Software Product Lines using **FODA**: A formal approach

Carlos Camacho

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Máster en Programación y Técnologia Software

Julio 2012

Calificación: 6

Directores:

Luis Llana Díaz
César Andrés Sánchez

Autorización de difusión

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Software Product Lines using `FODA`: A formal approach", realizado durante el curso académico 2011-2012 bajo la dirección de Luis Llana Díaz y con la colaboración externa de dirección de César Andrés Sánchez en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fecha 1 de julio de 2012

Carlos Camacho

Resumen en castellano

El término línea de producción *en inglés product line* evoca a menudo la imagen de una fábrica de coches con un conjunto de brazos mecánicos especializados en colocar piezas, o tareas específicas como atornillar, soldar o ensamblar para conseguir, como producto final, un coche de manera rápida, invirtiendo la menor cantidad de recursos posibles, entre ellos, tiempo y dinero. Esta metodología se ha aplicado en contextos totalmente diferentes a la fabricación de coches, como por ejemplo en el entorno de la alimentación o el textil, entre otros [32]. En los últimos años, en el campo del desarrollo de software se está investigando cómo aplicar los principios de esta metodología para el desarrollo de aplicaciones de software de alta calidad optimizando los recursos asignados para su implementación.

La metodología de las líneas de producción, también conocida como producción en masa dentro del contexto informático, es sustancialmente distinta a la metodología anterior. Si tomamos literalmente el significado clásico, alguien podría pensar que el problema al que se enfrenta el mundo informático es obtener n copias distintas de un determinado programa. Éste no es el principal problema de la informática, ya que copiando n veces el programa en diferentes archivos (ya sean medios físicos, internet, entre otros) podemos obtener un conjunto de copias fieles, rápidas y a bajo coste sobre el programa original.

El problema al que nos enfrentamos es el siguiente. Supongamos que tenemos una empresa E que está desarrollando un software de gestión para dos empresas distintas X e Y. Ambos desarrollos tienen partes comunes y no comunes entre si. La empresa E quiere reducir costes y aprovechar parte del desarrollo de X en Y y viceversa. Una posible alternativa para E sería implementar en primer lugar el producto para la empresa X y luego para Y. La alternativa que propone la aplicación de una metodología de línea de producción para este caso se basaría en los siguientes puntos. Primero definir todas aquellas características comunes y no comunes de los productos a desarrollar para las empresas X e Y. De esa manera determinar los puntos de variación y las variantes que puedan existir entre los componentes que conformarán el conjunto de productos finales, entre ellos X e Y. Seguidamente, implementar y probar de manera exhaustiva una *plataforma* que implemente el conjunto de funcionalidades requeridas por ambos clientes. Finalmente, desarrollar módulos que personalicen cada aplicación con las necesidades de cada una de las empresas.

Un ejemplo muy gráfico de este tipo de desarrollo lo podemos encontrar en la construcción de aplicaciones para teléfonos móviles. Pensemos en dos modelos diferentes de móviles de la misma compañía. Seguramente la plataforma para ejecutar todas las aplicaciones (llamadas, mensajes, IM, entre otras) es la misma en los dos sistemas. Sin embargo la posibilidad de utilizar WIFI o Bluetooth podría corresponder a cada teléfono en particular.

La unidad básica de una línea de producción es llamada característica o *en inglés feature*. Una característica puede ser un módulo, la utilización de una tecnología, o en un concepto más general, cualquier componente funcional reutilizable. Las relaciones entre las distintas características que componen una línea de producción de software, establecen las posibles

configuraciones de los productos. Al conjunto de relaciones y características se le es llamado modelo de características, también conocido como *feature models*.

Anteriormente mencionamos que el desarrollo de una plataforma común, además de tener que representar todas aquellas características comunes y no comunes entre los productos, debe ser un entorno altamente testeado. Determinar la existencia de fallos en dicha plataforma es una tarea no trivial. La utilización de métodos formales nos ayudan a automatizar este proceso; de aquí surge la necesidad de definir modelos formales relacionados con la producción masiva de software. La principal contribución de este Trabajo de Fin de Máster se centra en definir un marco formal para un modelo de características que actualmente se está utilizando en el desarrollo de software [1]. Proponemos no solo la sintaxis de las características y sus relaciones, sino también proveemos tres diferentes semánticas, operacional, denotacional y axiomática, equivalentes entre sí, que permitirán deducir propiedades interesantes de los sistemas. A continuación presentamos la estructura de este trabajo.

1. En el Capítulo 1 se realizará una reseña sobre las *Líneas de producción de software* y *Modelos de características*. Se hará una introducción sobre el proceso de desarrollo sistematizado de software y las distintas maneras de modelar las partes funcionales de un producto de software. De tal manera que el lector pueda entrar en contexto con el tema central de este proyecto. Entre las metodologías estudiadas están

   - **F**eature **O**riented **D**omain **A**nalysis (`FODA`).
   - **P**roduct **L**ine **U**se case modeling for **S**ystems and **S**oftware engineering (`PLUSS`)
   - **R**euse-Driven **S**oftware **E**ngineering **B**usiness (`RSEB`)

2. A continuación, en el Capítulo 2 se presentará un estado del arte sobre distintas técnicas que permiten la especificación de *Modelos de características* utilizando métodos formales; entre ellos los *Sistemas de Etiquetado de Transiciones*, *Álgebras de Procesos*, *Sistemas de Transiciones Modales* y *Lógica Proposicional*.

3. En el Capítulo 3 se especificará una nueva forma de representar los *Modelos de características* utilizados en `FODA`. Este capítulo contiene nuestras principales aportaciones al área. Primero se desarrollará un álgebra que traduzca los diagramas presentados por `FODA` (`fodaA`). Luego se definirán tres semánticas para `fodaA` y se demostrará que las semánticas definidas anteriormente son equivalentes.

4. Y por último en el Capítulo 4, se mostrará una herramienta, para el análisis automatizado de *Modelos de características* llamada `AT`. Adicionalmente se mostrará un caso de estudio no trivial para el modelado de software. En este caso se modelará una herramienta que permite hacer streaming de una señal de vídeo y se ilustrará el marco descrito en el capítulo anterior.

   Para concluir el Trabajo de Fin de Máster, en el Capítulo 5 se mostrarán las conclusiones y posibles trabajos futuros en esta área de investigación.

[1] Feature Oriented Domain Analysis (`FODA`)

Palabras clave

Métodos Formales, FODA, PLUSS, RSEB, Líneas de Producción de Software, Álgebra de Procesos, Semántica operacional, Semántica denotacional, Semántica axiomática.

Abstract

Product line term often evokes a car factory image with a specialized set of mechanical arms to individually assembly pieces, or do some specific tasks such as screwing, welding, etc. to be able to get as a final product, a car as soon as possible, spending the minimum amount of resources, among them time and money. This methodology has been applied in entirely different contexts, from cars manufacture to textile fabrics among others [32]. In recent years, inside the software development field it has been investigated how to apply the principles of this methodology, for high quality software development, optimizing the implementation time as well as the amount of money required to fulfill clients requirements. Product line methodology in the context of computer science, significantly differs from the other examples mentioned. If we use this approach, people might think that the problem may be solved easily by getting n different copies from some particular software. The main problem is not the creation of multiple copies of the same software into different locations.

The problem that we face is this. Suppose that we have a company E which is developing a management software for two different companies X and Y. Both developments have common and uncommon parts between each other. Company E wants to reduce costs and take advantage of reusing resources to build software products for X and Y. A possible alternative would be the implementation of a product for company X, and then build the product for Y separately. The proposed alternative by the application of a production line methodology for this case, should be based on the following points. First, define all the common and uncommon features between X and Y. In this way, determine the variation points and variants that may exist between the components that will compose the set of final products among the products for companies X and Y. Next, implement and test exhaustively a platform that represents the set of functionalities required by both companies.

A *common platform* design, for a diverse set of components is not a trivial task. It involves preparation for mass customization, focusing first on what is common to all products, and then in what is different [28]. Components must be created for being reused by all, or most of the possible *configurations* (functional variant of a software product). These components can be developed from scratch or derived from other platforms. This flexible design allows to reuse these components with different configurations for a particular solution; in this way, mass customization of a set of well defined products is provided. This customization requires effort, but flexibility is the key for mass customization success and it is a must for it. A reorganization process for the mass customization initiative may require additional organizational units to guide towards standardization of procedures and workflows. It may also require to adopt new technologies within the company. Software is flexible and easily customizable, but wrong decisions in terms of defining the production engineering process can be expensive, therefore, it is required to have a perfect knowledge of the business logic within the organization and of the solution to be implemented. On one hand, a lot of software products are derived from a common base, and they represent essentially the same context, because they are variations or successive variations of a single product. For

instance, if we strip down a car we can see that the main parts are the same (engine, transmission, chassis, among others), between the luxury and standard version. On the other hand, `SPL` [2] engineering aims to support a wide range of products. These products may be for the same client, for different clients or even for entirely different markets. As a result, variability management is a very important concept in `SPL` approach. Variability design is about incorporating components that represents the range of possible configurations for a product in the `SPL`. This variability, is defined during the domain engineering process [18]. When we use the term variability, we are referencing to the ability that something has to change in time. This variability is defined in purpose.

A graphic example of this type of development can be found in software developments for mobile phones. Consider two different models of phones from the same company. Surely the platform to run all applications (calls, messages, IM) is the same in both models. However, the possibility of using WIFI or Bluetooth for each phone may vary.

The basic unit of a production line is called feature. A feature can be a particular module, the use of a specific technology, and, in general, any reusable functional component. The relationships between features in a `SPL` builds products configurations, the relationships and features set is called feature model.

Since in the `SPL` frameworks software is made of reusable components it is very important to have tools that allow the correct development. Not only within any component, but also in the whole framework. The use of formal methods aims to automate this task. This raises the need to define formal models related to massive software production. The main contribution of this Master Thesis work focuses on defining a formal framework for a widely used feature model in software development[3]. It has been done under[4] the **Facultad de Informática** of **Universidad Complutense de Madrid**. We have defined a formal syntax to express the features and their relationships. And also we have defined three different formal semantics: an operational semantics, a denotational semantics and an axiomatic semantic. We have proved that all three semantics are equivalent. Here is the structure of this work.

- First, in Chapter 1, we give an introduction to *Software Product Lines* and *Feature Modeling*. We describe the systematic software development process, and we show some different approaches to model the functional parts of software products. In this way, the reader can get into the context of the central line of this research project. We overview a state of art over `SPLs`, detailing several methodologies to specify software systems. Among them are:
  - **F**eature **O**riented **D**omain **A**nalysis (`FODA`).
  - **P**roduct **L**ine **U**se case modeling for **S**ystems and **S**oftware engineering (`PLUSS`)
  - **R**euse-Driven **S**oftware **E**ngineering **B**usiness (`RSEB`)

[2] Software Product Line (`SPL`)
[3] Feature Oriented Domain Analysis (`FODA`)
[4] The Spanish MEC project TESIS (TIN 2009-14312-C02-01)

- Next, in Chapter 2 will overview the state of art over different formalisms that allow the specification of *Feature Models* using *Formal Methods*, including *Labelled Transition Systems*, *Process Algebras*, *Modal Transition Systems* and *Propositional Logic*.
- In Chapter 3 we present our formalism to represent *Feature Models* using `FODA`.
- In Chapter 4, we describe a tool, called `AT`, for the automated analysis of feature models. Additionally, we show a non-trivial case study for modeling video streaming software products.
- To conclude this Master Thesis Project, Chapter 5 shows some conclusions and some possible lines of future work.

Key words

Formal methods, FODA, PLUSS, RSEB, Software Product Lines, Process algebras, Operational semantics, Denotational semantics, Axiomatic semantics.

Contents

List of Figures ii

1 Classical approaches to Software Product Lines 1
- 1.1 Feature Models . . . 2
- 1.2 Feature Oriented Domain Analysis (FODA) . . . 3
- 1.3 Reuse-Driven Software Engineering Business (RSEB) or Feature-RSEB . . . 5
- 1.4 Product Line Use case modeling for Systems and Software engineering (PLUSS) 6

2 Formal Methods in Software Product Lines 8
- 2.1 Formal frameworks non related to feature models . . . 9
  - 2.1.1 Process Algebras . . . 9
  - 2.1.2 Labelled Transition Systems (LTS) . . . 10
  - 2.1.3 Modal Transition Systems (MTS) and Extended Modal Transition Systems (EMTS) . . . 11
  - 2.1.4 Modal I/O Automata . . . 12
- 2.2 Formal frameworks related to feature models . . . 13

3 Formal study of FODA 17
- 3.1 FODA Algebra . . . 17
- 3.2 Operational Semantics . . . 21
- 3.3 Denotational Semantics . . . 30
- 3.4 Axiomatic Semantics . . . 35

4 Tool and Case Study 43
- 4.1 Tool . . . 43
- 4.2 Case Study . . . 47

5 Conclusions and Future Work 54
- 5.1 Conclusions . . . 54
- 5.2 Future Work . . . 56

Bibliography 58

List of Figures

1.1 FODA representation. 3
1.2 Examples of FODA Diagrams. 4
1.3 RSEB representation. 5
1.4 Example of RSEB *OR* relationship. 5
1.5 Example of a PLUSS diagram. 6

2.1 PL-CCS example. 9
2.2 Labelled Transition System example. 11
2.3 Extended Model Transition System example. 12
2.4 Modal I/O automata example. 13
2.5 Process for the automated analysis of feature models. 14
2.6 Feature model example. 15

3.1 Mapping from FODA Diagram to fodaA. 18
3.2 Examples of translation from FODA Diagrams into fodaA grammar. 22
3.3 Rules defining the operational semantics of fodaA. 23
3.4 Application of the operational semantic rules 1/3. 27
3.5 Application of the operational semantic rules 2/3. 28
3.6 Application of the operational semantic rules 3/3. 29
3.7 Application of the denotational semantic rules. 33
3.8 Equations to remove requires and mandatory operators. 35
3.9 Equations to remove exclusion, and forbid operators. 35
3.10 Axioms to remove the conjunction operator. 36
3.11 Axioms for basic operators and optional features. 36
3.12 Transformation to normal form 1/3. 37
3.13 Transformation to normal form 2/3. 38
3.14 Transformation to normal form 3/3. 39

4.1 Example of AT XML input. 44
4.2 Example of AT script execution. 46
4.3 Example of AT denotational script output. 46
4.4 Video Streaming Software - FODA representation. 48
4.5 Labelled Transitions System of the Video Streaming Software. 50
4.6 Denotational Semantics of Video Streaming Software. 51
4.7 Deduction Rules applied to the Video Streaming Software (1/2). 52
4.8 Deduction Rules applied to the Video Streaming Software (2/2). 53

Agradecimientos

Primero, a Dios, por darme fe, fuerza, coraje y constancia para alcanzar este triunfo.

Me gustaría agradecer a la Universidad Complutense de Madrid haberme dado la oportunidad de cursar el Máster en Investigación Informática.

También, recordar que este trabajo de investigación no es producto del trabajo individual, sino de la suma del apoyo y esfuerzo de muchas personas que de manera directa o indirecta me han apoyado y ayudado, mi familia, mis amigos, compañeros del trabajo y profesores de la Universidad.

En particular, quiero agradecer a los profesores César Andrés y Luis Llana, por haberme brindado la oportunidad de haber trabajado junto a ellos en el desarrollo del máster. Sin su apoyo y dedicación, muy probablemente no hubiera podido cumplir los plazos de tiempo para terminar el trabajo.

A Mary y Rubén, por brindarme su apoyo incondicional para cualquier cosa que necesite de ellos, en casa, en el trabajo y en la Universidad, ¡Gracias!.

A Caro, Pedro, Mario y Francho, por apoyarme siempre en el trabajo y a Alessia por haberme ayudado con las correcciones del documento.

A mis padres, hermanas y familia, por siempre estar allí para mí, sin importar las circunstancias.

A mis amigos, que siempre me apoyaron, les agradezco en el alma que estén allí para mí.

Carlos.

A mis padres, porque a ellos les debo todo lo
que soy y soy el reflejo de ellos. Siempre
serán mi referencia en la vida, son las
personas que más admiro. A mis hermanas
porque siempre me han apoyado. A mi familia. Los quiero!

Chapter 1

Classical approaches to Software Product Lines

Quality means doing it right when no one is looking.

Henry Ford

Software Product Line, in short SPL, is a paradigm to be considered as a systematic and disciplined approach for software developing. It concerns all aspects of the production cycle of software systems. It requires expertise, in particular, in data management, design and algorithm paradigms, programming languages, and human-computer interfaces. It also demands understanding and appreciation of systematic design processes, test and validation of functional or non-functional properties, and large systems integration. As bigger and robust software systems are developed, new ways to set the order of the processes development must be analyzed to increase productivity, reliability and quality of software development companies and their products. From the beginning of software development, when structured and object oriented software paradigms were created, the promising gold was *to reuse functionalities*. Thus, in the last few years, there are several studies which presents the advantages of reusing the whole system architecture design, instead of reusing only already defined methods packaged in software libraries [34, 26, 7, 25, 10, 24]. In SPL modeling; there are several ways to represent products variability, some of them with a strong formal representation and others relying their main strengths on graphical and intuitive representations,

have been developed. In this chapter we describe three of the most relevant product line approches. These approaches will represent product variability with respect to how features are related with each other. So, the structure of this chapter is as follows. Section 1.1is an introduction to feature modeling. In Section 1.2 we introduce FODA. In Section 1.3, we present a FODA extension which introduces the **OR** relationship. Finally, in Section 1.4 we discuss PLUSS.

1.1 Feature Models

A feature model, is a way to represent the information of all-possible configurations for a specific product that can be built. Currently, there are several ways to represent feature models [8, 15, 37, 3, 11, 14, 35]. Basically, these features are related using a tree-like diagram, relating parent and children features. A *variation point* is a place where a decision can be made to determine if none, one or more features can be selected to be part of the final product.

The analysis of feature models, consists in the observation of its properties [4, 5]. These properties includes checking whether a feature model is void (it represents no products), checking whether a feature model has internal dead features (features which are part of the feature model, and they are not included in any of its represented products), or to simply determine the number of products represented by a feature model. Let us remark that the final purpose of this modeling, is to help the optimization of configuration options to a given product and structuring the requirements for any variant in the product line. However, for most of variability management studies in SPLs, formal analysis has not been considered.

In particular, graphical frameworks are mainly used to represent this variability, for instance FODA, RSEB and PLUSS [8, 15, 11]. This situation has the following disadvantage: "the automated analysis cannot be performed over non formal data structures". There are frameworks that represent these structures with formal approaches. For example, in [10] the authors show how to represent feature models using Boolean expressions. In [12], fea-

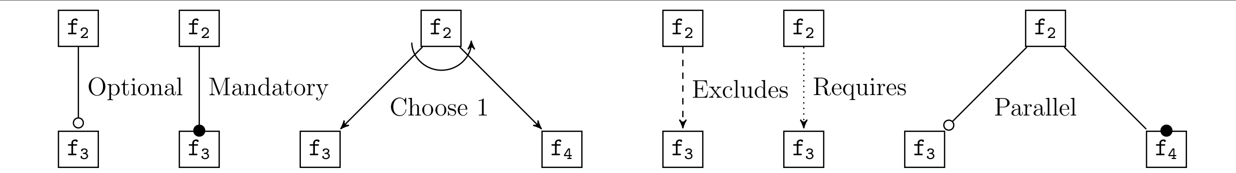


Figure 1.1: *FODA representation.*

ture models were represented using Labelled Transition Systems (LTSs), Modal Transition Systems (MTSs) and Extended Modal Transition Systems (EMTSs).

