to design, while analysing a game can be a means by which to understand and validate requirements.

7.4 Future Work

This research revealed some questions that need to be investigated in further studies. In this section, we will highlight our future work, including further evaluation in Section 7.4.1, investigating scalability in Section 7.4.2, further analysis techniques in Section 7.4.3 and a compiler approach in Section 7.4.4.

7.4.1 Further Evaluation

The implementation of our graph transformation games was evaluated by experiments to apply our transformation rules to different feature models with similar structure, i.e. feature models with root and one level or two levels of sub-features. Those experiments may not be sufficiently representative. Therefore, part of the

future work needs to apply the proposed approach to a wider range of different feature models with different structures. Moreover, it would be interesting to show the effect of feature models' structures on the size of the state space. For example, in Figure 7.1, we show two feature model examples with a similar number of features but in different structures. Table 7.1 shows the generation results for these two examples. The number of states decreased by about 69% between the first and the second example. The reason is that feature **C** in the second example cannot be proposed to be added unless its parent **B** has been added to the configuration, while in the first example both features **B** and **C** can be proposed at the same time.

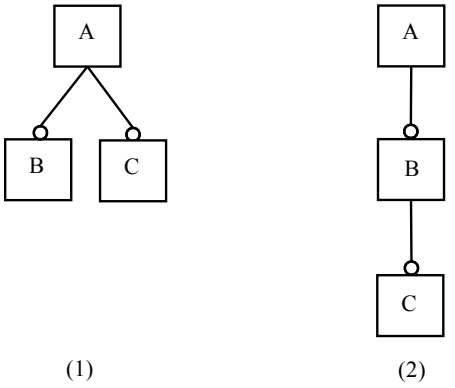


FIGURE 7.1: Feature Models with Different Structures

Feature model	No of states	No of transitions	Generation time (Seconds)
(1)	71	70	0.684
(2)	22	21	0.587

TABLE 7.1: Generation Results for Feature Models with Different Structures

7.4.2 Scalable Protocol

This thesis pointed out a scalability issue with both implementing and analysing our graph transformation games, caused by the dramatic increase in the size of the games' state space, which is affected by the number of features in the features

model. In future work, the top priority is to design different negotiation protocols which might restrict the moves of the negotiators, yielding a smaller game size. For example, in real business negotiations, the duration of the negotiation is very important as most of the negotiation processes are very time consuming. Furthermore, the negotiators may have high sensitivity to time, which means that they value their time and do not want to be involved in unnecessarily complex and time-consuming negotiations. In our negotiation games, if the negotiators have hundreds of available proposals, for example, it is not feasible to allow them to exchange all proposals because the time spent negotiating should be reasonable. Thus, the duration of the negotiation could be limited by a certain deadline.

7.4.3 Incomplete Information

This research presented two types of games that were used to analyse our graph transformation games in Chapter 4 and Chapter 5. Further studies need to be carried out to discover different types of games and strategies that might provide better analysis results. Moreover, in both types of games we used, we assumed that the game is of perfect information. This assumption is not always possible in reality. Therefore, a focus on games of incomplete information could provide more realistic results. The longer-term aim is to allow the use of this method to design and optimise negotiation strategies as part of the development or customisation of e-commerce applications or in the interaction between service requestors and providers in a business-to-business context.

7.4.4 Compiler Approach

As discussed in Chapter 5, we generated graph transformation games from Henshin to PRISM-games format for analysis as two-player turn-based stochastic games.

According to our experimental results, constructing the models in PRISM-games and generating strategies took up most of our time, especially with larger models. The reason is that we constructed a flat labelled transition system in PRISM-games rather than its specification. Therefore, implementing our negotiation of features using the PRISM-games model checker directly rather than in Henshin could produce interesting results. We carried out an initial implementation and conducted some experiments to provide preliminary results.
In PRISM-games, we model our negotiation of features as follows:

Players: similar to what we have carried out in our approach, we created two constructs for players: one for the requestor and the other for the provider. Inside each construct, we defined the transitions under the players' control. The transitions for the requestor start with the prefix "Req" and the transitions for the provider start with the prefix "Prov". The only difference is that we created a transition for each specific feature rather than its type. For example, to define a transition for the provider to accept adding the optional feature Catering, we created a transition with the following label: [Prov_accept_to_add_Catering].

Module: we created one module to define the behaviour of the negotiators, in which they take turns. The state of the module is defined by a set of variables.

Variables: we defined a set of variables as following:

1. We introduced a variable *turn* to schedule taking turns.
2. We defined an integer variable for each feature where its value represents a feature's status as follows:
   (a) 0 indicates that the feature has not been selected in the configuration.

(b) 1 indicates that the feature has been selected in the configuration.

(c) 2 indicates that the feature has been proposed. This value is needed to ensure not to make the same proposal again.

3. We defined two variables to control pass transitions, so if both players pass turn, the game stops.

Rewards: in this implementation, we created two reward structures: "req" and "prov". We assigned rewards to transitions to indicate accepting adding and withdrawing features. For example, we assign a reward after accepting adding Catering as follows:

[Prov_accept_to_add_Catering] turn=1: 5;

Experimental Results: we conducted some experiments to measure the model construction time and strategy generation time. In all experiments, we used feature models with one level of optional sub-features. We also increased the memory allocated to Java to 10GB in order to avoid lack of memory error. In Table 7.2, we show the results generated using single-objective strategies. We used examples containing three to 10 optional features. In Table 7.3, we show the results generated using similar examples but with multi-objective strategies.

No of features	No of states	No of transitions	Construction time in PRISM-games (Seconds)	Strategy generation time (Seconds)
3	293	521	0.024	0.002
4	1059	2067	0.028	0.008
5	3657	7597	0.037	0.039
6	12299	26723	0.082	0.181
7	40701	91521	0.244	0.694
8	133251	307939	0.868	3.221
9	432833	1022981	4.783	13.753
10	1397259	3365139	16.112	35.548

TABLE 7.2: The Generation Results with Single-objective Strategies

No of features	No of states	No of transitions	Construction time in PRISM-games (Seconds)	Strategy generation time (Seconds)
3	293	521	0.024	0.081
4	1059	2067	0.028	0.267
5	3657	7597	0.037	0.926
6	12299	26723	0.082	3.363
7	40701	91521	0.244	11.454
8	133251	307939	0.868	36.464
9	432833	1022981	4.783	128.304
10	1397259	3365139	16.112	424.068

TABLE 7.3: The Generation Results with Multi-objective Strategies

These results show a significant improvement in both construction and strategy generation time compared to our approach. For example, in the third row in Table 7.2, only 0.037 seconds are required to construct the model with five features and 0.039 seconds to generate the strategy, whereas in the last row in Table 5.3, 7088.807 seconds are required to construct a model with the same number of features and 2462.566 seconds to generate the strategy. However, one of the limitations of this implementation is that, in order to keep the consistency of the configuration with its feature model's constraints, it is necessary to control this manually by using the feature's status variables described above.

As discussed above, generating our negotiation of features using the PRISM-games model checker directly could yield a significant improvement in the generation and

analysis time. Therefore, we are very interested in providing an alternative approach as part of our future work.

7.5 Summary

In this chapter, we covered the contributions presented in the thesis. We provided three main contributions: graph transformation games, implementing graph transformation games and analysing them. Negotiation games are implemented as graph transformation games where the states of the games are given by graphs and the rules of the games are defined by graph transformation rules. We discussed the steps in implementing our graph transformation games, which include defining the game metamodel and designing game rules. We also discussed the two different analyses of our graph transformation games. Both the implementation and the analysis were evaluated by different experiments. Then, we provided an overview of our future work and possible open research directions.

