XTraQue: Traceability for Product Line Systems

Waraporn Jirapanthong[1]
Faculty of Information Technology
Dhurakij Pundit University
110/1-4 Prachachuen Road
Laksi, Bangkok 10210, Thailand
waraporn@it.dpu.ac.th

Andrea Zisman
Department of Computing
City University
Northampton Square,
London EC1V 0HB, UK
a.zisman@soi.city.ac.uk

Abstract

Product line engineering has been increasingly used to support the development and deployment of software systems that share a common set of features and are developed based on the reuse of core assets. The large number and heterogeneity of documents generated during the development of product line systems may cause difficulties to identify common and variable aspects among applications, and to reuse core assets that are available under the product line. In this paper, we present a traceability approach for product line systems. Traceability has been recognised as an important task in software system development. Traceability relations can improve the quality of the product being developed and reduce development time and cost. We present a rule-based approach to support automatic generation of traceability relations between feature-based object-oriented documents. We define a traceability reference model with nine different types of traceability relations for eight types of documents. The traceability rules used in our work are classified into two groups namely (a) direct rules, which support the creation of traceability relations that do not depend on the existence of other relations, and (b) indirect rules, which require the existence of previously generated relations. The documents are represented in XML and the rules are represented in an extension of XQuery. A prototype tool called XTraQue has been implemented. This tool, together with a mobile phone product line case study, has been used to demonstrate and evaluate our work in various experiments. The results of these experiments are encouraging and comparable with other approaches that support automatic generation of traceability relations.

Keywords: Software traceability, product line, traceability relations, traceability rules, feature-based object-oriented documents

1. Introduction

Product line systems in which software systems share a common set of features and are developed based on the reuse of core assets have been recognised as an important paradigm for software systems engineering. Recently, a large number of software systems are being developed and deployed in this way in order to reduce cost, effort, and time during system development. Various methodologies and approaches have been

[1] This work has been partially supported by Dhurakijpundit University, Thailand.

proposed to support the development of software systems based on product line development. Examples of these methodologies and approaches are FeatuRSEB[29], FAST [79], FORM [35], FODA [24], PuLSE [6], KobrA [4], and the work in [9], [10], and [29].

The above methodologies and approaches are also known as *domain engineering* approaches and emphasise a group of related applications in a domain, instead of single applications. Their main focus is the identification and analysis of commonality and variability principles among applications in a domain in order to engineer reusable and adaptable components and, therefore, support product line development. However, although the support for identifying and analysing common and variable aspects among applications and the engineering of reusable and adaptable components are important for product line development, they are not easy tasks. This is mainly due to the large number and heterogeneity of documents generated during the development of product line systems. Other difficulties are concerned with the (a) necessity of having a basic understanding of the variability consequences during the different development phases of software products by all parties involved [74][75][68], (b) necessity of establishing relationships between product members and product line artefacts, and relationships between product members artefacts [5][48], (c) poor support for capturing, designing, and representing requirements at the level of product line and the level of specific product members [44][23], (d) poor support for handling complex relations among product members [5][48], and (e) poor support for maintaining information about the development process [47].

In contrast, requirements traceability has been recognized as an important activity in software system development [1][16][21][27][45][53][60][66]. In general, traceability relations can improve the quality of a product being developed, and reduce the time and cost of development. In particular, traceability relations can support evolution of software systems, reuse of parts of a system by comparing components of new and existing systems, validation that a system meets its requirements, understanding of the rationale for certain design and implementation decisions, and analysis of the implications of changes in the system.

Support for traceability in software engineering environments and tools are not always adequate [60][66]. Some existing approaches assume that traceability relations should be established manually [19][33][63][65], which is error-prone, difficult, time consuming, expensive, complex, and limited on expressiveness given the fact that the relations are mainly hyperlinks without any semantic meaning. Therefore, despite its importance, traceability is rarely established. In order to alleviate this problem, more recently, other approaches have been proposed to support semi- or fully-automatic generation of traceability relations [1][16][21][46][54][57][59][66][72]. However, the majority of these approaches do not support generation of traceability relations for various types of documents produced during product line engineering.

As affirmed in [5][7][36][48][64][76], traceability relations can be used to mitigate the difficulties associated with product line engineering. More specifically, traceability relations can assist with the (i) identification of common and variable functionalities in product members, (ii) reduction of inconsistencies

between product members, (iii) reuse of core assets that are available in a product line system, (iv) maintenance of historical information of the development process, and (v) establishment of relationships between product line and product members specification documents. However, the majority of the approaches concerning traceability for product line systems focus on traceability metamodels and do not provide ways of generating traceability relations automatically.

