POLITECNICO DI MILANO

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
Corso di Laurea in Ingegneria Informatica

Tesi di laurea specialistica

SYNTHESIZING PRODUCT LINE CONTROLLERS FROM SCENARIO-BASED SPECIFICATIONS

Relatori:
Prof. Carlo GHEZZI
Prof. Patrick HEYMANS

Correlatori:
Prof. Joel GREENYER
Maxime CORDY

Tesi di laurea di:
Erika GRESSI
Matr. n. 750909

Anno Accademico 2012–2013

Contents

1 Introduction **1**
- 1.1 Running example 6
- 1.2 Overview 7

2 Foundations **9**
- 2.1 Software Product Lines 9
  - 2.1.1 The SPLE framework 12
  - 2.1.2 Variability modelling 15
- 2.2 Featured Transition Systems 20
  - 2.2.1 Syntax and semantics of LTS 20
  - 2.2.2 Syntax and semantics of FTS 21
- 2.3 Modal Sequence Diagrams 25
  - 2.3.1 MSD activation, progress and termination 25
  - 2.3.2 Message attributes and violations 27
  - 2.3.3 Environment assumptions 29
  - 2.3.4 Parameterized and forbidden messages 30
  - 2.3.5 The play out Semantics 32
  - 2.3.6 Satisfiability, consistency and consistent executability . . . 34
- 2.4 Formal scenario-based specification of SPLs 35

3 Featured Synthesis **39**
- 3.1 The approach 39
- 3.2 Featured Reachability 43
  - 3.2.1 Retrieving the FD expression 43
  - 3.2.2 Retrieving the set of featured transitions. 45
  - 3.2.3 Winning condition 48
  - 3.2.4 Featured Reachability 52
- 3.3 Featured Büchi 53

4 Implementation **57**
4.1 Scenariotools . . . 57
4.1.1 Modeling . . . 57
4.1.2 Simulation . . . 60
4.1.3 Synthesis . . . 60
4.1.4 Configuring Executable Specifications . . . 61
4.2 Architecture . . . 63
4.2.1 Inputs and outputs of the simultaneous synthesis . . . 63
4.2.2 Main structure and dependencies . . . 64
4.3 Using the package . . . 67
4.3.1 Modeling . . . 67
4.3.2 Automated generation of the ECore class model . . . 70
4.3.3 Running the simultaneous synthesis and showing the featured state graph . . . 70
4.4 Limitations and perspectives . . . 73

5 Evaluation **74**
5.1 Applicability Evaluation . . . 74
5.1.1 CoPAInS Overview . . . 75
5.1.2 Featured synthesis outcome . . . 78
5.2 Performance Evaluation . . . 80
5.2.1 CoPAInS and Vending Machine examples . . . 80
5.2.2 Comparison between featured and sub-optimal synthesis . 81
5.2.3 Comparison between featured and incremental on-the-fly synthesis . . . 83
5.2.4 Discussion . . . 89

6 Conclusion and Outlook **92**

A Whole SPL specification and FGG for the CoPAInS example **95**
A.1 SPL specification . . . 95
A.2 Featured Game Graph . . . 104

Bibliography **105**

List of Figures

1.1 Schema of the synthesis approach for Software Product Lines. . . 5

2.1 Effect of PLE on developing costs 2.1(a) and time to market 2.1(b) when the number of products in the product line increases. 11
2.2 Schema of the SPLE framework [Pohl et al. 2005]. 13
2.3 Types of edge in a feature diagram. 17
2.4 The FD capturing the set of valid products for the Vending Machine example 18
2.5 FTS of the traffic light controller 22
2.6 FTS of the Vending Machine example 23
2.7 The object system and MSDs with concrete lifelines. 26
2.8 The possible combinations of temperature and execution kind for an MSD message. 27
2.9 Assumption MSD for the Vending Machine example. 29
2.10 MSD with a parameterized message for the Vending Machine example. 31
2.11 MSD with a forbidden message for the Vending Machine example. 32
2.12 Valid and violating super-steps 33
2.13 The SPL specification for our VendingMachine example. 36

3.1 Featured game graph example 42
3.2 Retrieving the Boolean formula specifying the valid products from the FD 45
3.3 Deriving a concise game graph from a SPL specification 47
3.4 Retrieving the winning condition for different products 50
3.5 Retrieving the winning condition for the VendingMachine example. 51
3.6 Two iterations of the Büchi algorithm applied to the VendingMachine example. 56

4.1 Overview of the package structure for the VendingMachine example. 58

4.2 The contents of the base package and Tea package from the VendingMachine example. 59
4.3 The implementation of the FD in ScenarioTools for the VendingMachine example. 59
4.4 A screenshot of the ScenarioTools runtime during the simulation of the VendingMachine example. 60
4.5 Part of a state space diagram explored by the ScenarioTools synthesis for the VendingMachine example. The graph considers the product with all features enabled 61
4.6 Overview of the models involved in the execution of MSDs for a static object system, with static lifeline bindings 62
4.7 The inputs and outputs of the plug-in 64
4.8 The architecture of the plug-in and its main dependencies illustrated by a UML class diagram. 65
4.9 Extension of the existing ECore class model build through EMF. . 66
4.10 Overview of the diagrams involved in the SPL specification of the VendingMachine example in the Papyrus editors. 67
4.11 Some of the stereotypes provided by ScenarioTools 68
4.12 Some of the steps to create a concrete object system 69
4.13 Simultaneous synthesis of the VendingMachine example. 71
4.14 Generating the featured game graph for the VendingMachine example. 72

5.1 The Object System for the CoPAInS example 76
5.2 The FD for the CoPAInS example 76
5.3 Two MSDs from the CoPAInS SPL specifiction. 78
5.4 The featured synthesis outcome for the CoPAInS example. 79
5.5 An excerpt of the FGG for the CoPAInS example. 79
5.6 Cascading example with only one hot executed message per MSD. 82
5.7 The specification used to show the difference between an OTF and a NOTF synthesis in terms of number of visited states. . . . 85
5.8 The difference between an OTF and a NOTF synthesis in terms of number of visited states. 86
5.9 Cascading example with two hot executed messages per MSD for features at level 2. 88
5.10 The difference between an incremental OTF and a featured NOTF synthesis in terms of number of visited states. 90
5.11 Cascade example with one hot, executed message per MSD. Synthesis times and number of states explored for the three approaches. 91

A.1 FD representing the valid combinations of features for the CoPAInS SPL. 95
A.2 **PatientAsksForHelp R1)** (liveness requirement) After the patient asks for help the alarm is set on and eventually the patient must be helped. 96
A.3 **PatientAsksForHelp A1)** We assume that the patient does not ask for help twice before the alarm is called off. 96
A.4 **CallHelper R1)** After the alarm is set on, the HCS sends an help request to the smartphone of the first helper in the list. The smartphone shows the message to its owner. 97
A.5 **CallHelper R2)** After the helper reads the message, he could reply YES or NO. In the first case the smartphone tells the HCS that the first helper is available to help. Otherwise the HCS is informed that the helper is not available. 98
A.6 **CallHelper A1)** We assume that after the smartphone shows the message to the helper, either he answers with his availability or with his unavailability. Through this assumption, we basically rule out the scenario in which the helper does not reply at all. . . . 99
A.7 **CallAmbulance R1)** Whenever the last helper is not able to help, the HCS sends a request to the ambulance. 100
A.8 **CallAmbulance A1)** The ambulance is always available to help. 100
A.9 **DoorOpened R1)** When the helper enters the room, the alarm is set off. 101
A.10 **DoorOpened R2)** When the ambulance enters the room, the alarm is set off. 101
A.11 **DoorOpened A1)** We assume that when the helper replies that he is available to help, he will eventually show up. 102
A.12 **DoorOpened A2)** We assume that when the ambulance replies that it is available to help, he will eventually show up. 102
A.13 **DoorOpened A3)** When the helper enters the room, after the alarm is called off, the patient is helped. 103
A.14 **DoorOpened A4)** When the ambulance enters the room, after the alarm is called off, the patient is helped. 103
A.15 FGG generated when exploring the state space of the CoPAInS SPL specification. 104

List of Tables

5.1 Informal requirements and assumptions for the CoPAInS example. 77
5.2 Comparison on synthesis times and number of explored states for the featured synthesis, sub-optimal and incremental OTF synthesis approach. 81
5.3 Synthesis times for the cascading example. Comparison between the featured synthesis and the sub-optimal approach. 84
5.4 Synthesis times for the cascading example. Comparison between the featured NOTF synthesis and the incremental OTF approach. 87
5.5 Synthesis times for the modified cascading example. Comparison between the featured NOTF synthesis and the incremental OTF approach. 89

List of Algorithms

1 FD 44
2 Post(S) 46
3 FR(G) 53
4 FB 55

Abstract

Modern software-intensive systems usually consist of a set of components which interact to achieve their functionalities. Often those systems are part of a family of product variants, also called Software Product Line (SPL). Variability in an SPL is commonly captured by features, i.e. functions or components that may or may not be present. Although reusing features leads to lower costs and shorter time in the SPL development process, it also adds complexity to the design task.

In order to cope with that complexity, an intuitive, yet precise way to design those systems is required. Recent work proposes an SPL scenario-based specification where each feature is associated to a set of Modal Sequence Diagrams (MSD) specifying its behavioral aspects, allowing us to obtain the specification of a product by composing that of its constituent features.

However inconsistencies may be introduced. If they remain undetected, some product could turn out to be unrealizable, leading to costly iterations in the later development. Thus it is important to ensure that each product is implementable, beforehand.

This thesis presents a technique to automatically check the consistency of an SPL specification as a whole, instead of performing a separate check on each product. We exploit the fact that if variants are similar, the state graphs induced by each product's specification are also likely to be similar. Given an SPL specification we derive a global state graph representing all possible executions of all possible products in any environment. In such a graph we maintain a link between a given execution and the features needed to trigger it. This information is then used to determine which products are realizable.

We evaluate the applicability and efficiency of our approach, achieving benefits over performing individual checking of each variant separately. We also compare our technique with an alternative incremental approach, optimized to be on-the-fly, i.e. exploring only parts of the state space. We discover that the latter still performs better when more of those graph portions can be avoided. We expect that remarkable results could be achieved by making our technique on-the-fly.

Sommario

I sistemi *software-intensive* spesso consistono di un insieme di componenti che interagiscono per adempiere funzionalità complesse. Talvolta essi sono parte di una famiglia di prodotti simili, detta *Software Product Line* (SPL). Le differenze e le caratteristiche comuni tra ogni variante possono essere espresse da *feature*, cioè funzionalità o componenti facenti parte o meno del prodotto. Sebbene il riutilizzo di feature porti a una riduzione di costi e tempi nel processo di sviluppo di SPL, esso aggiunge maggiore complessità al design.

Per far fronte a tale complessità, un recente lavoro propone una specifica SPL basata su scenari in cui, ad ogni feature, viene associato un insieme di Modal Sequence Diagram (MSD) che ne descrive il comportamento, permettendo di ottenere la specifica di un prodotto come combinazione delle specifiche delle feature che lo compongono.

Tuttavia le eventuali inconsistenze nelle specifiche, se non tempestivamente rilevate, portano a costose iterazioni nella fase di sviluppo di prodotti non realizzabili. È dunque importante verificarne anticipatamente la consistenza.

Questa tesi presenta una tecnica per la verifica automatica della consistenza di un'intera specifica SPL, alternativa all'esecuzione di verifiche individuali su ogni prodotto. Sfruttando il fatto che le specifiche di singole varianti generano *state-graph* simili, deriviamo uno state-graph globale da una specifica SPL, che rappresenta tutte le possibili esecuzioni per tutti i possibili prodotti. In questa fase viene mantenuto un legame tra un'esecuzione e le feature necessarie ad attivarla. Tali informazioni vengono successivamente utilizzate per determinare i prodotti realizzabili.

Vengono valutate l'applicabilità e le prestazioni del nostro approccio, ottenendo vantaggi rispetto alla verifica consecutiva di ogni variante. In aggiunta ci confrontiamo con un approccio incrementale, ottimizzato per essere on-the-fly, ovvero per esplorare solo alcune parti dello state-graph. Scopriamo che quest'ultimo metodo risulta più efficiente quando molte porzioni del grafo possono essere evitate. Riteniamo che notevoli risultati potrebbero essere raggiunti rendendo la nostra tecnica on-the-fly.

Sommario esteso

I moderni sistemi software-intensive sono tipicamente costituiti da un insieme di componenti che interagiscono al fine di adempiere funzionalità complesse. Spesso tali sistemi sono parte di una famiglia di prodotti, ovvero un insieme di varianti di prodotto anche chiamata Software Product Line (SPL). Le differenze e le caratteristiche comuni tra ogni variante possono essere espresse da *feature*, cioè funzionalità o componenti facenti parte o meno del prodotto. Diversi prodotti della famiglia possono essere gestiti come diverse combinazioni di queste feature. Sebbene il riutilizzo di feature porti a una riduzione di costi e tempi nel processo di sviluppo di SPL, esso aggiunge maggiore complessità al design.

Per far fronte a tale complessità, è necessario adottare un metodo che sia intuitivo, ma anche preciso, per la specifica di SPL. Un recente lavoro propone una specifica SPL basata su scenari in cui ogni feature viene associata ad un insieme di Modal Sequence Diagram (MSD). Questi ultimi rappresentano un'estensione degli UML sequence diagram e permettono di descrivere gli aspetti comportamentali di ciascuna feature separatamente, gestendo la specifica di ogni prodotto come combinazione delle specifiche delle feature che lo compongono.

Tuttavia se le eventuali inconsistenze nelle specifiche non vengono tempestivamente rilevate, qualche prodotto nella SPL potrebbe rivelarsi irrealizzabile solo in fase di sviluppo, portando a costose iterazioni. È dunque importante verificare la consistenza delle specifiche in maniera preventiva.

Recentemente abbiamo proposto un approccio di tipo game-based per verificare la consistenza di specifiche SPL. La verifica di realizzabilità di un prodotto puó essere effettuata a partire dallo state-graph indotto dall'insieme di MSD che descrivono gli aspetti comportamentali dello stesso. Sfruttando il fatto che per varianti simili tali state-graph sono molto probabilmente simili, abbiamo sviluppato una tecnica per ottenerli in maniera incrementale, utilizzando di volta in volta le informazioni recuperate durante le verifiche precedenti. Un limite di questo approccio è che la sua efficienza dipende dall'ordine in cui le specifiche di ogni singola variante sono derivate ed è perciò soggetta a grandi variazioni.

Partendo dalla stessa intuizione, questa tesi presenta una tecnica per la verifica automatica della consistenza di un'intera specifica SPL, alternativa all'ese-

cuzione di una verifica indipendente su ogni variante. Data una specifica SPL, deriviamo uno state-graph globale, rappresentante tutte le possibili esecuzioni di tutti i prodotti della SPL con qualsiasi ambiente, nel quale viene mantenuto un collegamento tra una data esecuzione e le feature necessarie ad attivarla. Tali informazioni vengono successivamente utilizzate per determinare l'insieme dei prodotti realizzabili.

L'approccio viene implementato in *ScenarioTools*, una suite di strumenti che supporta la creazione di specifiche SPL e la verifica della loro realizzabilità. La nostra estensione permette di ottenere l'insieme di prodotti realizzabili data una specifica SPL e di visualizzare lo state-graph globale utilizzato durante il processo.

L'applicabilità dell'approccio viene valutata su diversi casi pratici, al fine di verificarne la capacità di riconoscere la non realizzabilità di alcuni prodotti, date specifiche SPL parzialmente inconsistenti. Valutiamo anche l'efficienza del nostro metodo confrontandolo con quello basato sulla verifica consecutiva di ogni singolo prodotto, ottenendo notevoli benefici sia sul numero di stati esplorati che sul tempo impiegato. Effettuiamo un confronto anche con l'approccio incrementale precedentemente menzionato, che è ottimizzato per essere on-the-fly, ovvero permette di esplorare solo parzialmente lo state-space indotto dalla specifica evitando alcuni percorsi alternativi. Scopriamo che quest'ultimo risulta più efficiente quando molte porzioni del grafo possono essere evitate. Sulla base dei test effettuati, ci aspettiamo che risultati ancora più significativi potranno essere ottenuti rendendo la nostra tecnica on-the-fly.

Chapter 1

Introduction

The relationship between software and our life is becoming more and more tight. Many of our daily activities critically depend on *software-intensive systems*, i.e. systems where software contributes to the design, construction, deployment, and evolution of the system as a whole [IEEE-Std-1471-2000]. We can retrieve those systems in pretty much all sectors and activities such as banking, communications, transportation and medicine.

In order to fulfill user needs, mostly every software development organization looks for new ways to improve its performance. That is not only about productivity. Before increasing the operational efficiency, a primarily important aspect is eliminating inefficiencies within the development process itself, i.e. in the set of steps that leads from the general informal requirements to the executable code fulfilling those requirements.

One of the major issues in the development of a software product is that very often informal requirements are ambiguous and sometimes even conflicting. Thus engineers usually express user requirements more formally as a *specification*. While requirements describe what the purpose of the system is, a specification describes the set of allowed behaviors of a system [Abadi et Al. 1989] in order to fulfill the requirements. A specification is a more technical representation of user needs, but still close to users intuition. This latter characteristic is fundamental to understand if the specification is an adequate description of the real-life problem. A specification with which users happen to be comfortable with is the one based on scenarios [Wiedenhaupt et Al. 1998] i.e. connected sequences of events or actions taken that should or should not occur within the system under certain circumstances.

One peculiarity of the systems we are considering is that they are reactive. A *reactive system* is commonly defined as the one that "*keeps an ongoing relationship with its environment*" [Harel and Pnueli 1985]. Environment events cannot be controlled by the system. They are intended as stimuli to which the system

will react. Systems having no control over environment actions are called *open systems*.

UML sequence diagrams [OMG UML 2011] are a popular example of *scenario-based specification* used to describe reactive systems. They represent interactions between different components in a system and its environment in form of messages exchanged in a particular order. Sequence diagrams are often not expressive enought to represent the whole complexity of software systems. They emphasize synchronous message passing and, consequently, they are not particularly effective in describing concurrent activities. Moreover, even in the context of open reactive systems, one may at least make some assumption on the behavior of the environment. In UML sequence diagrams this is not possible.

To cope with those limitations we use *Modal Sequence Diagrams* (MSD) in our approach. MSDs are an interesting language for *scenario-based specifications* that extends UML sequence diagrams to describe sequences of messages that may, must, or must not occur in a system, allowing a distinction between possible and necessary behavior [Harel and Maoz 2008]. An MSD specification typically contains a set of MSDs that specify the requirements of the system, also called *requirement MSDs*. However, also assumption MSDs can be included, allowing engineers to make assumptions on the environment's behaviour. Besides specifying what a system must, may or may not do, such an extended MSD specification is also able to assume how the environment reacts to system events.

Given the typical complexity of software requirements, inconsistencies between scenarios represent a common risk. If a specification happens to be inconsistent, no system will be able to realize it and, as stated in [Abadi et Al. 1989] a specification is useless if it cannot be realized by any concrete implementation. Unfortunately very often inconsistencies are detected only later in the development process, leading to costly iterations. To increase the efficiency of a software development process, it is necessary to avoid the useless effort of designing and implementing products provided with an inconsistent specification. The problem of automatically constructing a system model from a consistent specification is known as *synthesis*. It performs a constructive proof that the specification is implementable, and that the product is realizable. This method has many considerable advantages. First, in case the specification is not consistent, no time is wasted in the implementation of an unrealizable product. On the other hand, if it is consistent, we can avoid an extensive verification in the final phase of the development process, since the correctness of the implementation with respect to the specification is inherently given. Finally, it can drive the development of a software product through the synthesis of a *controller*, i.e. a strategy that the system can follow to satisfy the specification regardless of the actions of the environment.

Adopting synthesis could be particularly convenient when the system com-

plexity grows and the probability of conflicting requirements, thus inconsistent specification, increases. One example is when not only a single system, but rather different variants of the same system are developed. Systems in which variability plays such a preminent role, are known as *variability-intensive systems.* A particular class of variability-intensive systems, those in which variability is systematically planned, are *Software Product Lines* (SPLs). Different products in a SPL, also referred to as a *family* of products, can be considered *variants* if they share a set of common characteristics, but also differ in at least one missing or added chunk of functionality. Commonalities and variabilities between products are typically expressed in terms of *features*. Many definitions of the term *feature* exist [Classen et Al 2008]. In the rest of the thesis we stick with the definition in [Kang et al. 1990] for which a feature is a *a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems.*

In the context of a product line, given a set of features, a product can be represented as a feature combination, in the same way in which similar, yet different lego structures can be build using different subsets of the same set of bricks. Among all the possible combinations, only some of them are legal. To specify valid products, one commonly uses *Feature Diagrams* (FDs), a graphical representation of a product line by means of nodes, representing features, and edges, representing constrainst on the possible feature combinations.

The goal of *Software Product Line Engineering* (SPLE) is to consider all the variants together throughout the whole development process, exploiting commonalities between different products by systematically reusing them. This approach has many advantages. First, it leads to an effective reduction of development costs and time to market, since commonalities are only developed once. A second key advantage of SPLE and reuse, is that commonalities are tested in different situations and products, supporting fault detection and a consequent enhancement in the quality of the SPL as a whole after error correction. Finally, customer satisfaction is a natural consequence of the advantages above. Nevertheless, variability introduces an additional dimension of possible conflicts in requirements, given potential dependencies and incompatibilities created when combining requirements related to individual features. Usually those conflicts are almost impossible to identify for engineers.

Designing a scenario-based specification for an SPL and automatically checking its consistency is a major challenge in SPLE and has not yet been deeply explored. That is the challenge that this thesis intends to undertake.

Inspired from [Greenyer et al. 2011] we adopt a scenario-based specification for SPLs, which consists of MSDs and FDs. In that method, the individual behavior of each feature in the FD is specified as a set of MSDs. The overall MSD specification for a particular variant of the product line can be achieved by composing the MSDs associated to its features. In other words the MSD specification

of a product is the union of the MSDs specifying its constituent features. Moreover MSDs are particularly suitable to describe a feature by extending behavioural aspects of existing similar features.

Given an SPL specification we want to identify which products, i.e. combinations of features, have a consistent specification and for each of them, synthesize a controller that realizes its specification. Not many approaches exist for the realizability checking of SPLs.

Intuituitively, it could be performed by a separate synthesis for each individual product. Although correct, this is a sub-optimal approach since it does not take any advantage of variability and commonalities between products.

An alternative to the individual synthesis is the method developed by [Greenyer et al. 2011] and based on *Featured Transition Systems* (FTSs). FTSs are a recent formalism for modeling the behavior of a set of products. They consist in a directed graph in which nodes are the states of the interaction between the system and the environment, whereas transitions express a system or an environment event and are labelled with constraints over a set of features from an attached FD. That way a product can execute the transition only when its consisting features satisfy the associated constraints. However applying model checking to check the consistency of an SPL scenario-based specification has a few drawbacks and sometimes it returns false negatives.

To cope with those limitations we recently proposed an incremental synthesis approach [Greenyer et Al. 2013] handling the problem with a game-based technique. There we view the realizability-checking problem as the problem of finding a strategy in an infinite game played by the system against the environment. Our technique consists of an extension of the algorithm for solving Büchi games which are games requiring to infinitely often reach a state with given characteristics. Given an SPL specification, we first derive similar products from the FD iteratively by exploiting common features. Then we synthesize each product specification taking advantage of the outcome of similar, already synthesized ones. The efficiency of this method depends on the order in which products are synthesized and is thus subject to large variations. Often only slight improvements are achieved with respect to synthesizing each variant individually. Besides, even if commonalities between products are exploited by systematically synthesizing controllers based on similar ones, again one synthesis per product is performed.

This thesis proposes an alternative, more radical approach to the synthesis of similar products in an SPL, with the goal to increase the efficiency even more. We still handle the problem of checking the realizability of a given specification as a two-player game between the system and the environment, which has to be won by the system. Besides, we synthesize the whole SPL specification, i.e. all products, in one synthesis. Commonalities between products are exploited at a

deeper level to reduce the synthesis effort.

The principles of our approach are given in Figure 1.1 and explained in the following.

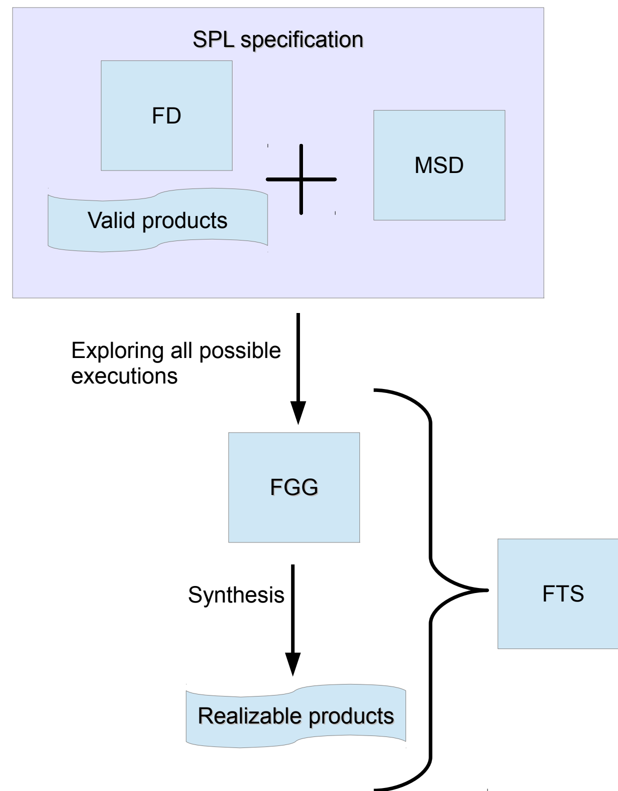


Figure 1.1: Schema of the synthesis approach for Software Product Lines.

First from an SPL specification, we build an extension of Büchi-Game automaton, which is a representation of all possible executions of the system with any environment for all possible products. The extension, also called *Featured Game Graph* (FGG) throughout the thesis, is born from the observation that if products in an SPL are similar, their corresponding Büchi-Game automata are very likely to share states and transitions. A major difference between our extension and a Büchi-Game automaton is that it maintains a link between a given execution and the features needed to trigger that execution. The information is stored into transitions in the automaton, consequently corresponding not only to a possible system or environment event given a particular state, but also to the particular subset of features in the SPL for which the event is admissible.

Then we design an algorithm that determines which products have a consistent specification. This algorithm makes use of the information regarding features contained in the automaton. During the process, transitions labeled with features that are not part of the product we are considering are simply ignored.

In addition to the consistency checking, the algorithm also synthesizes an FTS, to concisely represent a global controller of the set of realizable products. The difference between an FTS and an FGG is that the latter also contains possible executions for unrealizable products.

From this FTS one can, for example, extract a controller for a given consistent product, or even generate code with implemented variability. However, this is not in the scope of this thesis.

Our technique is implemented in the *ScenarioTools* tool suite [1], a collection of Eclipse-based tools which support the modeling, simulation and synthesis of MSD specifications.

1.1 Running example

Throughout this thesis, we use a simple running example which is inspired by [Fantechi and Gnesi 2008] and consists in a family of simplified vending machines. We will commonly refer to this example as the *Vending Machine example*.

In its basic form the vending machine takes a coin, returns change, dispenses a cup and then prepares tea by pouring sugar, hot water and tea powder into the cup. In such a scenario we identify three main interacting entities:

- the person, e.g. a *student*, who inserts a coin into the vending machine to activate it;
- a (general) *dispenser* responsible for dispensing cups, returning change and pouring sugar, tea powder and hot water;
- the central controller which controls the whole system and interacts with the dispenser, giving it instructions on what to do. We generally refer to that central controller as *machine*.

The general requirements for this system are informally listed below.

- A vending machine is activated by a coin. After the student pays, the dispenser should return some change.

[1] ScenarioTools http://scenariotools.org

- When change is returned, a cup is dispensed. The presence of the cup is noticed by a sensor which we consider part of the same machine entity, for simplicity.
- Tea should be prepared if and only if the sensor senses that a cup is already in the dispenser.
- Preparing tea implies pouring sugar, tea powder and hot water into the cup. We are not interested in the order in which those ingredients are served.

A number of variants of this basic machine can also be considered. In this work we examine the following variability.

- Pouring sugar is optional. It could also be poured by hand.
- Vending machines could only serve tea, without also dispensing cups.
- Even if rare, also machines which only distribute cups exist.

In the next chapters we will first use the Vending Machine example to describe the different models employed by our approach. Then also to better illustrate the approach itself.

1.2 Overview

This thesis is structured as follows.

Chapter 2 gives the essential concepts and definitions related to Software Product Lines (SPLs) and the models employed by our approach. In particular we present there Feature Diagrams (FDs), Featured Transition Systems (FTSs), Modal Sequence Diagrams (MSDs) and a recent approach for specifying SPLs as a combination of FDs and MDSs.

Chapter 3 illustrates how the models presented in Chapter 2 are used in order to develop a technique for systematically derive consistent products from an SPL specification. It details our approach, consisting of an extension of the algorithm for solving Büchi games, and explains how the presence of features impacts our algorithms.

Chapter 4 outlines our implementation. It consists of an Eclipse plug-in which is integrated into ScenarioTools, an Eclipse based tool suite supporting the modeling, simulation and synthesis of MSD specifications. We describe the general architecture of our plug-in and outline how to actually use our extension within the tool.

Chapter 5 shows our evaluation results, which cover both applicability and performance. Applicability evaluation is carried out on the Vending Machine example as well as on an Ambient Intelligence system case study.

Chapter 6 reviews our work, emphasizing contributions and limitations and giving insight into future developments.

Appendix A is the appendix. It contains the whole SPL specification modeled for the Ambient Intelligence system case study presented to assess the applicability of our approach. It also includes the full featured game graph generated by our technique.

Chapter 2

Foundations

This chapter introduces the fundamental concepts and models used in this thesis. Section 2.1 presents Software Product Lines (SPLs), a method for the development of variability-intensive systems. Also Feature Diagrams (FDs), a common way to represent SPLs, are introduced there. Section 2.2 discusses Featured Transition Systems (FTSs), a formalism for describing the behaviour of an SPL. Next, Section 2.3 covers Modal Sequence Diagrams (MSDs), a scenario-based formalism for specifying the behavior of systems of components. Last, Section 2.4 describes the SPL specification, a recent model to specify the behaviour of an SPL as a combination of FDs and MSDs.

2.1 Software Product Lines

This chapter introduces Software Product Line Engineering (SPLE), an emerging software engineering paradigm dealing with variability-intensive systems.

A *product line* is a family of goods sharing a common, managed set of characteristics, usually referred as *common features* or *core assets*, and satisfying the needs of a particular market segment.

The invention of the production line is attributed to Henry Ford in the 1910's, when he conceived the idea of planning beforehand which parts would have been used in different car types. Thus he designed the Model T platform [Alizon et Al. 2009], a set of underbody core assets forming a skeleton from which several Model T models could be realised. The production of those tailored models was then outsourced to specialized companies. In that moment the way in which goods were manufactured switched from producing individual handcrafted goods to producing standard goods in much larger quantities. The Model T family has been one of the most successful products in automotive history for two reasons. First sharing manufacturing processes for common components

effectively reduced production costs, enabling a cheaper production for a mass market. Secondly, each model came with a deep level of customization taking into account the customers' requirements and giving them what they wanted. The notion of product line has been anchored in many industries ever since. Many companies took advantage from a product line approach, such as Airbus, Dell and even McDonald's.