Appendix A

Transformation Rules

A.1 Rules for Alternating-offer Protocol

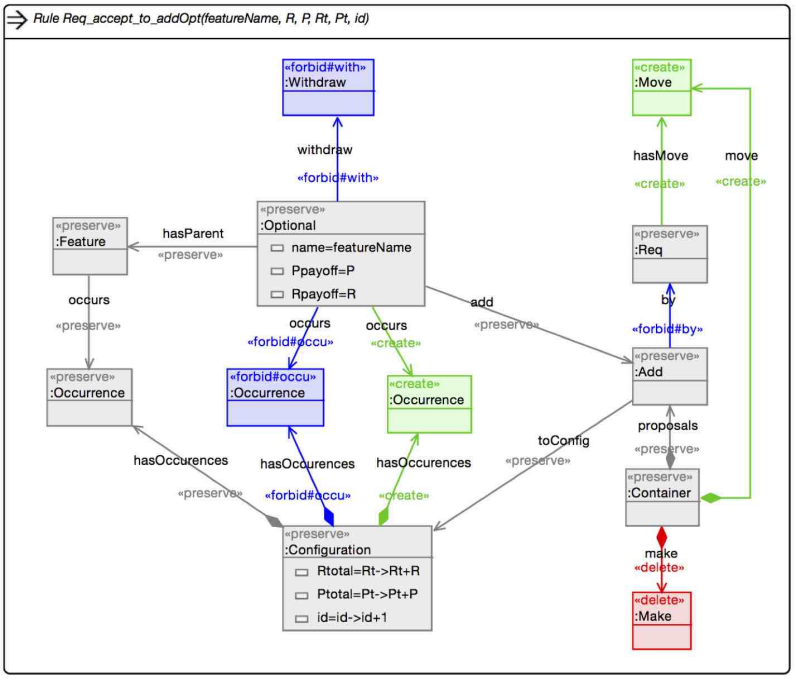


FIGURE A.1: Transformation Rule *Req_accept_to_addOpt*

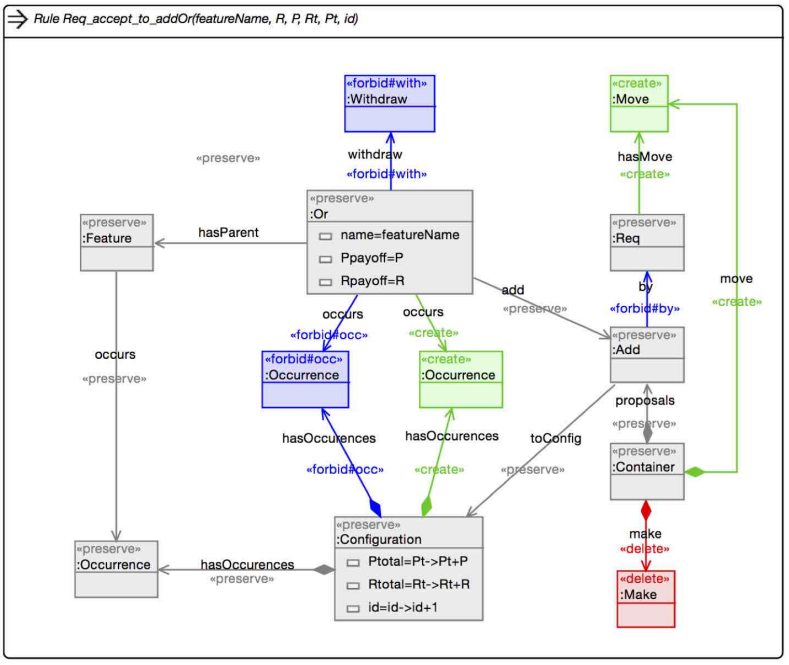


FIGURE A.2: Transformation Rule *Req_accept_to_addOr*

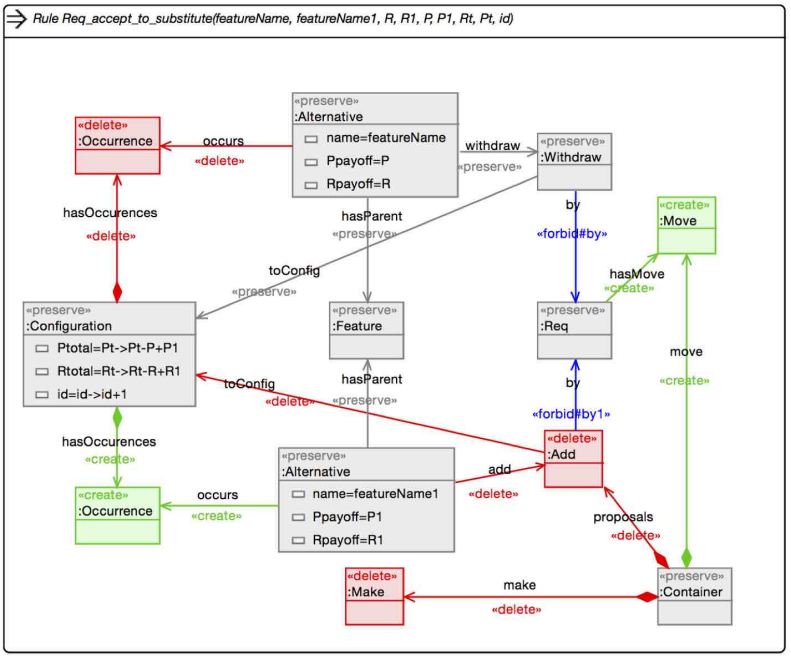


FIGURE A.3: Transformation Rule *Req_accept_to_substitute*

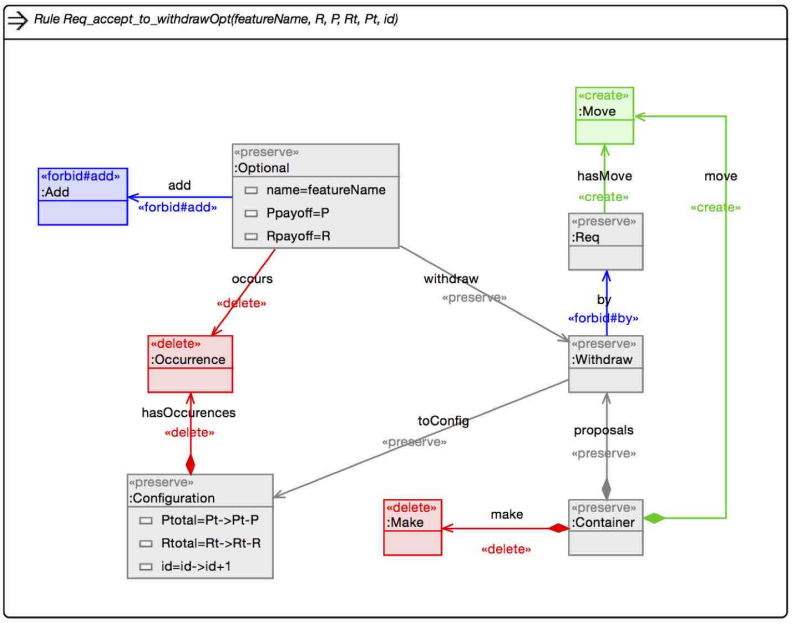


FIGURE A.4: Transformation Rule *Req_accept_to_withdrawOpt*

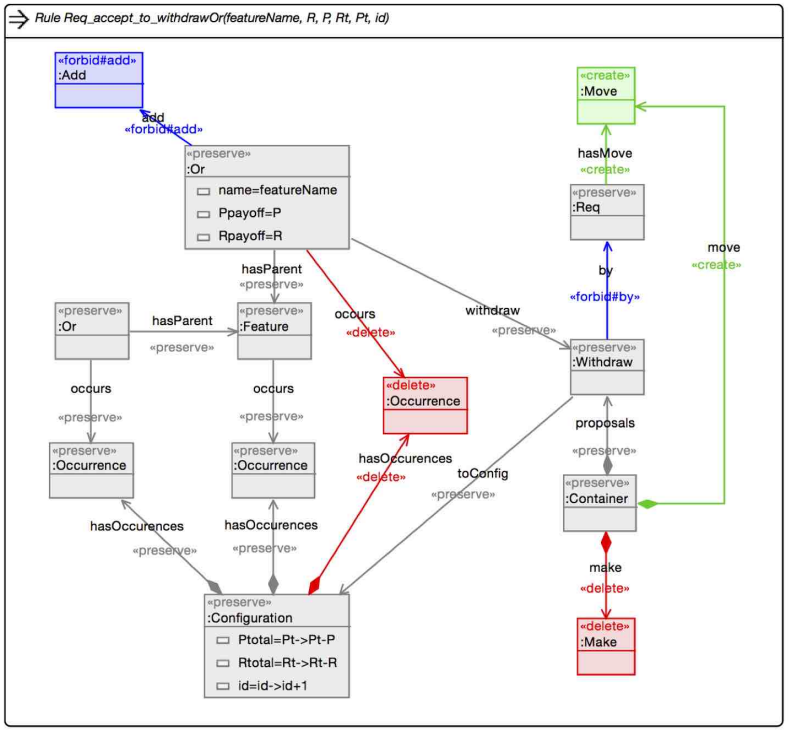


FIGURE A.5: Transformation Rule *Req_accept_to_withdrawOr*

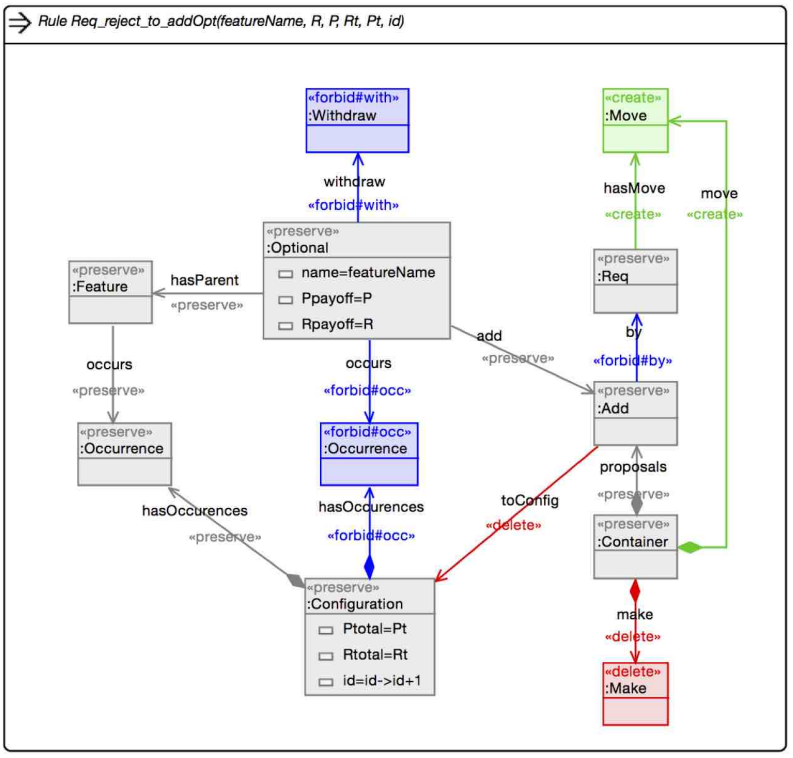


FIGURE A.6: Transformation Rule *Req_reject_to_addOpt*

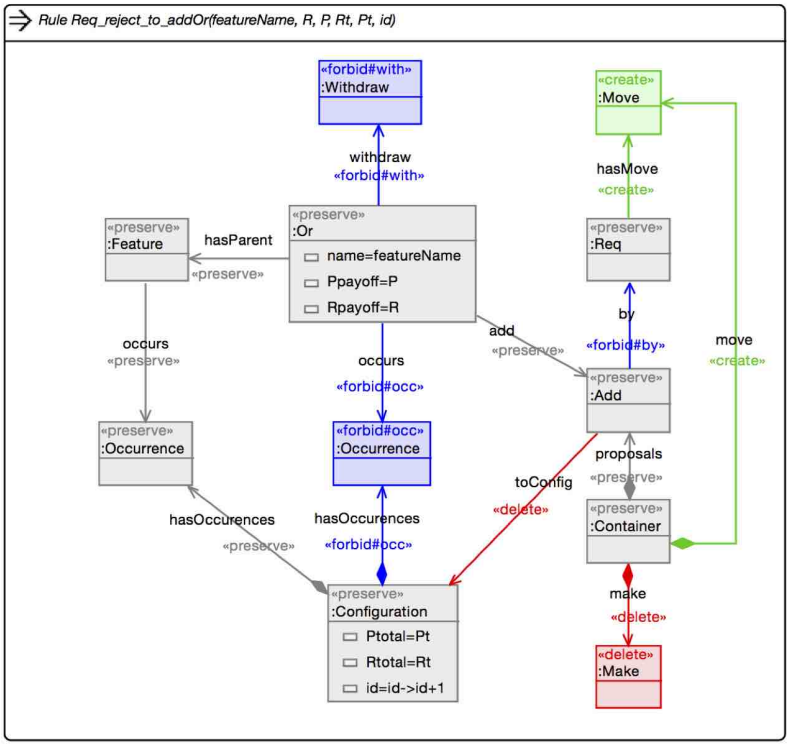


FIGURE A.7: Transformation Rule *Req_reject_to_addOr*

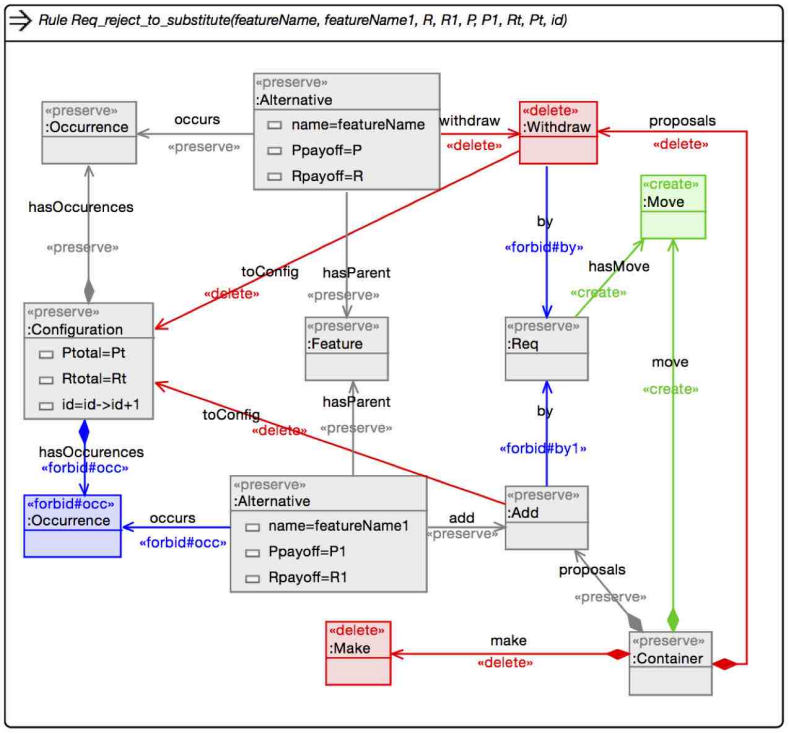


FIGURE A.8: Transformation Rule *Req_reject_to_substitute*

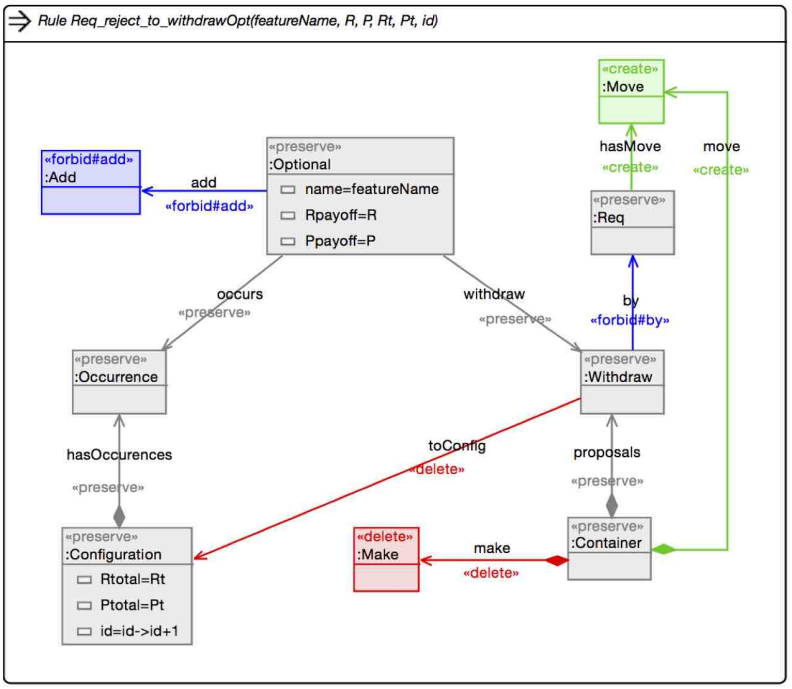


FIGURE A.9: Transformation Rule *Req_reject_to_withdrawOpt*

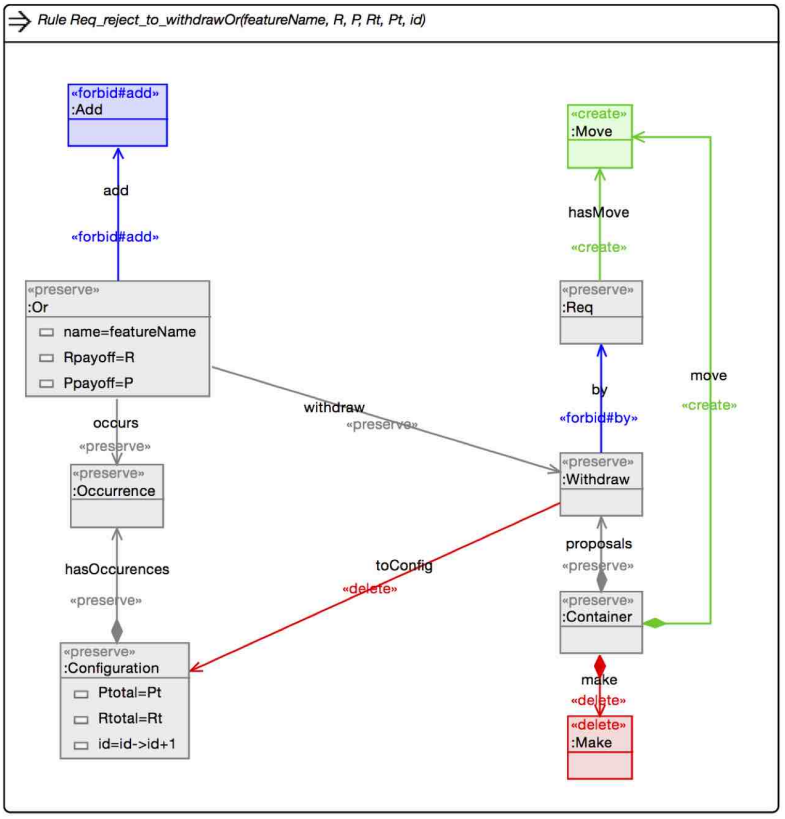


FIGURE A.10: Transformation Rule *Req_reject_to_withdrawOr*

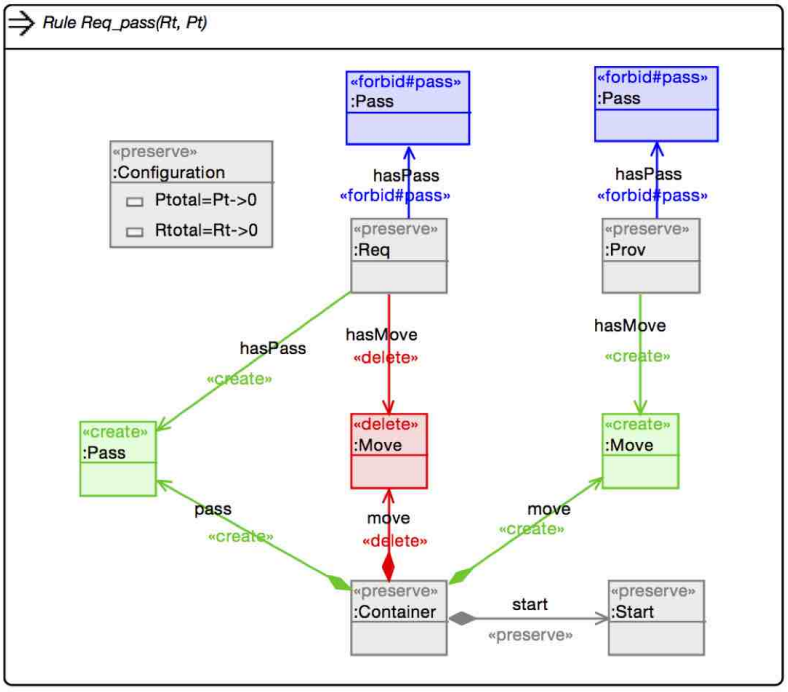


FIGURE A.11: Transformation Rule *Req_Pass*

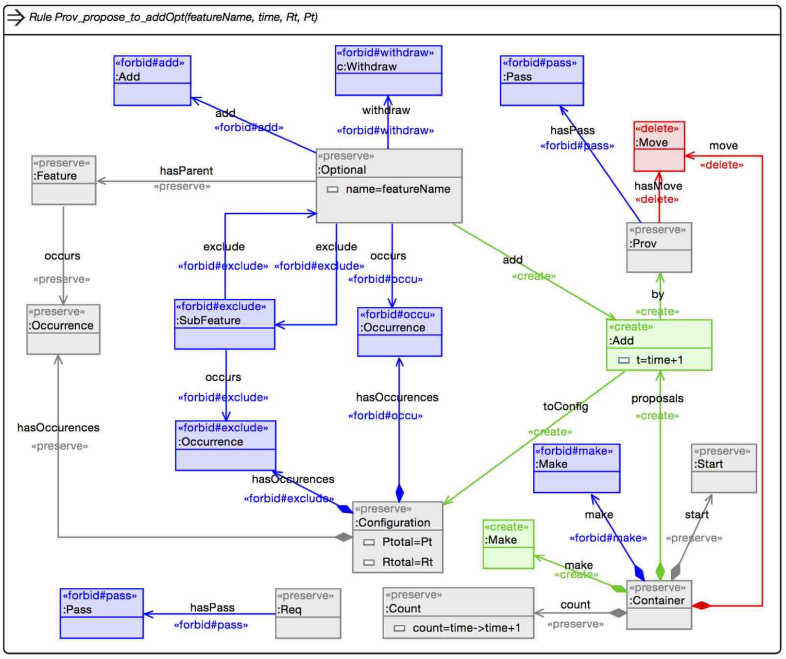


FIGURE A.12: Transformation Rule *Prov_propose_to_addOpt*

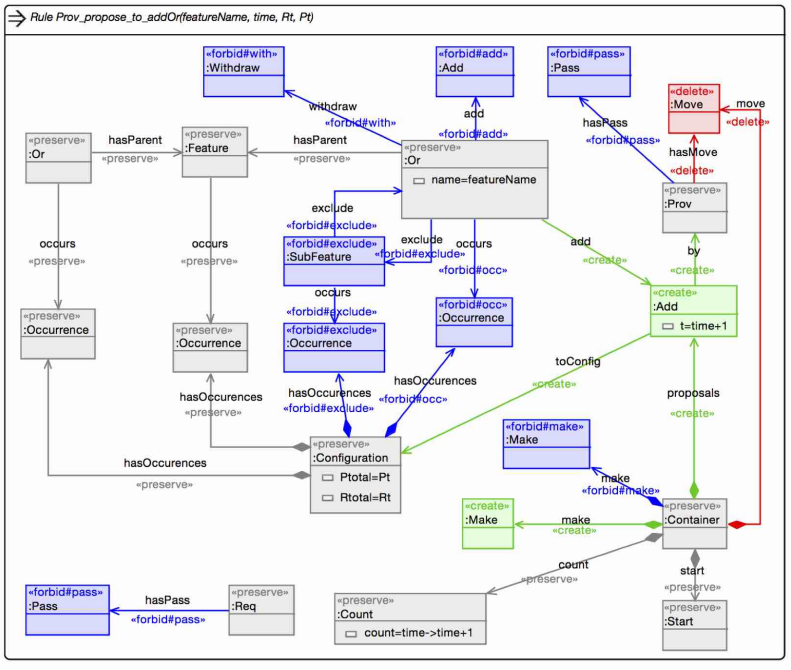


FIGURE A.13: Transformation Rule *prov_propose_to_addOr*

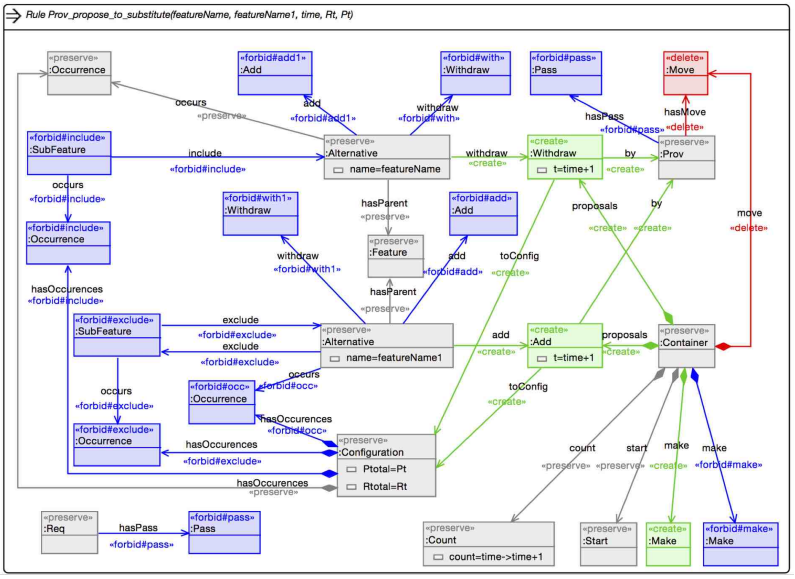


FIGURE A.14: Transformation Rule *Prov_propose_to_substitute*

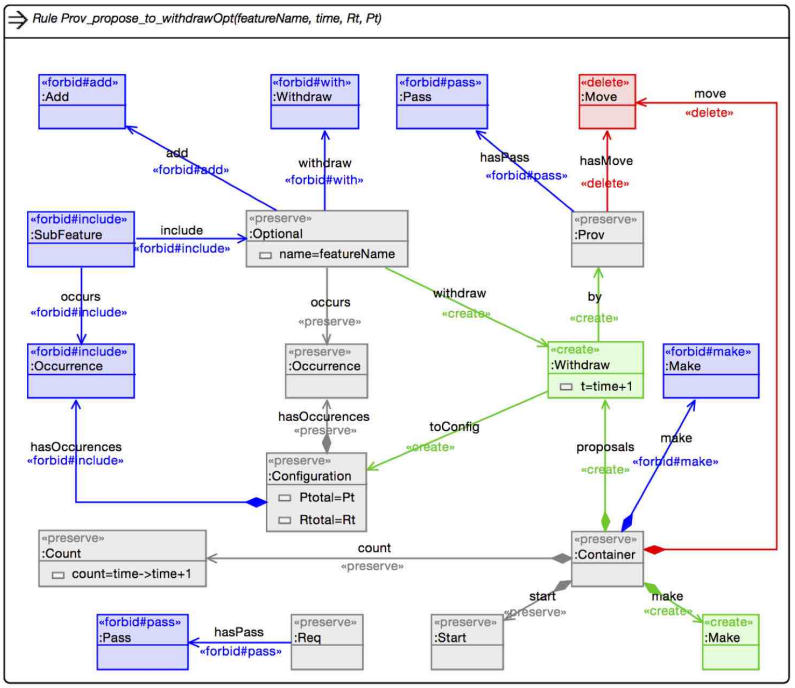


FIGURE A.15: Transformation Rule *Prov_propose_to_withdrawOpt*

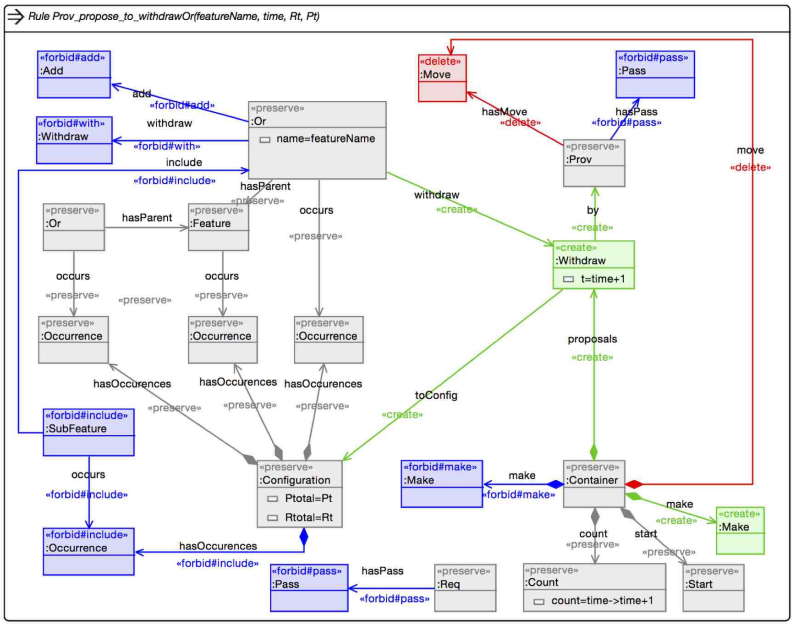


FIGURE A.16: Transformation Rule *Prov_propose_to_withdrawOr*

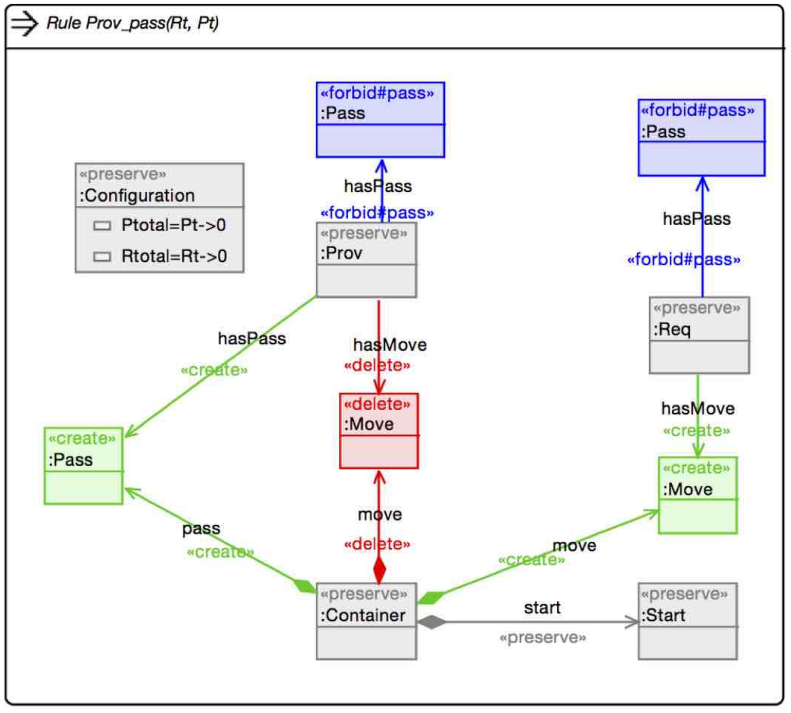


FIGURE A.17: Transformation Rule *Prov_Pass*

A.2 Backward Induction Rules

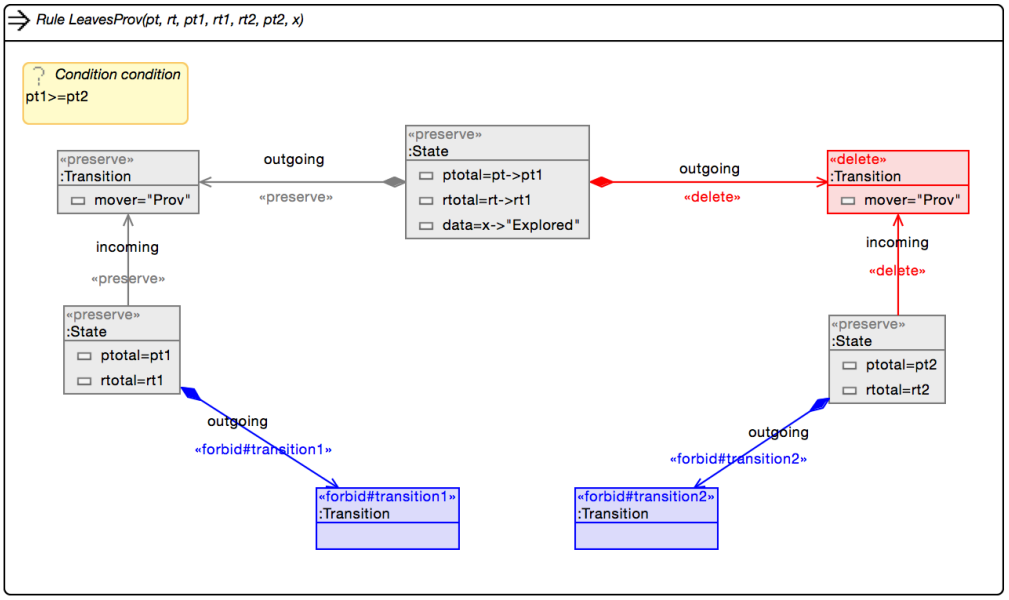


FIGURE A.18: Transformation Rule *LeavesProv*

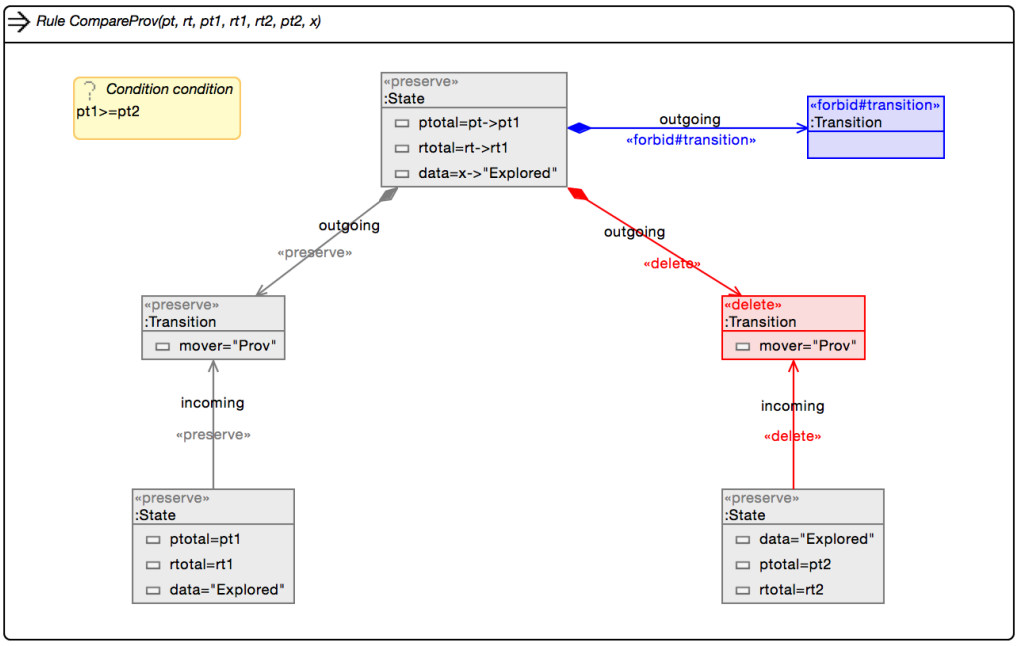


FIGURE A.19: Transformation Rule *CompareProv*

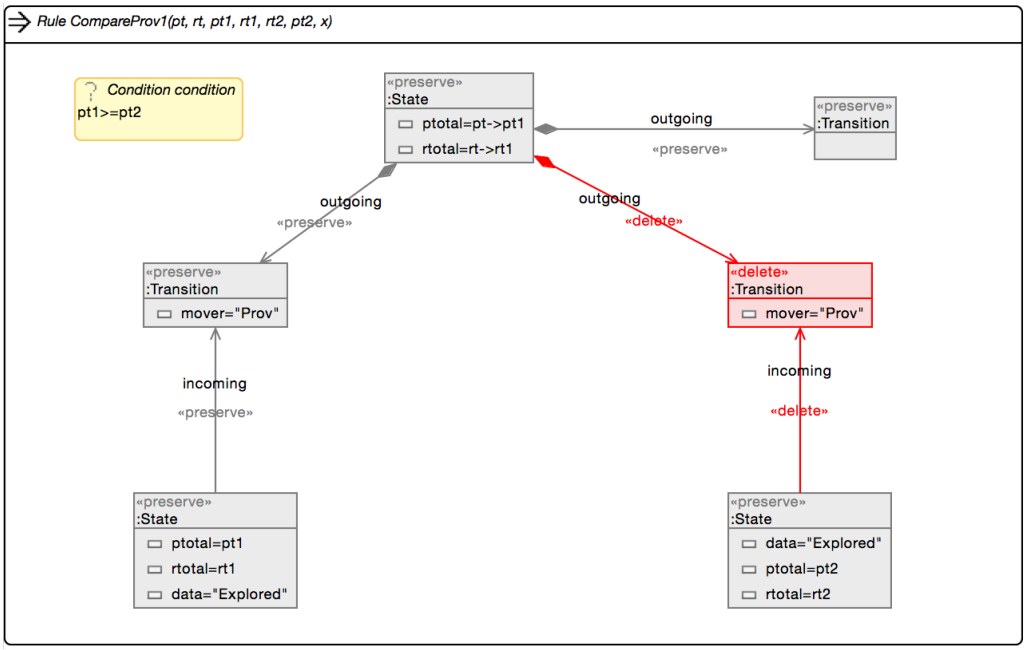


