Feature Modelling:
A Survey, a Formalism and a Transformation for Analysis

Thomas De Vylder

University of Antwerp, Belgium

Abstract

Feature modelling is the greatest contribution of software product line engineering to software engineering. It is a must when dealing with reusable software. It helps to identify and capture variability. First a survey is given in which feature modelling is explained in detail. Second a formalism is presented in which feature models are easily designed as feature diagrams. This meta-model is created using a tool for multi-formalism and meta-modelling called AToM^3. It provides an interface to design models that comply to the formalism. The created meta-model allows feature diagrams that are not trees, but more general graphs. In addition to the parental relationships of features and concepts, cross-graph constraints are possible. Thirdly, instead of creating a new formal analysis tool for the introduced formalism, a transformation to a lightweight modelling language called Alloy is presented. In domain engineering, formal analysis can provide useful insights into the design. The provided transformation, based on the model-driven architecture, can be automatically analysed by the Alloy Analyzer. As a final point a conclusion is given and some future work is proposed.

Keywords:
Domain Engineering, Software Product Line Engineering, Model-Driven Architecture, Formalism, AToM^3, Formal Analysis, Transformation, Alloy Analyzer

1. Introduction

For some people feature modelling is still quite a vague subject. Therefore this paper will give a survey in which not only the details of feature modelling are explained, but also the context from which it originated. There are not a lot of tools out there that provide an easy interface to design feature models. Hence in this paper a new feature modelling tool is introduced that is crafted using a tool for multi-paradigm modelling. Formal analysis, which will also be discussed in this paper, is an important part of feature modelling. A tool that

Email address: thomas.devylder@student.ua.ac.be (Thomas De Vylder)

allows feature modelling should provide functionality to perform such analysis on the models. Consequently a method is proposed that allows formal analysis. Instead of adding direct functionality to the tool, a transformation is proposed to a textual language. Using an already published tool to perform analysis on that language, one can thus perform analysis on feature models. This method is similar to the work of Anastasakis et al. (2010).

Section 2 presents the survey of feature modelling. Section 3 introduces the tool to design feature models. Section 4 presents the problem of formal analysis and gives the transformation as a solution. Section 5 concludes and section 6 presents interesting possibilities for future work.

2. A Survey of Feature Modelling

2.1. Origins and Definitions

Feature modelling was first introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang et al. (1990). FODA is a domain analysis method that became part of Model-Based Software Engineering (MBSE). In this paper FODA is described on its updated definition by Withey (1996) and Czarnecki and Eisenecker (2000). The FODA process consists of two phases[1].

The first phase is called context analysis. This phase defines the scope of the domain in which the products are created. Knowledge of the domain is used to bound this scope. Furthermore, relationships are created between this domain and other domains or entities.

The second phase is called domain modelling. During this phase the engineer will try to produce a domain model by identifying and modelling the main commonalities and variabilities between the applications in the domain. Domain modelling consists of three steps.

1. Information analysis converts domain knowledge to the form of domain entities and their relations. Examples of such techniques are object-oriented modelling, entity-relationship modelling or semantic networks. This step produces the information model.
2. Feature analysis "captures a customer's or end-user's understanding of the general capabilities of applications in a domain. For a domain, the commonalities and differences among related systems of interest are designated as features and are depicted in the feature model." - Czarnecki and Eisenecker (2000)
3. Operational analysis investigates the commonalities and differences between control and data flows of the domain. It yields the operational model that captures the behavioural relationships between the objects in the information model and the features in the feature model.

[1] Originally FODA contained a third phase called architectural modelling. This phase was converted into the Domain Design phase of the MBSE.

Another important result of the domain modelling phase is the domain dictionary, which defines the terminology used in the domain.

Since its introduction feature modelling has been vigorously used in the Software Product Line (SPL) community. As stated by Clements and Northrop (2002), a SPL is a family of programs. When the building blocks of the programs are features, every program in a SPL is identified by a unique and legal combination of features. This is called a *feature configuration*.

Following the conceptual modelling perspective of Smith and Medin (1981) a *feature* expresses the commonalities and differences of a concept instance. A *concept* can be any element or structure in the domain. A feature is defined by Kang et al. (1990) as a "prominent or distinctive user-visible aspect, quality or characteristic of a software system or system".

SPL engineering tries to systematically and efficiently create similar programs. The FODA analysis is used to identify the features in a domain that is covered by a certain SPL.

Features can occur at any level (e.g. high-level system requirements, architectural level, subsystem and component level and object and procedure level). Modelling their semantics requires some modelling formalism (e.g. object diagrams, interaction diagrams, state-transition diagrams, synchronization constraints). *Feature models* are just one of the many models describing a piece of reusable software. A *feature diagram* is a visual notation of a feature model.

2.2. Notations and Semantics

There are many possible notations for feature models. First the basics will be explained.

2.2.1. Basic Feature Models

A diagram can be viewed as a graph of nodes. Nodes can be either *feature nodes* or *concept nodes*[2]. The parent node of a feature node is either a feature node or a concept node. When the diagram is a tree (like in most cases), the root will thus be a concept node. In Figure 1 an example of a simple feature diagram is given. Feature_0 and Feature_1 are features of Concept_0 and Feature_2 is a feature of Feature_0 and thus a *sub-feature* of Concept_0.

There are four types of relationships between a parent node and its child nodes (sub-features).

1. The *mandatory* relationship implies that the child node is included in the description of a concept instance if and only if the parent node is included. Note that a concept node is always included. The mandatory relationship is represented by a simple edge from the parent node to the child node that ends with a filled circle. An example of a feature diagram with mandatory features is given in figure 1.