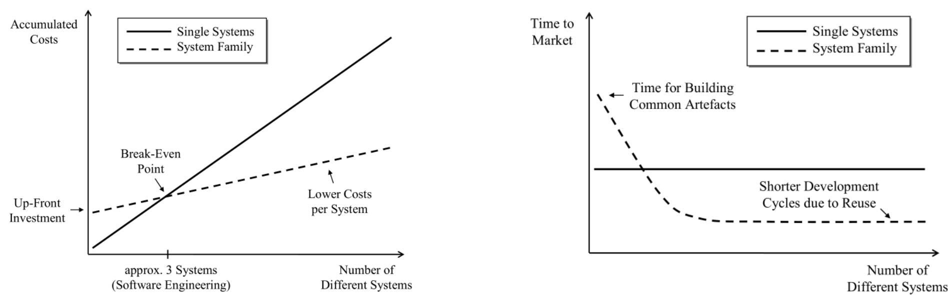
Unlike for manifacturing industries, the concept of product line in software engineering is quite anew, but rapidly emerging as a viable and important software development paradigm [Northrop 2002]. A *Software Product Line* (SPL) is a product line in which the set of products is represented by a set of software applications. [Clements and Northrop 2001] define an SPL as "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*". In this definition we can distinguish two main concepts: *variability* and *reusability*.

The definition of variability can be ambiguous. In [Pohl et al. 2005] variabilities of an SPL are both "*the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts*". [Coplien 1999], instead, refers to variabilities as "*the variations between* [...] *products*" of a product line. In this thesis we adopt this second definition, considering variabilities a synonym for differences. *Commonalities* denote common shared characteristics between products or, as stated in [Coplien 1999], "*an assumption held uniformly across a given set of objects*".

Another key concept is that of reusability. Reuse is achieved by developing a set of *core assets* representing the commonalities among all products in a product line.

Software Product Line Engineering (SPLE) is defined as "*a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation*" [Pohl et al. 2005]. A *platform* is any base of technologies on which other technologies or processes are built. This definition is strictly connected to reuse. By introducing common platforms, car manufacturers were able to increase their sales meanwhile reducing their production costs. This was done to accomplish an increasing demand for personalised products. "*Mass customisation is the large-scale production of goods tailored to individual customers' needs*" [Davis 1987].

The main goal of SPLE is to combine *platform* and *mass customisation* to produce customized products at reasonable cost. In particular we could outline four key motivations behind the PLE paradigm. These motivations are valid for PLE in general, then also for SPLE. A more complete list can be found in [Pohl et al. 2005, Pohl et al. 2001].



(a) cost for developing with and without PLE

(b) time to market with and without PLE

Figure 2.1: Effect of PLE on developing costs 2.1(a) and time to market 2.1(b) when the number of products in the product line increases.

Reduction of development costs Reusing platform artefacts significantly reduces cost as the common set of core assets is only developed once. To be available for developing specific products, common artefacts should yet be created beforehand and this implies an up-front investment. Thus, to reduce the costs per product, a PLE paradigm is needed to plan reuse methodically. When the number of products in the product line increases the costs per system adopting this paradigm are significantly lower than developing each single system independently. This is shown in Figure 2.1(a).

Reduction of time to market A second key advantage in adopting a PLE paradigm is the shorter time to market. An initial higher time for building common artifacts should be considered. Figure 2.1(b) illustrates how reusing platform artefacts reduces development cycles when the number of goods in the product line is high.

Increased quality Reuse not only leads to economies of scale, but also to an enhancement of quality. In fact common components are used in different products and tested in different situations supporting fault detection with a consequent improvement in the quality of the product line as a whole after error correction. Moreover, "*the same design techniques that lead to good reuse also lead to extensibility and maintainability over time*" [Coplien 1999]. Reuse can be indeed exploited to reduce maintenance effort by adopting reuse of test procedures for the core assets and possibly propagating error correction to all the products in the product line in which the modified component is being used.

Benefits for the customer All the motivations presented so far also produce

customers' satisfaction as they allow higher quality products at reasonable, even lower prices. Furthermore products in a product line generally support common functionalities (e.g. common user interfaces in case of SPL) and let customers reduce learning effort as well when dealing with a product derived from the same platform.

To pursue those goals [Pohl et al. 2005] also define a framework to systematically plan software reuse. We will introduce this framework in the next section.

2.1.1 The SPLE framework

In order to avoid undesiderable situations in which unmanaged reuse leads to higher costs and time to market than developing single systems from scratch, a methodic schema telling how common artefacts should be created and reused is needed. This section introduces the SPLE framework. Its principles and definitions mainly refers to [Pohl et al. 2005].

Software Development Life Cycles (SDLC) are structures defining a detailed methodology for the successfull development of a software product. Before the final deployment, SDLCs are traditionally composed of four main phases: requirements engineering, design, implementation and testing.

The SPLE framework takes his basic idea from [Weiss and Lai 1999], i.e. splitting the SDLC into two processes, *domain engineering* and *application engineering*, the former planning the development of the reusable core assets and the latter deriving single product line applications by reusing these common previously build artefacts. Both processes are compoused of several sub-processes which consist in an adaptation of the general SDLC main phases. The output of a sub-process of domain or application engineering is called *development artefact*. Note that although domain and application engineering have to be achieved one after the other, there is no mandatory sequential order in which their sub-processes should be performed, i.e. no mandatory waterfall model [Benington 1983] to be followed. Thus, to streamline the development of the sub-processes and their activities, any existing software development model is accepted, such as the spiral model [Boehm 1988] and the iterative and incremental development [Larman and Basili 2003]. The whole schema is depicted in Figure 2.2.

The domain engineering process is defined as follows.

Definition 2.1. (DOMAIN ENGINEERING)
Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realized.

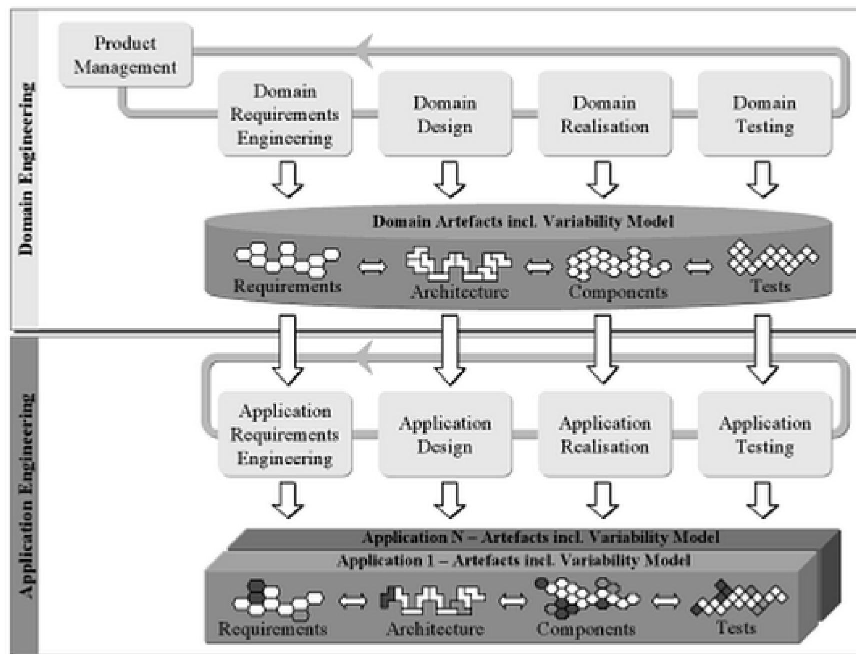


Figure 2.2: Schema of the SPLE framework [Pohl et al. 2005].

It can be seen as a further upstream life cycle which encorporates the concept of platform as defined in Section 2.1. In particular the platform represents the output of the whole domain engineering process and consists of all the domain artefacts, i.e. reusable development artefacts created in the sub-processes of domain engineering.

In the following, we briefly outline the different tasks of the several domain sub-processes as well as their domain artefacts.

Product management a starting up phase with a special regard to economic aspects. It defines the scope of the SPL, i.e. what should and should not be inside the product family. By analyzing the company goals it produces a *product roadmap* describing the set of features in the SPL and differentiating them into common reusable features and variable features that are application-specific in the sense that they are only part of some product.

Domain Requirement Engineering deals with requirement elicitation and documentation. It provides both common and variable requirements, i.e. requirements that are common to all applications and requirements that differ among several applications. It does not provide the requirement specification for a particular product as a whole. Its output artefact is the *domain*

variability model which defines commonalities and variabilities as well as variability dependencies and constraints.

Domain Design is responsible for the reference architecture of both reusable and application-specific components. Its output, the *domain architecture*, also defines the set of common rules guiding the realization of the different parts and the way they can be combined to form future applications.

Domain Realization provides the *domain realization artefacts*, i.e. the detailed design and implementation of reusable software components. Note that these implementations do not consist of runnable applications. Moreover their interfaces should consider and support many different contexts.

Domain Testing encompasses validation and verification. At this stage there is no running application to be tested in a traditional way, so testing only deals with the reusability of components, tested in different situations and scenarios. Its output, the set of *domain test artefacts*, focuses on allowing the large-scale testing reuse in application testing.

Thus, the main difference between domain and single system engineering is that domain artefacts are loosely coupled and related to partial components rather than specific final applications. The goal of this first process is to allow reuse in the following application engineering process, defined as follows.

Definition 2.2. (APPLICATION ENGINEERING)
Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability.

By exploiting commonalities and variabilities in the product line during the development of customer-specific applications, it allows mass customization as defined in Section 2.1. Similarly to the domain engineering process, it is divided into several sub-processes, each producing a set of *application artefacts*, one for every specific product line application.

Application Requirement Engineering produces the *application variability model* that documents variability bindings made for each single application. It is derived from the domain variability model, which is given as input. Possible adapted or newly introduced variants are also documented. Requirement elicitation is restricted by the already defined variability dependencies and constraints.

Application Design deals with the specific *application architecture*, adapting the structure of final products given the domain architecture and following the common defined rules. In this phase stakeholders should consider costs and efforts of adapting the structure and compare them to the ones relative to a development from scratch.

Application Realisation encompasses the implementation of the specific application by assembling reusable and application-specific components in the *application realisation artefacts*. Only part of these artefacts has to be developed. The implementation should yet comply the reusable interfaces created in the domain realisation artefact, which is given as input.

Application Testing validates and verifies applications against their specifications by reusing domain test artefacts given as input. This phase produces the *application testing artefacts*, which could include some additional tests for application-specific developments.

The main difference between application and single-system engineering is that application artefacts are not created anew but are derived from the platform, the output of the previous domain engineering process. A planned and systematic reuse of common core assets allows to complete application artefacts instead of creating them from scratch. Nevertheless more attention should be paid in conforming to reusable parts.

2.1.2 Variability modelling

In the context of software product lines engineering the commonality and variability between products are typically expressed in terms of features.

Many definitions of the term *feature* exist [Classen et Al 2008]. In this work we see features as an expression of the user's requirements and, according to the definition of [Kang et al. 1990], we consider them as a *prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system*. It generally represents either an optional added chunk of functionality or an aspect that cross-cut many components. It should be *user-visible* in the sense that it should be relevant to both stakeholders and engineers.

A *product* is a combination of features that describes a member of the product line. When dealing with a set of features S a product can also be defined as an element of the powerset $\mathcal{P}(S)$ which contains all the possible combinations of features. Among these combinations some are invalid as there may exist dependencies between the features. For example, two features can exclude each others, or a feature can require another one. To specify which combinations are valid, one commonly uses *feature diagrams* (FD), a family of modelling

languages that were first introduced as part of the *Feature Oriented Domain Analysis* (FODA) method in 1990 [Kang et al. 1990].

Feature Diagrams

In the following we describe the visual notations used throughout this thesis for the definition of features and their dependencies.

An FD is a combination of *nodes* and *edges* organized in a tree-like structure. The root node is called *concept* and represents the complete system.

Nodes graphically represent features, which may be classified as either *primitive* or *compound* features [Schobbens et al. 2006]. Primitive features are usually the ones that actually represent a product feature. Therefore enabling a primitive feature directly influences the nature of the derived product. On the contrary compound features, also called *decomposable features*, do not generally express product features by themselves, but are often used for decomposition purpose only. The selection of a compound feature influences the derivable product only with respect to the primitive features represented by their descendant nodes.

Nodes are related by means of edges. We can distinguish two types of edges: *decomposition* and *constraint* edges [Schobbens et al. 2006].

Decomposition edges hierarchically organize the FD nodes by relating parent feature with its child sub-features. In other words each relation specifies how a parent feature is decomposed into child sub-features. Intuitively a child feature can be chosen only if its parent is. Three types of decomposition exist:

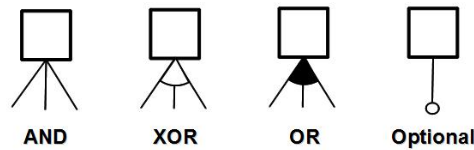
AND An AND decomposition requires that when a parent feature is chosen, also all its child features have to.

XOR A XOR decomposition expresses child features as alternatives. Thus exactly one of the child features must be enabled when the feature represented by its parent is.

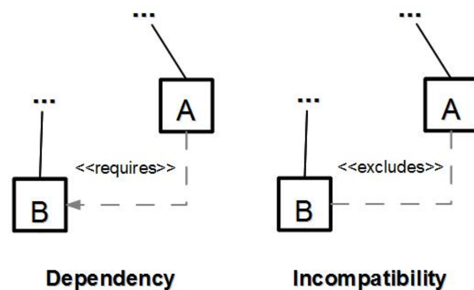
OR Within an OR decomposition every possible combination of the child features can be chosen, provided that at least one of them is.

In addition to AND, XOR and OR decompositions, *optional* relations are also available.

Optional It is also possible to express that a particular child feature may or may not be chosen. Optional relations are depicted as a solid line with a blank circle at the end connecting the parent feature to his optional child one. They can also be used within the other AND, XOR and OR relations.



(a) Decomposition edges



(b) Constraint edges

Figure 2.3: Types of edge in a feature diagram.

Figure 2.3(a) illustrates the graphical notation of the different decomposition edges. By enabling different features in the tree, different products can be derived.

It may happen that the enabling of some feature is allowed only if another feature is enabled as well. Two features can also be exclusive. In this case their combination should be forbidden. When many such constraints exist, decomposition edges are not sufficient to express all of them. *Constraint edges* , also called cross-tree constraint, fill this gap. There commonly exist two types of constraint edges:

Dependency If feature A depends on feature B, whenever feature A is selected, feature B must be selected as well. This relation is depicted as an arrow from feature A to feature B labelled with a «`requires`» statement.

Incompatibility If feature A is incompatible with feature B, whenever feature A is selected, we cannot select feature B. This is modelled by an edge between feature A to feature B labelled with an «`excludes`» statement.

Figure 2.3(b) illustrates the graphical notation of the different decomposition edges.

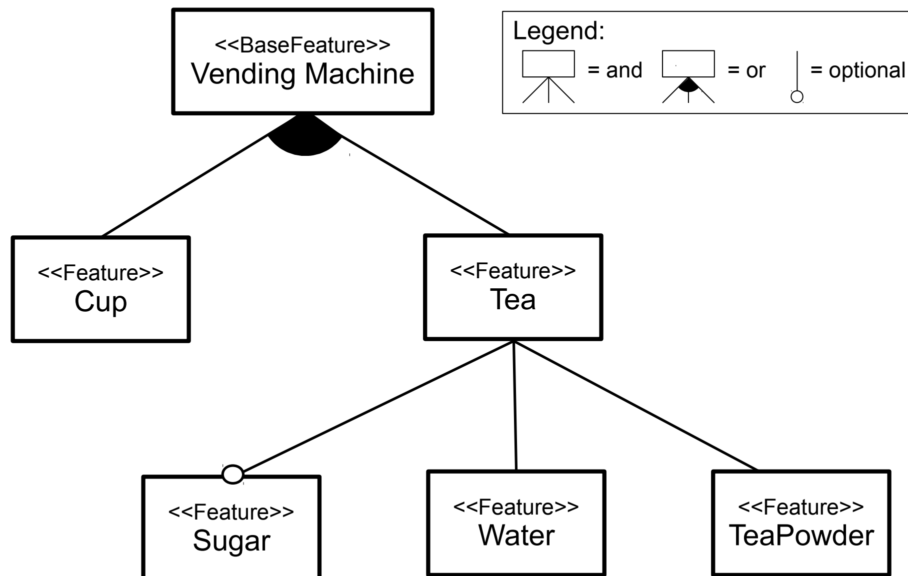


Figure 2.4: The FD capturing the set of valid products for the Vending Machine example

For the SPL of vending machines introduced in Section 1.1 five features can be readily identified: the sale of tea, the sale of cups, the ability to pour sugar, the ability to pour water and the ability to pour tea powder. Not every combination of those features yields to a valid system. For instance, the ability to pour tea powder makes sense only when considering the sale of tea. The set of products which are considered valid in the Vending Machine example are captured by the FD in Figure 2.4. It describes five product variants: the sale of tea with or without sugar, the sale of both cups and tea with or without sugar and, finally, the sale of cups alone.

Many different FD notations exist in literature, slightly different from the one described so far. For example the original [Kang et al. 1990] model dependency and incompatibility relations by means of textual constraint instead of constraint edges. Instead of drawing a special edge between features A and B as in 2.3(b), one could add the textual constraint «`A requires B`» or «`A mutex B`» to express dependency resp. incompatibility between features A and B. [Kang et al. 1998] propose to substitute the tree-like structure with a directed acyclic graph (DAG) structure allowing more than one parent for the same sub-feature. [Czarnecki et al. 2005] suggest cardinality-based feature models, in order to represent products with an arbitrary number of components. We remind the interested reader to [Schobbens et al. 2006] for a complete survey.

Formal definition and semantics of FD

Based on the abstract syntax definition due to [Schobbens et al. 2006, Schobbens et al. 2007], our notation can be formally defined as follows.

Definition 2.3. (FEATURE DIAGRAM)
An FD is a tuple $d = (N, P, r, \lambda, DE, CE)$, where

- N is the set of features;
- $P \subseteq N$ is the set of primitive features;
- $r \in N$ is the root feature. Only r has no parent: $\forall n \in N.(\nexists n' \in N.n' \rightarrow n) \Leftrightarrow n = r$;
- $\lambda : N \rightarrow NT$ labels each feature with an operator from NT, given $NT = \{and, or, xor, opt\}$.[1];
- $DE \subseteq N \times N$ is the set of decomposition edges, which must form a tree with r as its root;
- $CE \subseteq N \times GCT \times N$ is the set of constraint edges, given $GCT = \{requires, excludes\}$.[2];

We define a valid product as follows.

Definition 2.4. (VALID PRODUCT)
A valid product is any $p \subseteq N$ which

- contains the root: $r \in p$;
- the meaning of nodes is satisfied: $\forall n \in p$, with sons $s_1 ... s_k$ and $\lambda(n) = op_k$ then $op_k(s_1 \in p, ..., s_k \in p)$ must evaluate to true.
- the model must satisfy all graphical constraints: $\forall (n_1, op_2, n_2) \in CE$, $op_2 \ (n_1 \in p, \ n_2 \in p)$ evaluates to true;
- If s is in the model and s is not the root, one of its parents n must be in the model too: $\forall s \in p.s \neq r \exists n \in p : n \rightarrow s$.

The semantics of a feature diagram d, notated as $[\![d]\!]_{FD}$, is a Boolean formula that describes the set of valid products, i.e. the combination of features that satisfy the constraints expressed by the FD. Such a combination is actually a product line.

---

[1] NT (Node Type) is the set of decomposition operators

[2] GCT (Graphical Constraint Type) is the set of types of graphical constraint edges offered by the FD

2.2 Featured Transition Systems

a

In this chapter we present *Featured Transition Systems* (FTSs) a formalism presented in [Classen et al. 2010, Classen et al. 2011, Classen 2011] as an extension of *Labelled Transition Systems* (LTSs) to model the behaviour of variability-intensive systems.

2.2.1 Syntax and semantics of LTS

One of the more common models used to represent the behaviour of a single system are *Labelled Transition Systems* (LTS) [Baier and Katoen 2008, Cordy et al. 2013]. An LTS is a directed labelled graph in which vertices are states and edges are transitions between states. Transitions represent the capability of the system to perform a state change. A set of initial states represents the possible system configuration at lauch. Moreover they can be extended with one or two forms of labelling. Transitions are usually labelled with an action causing the state change, whereas states can be labelled with atomic propositions, i.e. assertions that are true in the states labelled with them. Formally, LTS are defined as follows.

Definition 2.5. (LABELLED TRANSITION SYSTEM)
A LTS is a tuple $lts = (S, Act, trans, I, AP, L)$, where

- S is a set of states;
- Act is a set of actions;
- $trans \subseteq S \times Act \times S$ is a set of transitions, with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$;
- $I \subseteq S$ is a set of initial states;
- AP is a set of atomic propositions;
- $L : S \to 2^{AP}$ is a labelling function.

In the rest of this thesis atomic prepositions are omitted from figures to avoid clutter.

An *execution* (also called run or behaviour) is an infinite alternate sequence σ of states and actions. More formally $\sigma = s_0\alpha_1 s_1\alpha_2...$ with $s_0 \in I$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$.

A *path* is an execution from which the information about the transitions has been removed. For $s \in S$, $paths(ts, s)$ denotes the set of all non-empty (potentially infinite) sequences $\pi = s_0 s_1 ...$.

The semantics of an LTS, noted $[\![ts]\!]_{LTS}$, is its set of paths. Formally:

Definition 2.6. (SEMANTICS OF LTS)
The semantics of a labelled transition system lts is its set of paths,

$$[\![ts]\!]_{LTS} = \bigcup_{s_0 \in I} paths(ts, s_0)$$

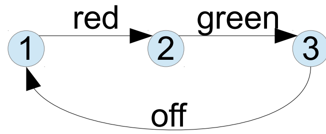
The semantic domain of an LTS is thus the power set of the (infinite) set of all possible (finite and infinite) paths.

2.2.2 Syntax and semantics of FTS

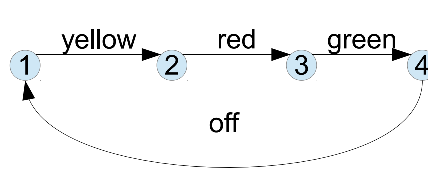
LTSs are a common and well-studied mathematical representations for the behaviour of single systems. Thus in the context of SPLs, an LTS is often used to represent the behaviour of a single product. When moving from single systems to SPLs, one can model the whole system family by modelling the behaviour of each product separately with a different LTS. However this sub-optimal approach does not scale to a large number of variants, because it does not take into consideration commonalities and variabilities between the various instances of the system. When modelling the behaviour of the whole SPL, a concise formalism is preferable in order to exploit the similarities between products.

Featured Transition Systems (FTSs) extend the concept of LTS to SPLs by making features first-class concepts of the formalism [Classen 2011]. Before giving a formal FTS definition, let us illustrate its basic workings through an example. Figures 2.5(a) and 2.5(b) show two conventional LTS modelling the behaviour of two different variants of a traffic light controller. The former represents a basic traffic light switching between red and green, whereas the latter shows yellow before switching to red. The SPL containing the two variants can be documented in an FD as shown in Figure 2.5(c). The ability to show yellow before red in the second variant can thus be represented by an additional optional feature `Y`. To model the behavioural impact of feature `Y`, additional states and transitions are required.

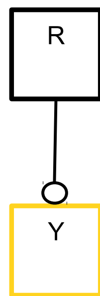
FTSs relate system behaviours to features by associating transitions with Boolean constraints defined over a set of features from an attached feature diagram. These constraints, called *feature expressions*, specify for which combinations of features a given transition is available. They are graphically represented as a second label displayed next to the action label. The two labels are separated by a slash.



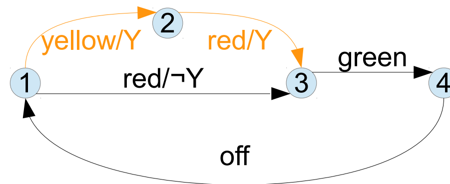
(a) LTS for the basic variant



(b) LTS for the variant containing feature Y



(c) FD of the two variants



(d) resulting FTS representing both variants

Figure 2.5: FTS of the traffic light controller

A feature expression is formally defined as follows.

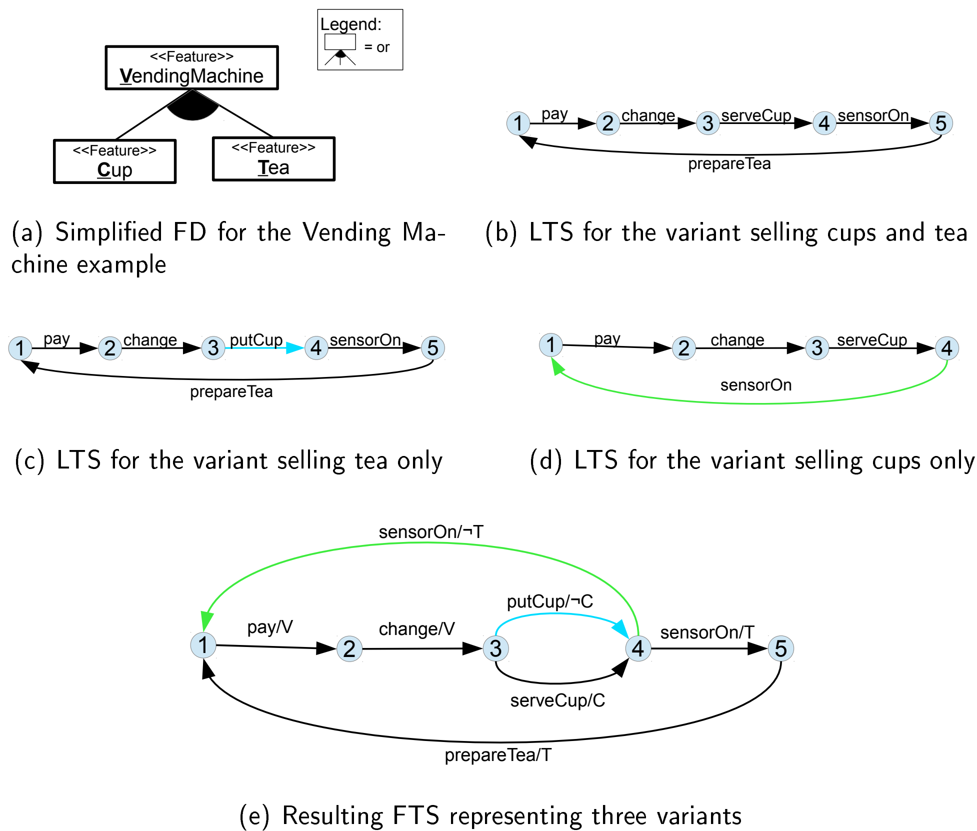
Definition 2.7. (FEATURE EXPRESSION)
A feature expression exp defined over a set of features F is a total function

$$exp : \mathcal{P}(F) \rightarrow \{\top, \bot\}$$

For a given product p, $exp(p)$ returns $\top$ if and only if the features of p satisfy the constraints expressed by exp. In this case, we say that p satisfies exp. We denote by $[\![exp]\!] \subseteq \mathcal{P}(F)$ the set of products that satisfy exp and by $\top$ the feature expression such that $[\![\top]\!] = \mathcal{P}(F)$ [Cordy et al. 2013].

A product p can execute an FTS transition if and only if its set of features satisfies the associated constraints.

The FTS shown in Figure 2.5(d), represents the behaviour of both variants of the traffic light example in a single and compact model. Transition ①→② is labelled with both an action, `yellow`, and a feature expression, `Y`, specifying that only products containing the «`Y`» feature can execute it. However additional features not only add states and transitions, but also remove them. The «`Y`» feature removes the ①→③ transition in the second variant. In the resulting FTS



(a) Simplified FD for the Vending Machine example

(b) LTS for the variant selling cups and tea

(c) LTS for the variant selling tea only

(d) LTS for the variant selling cups only

(e) Resulting FTS representing three variants

Figure 2.6: FTS of the Vending Machine example

this is done by labelling it with $\neg$Y. This way only one of the transitions leaving (1) can exist in a product. Features in an FTS can thus be non-monotonic, i.e. they can remove behaviours [Classen 2011]. In other words, the feature expression can also mean that a transition is available for all products except for those including a particular feature. Consider a simplified version of the Vending Machine example, whose set of valid products is represented by the FD in Figure 2.6(a). Figures 2.6(c) and 2.6(d) show the impact of removing features «Cup» and «Tea» from a machine serving both. In consequence, transitions colored in blue resp. green in the resulting FTS in Figure 2.6(e) are labelled with a feature expression representing the fact that they only belong to products without feature «Cup» resp. «Tea».

As shown in the previous examples, FTSs are LTSs extended with an additional labelling function and a feature diagram (FD) that defines the set of features and captures the set of valid products in the SPL [Classen et al. 2011]. Formally, FTSs are defined as follows.

Definition 2.8. (FEATURED TRANSITION SYSTEM)
An FTS is a tuple $fts = (S; Act; trans; I; AP; L; d; \gamma)$, where

- $S; Act; trans; I; AP; L$ are defined as in Definition 2.5;
- d is a feature diagram as defined in Definition 2.3;
- $\gamma : trans \rightarrow \mathcal{P}(F) \rightarrow \{\top, \bot\}$ is a total function, labelling each transition with a feature expression, i.e., a Boolean expression over the features.

The FTS formalism allows a scalable and concise modelling of each product in the SPL. When modelling with FTS the size of a model, measured in number of elements, increases linearly with the number of features; unlike the number of products which increases exponentially [Classen 2011].

It is possible to obtain the LTS modelling the behaviour of a particular product of the SPL by computing the *projection* of the FTS onto that product. This operation consists in removing all transitions of the FTS whose feature expression does not evaluate to true in the product.

Definition 2.9. (PROJECTION IN FTS)
The projection of an FTS fts to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the $LTS(S, Act, trans', I, AP, L)$ where $trans' = \{t \in trans | p \in [\![\gamma(t)]\!]\}$

Diagrams 2.5(a), 2.5(b) can be obtained from the FTS in 2.5(d) with the projections

$$(a) fts_{|\{r\}}, (b) fts_{|\{r,y\}}$$

Similarly LTSs in Figures 2.6(b), 2.6(c) and 2.6(d) can be derived from the FTS in Figure 2.6(e). Each LTS, obtained through projection from an FTS, represents the behaviour of a particular product of the SPL. Then the semantics of an FTS, noted $[\![fts]\!]_{FTS}$ is defined as a function with domain $[\![d]\!]_{FD}$ that associates each valid product with the semantics of the projection of the FTS onto that product.

Definition 2.10. (SEMANTICS IN FTS)

$$\forall p \in [\![d]\!]_{FD} \bullet [\![fts]\!]_{FTS}(p) = [\![fts_{|p}]\!]_{TS}$$

[Classen et al. 2010]

2.3 Modal Sequence Diagrams

This section describes Modal Sequence Diagrams (MSDs) and their semantics. The following definitions recall the ones from [Greenyer 2011, Harel and Maoz 2008, Maoz 2009, Harel and Marelly 2003].

Modal Sequence Diagrams (MSDs) are a variant of Live Sequence Charts (LSCs) [Harel and Marelly 2003], a graphical formalism to specify the requirements on the interaction between environment and system components.

We consider systems of objects that exchange *messages* and we call them *object systems*. An object system can be described by means of a *package* containing classes. Each object is an instance of a *class* and can both send and receive messages. Each class can have *operations* telling what are the types of message that any object instance of that class can receive. The top of Figure 2.7 shows the package `ServeTea` containing the classes `Student`, `Machine` and `Dispenser` with the respective operations.

A message in the MSD is identified by the operation set defined by the class of the receiving object and by the sending and receiving objects. The objects of the system are subdivided into *system objects* and *environment objects*. The middle of Figure 2.7 presents an example of an object system containing the environment object `stu:Student` and two system objects `mac:Machine` and `dis:Dispenser`. Two MSDs are shown at the bottom: `PutCup` and `PrepareTea`. Intuitively, the two MSDs express the following requirements. MSD `PutCup` specifies that if the dispenser receives the message `change` from the student, the student can send the message `putCup` to the Machine. If this happen, the Machine must send the message `sensorOn` to the Dispenser. The MSD `PrepareTea` specifies that if the Machine receives the message `sensorOn` from the Dispenser it must send the message `prepareTea` back to the Machine.

We name *event* the sending or receiving of a message by an object in the object system. In the following we assume only synchronous messages where the sending and receiving of a message is considered as a single event. Both system and environment objects can send and receive messages. An *environment event* in an event where the sending object is an environment object. Otherwise the event is a *system event*.