FIGURE A.20: Transformation Rule *CompareProv1*

Appendix B

Implementation

B.1 Generating Graph Transformation Games

```
/**
 This code has been taken from the code provided with Henshin examples and
    written by Christian Krause
 */
public class GenerateStateSpace {
public static final String PATH = "src/";
   public static void run(String path){
      System.out.println("Generating state spaces...");
      System.out.println("MaxMemory: " + Runtime.getRuntime().maxMemory()
   / (1024 * 1024) + "MB\n");
      // Load the state space and create a state space manager:
      StateSpaceResourceSet resourceSet = new StateSpaceResourceSet(path);
      StateSpace stateSpace =
   resourceSet.getStateSpace("Test.henshin_statespace");
```

```
      StateSpaceManager manager = new
   ParallelStateSpaceManager(stateSpace);
      try {
         System.out.println("States\tTrans\tGenTime");
            // First reset the state space:
            manager.resetStateSpace(false);
            // Then explore it again:
            long genTime = System.currentTimeMillis();
            new StateSpaceExplorationHelper(manager).doExploration(-1, new
   NullProgressMonitor());
            genTime = (System.currentTimeMillis() - genTime);
            System.out.println(stateSpace.getStateCount() + "\t" +
                        stateSpace.getTransitionCount() + "\t" +
                        genTime);
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      finally {
         manager.shutdown();
      }
      System.out.println();
   }
   public static void main(String[] args) {
      run(PATH);
   }
}
```

B.2 Extensive-form Graph Transformation Games

B.2.1 Creating A State Space Metamodel and Instance

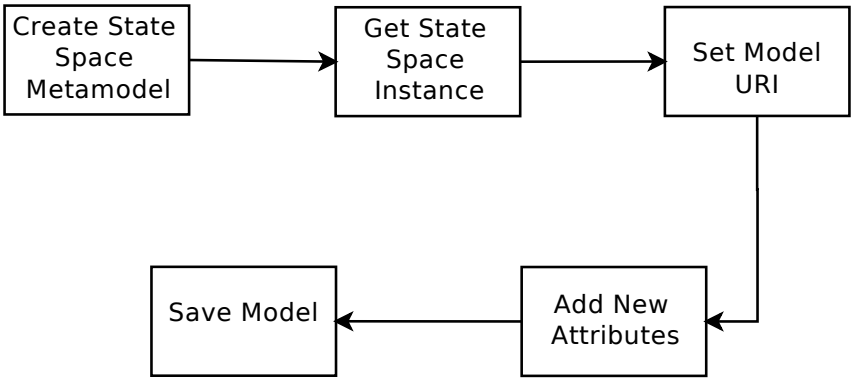


FIGURE B.1: Creating A State Space Model and Instance

```
// create a model of the state space
      Model stateSpaceModel =
   StateSpaceFactory.eINSTANCE.createModel(stateSpace.eResource());
      // get the graph of the state space
       EGraph stateSpaceGraph = stateSpaceModel.getEGraph();