[2]Sometimes large diagrams are split up into a number of smaller diagrams. The roots of these smaller diagrams are features of the concept node in the original diagram.

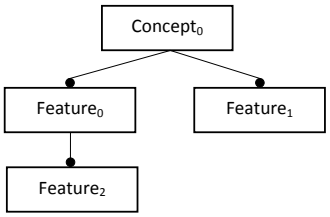

Figure 1: An example of a feature diagram with mandatory features.

2. The *optional* relationship implies that the child node may be included in the description of a concept instance if and only if the parent node is included. This relationship is represented by a simple edge from the parent node to the child node that ends with an empty circle. An example of a feature diagram with optional features is given in Figure 2. Every instance of Concept$_0$ must have Feature$_0$ and Feature$_1$ and every instance of Feature$_1$ may have Feature$_2$ and Feature$_3$.

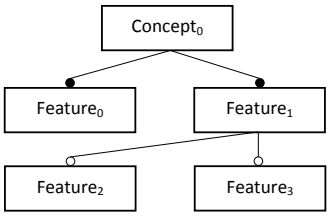

Figure 2: An example of a feature diagram with optional features.

3. The *or*-relationship implies that at least one of the child nodes is included in the description of a concept instance if and only if the parent node is included. The nodes of a set of or-features are connected by a filled arc. An example of a set of or-features in a feature diagram is given in Figure 3. Note that an or-feature is also optional or mandatory. If one feature of a set of or-features is optional, all features of the set can be replaced by optional features. This transformation is called *normalization*.

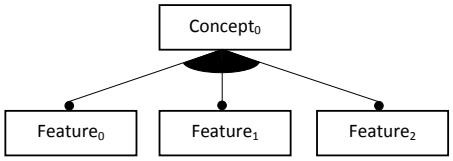

Figure 3: An example of a feature diagram with or-features.

4. The *alternative* relationship (xor-relationship) implies that one of the child nodes must be included in the description of a concept instance if and only if the parent node is included. The nodes of a set of alternative features are connected by an empty arc. An example of a set of alternative features in a feature diagram is given in Figure 4. Note that an alternative feature is also optional or mandatory. If one feature of a set of alternative features is optional, all features of the set can be replaced by optional alternative features. This transformation is also called normalization.

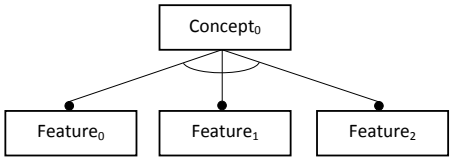

Figure 4: An example of a feature diagram with alternative features.

Feature diagrams can also contain cross-graph constraints. The most common constraints are the requires and excludes constraints. If in a feature diagram a feature requires another feature, than that feature can only be part of an instance of the feature diagram if the other feature is included in the instance. If a feature excludes another feature in a feature diagram, than that feature cannot be part of any instance of the feature diagram that includes the other feature. Such constraints are normally just added to the diagram as text. One can of course imagine other less trivial constraints. Therefore a special constraint language can be useful.

2.2.2. Cardinalities and Extended Feature Models

Some authors[3] propose to add cardinalities to feature models. There main motivation was driven by practical applications (Czarnecki et al. (2002)) and conceptual completeness. Czarnecki and Eisenecker (2000) strongly recommend not to use cardinalities in feature models because the only semantics of an arc is whether to assert a feature or not. Annotating an arc with cardinality four only implies to assert the feature four times. For example asserting the sentence "a car has a wheel" four times still means "a car has a weal". To make cardinalities useful, one should add some label to the arc indicating the "part of" relation, though it is better not to do this to avoid cluttering feature diagrams with structural information.

Other authors[4] propose to extend feature models with additional informa-

[3]Czarnecki et al. (2005); Riebisch et al. (2002)

[4]Batory (2005); Czarnecki et al. (2005); Benavides et al. (2005a); Czarnecki and et al. (2005); Kang et al. (1998); Benavides et al. (2005b); Batory et al. (2006); Streitferdt et al. (2003)

tion called feature attributes. FODA already considered this and introduced relations between features and feature attributes. These types of feature models are called extended feature models. In the next sections we will only consider the basic feature models.

3. A Feature Modelling Tool

3.1. Introduction to AToM^3

AToM^3 is a tool for multi-formalism and meta-modelling. In this paper only a small part of the functionality of the tool is used. A complete overview of the capabilities of AToM^3 is given by De Lara et al. (2002). The overall picture of AToM^3 is that everything is designed as a model. Only the used functionality will now be presented.

AToM^3 allows the user to create a visual and concrete representation of meta-models. From such a representation of a meta-model it is possible to auto generate a toolbar from which visual and concrete representations of models can be designed that comply to the meta-model. The layout of this toolbar can also be modelled since it is also a model.

3.2. The AToM^3 Meta-Model for Feature Models

In this paper a meta-model for feature models is presented. It was designed using the meta-model CD_ClassDiagramsV3: a meta-model which allows the user to design other meta-models as class diagrams. The designed meta-models may consist of classes and associations and there can be hierarchies among classes. The complete visual representation of the meta-model for feature models is given in Figure 5.

A feature is modelled as a concept with a parent. This makes it easier for a user to design models. Besides their name (which is required to start with a capital letter) features and concepts can contain additional information: a semantic description, a rationale, clients and stakeholders, exemplar systems and a priority.

Since AToM^3 doesn't allow direct relations between connections, the alternative set and or set is represented by a separate entity. Making a decoration of an edge stick to the end of that edge is quite infeasible in AToM^3. Therefore the decoration that indicates if a feature is optional or mandatory is also represented by a separate entity called a port.