1.2 Feature Oriented Domain Analysis (FODA)

Feature models were first introduced as part of the FODA [1] method, by Kang back in 1990 [8]. FODA is a methodology originated from a study of different approaches of domain analysis. It gives a view of the requirements and architecture aspects over the assets to be built [27]. This methodology focuses on the *identification* of product features in a well defined *domain*. Moreover, the commonalities and variabilities of these features must be specified. Also, this model allows to graphically represent *features* and their *relationships*, in order to define *products* in a SPL. The graphical structure of a FODA model is represented by a FODA Diagram.

A FODA *Diagram* is essentially a graph, which is an intuitive and easy way to represent relevant information about features. SPL variability is represented within the nodes. Nodes represent SPL features, meanwhile arcs represents relationships among these features. A FODA diagram representation is described in Figure 1.1.

Example 1.2.1 Next, we explain a few examples about FODA, and how relationships are represented. Let us consider the FODA diagrams of Figure 1.2. Examples **a** and **b** show two simple SPLs with only two features A and B. In Example **a**, feature A will appear in all valid products of this SPL, while feature B is optional. This means, that the valid set of products for Example **a**, will be composed by two products, one with only feature A, and other with features A and B. In Example **b**, both features are mandatory, that is, any product generated

[1] Feature-Oriented Domain Analysis

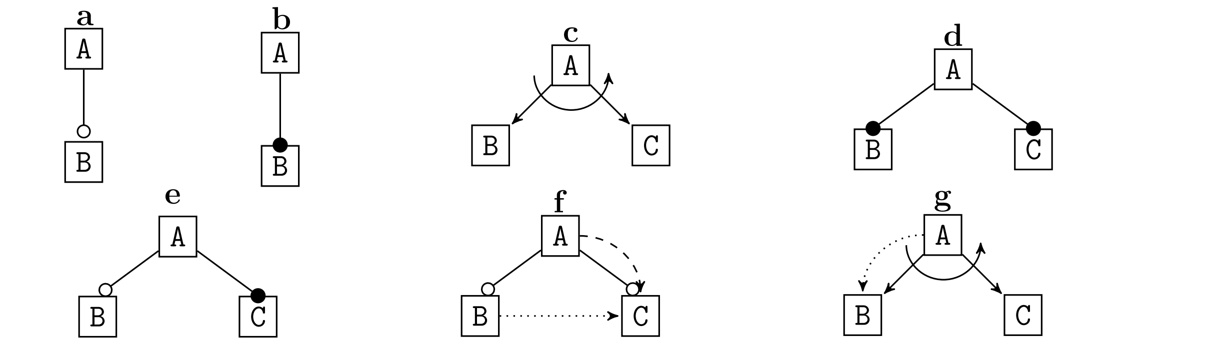


Figure 1.2: *Examples of FODA Diagrams.*

from this SPL will contain features A and B, this example has only one valid product.

Example, **c**, represents a SPL with a *Choose 1* operator. In this diagram three different features are presented, A, B and C . Any valid product of **c** will contain A, and only one of features B and C. In order to make the modeling task easier, the *parallel* operator is included. This operator is shown in **d** and **e**: from one feature, there are a set of branches. This operator represents the union of all features which depend on a common feature as children. For instance, Example **d** will only have one valid product, the one that contains the features A, B and C. Example **e** represents a SPL with two valid products. The first one with the features A, B and C, meanwhile the second product will only have features A and C.

Finally, complex properties can be expressed in FODA as those represented in examples **f** and **g**. The first one, combines the parallel operator of two optional features and two constraints. The first operator indicates that feature A must be included in all valid products of **f**, while features B and C are optional. There is an excludes relation: If A is included in a product, C must not appear. Thus, since A is mandatory in all products, C cannot appear in any product. Next let us focus on feature B. If B is included, then there exists a requires constraint on C, that is, if B is included in a product then C must appear in the same product. But, according to the previous restriction, C cannot appear in any valid product of **f**. Thus, there is only one valid product of Example **f**, the one containing feature A. Example **g** combines the *Choose 1* operator with a *Require* constraint. There are two valid product of

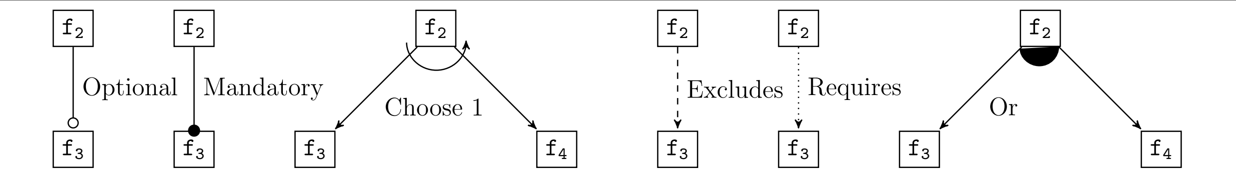


Figure 1.3: *RSEB representation.*

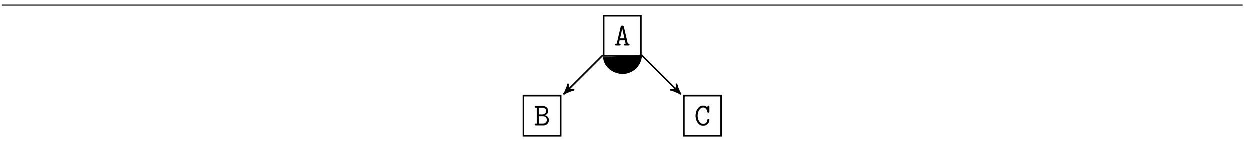


Figure 1.4: *Example of RSEB OR relationship.*

g, the one with features A and B and other with features A, C and B. The last one product is generated because when a feature is checked to be processed and there are any further processable features in the term, the marked feature must be integrated to the product. □

1.3 Reuse-Driven Software Engineering Business (RSEB) or Feature-RSEB

In [15] a FODA extension called Feature-RSEB was presented. The initial relationships that represent associations between features are the same as in FODA, plus an additional OR-relationship. This new relationship requires that at least one of the features should be selected. Thus, it does not have the restriction of the *Choose 1* operator of FODA, where only one option was selected. The notation of RSEB is presented in Figure 1.3.

Example 1.3.1 According to the SPL in Figure 1.4, a product must include the feature A, and can include the features B or C or any combination of both, with the restriction that one of them must be included. In this specific case, there are only three valid products: with features A and B; with features A and C; and with features A, B and C. □

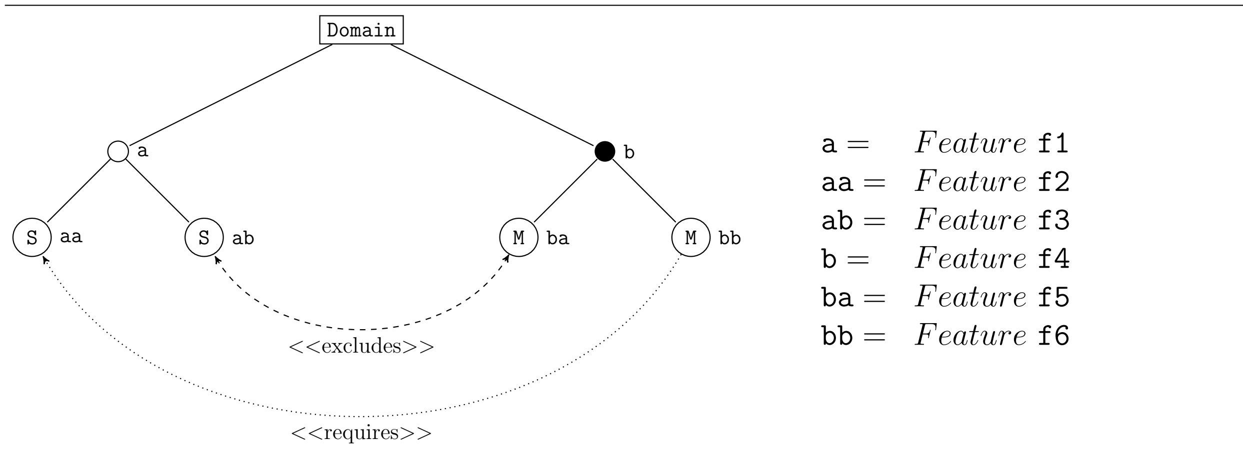


Figure 1.5: *Example of a PLUSS diagram.*

1.4 Product Line Use case modeling for Systems and Software engineering (PLUSS)

In [11], the Product Line Use case modeling for Systems and Software engineering approach, in short PLUSS, was presented as a FODA extension [15]. This methodology uses case modeling as an appendix to the domain engineering analysis of the system, this means that a PLUSS model may contain a use case diagram to extend the model information. Authors argue that feature models are better suited for domain modeling than UML use cases. In this approach, they also use the same relations previously defined in RSEB. The difference is that the modeling is complemented with a high level view. PLUSS has the relation *At least one of many*. This relation corresponds to the *Or* in RSEB. Moreover they renamed the *Or* relation to *multiple adaptor*, and, the *Choose 1* operator of FODA is denoted as *single adaptor*.

Example 1.4.1 The graphical representation varies from the previous studies of FODA and RSEB. An example of this approach is presented in Figure 1.5. Regarding the notation a filled black circle represents a *Mandatory* feature, while a non filled circle represents an *Optional* feature. We have that the *Choose 1* relationship is represented using the letter **S** in a circle and the *Or* relationship is denoted by using the letter **M** in a circle. Finally, the *Requires* and *Excludes* relationships are represented as in UML, grouped by the << and

>> symbols.

Now, let us explain Figure 1.5 in detail. The product line is composed of a set of features F= {`f1`, `f2`, `f3`, `f4`, `f5`, `f6`}. In this example does not appear any feature referenced as `f1`, `f2`, `f3`, `f4`, `f5` or `f6`. Instead are showed as `a`, `aa`, `ab`, `b`, `ba`, `bb`. In `PLUSS`, features are named using a hierarchy notation. Features in the first level of the tree, are showed with one symbol starting from `a`, next `b` and so forth. In the next level, another symbol is concatenated in alphabetic order[2]. For example, the first child of feature `a` is named `aa`, the second child is named `ab` and so forth. In this particular case, Example 1.5 will generate a set of valid products represented by `prod` (K)={[`b`, `ba`], [`a`, `aa`, `b`, `ba`], [`a`, `aa`, `b`, `bb`], [`a`, `aa`, `b`, `ba`, `bb`]}. Note that {`b`, `ba`} and {`ba`, `b`} represent the same product. To give a simple view of the generated products, they are showed as [`b`,`ba`] instead of {`b`, `ba`} and {`ba`, `b`}. □

In the next Chapter, we describe some approaches to formally represent Product Families or `SPLs` by using mathematical frameworks. These formal frameworks may, or may not be related to the representation of feature models. In particular, the next Chapter introduces the novel approach presented as part of this research work, starting from studies that are not directly related with software development.

[2] A character precedence order must be defined in order to fulfill `PLUSS` notation.

Chapter 2

Formal Methods in Software Product Lines

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupéry

Formal methods use formal *syntax* to write precise system specifications. This syntax is usually textual but can be graphical. A *semantics* is also provided, this occur when a precise meaning is given for each description in the language. A specification of a system might cover several aspects, including its functional behavior, its structure or architecture, or even non-functional behavior such as timing or performance criteria. Formal specification can also be used to support SPLs descriptions. A variety of different formal specification techniques exists, some are general purpose while others stress relevant aspects of a particular application domain.

We classify the formal frameworks that represent SPLs in two categories. On the one hand, those models that are not directly related to Feature Modeling [38, 12, 30, 29, 13, 2, 1]. On the other hand those models that are directly related to Feature Modeling; that is, those proposals that give semantics to previous non-formal frameworks, such as Feature Models [3, 5].

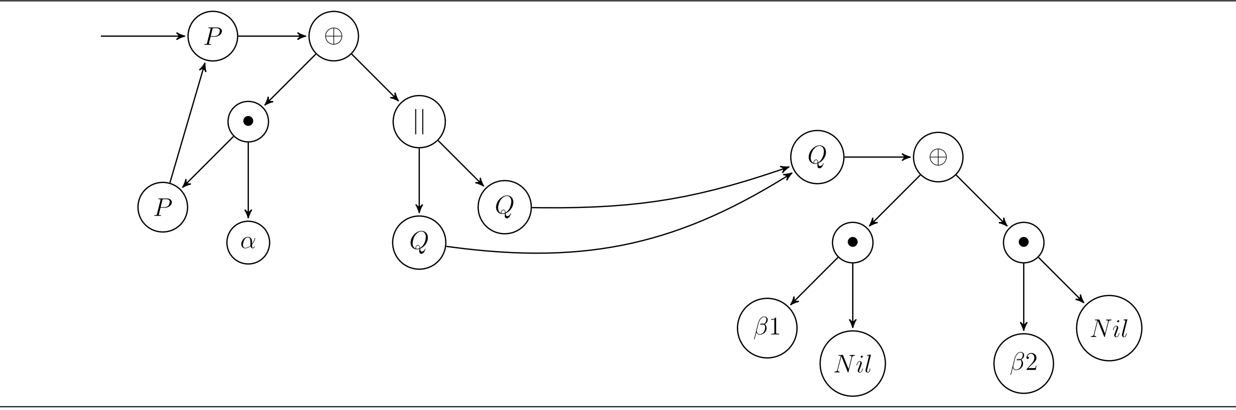


Figure 2.1: *PL-CCS example.*

2.1 Formal frameworks non related to feature models

We cover in this section algebraic and behavioral formal frameworks to model systems variability.

2.1.1 Process Algebras

When a system behavior is modeled, there must be a way to formally express a set of states and transition rules from which a transition from one state to other is possible. The execution of a set of states and transition rules can be defined as a *process*. These processes can be executed in a serial way, which implies the execution of each process one by one. Or they can be executed in parallel, this means concurrently, so all processes are executed at the same time, and the behavior of any process can have influence in others behavior.

Behavior modeling is based on the study of how data is manipulated. This modeling includes the definition of control mechanisms which handles and manipulates systems data. These processes are dynamic and active; the data itself is a static and passive object. The system behavior is defined as the execution of several concurrent processes, where these processes exchange data to influence each others behavior.

In [17], authors present an algebra based on the classical CCS [33] process algebra called PL-CCS. It extends a classic Algebra model like CCS. In this case, variability is intended

to be modeled as the same way that systems behavior is. In this approach, the parallel operator from CCS is used , also a binary variant operator is used, optional, non-deterministic selection, restriction and nil are described as in our approach. It also described a LTS for describing the communication actions and the transition relations for all defined semantics. In this approach there is no relation that defines a constraint like a requires between two features. It is important to note that in this model there is no relation with a previously defined SPL feature model like FODA, RSEB or PLUSS. In figure 2.1 there is an example of the approach of [33] (For further information over operators and how transitions are processed over states, please refers to this article). In this example, it is shown how variability is described using basic process algebras. Relationships are used to describe configuration desicions among several features. Also in [22, 23] they define SPLs as idempotent semirings or diods, where the algebraic terms represents features. In [23] they extends the notion of the multi-view reconciliation problem, in which case they tries to merge different views of the domain context from which the project is. This means to define the project from different views.

2.1.2 Labelled Transition Systems (LTS)

The following frameworks [38, 12, 30, 29, 13, 2, 1] adapt and extend LTSs to represent SPLs. An LTS can be seen as the representation of the behavior of a system, that performs events which represents a change in the system. When an LTS represent a SPL, these events usually refer to features. Let us note that the LTS formalism has been widely studied in the literature, therefore there are several semantic notions for them. In [38] a discussion over a novel point of view over Product Families showing traditional modeling formalism such as Labeled Transition Systems is presented.

In the Figure 2.2 (extracted from [13]) a basic expending machine, that sales coffee, tea and coke is represented. All states and event from which a transition from one state to other can happen are depicted. Note that it can be easily seen that transitions among

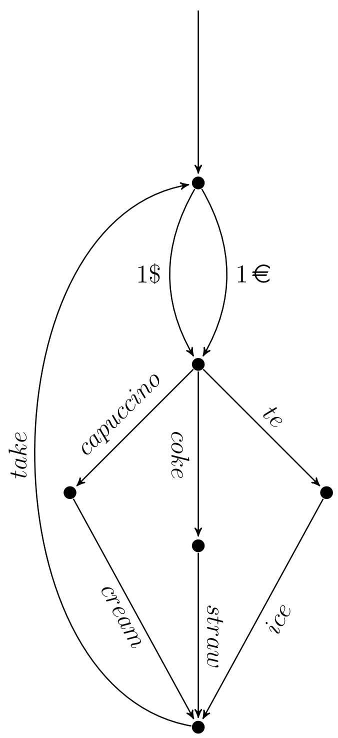


Figure 2.2: *Labelled Transition System example.*

states represents features, and states represents variation points inside the system domain.

2.1.3 Modal Transition Systems (MTS) and Extended Modal Transition Systems (EMTS)

A MTS is an extension of LTS where two types of transitions are defined: *at least k of n* and *at most k of n*. They represent the fact that some features are always selected while others may be included or discarded.

In [38] the concept of representing SPLs using Modal Transition Systems was proposed. This model supports optionality and variability over the different components of the SPL. One disadvantage of the MTSs models is that they do not have the possibility to define constraints between features. To overcome this problem in [12] an extension of the MTS of [38] was described.

Example 2.1.1 In figure 2.3, two MTS diagrams are shown, in which the *at least k of n*

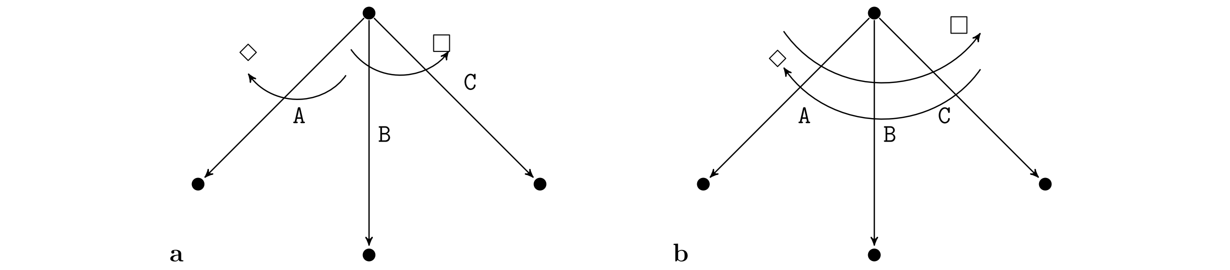


Figure 2.3: *Extended Model Transition System example.*

constraint, denoted with the □ symbol, and the *at most k of n* constraint, represented with the ◇ symbol were used. Example **a** describes a model in which features `B` and `C` must be included in all products, and `a` is optional in all products. In Example **b**, as `A`, `B` and `C` are covered with the □ and ◇ relations, only one feature must be used in all products. Using these two relationships together an or-exclusive relation is developed . □

2.1.4 Modal I/O Automata

There are some ways to model Software Product Lines; some of these representations tries to describe architectural or structural components of systems, also the behavior of individual modules being part of the whole system. One of these approaches is Modal I/O Automatas.

An I/O automata is about modeling interfaces to provides communication between components, this communication is represented using an automata-based language, were components are modeled as automatas, this means that inputs and outputs are guaranteed by the model specification, in this representation, the way that components may call each others is defined .

In [30] a formalization where interface automata corresponds to a subset of modal transition systems is described. The modal I/O automata is defined, as an extension of an interface automata.

In figure 2.4 taken from [30] an interface automata for a component is defined. This model represents a component that may send a package, and must receive an acknowledgment. In this example, a fail input, which is activated when the package is not received is

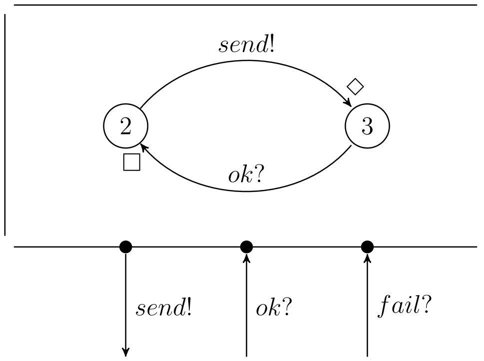


Figure 2.4: *Modal I/O automata example.*

also defined. A transition between states is denoted with an ◇ is intended to be a optional communication channel, and when is denoted with a □, it is a must transition.