       // get the root object of the state space graph
      EObject stateSpaceRoot = stateSpaceGraph.getRoots().get(0);
      //get the EPackage of the model
       EPackage stateSpacePackage = stateSpaceRoot.eClass().getEPackage();
       //set the URI of the model package
       stateSpacePackage.setNsURI("www.statespace.com");
       // set the URI of the state space instance to be similar to the
   model


   stateSpaceRoot.eResource().setURI(URI.createURI("www.statespace.com"));
       // get Storage class
```

```
    EClass storageClass = (EClass) stateSpacePackage.eContents().get(5);
    // get data attribute of the Storage Class and set its type to
    String
storageClass.getEAttributes().get(0).setEType(EcoreFactory.eINSTANCE.
    getEcorePackage().getEString());
    // create ptotal attribute
    EAttribute ptotal = EcoreFactory.eINSTANCE.createEAttribute();
    ptotal.setName("ptotal");
    ptotal.setChangeable(true);
 ptotal.setEType(EcoreFactory.eINSTANCE.getEcorePackage().getEInt());
    // create rtotal attribute
    EAttribute rtotal = EcoreFactory.eINSTANCE.createEAttribute();
    rtotal.setName("rtotal");
    rtotal.setChangeable(true);
rtotal.setEType(EcoreFactory.eINSTANCE.getEcorePackage().getEInt());
    //get State Class
    EClass stateClass = (EClass) stateSpacePackage.eContents().get(1);
    //add ptotal attribute to State class
    stateClass.getEStructuralFeatures().add(ptotal);
    //add rtotal attribute to State class
    stateClass.getEStructuralFeatures().add(rtotal);
    // create mover attribute
    EAttribute mover = EcoreFactory.eINSTANCE.createEAttribute();
    mover.setName("mover");
    mover.setChangeable(true);
mover.setEType(EcoreFactory.eINSTANCE.getEcorePackage().getEString());
    // create transitionLabel attribute
    EAttribute transitionLabel =
EcoreFactory.eINSTANCE.createEAttribute();
```

```
        transitionLabel.setName("transitionLabel");
        transitionLabel.setChangeable(true);
    transitionLabel.setEType(EcoreFactory.eINSTANCE.getEcorePackage().