One can view these modifications to the original representation of a feature diagram as extensions because the entities give the designer the choice in representation. For example if two parent features have the same child feature one can choose to either redraw an entire new connection, or reuse part of the already drawn connections. This not only makes the representation clearer, it also can significantly shorten the time required to design feature models. Another modification to the original representation is that the ports can be moved from the top centre of the rectangle representing a feature. Once a port is connected to a feature, it will automatically move to the top centre of the rectangle of

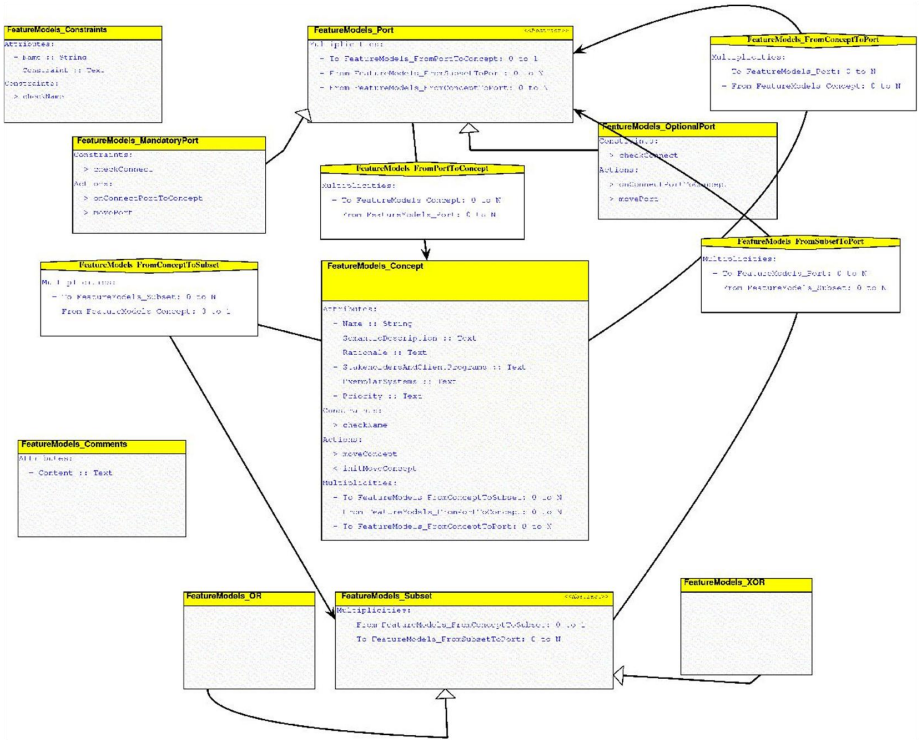

Figure 5: The complete visual representation of the meta-model for feature models that complies to the meta-model CD_ClassDiagramsV3.

that feature. When you want to add another port to the same feature, you can move the first port and it will automatically stay at the top of the rectangle. A feature can thus be optional to one parent, and mandatory to another. It is possible to draw more general graphs instead of just trees.

The meta-model also includes necessary constraints. For example a feature can only be connected to the same parent once. These constraints force the user to design legal feature models. It is impossible to draw an illegal feature model.

To create cross-graph constraints there exists an entity Constraint in which the user can add constraints. These constraints also have a name which is required to start with a capital letter. One could argue that it would be nicer to create a visual representation of constraints, for instance create special entities (for example denoting *and*, *or*, *requires*, *excludes*, or brackets) and connect them to each other. For simple cases this is feasible, but if a feature or concept has to be in two separate constraints a new reference is required. For only a few relatively easy constraints containing the same feature or concept, the feature diagram will become very unclear. The AToM^3 drawing space is also limited so the designer would quite soon run out of space. So to eliminate all these problems a simple textual modelling language was used. To avoid the work of

creating a compiler to map this language to Alloy[5] (see subsection 4.1), Alloy itself was used.

As a final touch the AToM^3 meta-model allows the creation of comments. One might think this is a standard option in every formalism, but most of them do not provide the support for comments. To give users better support for documenting there models, this ability is thus provided.

3.3. Designing Feature Models with the Tool

Now that the formalism is complete, it is very easy to design feature models. One must load the formalism using AToM^3 and then the designing can begin. An example that illustrates how to design a feature model is given in Figure 6. Note that the buttons are edited to make it more clear to the user what he can use. An example of a feature diagram of a simple traffic model is given in Figure 7.

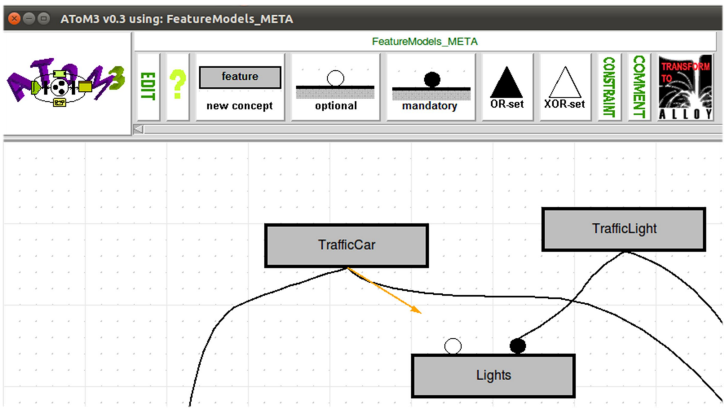

Figure 6: An example of how to use the AToM^3 meta-model of feature models by using the toolbar.