2.2 Formal frameworks related to feature models

Regards the second category, it is worth to mention the following works [6, 3, 9, 20, 5]. In these frameworks, provide a semantics to existing feature models [8, 27, 15, 11] such as FODA and its extension RSEB. In [5] they perform an automated analysis of feature models. Figure 2.5 [1], describes the process of their conceptual framework. It is defined as a two steps process, where the input parameter *(a feature model)* is translated into a formal specification. Then the representation of this input parameter, is analyzed to generate a set of products, and from that point, perform the analysis of the model. This methodology is very similar to the one developed in this research project.

In [3], the authors use the model defined in [31] to model FODA. They showed relations among feature diagrams, grammars and propositional formulas, in which case, enable the use of SAT solvers to make automated support over a feature diagram.

In [6], and its extension [20], the authors define a semantic for FODA, based on what they call *tree feature diagram*.

A different way to model variability using formal methods is the computing of feature

[1]Picture taken from: Automated Analysis of Feature Models 20 Years Later: A Literature Review

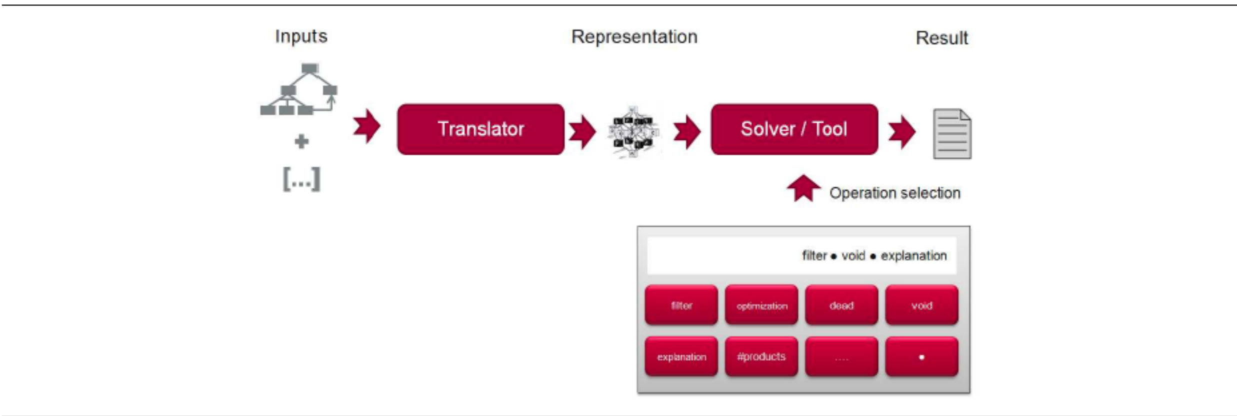
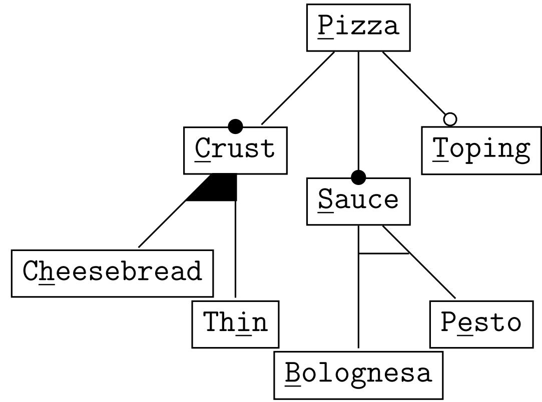


Figure 2.5: *Process for the automated analysis of feature models.*

models using propositional logic formulas. Where feature models are characterized and matched with an equivalent logical structure which corresponds to their syntactic elements. In mathematical terms, an *algebra* consists of a set of symbols denoting *values* of some type, and *operations* on the values. In these approaches, logical operators are used over the defined operands which represent features. These works use basic structures from feature models, like mandatory/optional, feature groups, and implies and excludes relationships. They are willing to reuse the analysis and configuration of already existing logic-based tools, such SAT solvers and Binary-Decision Diagrams (BDD) libraries. For example, in Figure 2.6 from [10], it is shown how a feature model is translated into their logical equivalent formula. In [10] the problem of opposite translation is considered; which is the extraction of feature models from propositional formulas. Some articles also follows an algebraic approach like [22, 23, 17]. Finally, in [31, 10] `SPLs` are represented with propositional logic expressions.

As we will see in this chapter, our approach falls in the category of frameworks which are related to `SPLs`, although it is closely related to some works the are not related to `SPL`, since our approach models `FODA` by using process algebras as a basic component.

Our objective in this research project, is to provide `FODA` with a *formal semantics* that removes any ambiguity or lack of precision like in [20]. By formal semantics we mean a formal syntax and the semantics for this formal language. In our opinion this point is



Propositional formula

$$
\begin{array}{ll}
\multicolumn{2}{l}{F=} \\
\textit{child} - \textit{parent}: & (C \longrightarrow P) \wedge (S \longrightarrow P) \wedge (T \longrightarrow P) \wedge \\
 & (H \longrightarrow C) \wedge (I \longrightarrow C) \wedge \\
 & (B \longrightarrow S) \wedge (E \longrightarrow S) \wedge \\
\textit{mandatory}: & (P \longrightarrow C) \wedge (P \longrightarrow S) \wedge \\
\textit{or} - \textit{group}: & (C \longrightarrow H \vee I) \wedge \\
\textit{xor} - \textit{group}: & (S \longrightarrow B \veebar E)
\end{array}
$$

Figure 2.6: *Feature model example.*

not present in [3, 9, 5] because they only presents theoretical frameworks but not really a syntax and semantics to enable the automated analysis of feature diagrams. Another objective is to provide a simple, structured and flexible framework that could incorporate new characteristics in the future.

The approach used to develop our semantics is similar to [17], but our intention has not been to extend a previous process algebra but to define a novel algebra to represent FODA. In this sense our approach is simpler and the semantic is also simpler. In particular, we do not need the fix point theory because there is no recursion in FODA. The algebraic approach in [22, 23] is similar to our denotational semantics. In our framework we have a syntax that formalizes the FODA diagrams. Next we have defined an operational semantics. This semantics is intuitive and we can extract easily the products of a SPL. After the operational semantics is defined, we have presented the denotational semantics. This semantics is more

appropriate to obtain the products of a SPL. We have proved that both semantics are equivalent. Finally we have defined another semantics that is very typical in the Process Algebra community, the axiomatic semantics. This semantics is very appropriate to deal with equivalences between SPLs. As far as we know, this semantics does not appear in any of the previous frameworks.

In the next Chapter, we describe our approach and by a set of examples, to be able to check how FODA methodology is represented over Process Algebras.

Chapter 3

Formal study of FODA

Pure mathematics is, in its way, the poetry of logical ideas.

Albert Einstein

This chapter will specify a novel approach to formalize FODA diagrams. This approach was defined by using Process Algebras as basis to build a formal representation of FODA graphics, which in a first place, lacks of a formal representation. Let us note, that FODA methodology was originated from a study of different approaches of domain analysis. It gives a view of the requirements and architecture aspects of the assets to be constructed. In particular, FODA can represent different relations between features belonging to a SPL. These relations are: *optional*, *mandatory*, *choose 1*, and *parallel*. Also it is presented the *requires*, and *excludes* constraints.

This Chapter describes the steps of our FODA formalization methodology. In Section 3.1 the representation and translation from FODA diagrams to our fodaA algebra is presented. Finally, in Section 3.2, Section 3.3 and Section 3.4 the operational, denotational and axiomatic semantics are specified.

3.1 FODA Algebra

The syntax of our algebra and how the FODA Diagrams are translated into the syntax will be presented in this section. First, we will define the syntax of the algebra and we will explain

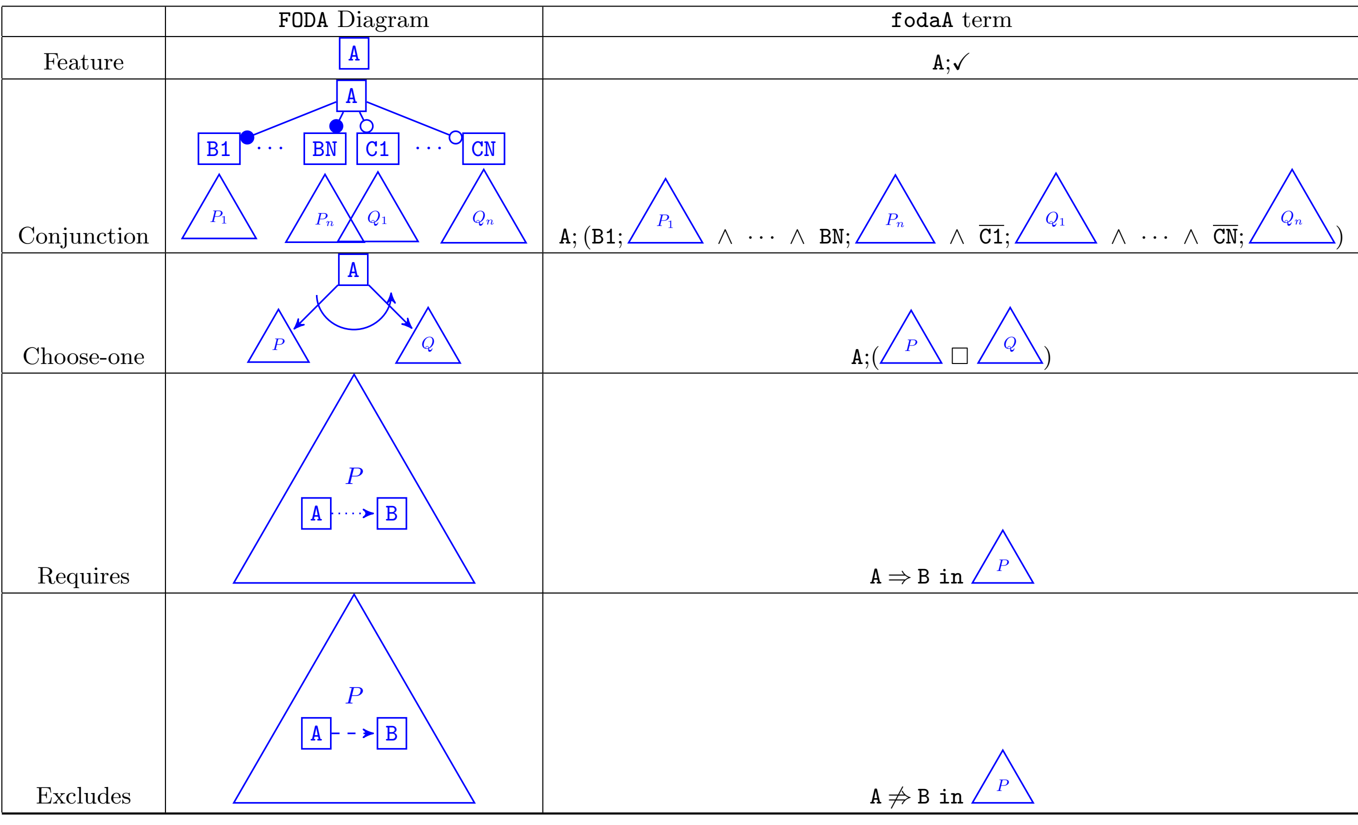


Figure 3.1: *Mapping from* `FODA` *Diagram to* `fodaA`.

the intuitive meaning of the operators. Next, we will introduce the translation of a `FODA` Diagram into our syntax.

The *syntax* concerns the principles and rules for constructing *terms*. We will define the language `fodaA` by means of an Extended BNF expression. Let us remember that the language of `fodaA` allows us to describe `SPLs`. In order to define the syntax, we will need to fix the set of *features*. From now on $\mathcal{F}$ denotes the set of features and `A`, `B`, `C`... denote isolated features.

In the syntax of the language there are two sets of operators. On the one hand *main operators*, such as $\cdot \,\square\, \cdot$, $\cdot \wedge \cdot$, $\texttt{A}; \cdot$, $\overline{\texttt{A}}; \cdot$, $\texttt{A} \Rightarrow \texttt{B}\ \texttt{in}\ \cdot$, $\texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ \cdot$, that directly correspond to relationships in `FODA` Diagrams. On the other hand, we will find in the syntax *auxiliary operators*, such as `nil`, ✓, $\cdot \backslash \texttt{A}$, $\cdot \Rightarrow \texttt{A}$, which we will need to define the semantics of the

language.

Definition 3.1.1 A *software product line* is a term generated by the following Extended BNF-like expression:

$$
\begin{array}{rl}
P ::= & \checkmark \mid \mathtt{nil} \mid \mathtt{A}; P \mid \overline{\mathtt{A}}; P \mid \\
 & P \,\square\, Q \mid P \wedge Q \mid \mathtt{A} \not\Rightarrow \mathtt{B} \;\mathtt{in}\; P \mid \\
 & \mathtt{A} \Rightarrow \mathtt{B} \;\mathtt{in}\; P \mid P \backslash \mathtt{A} \mid P \Rightarrow \mathtt{A}
\end{array}
$$

where $\mathtt{A}, \mathtt{B} \in \mathcal{F}$. We will denote the set of terms of this algebra by `fodaA`. □

In order to avoid writing too many parentheses in the terms we are going to assume left-associativity in binary operators and the following precedence in the operators (from higher to lower priority): $\mathtt{A}; P$, $\overline{\mathtt{A}}; P$, $P \square Q$, $P \wedge Q$, $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$, $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$, $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$, $P \backslash \mathtt{A}$, and $P \Rightarrow \mathtt{A}$. We will show (Proposition 3.2.11) that the binary operators are commutative and associative. So we can assume that the *choose-one* operator $(\cdot \,\square\, \cdot)$ and the *conjunction* operator $(\cdot \wedge \cdot)$ are n-ary operators instead of just binary operators.

The previous Extended BNF expresses that a term of `fodaA` is a sequence of operators and features. An `fodaA` term represents sets of *products*. Following we will introduce the formal definition of products of an `SPL` expressed in `fodaA` in Section 3.2. Basically, a product is a set of features that can be derived from the `fodaA` term. Next we will explain the meaning of each operator by using some examples.

There are two terminal symbols in the language: `nil` and ✓. We will need them to define the semantics of the language. Let us note that the products of a term in `fodaA` will be computed following some rules. The computation will finish when no further steps are allowed. This fact is represented by the `nil` symbol. During the computation of an `fodaA` term, we have to annotate when a *valid product* of the term has been computed. This fact is represented by the ✓ symbol.

The operators $\mathtt{A}; P$ and $\overline{\mathtt{A}}; P$ add the feature `A` to any product that can be obtained from P. The operator $\mathtt{A}; P$ indicates that `A` is mandatory while $\overline{\mathtt{A}}; P$ indicates that `A` is optional. There are two binary operators: $P \,\square\, Q$ and $P \wedge Q$. The first one represents the *choose-one* operator while the second one represents the *conjunction* operator.

Example 3.1.2 The term $\mathtt{A};\overline{\mathtt{B}};\checkmark$ represents an SPL with two valid products. We have a product with only the feature A and another product with the features A and B. This is because feature B is optional in this case.

The term $\mathtt{A};\checkmark \,\square\, (\mathtt{B};\checkmark \,\square\, \mathtt{C};\checkmark)$ has three valid products with one feature in each one. The first one consists in feature A, the second one consists in feature B and the third one consists in feature C.

We will show that $\square$ is commutative and associative so we could rewrite the previous term without parentheses: $\mathtt{A};\checkmark \square \mathtt{B};\checkmark \square \mathtt{C};\checkmark$. Therefore, we can consider that this operator is not only for choosing 1 out of 2 options but for choosing 1 feature from n options.

The term $\mathtt{A};(\mathtt{B};\checkmark \wedge \mathtt{C};\checkmark)$ represents a mandatory relationship; and we will see that this term has only one product with three features: A, B, and C. As well as the *choose-one* operator, we will show that the $\wedge$ operator is commutative and associative. So we can also assume that this is an n-ary operator. $\square$

The constraints are easily represented in fodaA. The operator $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$ represents the *require* constraint of FODA Diagram. The operator $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$ represents the *exclusion* constraint in FODA.

Example 3.1.3 The term $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \mathtt{A};\checkmark$ has one valid product with the features A and B.

Let us consider $P = \mathtt{A};(\mathtt{B};\checkmark \,\square\, \mathtt{C};\checkmark)$. This term has two valid products: The first one has the features A and B while the second one has the features A and C.

If we add to the previous term the following constraint $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$, then this new term has only one product with the features A and C. $\square$

The operator $P \Rightarrow \mathtt{A}$ is necessary to define the semantics of the $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$ operator. When we compute the products of the term $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$, we take to take into account if product A has been produced. In the case it is produced, we have to annotate that we need to produce B in the future. The operator $P \Rightarrow \mathtt{B}$ is used for this purpose. The same happens with the operator $P\backslash\mathtt{B}$. When we compute the products of $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$, if the feature A

is computed at some point, we annotate that $\mathtt{B}$ must not be included. The operator $P\backslash\mathtt{B}$ indicates that product $\mathtt{B}$ is forbidden.

Any SPL expressed with a FODA Diagram notation can be easily translated to fodaA. But it is also important to note that there are terms in fodaA that cannot be represented by a FODA Diagram, as in the case of the auxiliary operators like nil, $P \Rightarrow \mathtt{A}$, and $P\backslash\mathtt{A}$. The matching table used to translate FODA Diagrams to fodaA syntax is presented in Figure 3.1. Any FODA Diagram can be translated into fodaA by using the rules in this figure. Let us note that the mapping rules consider the binary operator $\square$. This is to simplify the figure, we can represent also n-ary *choose-one* diagrams from FODA because, as we have already said, the $\square$ operator in fodaA is commutative and associative (Proposition 3.2.11).

Example 3.1.4 The translation of the FODA Diagram in Figure 1.2 by using the mapping rules of Figure 3.1 is presented in Figure 3.2. $\square$

3.2 Operational Semantics

So far, we only have a syntax to express the SPLs in fodaA. Before doing anything else we need to provide the semantics for the fodaA language. The idea is to define a labelled transition system for any term $P \in \mathtt{fodaA}$. The transitions are *annotated* with the set $\mathcal{F} \cup \{\checkmark\}$, being $\mathcal{F}$ the set of features and $\checkmark \notin \mathcal{F}$. In particular, if $\mathtt{A} \in \mathcal{F}$, the transition $P \xrightarrow{\mathtt{A}} Q$ means that there is a product of P that contains the feature $\mathtt{A}$. The transition $P \xrightarrow{\checkmark} Q$ means that we have just obtained a product of P. The formal operational semantic rules of the algebra are presented in Figure 3.3. Some of the rules in the operational semantics have negative premises, for instance $P \stackrel{\checkmark}{\not\longrightarrow}$ means that there is no P_1 such that $P \xrightarrow{\checkmark} P_1$. These negative premises are not a problem in the rules because they always refer to simpler syntactical terms, so it is trivial to find a stratification [16] to prove the transition system is consistent.