        getEString());

        //get State Class
        EClass transitionClass = (EClass)
    stateSpacePackage.eContents().get(3);
        transitionClass.getEStructuralFeatures().add(transitionLabel);
        transitionClass.getEStructuralFeatures().add(mover);

        // save .core file after changing the type of data attribute
        resourceSet.saveEObject(stateSpacePackage, "/Users/Mohammed");
```

B.2.2 Generating An Instance

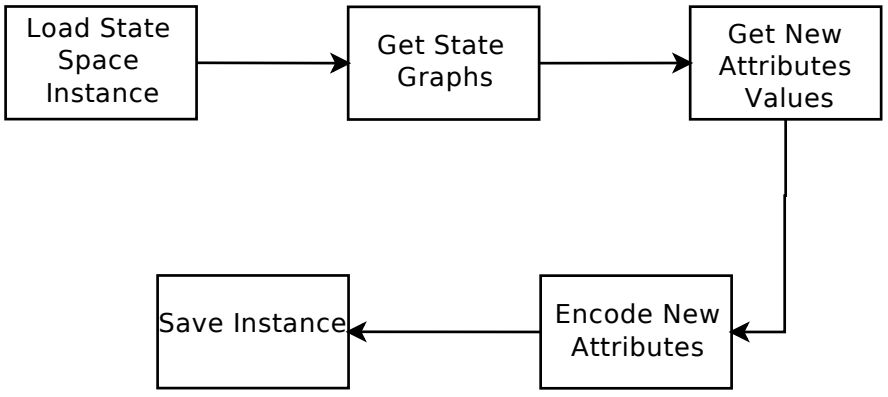


FIGURE B.2: Generating A State Space Instance

```
//load the EGraph of the StateSpace.xmi instance created by Create.java
      EGraph graph = new
    EGraphImpl(resourceSet.getResource("StateSpace.xmi"));
      //get the root of the instance
```

```
    EObject instanceRoot = graph.getRoots().get(0);
    //get all states objects from StateSpace.xmi
       EObject stateInstance = instanceRoot.eContents().get(i);
       //get the EGraph of each state in the stateSpace
Test.henshin_statespace
       EGraph stateGraph =
manager.getModel(stateSpace.getStates().get(i)).getEGraph();
       //get the root object of the stateGraph
       EObject graphRoot = stateGraph.getRoots().get(0);
       //get the featuremodel object
       EObject featureModel = graphRoot.eContents().get(0);
       //get the configuration object
       EObject config = featureModel.eContents().get(0);
       //get Ptotal value from the configuration
       int pTotal = (int)
config.eGet(config.eClass().getEStructuralFeature("Ptotal"));
       //get Rtotal value from the configuration
       int rTotal = (int)
config.eGet(config.eClass().getEStructuralFeature("Rtotal"));
       //encode the value of Ptotal to the State ptotal attribute
    stateInstance.eSet(stateInstance.eClass().getEStructuralFeature
       ("ptotal"), pTotal);
       //encode the value of Rtotal to the State rtotal attribute
    stateInstance.eSet(stateInstance.eClass().getEStructuralFeature
       ("rtotal"), rTotal);
       resourceSet.saveEObject(graph.getRoots().get(0),
"/Users/Mohammed");
```