3.4. Facts and Remarks

The formalism provides an interface that is very intuitive to use. People with no experience in programming or modelling can use it without any problems. Everybody can quickly and easily draw nice feature diagrams. There are however two caveats. When creating a complex feature diagram, one quickly runs out of space. In AToM^3 the drawing space is limited so this becomes a problem. The other problem is that the constraint language is not that easy to use. One must also note that although features and concepts can contain extra information, this information is not used during the analysis from section

[5] Jackson (2006)

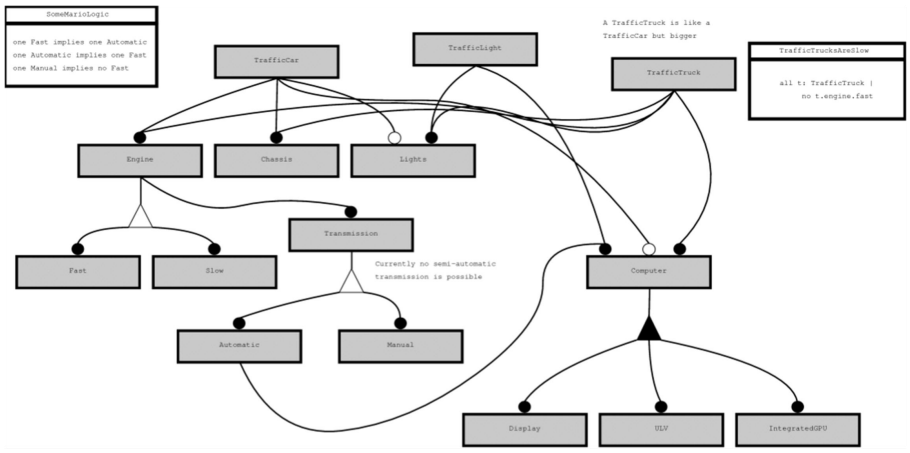

Figure 7: The complete feature model of a traffic model designed using the AToM^3 meta-model.

4. Certain information might even conflict with the results of the analysis. So when performing analysis on a feature model one must be careful. Possible solutions are presented in section 6.

4. Formal Analysis of Feature Models

There has already been a lot of work[6] on formal analysis of feature models. Even in the original FODA report automated analysis was already identified as a critical task. However there isn't a consensus about which operations should be included in this task. A nice overview of proposed operations is given by Benavides et al. (2006). A possible operation could be determining the satisfiability of the feature model. A feature model is satisfiable if at least one product is represented by the feature model. Other possible operations are finding a product, counting all products, obtaining all products, calculating variability and commonality, detecting dead features, and so on.

Instead of creating a new tool that can perform formal analysis on feature models, an existing and popular tool for formal analysis is used, called the Alloy Analyzer from Jackson (2002). A feature model that is represented in the AToM^3 meta-model of section 3 will be transformed into an Alloy model using an automated transformation process. As mentioned in the introduction this approach is similar to the work of Anastasakis et al. (2010) where they propose a method to transform models from the Unified Modelling Language (UML) to Alloy.

[6]Batory et al. (2006); Czarnecki and et al. (2005); Deursen and Klint (2002); Mannion (2002); von der Massen and Lichter (2003); Wang et al. (2005); Zhang et al. (2004)

4.1. Transformation to Alloy

4.1.1. Introduction to Alloy

As mentioned in section 3 Alloy is a textual modelling language. Furthermore it is based on first-order relational logic. A model in Alloy typically contains a number of signatures, fields, facts, asserts and predicates.

A signature can be compared to a description of a class without any member functions. It contains a set of fields, but these field are treated as relations to other signatures. These relations are interpreted as sets of tuples of atoms. Atoms are the smallest possible entities in Alloy, for instance an empty signature. The parental relation type of a field can be specified using quantifiers: all, some, no, one, lone. These quantifiers will be used extensively. The semantics of the quantifiers will now be explained.

some X
There is at least one entity of signature X.

no X
There are no entities of signature X.

one X
There is exactly one entity of signature X. If there is no quantifier specified, the one quantifier is used.

lone X
There are either zero or one entities of signature X.

all x: X | formula
Every entity x of signature X satisfies the formula. If there are no entities in x then this statement is trivially true.

some x: X | formula
One or more entities x of signature X satisfy the formula. If there are no entities in x then this statement is also trivially false.

no x: X | formula
Exactly zero entities x of signature X satisfy the formula.

one x: X | formula
Exactly one entity x of signature X satisfies the formula.

lone x: X | formula
Either zero or one entities x of signature X satisfies the formula.

A fact is a statement that defines a constraint on the entities of the signatures. A predicate is a constraint that is parameterised and it can be included in other predicates or facts.

The Alloy Analyzer provides analysis by searching for instances of a model written in Alloy. Not only can it find instances, it can also check if assertions are satisfied or violated. Both capabilities are achieved by an automated translation

of the model into a Boolean expression. The Boolean expression is then analysed by Satifiability (SAT) solvers.

The user can specify a scope on the model elements to bound the domain. The analysis is then performed within that scope. Jackson (2006) provide more information about analysis within scopes. More information about Alloy is given by Jackson (2006) though the previous description of Alloy suffices to understand the rest of the paper.

To apply the transformation to an Alloy model, a button was added to the formalism (see Figure 6). Since every drawn feature diagram is legal, the transformation will terminate correctly. The user will only need to input a pathname to where he wants to save the transformed model. No other information is needed.

4.1.2. Rules to Transform the AToM^3 Model to Alloy

Since the domain of Alloy is different than the domain implied by the metamodel introduced in section 3, a set of rules have to be constructed. These rules will form the transformation process. It is of course possible to create another set of rules that construct a different model out of the same input model. Depending on the goal of the transformation a different set can be choosen. For this work the only goal was to create an Alloy model to perform some useful analysis. Therefore the chosen set of rules was the set that constructs an easy to read Alloy model that preserves almost all information of the input model without being too much work to implement.