We present a rule-based approach to allow automatic generation of traceability relations between elements of documents created during the development of product line systems. We are interested in documents generated in feature-based object-oriented methodologies and have chosen to use an extension of the FORM [35] methodology. A feature-based approach supports domain analysis and design, while an object-oriented approach assists with the development of various product members. We present a traceability reference model with nine different types of traceability relations for eight types of documents. The different types of traceability relations are satisfiability, dependency, overlaps, evolution, implements, refinement, containment, similar, and different. The documents include feature, subsystem, process, and module models representing product line information, and use cases, class, statechart, and sequence diagrams representing product members' information. In our approach, the documents are represented in XML and the different types of traceability relations are identified by using traceability rules expressed in an extension of XQuery [82]. The textual sentences of the XML documents are annotated with part-of-speech assignments indicating the grammatical roles of the various words in the sentence. These grammatical roles are used to assist with the matching of textual terms in the documents. The traceability rules are classified as *direct rules*, i.e., rules that support the creation of traceability relations that do not depend on the existence of other relations; and *indirect rules*, i.e., rules that require the existence of previously generated relations. In both types of rules, when a matching expected by a rule is found, a traceability relation is created between parts of the documents being compared by the rule.

A prototype tool called XTraQue[2] has been implemented. This tool allows for the generation of traceability relations by interpreting traceability rules. It also offers support for creating new traceability rules and translating documents into XML format. In order to illustrate and evaluate our work, we use a case study from a mobile phone product line system. The case study has been developed based on study, analysis, and discussions of mobile phone domains, and ideas in [50][51]. Examples of this case study are used throughout the paper for illustration. Our work has been evaluated in terms of precision and recall measurements. The results of our evaluation are also presented in the paper.

The remaining of this paper is structured as follows. In section 2, we describe a traceability reference model with the main documents and traceability relation types identified in our work. In section 3, we present our approach, describe traceability rules, and illustrate the work through examples. In section 4, we discuss some implementation issues. In section 5, we describe the case study and present the results of

[2] The name XTraQue stands for documents in XML format (X), traceability platform (Tra), and the use of rules specified in XQuery (Que).

evaluating our work. In section 6, we describe related work. Finally, in section 7, we summarise our approach and discuss directions for future work.

2. Traceability Reference Model

As in [40], we propose the use of a feature-based object-oriented engineering approach to support the development of product line systems. A feature-based approach supports domain analysis and domain design, enhances communication between customers and developers in terms of product features, and assists with the development of product line architecture. On the other hand, an object-oriented approach assists with the development of the various product members in a product line system. We propose to use an extension of the FORM (Feature-Oriented Reuse Method) methodology [35] due to its simplicity, maturity, practicality, and extensibility characteristics.

Our work concentrates on documents generated by the FORM methodology such as feature, subsystem, process, and module models, for the product line level; and object-oriented documents such as use case specifications, class, statechart, and sequence diagrams, for the product members. Table 1 presents a summary of the documents used in our work. As shown in the table, these documents represent information in different phases of product line engineering namely *domain analysis* and *domain design,* and different levels of specialisation in product line engineering namely *product line* and *product member* levels.

Table 1: Documents used in our approach

	Domain Analysis	Domain Design
Product Line Level	Feature model	Subsystem model Process model Module model
Product Member Level	Use Cases	Class diagram Statechart diagram Sequence diagram

In our work, we assume that for each line of software system being developed, there is a single instance of feature and subsystem models, but there may exist various instances of process and module models and various instances of documents in the product member level (i.e., use cases, class, statechart, and sequence diagrams). This assumption is not unrealistic since the product line level represents general characteristics of a group of product members being developed, while the product member level is concerned with the various products in the group. Moreover, for a certain product line, it is possible to have different behaviour for the subsystems represented by different process and module models, and for a certain product member, it is possible to have various ways of using and interacting with the product represented by different use cases, sequence diagrams, and statechart diagrams.

In the next subsections we discuss the various documents used in our work and the different types of traceability relations. In our approach, the documents are represented in XML. The textual sentences in the XML documents are annotated with part-of-speech assignments by using a general purpose grammatical

tagger called CLAWS [14]. This grammatical tagger assumes the British National Corpus [41]; i.e., a collection of samples of written and spoken English language from various sources.

We have chosen XML as a basis for our approach due to several reasons: (a) XML has become the de facto language to support data interchange among heterogeneous tools and applications, (b) the existence of large numbers of applications that use XML to represent information internally or as a standard export format (e.g. Unisys XML exporter for Rational Rose [62], Borland Together [8], ArgoUML [2]), and (c) to allow the use of XQuery [82] as a standard way of expressing traceability rules. Moreover, the OMG promotes the use of XML Metadata Interchange (XMI) [52] to enable interchange of metadata between modelling tools that are based on OMG-UML and metadata repositories. XMI integrates OMG-UML modelling standards with Meta Object Facilities (MOF) and XML-W3C standard. Furthermore, our approach combines feature-oriented and object-oriented documents and, therefore, requires a common representation for these document types. For each document type used in our approach we have created XML Schemas and XML documents for the mobile phone case study. The complete set of XML Schemas and XML documents can be found in [83].