B.3 Stochastic Graph Transformation Games

B.3.1 Mapping from Henshin to PRISM-games

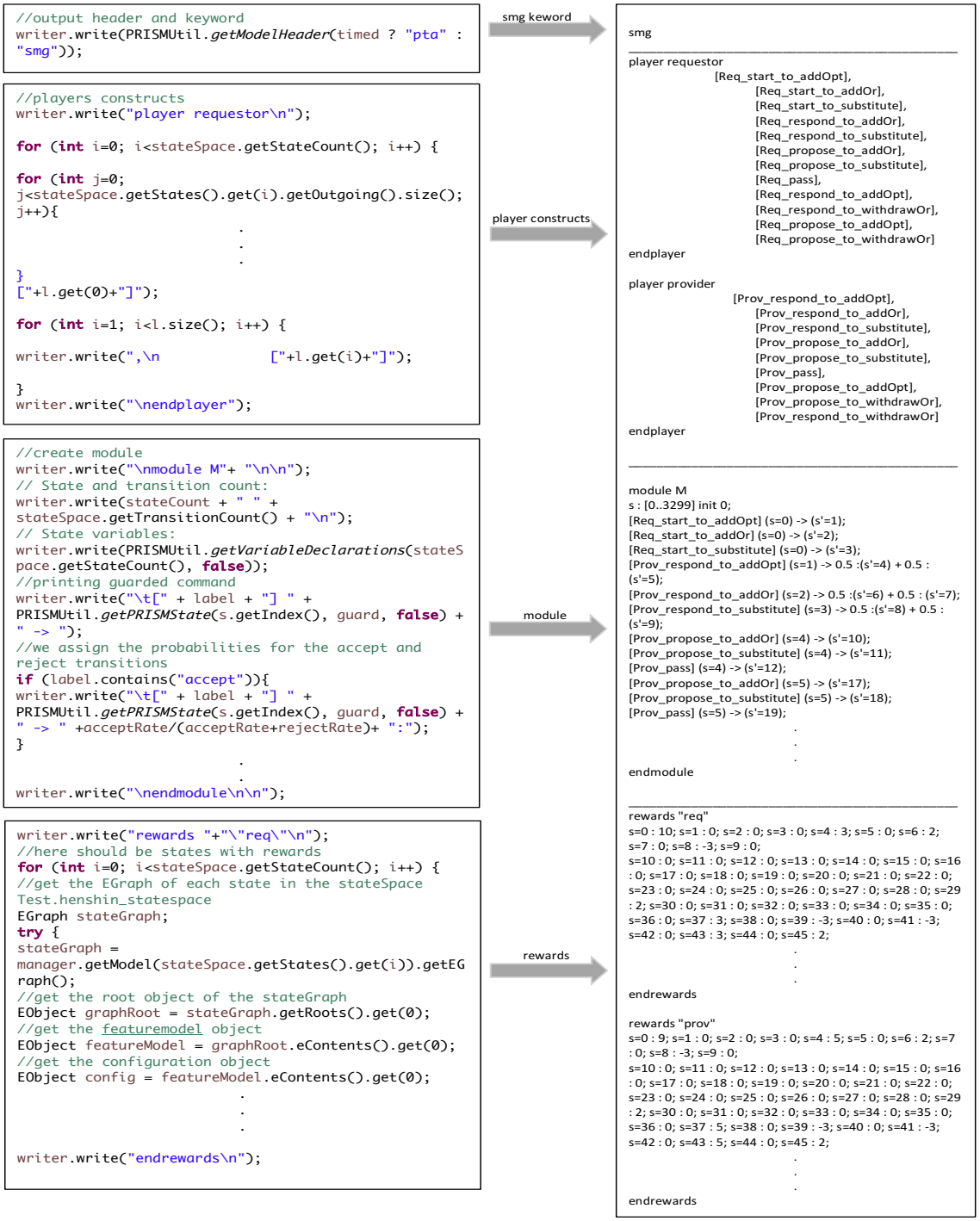


FIGURE B.3: Mapping from Henshin to PRISM-games

B.3.2 SMGs File

```
smg

player requestor
        [Req_start_to_addOpt],
        [Req_start_to_addOr],
        [Req_start_to_substitute],
        [Req_respond_to_addOr],
        [Req_respond_to_substitute],
        [Req_propose_to_addOr],
        [Req_propose_to_substitute],
        [Req_pass],
        [Req_respond_to_addOpt],
        [Req_respond_to_withdrawOr],
        [Req_propose_to_addOpt],
        [Req_propose_to_withdrawOr]
endplayer

player provider
        [Prov_respond_to_addOpt],
        [Prov_respond_to_addOr],
        [Prov_respond_to_substitute],
        [Prov_propose_to_addOr],
        [Prov_propose_to_substitute],
        [Prov_pass],
        [Prov_propose_to_addOpt],
        [Prov_propose_to_withdrawOr],
        [Prov_respond_to_withdrawOr]
endplayer

module M
s : [0..3299] init 0;
        [Req_start_to_addOpt] (s=0) -> (s'=1);
        [Req_start_to_addOr] (s=0) -> (s'=2);
        [Req_start_to_substitute] (s=0) -> (s'=3);
        [Prov_respond_to_addOpt] (s=1) -> 0.5 :(s'=4) + 0.5 : (s'=5);
        [Prov_respond_to_addOr] (s=2) -> 0.5 :(s'=6) + 0.5 : (s'=7);
        [Prov_respond_to_substitute] (s=3) -> 0.5 :(s'=8) + 0.5 : (s'=9);
        [Prov_propose_to_addOr] (s=4) -> (s'=10);
        [Prov_propose_to_substitute] (s=4) -> (s'=11);
        [Prov_pass] (s=4) -> (s'=12);
        [Prov_propose_to_addOr] (s=5) -> (s'=17);
        [Prov_propose_to_substitute] (s=5) -> (s'=18);
        [Prov_pass] (s=5) -> (s'=19);
        [Prov_propose_to_addOpt] (s=6) -> (s'=13);
        [Prov_propose_to_withdrawOr] (s=6) -> (s'=14);
        [Prov_propose_to_substitute] (s=6) -> (s'=15);
        [Prov_pass] (s=6) -> (s'=16);
        [Prov_propose_to_addOpt] (s=7) -> (s'=20);
        [Prov_propose_to_substitute] (s=7) -> (s'=21);
        [Prov_pass] (s=7) -> (s'=22);
        [Prov_propose_to_addOpt] (s=8) -> (s'=26);
        [Prov_propose_to_addOr] (s=8) -> (s'=27);
        [Prov_pass] (s=8) -> (s'=28);
        [Prov_propose_to_addOpt] (s=9) -> (s'=23);
        [Prov_propose_to_addOr] (s=9) -> (s'=24);
        [Prov_pass] (s=9) -> (s'=25);
        [Req_respond_to_addOr] (s=10) -> 0.5 :(s'=45) + 0.5 : (s'=46);
        [Req_respond_to_substitute] (s=11) -> 0.5 :(s'=63) + 0.5 : (s'=64);
        [Req_propose_to_addOr] (s=12) -> (s'=47);
        [Req_propose_to_substitute] (s=12) -> (s'=48);
        [Req_pass] (s=12) -> (s'=49);
        [Req_respond_to_addOpt] (s=13) -> 0.5 :(s'=43) + 0.5 : (s'=44);
        [Req_respond_to_withdrawOr] (s=14) -> 0.5 :(s'=72) + 0.5 : (s'=73);
        [Req_respond_to_substitute] (s=15) -> 0.5 :(s'=58) + 0.5 : (s'=59);
        [Req_propose_to_addOpt] (s=16) -> (s'=54);
        [Req_propose_to_withdrawOr] (s=16) -> (s'=55);
        [Req_propose_to_substitute] (s=16) -> (s'=56);
        [Req_pass] (s=16) -> (s'=57);
        [Req_respond_to_addOr] (s=17) -> 0.5 :(s'=29) + 0.5 : (s'=30);
        [Req_respond_to_substitute] (s=18) -> 0.5 :(s'=39) + 0.5 : (s'=40);
        [Req_propose_to_addOr] (s=19) -> (s'=34);
        [Req_propose_to_substitute] (s=19) -> (s'=35);
        [Req_pass] (s=19) -> (s'=36);
        [Req_respond_to_addOpt] (s=20) -> 0.5 :(s'=37) + 0.5 : (s'=38);
        [Req_respond_to_substitute] (s=21) -> 0.5 :(s'=41) + 0.5 : (s'=42);
        [Req_propose_to_addOpt] (s=22) -> (s'=31);
        [Req_propose_to_substitute] (s=22) -> (s'=32);
        [Req_pass] (s=22) -> (s'=33);
        [Req_respond_to_addOpt] (s=23) -> 0.5 :(s'=52) + 0.5 : (s'=53);
        [Req_respond_to_addOr] (s=24) -> 0.5 :(s'=50) + 0.5 : (s'=51);
        [Req_propose_to_addOpt] (s=25) -> (s'=60);
        [Req_propose_to_addOr] (s=25) -> (s'=61);
        [Req_pass] (s=25) -> (s'=62);
        [Req_respond_to_addOpt] (s=26) -> 0.5 :(s'=70) + 0.5 : (s'=71);
        [Req_respond_to_addOr] (s=27) -> 0.5 :(s'=65) + 0.5 : (s'=66);
        [Req_propose_to_addOpt] (s=28) -> (s'=67);
        [Req_propose_to_addOr] (s=28) -> (s'=68);
        [Req_pass] (s=28) -> (s'=69);
        [Req_propose_to_withdrawOr] (s=29) -> (s'=98);
        [Req_propose_to_substitute] (s=29) -> (s'=99);
        [Req_pass] (s=29) -> (s'=100);
                                .
                                .
                                .
                                .
                                .
                                .
                                .
                                .

endmodule

rewards "req"
s=0 : 10; s=1 : 0; s=2 : 0; s=3 : 0; s=4 : 3; s=5 : 0; s=6 : 2; s=7 : 0; s=8 : -3; s=9 : 0;
s=10 : 0; s=11 : 0; s=12 : 0; s=13 : 0; s=14 : 0; s=15 : 0; s=16 : 0; s=17 : 0; s=18 :
0; s=19 : 0; s=20 : 0; s=21 : 0; s=22 : 0; s=23 : 0; s=24 : 0; s=25 : 0; s=26 : 0; s=27
: 0; s=28 : 0; s=29 : 2; s=30 : 0; s=31 : 0; s=32 : 0; s=33 : 0; s=34 : 0; s=35 : 0;
s=36 : 0; s=37 : 3; s=38 : 0; s=39 : -3; s=40 : 0; s=41 : -3; s=42 : 0; s=43 : 3; s=44 :
0; s=45 : 2; s=46 : 0; s=47 : 0; s=48 : 0; s=49 : 0; s=50 : 2; s=51 : 0; s=52 : 3; s=53
: 0; s=54 : 0; s=55 : 0; s=56 : 0; s=57 : 0; s=58 : -3; s=59 : 0; s=60 : 0; s=61 : 0;
s=62 : 0; s=63 : -3; s=64 : 0; s=65 : 2; s=66 : 0; s=67 : 0; s=68 : 0; s=69 : 0; s=70 :
3; s=71 : 0; s=72 : -3; s=73 : 0; s=74 : 0; s=75 : 0; s=76 : 0; s=77 : 0; s=78 : 0; s=79
: 0; s=80 : 0; s=81 : 0; s=82 : 0; s=83 : 0; s=84 : 0; s=85 : 0; s=86 : 0; s=87 : 0;
s=88 : 0; s=89 : 0; s=90 : 0; s=91 : 0; s=92 : 2; s=93 : 0; s=94 : 3; s=95 : 0; s=96 : -
3; s=97 : 0; s=98 : 0; s=99 : 0; s=100 : 0;
                                .
                                .
                                .
                                .
                                .
                                .
                                .
                                .

endrewards
rewards "prov"
s=0 : 9; s=1 : 0; s=2 : 0; s=3 : 0; s=4 : 5; s=5 : 0; s=6 : 2; s=7 : 0; s=8 : -3; s=9 : 0;
s=10 : 0; s=11 : 0; s=12 : 0; s=13 : 0; s=14 : 0; s=15 : 0; s=16 : 0; s=17 : 0; s=18 :
0; s=19 : 0; s=20 : 0; s=21 : 0; s=22 : 0; s=23 : 0; s=24 : 0; s=25 : 0; s=26 : 0; s=27
: 0; s=28 : 0; s=29 : 2; s=30 : 0; s=31 : 0; s=32 : 0; s=33 : 0; s=34 : 0; s=35 : 0;
s=36 : 0; s=37 : 5; s=38 : 0; s=39 : -3; s=40 : 0; s=41 : -3; s=42 : 0; s=43 : 5; s=44 :
0; s=45 : 2; s=46 : 0; s=47 : 0; s=48 : 0; s=49 : 0; s=50 : 2; s=51 : 0; s=52 : 5; s=53
: 0; s=54 : 0; s=55 : 0; s=56 : 0; s=57 : 0; s=58 : -3; s=59 : 0; s=60 : 0; s=61 : 0;
s=62 : 0; s=63 : -3; s=64 : 0; s=65 : 2; s=66 : 0; s=67 : 0; s=68 : 0; s=69 : 0; s=70 :
5; s=71 : 0; s=72 : -4; s=73 : 0; s=74 : 0; s=75 : 0; s=76 : 0; s=77 : 0; s=78 : 0; s=79
: 0; s=80 : 0; s=81 : 0; s=82 : 0; s=83 : 0; s=84 : 0; s=85 : 0; s=86 : 0; s=87 : 0;
s=88 : 0; s=89 : 0; s=90 : 0; s=91 : 0; s=92 : 2; s=93 : 0; s=94 : 5; s=95 : 0; s=96 : -
3; s=97 : 0; s=98 : 0; s=99 : 0; s=100 : 0;
                                .
                                .
                                .
                                .
                                .
                                .
                                .
                                .
```