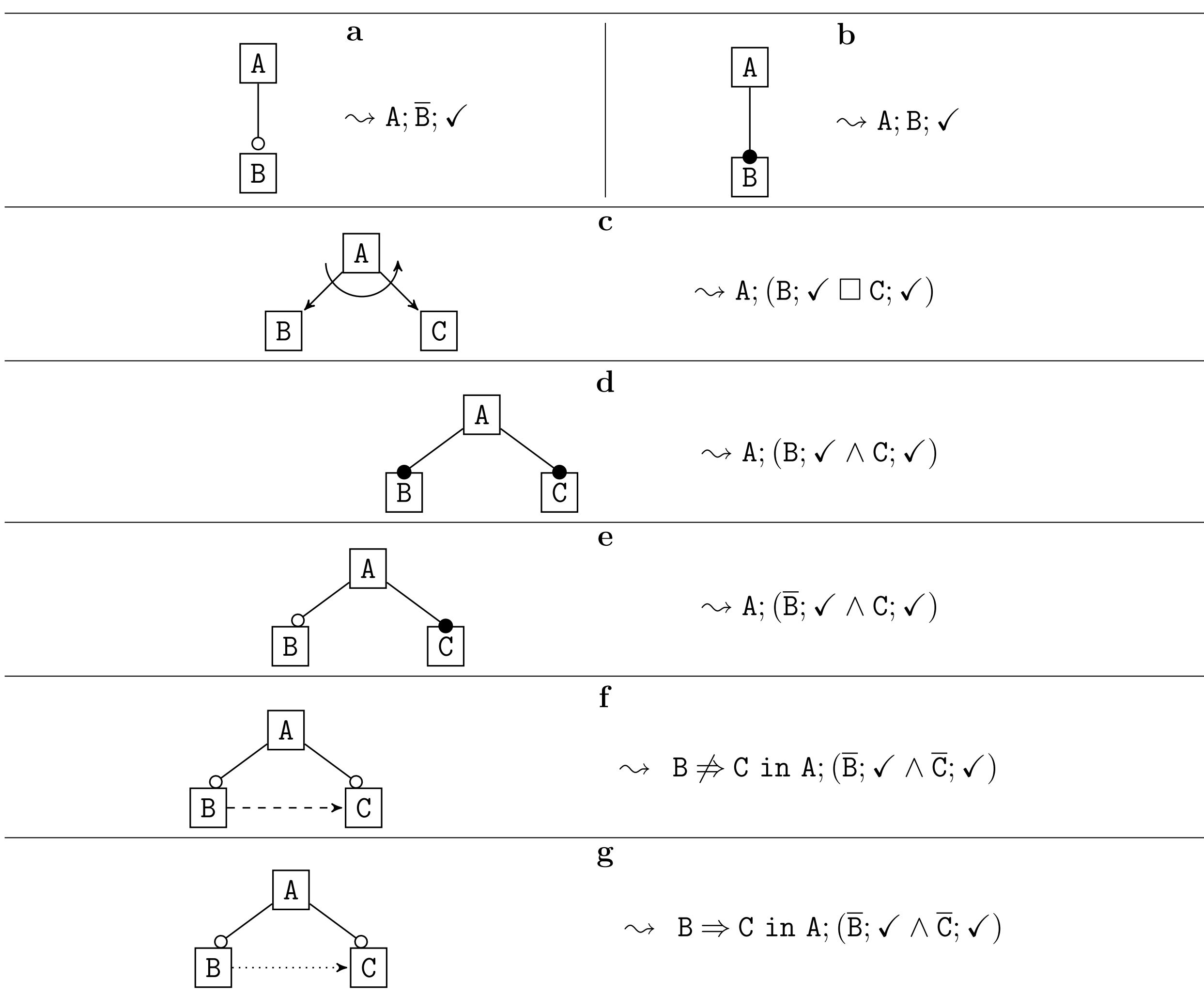


Figure 3.2: *Examples of translation from* ***FODA*** *Diagrams into* `fodaA` *grammar.*

Definition 3.2.1 Let $P, Q \in$ `fodaA` and $a \in \mathcal{F} \cup \{\checkmark\}$ There is a transition from P to Q labelled with the symbol a, denoted by $P \xrightarrow{a} Q$, if it can be deduced from the rules in Figure 3.3. □

Before giving any properties of this semantics let us justify the rules in Figure 3.3. We will define the products of an SPL from the set of traces obtained from the defined transitions.

First we have the rule [**tick**]. The intuitive meaning of this rule is that we have reached a point where a product of the SPL has been computed. Let us note that `nil` has no transition, this means that `nil` does not have any valid products.

[**tick**]	$\checkmark \xrightarrow{\checkmark} \texttt{nil}$	[**feat**]	$\texttt{A}; P \xrightarrow{\texttt{A}} P$
[**ofeat1**]	$\overline{\texttt{A}}; P \xrightarrow{\texttt{A}} P$	[**ofeat2**]	$\overline{\texttt{A}}; P \xrightarrow{\checkmark} \texttt{nil}$
[**cho1**]	$\dfrac{P \xrightarrow{a} P_1}{P \,\Box\, Q \xrightarrow{a} P_1}$	[**cho2**]	$\dfrac{P \xrightarrow{a} P_1}{Q \,\Box\, P \xrightarrow{a} P_1}$
[**con1**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1, P \not\xrightarrow{\checkmark} \texttt{nil}}{P \wedge Q \xrightarrow{\texttt{A}} P_1 \wedge Q}$	[**con2**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1, P \not\xrightarrow{\checkmark} \texttt{nil}}{Q \wedge P \xrightarrow{\texttt{A}} Q \wedge P_1}$
[**con3**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1, Q \xrightarrow{\checkmark} \texttt{nil}}{P \wedge Q \xrightarrow{\texttt{A}} P_1 \wedge Q}$	[**con4**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1, Q \xrightarrow{\checkmark} \texttt{nil}}{Q \wedge P \xrightarrow{\texttt{A}} Q \wedge P_1}$
[**con5**]	$\dfrac{P \xrightarrow{\checkmark} \texttt{nil}, Q \xrightarrow{\checkmark} \texttt{nil}}{P \wedge Q \xrightarrow{\checkmark} \texttt{nil}}$	[**req1**]	$\dfrac{P \xrightarrow{\texttt{C}} P_1,\ \texttt{C} \neq \texttt{A}}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\texttt{C}} \texttt{A} \Rightarrow \texttt{B in } P_1}$
[**req2**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\texttt{A}} P_1 \Rightarrow \texttt{B}}$	[**req3**]	$\dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\checkmark} \texttt{nil}}$
[**excl1**]	$\dfrac{P \xrightarrow{\texttt{C}} P_1,\ \texttt{C} \neq \texttt{A} \wedge \texttt{C} \neq \texttt{B}}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{C}} \texttt{A} \not\Rightarrow \texttt{B in } P_1}$	[**excl2**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{A}} P_1 \backslash \texttt{B}}$
[**excl3**]	$\dfrac{P \xrightarrow{\texttt{B}} P_1}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{B}} P_1 \backslash \texttt{A}}$	[**excl4**]	$\dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\checkmark} \texttt{nil}}$
[**forb1**]	$\dfrac{P \xrightarrow{\texttt{B}} P_1,\ \texttt{B} \neq \texttt{A}}{P \backslash \texttt{A} \xrightarrow{\texttt{B}} P_1 \backslash \texttt{A}}$	[**forb2**]	$\dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{P \backslash \texttt{A} \xrightarrow{\checkmark} \texttt{nil}}$
[**mand1**]	$\dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{P \Rightarrow \texttt{A} \xrightarrow{A} \checkmark}$	[**mand2**]	$\dfrac{P \xrightarrow{\texttt{A}} P_1}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{A}} P_1}$
[**mand3**]	$\dfrac{P \xrightarrow{\texttt{B}} P_1,\ \texttt{A} \neq \texttt{B}}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{B}} P_1 \Rightarrow \texttt{A}}$		

$$\texttt{A}, \texttt{B}, \texttt{C} \in \mathcal{F},\ \ a \in \mathcal{F} \cup \{\checkmark\}$$

Figure 3.3: *Rules defining the operational semantics of* `fodaA`*.*

Rules [**feat**], [**ofeat1**], and [**ofeat2**] deal directly with the computation of features. Rule [**ofeat2**] means that we have a valid product without considering an optional feature, in other words, this rule is the one that establishes the difference between an optional and a mandatory feature. A feature `A` is optional in P if $P \xrightarrow{A} P_1$ and $P \xrightarrow{\checkmark} \texttt{nil}$ [1]. In this sense, the transition $P \xrightarrow{\checkmark} \texttt{nil}$ indicates not only that P has already computed a valid product, but also it indicates that if P can compute any other features, these additional features are

[1] Lemma 3.2.2 states that if $P \xrightarrow{\checkmark} P_1$, then $P_1 = \texttt{nil}$.

optional.

Rules [**cho1**] and [**cho2**] deal with the *choose-one* operator. Both rules are symmetrical to each other. These rules indicate that the computation of $P \Box Q$ must choose between the features in P or the features in Q.

Rules [**con1**] to [**con5**] deal with the *conjunction* operator. The main rules are [**con1**] and [**con2**]. These rules are symmetrical to each other. They indicate that any product of $P \wedge Q$ must have the features of P and Q. This is true as long as we do not have optional features. This is the purpose of the condition $P \stackrel{\checkmark}{\not\rightarrow} \mathtt{nil}$ in the premises of rules [**con1**] and [**con2**]. If a member of the *conjunction delivers* an optional feature, it must wait until the other member has *delivered* all its non optional features (rules [**con3**] and [**con4**]). Finally rule [**con5**] indicates that both members have to agree in order to *deliver* a product.

Rules [**req1**], [**req2**], and [**req3**] deal with the *require* constraint. Rule [**req1**] indicate that $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$ behaves like P as long as feature $\mathtt{A}$ has not been computed. Rule [**req2**] indicates that $\mathtt{B}$ is mandatory once $\mathtt{A}$ has been computed. Finally [**req3**] is necessary for Lemma 3.2.2.

Rules [**excl1**] to [**excl4**] deal with the exclusion constraint. Rule [**excl1**] indicates that $\mathtt{A} \not\Leftrightarrow \mathtt{B}\ \mathtt{in}\ P$ behaves like P as long P does not compute feature $\mathtt{A}$ or $\mathtt{B}$. Rule [**excl2**] indicates that once P produces $\mathtt{A}$, feature $\mathtt{B}$ must be forbidden. Rule [**excl3**] indicates just the opposite: when feature $\mathtt{B}$ is computed, then $\mathtt{A}$ must also be forbidden. This rule might be surprising, but there is no reason to stem $\mathtt{A} \not\Leftrightarrow \mathtt{B}\ \mathtt{in}\ P$ in order to compute feature $\mathtt{B}$. So if $\mathtt{A} \not\Leftrightarrow \mathtt{B}\ \mathtt{in}\ P$ computes feature $\mathtt{B}$ then feature $\mathtt{A}$ must be forbidden. Otherwise the exclusion constraint would not have been fulfilled.

Rules [**forb1**] and [**forb2**] deal with the auxiliary operator $P \backslash \mathtt{A}$ that forbids the computation of feature $\mathtt{A}$. Let us note that there is no rule that computes $\mathtt{A}$. This means that if feature $\mathtt{A}$ is computed by P, the computation is blocked and no products can be produced.

Rules [**mand1**], [**mand2**], and [**mand3**] deal with the auxiliary operator $P \Rightarrow \mathtt{A}$ that indicates that $\mathtt{A}$ is mandatory. Rule [**mand1**] indicates that feature $\mathtt{A}$ must be computed

before *delivering* a product. Rules [**mand2**] and [**mand3**] indicates that $P \Rightarrow \texttt{A}$ behaves like P. We need two rules in this case because there when feature A is computed it is no longer necessary to continue considering this feature as mandatory. So the operator can be removed from the term.

We can see the operational semantics of a term as a *computational* tree (see the examples in Figures 3.4, 3.5, and 3.6). The root is the term itself and the branches are labelled with features. The branches of the tree represent the products of the term. We obtain a valid product when we reach a node that has an outgoing arc labelled with ✓. At this point no more features can be obtained in the corresponding branch. This is what the following Lemma establishes.

Lemma 3.2.2 Let $P, Q \in \texttt{fodaA}$, if $P \xrightarrow{\checkmark} Q$ then $Q = \texttt{nil}$. □

Once we have defined the operational semantics of the algebra, we can define the traces of an SPL, from these traces we will obtain its products.

Definition 3.2.3 A trace is a sequence $s \in (\mathcal{F} \cup \{\checkmark\})^*$, the empty trace is denoted by ϵ. Let s_1 and s_2 be traces, we denote the concatenation of s_1 and s_2 by $s_1 \cdot s_2$. Let $\texttt{A} \in \mathcal{F}$ and let s be a trace, we say that A *is in the trace* s, written $\texttt{A} \in s$, if there exist traces s_1 and s_2 such that $s = s_1 \cdot \texttt{A} \cdot s_2$.

We can extend the transitions in Definition 3.2.1 to traces. Let $P, Q, R \in \texttt{fodaA}$, we inductively define the transitions $P \xrightarrow{s} R$ as follows

- $P \xrightarrow{\epsilon} P$.
- If $P \xrightarrow{\texttt{A}} Q$ and $Q \xrightarrow{s} R$, then $P \xrightarrow{\texttt{A}\cdot s} R$.

□

Not all traces of an SPL produce valid products. Only traces ending with the symbol ✓ can be considered as products, therefore, in order to obtain the products of an SPL, we need to take its *successful* traces. That is the traces that end with the ✓ symbol. Let us note that the ✓ symbol is not a feature, so we do not include it in the trace.

Definition 3.2.4 Let $P \in \texttt{fodaA}$ and $s \in \mathcal{F}^*$, we say that s is a *successful trace* of P, written $s \in \texttt{tr}(P)$, if $P \xrightarrow{s} Q \xrightarrow{\checkmark} \texttt{nil}$. □

In the FODA formalism we cannot represent *the order* in which the features are produced. There could be situations where this is not the case, but in this work we have decided to stick to the usual results. For this reason traces are too strong to capture the products. Different traces can define the same product. For instance, the product obtained from the trace AB is the same as the one represented by the trace BA. Thus in order to get the products of an SPL we have to consider the traces not as sequences but as sets.

Definition 3.2.5 Let s be a trace. The *set induced by the trace*, written $[s]$, is the set obtained from the elements of the trace without considering their position in the trace.

Let $P \in \texttt{fodaA}$, we define *the products of* P, written $\texttt{prod}\,(P)$, as

$$\texttt{prod}\,(P) = \{[s] \mid s \in \texttt{tr}(P)\}$$

□

Example 3.2.6 Let us consider the trace $s = \texttt{ABC}$. It shows that $[s] = \{\texttt{A}, \texttt{B}, \texttt{C}\}$.

Now let us consider the fodaA term $P = \texttt{A}; (\overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark)$. The possible computations of P are:

$$
\begin{array}{ll}
P \xrightarrow{\texttt{A}} & \overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\checkmark} \texttt{nil} \\
P \xrightarrow{\texttt{A}} & \overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\texttt{B}} \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\checkmark} \texttt{nil} \\
P \xrightarrow{\texttt{A}} & \overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\texttt{B}} \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\texttt{C}} \checkmark \wedge \checkmark \xrightarrow{\checkmark} \texttt{nil} \\
P \xrightarrow{\texttt{A}} & \overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\texttt{C}} \overline{\texttt{B}}; \checkmark \wedge \checkmark \xrightarrow{\checkmark} \texttt{nil} \\
P \xrightarrow{\texttt{A}} & \overline{\texttt{B}}; \checkmark \wedge \overline{\texttt{C}}; \checkmark \xrightarrow{\texttt{C}} \overline{\texttt{B}}; \checkmark \wedge \checkmark \xrightarrow{\texttt{B}} \checkmark \wedge \checkmark \xrightarrow{\checkmark} \texttt{nil}
\end{array}
$$

Thus $\texttt{tr}(P) = \{\texttt{A}, \texttt{AB}, \texttt{ABC}, \texttt{AC}, \texttt{ACB}\}$. So $\texttt{prod}\,(P) = \{[\texttt{A}], [\texttt{AB}], [\texttt{ABC}], [\texttt{AC}]\}$ since $[\texttt{ABC}] = [\texttt{ACB}]$ □

In order to illustrate the operational semantics, next we will review the examples presented in Figure 3.2, giving their semantics.

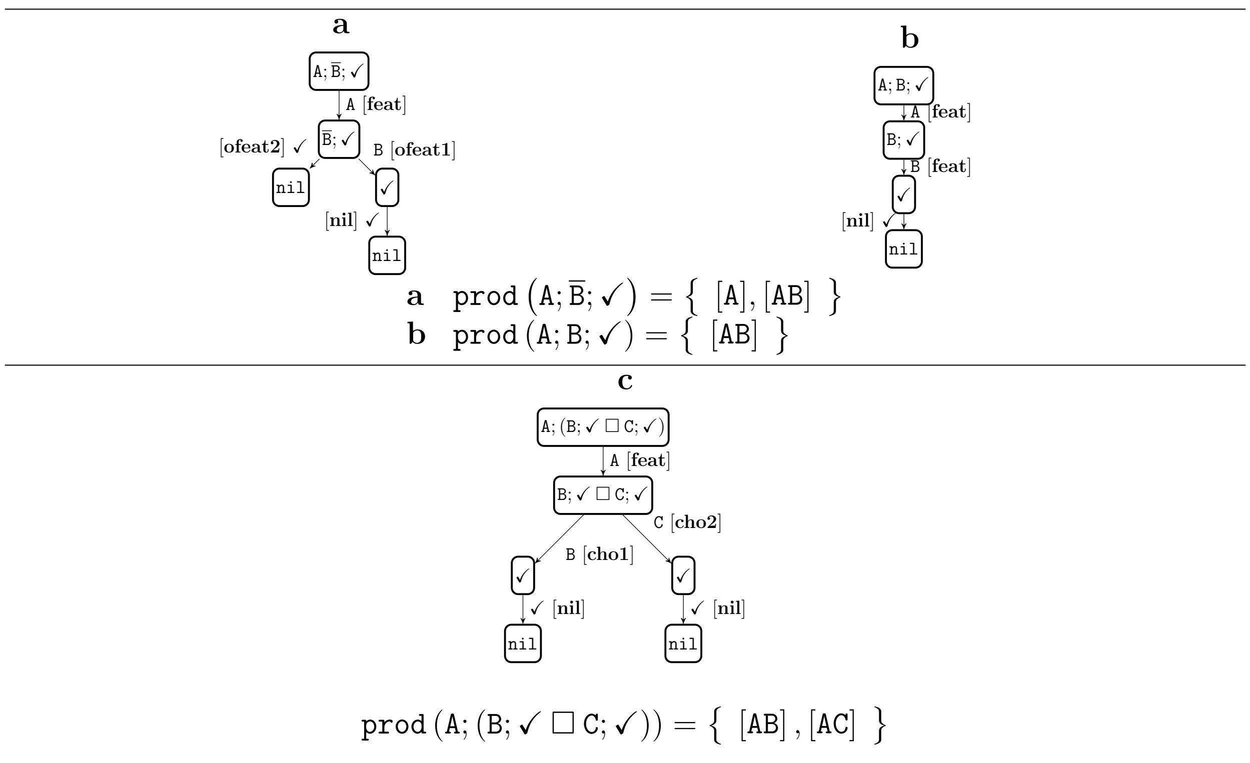


Figure 3.4: *Application of the operational semantic rules 1/3.*

Example 3.2.7 The semantics of the terms in Figure 3.2 have been split in Figures 3.4, 3.5, and 3.6.

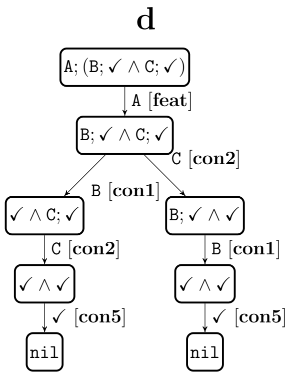
First let us comment the differences between examples **a** and **b**. In example **a** feature B is optional while in **b** it is mandatory. This is reflected in example **b** by the fact that there is a branch corresponding to the transition $\overline{\mathtt{B}}; \checkmark \xrightarrow{\checkmark} \mathtt{nil}$. This branch is not in Example **a**.