A *run* or *execution* is an infinite sequence of messages interchanged between both system and environment objects.

2.3.1 MSD activation, progress and termination

Every object in the object system that partecipates in the interaction is graphically represented by a *lifeline*. Each lifeline in the MSD represents an object. The link between the former and the represented object is given by the label of the lifeline.

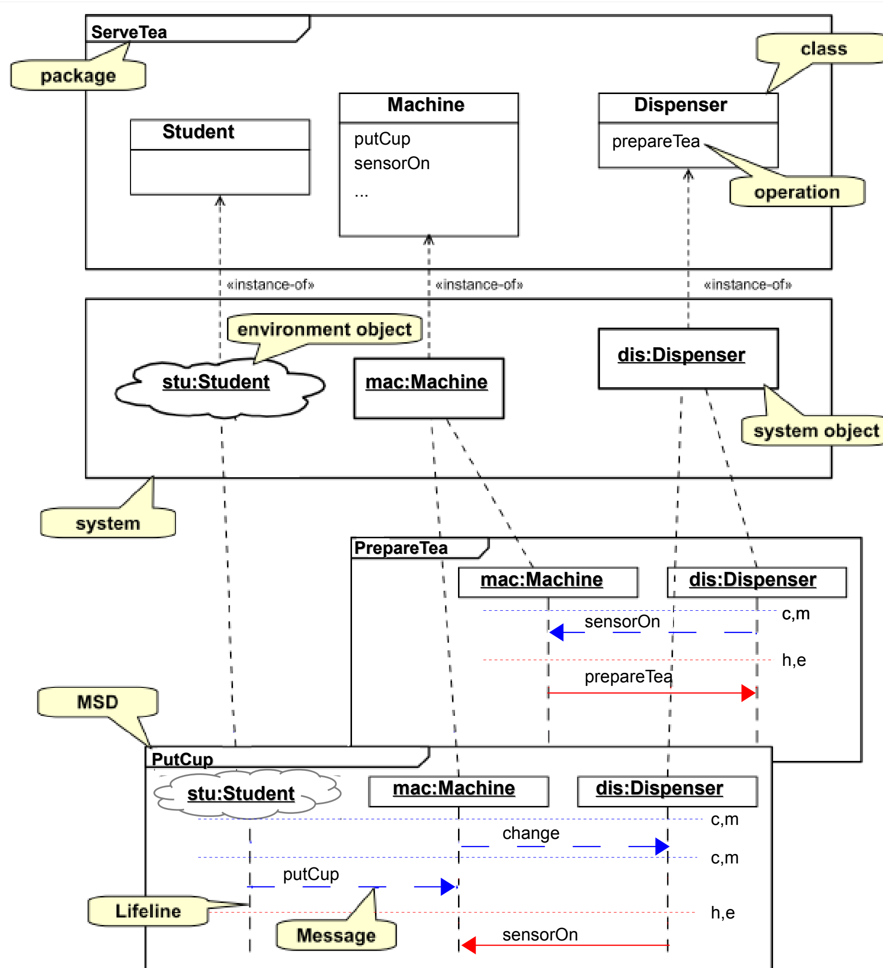


Figure 2.7: The object system and MSDs with concrete lifelines.

A lifeline is composed of a *head* containing the label and a *stem*, a vertical dashed line reproducing the timeline of the object [OMG UML 2011]. In an MSD the shape of each head depends on the nature of the object it represents. The head of a system object is depicted as a rectangle, as the head of the lifeline `msc:Machine` in Figure 2.7, whereas the head of an environment object is depicted as a cloud, as in the case of lifeline `stu:Student`.

An MSD *message*, also called *diagram message*, is represented as an arrow between the stems of two lifelines. The points on each stem where the arrows are attached are called *locations*.

An *MSD specification* is defined by a set of MSDs where each lifeline represents an object in a given object system and the exchanged messages represent

message semantics:	cold (can be “violated”)	hot (must not be “violated”)
monitored (may happen)	c,m	h,m
executed (must happen)	c,e	h,e

Figure 2.8: The possible combinations of temperature and execution kind for an MSD message.

the interaction between them.

A diagram message is *unifiable* with an event of the object system iff

1. the sending and receiving lifeline of the diagram message correspond respectively to the sending and receiving object in the object system and

2. the diagram message and the event in the object system refer to the same operation of the receiving object in the object system.

When an event is unifiable with the first message of the MSD, an *active* copy of the MSD, also called *active MSD*, is created. Intuitively the active MSD progresses as further events occur that are unifiable with subsequent messages in the run. The progress is captured by the set of passed locations. This set is called the current *cut*. The active MSD terminates when the cut reaches the end of the diagram.

The MSDs interpretation can be either *iterative* or *invariant*. In the first case only a single active copy of the same MSD can be created. Therefore the iterative interpretation does not allow the creation of a second active copy before the first one terminates. Conversely, in the invariant interpretation, multiple copies of the same MSD are allowed at the same time. In both interpretations, multiple different MSDs can be active at the same time. In this thesis, we assume an iterative interpretation.

2.3.2 Message attributes and violations

MSDs are subdivided into existential MSDs and universal MSDs. *Existential MSDs* specify sequence of events that must be possible to occur in at least one run, whereas *universal MSDs* represent requirements that must be satisfied by all occurring sequences of events. Within the scope of this thesis, we only consider universal MSDs.

MSDs specify not only sequence of events that may occur as in basic sequence diagrams, but also events that must or must not occur in a system. This is expressed by introducing a modality for the exchanged messages. Message attributes encode *safety* and *liveness* requirements for the occurring events. Safety is expressed by a *temperature* that can be either hot or cold. A *hot* message is represented by a red arrow and specifies that only the corresponding event must occur and no other events that we expect before or later. Conversely a *cold* message is represented by a blue arrow and specifies that also some other event in the referring MSD may happen. Liveness is expressed by an *execution kind* that can be either *monitored* or *executed*. A monitored message is represented by a dashed line and specifies that the corresponding event may be observed whereas an executed message, represented by a solid line, specifies that the corresponding event must eventually happen.

The four possible combinations are summarized in Figure 2.8, whereas the two MSDs in Figure 2.7 show possible uses for both hot, executed messages and cold, monitored messages.

While the MSD progresses, when the cut of the active MSD is immediately before a message, that message is enabled. Therefore an *enabled message* represents the next event in the ongoing MSD. The cut also has the same safety and liveness attributes of the MSDs messages. In particular, the attributes of the enabled message define the ones of the corresponding cut. For example if the enabled message is hot and executed the cut that enables the message is also hot and executed. If more than one message is enabled at the same time the attributes of the cut are derived as follows. If one of the enabled messages is hot, the cut is also hot, whereas if all the enabled messages are cold, the cut is cold. Similarly if one of the enabled messages is executed, the cut is executed, while if all the enabled messages are monitored, the cut results to be monitored. When the cut is executed the executed enabled message is also called *active message*.

Back to the MSDs shown in Figure 2.7, the nature of the cut before each message is expressed by a dotted blue or red line respectively indicating a cold or a hot cut. In addition, a label next to the dotted line indicates whether the cut is cold and monitored (c,m) or hot and executed (h,e) which are the only two combinations considered in this example. Supposing that the MSD `PutCup` is active and messages `change` and `putCup` have been sent, `sensorOn` is enabled. In this case, since it is an executed message, `sensorOn` is also active. Its sending causes the MSD `putCup` to terminate and, at the same time, it triggers the creation of the MSD `prepareTea`.

A *safety violation* or *hot violation* occurs when the enabled message is hot, but another event happens instead which is unifiable with a message in the same MSD that is not currently enabled. For example, in MSD `PutCup` there would be a safety violation if either message `change` or message `putCup` was sent instead of

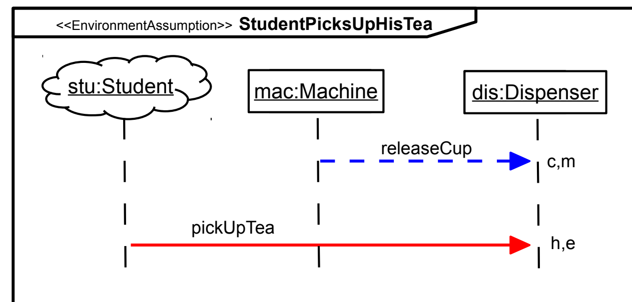


Figure 2.9: Assumption MSD for the Vending Machine example.

message `sensorOn` when active. If the same happens when the enabled message is cold, it is called a *cold violation*. Safety violations are never allowed and all the runs in which they occur are never accepted by the MSD. Conversely cold violations are allowed, but the active copy of the MSD in which they take place is discarded. A *liveness violation* occurs when the cut is executed, but the event unifiable with the active message never happens. Similarly to safety violations, liveness violations are never allowed. No violation takes place if the same happens in a monitored cut.

2.3.3 Environment assumptions

Even though environment events are uncontrolled, one may often make assumptions on the behavior of the environment [Zave and Jackson 1997]. For instance, in our Vending Machine example, one could make the assumption that a student always takes his cup away from the dispenser when his tea is ready.

Environment assumptions can be expressed by *assumption MSDs* that together with requirement MSDs form an MSD specification. Assumption MSDs are used both to describe a possible environment behaviour and to define how the environment reacts to system events. They are graphically represented by adding a stereotype annotation «`Environment Assumption`» in the MSD label. Figure 2.9 illustrates an assumption MSD for our Vending Machine example. By this MSD we assume that when the Dispenser releases the cup containing the tea, the Students picks it up.

Environment events are classified as *spontaneous* or *non-spontaneous* events.

Intuitively a non-spontaneous event only occurs as a reaction to other system or environment events. Otherwise if an event can happen independently from other events, it is a spontaneous event.

2.3.4 Parameterized and forbidden messages

Further extensions has been made to enrich MSDs and diagram messages. We focus more particularly on parametrized messages, MSD conditions and forbidden messages.

Parameterized messages

An MSD can contain messages with parameters. Here we only consider messages that can carry at most one parameter. *Parameterized messages* imply the need for extending *unifiability*. A parameterized message is *parameter unifiable* with an event of the object system iff

1. the sending and receiving lifeline of the diagram message correspond respectively to the sending and receiving object in the object system and
2. the diagram message and the event in the object system refer to the same operation of the receiving object in the object system and
3. the parameter of the message carries either an undefined value or a value that equals the one carried by the event.

Back to our Vending Machine example, instead of using the message `releaseCup`, as in the assumption MSD in Figure 2.9, we could use a parameterized message with a boolean parameter, e.g. `releaseCup(true)`. This way we could use the same parameterized message with a different parameter value to express that the cup cannot be taken away. Figure 2.10 shows an MSD with a parameterized message. It specifies that after the Dispenser sends message `sensorOn` to the Machine, informing that a cup has been put in the Dispenser, the Machine must send the message `releaseCup(false)` back to the Dispenser, indicating that the cup should be blocked.

Conditions

The progress of an MSD can even be decided explicitly by means of MSD conditions, provided with a condition expression and a temperature. The condition expression is a Boolean-valued expression written in the Object Constraint Language (OCL) [OMG OCL 2012].

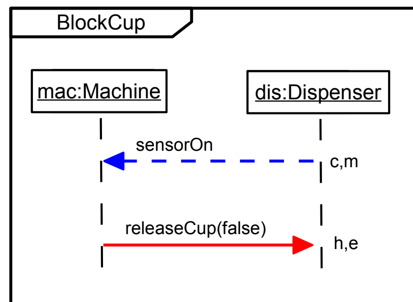


Figure 2.10: MSD with a parameterized message for the Vending Machine example.

From a graphical point of view they are represented as hexagons containing the condition expression. They can cover one or multiple lifelines. The border of the hexagon may be either blue and dashed or red and solid if the MSD condition is respectively cold or hot.

As for diagram messages, a condition is *enabled* when the cut of the active MSD is immediately before the condition. If the condition spans multiple lifelines this has to be true for all the lifelines covered. Conditions are evaluated as soon as they are enabled. In both cold and hot condition cases, if the expression evaluates to true, the cut of the MSD progresses. If an enabled cold condition evaluates to false it leads to a cold violation and the active copy of the MSD is discarded. Instead, if a hot condition valuates to false the MSD is stucked and cannot progress until the condition valuates to true. Therefore if a hot condition never evaluates to true, the MSD never progresses leading to a liveness violation.

Forbidden messages

By mean of *forbidden messages*, we can determine events that should not happen. In Section 2.3.2 we specified that hot and cold violations will occur if an event occurs in an executed cut and is unifiable with a message of the same MSD that is not currently enabled. Besides, sometimes, we would like to indicate that even events that are not part of the MSD are forbidden to occur while the MSD is active.

Graphically, forbidden messages are labelled with a «`forbidden`» stereotype and specified at the end of the MSD in which they are forbidden to occur after

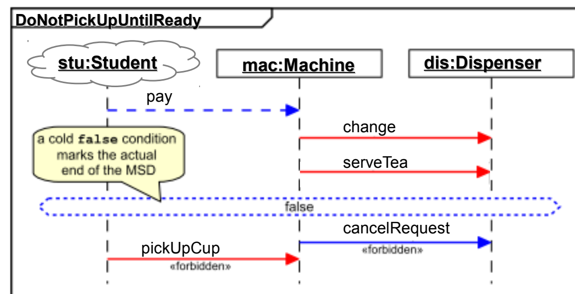


Figure 2.11: MSD with a forbidden message for the Vending Machine example.

a cold false condition covering all the lifelines.

Forbidden messages have a temperature, but no execution kind. When a cold forbidden message is sent during the MSD progression, a cold violation occurs and the MSD is terminated. Thus a cold forbidden message could be used for example to express the fact that a certain order of events should be kept after the event that triggers the particular MSD occurs, except in the case that the forbidden message is sent anytime after the MSD activation and before his normal termination. Consider the MSD in Figure 2.11. If the forbidden message `cancelRequest` happens after message `pay` and before message `serveTea`, for example after message `change`, the MSD terminates with a cold violation.

When a hot forbidden message is sent during the MSD progression, a hot violation occurs instead. As already stated, hot violations are never permitted. A hot forbidden message is always used to specify that if a certain event happens meanwhile the MSD progresses, it will lead to a safety problem in the system.

Again in Figure 2.11 if the forbidden message `pickUpTea` happens before message `serveTea` the MSD terminates with a safety violation.

2.3.5 The play out Semantics

The play-out algorithm was originally proposed as an operational interpretation for universal LSCs [Harel and Marelly 2002, Harel and Marelly 2003, Maoz and Harel 2006], but it can be extended to universal MSDs.

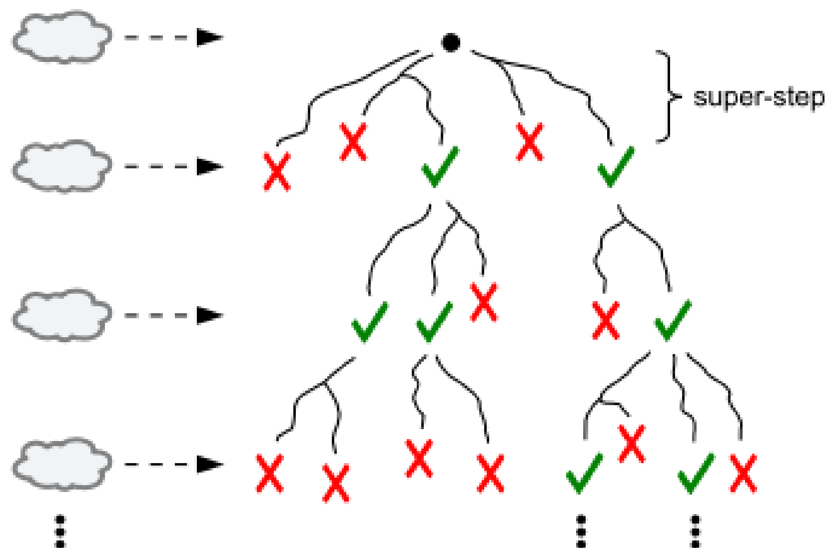


Figure 2.12: Valid and violating super-steps

To determine which message and object sends after a certain sequence of events in the object system one could implement a *controller*. This controller can be non-deterministic. The basic assumptions of the play-out semantics are that instead of implementing a controller for the system objects, we just interpret the MSDs/LSCs, which tells which messages can, must or must not be sent by system objects given a certain sequence of already happened event [Greenyer 2011].

The system starts listening for the first environment event to occur. If this event is unifiable with the first message of an MSD in the MSD specification, then an active copy of the respective MSD is created as already explained in Section 2.3.1. As further unifiable events occur, the active copy progresses and also other MSDs copies could be activated. Normally more than one MSD is active at the same time and more than one of them is in an executed cut. As a consequence more than one event is active, i.e. both enabled and executed. The play-out algorithm chooses between this set of active events non-deterministically in such a way that the chosen event would not lead to a violation in any other active MSD. Then it executes this event by ordering the particular responsible sending object to send the corresponding message. When no more active MSD in an executed cut is left, the set of possible active messages to send is empty, thus the algorithm stops sending messages and waits for the next environment event to occur. Then the process is carried over.

We define *step* the sending of a message in the play-out algorithm, whereas

a *super-step* is the set of messages interchanged between when the algorithm listens for an environment event and when it waits for the next one. In Figure 2.12 supersteps are depicted as wiggly black lines, whereas clouds represent the environment events that occur.

It may happen that the algorithm cannot progress because all the active events are forbidden to occur, since any of them would lead to a safety violation in another MSD. This problem typically arises when the MSD specification is inconsistent. Still, the play-out algorithm could even get stuck if the specification is consistent. The successful termination strictly depends on the order according which the algorithm chooses the messages to be sent. In the case of a consistent specification, safety violations can be avoided choosing a proper order of execution of the active events. The only way to find the right order of system events to execute would be to know in advance which sequence of messages would avoid a safety violation. Since unfortunately the play-out algorithm cannot *look ahead* in his process, hot violations are still possible to occur even if the specification is consistent.

An enhanced version of the algorithm, called *smart play-out*, was proposed by [Harel et al. 2002]. In their work they introduce the ability to look ahead one super-step. Although the improved algorithm can skip certain bad sequences of events, it has been proven that even in this case not all the violations can be bypassed, since the smart play-out can still choose a valid super-step which will lead to a hot violation in further super-steps. Figure 2.12 represents a set of valid and violating supersteps which are respectively depicted as green check symbols and red Xs. Suppose that the algorithm is in its initial state, depicted as a black dot. The only sequence of super-steps by which the play-out algorithm could avoid hot violations is the one on the right. However, since the smart play-out has the ability to only look ahead one super-step, it could also choose to execute another valid super-step on the left and then inevitably run into a hot violation in the third super-step. Thus some of the violations can be avoided, but not all of them.

2.3.6 Satisfiability, consistency and consistent executability

We already assumed synchronous messages, i.e. messages where the sending and receiving is considered as a single event. In order to define MSD satisfiability, one more assumption is required: although we consider the system to be able to send as many messages as it needs to before the next environment event happens, we also have to forbid runs with infinite sequence of system events. This means that the system should be faster than the environment when needed, but it also

must "listen" to environment events infinitely often. Thus a run *satisfies* an MSD specification iff:

1. it is accepted by all requirement MSDs in the set or
2. it is not accepted by at least one assumption MSD in the set and
3. it does not contain an infinite sequence of system events.

A run is accepted by an MSD if neither safety nor liveness violations occurs in the MSD.

An MSD specification is *consistent* iff it is possible for the system objects to react to every possible sequence of environment events so that the resulting run satisfies the MSD specification.[Greenyer et al. 2011] In this case it is possible to find a controller for the system able to satisfy the MSD specification when combined with any possible behavior of the environment (respecting the nonspontaneous events assumption).

So far we have considered that the system can also send messages that are not currently active. This possibility is not generally accepted from engineers, since no MSD states that these messages should occur [Greenyer 2011]. A more restrictive definition of a controller named *consistently executing controller* fills this gap. A consistently executing controller is a controller that only sends messages that correspond to active events. An MSD specification is *consistently executable* iff it is possible to find a consistently executing controller for the system that satisfies the MSD specification when combined with any possible behaviour of the environment, respecting the nonspontaneous events assumption. In this case we also say that there exists an *admissible strategy* for the play-out algorithm.

Consistent executability is a stronger property than consistency. If the play-out algorithm can satisfy the MSD specification, the specification must be consistent, whereas it could still not be possible to find a strategy for an MSD specification even if the specification is consistent. Besides, the play-out algorithm can always find a strategy for an MSD specification that is consistently executable.

2.4 Formal scenario-based specification of SPLs

Modern software-intensive systems usually consist of a set of components which interact to allow the achievement of their functionalities. Dealing with variability-intensive systems adds more complexity, since different interactions of components have to be considered. In order to cope with that complexity an intuitive, yet precise way to design those systems is required.

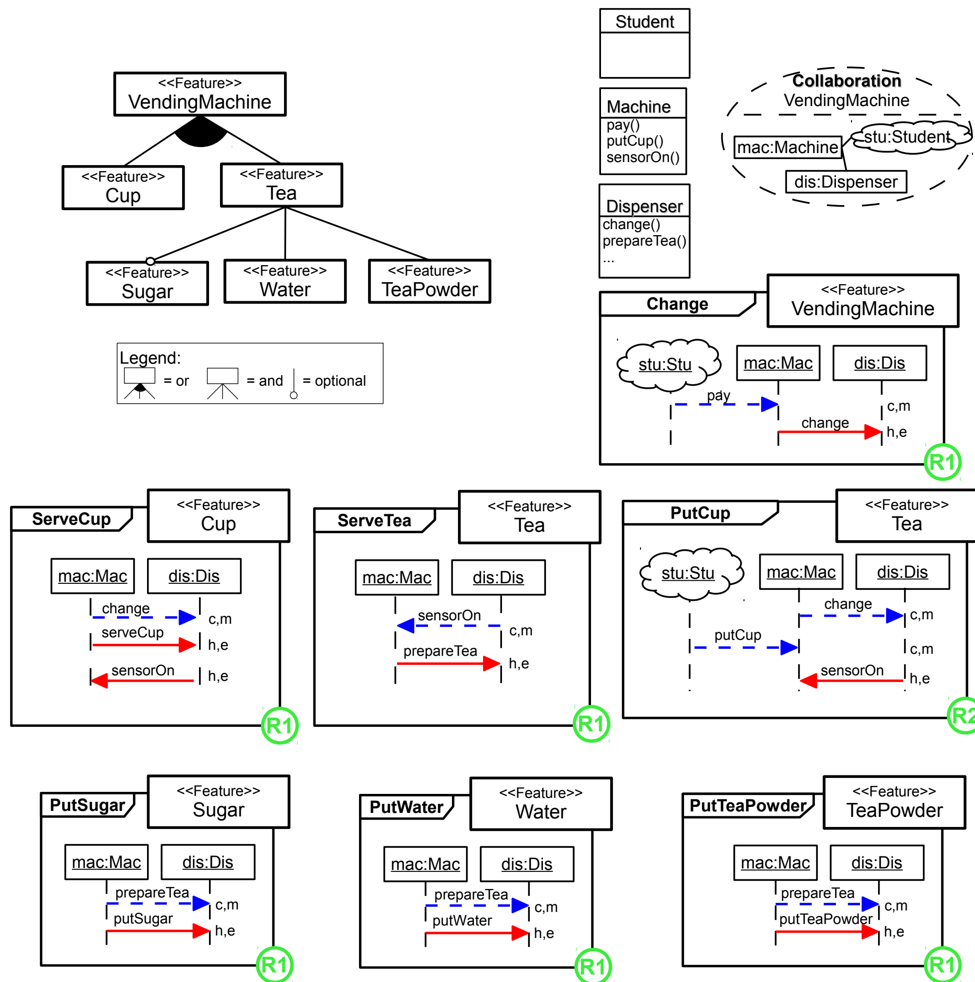


Figure 2.13: The SPL specification for our VendingMachine example.

This section presents a scenario-based approach to design product lines recently proposed by [Greenyer et al. 2011]. This model will be referred to as *SPL specification* in the rest of this thesis. It provides a precise specification of the interaction behaviour of components in product lines by combining FDs and MSDs together.

In an SPL specification each feature in an FD can be associated with a *feature specification* consisting of a package containing one or multiple MSDs. A feature specification describes the behavioural aspects of the single feature independently from the others.

Figure 2.13 shows the SPL specification for our VendingMachine example. Note that not all of the MSDs presented in the previous section are included in the

specification. The feature specification for Feature «Tea» represents the ability to pour tea in a cup and involves the two MSDs «ServeTea» and «PutCup». Apart from feature «Tea», all other features' specifications are composed of only one MSD.

The major advantage of using MSDs when modeling the behaviour of a family of products, is that the whole MSD specification for a product can be naturally obtained as the union of the MSD specifications of the features comprising the product. This implies that in a SPL specification the requirement and assumption MSDs for a given product can be composed by combining the requirement resp. assumption MSDs specified for its constituent features.

For the VendingMachine example only requirement MSDs are involved. The whole FD represents a set of 5 valid products. For simplicity consider only features «VendingMachine», «Cup» and «Tea» and the three products which can be derived from them. Consider product {{VendingMachine}{Tea}}. Its MSD specification is given by the MSDs «Change», «ServeTea» and «PutCup». The first MSD specifies that when the student pays for a product, the machine returns some change. MSD «ServeTea» specifies that after the sensor turns on, meaning that a cup is in the dispenser, the tea can be prepared and poured into the cup. Finally MSD «PutCup» means that after change is returned, the student *could* provide a cup in which his tea should be poured. As a consequence the sensor of the vending machine turns on. This is because the sensor notices the presence of the cup that has been put into the dispenser. The union of those three MSDs forms the MSD specification for the considered product and express the ability of the vending machine to return change after the students pays for a tea, then wait for the student to put a cup into the dispenser, turn on the sensor after noticing the presence of a cup and finally prepare a tea to be poured into the cup. Consider now feature «Cup». Since «Cup» and «Tea» are in an OR relationship, product {{VendingMachine}{Tea}} can be extended with feature «Cup». This feature alone represents the fact that after change is returned, a cup is put in the dispenser by the machine. Then the sensor, noticing the presence of a cup in the dispenser, turns on. But when combined with product {{VendingMachine}{Tea}} this feature contributes to the whole specification of the vending machine expressing its ability to first put a cup in the dispenser and then pour tea into it. In this case there is no need for the system to wait for the student putting a cup into the dispenser, because cups are dispensed directly by the vending machine.

An MSD specification typically also includes the description of an object system. When dealing with SPL specifications, instead of including one for each feature, only one definition of object system can be assumed for all products and features for simplicity [Greenyer et Al. 2013]. This is done by including a class diagram and a collaboration diagram in the feature specification of the

root feature. A collaboration diagram is a *UML composite structure diagram for describing instances that collectively accomplish some desired functionality* [OMG UML 2011]. The nodes in this diagram, called *roles*, represent an object in the object system and are typed over the classes in the class diagram. Operations listed for each class correspond to the messages that a role can receive when interacting with another role. Interactions between roles, i.e their ability to exchange messages, are represented by edges in the collaboration diagram. A role can represent a system or an environment object. This is modeled by a stereotype on the roles and graphically depicted with a rectangle resp. cloud shape.

In the example in Figure 2.13 those additional diagrams are included in feature «VendingMachine». When an object is not part of a particular feature or product there is simply no interaction with it. For example Student is not part of the feature specification for «TeaPowder» as this is something only related to putting some tea powder in the cup when preparing tea and does not involve any possible interaction with the student who ordered the tea.

The SPL specification provides an intuitive way to derive the behaviour of each product as the union of the behavioural aspects of its composing features. Modeling the specification of each feature with a package allows the engineer to employ the *package merge* mechanism [OMG UML 2011]. Package merge consists in merging the contents of one or multiple packages into another package. In this case it allows packages representing single feature specifications to be merged into a *consolidated product specification package*, i.e. a package containing all MSDs from each feature specification. All those MSDs can be then composed to form the MSD specification for the product.

The SPL specification constitues the input of our approach to synthesize SPLs, as described in the next chapters.

Chapter 3

Featured Synthesis

This chapter presents our approach for synthesizing specifications of component interactions based on a combination of Modal Sequence Diagrams and Feature Diagrams. The realizability-checking problem can be seen as the problem of finding a strategy in an infinite game played by the system against the environment. Our technique consists of an extension of the algorithm for solving Büchi games which are games requiring to infinitely often reach a state with given characteristics.

Section 3.1 introduces our general approach and how the presence of features impacts our algorithms. Section 3.2 explains the algorithm for solving reachability games for variability-intensive systems and its main subroutines. Finally the Featured Büchi algorithm extension, which is based on the the Featured Reachability algorithm, is described in Section 3.3.

3.1 The approach

The problem of checking the realizability of a given specification can be seen as a two-player game between the system and the environment, which has to be won by the system.

The game appears in the form of a *game graph*. A game graph is an LTS in which every state corresponds to exactly one cut in every MSD that is part of the specification. The initial state corresponds to no active MSDs. A move from the system (resp. the environment) in the game models a message sent by the system (resp. the environment), thereby advancing the cut of one or more MSD in the specification. Transitions represent one among the admissible moves given a particular state, i.e. one among the active events in the MSD specification. They are labelled with the name of the event they represent. Since the system cannot force environment events to happen, from the system's perspective, his

transitions are controllable whereas the environment's transitions are seen as uncontrollable. Given the hypothesis that the system can either send a message or wait for the next environment message, all transitions leaving a given state are either controllable or uncontrollable.

A *strategy* [Maler et Al 1995] is a function that during the course of the game defines which move the system should execute. In other words for each state in the game graph it restricts the set of admissible moves in order that all the remaining runs of the system meet certain criteria [Pnueli et Al 1998]. A strategy is *winning* if it guarantees that the system wins (according to a given definition of winning) no matter what the environment does [Pnueli et Al 1998].

A product in a SPL is realizable when the specification composed of the MSDs of its features is consistent. That could be checked on the corresponding game graph verifying whether a strategy exists for the system to reach a *goal state* infinitely often regardless of what the environment does. A goal state is a state satisfying the following condition, also called *goal condition*.

1. no safety violation occurred in any requirement MSD and
2. no active event remains in any active requirement MSD or
3. a safety violation occurred in an assumption MSD or
4. there is at least one active event in an active assumption MSD

Games with a winning condition requiring to infinitely often visit goal states are called *Büchi games* or games with Büchi objectives.

Checking whether products in a SPL are realizable could be done by checking the consistency of the different specifications separately. Our approach represents an alternative to performing an individual synthesis for each variant. Since products in an SPL are similar, their game graphs are very likely to share states and transitions. By exploiting the commonality between products, our technique provides a way to synthesize all products simultaneously, i.e. by performing only one synthesis. In order to do that, we need to link different possible behaviours for the system with the features needed to trigger those behaviours.

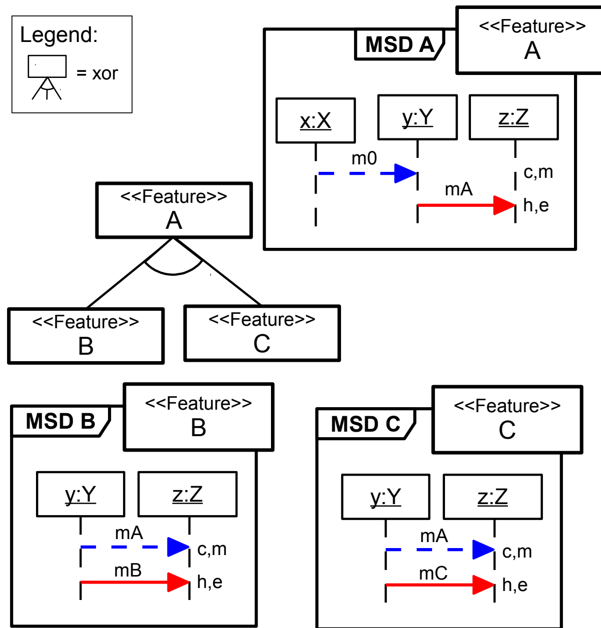
We introduce the idea of the *featured game graph* as an extention of the game graph. A featured game graph is a game graph in which states still represent particular cut configurations of the set of active MSDs whereas transitions (called *featured transitions* hereafter) contain additional information about the features needed to trigger them. Thus a featured transition not only corresponds to a move given a particular state, but also the particular subset of features in the SPL for which the move is admissible. Depending on the features involved, the set of MSDs which may be activated can change. Also, depending on the product, the

same event may advance the cut of the MSDs in the specification in a different way. Thus given a state in the featured game graph, different featured transitions for the same event may lead to different successors. When such a situation takes place, i.e. every time the same event leads to n different successors, n featured transitions exiting from the same state are generated, each labeled with a feature expression representing the combination of features for which the event advances the cut as in the corresponding target state. Otherwise, when there's no ambiguity and only one successor exists for the considered event, featured transitions are labeled with the features corresponding to the newly activated MSDs. If no new MSD is activated, and the event just advances the cut of the MSDs that are already active in the source state, the feature expression is a «True» statement.