FIGURE B.4: SMGs File

B.3.3 Strategy Exported File

$MD.strat-v0.1	21 0	44 0	67 0	90 0
Adv:	22 1	45 0	68 0	91 1
0 0	23 0	46 0	69 0	92 0
1 0	24 0	47 0	70 0	93 0
2 0	25 1	48 0	71 0	94 0
3 0	26 0	49 0	72 0	95 0
4 0	27 0	50 0	73 0	96 0
5 0	28 1	51 0	74 0	97 0
6 0	29 0	52 0	75 1	98 0
7 0	30 0	53 0	76 0	99 0
8 0	31 0	54 0	77 0	100 1
9 0	32 0	55 0	78 0	101 0
10 0	33 0	56 0	79 1	.
11 0	34 0	57 0	80 0	.
12 0	35 0	58 0	81 0	.
13 0	36 0	59 0	82 1	.
14 0	37 0	60 0	83 0	.
15 0	38 0	61 0	84 0	.
16 0	39 0	62 0	85 0	.
17 0	40 0	63 0	86 0	.
18 0	41 0	64 0	87 1	.
19 0	42 0	65 0	88 0	.
20 0	43 0	66 0	89 1	.

FIGURE B.5: Strategy Exported File

Bibliography

[1] Muhamet Yildiz. *Backward Induction: Lecture Notes (Economic Applications of Game Theory)*. Massachusetts Institute of Technology, 2012. Available at `http://ocw.mit.edu/courses/economics/14-12-economic-applications-of-game-theory-fall-2012/lecture-notes/`. Accessed: 2016-06-16.

[2] Ronald Ashri, Iyad Rahwan, and Michael Luck. Architectures for negotiating agents. In *International Central and Eastern European Conference on Multi-Agent Systems*, pages 136–146. Springer, 2003.

[3] Bernard Mayer. *The dynamics of conflict resolution: A practitioner's guide*. John Wiley & Sons, 2010.

[4] Martin Bichler, Gregory Kersten, and Stefan Strecker. Towards a structured design of electronic negotiations. *Group Decision and Negotiation*, 12(4):311–335, 2003.

[5] William N Robinson and Vecheslav Volkov. Supporting the negotiation life cycle. *Communications of the ACM*, 41(5):95–102, 1998.

[6] A Lax David and K Sebenius James. *The Manager as Negotiator: Bargaining for Cooperation and Competitive Gain*. NY: Free Press, 1986.

[7] Gregory E Kersten. Modeling distributive and integrative negotiations. review and revised characterization. *Group Decision and Negotiation*, 10(6):493–514, 2001.

[8] Robert H Guttman and Pattie Maes. Cooperative vs. competitive multi-agent negotiations in retail electronic commerce. In *International Workshop on Cooperative Information Agents*, pages 135–147. Springer, 1998.

[9] Nicholas R Jennings, Peyman Faratin, Alessio R Lomuscio, Simon Parsons, Michael J Wooldridge, and Carles Sierra. Automated negotiation: prospects, methods and challenges. *Group Decision and Negotiation*, 10(2):199–215, 2001.

[10] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[11] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[12] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature oriented reuse method with domain specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[13] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[16] Jules White, Douglas C Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 225–234. IEEE, 2008.

[17] Jules White, Brian Dougherty, Doulas C Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, pages 11–20. Carnegie Mellon University, 2009.

[18] Goetz Botterweck, Daren Nestor, André PreuBner, Ciarán Cawley, and Steffen Thiel. Towards supporting feature configuration by interactive visualisation. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 125–131, 2007.

[19] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01, 2007.

[20] Mike Mannion. Using first-order logic for product line model validation. In *International Conference on Software Product Lines*, pages 176–187. Springer, 2002.

[21] Danilo Beuche. Variant management with pure::variants. *Pure-systems GmbH, Tech. Rep*, 2003.

[22] Ross Buhrdorf, Dale Churchett, and Charles W Krueger. Salion's experience with a reactive software product line approach. In *International Workshop on Software Product-Family Engineering*, pages 317–322. Springer, 2003.

[23] Jules White, Douglas C Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 129–140. IEEE, 2007.

[24] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, and Ebrahim Bagheri. Automated planning for feature model configuration based on functional and non-functional requirements. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 56–65. ACM, 2012.

[25] Martin L Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 76–85. IEEE, 1998.

[26] Timo Asikainen, Tomi Männistö, and Timo Soininen. Representing feature models of software product families using a configuration ontology. In *Proc of the ECAI*. Citeseer, 2004.

[27] Tuan Nguyen and Alan Colman. A feature-oriented approach for web service customization. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 393–400. IEEE, 2010.

[28] Silva Robak and Bogdan Franczyk. Modeling web services variability with feature diagrams. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 120–128. Springer, 2002.

[29] Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard. Variability modeling and QoS analysis of web services orchestrations. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 99–106. IEEE, 2010.

[30] Muhammad Naeem and Reiko Heckel. Towards matching of service feature models based on linear logic. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 13. ACM, 2011.

[31] Martin J Osborne. *An introduction to game theory*. Oxford University Press New York, 2004.

[32] Roger B Myerson. Game theory: analysis of conflict. *Harvard University*, 1991.

[33] Theodore L. Turocy and Bernhard von Stengel. Game theory. In Editor in Chief: Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 403 – 420. Elsevier, New York, 2003.

[34] J von Neumann, Oskar Morgenstern, et al. *Theory of games and economic behavior*, volume 60. Princeton university press Princeton, 1944.

[35] T. Roughgarden. An Algorithmic Game Theory Primer. In *Proceedings of the 5th IFIP International Conference on Theoretical Computer Science (TCS). An invited survey*. Citeseer, 2008.

[36] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[37] Martin J Osborne and Ariel Rubinstein. *A course in game theory*. MIT press, 1994.

[38] Richard Gibson. *Regret minimization in games and the development of champion multiplayer computer poker-playing agents.* PhD thesis, University of Alberta, 2014.

[39] Ichiro Obara. *Extensive Game with Perfect Information: Lecture Notes (Noncooperative Game Theory).* University of California, Los Angeles, 2012. Available at `http://www.econ.ucla.edu/iobara/201B.html`. Accessed: 2016-07-14.

[40] Y. Narahari. *Extensive Form Games: Lecture Notes (Game Theory).* Indian Institute of Science, 2012. Available at `http://lcm.csa.iisc.ernet.in/gametheory/lecture.html`. Accessed: 2016-07-14.

[41] Drew Fudenberg and Jean Tirole. *Game Theory.* MIT Press, Cambridge, MA, 1991.

[42] Jonathan Levin. *Extensive Form Games: Lecture Notes (Economics 203).* Stanford University, 2002. Available at `http://www.stanford.edu/~jdlevin/teaching.html`. Accessed: 2016-07-14.

[43] M Utku Ünver. *Extensive-Form Games with Perfect Information: Lecture Notes (Graduate Micro Theory II - 2).* Boston College, 2013. Available at `https://www2.bc.edu/~unver/teaching/gradmicro/maingradmicro2-II.html`. Accessed: 2015-01-28.

[44] Geir B Asheim. *Extensive Games with Perfect Information: Lecture Notes (Game Theory).* University of Oslo, 2001. Available at `http://folk.uio.no/gasheim/game03t3.pdf`. Accessed: 2017-01-12.