Now let us focus on Examples **c** and **d**. They show the difference between the *conjunction* operator and the *choose-one* operator. In the *choose-one* operator the member that is not needed for the computation *disappears*, while in the *conjunction* operator the other member remains. As a result Example **c** has the traces AB and AC, giving two different products [AB] and [AC]. Whereas the traces in example **d** are ABC and ACB, giving just the product [ABC][2].

Regarding the rest of the examples, the most remarkable aspects is the that in Example **e**

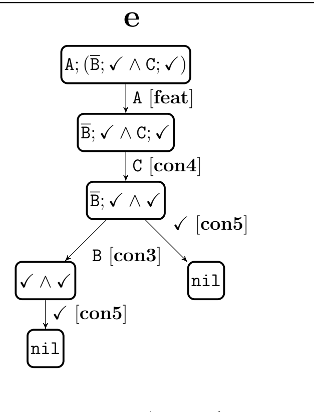
[2]Let us note that [ABC] = [ACB].

d



$$\texttt{prod}\,(\texttt{A};(\texttt{B};\checkmark \wedge \texttt{C};\checkmark)) = \{\ [\texttt{ABC}]\ \}$$

e



$$\texttt{prod}\left(\texttt{A};(\overline{\texttt{B}};\checkmark \wedge \texttt{C};\checkmark)\right) = \{\ [\texttt{ABC}], [\texttt{AC}]\ \}$$

Figure 3.5: *Application of the operational semantic rules 2/3.*

the only applicable rule in the second transition is [**con4**] because feature B is optional. So the only possible transition is the one labelled with feature C.

□

To conclude this section we will present some properties of the semantics we have been discussing. These properties indicate that the *require*, *exclusion*, *mandatory* and *forbidden* operators are behaving as expected.

Proposition 3.2.8 Let $P \in \texttt{fodaA}$ and $\texttt{A},\texttt{B} \in \mathcal{F}$. Then we have the following properties:

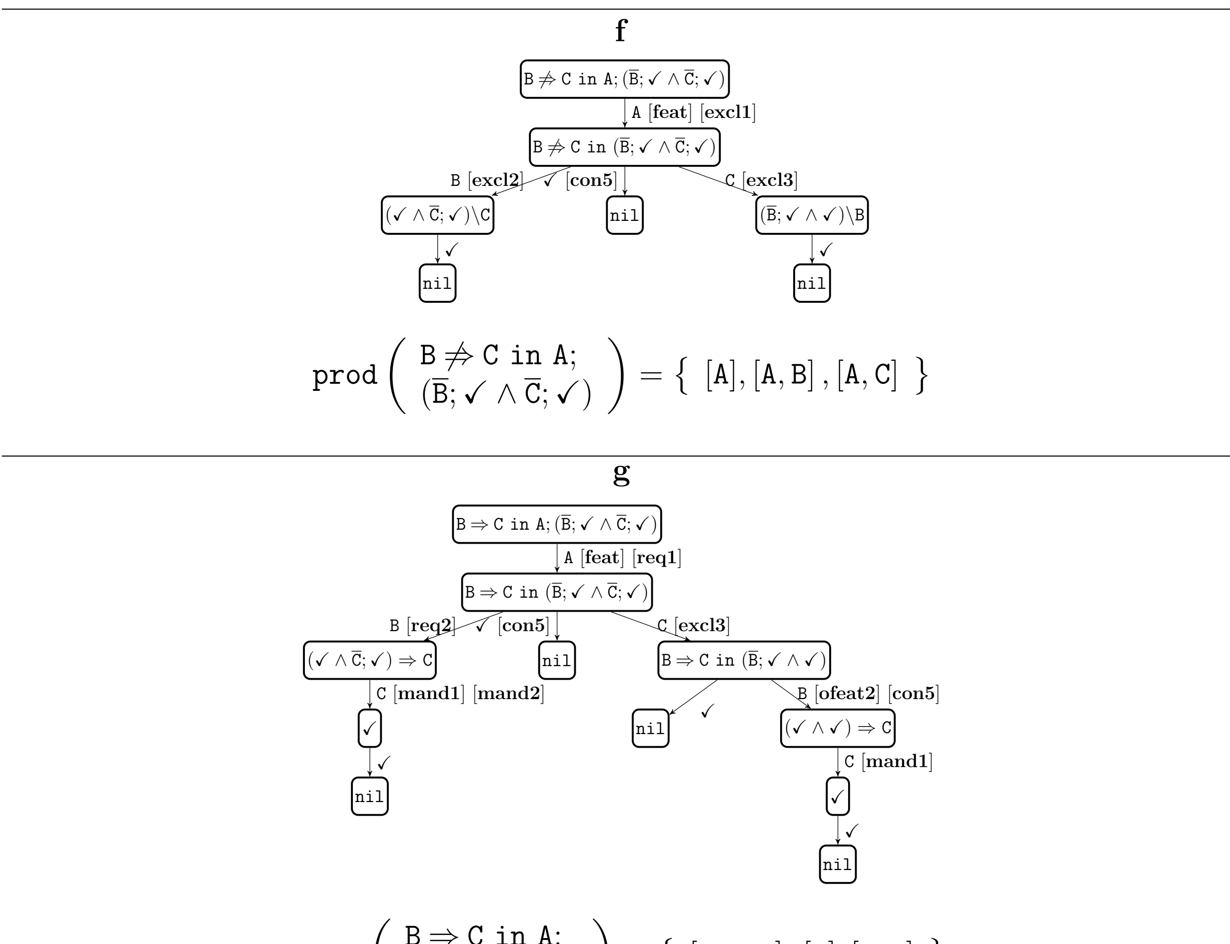


Figure 3.6: *Application of the operational semantic rules 3/3.*

1. If $s \in \texttt{tr}(P \Rightarrow \texttt{A})$, then $\texttt{A} \in s$.

2. If $s \in \texttt{tr}(P \backslash \texttt{A})$, then $\texttt{A} \notin s$.

3. If $s \in \texttt{tr}(\texttt{A} \Rightarrow \texttt{B} \texttt{ in } P)$ and $\texttt{A} \in s$, then $\texttt{B} \in s$.

4. If $s \in \texttt{tr}(\texttt{A} \not\Rightarrow \texttt{B} \texttt{ in } P)$ and $\texttt{A} \in s$, then $\texttt{B} \notin s$.

□

Once we have defined the products of SPL, it is the time to define an equivalence relationship based on products.

Definition 3.2.9 Let $P, Q \in$ `fodaA`. We define that they are equivalent, written $P \equiv Q$ if the products derived from both `SPLs` are the same: $\texttt{prod}\,(P) = \texttt{prod}\,(Q)$. $\square$

Since the relation we have just defined is based on set equality it is also an equivalence relationship. In Section 3.3 we will see that it is also a congruence.

Proposition 3.2.10 Let $P, Q, R \in$ `fodaA`. The following properties hold:

- $P \equiv P$.
- If $P \equiv Q$ then $Q \equiv P$.
- If $P \equiv Q$ and $Q \equiv R$ then $P \equiv R$.

$\square$

Next we are going to present some basic properties of the algebra, such as the commutativity and associativity of the binary operators. These properties are quite important since they allow us to extend the binary operators to n-ary operators.

Proposition 3.2.11 Let $P, Q, R \in$ `fodaA`. We have the following properties:

Commutativity $P \,\square\, Q \equiv Q \,\square\, P$ and $P \wedge Q \equiv Q \wedge P$.

Associativity $P \,\square\, (Q \,\square\, R) \equiv (P \,\square\, Q) \,\square\, R$ and $P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$.

$\square$

3.3 Denotational Semantics

The *denotational* semantics is more abstract than the operational one, since it does not rely on computation steps. In this section we provide a denotational semantics for `fodaA`. In order to define this denotational semantics, the first thing to do is to establish the *mathematical domain* where the syntactical objects of `fodaA` will be represented.

As we have commented in Section 3.1, the semantics of any `fodaA` expression is given by its set of products, and each product can be characterized by its features. So the mathematical domain we need is $\mathcal{P}(\mathcal{P}(\mathcal{F}))$[3], remembering that $\mathcal{F}$ is the set of features. The next step is to define a semantic operator for any of the syntactical operators in `fodaA`. This is done in the following definition.

Definition 3.3.1 Let $P, Q \in \mathcal{P}(\mathcal{P}(\mathcal{F}))$ be two sets of products and let $\mathtt{A}, \mathtt{B} \in \mathcal{F}$ be two features. We define the following operators:

- $[\![\mathtt{nil}]\!] = \varnothing$
- $[\![\checkmark]\!] = \{\varnothing\}$
- $[\![\mathtt{A};\cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\mathtt{A};\cdot]\!](P) = \{\{\mathtt{A}\} \cup p \mid p \in P\}$$

- $[\![\overline{\mathtt{A}};\cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\overline{\mathtt{A}};\cdot]\!](P) = \{\varnothing\} \cup \{\{\mathtt{A}\} \cup p \mid p \in P\}$$

- $[\![\cdot \,\square\, \cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \times \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\cdot \,\square\, \cdot]\!](P, Q) = P \cup Q$$

- $[\![\cdot \wedge \cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \times \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\cdot \wedge \cdot]\!](P, Q) = \{p \cup q \mid p \in P,\ q \in Q\}$$

- $[\![\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot]\!](P) = \begin{array}{l}\{p \mid p \in P, \mathtt{A} \notin p\} \cup \\ \{p \cup \{\mathtt{B}\} \mid p \in P, \mathtt{A} \in p\}\end{array}$$

[3] If X is a set, $\mathcal{P}(X)$ denotes the power set of X.

- $[\![\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot]\!](P) = \begin{array}{l}\{p \mid p \in P, \mathtt{A} \notin p\} \cup \\ \{p \mid p \in P, \mathtt{B} \notin p\}\end{array}$$

- $[\![\cdot \Rightarrow \mathtt{A}]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\cdot \Rightarrow \mathtt{A}]\!](P) = \{p \cup \{\mathtt{A}\} \mid p \in P\}$$

- $[\![\cdot \backslash \mathtt{A}]\!] : \mathcal{P}(\mathcal{P}(\mathcal{F})) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{F}))$ as

$$[\![\cdot \backslash \mathtt{A}]\!](P) = \{p \mid p \in P, \mathtt{A} \notin p\}$$

□

Once we have defined the semantic operators over a set of products, we can define the denotational semantics of any `fodaA` expression. It is defined inductively in the usual way.

Definition 3.3.2 The denotational semantics of `fodaA` is the function $[\![\cdot]\!] : \mathtt{fodaA} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{F}))$ inductively defined as follows: for any n-ary operator $\mathtt{op} \in \{\mathtt{nil}, \checkmark, \mathtt{A};\cdot, \overline{\mathtt{A}};\cdot, \cdot \Box \cdot, \cdot \wedge \cdot, \mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot, \mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot, \cdot \Rightarrow \mathtt{A}, \cdot \backslash \mathtt{A}\}$[4]:

$$[\![\mathtt{op}(P_1, \ldots P_n)]\!] = [\![\mathtt{op}]\!]([\![P_1]\!], \ldots, [\![P_n]\!])$$

□

Example 3.3.3 In order to illustrate the denotational semantics, we will apply it to the examples presented in Figure 3.2. The results are presented in Figure 3.7 . □

The rest of this section is devoted to proving that the set of products computed by the operational semantics coincides with the one computed with the denotational semantics.

[4] `nil` and $\checkmark$ are 0-ary operators; $\mathtt{A};\cdot$, $\overline{\mathtt{A}};\cdot$ $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot$, $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot$, $\cdot \Rightarrow \mathtt{A}$, $\cdot \backslash \mathtt{A}$ are 1-ary operators; $\cdot \Box \cdot$ and $\cdot \wedge \cdot$ are 2-ary operators.

a

$$
\begin{aligned}
\llbracket \checkmark \rrbracket &= \{\varnothing\} \\
\llbracket \overline{\mathtt{B}}; \checkmark \rrbracket &= \llbracket \overline{\mathtt{B}}; \cdot \rrbracket(\llbracket \checkmark \rrbracket) = \llbracket \overline{\mathtt{B}}; \cdot \rrbracket(\{\varnothing\}) = \{\varnothing, \{\mathtt{B}\}\} \\
\llbracket \mathtt{A}; \overline{\mathtt{B}}; \checkmark \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\llbracket \overline{\mathtt{B}}; \checkmark \rrbracket) = \llbracket \mathtt{A}; \cdot \rrbracket(\{\varnothing\} \cup \{\mathtt{B}\}) = \\
& \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{B}\}\}
\end{aligned}
$$

b

$$
\begin{aligned}
\llbracket \mathtt{B}; \checkmark \rrbracket &= \llbracket \mathtt{B}; \cdot \rrbracket(\llbracket \checkmark \rrbracket) = \llbracket \mathtt{B}; \cdot \rrbracket(\{\varnothing\}) = \{\{\mathtt{B}\}\} \\
\llbracket \mathtt{A}; \mathtt{B}; \checkmark \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\llbracket \mathtt{B}; \checkmark \rrbracket) = \llbracket \mathtt{A}; \cdot \rrbracket(\{\mathtt{B}\}) = \{\{\mathtt{A}, \mathtt{B}\}\}
\end{aligned}
$$

c

$$
\begin{aligned}
\llbracket \mathtt{B}; \checkmark \rrbracket &= \{\{\mathtt{B}\}\} \\
\llbracket \mathtt{C}; \checkmark \rrbracket &= \{\{\mathtt{C}\}\} \\
\llbracket \mathtt{B}; \checkmark \,\square\, \mathtt{C}; \checkmark \rrbracket &= \llbracket \cdot \,\square\, \cdot \rrbracket(\llbracket \mathtt{B}; \checkmark \rrbracket, \llbracket \mathtt{C}; \checkmark \rrbracket) = \\
& \llbracket \mathtt{B}; \checkmark \rrbracket \cup \llbracket \mathtt{C}; \checkmark \rrbracket = \{\{\mathtt{B}\}, \{\mathtt{C}\}\} \\
\llbracket \mathtt{A}; (\mathtt{B}; \checkmark \,\square\, \mathtt{C}; \checkmark) \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\llbracket \mathtt{B}; \checkmark \,\square\, \mathtt{C}; \checkmark \rrbracket) = \\
& \llbracket \mathtt{A}; \cdot \rrbracket(\{\{\mathtt{B}\}, \{\mathtt{C}\}\}) = \{\{\mathtt{A}, \mathtt{B}\}, \{\mathtt{A}, \mathtt{C}\}\}
\end{aligned}
$$

d

$$
\begin{aligned}
\llbracket \mathtt{B}; \checkmark \wedge \mathtt{C}; \checkmark \rrbracket &= \llbracket \cdot \wedge \cdot \rrbracket(\llbracket \mathtt{B}; \checkmark \rrbracket, \llbracket \mathtt{C}; \checkmark \rrbracket) = \\
\llbracket \cdot \wedge \cdot \rrbracket(\{\{\mathtt{B}\}, \{\mathtt{C}\}\}) &= \{\{\mathtt{B}, \mathtt{C}\}\} \\
\llbracket \mathtt{A}; (\mathtt{B}; \checkmark \wedge \mathtt{C}; \checkmark) \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\{\{\mathtt{B}, \mathtt{C}\}\}) = \\
& \{\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\}
\end{aligned}
$$

e

$$
\begin{aligned}
\llbracket \overline{\mathtt{B}}; \checkmark \wedge \mathtt{C}; \checkmark) \rrbracket &= \llbracket \cdot \wedge \cdot \rrbracket(\{\varnothing, \{\mathtt{B}\}\}, \{\{\mathtt{C}\}\}) = \\
& \{\{\mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}\} \\
\llbracket \mathtt{A}; (\overline{\mathtt{B}}; \checkmark \wedge \mathtt{C}; \checkmark) \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\llbracket \overline{\mathtt{B}}; \checkmark \wedge \mathtt{C}; \checkmark) \rrbracket) = \\
& \llbracket \mathtt{A}; \cdot \rrbracket(\{\{\mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}\}) = \\
& \{\{\mathtt{A}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\}
\end{aligned}
$$

f

$$
\begin{aligned}
\llbracket \overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark \rrbracket &= \llbracket \cdot \wedge \cdot \rrbracket(\llbracket \overline{\mathtt{B}}; \checkmark \rrbracket, \llbracket \overline{\mathtt{C}}; \checkmark \rrbracket) = \\
& \llbracket \cdot \wedge \cdot \rrbracket(\{\varnothing, \{\mathtt{B}\}\}, \{\varnothing, \{\mathtt{C}\}\}) = \\
& \{\varnothing, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}\} \\
\llbracket \mathtt{A}; (\overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark) \rrbracket &= \llbracket \mathtt{A}; \cdot \rrbracket(\llbracket \overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark \rrbracket) = \\
& \llbracket \mathtt{A}; \cdot \rrbracket(\{\varnothing, \{\mathtt{B}\}, \{\mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}\}) = \\
& \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{B}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\} \\
\left\llbracket \begin{array}{l} \mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{A}; \\ (\overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark) \end{array} \right\rrbracket &= \llbracket \mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \cdot \rrbracket \left(\begin{array}{l} \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{B}\}, \\ \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\} \end{array} \right) = \\
& \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{B}\}\}
\end{aligned}
$$

g

$$
\begin{aligned}
\left\llbracket \begin{array}{l} \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{A}; \\ (\overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark) \end{array} \right\rrbracket &= \llbracket \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \cdot \rrbracket \left(\begin{array}{l} \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{B}\}, \\ \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\} \end{array} \right) = \\
& \{\{\mathtt{A}\}, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}, \{\mathtt{A}, \mathtt{C}\}\}
\end{aligned}
$$

Figure 3.7: *Application of the denotational semantic rules.*

In order to prove this, first we need some auxiliary results that relate to the operational semantics with the denotational operators from Definition 3.3.1.

The first result deals with the termination of a trace. The products of an SPL are computed from the bottom up. That means that the first product *computed* by the denotational semantics is the product with no features. Let us note that $\{\varnothing\} = \mathtt{prod}(\checkmark) = \llbracket \checkmark \rrbracket$, but $\varnothing = \mathtt{prod}(\mathtt{nil}) = \llbracket \mathtt{nil} \rrbracket$.

Lemma 3.3.4 Let us consider $P \in \mathtt{fodaA}$, if $P \xrightarrow{\checkmark} \mathtt{nil}$ then $\varnothing \in \llbracket P \rrbracket$ $\square$

Next we will present a lemma for each operator of the syntax. Each of these lemmas

indicates that the corresponding semantic operator is well defined in Definition 3.3.1. These results will be needed in the inductive case of Theorem 3.3.12.