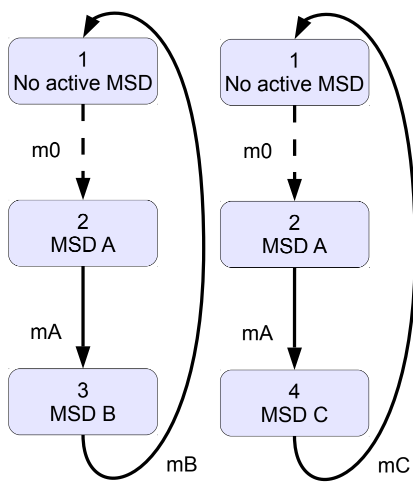
Suppose we have a set of two products {{A,B}{A,C}} as represented by the SPL specification in Figure 3.1(a). The corresponding game graphs are given in Figure 3.1(b). In this example the event «mA» corresponds to two different transitions, depending on the product that is considered, i.e. depending on if either feature B or feature C is present. This information is embedded in the featured transitions of the featured game graph shown in Figure 3.1(c). by means of the feature expressions «B$\wedge$!C» and «!B$\wedge$C». «A» labels the feature transition from state (1) to state (2) since the MSD in feature A is activated by the event «m0». The other featured transitions are labeled with a «True» statement.

A featured game graph is a concise representation for all possible execution scenarios of the system in a SPL. The information contained in the featured game graph is then used by an extension of the algorithm for solving Büchi games that determines the set of products having a consistent specification.

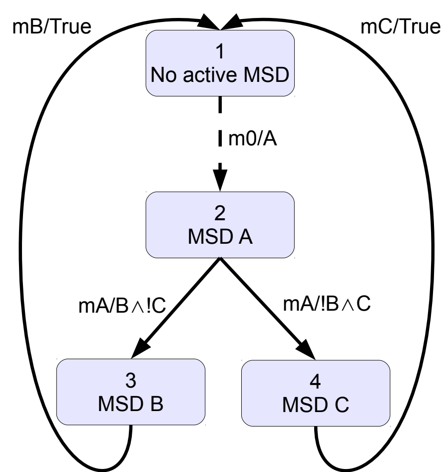
Details about how to pursue that target and the major novelties introduced are discussed in the next sections.



(a) SPL specification for two products {{A,B}{A,C}}



(b) Two game graphs



(c) One featured game graph

Figure 3.1: Featured game graph example

3.2 Featured Reachability

A reachability game consists of finding a strategy for the system to eventually reach a goal state regardless of what the environment does and the uncontrollable transitions it takes.

Before presenting our extension to solve reachability games when dealing with variability-intensive systems and featured game graphs, we describe the fundamental subroutines used by the main algorithm

The rest of the Section is organized as follows. Section 3.2.1 explains how the semantics for the set of valid products is generated from the FD. That information is considered when generating successors for a given state in order to avoid the unnecessary evaluation of featured transitions corresponding to combination of features that does not respect the contraints imposed by the FD. The technique to generate successors and featured transitions is illustrated in Section 3.2.2. The backward propagation step and the computation of the winning condition are discussed in Section 3.2.3. Finally Section 3.2.4 presents the main algorithm.

3.2.1 Retrieving the FD expression

In order to synthesize the set of valid product variants simultaneously, we need to derive a concise feature expression representing the product line from the feature diagram. In particular that expression should represent the semantics of the FD, i.e. the set of valid product variants or feature combinations that fulfill the FD contraints. Those products are not yet the realizable ones, since inconsistencies that may arise from their specification are not yet considered. Still, since realizable products are always valid products as well, this formula will be used during the reachability step to avoid exploring unnecessary transitions related to not valid products, thus enhancing the algorithm efficiency.

The algorithm is shown in Algorithm 1. A Boolean formula fd is derived incrementally following a top-down exploration of the FD according to the following rules.

1 **Mandatory root.** First, since the root is always mandatory, the root feature is added to the formula (line 1).

2 **Parent implies required children.** For each feature that is not a leaf in the FD tree we should state that the presence of that feature in a product implies the one of its directly required subfeatures. In order to do so we inspect the constraints imposed by the way the parent feature is decomposed into child features, i.e. its decomposition kind. In case it is an AND, all its mandatory child features are required. A feature is mandatory when

Algorithm 1 FD

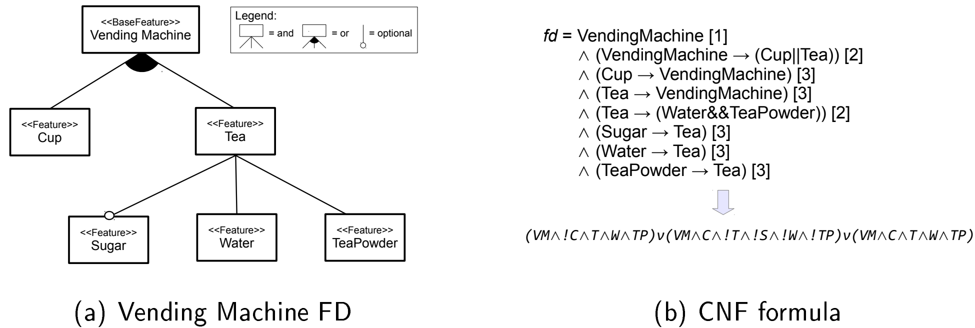
```
1:  fd ← root;
2:  for all feature do
3:      if (feature! = leaf) then
4:          switch (decompositionKind(feature)) do
5:              case AND
6:                  childDecomposition ← ⋀_{i:childFeature_i != OPTIONAL} childFeature_i;
7:              case OR
8:                  childDecomposition ← ⋁_i childFeature_i;
9:              case XOR
10:                 childDecomposition ← ⊕_i childFeature_i;
11:         fd ← fd ∧ (feature ⇒ childDecomposition);
12:         for all childFeature do
13:             fd ← fd ∧ (childFeature ⇒ feature);
    return fd;
```

it is not labelled as optional. We derive the $childDecomposition$ subformula as the cumulative and of all the mandatory children of the considered feature. In case of an OR every possible combination of the child features can be chosen, provided that at least one of them is. If it is a XOR decomposition one and only one between its child features is required. In those latter cases the $childDecomposition$ subformula is computed as a cumulative OR resp. XOR of the feature's children. Once we derive that subformula, we add to the main formula an implication from the parent feature to the respective childDecomposition (lines 2-11).

3 Child implies parent. A child feature can be chosen only if its parent is. For each child feature we add an implication from the child feature to its parent to the main formula fd (lines 12-13).

The returned formula can be easily translated in *Conjunctive Normal Form* (CNF) for readability purpuses. A CNF formula is a conjunction of clauses, where each clause is a disjunction of literals, and each literal is either a positive or negative propositional variable.

For the Vending Machine FD in Figure 3.2(a) our technique first adds the root «VendingMachine» to the fd formula. Then it considers the features directly required by the root. «VendingMachine» can choose «Cup», «Tea» or both. The formula «VendingMachine»→(«Cup»∨«Tea») is added to the main



(a) Vending Machine FD (b) CNF formula

Figure 3.2: Retrieving the Boolean formula specifying the valid products from the FD

formula as required by the second rule. Then also «Cup»→«VendingMachine» and «Tea»→«VendingMachine» are added since a child feature presence implies its parent's one. The algorithm proceeds in a similar way with the next feature that is not a leaf, i.e. «Tea». Here all the child features are required, except for «Sugar» that is optional. The complete expression for the fd formula and its equivalent CNF are given in Figure 3.2(b). They represent the set of valid products derived from the FD in Figure 3.2(a).

3.2.2 Retrieving the set of featured transitions.

This section illustrates our technique to generate successors in a featured game graph from a SPL specification. A major difference with dealing with successors for standard game graphs is that in the feature extension given a particular source state, the same admissible action can lead to different successors. This is because featured transitions should also consider the different set of valid feature combinations for which the same event is admissible. In other words the same event could cause different MSDs to be activated thus leading to different cut configurations.

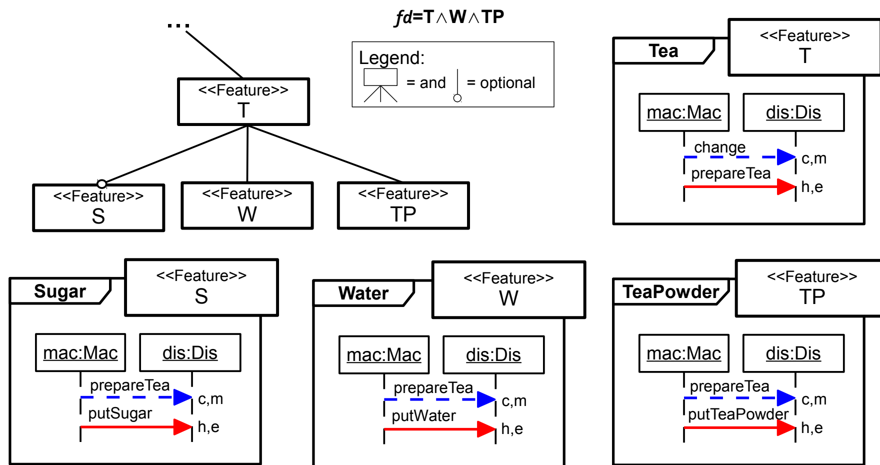
Algorithm 2 shows our technique to generate successors for a given state for all possible variants in the productline, i.e. all the possible ways the set of active MSDs in the source state can progress. Meanwhile the algorithm also retrieves all the featured transitions exiting from the souce state and labels them with both the name of the associated event and a Boolean CNF expression representing a combination of the features triggered by the event. First a set of the admissible events that can occur given the current state (A(S)) is retrieved (line 2). Then for each event we consider if it activates any MSD, i.e. if it is unifiable with the first message of any MSD in the whole MSD specification. *NewlyActivatedMSDs* is a set containing all the MSDs activated by the event

Algorithm 2 Post(S)

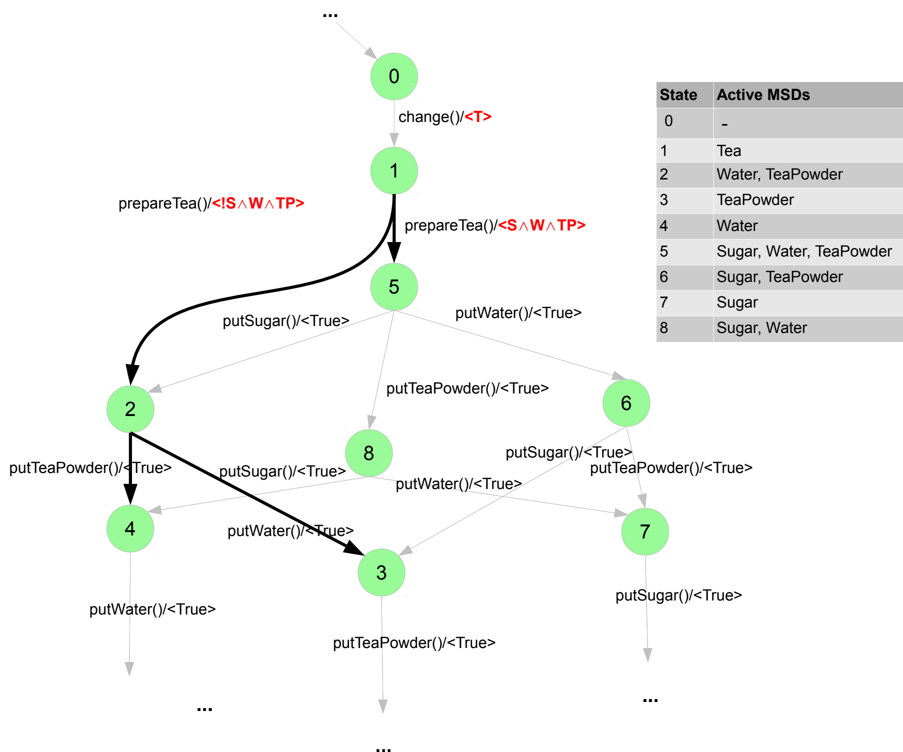
```
1: FeaturedTrans ← ∅;
2: NextEvents ← {α|α ∈ A(S)}
3: while (NextEvents ≠ ∅) do
4:     Take α from NextEvents;
5:     NewlyActivatedFeatures[α] ← ∅;
6:     NewlyActivatedMSDs[α] ← {msd|firstMsg(msd) = α};
7:     if (NewlyActivatedMSDs = ∅) then
8:         FeaturedTrans ← FeaturedTrans ∪ {e = (S, α, S', True)|S' = Next(S, α, True)};
9:     while (NewlyActivatedMSDs ≠ ∅) do
10:        Take msd from NewlyActivatedMSDs;
11:        NewlyActivatedFeatures[α] ← NewlyActivatedFeatures ∪ {f|msd ∈ MSDSpecification(f)};
12:    PowerSet ← 2^NewlyActivatedFeatures
13:    while (PowerSet ≠ ∅) do
14:        Take P from PowerSet;
15:        if (P ∈ fd) then
16:            FeaturedTrans ← FeaturedTrans ∪ {e = (S, α, S', P)|S' = Next(S, α, P)};
17: return FeaturedTrans
```

(line 6). In case this set is empty, the event does not activate any MSD, and the featured transition is labelled with a `True` statement, meaning that it can be explored by any product (lines 7-8). $Next(S, \alpha, True)$ is the state reached from S when the featured transition labeled with the event α and the feature expression $True$ is taken. In that case successors are generated like in standard game graphs, progressing the set of active MSDs in the sourceState. Otherwise from the set of MSDs just activated, the set of features activated by the event is computed in *NewlyActivatedFeatures* (line 11). A feature is activated when one or more of the MSDs within its MSD specification are activated. Subsequently the powerset of all the possible combinations of the activated features are retrieved. A featured transition for each combination in the powerset is generated (lines 12-16) except for the ones which does not belong to a valid product, that are discarded. This is done by means of the Boolean formula *fd* (line 15) explained in Section 3.2.1. $Next(S, \alpha, P)$ is the successor of state S reached through the featured transition labeled with the event α and the feature expression representing theproduct P. Successors reached by featured transitions labelled with such a combination of features are generated by adding the set of MSDs just activated to the set of

MSDs resulting from the progress of the ones already active in the source state. The set of featured transitions is returned as output.



(a) Extract from the Vending Machine SPL specification



State	Active MSDs
0	-
1	Tea
2	Water, TeaPowder
3	TeaPowder
4	Water
5	Sugar, Water, TeaPowder
6	Sugar, TeaPowder
7	Sugar
8	Sugar, Water

(b) Featured game graph

Figure 3.3: Deriving a concise game graph from a SPL specification

Figure 3.3 shows the effects of the procedure applied to an extract of the VendingMachine example.

The SPL specification is given in Figure 3.3(a). It specifies the behavior for two variants of the product depending on if considering the optional feature `Sugar` or not. In this simple example the MSD specification for each feature in the MSD is composed of only one MSD. Thus for each newly actived MSDs a new feature will be activated. An extract of the corrensponding game graph is given in Figure 3.3(b). Let's consider state (1) as a source state. The set of active MSDs for state (1) is `Tea` and the cut is right before the event `prepareTea`. That event is the only one active and happens to activate 3 different MSDs, `Sugar`, `Water` and `TeaPowder`, which are added to the *NewlyActivatedMSDs* set. The set of corresponding activated features is {`S,W,TP`} which is the *NewlyActivatedFeatures* set. Then the powerset of the *NewlyActivatedFeatures* is computed which is {{∅}; {`!S,!W, TP`}; {`!S, W, !TP`}; {`S, !W, !TP`}; {`S, !W, TP`}; {`S, W, !TP`}; {`!S, W, TP`}; {`S, W, TP`}}. It contains all the possible combinations of the 3 newly activated features. Among those combinations only {`!S, W, TP`} and {`S, W, TP`} are valid. Both `W` and `TP` are in fact required by the `T` feature. Two featured transitions for the event `prepareTea` are then explored from the source state, one for each valid combination among the powerset of the activated features. The transitions are the ones leading to state (2) and (3). Now let's consider state (2) as a source state. The set of active MSDs for state (2) is {`Water, TeaPowder`} and corresponds to the set of MSDs activated by the event `prepareTea` plus the MSDs that where already active in state (1) after `prepareTea` is performed. The active MSD Tea terminates when the cut reaches the end of the diagram after `prepareTea` happens, thus for state (2) the set of active MSDs is equal to the set of MSDs activated by `prepareTea`. The cuts are right before `putWater` for MSD `Water` and right after `putTeaPowder` for MSD `TeaPowder`. `putWater` and `putTeaPowder` are the admissible events. Neither of them activates an MSD. Two featured transitions are thus explored from state (2), one for each admissible event, and are both labelled with a `True` statement, since none of the events triggers a new feature.

3.2.3 Winning condition

This Section presents an extention of the winning condition, an essential concept used in Büchi games to iterativelly investigate whether for a given state it is possible to infinitely often reach a goal state.

In its original form, the winning condition associates each given state with a winning flag, specifying if the state results winning or not. The crucial point of our featured extension is that the boolean win flag for a state is replaced

by a featured expression identifying the subset of products which is currently known to be winning. In other words the winning condition does not produce a `True/False` statement anymore, but a featured expression representing the subset of products for which a state is winning. To mark a state as winning we cannot simply check if it eventually reaches a goal state. We also need to consider the set of features labelling the path from that state to a goal state, if reachable, and check if their combination is admitted in at least one valid product. Then the combination is stored in the Win map entry of the state.

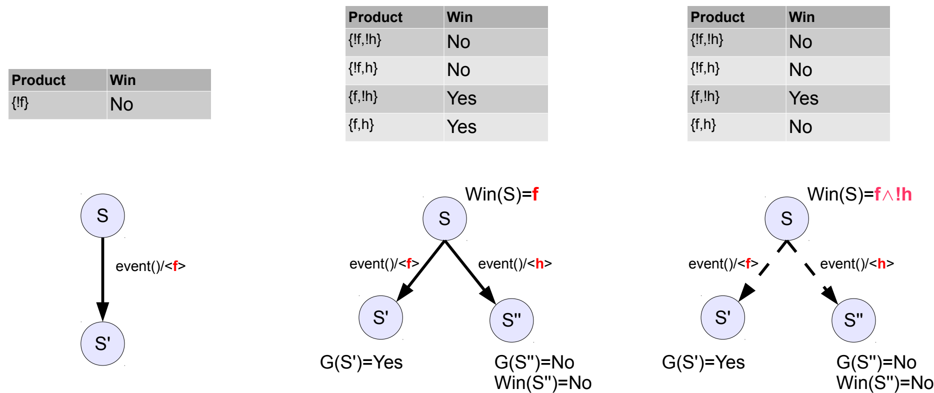
The combination is computed as shown in line 14 of Algorithm 3. γ is the feature expression representing the set of products for which a given featured transition (S, α, S', γ) can be triggered when event α occurs. $GoalProducts$ (resp. Win) is the mapping between each state in the featured game graph and the feature expression telling for which products that state is a goal state (resp. winning state). $GoalProducts(S')$ (resp. $Win(S')$) is the feature expression telling for which products the target state (S') is a goal state (resp. winning state). The formula distinguishes two cases for which the featured transitions exiting from the state can be either all controllable or all uncontrollable. We do not consider featured game graphs in which from the same source state both controllable and uncontrollable transitions can be chosen.

First note that the formula related to the controllable case in Algorithm 3 can also be written as:

$$\Big(\bigvee_{(S,\alpha,S',\gamma)\in Post_c(S)} \gamma\Big) \wedge \bigvee_{(S,\alpha,S',\gamma)\in Post_c(S)} \Big(\gamma \Rightarrow \big(GoalProducts[S'] \vee Win[S']\big)\Big)$$

That alternative formula in case of controllable transitions is easily comparable to the one for the uncontrollable case and let us better distinguish the differences between the two. The meaning of the first part of the formula is that a product can be winning if it can trigger at least one transition. Consider Figure 3.4(a). For products that does not contain f as a feature, state (S') is not reachable from state (S). Thus we can already conclude that (S) is not a winning state for the product, without considering the nature of the transition or if its successors are goal states. The difference between the controllable and the uncontrollable case resides in the second part of the formula.

Let's consider the controllable case first. A state can be marked winning if among all the controllable transitions that a product can execute, at least one leads to a state that is a winning or a goal state. In other words for at least one among the featured transitions exiting from a state the following condition should be verified: if the featured transition can be executed, the successor reached through that featured transition should be a goal or a winning state. Thus the



Product	Win
{!f}	No

Product	Win
{!f,!h}	No
{!f,h}	No
{f,!h}	Yes
{f,h}	Yes

Product	Win
{!f,!h}	No
{!f,h}	No
{f,!h}	Yes
{f,h}	No

(a) Products without at least one executable transition are never winning

(b) controllable case

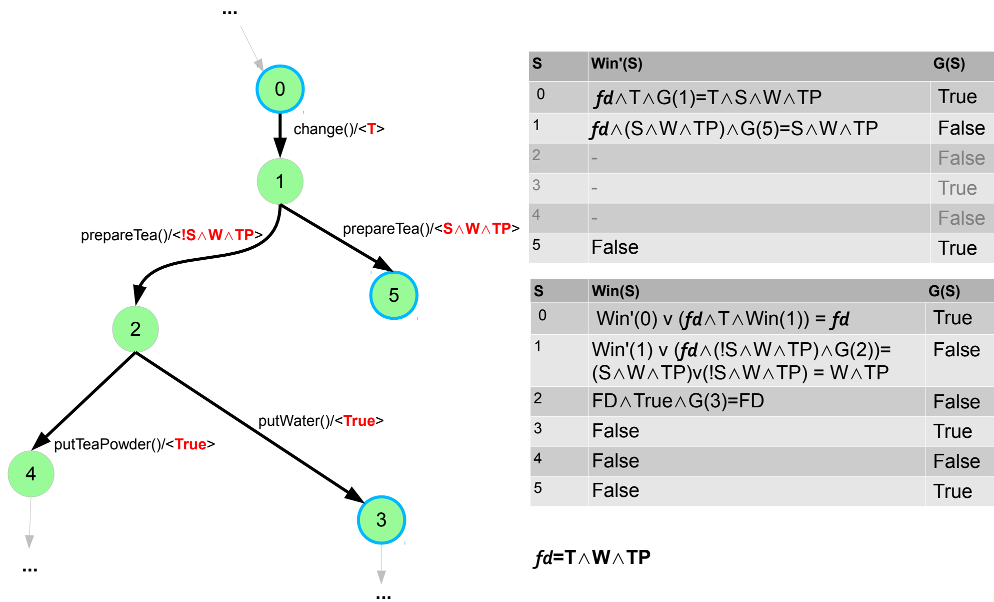
(c) uncontrollable case

Figure 3.4: Retrieving the winning condition for different products

implication in the formula. In the controllable case the system can choose which transition to execute for a given product when more than one eventually reaches a goal state.

Instead in the uncontrollable case a state is winning for a product if whenever the product can execute the transition, it eventually reaches a goal or a winning state through that transition. In other words the product must ensure that every uncontrollable transition that it can trigger leads to a winning or a goal state.

A simple example is given in Figures 3.4(b) and 3.4(c). Both shows state (S) reaching a goal state (S') and a loosing state (S''). With loosing we mean a state that is neither goal nor winning. The difference between the two figures resides in the nature of the transitions. Dashed arrows indicates uncontrollable transitions in the first figure, solid arrows controllable transitions in the latter. Transitions to (S') are triggered by products with feature `f`, whereas transitions to (S'') can only be executed by products with feature `h`. Consider the powerset of the two features `f` and `h` representing four different products which are {{`!f,!h`}; {`!f,h`}; {`f,!h`}; {`f,h`}}. As for the case in Figure 3.4(a), product {`!f,!h`} cannot be marked as winning since it has no transition leading to a winning state. In this case it actually has no transition at all. Similarly product {`!f,h`} has no transition leading to a winning or a goal state in both cases. Product {`f,h`} is winning in the controllable case, since the transition triggered by f leads to a winning state and one is enough. It is not in the uncontrollable case, since all the triggered transitions must lead to a winning or goal state and for the transition



S	Win'(S)	G(S)
0	*fd*∧T∧G(1)=T∧S∧W∧TP	True
1	*fd*∧(S∧W∧TP)∧G(5)=S∧W∧TP	False
2	-	False
3	-	True
4	-	False
5	False	True

S	Win(S)	G(S)
0	Win'(0) v (*fd*∧T∧Win(1)) = *fd*	True
1	Win'(1) v (*fd*∧(!S∧W∧TP)∧G(2))= (S∧W∧TP)v(!S∧W∧TP) = W∧TP	False
2	FD∧True∧G(3)=FD	False
3	False	True
4	False	False
5	False	True

Figure 3.5: Retrieving the winning condition for the VendingMachine example.

triggered by h the condition is not verified. Finally product {f,!h} happens to be winning in both cases since the transition triggered by f is the only executable one and reaches a goal state.

If the winning expression for the same state was already computed by a previous evaluation, that value is considered in the new computation. The value is updated when the previously computed combination is a subset of the newly activated one, i.e. if the state happens to be winning for a larger set of products. In that case the predecessors of the interested state are also marked for re-evaluation. This is done to allow the condition to be applied not only for the successors, but also for states reachable in more than one step. In fact we want a state to be winning for a product if it *eventually* reaches a goal state, without considering the number of steps it should take.

Now consider an extract from the featured game graph of our VendingMachine example given in Figure 3.5. In this example only two products or feature combinations are valid, that are {T, S, W, TP} and {T, !S, W, TP}. Suppose states (0), (3) and (5) satisfy the goal property. Suppose the algorithm has already generated the graph and state (5) has just been marked goal. The algorithm then backpropagates to evaluate the winning condition for its predecessors.

Note that goal states are not necessarily winning themselves. The successor of (1) is already a goal state. *Win** then considers the label from state (1) to state (3) that is {S, W, TP}. That combination is checked to be valid with respect to the *fd* expression. The winning expression is then updated and also the predecessor of state (1) is marked for re-evaluation. The winning condition for state (0) considers the complete path to the goal state (3) by considering the feature which leads to its successor {T} and the information stored in its successor about how to reach the actual goal state that makes him winning, that is {S, W, TP}. The winning expression for state (0) is {T, S, W, TP}. Then the algorithm continues with his exploration until it reaches the goal state (3). The information about how to reach that further goal state is then backpropagated to its predecessors in a similar way until it reaches state (0). An update for its winning expression takes place and the state happens to be winning also for product {T, !S, W, TP}. State (0) results to be winning for all the valid products.

3.2.4 Featured Reachability

Our approach to solve reachability games is an extension of [David et Al 2009] and is given in Algorithm 3. Its output is a function that associates to a state the set of products that can guarantee to reach a goal state from this state.

Roughly it involves an interleaved combination of *forward exploration* of the featured game graph (lines 7-12) together with *back-propagation* of information of winning states (lines 13-17). The forward exploration is performed in a depth-first way. The subroutine *Post(S)* (lines 2, 10) generates all the valid successors for each source state and labels all featured transitions with a subset of products for which that transition is admissible. The featured nature of the game graph represents a first novelty for our approach. Those featured transitions are then added to the *Waiting* pipeline to proceed with the exploration. The algorithm also marks *Visited* states once explored (line 8) and adds each explored transition to the *Depend* map entry of its target state(line 9). When a goal state is encountered, the algorithm marks the transitions in its Depend map for re-evaluation by adding them again to the Waiting pipeline. This is done to re-evaluate the predecessor states in the backward propagation (lines 11-12) to determine whether for them reaching a goal state can be guaranteed. By checking the set of already Visited state, featured transitions leading to states that were already visited previously are analyzed only once by the forward exploration. Any further examination directly goes into the backward propagation which is responsible for performing a re-evaluation of the predecessors of a state anytime the winning condition of that state is updated. The backward propagation for

Algorithm 3 FR(G)

```
1: Visited ← {S_0};
2: Waiting ← {e = (S_0, α, S', γ) | e ∈ Post(S_0)};
3: Win ← ⊥;
4: Depend[S_0] ← ∅;
5: while (Waiting ≠ ∅) do
6:     Take e = (S, α, S', γ) from Waiting;
7:     if (S' ∉ Visited) then
8:         Visited ← Visited ∪ {S'};
9:         Depend[S] ← {e};
10:        Waiting ← Waiting ∪ Post(S');
11:        if (S' ∈ G) then
12:            Waiting ← Waiting ∪ {e};
13:    else
14:        Win* ← (see below);
15:        if ((Win* ⇏ Win[S])) then
16:            Waiting ← Waiting ∪ Depend[S];
17:            Win[S] ← Win*;
   return Win
```

Line 14:

$$Win* \leftarrow Win[S] \vee \bigvee_{(S,\alpha,S'',\gamma)\in Post_c(S)} \Big(\gamma \wedge \big(GoalProducts[S''] \vee Win[S'']\big)\Big) \vee \Bigg(\bigvee_{(S,\alpha,S'',\gamma)\in Post_u(S)} \gamma \wedge \bigwedge_{(S,\alpha,S'',\gamma)\in Post_u(S)} \Big(\gamma \Rightarrow \big(GoalProducts[S''] \vee Win[S'']\big)\Big)\Bigg);$$

the Featured Reachability algorithm, mostly consists of the winning condition presented in the previous section.

3.3 Featured Büchi

To check whether an MSD specification is realizable, we have to check if the system can guarantee that at least one of the goal states must be visited infinitely often by only choosing controllable transitions in the graph. Games with a winning condition requiring to reach a goal state infinitely often are called Büchi games. When this is true for the initial state of the graph, we can derive a consistently executable controller for the system. As for solving reachability games, when managing futured game graphs instead of game graphs for each single product, we need to compute the set of products for which a given state can reach a goal state infinitely often. When such a valid combination of features

exists for the initial state, it represents the set of realizable products. If this is the case, the algorithm can synthesize an FTS, i.e. an automaton representing the behaviour of a set of products, that is actually a concise representation of consistenly executable controllers for the set of realizable products.

Our approach to solve Büchi games for product lines is an extension of [David et Al 2009] and is given in Algorithm 4. It is based on the algorithm for solving reachability games as explained in the previous section. Initially goal states in the featured game graph are considered goal states for all the products (line 1). *Win* maps a state to the set of valid products for which the state eventually reaches one of the goal states and is computed by means of the procedure shown in Algorithm 3 (line 2). Then the algorithm checks whether for some goal state the set of products for which it is a goal state is a subset of the set of products for which it is a winning state (possibly none). The set of products for the goal states for which the condition holds are then updated to the set of products for which those states are winning. This is because if for a particular product a state `S'` is goal but not winning, then it is not sure for states reaching that state to reach a goal state infinitely often. If `S'` is the only goal state they can reach, then they are winning for a reachability game, since they reach `S'`, but not for a Büchi game, since `S'` cannot reach a goal state itself. To compute which states are winning and for which products given the updated set of goal states the algorithm back-propagate again each time the *Goal* map is updated. A new *Win* map is computed by calling the reachability procedure again. Note that only the back-propagation part is executed when calling the procedure within the loop (lines 3-5). The set of realizable products is returned.