First a model name needs to be created because this is required in an Alloy model. When the original feature diagram has a name, the Alloy model will have the same name. When there is no name, a model name will be constructed from the filename of the model. Then additional information about the model is added in comment: the author, the date created and the description. When the original model doesn't have an author or description, this information will be omitted.

Now for the hard work. Features and concepts are transformed into signatures with the same name. Their fields indicate the relations to the sub-features. The name of a field will be the lowercase version of the name of the feature. If a mandatory sub-feature is not in a set it will get the default relation type one. Every other sub-feature will get the relation type lone. If an alternative set contains an optional feature the relation type for the set is lone, otherwise the relation type for the alternative set is one. This extra refinement is implemented as fact of the signature. If an or set doesn't contain an optional feature the relation type is some, else nothing more has to be specified by using the normalization property from subsection 2.2.

The information of whether a signature represents a feature or a concept can be implied by the complete set of signatures. Although Alloy doesn't need an ordering on the declaration of signatures, the signatures are ordered on the number of fields. Since signatures with lots of fields will probably contain fields with a relation to signatures with less fields (this is when modelling in a structured way), 'emptier' signatures will be declared first. Thus the readability of

the model is improved.

Since constraints in the feature diagram are modelled in Alloy, these are quite easily put in the Alloy model. They are transformed into facts with the same name as the constraint, and as constraint just the same text as in the feature diagram.

4.1.3. Added Analysis Support

For every concept a predicate is added which allows the user to create a representation of the feature model using the Alloy Analyzer by using the run functionality. This will make the Alloy Analyzer try to find instances that comply to the predicate. An additional predicate NoOrphanFeatures is also added which will make sure no entities of signatures are considered that don't have a parent. For example an engine that is not part of any TrafficCar or TrafficTruck will not be part of the domain. Since this is not a requirement for every analyzer of feature models and might even be a useful side-effect of the transformation to Alloy, this constraint is put in a predicate. If the analyzer wants to use the predicate as a constraint on the domain, he will need to put it in a fact.

To show how to perform some operations of analysis on the feature model, extra assertions are automatically added to the transformed model.

First the operation of determining the satisfiability of the feature model will be explained. When an assertion is created in Alloy, the Alloy Analyzer will try to violate it in every possible way. So the method used to see if a feature model contains a product is to say to the Alloy Analyzer 'you can NOT create a product'. The Alloy Analyzer will accept the challenge and try to find a counter example. If it finds a counter example, it means that the Alloy Analyzer can find a product and thus the feature model is satisfiable. If it cannot find a counter example, it means that the feature model will most likely contain a concept that does not describe any products. Not only answers this method the satisfiability question, it also provides a nice way to find products.

Dead features are dealt with in the same way. An assertion is made stating that there can NOT be a product containing that feature. The Alloy Analyzer will again accept the challenge and try to find a counter example. If it finds a counter example, it means that the feature is part of some particular product or products. If it doesn't find any products containing that feature it is most likely that it is a dead feature. To create every possible path of every concept to the feature investigated takes quite some time. Therefore an optimization is done using the NoOrphanFeatures predicate. This method also provides an easy way to find products containing a certain feature.

4.1.4. Characteristics of the Model Transformation Language

In this subsection the transformation characteristics from the transformation to Alloy will be briefly discussed based on the feature model from Czarnecki and Helsen (2006). Figure 8 represents the feature model for the characteristics.

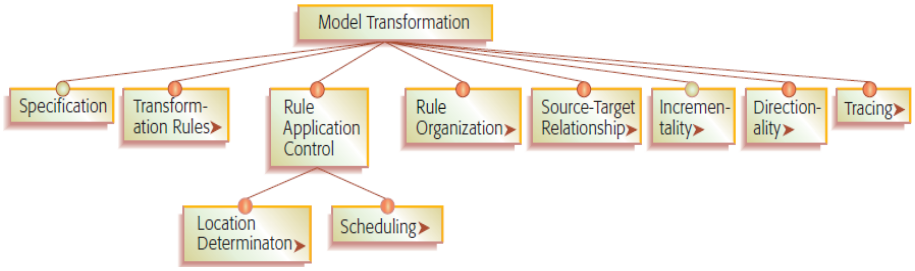

Figure 8: Feature model of the characteristics of the Model Transformation Language.

Specification
The transformation has one precondition: that the input is legal with regard to the AToM^3 formalism. But since one cannot drawn illegal models there is no need for such a precondition. The postcondition is that the transformation is terminated and succeeded, which of course cannot be checked in execution since it would imply rewriting almost all of the logic.

Transformation rules
The transformation units are defined as methods from the class TransformerFM2Alloy. The rules are already discussed in section 4.1.2.

Rule application control
The rule application strategy is deterministic. The order is defined by the calls made from the first method called till the last method called.

Rule organization
The rules are coded to be very reusable. There was no need to make the methods more modular then they currently are.

Source-target relationship
The target model is a new Alloy model. The source model is the AToM^3 model that complies to the feature model formalism from section 3.

Incrementality
Currently there is no support for an incremental transformation. If the original feature model is updated, a new transformation to Alloy is required.

Directionality
Currently the transformation is unidirectional: from a feature model to an Alloy model. It is possible to make it multidirectional.

Tracing
There is no dedicated support for tracing. This would also be hard (even quite impossible) to implement because Alloy wasn't made to support tracing.