Lemma 3.3.5 Let $P \in \texttt{fodaA}$ and $\texttt{A} \in \mathcal{F}$, then $\texttt{prod}\,(\texttt{A}; P) = [\![\texttt{A};]\!](\texttt{prod}\,(P))$ and $\texttt{prod}\left(\overline{\texttt{A}}; P\right) = [\![\texttt{A};]\!](\texttt{prod}\,(P))$ $\square$

Lemma 3.3.6 Let $P, P' \in \texttt{fodaA}$, then we have that

$\texttt{prod}\,(P \,\square\, P') = [\![\square]\!](\texttt{prod}\,(P)\,, \texttt{prod}\,(P'))$. $\square$

Lemma 3.3.7 Let $P, P' \in \texttt{fodaA}$, then we have that

$\texttt{prod}\,(P \wedge P') = [\![\wedge]\!](\texttt{prod}\,(P)\,, \texttt{prod}\,(P'))$. $\square$

Lemma 3.3.8 Let $P \in \texttt{fodaA}$ and $\texttt{A} \in \mathcal{F}$, then we have that

$\texttt{prod}\,(P \Rightarrow \texttt{A}) = [\![\cdot \Rightarrow \texttt{A}]\!](\texttt{prod}\,(P))$. $\square$

Lemma 3.3.9 Let $P \in \texttt{fodaA}$ and $\texttt{A} \in \mathcal{F}$, then we have that

$\texttt{prod}\,(P \backslash \texttt{A}) = [\![\cdot \backslash \texttt{A}]\!](\texttt{prod}\,(P))$. $\square$

Lemma 3.3.10 Let $P \in \texttt{fodaA}$ and $\texttt{A}, \texttt{B} \in \mathcal{F}$, then we have that $\texttt{prod}\,(\texttt{A} \Rightarrow \texttt{B in } P) = [\![\texttt{A} \Rightarrow \texttt{B in } \cdot]\!](\texttt{prod}\,(P))$. $\square$

Lemma 3.3.11 Let $P \in \texttt{fodaA}$ and $\texttt{A}, \texttt{B} \in \mathcal{F}$, then we have that $\texttt{prod}\,(\texttt{A} \not\Rightarrow \texttt{B in } P) = [\![\texttt{A} \not\Rightarrow \texttt{B in } \cdot]\!](\texttt{prod}\,(P))$. $\square$

Now we have the result we were looking for: The denotational semantics and the operational semantics are equivalent.

Theorem 3.3.12 Let $P \in \texttt{fodaA}$, we have $\texttt{prod}\,(P) = [\![P]\!]$. $\square$

An immediate result from the previous theorem is that the equivalence relation $\equiv$ is a congruence.

Corollary 3.3.13 The equivalence relation $\equiv$ is a congruence. For any n-ary operator $\texttt{op}$, and $P_1, \ldots P_n, Q_1, \ldots Q_n \in \texttt{fodaA}$ such that $P_1 \equiv Q_1, \ldots, P_n \equiv Q_n$, we have

$$\texttt{op}(P_1, \ldots, P_n) \equiv \texttt{op}(Q_1, \ldots, Q_n)$$

3.4 Axiomatic Semantics

[REQ1] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{C}; P) =_E \mathtt{C}; (\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P)$

[REQ2] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{A}; P) =_E \mathtt{A}; (P \Rightarrow \mathtt{B})$

[REQ3] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{B}; P) =_E \mathtt{B}; P$

[REQ4] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P \,\Box\, Q =_E\ (\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P) \,\Box\, (\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ Q)$

[REQ5] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \checkmark =_E \checkmark$

[REQ6] $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \mathtt{nil} =_E \mathtt{nil}$

[MAND1] $(\mathtt{A}; P) \Rightarrow \mathtt{A} =_E \mathtt{A}; P$

[MAND2] $(\mathtt{B}; P) \Rightarrow \mathtt{A} =_E \mathtt{B}; (P \Rightarrow \mathtt{A})$

[MAND3] $\checkmark \Rightarrow \mathtt{A} =_E \mathtt{A}; \checkmark$

[MAND4] $\mathtt{nil} \Rightarrow \mathtt{A} =_E \mathtt{nil}$

[MAND5] $(P \,\Box\, Q) \Rightarrow \mathtt{A} =_E P \Rightarrow \mathtt{A} \,\Box\, Q \Rightarrow \mathtt{A}$

Figure 3.8: *Equations to remove requires and mandatory operators.*

[EXCL1] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{C}; P) =_E \mathtt{C}; (\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P)$

[EXCL2] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{A}; P) =_E \mathtt{A}; (P \backslash \mathtt{B})$

[EXCL3] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ (\mathtt{B}; P) =_E \mathtt{B}; (P \backslash \mathtt{A})$

[EXCL4] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P \,\Box\, Q =_E\ (\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P) \,\Box\, (\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ Q)$

[EXCL5] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \checkmark =_E \checkmark$

[EXCL6] $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \mathtt{nil} =_E \mathtt{nil}$

[FORB1] $(\mathtt{A}; P) \backslash \mathtt{A} =_E \mathtt{nil}$

[FORB2] $(\mathtt{B}; P) \backslash \mathtt{A} =_E \mathtt{B}; (P \backslash \mathtt{A})$

[FORB3] $\checkmark \backslash \mathtt{A} =_E \checkmark$

[FORB4] $\mathtt{nil} \backslash \mathtt{A} =_E \mathtt{nil}$

[FORB5] $(P \,\Box\, Q) \backslash \mathtt{A} =_E P \backslash \mathtt{A} \,\Box\, Q \backslash \mathtt{A}$

Figure 3.9: *Equations to remove exclusion, and forbid operators.*

In this section we will present an axiomatic semantics for `fodaA`. We are going to present *sound* and *complete* axioms for the language. As usual, *soundness* means that the

[CON1] $(\mathtt{A}; P) \wedge Q =_E \mathtt{A}; (P \wedge Q)$.

[CON2] $P \wedge Q =_E Q \wedge P$.

[CON3] $P \wedge (Q \,\Box\, R) =_E (P \wedge Q) \,\Box\, (P \wedge R)$.

[CON4] $P \wedge \mathtt{nil} =_E \mathtt{nil}$

[CON5] $P \wedge \checkmark =_E P$

Figure 3.10: *Axioms to remove the conjunction operator.*

[PRE1] $\mathtt{A}; \mathtt{B}; P =_E \mathtt{B}; \mathtt{A}; P$.

[PRE2] $\overline{\mathtt{A}}; P =_E (\mathtt{A}; P) \,\Box\, \checkmark$.

[PRE3] $(\mathtt{A}; P) \,\Box\, (\mathtt{A}; Q) =_E \mathtt{A}; (P \,\Box\, Q)$

[PRE4] $\mathtt{A}; \mathtt{nil} =_E \mathtt{nil}$

[PRE5] $\mathtt{A}; \mathtt{A}; P =_E \mathtt{A}; P$.

[CHO1] $P \,\Box\, Q =_E Q \,\Box\, P$.

[CHO2] $(P \,\Box\, Q) \,\Box\, R =_E P \,\Box\, (Q \,\Box\, R)$.

[CHO3] $P \,\Box\, \mathtt{nil} =_E P$.

[CHO4] $P \,\Box\, P = P$.

Figure 3.11: *Axioms for basic operators and optional features.*

equalities deduced from the axiom system are indeed correct: $P =_E Q$ implies $P \equiv Q$. The completeness means that all the identities can be deduced from the axiom system, that is $P \equiv Q$ implies $P =_E Q$.

Definition 3.4.1 Let $P, Q \in \mathtt{fodaA}$. We will say that we *deduce the equivalence* of P and Q if $P =_E Q$ can be deduced from the set of equations in Figures 3.8, 3.9, 3.10, and 3.11. □

To prove the soundness it is enough to show that the operators are congruent (Theorem 3.3.13) and that each axiom is correct.

Proposition 3.4.2 Let $\mathtt{A}, \mathtt{B} \in \mathcal{F}$ be two features, and P and Q be terms of $\mathtt{fodaA}$. The equations in Figures 3.8, 3.9, 3.10 and 3.11 are correct. □

d

$$\begin{aligned}
\mathtt{A};(\mathtt{B};\checkmark\wedge\mathtt{C};\checkmark) &=_E && \text{[CON1]}\\
\mathtt{A};\mathtt{B};(\checkmark\wedge\mathtt{C};\checkmark) &=_E && \text{[CON2]}, \quad \text{[CON5]}\\
\mathtt{A};\mathtt{B};\mathtt{C};\checkmark
\end{aligned}$$

e

$$\begin{aligned}
\mathtt{A};(\overline{\mathtt{B}};\checkmark\wedge\mathtt{C};\checkmark) &=_E && \text{[CON2]}, \quad \text{[CON1]}\\
\mathtt{A};\mathtt{C};(\checkmark\wedge\overline{\mathtt{B}};\checkmark) &=_E && \text{[CON2]}, \quad \text{[CON5]}\\
\mathtt{A};\mathtt{C};\overline{\mathtt{B}};\checkmark &=_E && \text{[PRE2]}\\
\mathtt{A};\mathtt{C};(\mathtt{B};\checkmark\,\Box\,\checkmark) &=_E && \text{[PRE3]}\\
\mathtt{A};(\mathtt{C};\mathtt{B};\checkmark\,\Box\,\mathtt{C};\checkmark) &=_E && \text{[PRE1]}\\
\mathtt{A};(\mathtt{B};\mathtt{C};\checkmark\,\Box\,\mathtt{C};\checkmark)
\end{aligned}$$

Figure 3.12: *Transformation to normal form 1/3.*

To prove the completeness we will need the concept of *normal forms*. In order to define the normal forms we are going to prove that some operators are *derived* from the *basic* operators. These basic operators are the base ones ($\mathtt{nil}$ and $\checkmark$), the prefix operator ($\mathtt{A};P$), and the *choose-one* operator ($P\,\Box\,Q$). The following example will show how some operators can be removed.

Example 3.4.3 Let us consider the following SPL $P=\mathtt{A};\checkmark\wedge\mathtt{B};\checkmark$. It easy to compute its successful traces which are $\{\mathtt{AB},\mathtt{BA}\}$, so $\mathtt{prod}(P)=\{[\mathtt{AB}]\}$. This SPL has the same products as $\mathtt{A};\mathtt{B};\checkmark$. Indeed, by applying the indicated axioms we have the following deduction

$$\begin{aligned}
\mathtt{A};\checkmark\wedge\mathtt{B};\checkmark &=_E && \text{[CON1]}\\
\mathtt{A};(\checkmark\wedge\mathtt{B};\checkmark) &=_E && \text{[CON2]}\\
\mathtt{A};(\mathtt{B};\checkmark\wedge\checkmark) &=_E && \text{[CON1]}\\
\mathtt{A};\mathtt{B};(\checkmark\wedge\checkmark) &=_E && \text{[CON5]}\\
\mathtt{A};\mathtt{B};\checkmark
\end{aligned}$$

□

f

$$\begin{array}{rl}
\mathtt{A};(\overline{\mathtt{B}};\checkmark \wedge \overline{\mathtt{C}};\checkmark) =_E & \textbf{[PRE2]}\\
\mathtt{A};((\mathtt{B};\checkmark \,\square\, \checkmark) \wedge (\mathtt{C};\checkmark \,\square\, \checkmark)) =_E & \textbf{[CON3]}\\
\mathtt{A};\big((\mathtt{B};\checkmark \wedge \mathtt{C};\checkmark)\square & \\
(\mathtt{B};\checkmark \wedge \checkmark) \,\square\, (\checkmark \wedge \mathtt{C};\checkmark) \,\square\, (\checkmark \wedge \checkmark)\big) =_E & \textbf{[CON1]}\\
 & \textbf{[CON5]}\\
\mathtt{A};\big((\mathtt{B};\mathtt{C};\checkmark) \,\square\, (\mathtt{B};\checkmark) \,\square\, (\mathtt{C};\checkmark) \,\square\, \checkmark\big) &
\end{array} \tag{3.1}$$

$$\begin{array}{rl}
\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{A};(\overline{\mathtt{B}};\checkmark \wedge \overline{\mathtt{C}};\checkmark) =_E & (3.1)\\
\mathtt{A};\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \left(\begin{array}{l}(\mathtt{B};\mathtt{C};\checkmark)\square\\ (\mathtt{B};\checkmark) \,\square\, (\mathtt{C};\checkmark) \,\square\, \checkmark\end{array}\right) =_E & \textbf{[EXCL4]}\\
\mathtt{A};\left(\begin{array}{l}(\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{B};\mathtt{C};\checkmark)\square\\ (\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{B};\checkmark)\square\\ (\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{C};\checkmark)\square\\ (\mathtt{B} \not\Rightarrow \mathtt{C}\ \mathtt{in}\ \checkmark)\end{array}\right) =_E & \begin{array}{l}\textbf{[EXCL1]}\\ \textbf{[EXCL2]}\\ \textbf{[EXCL5]}\end{array}\\
\mathtt{A};\left(\begin{array}{l}(\mathtt{B};(\mathtt{C};\checkmark\backslash\mathtt{C}))\square\\ (\mathtt{B};(\checkmark\backslash\mathtt{C}))\square\\ (\mathtt{C};(\checkmark\backslash\mathtt{B}))\square\\ \checkmark\end{array}\right) =_E & \begin{array}{l}\textbf{[FORB1]}\\ \textbf{[FORB3]}\end{array}\\
\mathtt{A};\big((\mathtt{B};\mathtt{nil}) \,\square\, (\mathtt{B};\checkmark) \,\square\, (\mathtt{C};\checkmark) \,\square\, \checkmark\big) =_E & \textbf{[PRE5]}\\
\mathtt{A};\big(\mathtt{nil} \,\square\, (\mathtt{B};\checkmark) \,\square\, (\mathtt{C};\checkmark) \,\square\, \checkmark\big) =_E & \textbf{[CHO3]}\\
\mathtt{A};\big((\mathtt{B};\checkmark) \,\square\, (\mathtt{C};\checkmark) \,\square\, \checkmark\big) &
\end{array}$$

Figure 3.13: *Transformation to normal form 2/3.*

Definition 3.4.4 Let $P \in \mathtt{fodaA}$, we say that it is a *basic term* if it can be generated by the following grammar

$$P ::= \ \checkmark \mid \mathtt{nil} \mid \mathtt{A};P \mid P \,\square\, Q$$

We denote the set of basic terms as $\mathtt{fodaA_b}$. □

The set of axioms in Figures 3.8, 3.9, 3.10, plus the axiom **[PRE2]** in Figure 3.11 allow the non-basic operators to be removed from any $P \in \mathtt{fodaA}$. The idea is to prove that $Q \in \mathtt{fodaA_b}$ exists such that $P \equiv Q$.

Let us suppose that we have a term $P \in \mathtt{fodaA}$ that contains a non-basic operator. Then we can find $Q \in \mathtt{fodaA}$ where either, the non-basic operator has disappeared or it is deeper

g

$$\begin{array}{rl}
\mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{A}; (\overline{\mathtt{B}}; \checkmark \wedge \overline{\mathtt{C}}; \checkmark) =_E & (3.1) \\
\mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{A}; \left(\begin{array}{l} \mathtt{B}; (\mathtt{C}; \checkmark \square \checkmark) \square \\ (\mathtt{C}; \checkmark \square \checkmark) \end{array} \right) =_E & \text{[REQ1]} \\
\mathtt{A}; \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \left(\begin{array}{l} \mathtt{B}; (\mathtt{C}; \checkmark \square \checkmark) \square \\ (\mathtt{C}; \checkmark \square \checkmark) \end{array} \right) =_E & \text{[REQ4]} \\
\mathtt{A}; \left(\begin{array}{l} \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{B}; (\mathtt{C}; \checkmark \square \checkmark) \square \\ \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ (\mathtt{C}; \checkmark \square \checkmark) \end{array} \right) =_E & \text{[REQ2]} \quad \text{[REQ4]} \\
\mathtt{A}; \left(\begin{array}{l} \mathtt{B}; (\mathtt{C}; \checkmark \square \checkmark) \Rightarrow \mathtt{C} \square \\ (\mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \mathtt{C}; \checkmark \square \mathtt{B} \Rightarrow \mathtt{C}\ \mathtt{in}\ \checkmark) \end{array} \right) =_E & \text{[REQ2]} \quad \text{[REQ5]} \\
\mathtt{A}; \left(\begin{array}{l} \mathtt{B}; (\mathtt{C}; \checkmark \Rightarrow \mathtt{C} \square \checkmark \Rightarrow \mathtt{C}) \square \\ (\mathtt{C}; \checkmark \square \checkmark) \end{array} \right) =_E & \begin{array}{c}\text{[MAND1]} \\ \text{[MAND3]} \\ \text{[CHO4]}\end{array} \\
\mathtt{A}; (\mathtt{B}; \mathtt{C}; \checkmark \ \square \mathtt{C}; \checkmark \square \checkmark) &
\end{array}$$

Figure 3.14: *Transformation to normal form 3/3.*

in the syntactic tree of Q. Then iterating this process we can make all non-basic operators disappear. Then we have the theorem we are looking for.

Theorem 3.4.5 Let $P \in \mathtt{fodaA}$, there exists $Q \in \mathtt{fodaA_b}$ so that $P =_E Q$. □

Since we know how to remove the non-basic operators of an `fodaA` term, we are going to focus on proving the completeness restricted to basic terms. To do so, we are going to define what are our *normal forms*, and then we are going to prove that any basic term can be transformed to a normal form by using the basic axioms in Figure 3.11.

In order to give the formal definitions of normal forms we need to give some auxiliary definitions. First we assume that there is an order relation $\leq \subseteq \mathcal{F} \times \mathcal{F}$ that must be isomorphic to the natural numbers in case $\mathcal{F}$ is infinite. Next, we will need the *vocabulary* of a basic `fodaA` term, that is the set of features appearing in the expression.

Definition 3.4.6 Let $P, Q \in \mathtt{fodaA_b}$ be two basic `fodaA` terms. We define the *vocabulary* as the function $\mathsf{voc} : \mathtt{fodaA_b} \rightarrow \mathcal{P}(\mathcal{F})$ defined inductively as:

- $\mathsf{voc}(\mathtt{nil}) = \mathsf{voc}(\checkmark) = \varnothing$

- $\mathsf{voc}(\mathtt{A}; P) = \{\mathtt{A}\} \cup \mathsf{voc}(P)$
- $\mathsf{voc}(P \,\Box\, Q) = \mathsf{voc}(P) \cup \mathsf{voc}(Q)$

□

Before giving the definition of normal forms, we are going to define a simpler case that is the case of pre-normal forms.

Definition 3.4.7 A basic `fodaA` term $P \in \mathtt{fodaA_b}$ is in *pre-normal*, written $P \in \mathtt{fodaA_{pre}}$, iff it has one of the following forms.

1. `nil`, ✓, or
2. If there exists a $n > 0$, $\{\mathtt{A_1}, \ldots, \mathtt{A_n}\} \subseteq \mathcal{F}$, and there exist $P_1, \ldots P_n \in \mathtt{fodaA_{pre}}$ with $P_i \neq \mathtt{nil}$ for $1 \leq i \leq n$ and $\{\mathtt{A_i}, \ldots \mathtt{A_n}\} \cap \mathsf{voc}(P_j) = \varnothing$ for $1 \leq j \leq n$ and either

   $$P = (\mathtt{A_1}; P_1) \,\Box \cdots (\mathtt{A_n}; P_n)$$

   or

   $$P = (\mathtt{A_1}; P_1) \,\Box \cdots (\mathtt{A_n}; P_n) \,\Box\, \checkmark$$

   In this case we say that the features $\{\mathtt{A_1}, \ldots, \mathtt{A_n}\}$ are at the top level of P.

□

Next we present an auxiliary lemma that will be used in Proposition 3.4.15. This lemma establishes that if a feature appears in the vocabulary of a pre-normal form then it appears in at least one product of the pre-normal form. Let us note that this result is not true[5] in ordinary terms because of the restrictions that might appear in the terms.

Lemma 3.4.8 Let $P \in \mathtt{fodaA_{pre}}$, then

$$\mathsf{voc}(P) = \{\mathtt{A} \mid \mathtt{A} \in p,\ p \in \mathtt{prod}\,(P)\}$$

□

[5]The vocabulary of an ordinary term has not been formally defined. Definition 3.4.6 could easily be extended to the set of features appearing in the syntax of a term.

The next lemma establishes that if a feature $\mathtt{A}$ appears in a pre-normal form P, the normal form can be transformed into another equivalent normal from Q so that $\mathtt{A}$ level of the syntax tree of Q.

Lemma 3.4.9 Let $P \in \mathtt{fodaA_{pre}}$ and let $\mathtt{A} \in \mathtt{voc}(P)$. Then there is $Q \in \mathtt{fodaA_{pre}}$ such that $P \equiv Q$ and $\mathtt{A}$ is at the top level of Q. □

The next proposition establishes the first result we need to prove the completeness. Any term can be transformed into an equivalent pre-normal form. The result is restricted to basic terms but, because of Theorem 3.4.5, it can be extended to any ordinary term.