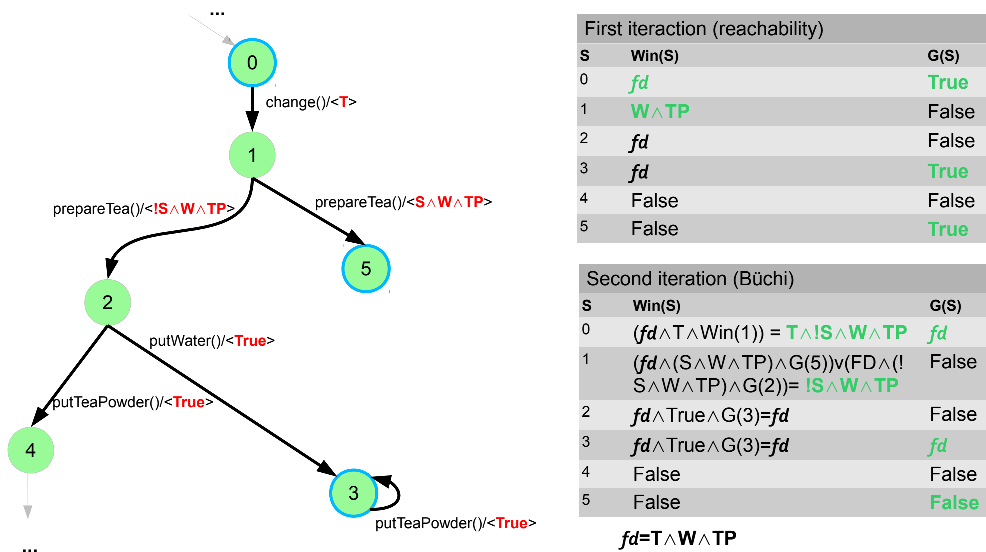
Intuitively, in case after the first call to the reachability procedure the *Win* map coincides with the *Goal* map, the algorithm terminates. That can happen only in case no goal state exists and no valid product is realizable. In fact otherwise since goal states are initially mapped to a `True` statement, having the same combination of products for the *Win* would mean having a `True` statement for the state entry of the *Win* map and this is not admitted by the winning condition of the featured reachability procedure. Each winning combination is checked to be a valid combination with respect to the FD constraints. Then the maximum set of products for the winning combination is the set of valid products.

Figure 3.6 shows a modification of the featured game graph in Figure 3.5. No state in the latter controller happens to be winning, since none of the goal states is infinitely reachable. Then a self-transition has been added for state (3). Suppose the first version of the *Win* map has already been computed from the first call of the reachability subroutine. The only difference with the *Win* map computed for the graph in Figure 3.5 is that *Win(3)* is the subset of all the valid products instead of a `False` statement. Then for each goal state in the goal

Algorithm 4 FB

1: $G \leftarrow \{(g, \top)|g \in G\}$;
2: $Win \leftarrow FR(G)$;
3: **while** $(\exists(g, P) \in G|P \not\Rightarrow Win[g])$ **do**
4: $\quad G \leftarrow \{(g, Win(g))|g \in G\}$;
5: $\quad Win \leftarrow FR(G)$;
6: **return** $Win[S_0]$

map it is checked whether they represent also products that are not winning. The condition holds for all the goal states entries, which are updated. The goal state (5) is not winning itself. This means that all its predecessors result winning when the winning condition stands for states eventually reaching a goal states, but not when the winning condition requires to reach them infinitely often. In fact if the goal state does not reach another goal state itself, it may be that its predecessors cannot reach a goal state infinitely often. Thus the goal entry for state (5) is updated with its entry from the win map. Similarly for goal states (0) and (3). The reachability subroutine is called again to compute a *Win* map considering the update information in the *Goal* map. The update for state (5) is back-propagated until state (0) for which the path to state (5) is nomore considered, since it does not appear as a goal state. The only realizable product for the algorithm is the one excluding feature {`S`}.



First iteraction (reachability)		
S	**Win(S)**	**G(S)**
0	*fd*	**True**
1	**W∧TP**	False
2	*fd*	False
3	*fd*	**True**
4	False	False
5	False	**True**

Second iteration (Büchi)		
S	**Win(S)**	**G(S)**
0	(*fd*∧T∧Win(1)) = **T∧!S∧W∧TP**	*fd*
1	(*fd*∧(S∧W∧TP)∧G(5))v(FD∧(!S∧W∧TP)∧G(2))= **!S∧W∧TP**	False
2	*fd*∧True∧G(3)=*fd*	False
3	*fd*∧True∧G(3)=*fd*	*fd*
4	False	False
5	False	**False**

Figure 3.6: Two iterations of the Büchi algorithm applied to the VendingMachine example.

Chapter 4

Implementation

This chapter describes the implementation of our technique in the ScenarioTools tool suite, a collection of Eclipse-based tools which support the modeling, simulation and synthesis of MSD specifications. Section 4.1 briefly introduces Scenariotools. Next, Section 4.2 illustrates the architecture of our extension. Finally, Section 4.3 gives an overview of how to actually use the extension within the tool.

4.1 Scenariotools

ScenarioTools[1] consists of an Eclipse-based modeling, synthesis and simulation tool suite for MSD specifications developed by the DEEPSE Group, Politecnico di Milano, Italy, and the Software Engineering Group, University of Paderborn, Germany. It currently supports the modeling and synthesis of both *static* and *dynamic software-intensive systems*. Static systems are systems made up of a fixed number of objects, whereas dynamic systems are systems that update dynamically at runtime and in which the number of components may vary as well as the communication relationships between them. In the following we only relate to static systems. For further details about possible applications of ScenarioTools to dynamic systems we refer the interested reader to [Ghezzi et Al 2012, Greenyer 2011] and the official webpage http://scenariotools.org.

4.1.1 Modeling

Scenariotools supports the modeling of SPL specifications as a combination of MSD specifications and FDs. That is done via the Papyrus UML editor in

[1] ScenarioTools http://scenariotools.org

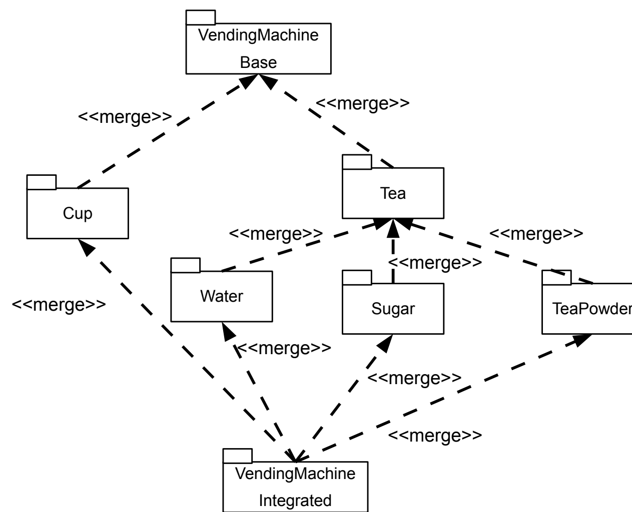


Figure 4.1: Overview of the package structure for the VendingMachine example.

Eclipse[2]. Extensions to Papyrus and UML, as part of ScenarioTools, allow users to specify temperature and execution kind for a message, as well as to label MSDs as assumptions within a graphical editor. The ScenarioTools *modeling scheme* for static systems consists of a package structure as the one shown in Figure 4.1 for the VendingMachine example.

The first package in the structure is the *base package*. It consists of a *class diagram* together with a *collaboration diagram* and specifies the structure of the system.

The base package is merged by other packages, which tipically contain a set of MSD schemas representing the behaviour for different features. Figure 4.2 shows the contents of the base package and the behavioural concerns specified by the package «Tea» from the VendingMachine example.

Eventually, packages are merged by an *integrated package*. When modeling an SPL, the integrated package should contain the FD representation, which is also modeled via a UML extension, inspired by [Possompes et Al 1998]. The integrated package represents the overall behaviour of the SPL, as the union of the MSDs contained in the packages it merges. Features are represented by UML components. Parent features are associated with a *port* by which one could specify whether their child features are in an AND, OR, or XOR relationship. Optionalities may be specified within the dependency connecting a feature with its child feature through a particular stereotype. Each feature is related to the

[2]Papyrus http://www.eclipse.org/papyrus/

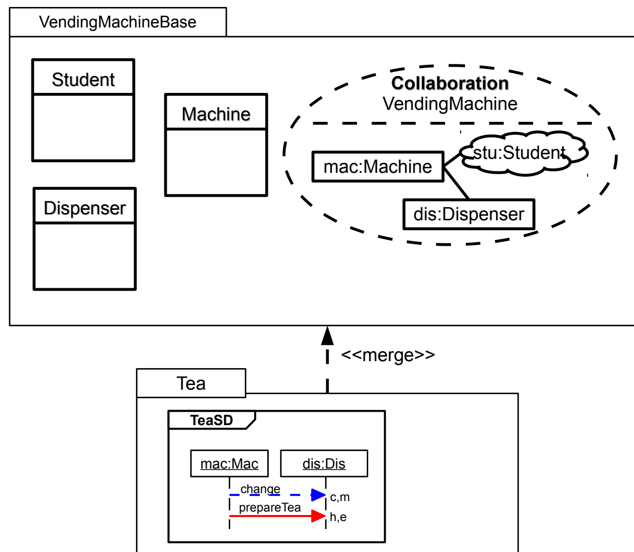


Figure 4.2: The contents of the base package and Tea package from the VendingMachine example.

corresponding package specifying the feature's behaviour. The implementation of the FD in ScenarioTools for the VendingMachine example is given in Figure 4.3.

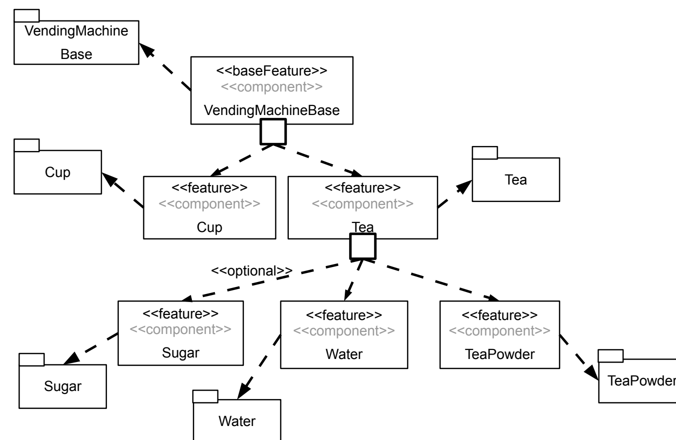


Figure 4.3: The implementation of the FD in ScenarioTools for the VendingMachine example.

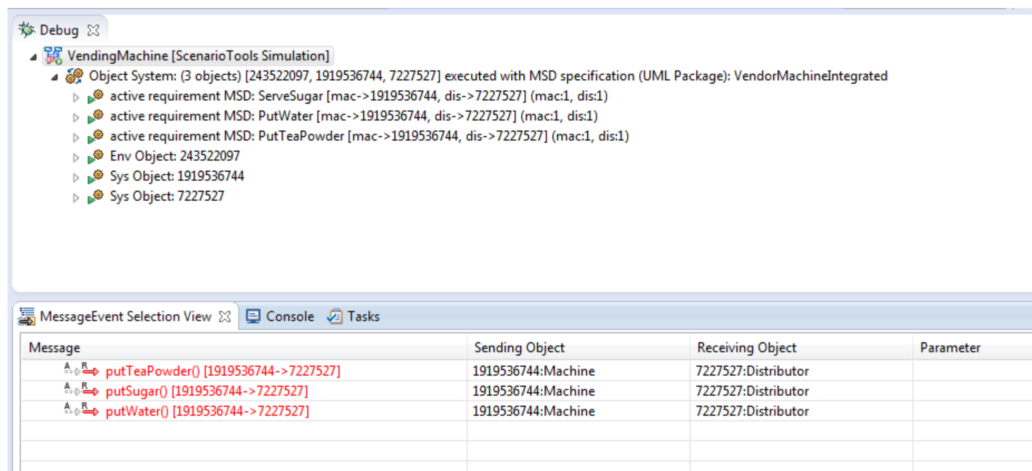


Figure 4.4: A screenshot of the ScenarioTools runtime during the simultation of the VendingMachine example.

4.1.2 Simulation

The simulation of the MSD specification is performed by means of the play-out algorithm [Harel and Marelly 2002]and integrates into the *Eclipse Debug Framework*.

Once the simulation is launched, the user is prompted to an environment similar to the one shown in Figure 4.4. The *Debug View* in the top left displays the active MSDs and objects in the object system. The *Variable View* in the top right provides the lifeline bindings. Finally the *MessageEvent Selection View* at the bottom presents the list of enabled messages for the current state. The nature of the message is described by label colors and icons at the left of the message name. For example in the picture all possible next events are in a hot and executed cut. At each step the user may choose the next event to perform by clicking one of the messages in the list.

4.1.3 Synthesis

ScenarioTools also supports the synthesis of MSDs specifications. First the state space described by the model can be explored. It considers reachable states in an execution, including states leading to a safety or liveness violation. A state graph is generated and can be visualized using Graphviz [3]. Figure 4.5 shows part of the state space diagram created for the VendingMachine example during the synthesis of the product with all the features enabled.

[3]Graphviz http://www.graphviz.org/

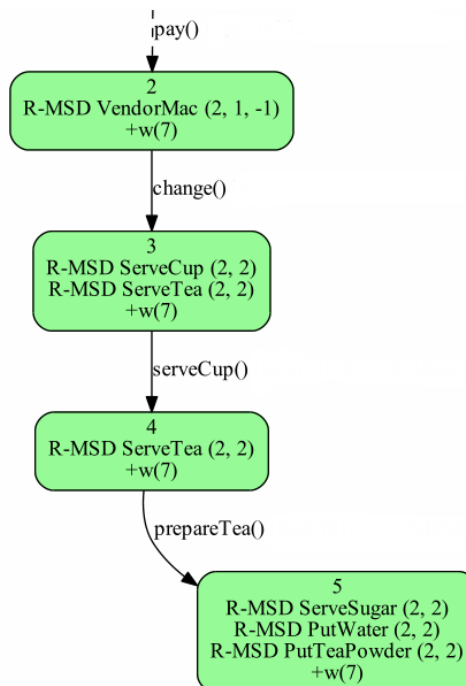


Figure 4.5: Part of a state space diagram explored by the ScenarioTools synthesis for the VendingMachine example. The graph considers the product with all features enabled

Each state contains the list of active MSDs in a particular cut and lifeline bindings. Transitions are labelled by the name of the message that is exchanged. Uncontrollable transitions are depicted by dashed arrows, controllable transitions by solid ones.

This state exploration mechanism is used by the synthesis algorithm to find whether a strategy exists for the given specification. The algorithm can either determine that the product is not realizable or extract a controller fulfilling its specification.

Visually, this controller appears as a state graph, except for violating states and transitions leading to them, which are removed.

4.1.4 Configuring Executable Specifications

Both synthesis and simulation in ScenarioTools use the same run-time logic. This ensures consistent results for both kinds of analysis. In order to perform the simulation and controller synthesis of MSD specifications, it is necessary to create a concrete object system, also called execution object model, based upon which the MSDs can then be interpreted. ScenarioTools allows the creation of

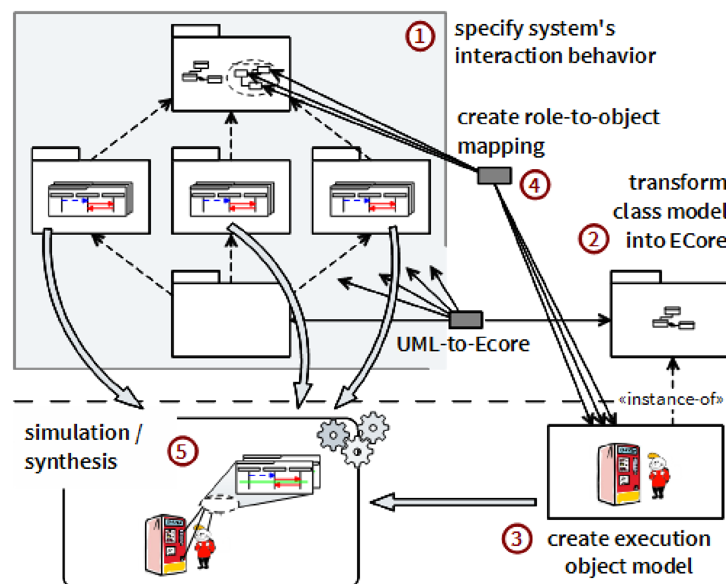


Figure 4.6: Overview of the models involved in the execution of MSDs for a static object system, with static lifeline bindings

the concrete object system through the Eclipse Modeling Framework (EMF) [4]. The EMF is a modeling framework and code generation facility. From a model specification, called Ecore model, EMF provides tools and runtime support to produce a set of Java classes for the model. An overview of the process is given in Figure 4.6. The models involved and their relation are explained below.

System's interaction behavior The MSD specification is modeled as described in Section 4.1.1.

ECore class model The specification's class diagrams are mapped to a corresponding Ecore class model through a UML-to-Ecore mapping.

Execution object model This model, together with the collaboration diagram describing the object structure, allows to create an object model.

Role-to-object mapping Lifelines in the MSD specification are mapped to objects. This mapping is possible when dealing with static systems since only one execution object model exists and lifelines are not bind dynamically to objects.

Simulation/synthesis Based on the concrete object model, MSDs can be interpreted for simulation and synthesis.

[4]Eclipse Modeling Framework http://www.eclipse.org/modeling/emf/

We will guide the user throughout those steps in Section 4.3.

4.2 Architecture

We extended ScenarioTools with the capability to synthesize SPL specifications simultaneously as described in Chapter 3. This section provides a high-level presentation of the extension and describes how it integrates with the tool.

The featured synthesis approach, also called *simultaneous synthesis* in the following to underline his capability of synthesizing an SPL *at once*, was developed in Java as an Eclipse plug-in using the *Plug-in Development Environment (PDE)* [5].

4.2.1 Inputs and outputs of the simultaneous synthesis

Before describing the structure of our extension we briefly present its inputs and outputs. Figure 4.7 represents the plug-in as a black box.

Input The simultaneous synthesis algorithm can be executed from a model file generated when creating the execution object model and specifying which models belong together. Such a model file is called `scenariorunconfiguration` and represents the input of the process.

Output The synthesis run yields two outputs. Right after the end of the process a pop-up is generated containing the following information:

- General outcome of the synthesis i.e. true if at least one product among those in the SPL is realizable, false otherwise.
- Number of explored states and transitions.
- Time taken by the algorithm to synthetize the SPL specification in milliseconds.
- Total number of goal and winning states in the featured game graph, i.e. states which are marked goal or winning for at least one product.
- Number of realizable products and a feature expression representing them.

The process also generates a model file, namely `msdruntime` file, containing the information needed to generate the featured game graph.

[5] Plug-in Development Environment http://www.eclipse.org/pde/

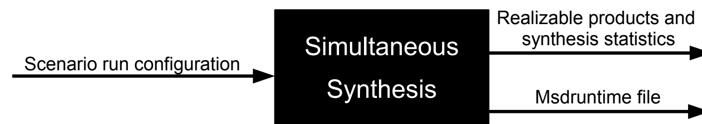


Figure 4.7: The inputs and outputs of the plug-in

4.2.2 Main structure and dependencies

This section presents the main structure of our extension as illustrated in the class diagram of Figure 4.8. Below we briefly describe the main classes and their roles in the process.

SimultaneousProductLineSynthesisAction It contains a run method that is invoked when the user executes the simultaneous synthesis from the `scenariorunconfiguration` model file. It handles the input file and calls the SimultaneousProductLineSynthesisJob.

SimultaneousProductLineSynthesisJob It is the job running in the background. It starts the algorithm, waits for its completion, then generates the `msdruntime` file and a pop-up containing the outcome of the synthesis.

SimultaneousProductLineSynthesis It represents the fundamental class of the project. It contains the implementations of the algorithms presented in Chapter 3. To accomplish those functions it relies on a Java library for *Binary Decision Diagrams*(BDD), called JDD, and on the EMF model, both described in the following.

JDD. A Java BDD library

.

Binary Decision Diagrams (BDD) [Bryant 1992, Andersen 1997, Akers 1978] are data structures representing a Boolean function. They are widely used in model checking [Burch et AL 1992], formal verification [Bryant 1995] and optimizing circuit diagrams [Fey et Al 2004]. We adopted BDDs to represent and manipulate feature expressions in our implementation by including the JDD[6] Java library in our project.

A feature in the feature diagram can be represented by a BDD variable. The library provides many functions which implement logical operations on BDD

[6] JDD library http://javaddlib.sourceforge.net/jdd/

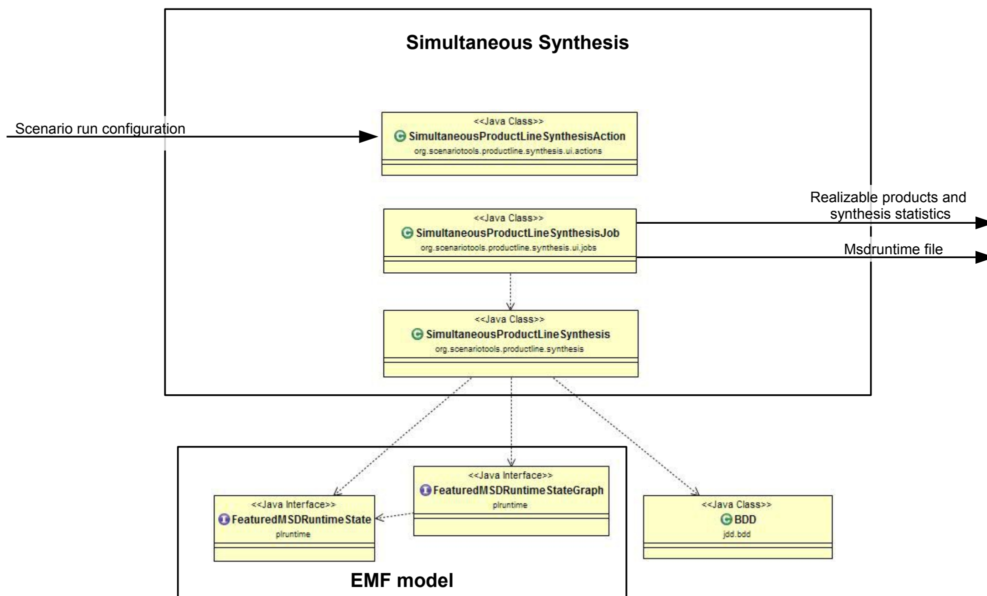


Figure 4.8: The architecture of the plug-in and its main dependencies illustrated by a UML class diagram.

variables like conjunction, disjunction and negation. They are extensively used in our plug-in to handle feature expressions, for example when labeling transitions in the featured game graph, retrieving the set of valid products from the FD or computing the winning condition. A BDD represents a set of products as feature combinations in which a feature takes one of three possible values: 1 when its presence is mandatory, 0 when its absence is mandatory, $-$ otherwise. The order of the features in the feature expression depends on the order in which their corresponding BDD variable is declared. Coming back to our VendingMachine example, suppose we have four features «Tea», «Water», «TeaPowder», «Sugar» and four corresponding BDD variables declared in the same order. The feature expression $\{111-\}$ corresponds to the CNF formula («Tea» $\wedge$ «Water» $\wedge$ «TeaPowder» $\wedge$ «Sugar») $\vee$ («Tea» $\wedge$ «Water» $\wedge$ «TeaPowder» $\wedge$ «!Sugar»). To provide the set of realizable products at the end of the process, the plug-in translates the BDD encoding this set to a concise CNF formula.

EMF model

.

As explained in Section 4.1.4 the EMF is the framework by which ScenarioTools allows the creation of the concrete object system. It consists of a modeling

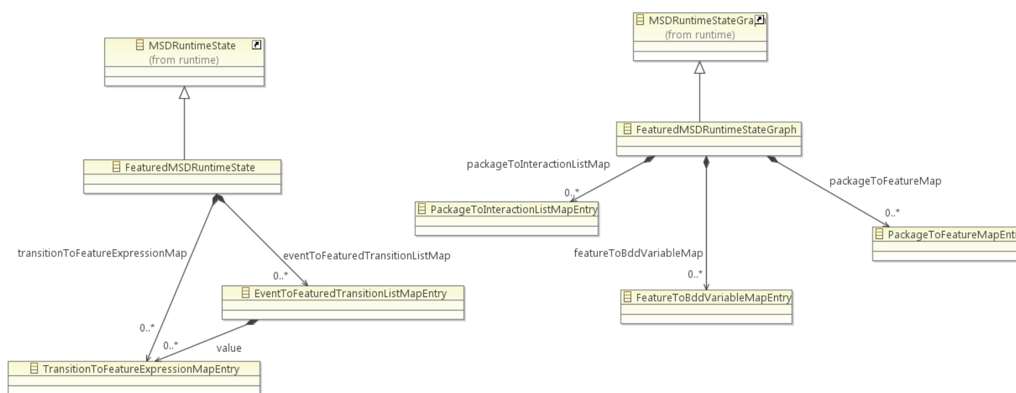


Figure 4.9: Extension of the existing ECore class model build through EMF.

framework and code generator facility which produces a set of Java classes for a structured data model specification destribed in XMI.

We benefited from the EMF support to model inheritance and extended the existing ECore class model in ScenarioTools to cope with the featured nature of transitions and game graph in our approach. Figure 4.9 shows the EMF model used by our plug-in. Each block in the model corresponds to both a Java class and Java interface generated by the code generator and enhanced to integrate with our plug-in. Below we briefly describe the main blocks.

FeaturedMSDRuntimeState It represents a state in the featured game graph. It inherits attributes and methods from the MSDRuntimeState which is the one corresponding to a set of active MSDs in a particular cut in the standard game graph. Thus it inherits information about currently active MSDs and possible safety violations in requirement and assumption MSDs. In addition it keeps track of the featured transitions exiting from the state. In a standard game graph only one transition labeled with the same event could exit from the same state. In the featured case though, it is possible to have more than one transition labeled with the same event exiting from the same state. That is because the same event may cause the set of active cuts of the MSD specification to progress in different ways, depending on the product which is considered. The set of featured transitions exiting from the state is represented by a nested map. At a first level we have a mapping between a standard transition and a feature expression label defining a featured transition. Then we map an event to a set of featured transitions which may be triggered by the event. To handle featured transitions `FeaturedMSDRuntimeState` interacts with `TransitionToFeatureExpressionMapEntry` and `EventToFeaturedTransitionListMapEntry`.

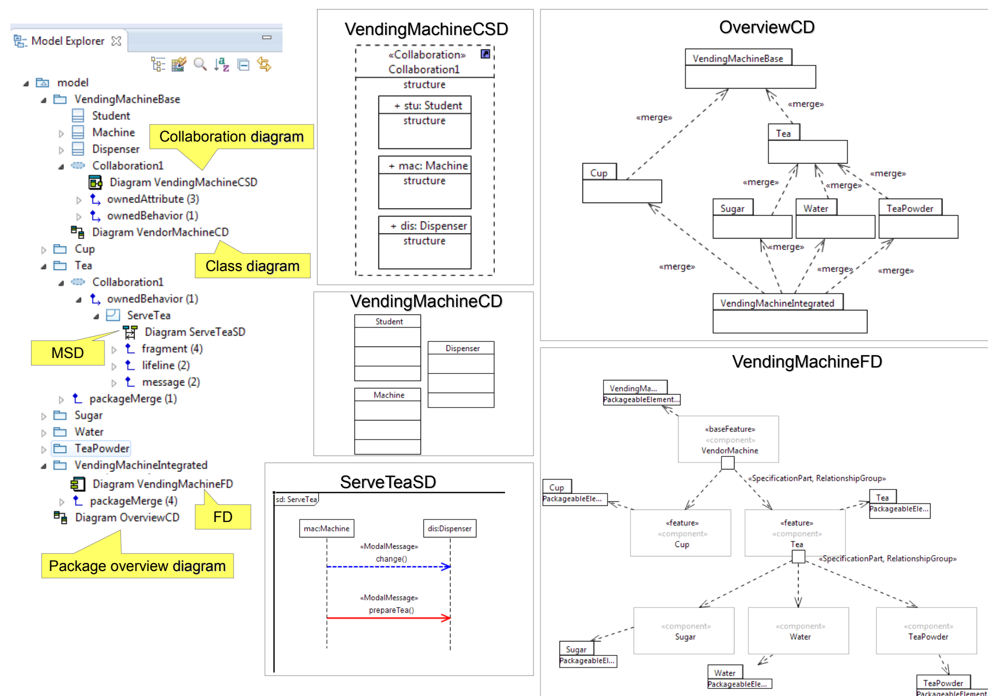


Figure 4.10: Overview of the diagrams involved in the SPL specification of the VendingMachine example in the Papyrus editors.

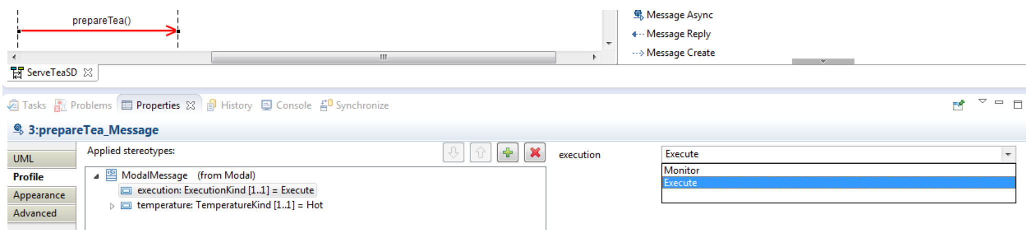
FeaturedMSDRuntimeStateGraph It represents the featured game graph. It contains methods responsible for exploring the state space from a state by generating all possible featured successors for all events that may happen from a state. It also maintains a map between features and BDD variables.

4.3 Using the package

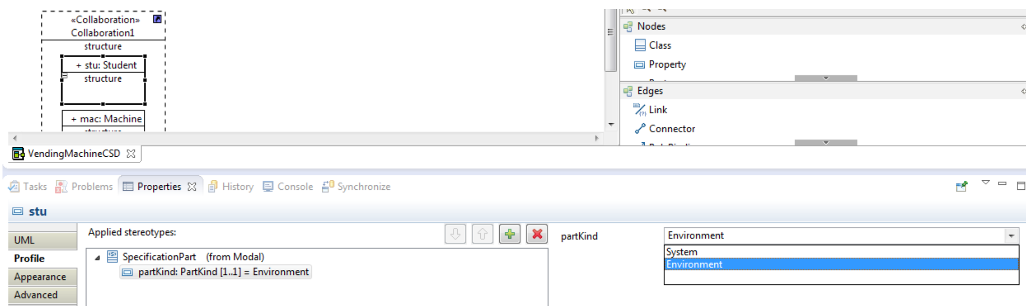
The aim of this section is to give an overview of how to use our plug-in within ScenarioTools. In order to run a simultaneous synthesis the user needs to model the SPL specification, generate a concrete object model and eventually run the synthesis. Those three steps are described in more details in the following.