[45] Wikipedia. Subgame perfect equilibrium, 2016. [Online; accessed 14-July-2016].

[46] Reinhard Selten. Spieltheoretische behandlung eines oligopolmodells mit nachfrageträgheit: Teil i: Bestimmung des dynamischen preisgleichgewichts. *Zeitschrift für die gesamte Staatswissenschaft/Journal of Institutional and Theoretical Economics*, pages 301–324, 1965.

[47] Ichiro Obara. *Subgame Perfect Equilibrium: Lecture Notes (Noncooperative Game Theory (Economics 201B))*. University of California, Los Angeles, 2012. Available at `http://www.econ.ucla.edu/iobara/spe201b.pdf`. Accessed: 2016-07-14.

[48] Michael Wooldridge. Thinking backward with Professor Zermelo. *IEEE Intelligent Systems*, 30(2):62–67, 2015.

[49] Albert Banal-Estañol. *Chapter 5: Backwards Induction and SPNE: Lecture Notes (Game Theory)*. City University London, 2008. Available at `http://albertbanalestanol.com/wp-content/uploads/gtc-chapter5.pdf`. Accessed: 2016-07-15.

[50] Emmanual N Barron. *Game theory: an introduction*, volume 2. John Wiley & Sons, 2013.

[51] Kousha Etessami and Mihalis Yannakakis. Recursive Markov decision processes and recursive stochastic games. In *International Colloquium on Automata, Languages, and Programming*, pages 891–903. Springer, 2005.

[52] Aistis Simaitis. *Automatic verification of competitive stochastic systems*. PhD thesis, University of Oxford, 2014.

[53] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. In *Proceedings of the 18th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems, TACAS'12, pages 315–330. Springer-Verlag, 2012.

[54] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 43(1):61–92, 2013.

[55] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, Aistis Simaitis, and Clemens Wiltsche. On stochastic games with multiple objectives. In *International Symposium on Mathematical Foundations of Computer Science*, pages 266–277. Springer, 2013.

[56] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. PRISM-games: A model checker for stochastic multi-player games. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 185–191. Springer, 2013.

[57] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[58] Rajeev Alur and Thomas A Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[59] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, Aistis Simaitis, Ashutosh Trivedi, and Michael Ummels. Playing stochastic games precisely. In *International Conference on Concurrency Theory*, pages 348–363. Springer, 2012.

[60] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002.

[61] Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and nondeterministic systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 499–513. Springer, 1995.

[62] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[63] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems*, pages 53–113. Springer, 2011.

[64] `http://www.prismmodelchecker.org/games/properties.php`. Accessed: 2016-07-18.

[65] Shaheen Fatima, Sarit Kraus, and Michael Wooldridge. *Principles of automated negotiation*. Cambridge University Press, 2014.

[66] Jiangbo Dang. *Autonomous Agents in Service-oriented Negotiation: Strategy, Protocol, and Coordination*. ProQuest, 2006.

[67] Iyad Rahwan. *Interest-based negotiation in multi-agent systems*. PhD thesis, The University of Melbourne, 2005.

[68] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[69] Reiko Heckel and Ping Guo. Conceptual modeling of styles for mobile systems. In *Mobile Information Systems*, pages 65–78. Springer, 2005.

[70] Gregor Engels and Reiko Heckel. Graph transformation as a conceptual and formal framework for system modeling and model evolution. In *International Colloquium on Automata, Languages, and Programming*, pages 127–150. Springer, 2000.

[71] H Ehrig, K Ehrig, U Prange, and G Taentzer. Fundamentals of algebraic graph transformation (Monographs in Theoretical Computer Science. an EATCS Series). Secaucus, 2006.

[72] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. *Fundam. Inf.*, 74(1):31–61, October 2006.

[73] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[74] Michael R Berthold, Ingrid Fischer, and Manuel Koch. Attributed graph transformation with partial attribution, 2000.

[75] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[76] Claudia Ermel, Enrico Biermann, Johann Schmidt, and Angeline Warning. Visual modeling of controlled emf model transformation using henshin. *Electronic Communications of the EASST*, 32, 2011.

[77] Enrico Biermann, Claudia Ermel, Leen Lambers, Ulrike Prange, Olga Runge, and Gabriele Taentzer. Introduction to AGG and EMF Tiger by modeling a conference scheduling system. *International Journal on Software Tools for Technology Transfer*, 12(3-4):245–261, 2010.

[78] `http://git.eclipse.org/c/henshin/org.eclipse.emft.henshin.git/tree/plugins/org.eclipse.emf.henshin.statespace/model/statespace.ecore`. Accessed: 2016-05-18.

[79] Tushar Deshpande, Panagiotis Katsaros, Scott A Smolka, and Scott D Stoller. Stochastic game-based analysis of the DNS bandwidth amplification attack using probabilistic model checking. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 226–237. IEEE, 2014.

[80] Mária Svoreňová and Marta Kwiatkowska. Quantitative verification and strategy synthesis for stochastic games. *European Journal of Control*, 30:15 – 30, 2016. 15th European Control Conference, {ECC16}.

[81] Marcelo Fantinato, Maria Beatriz Felgar de Toledo, and IM de S Gimenes. A feature-based approach to electronic contracts. In *The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06)*, pages 34–34. IEEE, 2006.

[82] Marcelo Fantinato, Itana Maria de S Gimenes, and Maria Beatriz F de Toledo. Web service e-contract establishment using features. In *International Conference on Business Process Management*, pages 290–305. Springer, 2006.

[83] Marcelo Fantinato, Maria Beatriz Felgar De Toledo, and Itana Maria De Souza Gimenes. WS-contract establishment with QoS: an approach based on feature modeling. *International Journal of Cooperative Information Systems*, 17(03):373–407, 2008.

[84] Felipe Gonçalves Marchione, Marcelo Fantinato, Maria Beatriz F de Toledo, and Itana Gimenes. Price definition in the establishment of electronic contracts for web services. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 217–224. ACM, 2009.

[85] Daniel Avila Vecchiato, Maria Beatriz Felgar de Toledo, Marcelo Fantinato, and Itana Maria de Souza Gimenes. Electronic contract negotiation and renegotiation using features. In *WEBIST (2)*, pages 313–318, 2010.

[86] Daniel Avila Vecchiato, MBF Toledo, Marcelo Fantinato, and IMS Gimenes. A feature-based toolkit for electronic contract negotiation and renegotiation. In *Proc. IADIS Int. Conf. WWW/Internet*, volume 2010, pages 3–10, 2010.

[87] Gabriel Costa Silva, Itana Maria de Souza Gimenes, Marcelo Fantinato, and BF de Toledo. Towards a process for negotiation of e-contracts involving web services. *VIII Simpósio Brasileiro de Sistemas de Informação (SBSI 2012)*, pages 267–278, 2012.

[88] S Segura, D Benavides, A Ruiz-Cortés, and P Trinidad. Toward automated refactoring of feature models using graph transformations. *VII Jornadas sobre Programación y Lenguajes, PROLE*, pages 275–284, 2007.

[89] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated merging of feature models using graph transformations. In *Generative and Transformational Techniques in Software Engineering II*, pages 489–505. Springer, 2008.

[90] David Benavides, S Trujillo, and P Trinidad. On the modularization of feature models. In *First European Workshop on Model Transformation*, 2005.

[91] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, pages 1–47, 2015.

[92] Frederik Deckwerth, Géza Kulcsár, Malte Lochau, Gergely Varró, and Andy Schürr. Conflict detection for edits on extended feature models using symbolic graph transformation. *arXiv preprint arXiv:1604.00347*, 2016.

[93] Géza Kulcsár, Frederik Deckwerth, Malte Lochau, Gergely Varró, and Andy Schürr. Improved conflict detection for graph transformation with attributes. *arXiv preprint arXiv:1504.02614*, 2015.

[94] Jesús García-Galán, Pablo Trinidad, and Antonio Ruiz-Cortés. Multi-user variability configuration: A game theoretic approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 574–579. IEEE, 2013.

[95] Xianrong Zheng, Patrick Martin, Wendy Powley, and Kathryn Brohman. Applying bargaining game theory to web services negotiation. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 218–225. IEEE, 2010.

[96] Yi Sun, Zhiqiu Huang, and Changbo Ke. Using game theory to analyze strategic choices of service providers and service requesters. *Journal of Software*, 9(11):2918–2924, 2014.

[97] Guido Boella and Leendert Van Der Torre. A game theoretic approach to contracts in multiagent systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(1):68–79, 2006.

[98] Ken Binmore and Nir Vulkan. Applying game theory to automated negotiation. *Netnomics*, 1(1):1–9, 1999.

[99] Jun Yan, Ryszard Kowalczyk, Jian Lin, Mohan B Chhetri, Suk Keong Goh, and Jianying Zhang. Autonomous service level agreement negotiation for service composition provision. *Future Generation Computer Systems*, 23(6):748–759, 2007.

[100] Andrew Byde, Michael Yearworth, Kay-Yut Chen, and Claudio Bartolini. Autona: A system for automated multiple 1-1 negotiation. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 59–67. IEEE, 2003.

[101] Pu Huang and Katia Sycara. A computational model for online agent negotiation. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 438–444. IEEE, 2002.

[102] Jin Baek Kim and Arie Segev. A framework for dynamic ebusiness negotiation processes. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 84–91. IEEE, 2003.

[103] Wei Shang, Yi-Jun Li, and An-Shi Xie. A game-theory based knowledge presentation for bilateral business negotiation. In *Machine Learning and Cybernetics, 2003 International Conference on*, volume 5, pages 2650–2653. IEEE, 2003.

[104] K-M Chao, Muhammad Younas, Rachid Anane, C-F Tsai, and V-W Soo. Degree of satisfaction in agent negotiation. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 68–75. IEEE, 2003.

[105] Sören Preibusch. Implementing privacy negotiation techniques in e-commerce. In *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, pages 387–390. IEEE, 2005.

[106] Benay Kumar Ray, Sunirmal Khatua, and Sarbani Roy. Negotiation based service brokering using game theory. In *Applications and Innovations in Mobile Computing (AIMoC), 2014*, pages 1–8. IEEE, 2014.

[107] H.N. Hindriks. Generating game strategies using graph transformations. *21st Twente Student Conference on IT*, 21, 2014.

[108] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[109] Matteo Cavaliere, Attila Csikász-Nagy, and Ferenc Jordán. Graph transformations and game theory: A generative mechanism for network formation. *University of Trento, Technical Report CoSBI 25/2008*, 2008.

[110] Łukasz Kaiser. Synthesis for structure rewriting systems. In *Mathematical Foundations of Computer Science 2009*, pages 415–426. Springer, 2009.

[111] Łukasz Kaiser and Łukasz Stafiniak. Playing general structure rewriting games. In *Proceedings of AGI '10*. Atlantis Press, 2010.