Proposition 3.4.10 Let $P \in \mathtt{fodaA_b}$, there exists a pre-normal form $Q \in \mathtt{fodaA_{pre}}$ such that $P =_E Q$. □

Normal forms are an extension of pre-normal forms. The problem with pre-normal forms is that there are syntactically different expressions that are equivalent.

Example 3.4.11 Let us consider the following $\mathtt{fodaA_b}$ expressions:

$$P = (\mathtt{A};\mathtt{C};\checkmark) \,\Box\, \mathtt{B};\checkmark$$
$$Q = (\mathtt{C};\mathtt{A};\checkmark) \,\Box\, \mathtt{B};\checkmark$$

Both expressions are in pre-normal form and both are equivalent.

The way to obtain a unique normal form for any $\mathtt{fodaA_b}$ expression is to use the above mentioned order among features. Let us assume $\mathtt{A} < \mathtt{B} < \mathtt{C}$; in this case we will say that P is in normal form while Q is not. □

Definition 3.4.12 Let us consider $P \in \mathtt{fodaA_{pre}}$. We will say that P is a *normal form*, written $P \in \mathtt{fodaA_{nf}}$ iff $P = \mathtt{nil}$, $P = \checkmark$ or if the sets $\{\mathtt{A_1}, \ldots, \mathtt{A_n}\}$ and $\{P_1, \ldots, P_n\}$ in Definition 3.4.7.2 verify:

- $\mathtt{A_i} < \mathtt{A_j}$ for $1 \leq i < j \leq n$.
- $\mathtt{A_i} < \mathtt{B}$ for any $\mathtt{B} \in \mathtt{voc}(P_j)$ for $1 \leq i \leq j \leq n$.

□

Example 3.4.13 Figures 3.12, 3.13 and 3.14 show the normal forms corresponding to the examples in Figure 3.2 assuming that $\mathtt{A} < \mathtt{B} < \mathtt{C}$. The examples **a**, **b**, and **c** are not included due to the fact, they are already the normal forms. □

Now we have our first result. Any expression can be transformed in a normal form.

Proposition 3.4.14 Let $P \in \mathtt{fodaA}_{\mathtt{pre}}$. Then there exists a normal form $Q \in \mathtt{fodaA}_{\mathtt{nf}}$ such that $P =_E Q$. □

The next result shows that two normal forms that are semantically equivalent, are also identical at the semantic level.

Proposition 3.4.15 Let $P, Q \in \mathtt{fodaA}_{\mathtt{nf}}$. If they are semantically equivalent, $P \equiv Q$, then they are syntactically identical $P = Q$.

□

Finally, we can prove the main result of this section: The deductive system is sound and complete.

Theorem 3.4.16 Let us consider $P, Q \in \mathtt{fodaA}$. Then $P \equiv Q$ if and only if $P =_E Q$. □

Chapter 4

Tool and Case Study

Nothing is particularly hard if you divide it into small jobs.

Henry Ford

This chapter presents the `fodaA` tool, called `AT`. This application allows users to work and interact with the formal framework presented in this research project. This tool is implemented in Python as a stand-alone project [1]. The tool is license is GPL v3 [2].

Also, in this chapter we present a case study analyzing a Video Streaming Software, for live broadcasting. This model represents a `SPL` modeled using `FODA`, and how their functional parts interact for building the set of valid products. This case study models a `SPL`, and is represented using `AT`. We also show the semantic rules applied to generate the set of products.

4.1 Tool

`AT` is a tool,which implements the denotational semantic of `fodaA`. This framework allows users to automatize the translation methodology presented in this master thesis, from `FODA` Diagrams to `fodaA` terms, and the analysis of these terms by producing the set of valid products. The tool obtains the data from a XML file in which a `FODA` feature model is defined.

[1] http://simba.fdi.ucm.es/at/denotational.py

[2] GNU GPL official website: http://www.gnu.org/copyleft/gpl.html

```
<xml version="1.0" encoding="ISO-8859-1">
        <requires feature_1="D" feature_2="G">
                <excludes feature_1="F" feature_2="E">
                        <mandatory_feature name="A">
                                <paralel>
                                        <optional_feature name="B">
                                                <checkmark></checkmark>
                                        </optional_feature>
                                        <optional_feature name="C">
                                                <choose_1>
                                                        <mandatory_feature name="D">
                                                                <mandatory_feature name="G">
                                                                        <checkmark></checkmark>
                                                                </mandatory_feature>
                                                        </mandatory_feature>
                                                        <mandatory_feature name="E">
                                                                <checkmark></checkmark>
                                                        </mandatory_feature>
                                                        <mandatory_feature name="F">
                                                                <optional_feature name="H">
                                                                        <checkmark></checkmark>
                                                                </optional_feature>
                                                        </mandatory_feature>
                                                </choose_1>
                                        </optional_feature>
                                </paralel>
                        </mandatory_feature>
                </excludes>
        </requires>
</xml>
```

Figure 4.1: *Example of* `AT` *XML input.*

The first step of the framework consists in transforming the feature model defined by the XML file, into the internal data format of the application. Next, the algorithm that applies the semantics rules to the terms of `fodaA` is applied. Then the set of possible products for the `fodaA` term are computed. Finally, a set of interesting and well studied properties [3, 14, 9, 20, 36] of `FODA` Diagrams are presented.

1. The set of products represented by the model.

2. Information about errors. `AT` will show if an error is showed if a feature can not be part of any product. In which case this feature model contains errors.

3. Information about warnings. `AT` will show warnings with respect to the requires and excludes constraints presented in `FODA`. Warnings are displayed if model constraints does not affect any product of the final set of valid products.

Figure 4.1 presents an example of a XML input for `AT` framework. `AT` is a very simple and intuitive application, the only software requirements are to have installed the python interpreter and the lxml library.

First we are going to build the input for AT. To be able to use this application you must know how to translate a FODA feature model into our XML syntax.

These are the basic constructors:

- XML indicator: This tag indicates that indeed this will be a XML file.

```
<xml version="1.0" encoding="ISO-8859-1"></xml>
```

- Mandatory features: This tag indicates that this will represent a Mandatory Feature inside the feature model.

```
<mandatory_feature name="your_feature_name_goes_here"></mandatory_feature>
```

- Optional features: This tag shows that this will represent an optional feature inside the model.

```
<optional_feature name="your_feature_name_goes_here"></optional_feature>
```

- Paralel constructor: This tag represents that we are going to group a set of features using the parallel operator.

```
<paralel></paralel>
```

- Choose 1 constructor: This tag indicates that we are going to group a set of features using the choose 1 relationship. Note: The childs of a choose 1 feature node, must be Mandatory features.

```
<choose_1></choose_1>
```

- Requires constraint: This tag indicates that feature A requires feature B.

```
<requires name="your_feature_A_name_goes_here" name="your_feature_B_name_goes_here"></requires>
```

- Excludes constraint: This tag indicates that feature A excludes feature B.

```
<excludes name="your_feature_A_name_goes_here" name="your_feature_B_name_goes_here"></excludes>
```

Figure 4.1 includes a feature model in XML format. Figure 4.2 describes how to use AT from the command line.

An output of AT is shown in Figure 4.3. In this example the set of valid products are described, and if there are any errors or warnings.

```
box$~ python denotational.py <input_file.xml>
```

Figure 4.2: *Example of **AT** script execution.*

```
box$~ python denotational.py example_5.xml
Checking example_5.xml
There are 4 products:
         Product 1 : ['B', 'C', 'D', 'A']
         Product 2 : ['B', 'C', 'A']
         Product 3 : ['B', 'D', 'A']
         Product 4 : ['B', 'A']
There is 0 warnings.
```

Figure 4.3: *Example of **AT** denotational script output.*

In the next section we present a case study of our theoretical approach, where a Video Streaming Software for Live Broadcasting is formally represented.

4.2 Case Study

The FODA Diagram of the case study for representing the Video Streaming Software is presented in Figure 4.4. This system incorporates the following features: VSS, TBR, VCC, 720Kbps, 256Kbps, H.264 and MPEG.2. The *initial feature*[3] for this SPL is VSS (Video Streaming Software). This feature does not represent any feature itself, but instead will represent the domain in which the SPL is defined. Any product of this SPL will need this first feature. In this feature model all possible combinations of the features are represented as a set of products by relating features, relationships and constraints. Let us note that each feature has an unique name which appears in the domain terminology dictionary of the *Video Streaming Software* SPL.

Next, we present all features, and indicate their intuitive meaning and the relationships between them. The initial first feature is VSS, (this mean Video Streaming Software). Its related features are: TBR (the Transmission Bit-Rate) and VCC (the Video Codec) features. There is a *Mandatory* relationship between them (it is graphically represented by using an arc with a black filled circle in the end). This relationship requires that this set of features are included in all implementations of the software, where the VSS feature is included. There are more *Mandatory* relationships in the diagram. For instance, feature 720Kbps has a mandatory relationship with its *parent*. There are other features in this system that are included as optional. For instance, feature 256Kbps is optional, meaning that this feature might be included in the final products of the SPL. Let us note that the inclusion or exclusion of this feature, may depend on relationships with other features. For example, if feature H.264 is included in a product, then feature 256Kbps is excluded, this means that 256Kbps must not appear in any product were feature H.264 is included. Finally, let us consider the features H.264 and MPEG.2, the choose 1 operator relates this two features. This means, that only one of the features will be included in the final product of the SPL. In this system, there is a set of *rules* that describe some *restrictions*. For instance, when MPEG.2 is included,

[3]Sometime called the hard-system or the domain of the SPL.

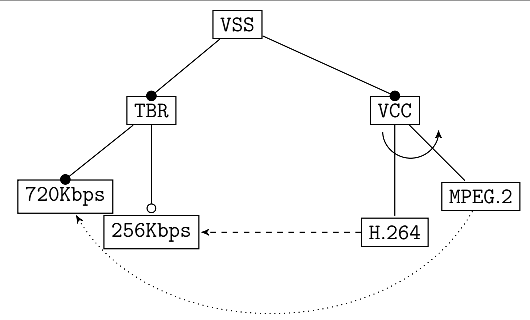


Figure 4.4: *Video Streaming Software - FODA representation.*

720Kbps is also included. Furthermore, when the feature H.264 is selected, 256Kbps must not appear. Let us consider the constraint that relates MPEG.2 with the feature 720Kbps. There is an *Implies* constraint, that is, if MPEG.2 feature is selected, then the speed of the TBR must be 720Kbps. Let us note that this constraint is not necessary since 720Kbps is mandatory, we have included it to show that sometimes FODA allows the inclusion of repetitive (or useless) information. There is another constraint that focuses on features H.264 and 256Kbps , it is an *Excludes* constraint, meaning that if H.264 appears, then the final product cannot contain 256Kbps. Finally, a *valid* product of this model will be a set of features that fulfill diagram constraints. For example, let us consider the following products: $\mathtt{pr}_1$ consisting in VSS, TBR, 720Kbps, VCC, and H.264, $\mathtt{pr}_2$ consisting in VSS, TBR, 720Kbps, VCC, and MPEG.2, and $\mathtt{pr}_3$ consisting in VSS, TBR, 720Kbps, 256Kbps, VCC, and H.264. In this case, $\mathtt{pr}_1$ and $\mathtt{pr}_2$ are *valid* products of this model, but $\mathtt{pr}_3$ is not a valid product of this model.

In order to present the formal analysis, we first focus on the *translation process*. The Video Stream Software consists in two components: TBR (the Transmission Bit-Rate) and VCC (the Video Codec). Since the two components are in any final product, they are related with a mandatory relationship which is represented in fodaA by the parallel operator. TBR is translated into a parallel of two features: 256Kbps and 720Kbps, feature 256Kbps is optional and 720Kbps is mandatory. The VCC can be either H.264 or MPEG.2, these features are related with the choose 1 operator. Finally, in the FODA diagram there are two constraints. The

first one is a require constraint, and the second one an exclude constraint. Formally, the complete translation of this model into our algebra is P_1:

$$
\begin{array}{lll}
P_1 & := & \texttt{MPEG.2} \Rightarrow \texttt{720Kbps in} \\
 & & \quad \texttt{H.264} \not\Rightarrow \texttt{256Kbps in } P_2 \\
\text{where} & & \\
\quad P_2 & := & \texttt{VSS}; (P_{21} \wedge P_{22}) \\
\quad P_{21} & := & \texttt{TBR}; (\texttt{720Kbps}; \checkmark \wedge \overline{\texttt{256Kbps}}; \checkmark) \\
\quad P_{22} & := & \texttt{VCC}; (\texttt{H.264}; \checkmark \, \Box \, \texttt{MPEG.2}; \checkmark)
\end{array}
$$

A labeled transition system associated to the term P_1 is depicted in Figure 4.5. We have not presented the full labeled transition system, we have skipped the subtrees (1) and (2) because from those subtrees we do not obtain new products, it is easy to check that the traces obtained from those subtrees does not give new products. From the tree depicted we obtain the following traces:

- `VSS TBR VCC 720Kbps H.264`,
- `VSS TBR VCC 720Kbps MPEG.2 720Kbps`,
- `VSS TBR VCC 720Kbps MPEG.2 256Kbps 720Kbps`,
- `VSS TBR VCC MPEG.2 720Kbps`, and
- `VSS TBR VCC MPEG.2 720Kbps 256Kbps`.

From those traces we obtain the products:

- $\{\texttt{VSS}, \texttt{TBR}, \texttt{VCC}, \texttt{720Kbps}, \texttt{H.264}\}$,
- $\{\texttt{VSS}, \texttt{TBR}, \texttt{VCC}, \texttt{720Kbps}, \texttt{MPEG.2}\}$, and
- $\{\texttt{VSS}, \texttt{TBR}, \texttt{VCC}, \texttt{720Kbps}, \texttt{MPEG.2}, \texttt{256Kbps}\}$

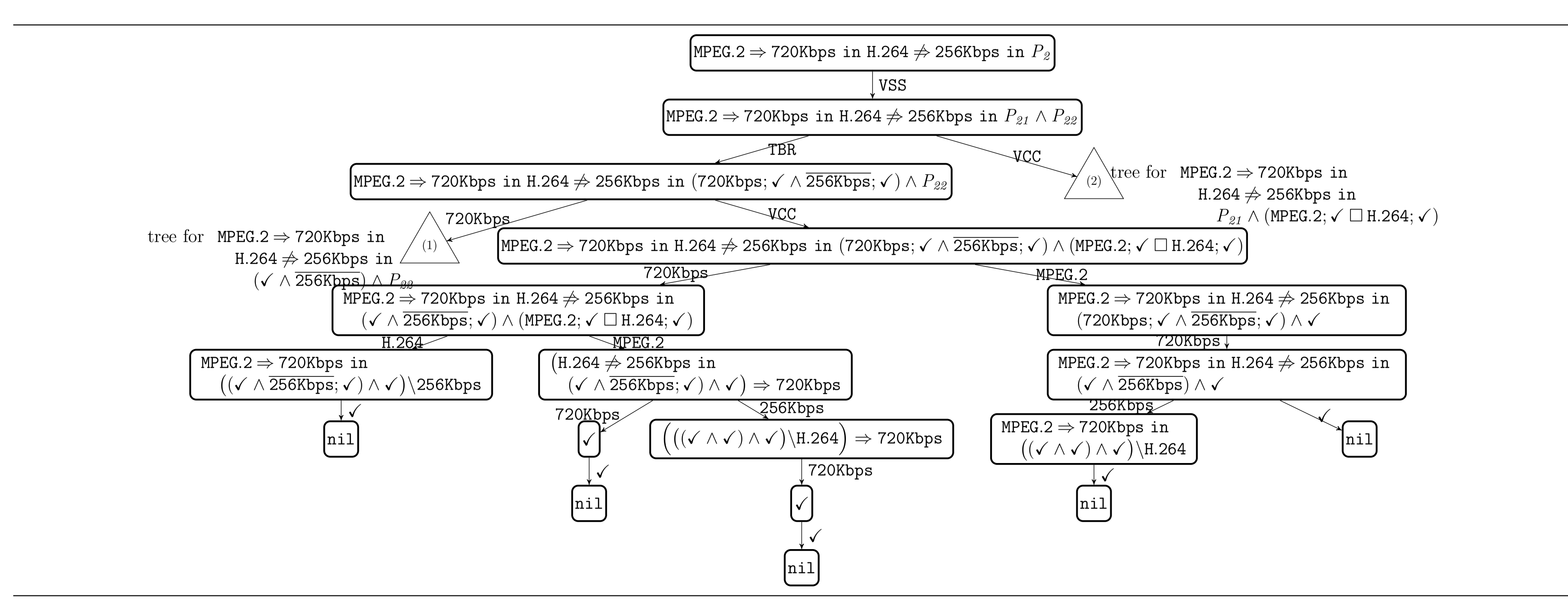


Figure 4.5: *Labelled Transitions System of the Video Streaming Software.*

$$
\begin{array}{lcl}
\llbracket \checkmark \rrbracket & = & \{\varnothing\} \\
\llbracket \texttt{H.264}; \checkmark \rrbracket & = & \{\{\texttt{H.264}\} \cup \varnothing\} \;=\; \{\{\texttt{H.264}\}\} \\
\llbracket \texttt{MPEG.2}; \checkmark \rrbracket & = & \{\{\texttt{MPEG.2}\}\} \\
\llbracket \texttt{720Kbps}; \checkmark \rrbracket & = & \{\{\texttt{720Kbps}\}\} \\
\llbracket \overline{\texttt{256Kbps}}; \checkmark \rrbracket & = & \{\varnothing\} \cup \{\{\texttt{256Kbps}\} \cup \varnothing\} \;=\; \{\varnothing, \{\texttt{256Kbps}\}\}
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket P_{22} \rrbracket & = & \llbracket \texttt{VCC}; (\texttt{H.264}; \checkmark \;\square\; \texttt{MPEG.2}; \checkmark) \rrbracket \;=\; \llbracket \texttt{VCC}; \cdot \rrbracket \big(\llbracket \square \rrbracket (\{\{\texttt{H.264}\}\}, \{\{\texttt{MPEG.2}\}\}) \big) = \\
 & & \llbracket \texttt{VCC}; \cdot \rrbracket \big(\llbracket \texttt{H.264}; \checkmark \rrbracket \cup \llbracket \texttt{MPEG.2}; \checkmark \rrbracket \big) \;=\; \llbracket \texttt{VCC}; \cdot \rrbracket \big(\{\{\texttt{H.264}\}, \{\texttt{MPEG.2}\}\} \big) = \{\{\texttt{VCC}, \texttt{H.264}\}, \{\texttt{VCC}, \texttt{MPEG.2}\}\} \\
\llbracket P_{21} \rrbracket & = & \llbracket \texttt{TBR}; (\texttt{720Kbps}; \checkmark \wedge \overline{\texttt{256Kbps}}; \checkmark) \rrbracket \;=\; \llbracket \texttt{TBR}; \cdot \rrbracket \big(\llbracket \wedge \rrbracket (\{\{\texttt{720Kbps}\}, \{\varnothing, \{\texttt{256Kbps}\}\}\}) \big) \;= \\
 & & \llbracket \texttt{TBR}; \cdot \rrbracket \big(\{\{\texttt{720Kbps}\}, \{\texttt{720Kbps}, \texttt{256Kbps}\}\} \big) = \{\{\texttt{TBR}, \texttt{720Kbps}\}, \{\texttt{TBR}, \texttt{720Kbps}, \texttt{256Kbps}\}\}
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket P_{21} \wedge P_{22} \rrbracket & = & \llbracket \wedge \rrbracket \left(\begin{array}{l} \{\{\texttt{VCC}, \texttt{H.264}\}, \{\texttt{VCC}, \texttt{MPEG.2}\}\}, \{\{\texttt{TBR}, \texttt{720Kbps}\}, \\ \{\texttt{TBR}, \texttt{720Kbps}, \texttt{256Kbps}\}\} \end{array} \right) = \\
 & & \left\{ \begin{array}{l} \{\texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \{\texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}, \texttt{256Kbps}\}, \\ \{\texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \{\texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \\
\llbracket P_{2} \rrbracket & = & \llbracket \texttt{VSS}; (P_{21} \wedge P_{22}) \rrbracket = \llbracket \texttt{VSS}; \cdot \rrbracket \left(\left\{ \begin{array}{l} \{\texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \\ \{\texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}, \texttt{256Kbps}\}, \\ \{\texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \\ \{\texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \right) = \\
 & & \left\{ \begin{array}{l} \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}, \texttt{256Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \\
\llbracket \texttt{H.264} \not\Rightarrow \texttt{256Kbps in } P_{2} \rrbracket & = & \llbracket \texttt{H.264} \not\Rightarrow \texttt{256Kbps in } \cdot \rrbracket \left(\left\{ \begin{array}{l} \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}, \texttt{256Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \right) = \\
 & & \left\{ \begin{array}{l} \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \\
\left\llbracket \begin{array}{l} \texttt{MPEG.2} \Rightarrow \texttt{720Kbps in} \\ \quad \texttt{H.264} \not\Rightarrow \texttt{256Kbps in } P_{2} \end{array} \right\rrbracket & = & \llbracket \texttt{MPEG.2} \Rightarrow \texttt{720Kbps in } \cdot \rrbracket \left(\left\{ \begin{array}{l} \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}, \texttt{256Kbps}\} \end{array} \right\} \right) = \\
 & & \left\{ \begin{array}{l} \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{H.264}, \texttt{720Kbps}\}, \\ \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}\}, \{\texttt{VSS}, \texttt{VCC}, \texttt{TBR}, \texttt{MPEG.2}, \texttt{720Kbps}, \texttt{256Kbps}\} \end{array} \right\}
\end{array}
$$

Figure 4.6: *Denotational Semantics of Video Streaming Software.*

Next, the denotational semantics for P_1 is presented in Figure 4.6. Let us note that the set of products is not modified after applying the last semantic operator $\llbracket \texttt{MPEG.2} \Rightarrow \texttt{720Kbps in } \cdot \rrbracket$. This means that the *requires* restriction is not necessary in this case. As it is expected (see Theorem 3.3.12), the set of products obtained by applying the denotational semantics coincides with the set of products computed with the operational semantics: $\texttt{prod}\,(P_1) = \llbracket P_1 \rrbracket$.