4.3.1 Modeling

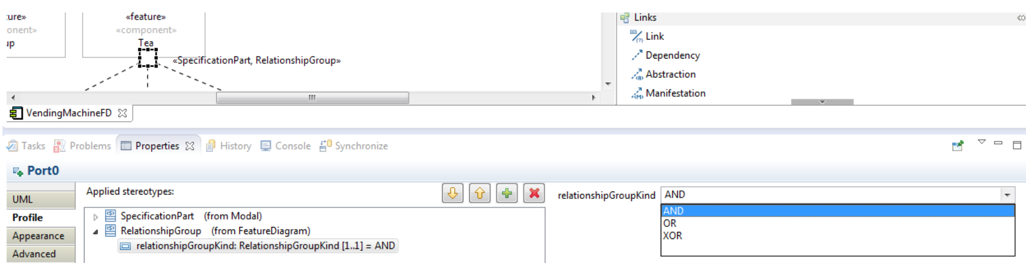
As already described in Section 4.1, modeling a SPL specification consists in modeling a package overview diagram, class diagram, collaboration diagram and



(a) Specifying that the «prepareTea» message is *executed*



(b) Specifying that `Student` represents an *environment object*

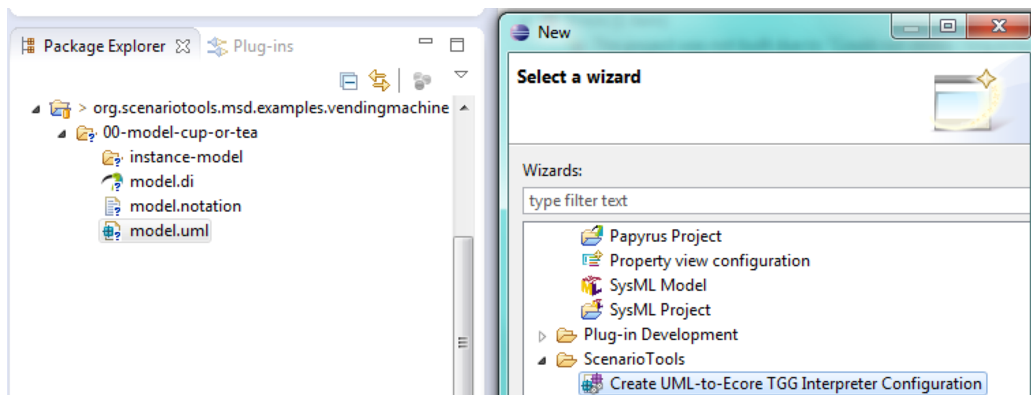


(c) Adding an *AND* decomposition between feature «Tea» and its child features

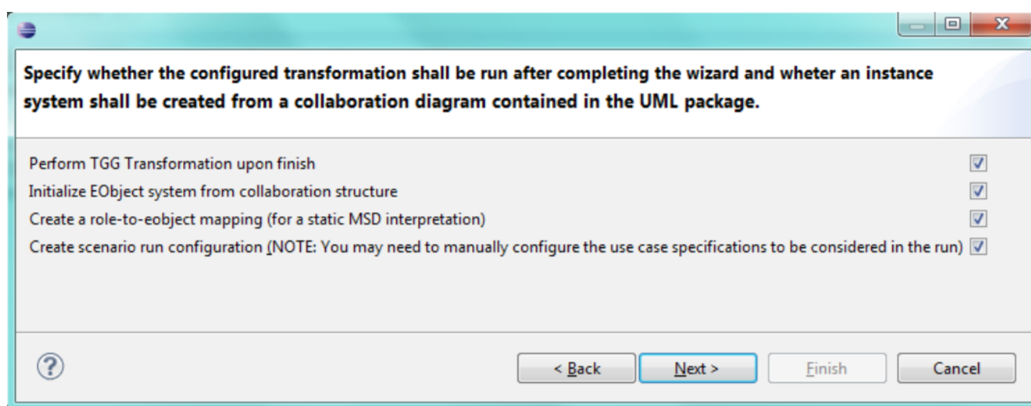
Figure 4.11: Some of the stereotypes provided by ScenarioTools

FD, besides modeling MSD specifications within some or all of the merged packages. Figure 4.10 shows the corresponding diagrams of the VendingMachine example in the Papyrus editors.

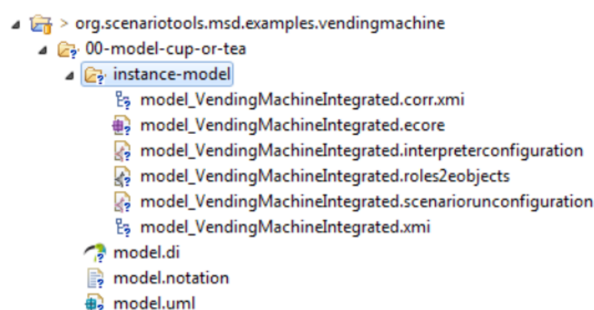
ScenarioTools provides a set of stereotypes which allow one to detail the different components involved in the different diagrams. For example a user could specify the temperature and execution kind of a message in an MSD (Figure 4.11(a)), whether a role represents an environment or a system object(Figure 4.11(b)), or the decomposition kind for a port in the FD (Figure 4.11(c)).



(a) The user can generate the concrete object system through the `create UML-to-Ecore TGG interpreter configuration` wizard



(b) One of the steps in the automated generation. The user can specify to create a scenario run configuration and role-to-object mapping.



(c) The files created after the transformation.

Figure 4.12: Some of the steps to create a concrete object system

4.3.2 Automated generation of the ECore class model

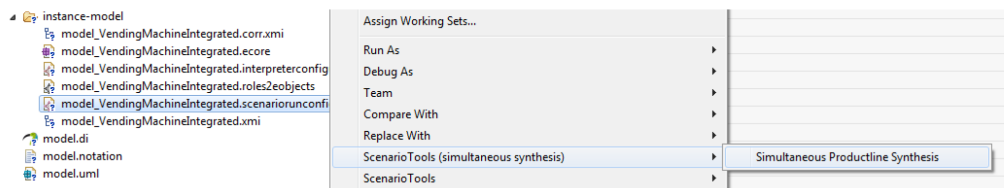
Once the SPL specification is modeled, the corresponding concrete object system can be created. ScenarioTools supports the user in this automated generation through the `create UML-to-Ecore TGG interpreter configuration` wizard (see Figure 4.12(a)). First a `uml` model file should be selected as the source of the UML-to-ECore transformation configuration. Then the user can select a merging package, usually the integrated package, that will be transformed into an Ecore package. All the UML packages merged by the selected package will be merged in this phase. The user can also specify to create an object system from a collaboration diagram in his model, to automatically create a role-to-object mapping and whether to create a scenario run configuration file, i.e. a configuration model specifying which models belong together, as illustrated in Figure 4.12(b). It is also possible to indicate the folder in which the generated files shall be stored. Figure 4.12(c) provides a list of the files created.

4.3.3 Running the simultaneous synthesis and showing the featured state graph

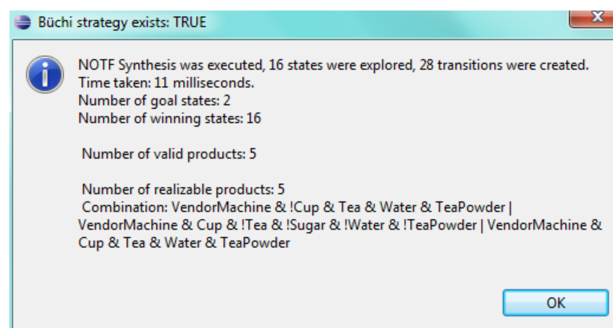
After generating the concrete object system it is possible to run the simultaneous synthesis of the SPL specification. The model file from which the synthesis can be executed is the scenario run configuration (see Figure 4.13(a)).

After the process ends, a pop-up provides information on the number of explored states and transitions, the time taken for the execution, the number of realizable products in the SPL and a feature expression representing them. The results of the simultaneous synthesis on the VendingMachine example are given in Figure 4.13(b).

The synthesis also produces a `msdruntime` file from which it is possible to generate the corresponding featured game graph as shown in Figure 4.14(a). Part of the graph for the VendingMachine example is given in Figure 4.14(b). Transitions are labeled with both the name of the exchanged message and the bdd expression representing the combinations of features for which the transition is contemplated. In this phase the bdd expression has not been translated to a CNF feature expression to avoid clutter. Also note in the graph that from state ⓶ the same evant «`change`» can lead to three different states depending on if only feature «`Tea`» (10), only feature «`Cup`»(01) or their combination (11) is taken into consideration.

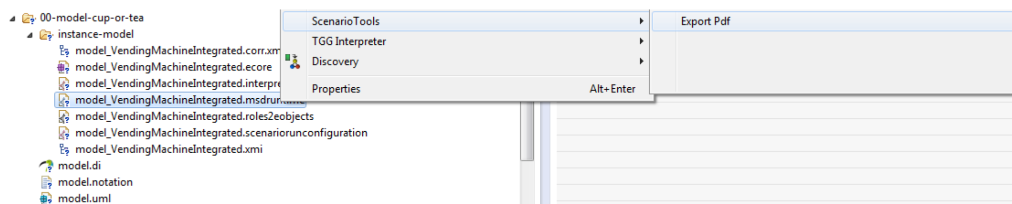


(a) Running the simultaneous synthesis from the *scenariorunconfiguration* file.

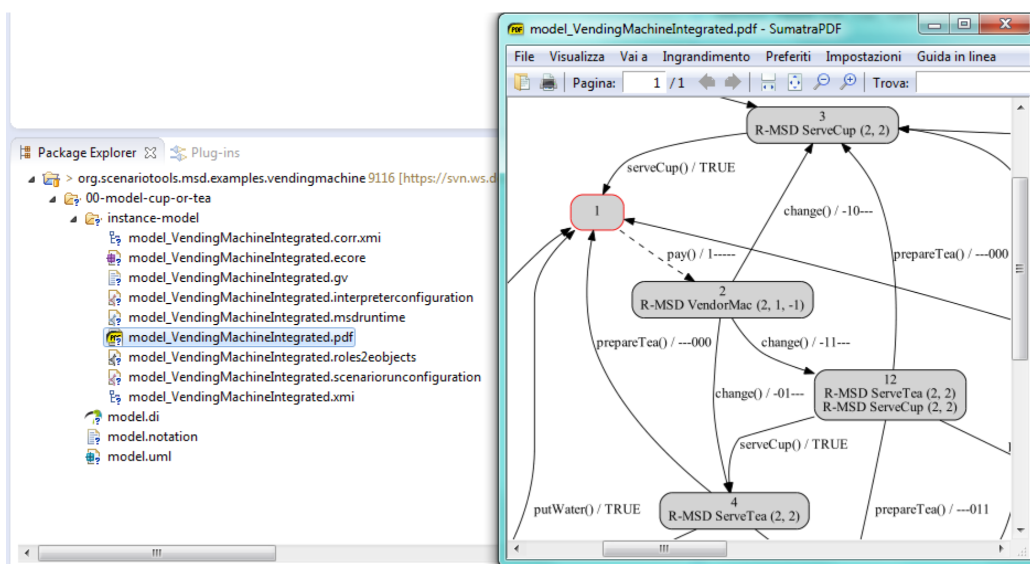


(b) Results are popped up after the process ends.

Figure 4.13: Simultaneous synthesis of the VendingMachine example.



(a) Generation of a pdf file containing the featured game graph from the *msdruntime* file.



(b) Part of the featured game graph.

Figure 4.14: Generating the featured game graph for the VendingMachine example.

4.4 Limitations and perspectives

Our implementation extends ScenarioTools to synthesize a whole SPL specification at once. Also, an FGG representing a global state graph can be generated, which maintains a link between a given execution and the features needed to trigger it.

However a couple of limitations exist. First, our plug-in does support simulation. Although the tool allows one to run the simulation on the SPL specification, the only target of the process is the product composed of all the feature specifications. One way to solve this drawback would be to let users select the features to include in the specification. Then, from the FGG, one should be able to extract the corresponding controller and execute it.

Another minor limitation of our technique is that when generating the feature expression representing the set of valid products in the FD, it does not consider cross-tree constraints. This fix should be easily applicable by extending Algorithm 1 in order to consider the respective constraints.

Chapter 5

Evaluation

In this chapter we evaluate the applicability and the efficiency of our approach. First, in Section 5.1 we apply our technique on our Vending Machine example and an ambient intelligence system case study. Then, in Section 5.2 we assess the efficiency of the featured synthesis algorithm against the successive checking of the individual products, using different technical evaluation examples.

5.1 Applicability Evaluation

As a first evaluation, we applied our methodology on our running example. We modeled the SPL specification given in Section 2.4 in ScenarioTools and checked the realizability of the SPL through our plug-in. For the Vending Machine case, the featured synthesis approach successfully identifies that of all five products are realizable.

We then ran the algorithm on a partially inconsistent specification, to verify whether our technique was able to identify unrealizable products. This second example is a simplified version of a use case from the *Conviviality and Privacy in Ambient Intelligent Systems* (CoPAInS) project[1], an ambient intelligence system case study developed by the University of Luxembourg. We will refer to it as the *CoPAInS example* hereafter.

In the next sections we give an overview of this example, the involved system and environment objects, its informal requirements and assumptions and finally present the outcome of the experiment, showing the ability of the algorithm to recognize specification inconsistencies.

[1] CoPAInS project: http://www.fnr.lu/en/Research-Programmes/Research-Programmes/Projects/Convivialty-and-Privacy-in-Ambient-Intelligence-Systems-CoPAInS

5.1.1 CoPAInS Overview

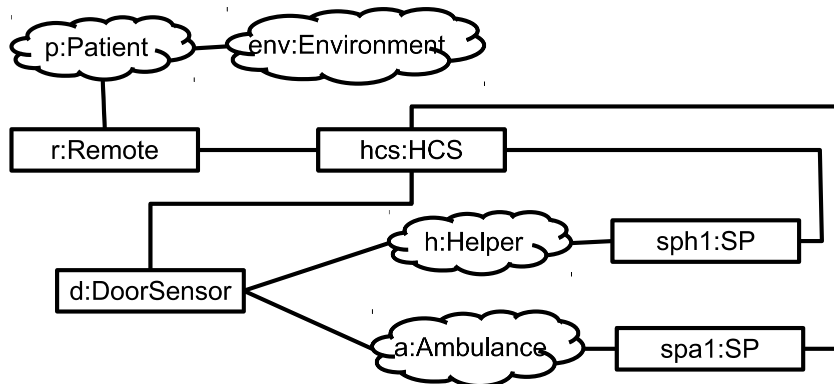
Ambient Intelligence (AmI) represents an emerging field of research and development which aims to provide a new level of assistance and support in people's daily activities, transforming living and working environments into *intelligent spaces*. A particular case of Ambient Intelligence is that of *Ambient Assisted Living* (AAL), whose primary goal is to address the needs of people with disabilities and ensure their health, safety, and well-being, as well as to help them maintain self-sufficient living. A major challenge for AAL and AmI systems in general, is to preserve both user's *privacy* and *convivial* interactions among humans and devices [Moawad et Al 2012, Efthymiou et Al 2012], hence the name of the project *Conviviality and Privacy in Ambient Intelligent Systems* [2].

In the following, we consider an AAL system product, the *Home Care Assistant*, originally conceived to inquire about privacy and conviviality properties of Ambient Intelligence systems. We adapt it to show the applicability of our methodology to SPLs by conceiving two variants, only one of which is realizable.

In its simplied version, the home care assistant works as follows. A patient is provided with a remote controller, by which a help request can be submitted. When the patient needs help, he can activate the alarm on his remote controller by pressing a button. This triggers the alarm activation in the main Home Care System (HCS). The HCS has a list of contacts to call in case of emergency. The list is ordered on the basis of the patient's preference and is scanned top down. The HCS starts contacting the first person in the list, asking him for his availability. Every contact, called *helper* hereafter, is provided with a smartphone and a particular app installed, by means of which he can accept or deny the request. If a user accepts to help, the HCS will wait for the helper to go to the patient's house. A sensor, installed on the main door will register the helper entering the room, calling off the alarm. Otherwise, if the helper is not available, the HCS will contact the next helper on the list. Optionally, the system could be implemented to call the ambulance, in case nobody on the list is available. In this latter case, we expect that the ambulance is also equipped with a smartphone, allowing the HCS to connect with it the same way it does with other helpers in the contacts list.

We modeled the Home Care Assistant as a static system in ScenarioTools. The sketch representing system and environment objects as well as their interactions is given in Figure 5.1. Static systems are systems in which the number of objects, such as the number of helpers and consequently the number of smartphones involved, is fixed. Hence we set the number of helpers in the list of contacts beforehand. For simplicity, if the ambulance is not involved, we consider that only one helper is in the list. Otherwise, we consider a list composed

[2] Also note that in french, "copain" is the translation for "buddy".



Environment Objects

Patient: The person who needs help.
Helper: The person who replies to the help request and possibly provides help in case of emergency.
Ambulance: Optionally the last contact in the contact list.
Environment: General entity representing the fact that the patient is helped and healthy again.

System Objects

Remote: Remote controller used by the patient to ask for help.
HCS (Home Care System): Main controller that captures the request, contacts a list of helpers and reacts according to their answer.
SP: Smartphone through which contacts are asked for help and reply to a help request.
DoorSensor: A sensor installed on the patient's door in order to register whenever a helper arrives.

Figure 5.1: The Object System for the CoPAInS example

of one helper and an ambulance.

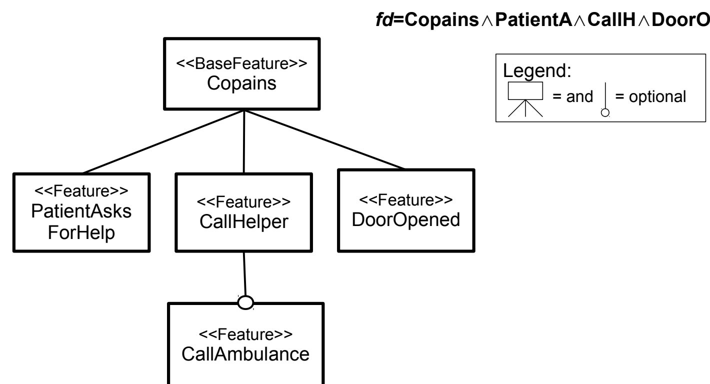


Figure 5.2: The FD for the CoPAInS example

For the CoPAInS example, three features can be readily identified: the ability to register a help request, the ability to call a helper and to react to his reply

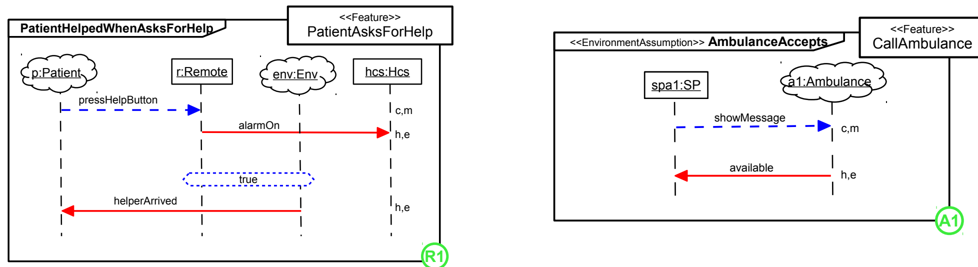
and the ability to monitor a door sensor. We also consider the ability to call the ambulance, when nobody else in the list is available, as a fourth optional feature. The FD used to capture the valid combinations of those features is shown in Figure 5.2. It represents two products. The difference between the two is given by the presence of the `CallAmbulance` feature.

For each of those features we identify system requirements and assumptions about the environment. They are described informally in Table 5.1.

Table 5.1: Informal requirements and assumptions for the CoPAInS example.

Feature	**Requirements**	**Assumptions**
Patient Asks For Help	**R1)** After the patient asks for help the alarm is set on and eventually the patient must be helped.	**A1)** We assume that the patient does not ask for help twice before the alarm is called off.
CallHelper	**R1)** After the alarm is set on, the HCS sends an help request to the smartphone of the first helper in the list. The smartphone shows the message to its owner. **R2)** After the helper reads the message, he could reply YES or NO. In the first case the smartphone tells the HCS that the first helper is available to help. Otherwise the HCS is informed that the helper is not available.	**A1)** We assume that after the smartphone shows the message to the helper, either he answers with his availability or with his unavailability. Through this assumption, we basically rule out the scenario in which the helper does not reply at all.
CallAmbulance	**R1)** Whenever the last helper is not able to help, the HCS sends a request to the ambulance.	**A1)** The ambulance is always available to help.
DoorOpened	**R1,R2)** When the helper enters the room, the alarm is set off. The same if the ambulace enters.	**A1,A2)** We assume that when the helper replies that he is available to help, he will eventually show up. That is also true for the ambulance. **A3,A4)** When the helper or the ambulance enter the room, after the alarm is called off, the patient is helped.

The MSD specifying Requirement **R1)** within the *PatientAsksForHelp* fea-



(a) MSD specifying Requirement R1 for the feature PatientAsksForHelp

(b) MSD specifying Assumption A1 for the feature CallAmbulance

Figure 5.3: Two MSDs from the CoPAInS SPL specifiction.

ture is given in Figure 5.3(a). It represents the general requirement for the Home Care Assistant and, also, its main goal. The primary objective of an AmI system is to ensure that whenever a customer needs help and asks for it, he will eventually be helped. The absence of Feature `CallAmbulance`, with its Assumption **A1)** saying that the ambulance is always available to help, introduces a liveness violation of this main goal. Without that assumption, if no helper in the list is available to help, nobody will finally assist the patient. One could assume the same for the helper, but this is not something reasonable in a real-life scenario. For instance it could happen that everybody is physically unable to help, as being out of town. Conversely, it is something people would typically expect from an ambulance. The main difference between the ambulance and a helper in our specification, is that the ambulance is always available to help. We model that through the assumption MSD in Figure 5.3(b). The whole SPL specification for the CoPAInS example is given in Appendix A.

As a consequence, the only realizable product in the SPL is the one including `CallAmbulance` in its combination of features.

5.1.2 Featured synthesis outcome

After modeling the SPL specification in ScenarioTools, we applied the featured synthesis on the family of two products. As presented in the previous section, the product without Feature `CallAmbulance` is inconsistent. Our technique successfully recognizes that the only realizable product in the SPL is the one considering the ambulance in its specification. The outcome of the run is given in Figure 5.4.

We then generated the FGG explored by the process. An excerpt of the FGG is shown in Figure 5.5. State (5) represents the step after the helper has read the

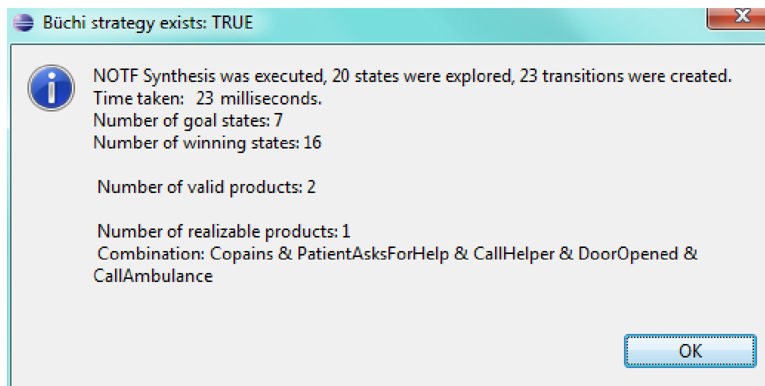


Figure 5.4: The featured synthesis outcome for the CoPAInS example.

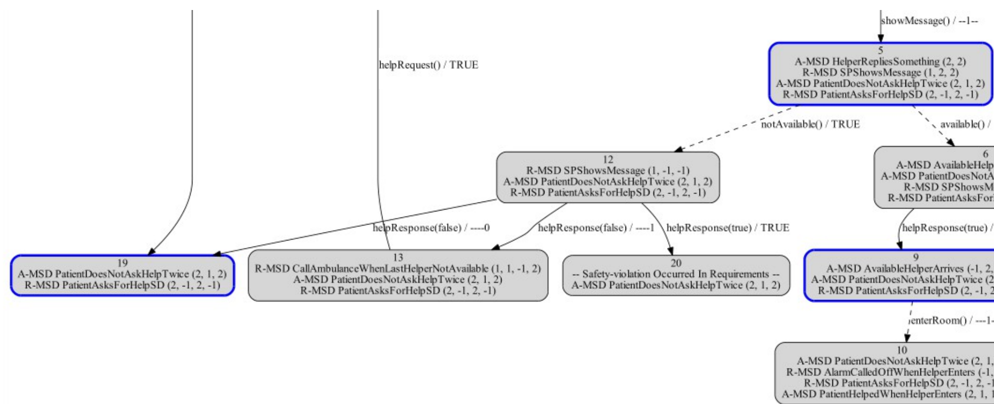


Figure 5.5: An excerpt of the FGG for the CoPAInS example.

message asking for his availability and before his reply. Two uncontrollable transitions exit from this state. If the helper is available to help no violation occurs. The helper will eventually help the patient and all the requirements in the specification will be fulfilled, even for the product not including Feature `CallAmbulance`. Otherwise the exploration reaches State (12). If the helper is unavailable, two different situations can occur. In case Feature `CallAmbulance` is included, State (13) is reached. Note that Transition «`helpResponse(false)`$\backslash----1$» represents the helper communicating his unavailability to the HCS when Feature `CallAmbulance` is included (1). Even in this case, all requirements are fulfilled, since the patient will be eventually helped by the ambulance. Instead, when the ambulance is not considered, State (19) is reached, which is a state without any outgoing transition, causing a liveness violation in the requirements. During the

backpropagation, as presented in Section 3.2.3, State (19) is labelled as winning for no product, since it does not reach any goal or winning state. In this case it does not reach any state at all. Instead, State (13) is winning, since it leads to a winning state. This is not shown here for brevity, but can be inferred from the whole FGG, which is given in Appendix A. State (12) is then labelled winning only for the combination of features including `CallAmbulance`, since it reaches a winning state, i.e. State (13), through a transition which is valid only when Feature `CallAmbulance` is considered. Through an additional backpropagating step, we analyze the winning condition for State (5). Its successors are State (12) and State (6), both reachable through uncontrollable transitions. In the uncontrollable case a state is winning for a product if all outgoing transitions from the considered state lead to a winning or a goal state for that product. In the example State (6) is winning for the whole SPL, whereas State (12) is winning only for the product which includes Feature `CallAmbulance`. Hence State (5) ends up winning for the only product for which both successors are winning, i.e., again, the one considering Feature `CallAmbulance`. This affects the winning condition of all previous states, until the algorithm reaches the initial state. Finally the algorithm finds that only the product including all features is realizable.

5.2 Performance Evaluation

In this section we assess the benefits of the featured synthesis against the successive checking of the individual products, also called *sub-optimal approach* hereafter. We further compare those results with the ones obtained through our previous iterative on-the fly approach.

All benchmarks were run on a Windows PC with a 2,4 GHz Intel Core 2 Duo processor and 4 GB of RAM.

5.2.1 CoPAInS and Vending Machine examples

In this first experiment we test the performance of the featured synthesis on both the CoPAInS and the Vending Machine examples.

We compare 100 synthesis runs of the featured algorithm with the successive synthesis of each product and the incremental synthesis of [Greenyer et Al. 2013]. Table 5.1(a) and Table 5.1(b) show the results of the comparison respectively for the CoPAInS and the Vending Machine examples. Each table provides the number of explored states and the time in milliseconds for each algorithm.

(a) CoPAInS example: 4 features, 2 products.

Algorithm	#States	avg Time (ms)
Featured Synthesis	20	25
Sub-optimal	35	43
Incremental OTF Synthesis	29	32

(b) Vending Machine example: 6 features, 5 products.

Algorithm	#States	avg Time (ms)
Featured Synthesis	16	10
Sub-optimal	46	34
Incremental OTF Synthesis	36	20

Table 5.2: Comparison on synthesis times and number of explored states for the featured synthesis, sub-optimal and incremental OTF synthesis approach.

We observe a noticeable improvement in the number of states explored by the synthesis. In the CoPAInS case 35 states are visited by the sub-optimal algorithms. This number decreases to 29 for our previous incremental approach. Only 20 states are visited by the featured synthesis. This already happens for a family of two products, when the second product is generated from the first one by just adding a feature to the combination. We also notice that in average 25 ms are taken by the featured synthesis, 32 ms by the incremental and 43 ms by the sub-optimal approach, which is an insignificant improvement.

The same comparison was run on the Vending Machine example and confirmed what we noticed for the CoPAInS case.

Although the realizability checking of both the CoPAInS and Vending Machine examples takes less than a second, this procedure can be very time-consuming for larger models. In the following, to further evaluate the benefit of our featured synthesis approach, we use some technical examples, which we systematically extend to create new examples with larger state spaces.

5.2.2 Comparison between featured and sub-optimal synthesis

In this first set of technical examples, named the *cascading example* hereafter, each SPL specification comes with an FD where each feature has at most two child features, connected with an OR-relationship. Features with distance i from the root may exist only if every feature with distance i - 2 in the diagram has exactly two child features. Each feature is connected with one MSD named after the feature. The first message of an MSD is a cold, monitored message

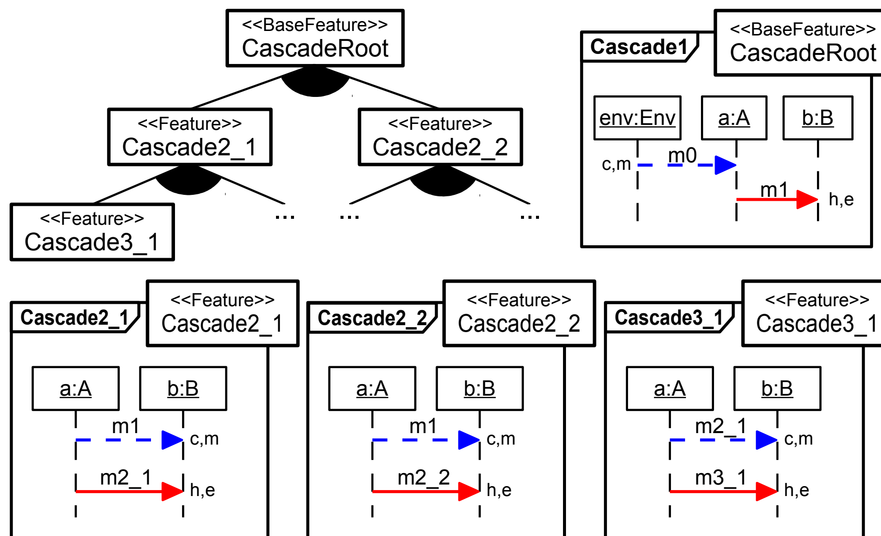


Figure 5.6: Cascading example with only one hot executed message per MSD.

and is followed by one hot, executed message. All messages are unifiable with system events except the first message in the MSD of the root feature, which is an environment message. The first message of a child feature MSD is named like the hot message of its parent's MSD. This way, one activation of an MSD triggers the activations of its child feature MSDs. The structure of our example is presented in Figure 5.6.

Based on that example, we compare the performance of the featured synthesis approach with respect to synthesizing every product independently. We test examples following the previous scheme, creating SPL specifications for 3, 5, 7, 9, 11, 13, and 15 features. The valid combinations of those features respectively lead to 3, 7, 15, 31, 63, 127, and 255 products, which represents an exponential increase in the number of products and different reachable combinations of cuts for each additional feature. We also created a second set of technical examples, based on a similar pattern. The only difference is that we relate a parent feature with its child sub-features with a XOR-decomposition instead of an OR. In this latter case the set of SPL specifications leads to a significantly lower number of products. Only 2, 3, 4, 5, 6, 7, 8 products are respectively created.

We repeated each experiment 20 times and computed the average of the synthesis times. Table 5.3 shows how synthesis time increases with respect to the number of features for our technical example. It provides the parent-child relationship, the number of features involved, the number of realizable products, and

for each approach the average number of explored states, the average synthesis time, and finally the speedup provided by the featured approach.

First, we notice that the benefits of the featured synthesis does not directly depend on the number of features, but rather on the number of their valid combinations. Cascade examples with 3 to 15 features were generated for both the XOR and the OR case. We already observe relevant improvements in the XOR case. The number of states explored and the average realizability checking time are almost the double when considering each product separately. However is when dealing with OR-decomposition and a higher number of products that the gain of the featured synthesis approach becomes remarkable. In the OR case, when synthesizing 255 products through the featured approach only 573 states are visited, against 22196 with the sub-optimal approach, and the process is almost 20 times faster (1260 ms against 25161 ms).

We conclude that the featured approach brings more noticeable improvements when the number of products grows more than linearly with the number of features.

Our interpretation of this result is that when the number of commonalities grows, such as in the OR case with respect to the XOR case, also the number of commonalities in the game graph of each product increases. Thus, exploring a global game graph instead of a set of different game graphs, allows the process to collapse common states, i.e. the reachable combinations of cuts, exploring a lower number of states and, consequently, saving synthesis time.

5.2.3 Comparison between featured and incremental on-the-fly synthesis

In this section the performance of the featured synthesis is measured against the incremental on-the-fly synthesis approach.