4.1.5. Analysis of a Traffic Model

As an example the feature model of Figure 7 is transformed into an Alloy model. This is done simply by clicking on the TransformToAlloy button. On the following three pages the transformed model is presented.

```
/*
        Author:         Thomas De Vylder

        Date:           2011-04-12

        Model:          TrafficModel

        Description:
        This is a simple traffic model. It contains a TrafficCar,
        a TrafficLight and a TrafficTruck modelled with feature models.
        It is an easy example to show most of the functionality.
*/

module TrafficModel

/*------------------------------The Feature Model------------------------------*/
//------------------------------Concepts and Features------------------------------
sig Chassis { }

sig Fast { }

sig Lights { }

sig Manual { }

sig Slow { }

sig Display { }

sig ULV { }

sig IntegratedGPU { }

sig Automatic {
        computer: Computer
}

sig Transmission {
        automatic: lone Automatic,
        manual: lone Manual
}
{       one ( automatic + manual )
}

sig TrafficLight {
        lights: Lights,
        computer: Computer
}

sig Engine {
        fast: lone Fast,
        slow: lone Slow,
        transmission: Transmission
}
{       one ( fast + slow )
}

sig Computer {
        display: lone Display,
        integratedGPU: lone IntegratedGPU,
        uLV: lone ULV
}
{       some ( display + integratedGPU + uLV )
}
```

```
sig TrafficCar {
	engine: Engine,
	chassis: Chassis,
	lights: lone Lights,
	computer: lone Computer
}

sig TrafficTruck {
	computer: Computer,
	chassis: Chassis,
	lights: Lights,
	engine: Engine
}

//------------------------------Constraints------------------------------
fact SomeMarioLogic {
	one Fast implies one Automatic
	one Automatic implies one Fast
	one Manual implies no Fast
}

fact TrafficTrucksAreSlow {
	all t: TrafficTruck | no t.engine.fast
}

/*------------------------------Extra Assertions------------------------------*/
-- determining the existence of the dead features

-- assertion should better be invalid
assert EngineIsADeadFeature {  NoOrphanFeatures implies no Engine }
check EngineIsADeadFeature

-- assertion should better be invalid
assert ChassisIsADeadFeature {  NoOrphanFeatures implies no Chassis }
check ChassisIsADeadFeature

-- assertion should better be invalid
assert FastIsADeadFeature {  NoOrphanFeatures implies no Fast }
check FastIsADeadFeature

-- assertion should better be invalid
assert LightsIsADeadFeature {  NoOrphanFeatures implies no Lights }
check LightsIsADeadFeature

-- assertion should better be invalid
assert ManualIsADeadFeature {  NoOrphanFeatures implies no Manual }
check ManualIsADeadFeature

-- assertion should better be invalid
assert SlowIsADeadFeature {  NoOrphanFeatures implies no Slow }
check SlowIsADeadFeature

-- assertion should better be invalid
assert AutomaticIsADeadFeature {  NoOrphanFeatures implies no Automatic }
check AutomaticIsADeadFeature

-- assertion should better be invalid
assert TransmissionIsADeadFeature {  NoOrphanFeatures implies no Transmission }
check TransmissionIsADeadFeature

-- assertion should better be invalid
assert ComputerIsADeadFeature {  NoOrphanFeatures implies no Computer }
check ComputerIsADeadFeature

-- assertion should better be invalid
assert DisplayIsADeadFeature {  NoOrphanFeatures implies no Display }
check DisplayIsADeadFeature

-- assertion should better be invalid
```

```
assert ULVIsADeadFeature {  NoOrphanFeatures implies no ULV }
check ULVIsADeadFeature

-- assertion should better be invalid
assert IntegratedGPUIsADeadFeature {  NoOrphanFeatures implies no IntegratedGPU }
check IntegratedGPUIsADeadFeature


-- determining the satisfiability of the feature model

-- assertion should better invalid
assert TrafficTruckIsNotSatisfiable { no TrafficTruck }
check TrafficTruckIsNotSatisfiable

-- assertion should better invalid
assert TrafficCarIsNotSatisfiable { no TrafficCar }
check TrafficCarIsNotSatisfiable

-- assertion should better invalid
assert TrafficLightIsNotSatisfiable { no TrafficLight }
check TrafficLightIsNotSatisfiable

/*------------------------------Extra Predicates------------------------------*/
pred NoOrphanFeatures {
        all feature: Engine | (some parent: TrafficTruck | parent.engine = feature) or
                (some parent: TrafficCar | parent.engine = feature)
        all feature: Chassis | (some parent: TrafficTruck | parent.chassis = feature) or
                (some parent: TrafficCar | parent.chassis = feature)
        all feature: Fast | (some parent: Engine | parent.fast = feature)
        all feature: Lights | (some parent: TrafficTruck | parent.lights = feature) or
                (some parent: TrafficCar | parent.lights = feature) or
                (some parent: TrafficLight | parent.lights = feature)
        all feature: Manual | (some parent: Transmission | parent.manual = feature)
        all feature: Slow | (some parent: Engine | parent.slow = feature)
        all feature: Automatic | (some parent: Transmission | parent.automatic = feature)
        all feature: Transmission | (some parent: Engine | parent.transmission = feature)
        all feature: Computer | (some parent: TrafficTruck | parent.computer = feature) or
                (some parent: Automatic | parent.computer = feature) or
                (some parent: TrafficCar | parent.computer = feature) or
                (some parent: TrafficLight | parent.computer = feature)
        all feature: Display | (some parent: Computer | parent.display = feature)
        all feature: ULV | (some parent: Computer | parent.uLV = feature)
        all feature: IntegratedGPU | (some parent: Computer | parent.integratedGPU = feature)
}

pred ShowOnlyConcepts { NoOrphanFeatures }
/*Note:  OnlyConcepts is true for most
cases, but in case it's a general graph the
pred NoOrphanFeatures is not sufficient to
show only concepts (some groups of features
could not be connected to a concept)*/
run ShowOnlyConcepts

pred ShowOneOfEachConcept {
        NoOrphanFeatures
        one TrafficCar
        one TrafficTruck
        one TrafficLight
}
run ShowOneOfEachConcept
```

Using the Alloy Analyzer one can easily perform analysis on this model. Using the auto generated predicates and assertions products can easily be found so the feature model is satisfiable. An example of a product with one instance for every concept is given in Figure 9. This figure was made by running the generated predicate ShowOneOfEachConcept.