Finally, we present the *deduction process* induced by the axioms in Figures 4.7 and 4.8. can observe how the initial expression P_1 in transformed until we obtain a pre-normal form (Figure 4.8, equation 4.5):

$$
\texttt{VSS}; \texttt{TBR}; \texttt{VCC}; \texttt{720Kbps}; \big((\texttt{H.264}; \checkmark \quad \square \quad \texttt{MPEG.2}; \texttt{720Kbps}; (\overline{\texttt{256Kbps}}; \checkmark \quad \square \quad \checkmark) \big)
$$

This term is not in normal form because of two reasons. First we have not established the order among features, we can assume the following ordering: $\texttt{VSS} < \texttt{TBR} < \texttt{VCC} < \texttt{720Kbps} <$

$$
\begin{aligned}
\mathtt{TBR}; (\mathtt{720Kbps}; \checkmark \wedge \overline{\mathtt{256Kbps}}; \checkmark) =_E &\quad \textbf{[CON1]}\\
\mathtt{TBR}; \mathtt{720Kbps}; (\checkmark \wedge \overline{\mathtt{256Kbps}}; \checkmark) =_E &\quad \textbf{[CON5]} \quad \textbf{[CON2]}\\
\mathtt{TBR}; \mathtt{720Kbps}; \overline{\mathtt{256Kbps}}; \checkmark &
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
P_2 = \mathtt{VSS}; (P_{21} \wedge P_{22}) =_E &\quad (4.1)\\
\mathtt{VSS}; \big(\mathtt{TBR}; \mathtt{720Kbps}; (\mathtt{256Kbps}; \checkmark \Box \checkmark)\big) \wedge \big(\mathtt{VCC}; (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \checkmark)\big) =_E &\quad \textbf{[CON1]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\overline{\mathtt{256Kbps}}; \checkmark) \wedge (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \checkmark)\big) =_E &\quad \textbf{[CON3]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\overline{\mathtt{256Kbps}}; \checkmark \wedge \mathtt{H.264}; \checkmark) \Box (\overline{\mathtt{256Kbps}}; \checkmark \wedge \mathtt{MPEG.2}; \checkmark\big) =_E &\quad \textbf{[CON1]} \quad \textbf{[CON2]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; (\overline{\mathtt{256Kbps}}; \checkmark \wedge \checkmark) \Box \mathtt{MPEG.2}; (\overline{\mathtt{256Kbps}}; \checkmark \wedge \checkmark)\big) =_E &\quad \textbf{[CON5]} \quad \textbf{[CON2]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; (\mathtt{H.264}; \overline{\mathtt{256Kbps}}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark) &
\end{aligned}
\tag{4.2}
$$

$$
\begin{aligned}
\mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ P_2 =_E &\quad (4.2)\\
\mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \big(\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; (\mathtt{H.264}; \overline{\mathtt{256Kbps}}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[EXCL1]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \big((\mathtt{H.264}; \overline{\mathtt{256Kbps}}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[EXCL4]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \mathtt{H.264}; \overline{\mathtt{256Kbps}}; \checkmark) \Box (\mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[EXCL2]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\mathtt{H.264}; (\overline{\mathtt{256Kbps}}; \checkmark \backslash \mathtt{256Kbps})) \Box (\mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[EXCL1]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\mathtt{H.264}; (\overline{\mathtt{256Kbps}}; \checkmark \backslash \mathtt{256Kbps})) \Box (\mathtt{MPEG.2}; \mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[FORB??]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box (\mathtt{MPEG.2}; \mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[FORB??]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box (\mathtt{MPEG.2}; \mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ \overline{\mathtt{256Kbps}}; \checkmark)\big) =_E &\quad \textbf{[EXCL??]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; (\checkmark \backslash \mathtt{H.264})\big) =_E &\quad \textbf{[EXCL??]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark) =_E &\quad \textbf{[FORB3]}
\end{aligned}
\tag{4.3}
$$

$$
\begin{aligned}
P_1 = \mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{H.264} \not\Leftrightarrow \mathtt{256Kbps}\ \mathtt{in}\ P_2 =_E &\quad (4.3)\\
\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark) &\quad \textbf{[REQ1]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark) &\quad \textbf{[REQ4]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big((\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{H.264}; \checkmark) \Box (\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) &\quad \textbf{[REQ1]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; (\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \checkmark) \Box (\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) &\quad \textbf{[REQ5]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box (\mathtt{MPEG.2} \Rightarrow \mathtt{720Kbps}\ \mathtt{in}\ \mathtt{MPEG.2}; \overline{\mathtt{256Kbps}}; \checkmark)\big) &\quad \textbf{[REQ2]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; (\overline{\mathtt{256Kbps}}; \checkmark \Rightarrow \mathtt{720Kbps})\big) &\quad \textbf{[MAND??]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; (\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \mathtt{720Kbps}; \overline{\mathtt{256Kbps}}; \checkmark) =_E &\quad \textbf{[PRE2]}\\
\mathtt{VSS}; \mathtt{TBR}; \mathtt{VCC}; \mathtt{720Kbps}; \big(\mathtt{H.264}; \checkmark \Box \mathtt{MPEG.2}; \mathtt{720Kbps}; (\mathtt{256Kbps}; \checkmark \Box \checkmark)\big) &
\end{aligned}
\tag{4.4}
$$

Figure 4.7: *Deduction Rules applied to the Video Streaming Software (1/2).*

$$
\begin{aligned}
P_1 = \texttt{MPEG.2} \Rightarrow \texttt{720Kbps in H.264} \not\Rightarrow \texttt{256Kbps in } P_2 &=_E && (4.3)\\
\texttt{MPEG.2} \Rightarrow \texttt{720Kbps in VSS; TBR; VCC; 720Kbps; (H.264; } \checkmark \ \Box \ \texttt{MPEG.2; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark) & && [\textbf{REQ1}]\\
\texttt{VSS; TBR; VCC; 720Kbps; MPEG.2} \Rightarrow \texttt{720Kbps in (H.264; } \checkmark \ \Box \ \texttt{MPEG.2; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark) & && [\textbf{REQ4}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big((\texttt{MPEG.2} \Rightarrow \texttt{720Kbps in H.264; } \checkmark) \ \Box \ (\texttt{MPEG.2} \Rightarrow \texttt{720Kbps in MPEG.2; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark)\big) & && [\textbf{REQ1}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; (MPEG.2} \Rightarrow \texttt{720Kbps in } \checkmark) \ \Box \ (\texttt{MPEG.2} \Rightarrow \texttt{720Kbps in MPEG.2; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark)\big) & && [\textbf{REQ5}] \qquad (4.5)\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ (\texttt{MPEG.2} \Rightarrow \texttt{720Kbps in MPEG.2; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark)\big) & && [\textbf{REQ2}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ \texttt{MPEG.2; } (\overline{\texttt{256Kbps}}\texttt{; } \checkmark \Rightarrow \texttt{720Kbps})\big) & && [\textbf{MAND??}]\\
\texttt{VSS; TBR; VCC; 720Kbps; (H.264; } \checkmark \ \Box \ \texttt{MPEG.2; 720Kbps; } \overline{\texttt{256Kbps}}\texttt{; } \checkmark) &=_E && [\textbf{PRE2}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ \texttt{MPEG.2; 720Kbps; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) & &&
\end{aligned}
$$

$$
\begin{aligned}
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ \texttt{MPEG.2; 720Kbps; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) &=_E && [\textbf{PRE1}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ \texttt{720Kbps; MPEG.2; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) &=_E && [\textbf{PRE3}]\\
\texttt{VSS; TBR; VCC; } \big(\texttt{720Kbps; H.264; } \checkmark \ \Box \ \texttt{720Kbps; 720Kbps; MPEG.2; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) &=_E && [\textbf{PRE5}]\\
\texttt{VSS; TBR; VCC; } \big(\texttt{720Kbps; H.264; } \checkmark \ \Box \ \texttt{720Kbps; MPEG.2; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) &=_E && [\textbf{PRE5}] \qquad (4.6)\\
\texttt{VSS; TBR; VCC; } \big(\texttt{720Kbps; H.264; } \checkmark \ \Box \ \texttt{720Kbps; MPEG.2; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) &=_E && [\textbf{PRE5}]\\
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \ \Box \ \texttt{MPEG.2; (256Kbps; } \checkmark \ \Box \ \checkmark)\big) & &&
\end{aligned}
$$

Figure 4.8: *Deduction Rules applied to the Video Streaming Software (2/2).*

$\texttt{H.264} < \texttt{MPEG.2} < \texttt{256Kbps}$. Second, the repetition of feature `720Kbps` in this term is not allowed in normal form. We obtain the normal form by applying the rules that appear in Equation(4.6), and the resulting term is:

$$
\texttt{VSS; TBR; VCC; 720Kbps; } \big(\texttt{H.264; } \checkmark \quad \Box \quad \texttt{MPEG.2; (256Kbps; } \checkmark \quad \Box \quad \checkmark)\big)
$$

That is indeed a normal form for the ordering presented above.

Chapter 5

Conclusions and Future Work

Where you start is not as important as where you finish.

Zig Ziglar

5.1 Conclusions

In this Master Thesis project, we have identified and studied several ways to represent Product Families, in particular, Software Product Lines. From that point, we realized that there are some proposals to define SPLs formally. All frameworks like FODA, RSEB, and PLUSS do not have a strong formal representation. In the literature, there are some proposals that provide formal semantics for FODA; but actually there are not any semantics with similar level of abstraction to that we use in our framework.

fodaA Syntax

We presented the syntax of our algebra as an EBNF-like expression. This includes the definition of the fodaA operators, and how to translate a FODA model into fodaA. All the diagrams presented in the FODA formalism were translated into fodaA. After presenting fodaA, we have introduced the formal semantics of this algebra. The approach we have followed in this work to provide the semantics, was inspired by the classical process algebra notion [33, 19, 21]. It is important to denote that the translation of any FODA Diagram is quite natural and simple in our approach.

`fodaA` Semantics

We have defined three different semantics for `fodaA`: first we have introduced an operational semantic of a `SPL`, next we have defined a denotational semantic that is less intuitive but easier to implement, and finally we have defined an axiomatic semantic. The axiomatic semantic allows to reason about the SPL without computing their concrete products. We have proved that these three semantics are equivalent.

Automated diagnosis

In addition to the formal framework, a tool called `AT` has been developed. The main goal of this tool is to support this novel theoretical framework. In particular, this tool implements the formal semantics presented in this Master Thesis. Moreover, an automatic diagnosis of any `FODA` Diagram can be easily done with this tool, where the following properties can be answered:

- *Can this `SPL` produce a valid product?*
- *If we remove some constraint, the final set of valid product changes?*
- *Is this a void model?*
- *Is `Pr1` a valid product?*
- *How many different products can be developed?*
- *Are there any dead features?*
- *Are there any false optional features?*
- *If there are features present in all configurations, can we group them in a single one as a core feature?*

Finally, an additional contribution of this work is a complete case study where a non-trivial system is analyzed. This system allows to configure a *Video Streaming Software* for live broadcasting. The users can configure several features: the video codification, the text bit rate, etc., and their constraints.

5.2 Future Work

First, we plan to study a several number of semantics for FODA. In our current semantics, the order in which the features are computed is not important. However, there are situations where this is no longer true. That could be the case, if we consider the cost associated with the production of a product. For instance, producing feature A and then B could have a cost of 1€, but producing B and then A might cost 2€. This extension would require the definition of new semantics for fodaA. Also, adding meta-data to models, will enable, for example, making for example economical decisions over feature models and checking if a product cost more than other.

Let us put a specific upgrade to our approach taking our research as basis. If it is upgraded the information represented in AT we should be able to represent a weight array associated with each feature inside the model. This array may be associated within the domain terminology dictionary of the SPL. So we add two extensions to our approach. The first is the inclusion of the numerical array to each feature. The second will be the upgrade of the *domain terminology dictionary* information for our model, this, for handling information about the meaning of each position of the array weight over each feature. In this case, this domain terminology dictionary must be defined in a well knew language, like UML. So forth we rely on the definition of these data to another hi-level information structure.

Another interesting line of research is to study a large and real project with our tool. We have just started studying Debian-based linux distributions (Debian[1] and Ubuntu[2]). We think that these kind of distributions matches particularly well with the concept of SPL. In this case features will consist in the different packages available in those distributions. Therefore there are the following open issues:

- Are there any dead features? That is if there is a package that can never be installed.
- How many different ways can I install a web server?.

[1] http://www.debian.org

[2] http://www.ubuntu.com

- By adding meta-data to the model we can infer if a configuration is better than other (For example, it is the most updated configuration available).

Also, popular software products like Content Management Systems can be analyzed. For example, Drupal [3], Expression Engine [4] or Plone [5]. These software products are built upon a set of modules that can be easily integrated to our approach by redefining them into features. And analyze them to check if those software products are truly consistent in their implementation.

Finally, another piece of software that might be included into the `SPL` category is the programming language Python[6]. Python has a management system of external modules with relations among them.

[3] `http://www.drupal.org`
[4] `http://www.expressionengine.com`
[5] `http://www.plone.org`
[6] `http://www.python.org`

Bibliography

[1] P. Asirelli, M.H.t. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Design and validation of variability in product lines. In *2nd International Workshop on Product Line Approaches in Software Engineering, PLEASE '11*, pages 25–30, New York, NY, USA, 2011. ACM.

[2] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. A deontic logical framework for modelling product families. In *VaMoS*, pages 37–44, 2010.

[3] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference, SPLC '05*, pages 7–20. Springer, 2005.

[4] D. Batory, D. Benavides, and A. Ruiz. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49:45–47, 2006.

[5] D. Benavides, S. Segura, and A. Ruiz. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[6] Y. Bontemps, P. Heymans, P. Schobbens, and J.C. Trigaux. Semantics of foda feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, pages 48–58. Springer, 2004.

[7] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley, 2000.

[8] P. Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, March 1976.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[10] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC*, pages 23–34. IEEE Computer Society, 2007.

[11] M. Eriksson, J. Borstler, and K. Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *Proceedings of the 9th International Conference on Software Product Lines*, pages 33–44. Springer-Verlag, 2006.

[12] A. Fantechi and S. Gnesi. A behavioural model for product families. In *6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, ESEC-FSE companion '07*, pages 521–524. ACM Press, 2007.

[13] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *Proceedings of the 2008 12th International Software Product Line Conference*, pages 193–202, Washington, DC, USA, 2008. IEEE Computer Society.

[14] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *First Alloy Workshop*, 2006.

[15] M. Griss and J. Favaro. Integrating feature modeling with the rseb. In *In Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, 1998.

[16] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[17] A. Gruler, M. Leucker, and K.D. Scheidemann. Modeling and model checking software product lines. In Gilles Barthe and Frank S. de Boer, editors, *10th IFIP WG 6.1 International Conference, FMOODS'08, LNCS 5051*, volume 5051 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2008.

[18] M. Harsu. A survey on domain engineering. Technical report, Institute of Software Systems, Tampere University of Technology, 2002.

[19] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[20] P. Heymans, P.Y. Schobbens, J.C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *IET Software*, 2(3):281–302, 2008.

[21] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[22] P. Höfner, R. Khédri, and B. Möller. Feature algebra. In *14th Int. Symp. on Formal Methods, FM'06, LNCS 4085*, pages 300–315. Springer, 2006.

[23] P. Höfner, R. Khédri, and B. Möller. An algebra of product families. *Software and System Modeling*, 10(2):161–182, 2011.

[24] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In *11th Int. Conf. on Fundamental approaches to software engineering, FASE'08, LNCS 4961*, pages 31–45. Springer, 2008.

[25] T. Käkölä and J. C. Dueñas. *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006.

[26] K. Kang, K. Sajoong, L. Jaejoon, K. Kijoo, S. Euiseob, and H. Moonhang. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 1998.

[27] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[28] K.Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[29] K. G. Larsen, U. Nyman, and A-Wasowski. On modal refinement and consistency. In *18th Int. Conf. on Concurrency Theory, CONCUR'07, LNCS 4703*, pages 105–119, 2007.

[30] K.G. Larsen, U. Nyman, and A. Wkasowski. Modal I/O automata for interface and product line theories. In *16th European conference on Programming, ESOP'07*, pages 64–79. Springer, 2007.

[31] M. Mannion. Using first-order logic for product line model validation. In *2nd International Software Product Line Conference, SPLC'02*, pages 176–187, London, UK, UK, 2002. Springer-Verlag.

[32] J. Mellado. *Estrategias de prueba de líneas de producto de sistemas de tiempo real especificados con diagramas de estados jerárquicos*. PhD thesis, Universidad Politécnica de Madrid, 2004.

[33] R. Milner. *A Calculus of Communicating Systems (LNCS 92)*. Springer, 1980.

[34] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[35] S. Segura. Extended support for the automated treatment of feature models. Technical report, Universidad de Sevilla, 2008.

[36] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz. Automated metamorphic testing on the analyses of feature models. *Information & Software Technology*, 53(3):245–258, 2011.

[37] P. Sochos, I. Philippow, and M. Riebisch. Feature-oriented development of software product lines: Mapping feature models to the architecture. In Mathias Weske and Peter Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, volume

3263 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30196-7_11.

[38] D. Fischbein S. Uchitel and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *Workshop on Role of software architecture for testing and analysis, ROSATEA'06*, pages 39–48. ACM Press, 2006.