Before showing the results of our experiments, we intuitively explain the difference between an *on-the-fly* (OTF) and a *not-on-the-fly* (NOTF) synthesis. On-the-fly means that the algorithm only explores parts of the game graph. To capture the impact that such a difference could make in terms of number of states explored, we refer to the example shown in Figure 5.7.

Such an example consists of two mandatory features, which combined together form one single product. For simplicity, we do not consider families of products at this stage. The first feature contains one MSD composed of an environment message, which is cold and monitored, and an hot executed message. This latter message activates all the three MSDs in the second feature. The order in which messages m2_1, m2_2 and m2_3 are executed after message m1, is non-deterministic.

Table 5.3: Synthesis times for the cascading example. Comparison between the featured synthesis and the sub-optimal approach.

Example kind	#Features	#Products	#States		Time (ms)		Speedup
			Feat.	N-feat.	Feat.	N-feat.	
XOR	3	2	4	6	12	12	1
XOR	5	3	6	11	21	34	1.62
XOR	7	4	8	16	29	41	1.41
XOR	9	5	10	22	31	57	1.84
XOR	11	6	12	27	32	77	2.41
XOR	13	7	14	33	38	81	2.13
XOR	15	8	16	39	45	82	1.82
OR	3	3	5	11	12	15	1.25
OR	5	7	11	42	17	45	2.65
OR	7	15	26	158	79	219	2.77
OR	9	31	56	546	138	451	3.27
OR	11	63	111	1718	219	1459	4.88
OR	13	127	243	6060	508	5862	11.54
OR	15	255	573	22196	1260	25161	19.97

A NOTF exploration, such as the one used by our algorithms, explores any different path of the state space, considering any alternative ordering between messages which cannot be deterministically sent. For this example, after message m1, six alternative paths are explored, i.e. all possible orderings of messages m2_1, m2_2 and m2_3. Those paths all lead to the same state, which, in this particular case, is the initial state. The resulting game graph is shown in Figure 5.8(a) .

Conversely, an OTF exploration could walk trough only one path among all alternatives, avoiding to visit an important part of the game graph which, for such a contained example, is already almost a half of the total number of states (only five states are visited against nine explored during the NOTF synthesis). For those reasons OTF algorithms are typically more efficient that NOTF ones. The OTF approach is motivated by the semantics of the winning condition calculated when only controllable transitions are involved, as presented in Section 3.2.3. In case more than one controllable transition is exiting from a given sourcestate for the same product, it is enought that only one among those outgoing transitions leads to a winning or a goal state. Consequently, in case a state is reachable infinitely often through different alternative paths, it is enough to explore just one among those paths to retrieve the winning condition for the start state and thus investigate on the realizability of the respective specification. Figure 5.8(b) shows the game graph visited by the OTF exploration.

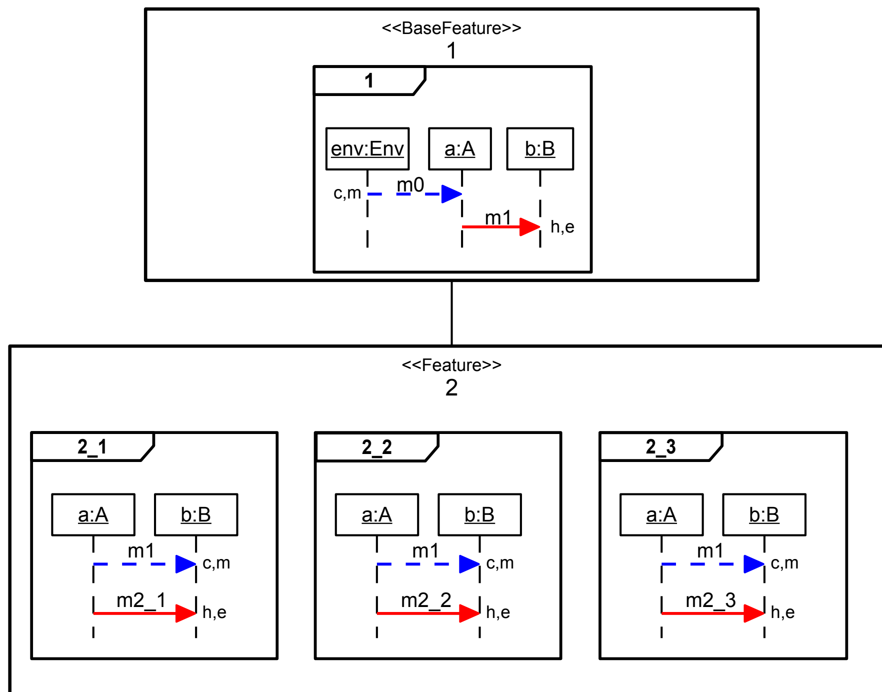


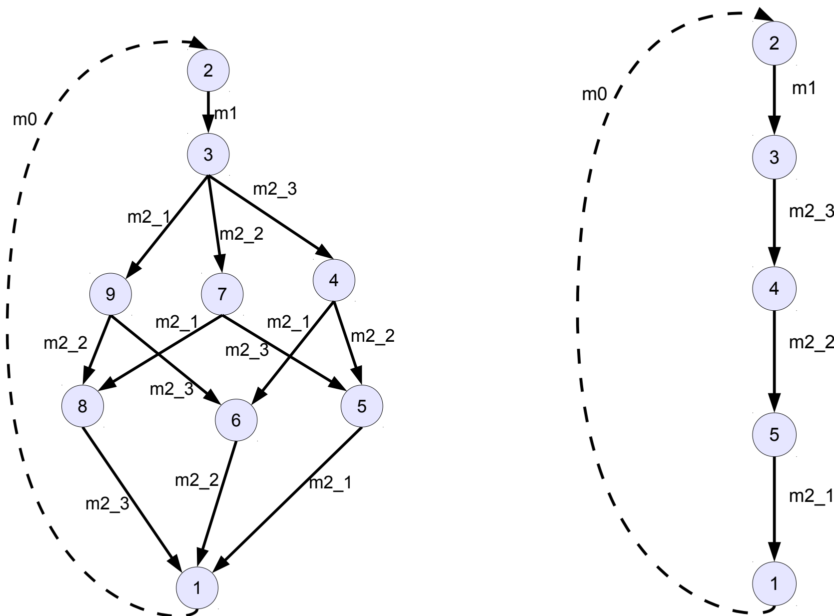
Figure 5.7: The specification used to show the difference between an OTF and a NOTF synthesis in terms of number of visited states.

Going more deeply into the OTF theory and explaining the techniques by which it could be implemented is not in the scope of this thesis. For this purpose, we remind the interested reader to [David et Al 2009] and [Cassez et Al 2005].

Cascading example with one hot executed message per MSD

We assess the performance of the featured synthesis against our incremental on-the-fly approach on the same set of technical examples sketched in Figure 5.6. Again we repeated each experiment 20 times, registered the number of explored states and computed the average of the synthesis times. The benchmark results for this second experiment are shown in Table 5.4.

We observe that the featured synthesis approach is still more efficient in most of the cases, even if the featured approach is not optimized to be OTF. Again, only slight improvements were registered by the featured synthesis in the XOR case, whereas when using an OR decomposition kind, more benefits can be observed. In case of 15 features, the featured approach achieves a speedup of almost 2. Although the enhancements with respect to the incremental on-



(a) States explored by a NOTF synthesis

(b) States explored by an OTF synthesis

Figure 5.8: The difference between an OTF and a NOTF synthesis in terms of number of visited states.

the-fly synthesis are not as remarkable as the ones brought by comparing the featured synthesis with the sub-optimal approach, which reached a speedup of almost 20, this is still a noticeable result. Furthermore the featured synthesis still outperforms in the number of states. When checking the realizability of the 255 products in the OR case with 15 features, only 573 against 2718 states are visited. This is a significant result considering that our algorithm is not on-the-fly.

Cascading example with two hot executed messages per MSD at level two

Interested about the outcome of our previous experiments, we investigate how far the featured NOTF methodology could go against an OTF approach. As a matter of fact, although the featured synthesis allows the realizability checking to save a lot of states during the exploration, being NOTF it still generates all possible successors of a given state, thus also paths that could be avoided by an OTF technique.

In this last experiment we modified the cascade example to intentionally in-

Table 5.4: Synthesis times for the cascading example. Comparison between the featured NOTF synthesis and the incremental OTF approach.

Example kind	#Features	#Products	#States		Time (ms)		Speedup
			NOTF-Feat.	OTF-Incr.	NOTF-Feat.	OTF-Incr.	
XOR	3	2	4	6	12	12	1
XOR	5	3	6	11	21	14	0.67
XOR	7	4	8	16	29	20	0.69
XOR	9	5	10	22	31	25	0.8
XOR	11	6	12	27	32	29	0.9
XOR	13	7	14	33	38	39	1.02
XOR	15	8	16	39	45	46	1.02
OR	3	3	5	10	12	11	0.92
OR	5	7	11	32	17	15	0.88
OR	7	15	26	86	79	96	1.21
OR	9	31	56	218	138	197	1.43
OR	11	63	111	506	219	341	1.56
OR	13	127	243	1190	508	834	1.64
OR	15	255	573	2718	1260	2267	1.8

crease the number of alternative paths that an OTF approach could avoid, to further stress our featured approach and see until which point it can outperform the incrementally OTF synthesis. For this purpose, a second hot, executed message was added to the MSD in features at level 2. This way MSDs at level 2 are activated by one cold, monitored message which triggers two hot, executed messages with the same name. Thus, the activation of an MSD in a feature at level 2 triggers two activations of an MSD for each child feature. Consequently, after the first hot, executed message is sent, the algorithm can non-deterministically choose between more than one alternative path, where every path is formed by one among the different ways in which the events belonging to the two activations could be interleaved. To better explain this concept consider Figure 5.9, which illustrates the modified structure. Consider the variant with 3 features composed by `CascadeRoot`, `Cascade2_1` and `Cascade2_2`. When considering the product with both children `Cascade2_1` and `Cascade2_2`, after m1 is sent, the NOTF algorithm explores all the six alternative paths which are:

- {{m2_1}{m2_1}{m2_2}{m2_2}};
- {{m2_1}{m2_2}{m2_1}{m2_2}};
- {{m2_2}{m2_2}{m2_1}{m2_1}};
- {{m2_2}{m2_1}{m2_2}{m2_1}};
- {{m2_1}{m2_2}{m2_2}{m2_1}};

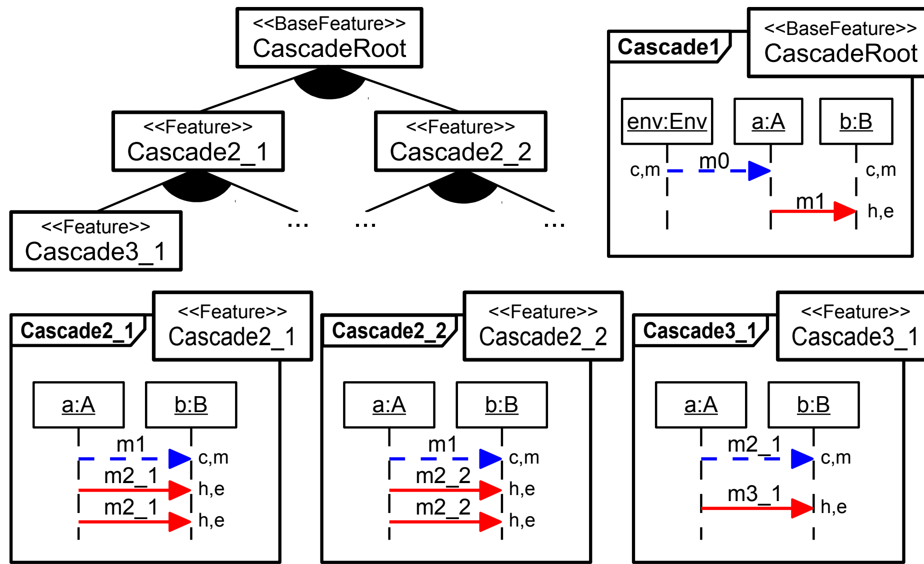


Figure 5.9: Cascading example with two hot executed messages per MSD for features at level 2.

- {{m2_2}{m2_1}{m2_1}{m2_2}}.

The OTF algorithm could avoid five out of those six paths.

Again, following the modified structure in Figure 5.9, we created SPL specifications for 3, 5, 7, 9, 11, 13, 15 features, executed both featured synthesis and incremental OTF synthesis on each specification, repeated each experiment 20 times, registered the number of explored states and computed the average of the synthesis times. The results for this final experiment are shown in Table 5.5.

We observe that the ratio feature-to-MSD remains the same, as well as the number of products generated. However we obtained significantly different results than in the previous experiments.

We notice that for 3, 5, 7, and 9 features the featured synthesis still performs better both on the number of states explored and on synthesis time. Consider again the case including 3 features. Figures 5.10(a) and 5.10(b) shows the comparison between the graphs that would be generated by a NOTF resp. OTF algorithm for the variant involving both child features. As previously noticed, the OTF approach could avoid five out of six paths. Now consider the whole SLP specification composed of 3 product variants, corresponding to the first line of Table 5.5. Figures 5.10(c) and 5.10(d) show the comparison between the featured game graph generated by the featured synthesis and the three graphs generated by the OTF synthesis. We observe that the number of states visited to check the realizability of the variant formed by all features is the same as the

Table 5.5: Synthesis times for the modified cascading example. Comparison between the featured NOTF synthesis and the incremental OTF approach.

Example kind	#Features	#Products	#States		Time (ms)		Speedup
			NOTF-Feat.	OTF-Incr.	NOTF-Feat.	OTF-Incr.	
OR (2mex lev.2)	3	3	10	14	43	108	2.51
OR (2mex lev.2)	5	7	29	50	197	250	1.27
OR (2mex lev.2)	7	15	83	142	383	450	1.18
OR (2mex lev.2)	9	31	245	374	712	805	1.13
OR (2mex lev.2)	11	63	704	886	1309	1087	0.83
OR (2mex lev.2)	13	127	2108	2126	7388	5237	0.7
OR (2mex lev.2)	15	255	7022	4926	23255	13364	0.57

number used to check the realizability of the set of three variants. Only two transitions, marked in red in the figure, are added to the graph. Conversely, even if exploring only parts of the whole state space for each variant, when considering all variants the incremental OTF synthesis visits a higher number of states.

However, when the number of avoidable paths grows, as in the case of 15 features, the incremental OTF technique starts performing better, even in the number of explored states.

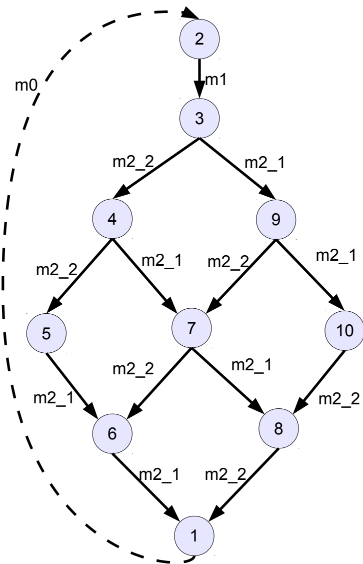
Our interpretation of this trend is that as the number of alternative paths grows, the benefits of avoiding the exploration of those paths become more significant than the reuse of those states for different variants.

5.2.4 Discussion

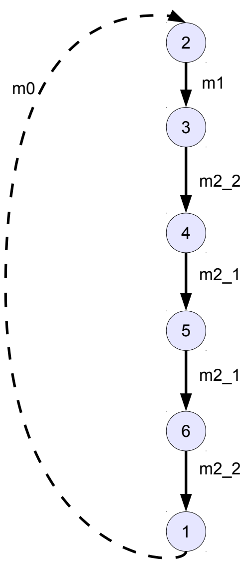
This section concludes our evaluation summarizing the benchmarks obtained through our experiments.

Figure 5.11 presents a graphical comparison of the number of states explored and the synthesis times for the cascading example, when only one hot, executed message is included in every MSD. The experiment shows remarkable improvements against both the sub-optimal and the Incremental OTF synthesis. In this latter case the results are particularly valuable given that the featured synthesis is not optimized to be OTF.

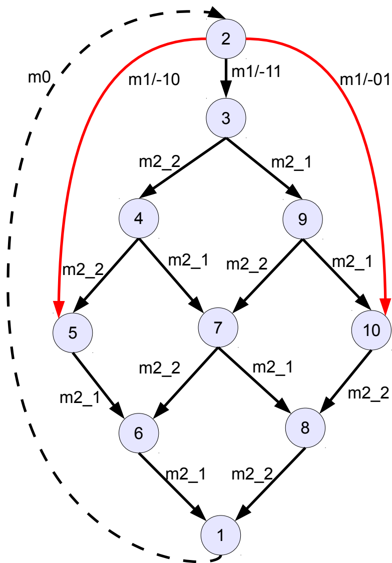
We further investigated the possible benefits of the featured synthesis against the Incremental OTF algorithm through a second set of technical experiments, enforcing the number of alternative paths in the game graph. For this purpose, we modified the cascade example adding a second hot, executed message in MSDs at level 2. As expected this latter experiment revealed that as the number of features grows, the chance to have different alternative paths in the game graph also increases. At some point the benefit given by avoiding a large number of parts



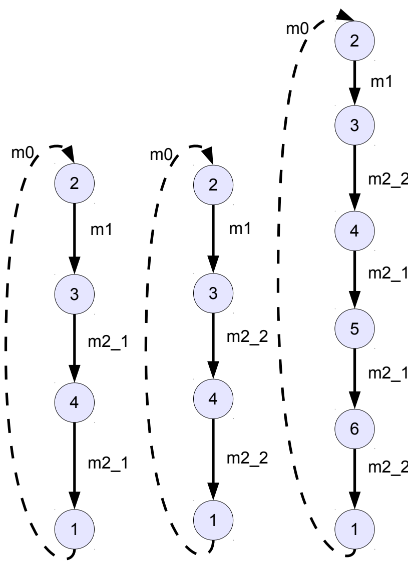
(a) NOTF synthesis on one variant from the modified cascade example with 3 features.



(b) OTF synthesis on one variant from the modified cascade example with 3 features.

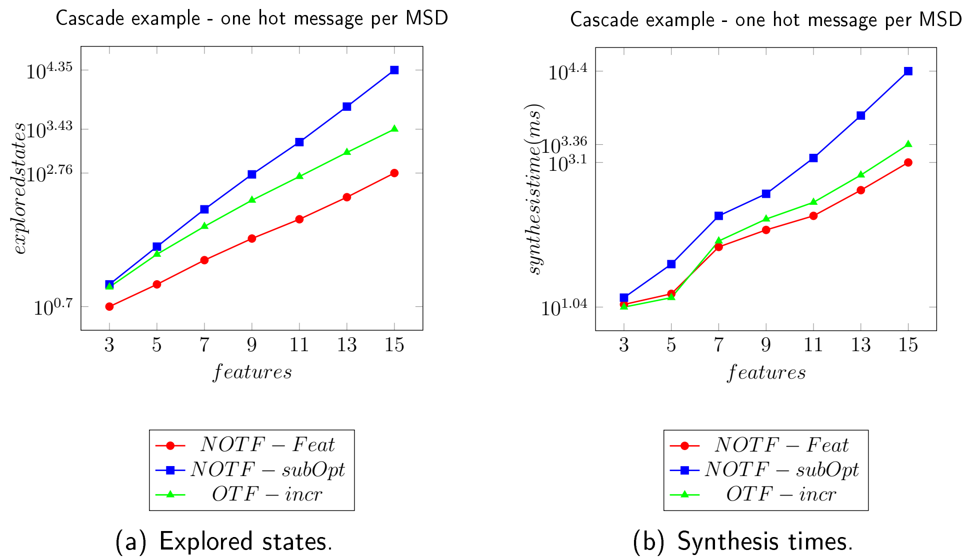


(c) Featured NOTF synthesis on all three variants from the modified cascade example with 3 features.



(d) OTF synthesis on all three variants from the modified cascade example with 3 features.

Figure 5.10: The difference between an incremental OTF and a featured NOTF synthesis in terms of number of visited states.



(a) Explored states.

(b) Synthesis times.

Figure 5.11: Cascade example with one hot, executed message per MSD. Synthesis times and number of states explored for the three approaches.

in the graph becomes higher than the advantage of exploiting the commonalities between the game graphs of each variant. At that point the featured synthesis approach looses efficiency against the incremental approach, which is OTF.

Chapter 6

Conclusion and Outlook

This thesis presented a novel approach to check the realizability of scenario-based Software Product Line (SPL) specifications.

For this purpose, we adopted SPL specifications [Greenyer 2011], a recent intuitive, yet precise way to model the behavior of a set of software products as a combination of Feature Diagrams (FD) [Kang et al. 1990] and Modal Sequence Diagrams (MSD) [Harel and Maoz 2008, Greenyer 2011]. We gave a comprehensive overview of the essential concepts related to SPLs and all the models employed by our approach in Chapter 2.

In order to find inconsistencies that may arise from the specification of a family of products, we managed the problem of checking the realizability of an SPL specification as the problem of finding a strategy in an infinite game played by the system against the environment. Our technique consists of an extension of the algorithm for solving Büchi games [David et Al 2009] which are games requiring to infinitely often reach a state with given characteristics. When applying the original technique on a single product MSD specification, the game appears in the form of a game graph, a Büchi automaton that accepts all infinite sequences of steps that respect the safety and liveness properties of the MSDs. We exploited the fact that if variants are similar, the game graphs induced by each product's specification, are also likely to be similar. Thus we implemented a methodology to derive a featured game graph (FGG), i.e. a global state graph representing all possible executions of all possible products with any environment, which maintains a link between a given execution and the features needed to trigger it. The main contribution of our work, is that using the information contained in the FGG, our technique provides a way to determine which products have a consistent specification by means of only one synthesis instead of performing a separate checking on each product. We presented the featured synthesis algorithms providing those functionalities in Chapter 3.

We implemented our methodology in ScenarioTools, a collection of Eclipse-

based tools which support the modeling, simulation and synthesis of SPL specifications. Our plug-in allows us to run the featured synthesis on an SPL specification, as previously modeled in ScenarioTools, to check its realizability. At the end of the process, our extension provides the set of realizable products in the SPL specification. It also allows users to generate the corresponding FGG used throughout the synthesis. We explained the main architecture of our implementation and provided an overview of its usage in Chapter 4.

We verified the applicability of our approach on two practical case studies, testing the ability to recognize partially inconsistent specifications. We also assessed the efficiency of the featured synthesis against the successive checking of the individual products and our previous iterative on-the-fly (OTF) approach [Greenyer et Al. 2013], using different technical evaluation examples. The experiments showed remarkable improvements against the former, both on number of states and synthesis times. In some cases our technique performed even 20 times better than checking each product separately. The comparison with the iterative OTF synthesis also brought noticeable results. When comparing the algorithms on a specification from which not many alternative paths in the game graph are produced, the featured synthesis performed almost 2 times better than the iterative approach in the synthesis time, and over 4 times better in the number of explored states. However, when many of those alternative paths are introduced in the graph, the incremental approach starts performing better for specifications with larger state spaces. Our interpretation of the results is that, being OTF, the latter can avoid those alternative paths. Consequently, it performs better when avoidable graph paths comprise a number of states that is higher than the number of states which can be collapsed from the game graphs of each variant. We dealt with applicability and performance evaluation in Chapter 5.

We identify three possible directions for future work. The first concerns a perspective to overcome a drawback of our plug-in, which does not provide the simulation for the whole SPL specification. Simulation currently only supports the execution of a single product's MSD specification via the play-out algorithm [Harel and Marelly 2002]. Although the tool allows us to run the simulation on the SPL specification, the only target of the process is the product composed of all the feature specifications. One way to solve this issue would be to let users select the features to include in the specification, i.e. defining the product. Then, from the FGG, one should be able to extract the corresponding controller and execute it.

Another minor limitation of our technique is that when generating the feature expression representing the set of valid products in the FD, it does not consider cross-tree constraints. This fix should be easily applicable by extending Algorithm 1 to consider the respective constraints.

The third, major challenge, is to optimize the featured synthesis to be on-the-

fly. Supported by our benchmarks, we believe that impressive results could be achieved in comparison to the incremental OTF technique, which still represents the most efficient alternative for more complex SPL specifications.

Appendix A

Whole SPL specification and FGG for the CoPAInS example

This appendix provides the screenshots of all diagrams modeled in ScenarioTools to assess the applicability of the featured synthesis on the CoPAInS example.

A.1 SPL specification

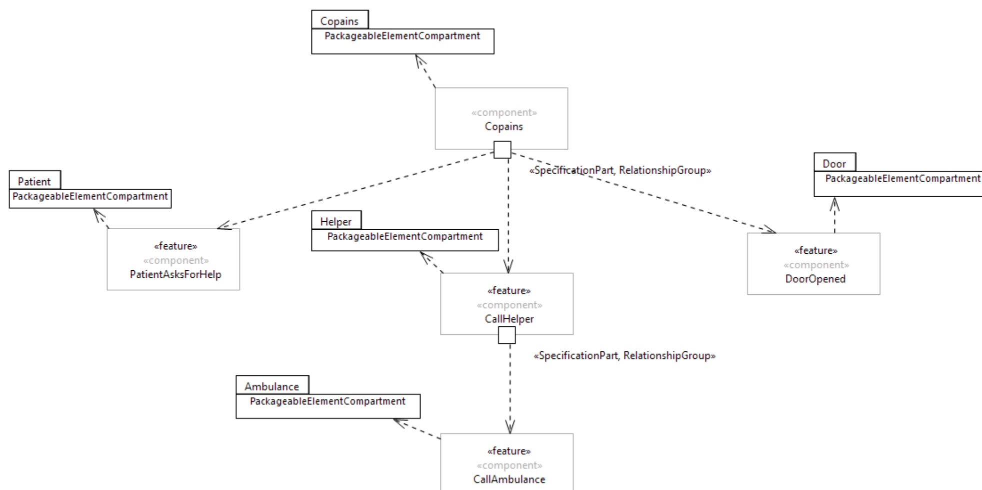


Figure A.1: FD representing the valid combinations of features for the CoPAInS SPL.

In the following, the system requirements and environment assumption represented by each MSD are described informally in the respective caption.

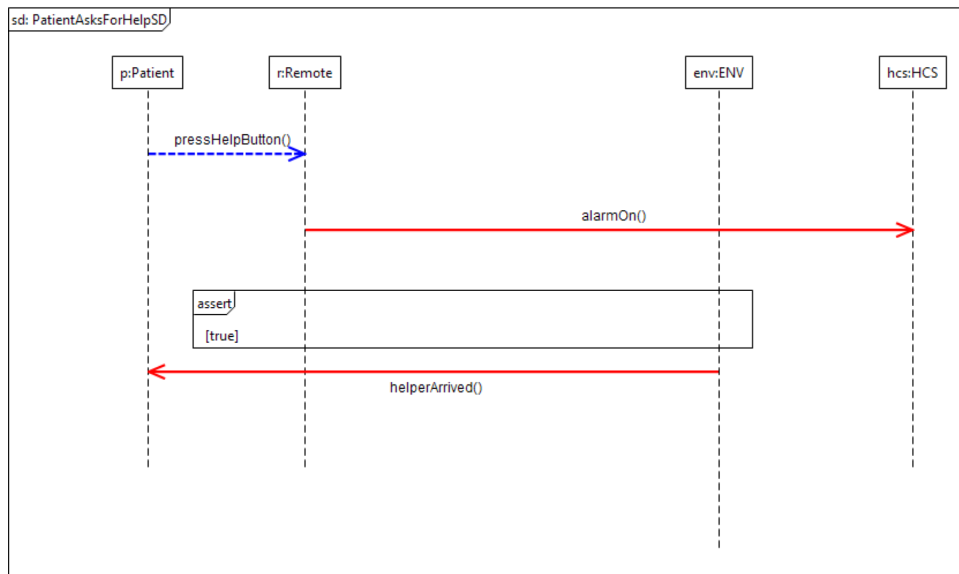


Figure A.2: **PatientAsksForHelp R1)** (liveness requirement) After the patient asks for help the alarm is set on and eventually the patient must be helped.

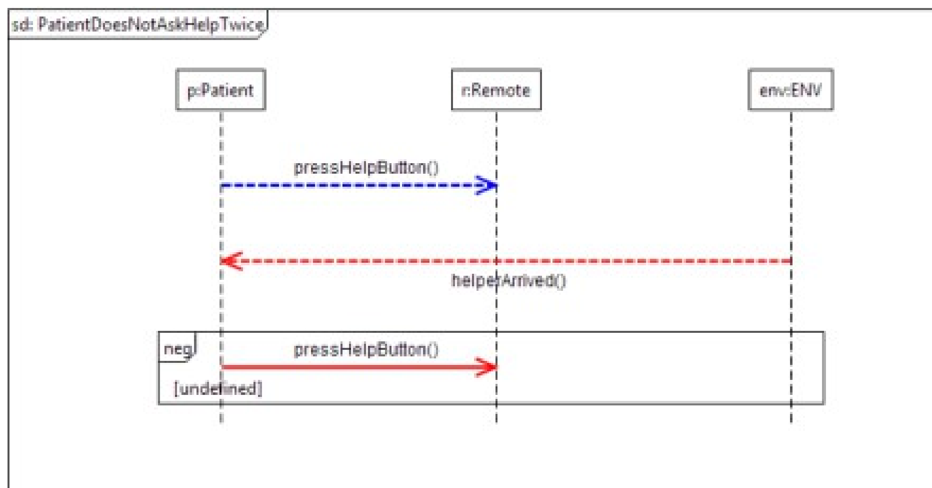


Figure A.3: **PatientAsksForHelp A1)** We assume that the patient does not ask for help twice before the alarm is called off.

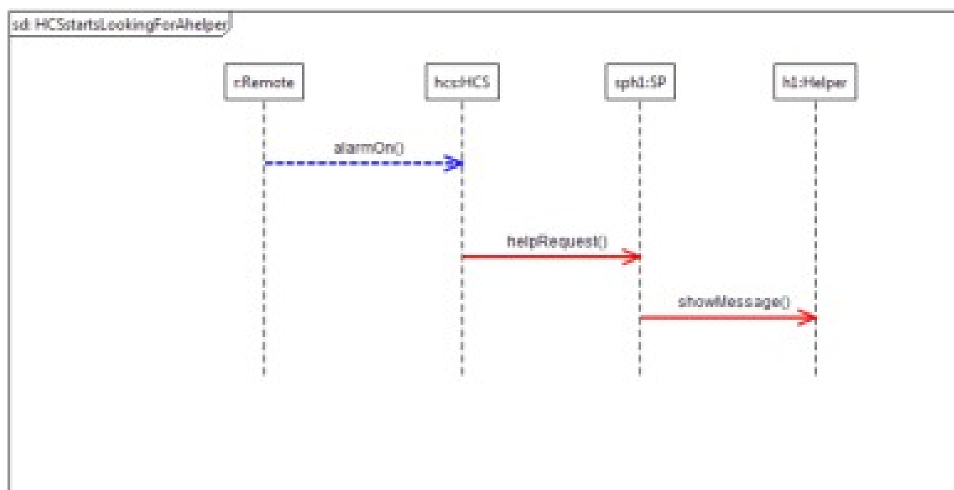


Figure A.4: **CallHelper R1)** After the alarm is set on, the HCS sends an help request to the smartphone of the first helper in the list. The smartphone shows the message to its owner.