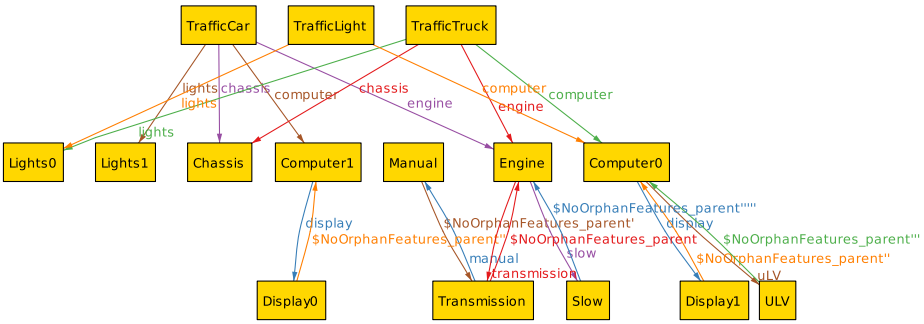

Figure 9: A representation of one instance per concept of the traffic model found by the Alloy Analyzer.

Of course other analysis can be performed. Using the auto generated assertions one can show an example concept for each feature, so there are no dead features. The constraint SomeMarioLogic is a bit confusing. One might think that this constraint holds for every individual product, but the Alloy Analyzer will create a world (called univ) in which multiple instances can exist. Therefore the constraint will hold for every set of products. If the intent was that the constraint should hold on every individual product, doing specialized analysis becomes harder. For example an assertion to show that every TrafficTruck has a manual transmission will look like the following.

```
NoOrphanFeatures and no TrafficCar and one TrafficTruck implies
                    all t:TrafficTruck | some t.engine.transmission.manual
```

So if the intent is that the constraint SomeMarioLogic should hold for every instance of a product, this will show that Automatic is a dead feature for concept TrafficTruck. This simple example illustrates that writing constraints for feature models in Alloy can be quite toilful.

4.2. Facts and Remarks

One can immediately see that there is a lot of potential. With just two simple assertions that are automatically generated by the tool, four operations of analysis can be performed. Since this research only had a limited amount of time, no further investigation has been performed, though some interesting remarks can already be made.

1. Alloy is based on first-ordered logic. Feature modelling tries to make life easier by abstracting away the details. It provides a very simple technique to describe programs and products. People with no mathematical

and programming background can understand and perform feature modelling. When using Alloy to perform analysis, the user will need a good understanding of the first-ordered logic, so the basic knowledge a person must have to perform feature modelling becomes larger, and thus Feature modelling becomes less attractive to the general population.

2. The Alloy Analyzer provides a lot of analysis support. Though it comes with a few pitfalls. For instance the performance of the Alloy Analyzer is decreased by the extensive use of integer types in a model. Furthermore the depth and value of the analysis will be limited to the functionality that is presented by the Alloy Analyzer.
3. The domain represented by the transformed model might be too precise. Though the transformed model is still on a very abstract level, information must be added. An nice example of such information is the decision whether or not two instances of a concept must have a separate instance for each feature. Another exemplar decision is whether or not an instance of a feature can exist without belonging to any concept instance. Such information must be added to the Alloy model in order to be able to perform analysis.
4. The domain represented by the transformed model might be too imprecise. Some information is lost when transforming the model. For instance the priority of a concept or feature. One might add this information as comments in the Alloy model but even then the information is not used by the Analyzer and thus not considered when performing analysis. So the key to remember is that one must be very careful when removing information during the transformation to another representation.
5. The two previous points can be seen as a warning to the analyzer. When performing analysis on the transformed model, one must be very careful when mapping the gained information back to the original model, since the analysis is actually done on a different model.

5. Conclusion

In this paper a survey of feature modelling was presented. Feature modelling provides an easy way to describe families of products. It is vital that a feature modelling tool allows formal analysis. A tool to design feature models is presented that uses AToM^3. Besides designing feature models it also allows a transformation to the Alloy language so that the Alloy Analyzer can be used to perform analysis. Already some basic operations of analysis are provided: determining the satisfiability, finding a product, determining dead features and finding products containing certain features. The contribution of this work is limited to the capabilities of the used tools: AToM^3 and the Alloy Analyzer. Although the set of capabilities of the presented tool are quite small, it can be useful for small to medium sized product families. Moreover the insight gained from developing and working with it can be a good starting point when developing a full fledged feature modelling tool.

6. Future Work

It can be very interesting to extend the capabilities of the tool. Since it is programmed in a structured way, new functionality can be easily added. There are countless possibilities to extend this work. The eight most important ones are listed below.