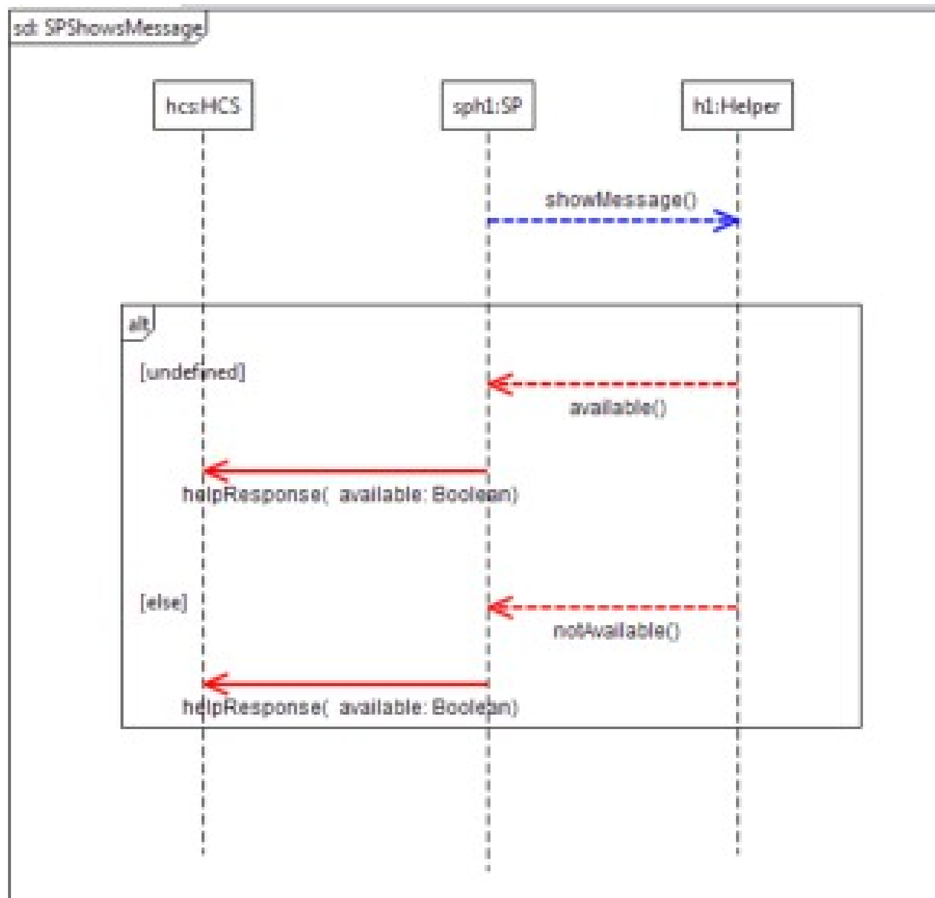


Figure A.5: **CallHelper R2)** After the helper reads the message, he could reply YES or NO. In the first case the smartphone tells the HCS that the first helper is available to help. Otherwise the HCS is informed that the helper is not available.

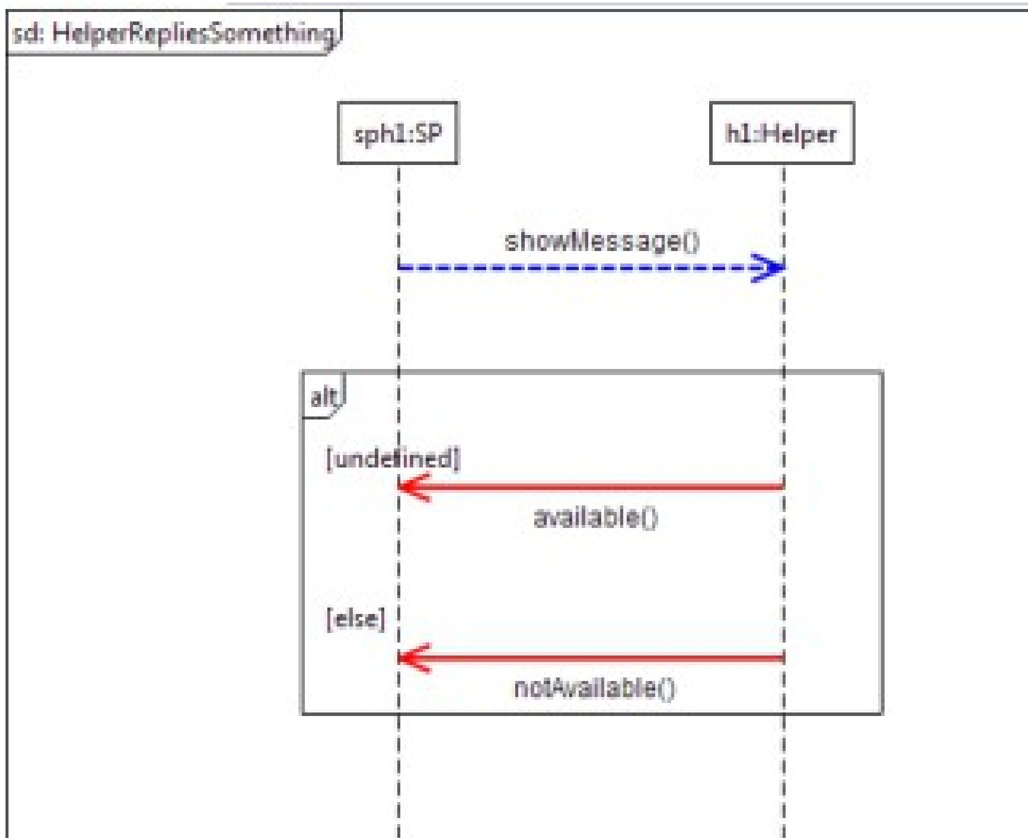


Figure A.6: **CallHelper A1)** We assume that after the smartphone shows the message to the helper, either he answers with his availability or with his unavailability. Through this assumption, we basically rule out the scenario in which the helper does not reply at all.

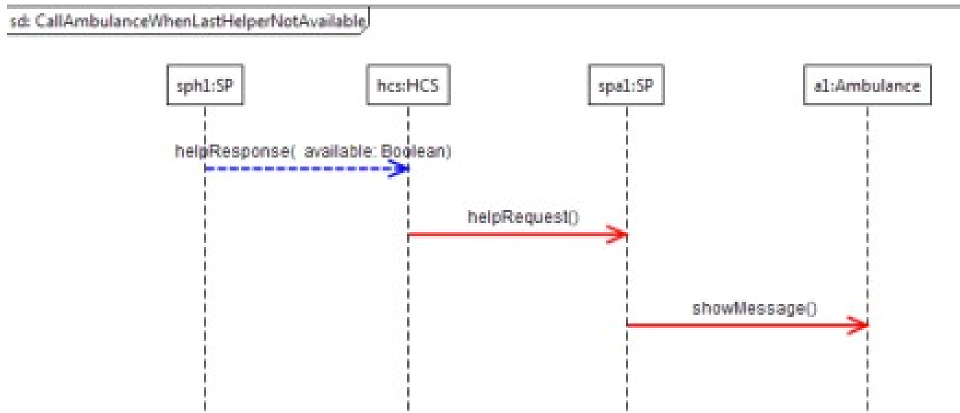


Figure A.7: **CallAmbulance R1)** Whenever the last helper is not able to help, the HCS sends a request to the ambulance.

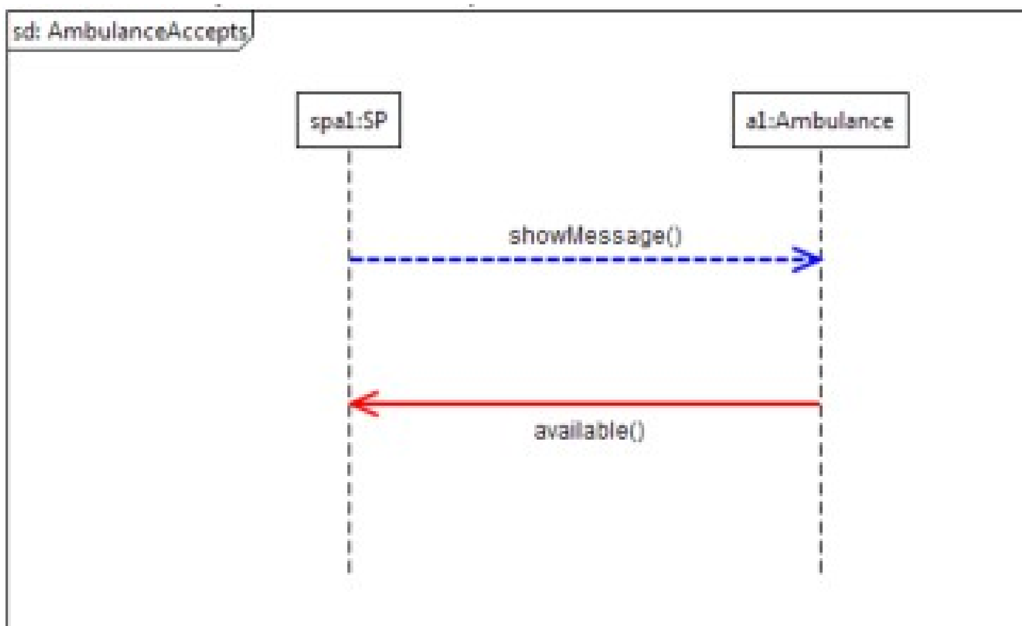


Figure A.8: **CallAmbulance A1)** The ambulance is always available to help.

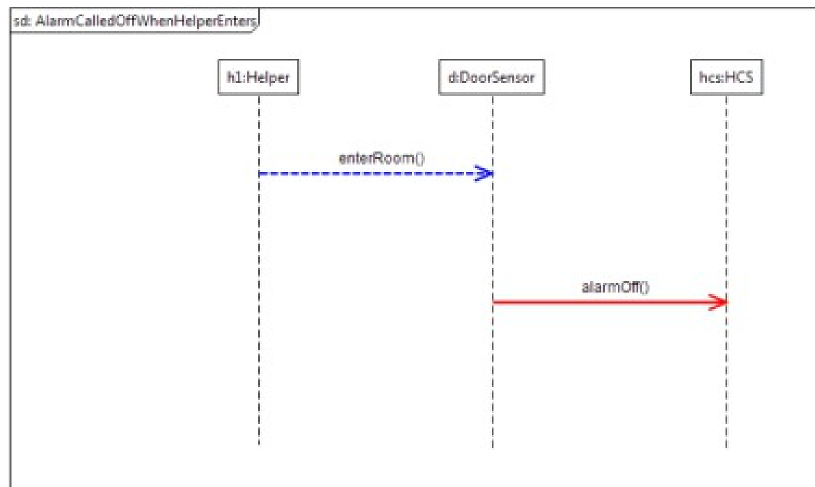


Figure A.9: **DoorOpened R1)** When the helper enters the room, the alarm is set off.

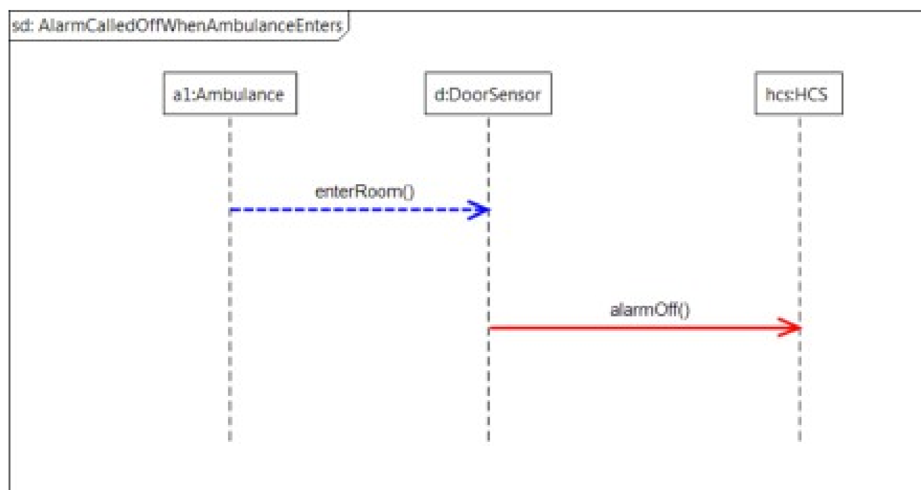


Figure A.10: **DoorOpened R2)** When the ambulance enters the room, the alarm is set off.

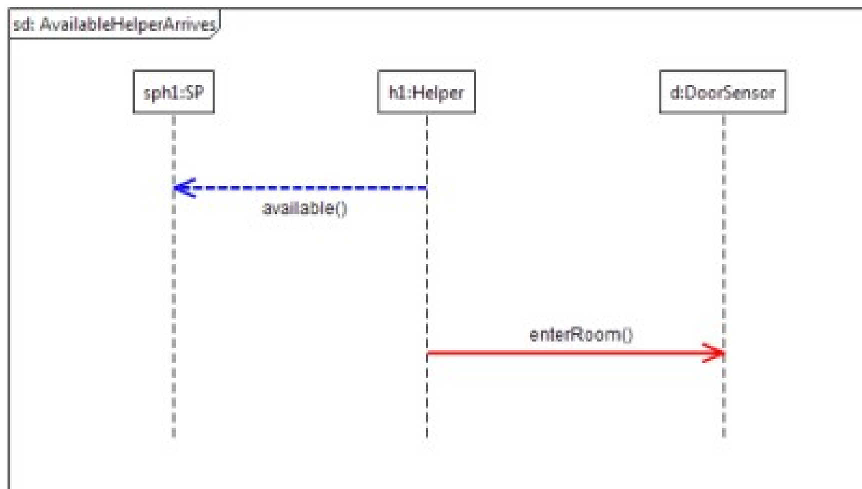


Figure A.11: **DoorOpened A1)** We assume that when the helper replies that he is available to help, he will eventually show up.

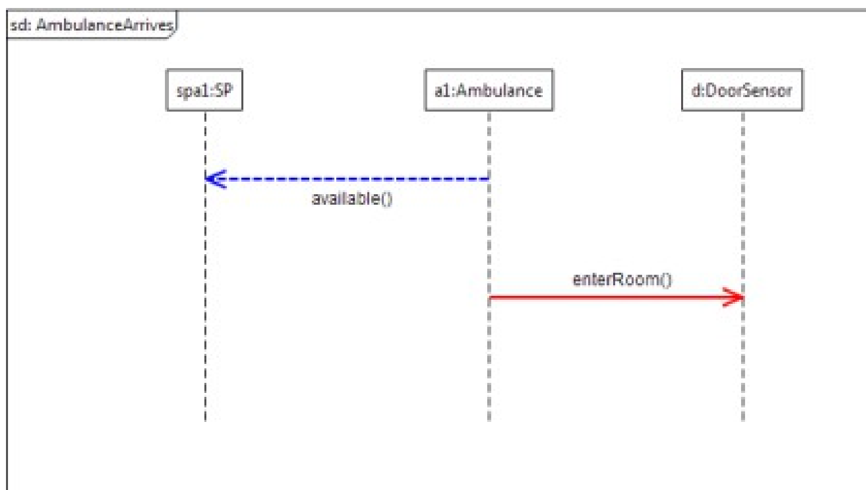


Figure A.12: **DoorOpened A2)** We assume that when the ambulance replies that it is available to help, he will eventually show up.

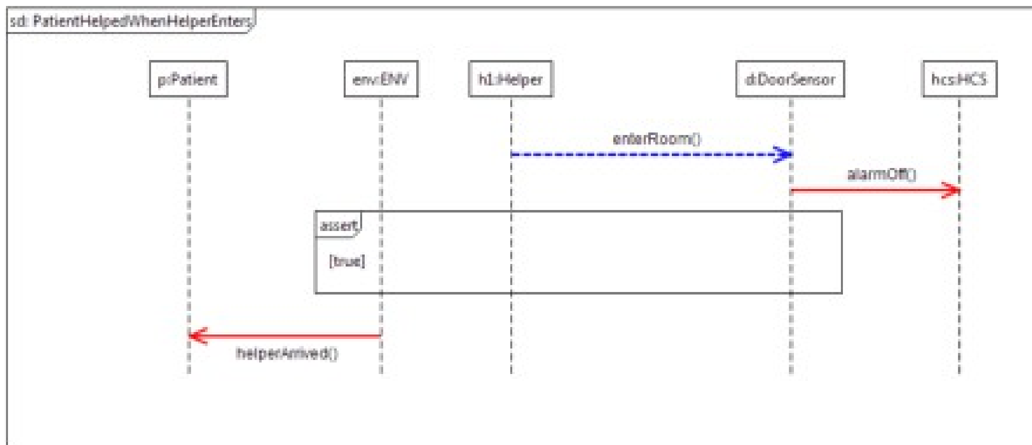


Figure A.13: **DoorOpened A3)** When the helper enters the room, after the alarm is called off, the patient is helped.

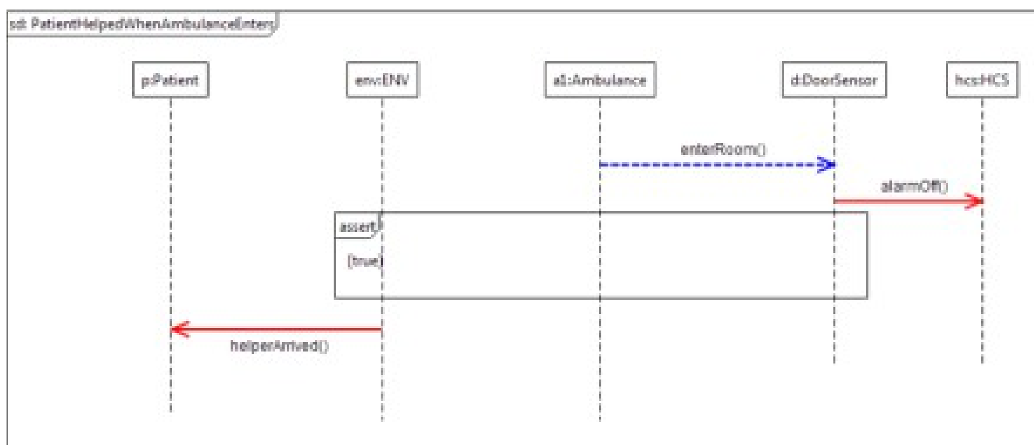


Figure A.14: **DoorOpened A4)** When the ambulance enters the room, after the alarm is called off, the patient is helped.

A.2 Featured Game Graph

Figure A.15 shows the whole FGG generated by the featured synthesis.

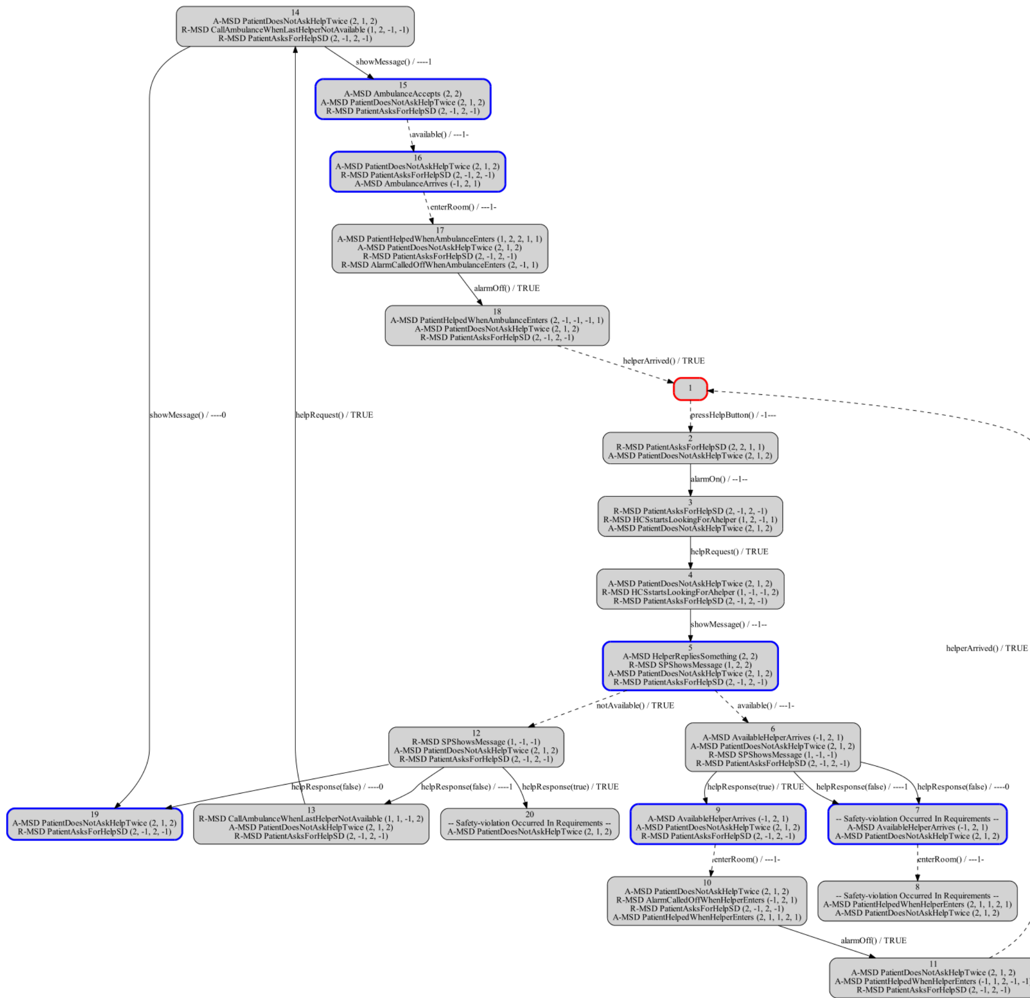


Figure A.15: FGG generated when exploring the state space of the CoPAInS SPL specification.

Bibliography

[Abadi et Al. 1989] Abadi, M., Lamport, L., and Wolper, P. (1989). *Realizable and unrealizable specifications of reactive systems.* In Automata, languages and programming (pp. 1-17). Springer Berlin Heidelberg.

[Akers 1978] Akers, S. B. (1978). *Binary decision diagrams*. Computers, IEEE Transactions on, 100(6), 509-516.

[Alizon et Al. 2009] Alizon, F., Shooter, S. B., and Simpson, T. W. (2009). *Henry Ford and the Model T: lessons for product platforming and mass customization*. Design Studies, 30(5), 588-605.

[Andersen 1997] Andersen, H. R. (1997). *An introduction to binary decision diagrams*. Lecture notes, available online, IT University of Copenhagen.

[Baier and Katoen 2008] Baier, C., and Katoen, J. P. (2008). *Principles of model checking* (Vol. 26202649). Cambridge: MIT press.

[Benington 1983] Benington, H. D. (1983). *Production of large computer programs*. Annals of the History of Computing, 5(4), 350-361.

[Boehm 1988] Boehm, B. W. (1988). *A spiral model of software development and enhancement*. Computer, 21(5), 61-72.

[Bontemps et Al. 2004] Bontemps, Y., Schobbens, P. Y., and Löding, C. (2004). *Synthesis of open reactive systems from scenario-based specifications.* Fundamenta Informaticae, 62(2), 139-169.

[Bontemps et Al. 2005] Bontemps, Y., Heymans, P., and Schobbens, P. Y. (2005). *From live sequence charts to state machines and back: a guided tour.* Software Engineering, IEEE Transactions on, 31(12), 999-1014.

[Bryant 1992] Bryant, R. E. (1992). *Symbolic Boolean manipulation with ordered binary-decision diagrams*. ACM Computing Surveys (CSUR), 24(3), 293-318.

[Bryant 1995] Bryant, R. E. (1995, November). *Binary decision diagrams and beyond: Enabling technologies for formal verification.* In Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on (pp. 236-243). IEEE.

[Burch et AL 1992] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). *Symbolic model checking: 10< sup> 20</sup> states and beyond.* Information and computation, 98(2), 142-170

[Caire 2012] Caire, P. (2012). *Diagram Analysis Report: Use Cases for Conviviality and Privacy in Ambient Intelligent Systems.*

[Cassez et Al 2005] Cassez, F., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2005). *Efficient on-the-fly algorithms for the analysis of timed games.* In CONCUR 2005-Concurrency Theory (pp. 66-80). Springer Berlin Heidelberg.

[Classen 2011] Classen, A. (2011). *Modelling and model checking variability-intensive systems* (Doctoral dissertation, Ph. D. dissertation).

[Classen et Al 2008] Classen, A., Heymans, P., and Schobbens, P. Y. (2008). *What's in a feature: A requirements engineering perspective.* In Fundamental Approaches to Software Engineering (pp. 16-30). Springer Berlin Heidelberg.

[Classen et al. 2010] Classen, A., Heymans, P., Schobbens, P. Y., Legay, A., and Raskin, J. F. (2010, May). *Model checking lots of systems: efficient verification of temporal properties in software product lines.* In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (pp. 335-344). ACM.

[Classen et al. 2011] Classen, A., Heymans, P., Schobbens, P. Y., and Legay, A. (2011, May). *Symbolic model checking of software product lines.* In Proceedings of the 33rd International Conference on Software Engineering (pp. 321-330). ACM.

[Clements and Northrop 2001] Clements, P., and Northrop, L. (2001). *Software Product Lines: Patterns and Practice.* Addison Wesley.

[Coplien 1999] Coplien, J. O. (1999). *Multi-paradigm Design for C+.* Addison-Wesley.

[Cordy et al. 2013] Cordy, M., Classen, A., Heymans, P., Legay, A., and Schobbens, P. Y. (2013). *Model checking adaptive software with featured*

transition systems. In Assurances for Self-Adaptive Systems (pp. 1-29). Springer Berlin Heidelberg.

[Czarnecki et al. 2005] Czarnecki, K., Helsen, S. and Eisenecker, U. *Staged Configuration Using Feature Models.* Software Process Improvement and Practice, special issue on Software Variability: Process and Management, 10(2):143 – 169, 2005.

[David et Al 2009] David, A., Behrmann, G., Bulychev, P., Byg, J., Chatain, T., Larsen, K. G., Pettersson, P., Rasmussen, J. I. , Srba, J. , Yi, W. , Joergensen, K. Y. , Lime, D. , Magnin, M. , Roux, O. H. and Traonouez, L. M. (2009). *Tools for Model-Checking Timed Systems*. Communicating Embedded Systems: Software and Design: Formal Methods, 165-225.

[Davis 1987] Stanley, D. M. (1987). *Future perfect*. Mass.: Addilson-Wesley, 157.

[Efthymiou et Al 2012] Efthymiou, V., Caire, P., and Bikakis, A. (2012). *Modeling and evaluating cooperation in multi-context systems using conviviality.* In Proceedings of BNAIC 2012 The 24th Benelux Conference on Artificial Intelligence.

[Fantechi and Gnesi 2008] Fantechi, A., and Gnesi, S. (2008, September). *Formal modeling for product families engineering.* In Software Product Line Conference, 2008. SPLC'08. 12th International (pp. 193-202). IEEE.

[Fey et Al 2004] Fey, G., Shi, J., and Drechsler, R. (2004, August). *BDD circuit optimization for path delay fault testability.* In Digital System Design, 2004. DSD 2004. Euromicro Symposium on (pp. 168-172). IEEE.

[Ghezzi et Al 2012] Ghezzi, C., Greenyer, J., and Manna, V. P. L. (2012, June). *Synthesizing dynamically updating controllers from changes in scenario-based specifications.* In Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on (pp. 145-154). IEEE.

[Greenyer 2011] Greenyer, J. *Scenario-based design of mechatronic systems*, Ph.D. dissertation, University of Paderborn, Oct. 2011.

[Greenyer et al. 2011] Greenyer, J., Sharifloo, A. M., Cordy, M., and Heymans, P. (2012, September). *Efficient consistency checking of scenario-based product-line specifications.* In Requirements Engineering Conference (RE), 2012 20th IEEE International (pp. 161-170). IEEE.

[Greenyer et Al. 2013] Greenyer, J., Brenner, C., Cordy, M., Heymans, P., and Gressi, E. (2013, August). *Incrementally synthesizing controllers from scenario-based product line specifications.* In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (pp. 433-443). ACM.

[Harel and Kugler. 2002] Harel, D., and Kugler, H. (2002). *Synthesizing state-based object systems from LSC specifications.* International Journal of Foundations of Computer Science, 13(01), 5-51.

[Harel and Marelly 2002] Harel, D. and Marelly, R. *Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach*. Software and System Modeling (SoSyM), 2:2003, 2002.

[Harel and Marelly 2003] Harel, D. and Marelly, R. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, Aug. 2003.

[Harel and Maoz 2008] Harel, D. and Maoz, S. *Assert and negate revisited: Modal semantics for UML sequence diagrams*. Software and Systems Modeling (SoSyM), 7(2):237–252, May 2008.

[Harel and Pnueli 1985] Harel, D., and Pnueli, A. (1985). On the development of reactive systems (pp. 477-498). Springer Berlin Heidelberg.

[Harel et al. 2002] Harel, D., Kugler, H., Marelly, R. and Pnueli, A. *Smart Play-Out of Behavioral Requirements*, In Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, (pp. 378-398).

[IEEE-Std-1471-2000] Hilliard, R. (2000). *Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems.* IEEE, http://standards. ieee. org.

[Kang et al. 1990] Kang, K., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990

[Kang et al. 1998] K. C. Kang, K., Kim, S., Lee, J. and Kim., K. *FORM: A Feature-Oriented Reuse Method.* In Annals of Software Engineering 5, pages 143–168, 1998.

[Larman and Basili 2003] Larman, C. and Basili, V. R. (2003). *Iterative and incremental developments. a brief history*. Computer, 36(6), 47-56.

[Maler et Al 1995] Maler, O., Pnueli, A., and Sifakis, J. (1995, January). *On the synthesis of discrete controllers for timed systems.* In STACS 95 (pp. 229-242). Springer Berlin Heidelberg.

[Maoz 2009] Maoz, S. *Polymorphic Scenario-Based Specification Models: Semantics and Applications*, In Andy Schürr and Bran Selic, editors, Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MODELS 2009, Denver, CO, USA, Oct. 4-9, 2009., volume 5795 of Lecture Notes in Computer Science, pages 499–513. Springer, 2009.

[Maoz and Harel 2006] Maoz, S. and Harel, D. *From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ*. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006, pages 219–230, 2006.

[Moawad et Al 2012] Moawad, A., Efthymiou, V., Caire, P., Nain, G., and Le Traon, Y. (2012). *Introducing conviviality as a new paradigm for interactions among IT objects.* In Proceedings of the Workshop on AI Problems and Approaches for Intelligent Environments (Vol. 907). CEUR-WS. org.

[Northrop 2002] Northrop, L. M. (2002). *SEI's software product line tenets*. Software, IEEE, 19(4), 32-40.

[OMG OCL 2012] Object Management Group (OMG). *OMG Object Constraint Language (OCL)* Version 2.3.1, January 2012.

[OMG UML 2011] Object Management Group (OMG). *OMG Unified Modeling Language TM (OMG UML), Superstructure* Version 2.4.1, August 2011.

[Pnueli et Al 1998] Pnueli, A., Asarin, E., Maler, O., and Sifakis, J. (1998). *Controller synthesis for timed automata.* In Proc. System Structure and Control. Elsevier.

[Pohl et al. 2001] Pohl, K., Böckle, G., Clements, P., Obbink, H., and Rombach, D. (2001). *Product Family Development* Proceedings Dagstuhl Seminar, University of Essen, Germany.

[Pohl et al. 2005] Pohl, K., Böckle, G., and Van Der Linden, F. (2005). *Software product line engineering: foundations, principles, and techniques*. Springer.

[Possompes et Al 1998] Possompes, T., Dony, C., Huchard, M., and Tibermacine, C. (2011, July). *Design of a UML profile for feature diagrams and its*

tooling implementation. In Proceedings of the Twenty-Third International Conference on Software Engineering & Knowledge Engineering (pp. 693-698).

[Schobbens et al. 2006] Schobbens, P.-Y, Heymans, P., Trigaux, J.-C. and Bontemps, Y. *Feature Diagrams: A Survey and A Formal Semantics.* RE'06, pages 139–148, 2006.

[Schobbens et al. 2007] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. *Generic semantics of feature diagrams.* Computer Networks, Special Issue on Feature Interactions in Emerging Application Domain, 51(2):456–479, 2007.

[Weiss and Lai 1999] Weiss, D.M. and Lai, C.T.R. (1999). *Software Product-Line Engineering - A Family-Based Software Development Process*. Addison-Wesley, Reading, Massachusetts.

[Wiedenhaupt et Al. 1998] Wiedenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998). *Scenario Usage in System Development: A Report on Current Practice.* IEEE Software, 33-45.

[Zave and Jackson 1997] Zave, P. and Jackson, M. *Four dark corners of requirements engineering*, ACM Trans. Softw. Eng. Methodol., 6(1):1–30, 1997.