1. AToM^3 only provides a limited amount of space to design models so it is very interesting to add support for collapsing features and concepts. This way large and complex feature models can still be easily representable. It might be interesting to then add functionality to import and export collapsed features in order to encourage reuse of models.
2. Currently the constraints are modelled in the textual modelling language of Alloy due to the limited time of the research. Providing a more suitable language (textual or visual) for feature models can be a great improvement to the tool. Similar work is done by Taentzer (2002) who introduced a visualization of the Object Constraint Language (OCL). It might also be useful to look at the high-performance reflective language Maude of Clavel et al. (1993).
3. There are already a lot of analysis operations provided. Adding more operations will only improve the depth and value of the analysis. It might also be interesting to research the limits of the analysis performed in Alloy. For instance counting the number of products in a family is not that trivial and in large complex families this will probably take too long.
4. For now every analysis operation currently available is automatically generated when transforming the model. It might be handy to have a menu in which the user can select which operations need to be generated. It can even be more interesting to perform the analysis of Alloy in the background of AToM^3. This way the transformed model is not seen by the user. The user doesn't even have to understand Alloy to perform the analysis.
5. The transformation to Alloy does not consider comments and other metadata (descriptions, clients and stakeholders, etc.). This information can be of great importance during the analysis so providing rules to transform this information is very interesting.
6. Functionality could be added to the formalism so that a feature diagram can be normalized (see section 3). This does not require a lot of work and can be very useful.
7. Currently a concept or feature can only have one edge to a feature. In most cases this is required, but in some cases it might be interesting to provide this functionality but then a lot of adjustments will be necessary.
8. The transformation process currently is unidirectional, does not support incremental changes and does not allow tracing. Adding support for these three features would make the current implementation even more useful. For example if one would see a problem by inspecting the Alloy model, one could change the Alloy model directly and then the feature model could

be updated automatically. However for this research these features were not required.

All this functionality will be of more value when feature modelling is more widely used, since one learns best from practical examples. This will also encourage the SPL community to make standards. It would for instance be very handy if a common constraint language would be created in which designers can very easily model cross-graph constraints.

References

Anastasakis, K., Bordbar, B., Georg, G., Ray, I., Jan. 2010. On challenges of model transformation from uml to alloy. Software & Systems Modeling 9 (1), 69–86.

Batory, D., 2005. Feature models, grammars, and propositional formulas. In: Software Product Lines Conference, LNCS 3714. pp. 7–20.

Batory, D., Benavides, D., Ruiz-Cortes, A., December 2006. Automated analysis of feature models: challenges ahead. Commun. ACM 49, 45–47.

Benavides, D., Ruiz-Corts, A., Trinidad, P., Segura., S., 2006. A survey on the automated analyses of feature models. In: Jornadas de Ingeniera del Software y Bases de Datos (JISBD).

Benavides, D., Trinidad, P., Ruiz-Corts, A., 2005a. Automated reasoning on feature models. In: LNCS, Advanced information Systems Engineering: 17th International Conference, CAISE 2005. Springer.

Benavides, D., Trinidad, P., Ruiz-Corts, A., 2005b. Using constraint programming to reason on feature models. In: The Seventeenth International Conference on Software Engineering and Knowledge Engineering.

Clavel, M., Durn, F., Eker, S., 1993. The Maude System.
URL `http://maude.cs.uiuc.edu/`

Clements, P., Northrop, L. M., 2002. Software Product Lines: Practices and Patterns. Addison-Wesley.

Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U. W., 2002. Generative programming for embedded software: An industrial experience report. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering. GPCE '02. Springer-Verlag, London, UK, pp. 156–172.

Czarnecki, K., Eisenecker, U. W., 2000. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Czarnecki, K., et al., 2005. Cardinality-based feature modeling and constraints: A progress report.

Czarnecki, K., Helsen, S., Jul. 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45 (3), 621–645.

Czarnecki, K., Helsen, S., Ulrich, E., 01/2005 2005. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10, 7 – 29.

De Lara, J., Vangheluwe, H., Posse, E., A. Vasudeva Murthy, I., Provost, M., Liang, W., 2002. AToM^3 A Tool for Multi-formalism and Meta-Modelling. URL `http://atom3.cs.mcgill.ca/index_html`

Deursen, A., Klint, P., 2002. Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10, 2002.

Jackson, D., 2002. Alloy Analyzer website. URL `http://alloy.mit.edu/`

Jackson, D., 2006. Software Abstractions: Logic, Language, and Analysis. The MIT Press.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, A. S., November 1990. Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute.

Kang, K. C., Kim, S., Lee, J., Kim, K., Kim, G. J., Shin, E., 1998. Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168.

Mannion, M., 2002. Using first-order logic for product line model validation. In: Proceedings of the Second International Conference on Software Product Lines. SPLC 2. Springer-Verlag, London, UK, UK, pp. 176–187.

Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I., 2002. Extending feature diagrams with uml multiplicities. In: 6th World Conference on Integrated Design & Process Technology (IDPT2002). Pasadena, California.

Smith, E. E., Medin, D., 1981. Categories and Concepts. Harvard University Press, Cambridge, MA.

Streitferdt, D., Riebisch, M., Philippow, I., 2003. Details of formalized relations in feature models using ocl. In: Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS 2003), Huntsville, USA. IEEE Computer Society. pp. 45–54.

Taentzer, G., 2002. A visualisation of OCL. URL `http://tfs.cs.tu-berlin.de/vocl/`

von der Massen, T., Lichter, H., 2003. RequiLine: A Requirements Engineering Tool for Software Product Lines. In: van der Linden, F. (Ed.), Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5). LNCS 3014. Springer Verlag, Siena, Italy.

Wang, H., Li, Y. F., Sun, J., Zhang, H., Pan, J., 2005. A semantic web approach to feature modeling and verification. In: In Workshop on Semantic Web Enabled Software Engineering (SWESE05).

Withey, J., November 1996. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Zhang, W., Zhao, H., Mei, H., 2004. A propositional logic-based method for verification of feature models. In: Davies, J., Schulte, W., Barnett, M. (Eds.), Formal Methods and Software Engineering. Vol. 3308 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Berlin, Heidelberg, Ch. 16, pp. 115–130.