2.1. Document Types

2.1.1 Feature Model

A feature model describes common and variable aspects (features) of a line of applications in a domain. More specifically, it describes the abstraction of domain knowledge from domain experts. In the FORM methodology [35], a feature model is composed of two parts: (a) a graphical hierarchy of features, and (b) a textual specification. An example of the textual specification template proposed by the FORM methodology for *Text Messages* is presented in Figure 1[3].

Feature-name:	Text Messages
Description:	The phone can edit, send, and receive a short text message
Issues and decision:	Text message over mobile phone is a way of communication
Type:	Application capability
Commonality:	Mandatory
Composed-of:	Sending Text Messages, Receiving Text Messages, Editing Text Messages
Composition-rule:	-
Allocated-to-subsystem:	Messaging

Figure 1: Textual template for feature model

[3] Due to lack of space, we do not present in this paper the graphical representations of the documents used in our work.

```
<Feature_Model>
<Feature>
  <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2> </Feature_name>
   <Description>
     <AT0> The </AT0> <NN1> phone </NN1> <VM0> can </VM0> <VVI> edit </VVI> <SC>,</SC>
     <VVI> send </VVI> <SC>,</SC> <CJC> and </CJC> <VVI> receive </VVI> <AT0> a </AT0>
     <AJ0> short </AJ0> <NN1> text </NN1> <NN1> message </NN1> <SC>.</SC>
   </Description>
   <Issue_and_decision>
     <NN1> Text </NN1> <NN1> message </NN1> <II> over </II> <JJ> mobile </JJ> <NN1> phone </NN1>
     <VBZ> is </VBZ> <AT1> a </AT1> <NN1> way </NN1> <IO> of </IO>
     <NN1> communication </NN1>
   </Issue_and_decision>
   <Type>Application capability</Type>
   <Existential>Mandatory</Existential>
   <Relationship Type="composed_of">
    <Rel_feature> <VVG> Sending </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
    </Rel_feature>
    <Rel_feature> <VVG> Receiving </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
    </Rel_feature>
    <Rel_feature> <VVG> Editing </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
    </Rel_feature>
   </Relationship>
   <Allocated_to_subsystem> <NN1> Messaging </NN1> </Allocated_to_subsystem>
  </Feature>
  <Feature>
     <Feature_name> <VVG> Editing </VVG><NN1> Text </NN1><NN2> Messages </NN2>
     </Feature_name>
     <Description>
       <AT0> The </AT0> <NN1> phone </NN1> <VVZ> provides </VVZ> <AT1>an</AT1>
       <NN1> editor </NN1> <TO> to </TO> <VVI> create </VVI> <AT1> a </AT1>
       <JJ> new </JJ> <NN1> text </NN1> <NN1> message </NN1> <SC>.</SC>
       <VVG> Editing </VVG> <AT> the </AT> <NN1> text</NN1> <NN1> message </NN1>
       <VM> can </VM> <VBI> be </VBI> <VDN> done </VDN> <II> in </II> <JJ> different </JJ>
       <NN2> ways </NN2> <II21> such </II21> <II22> as </II22> <NN1> alpha </NN1>
       <NN1> mode </NN1> <CC> and </CC> <JJ> predictive </JJ>  </NN1> mode </NN1>
     </Description>
     <Type>Application capability</Type>
     <Existential>Mandatory</Existential>
     <Allocated_to_subsystem> <NN1> Messaging </NN1> </Allocated_to_subsystem>
     <Composition_rule>
       <VVZ> requires </VVZ> <NN1> text </NN1> <NN1> library </NN1> <NN1> feature </NN1>
     </Composition_rule>
  </Feature> …
</Feature_Model>
```

Figure 2: Extract of XML representation for feature model

The XML representation of a feature model used in our work is based on the textual specification shown in Figure 1. In this representation, a feature model is composed of many features as shown in the extract of Figure 2. Each feature has a name (<Feature_name>); a description in natural language sentences (<Description>); a description of possible issues and decisions that may have been raised during the feature analysis process (<Issue_and_decision>); a type (<Type>); an element <Existential> denoting if the feature is mandatory, optional, or alternative; relationships with other features (element <Relationship> with attribute *type* and associated features represented by element <Rel_feature>); and the name of a subsystem

that may contain the feature (<Allocated_to_subsystem>), if any. As shown in Figure 2, the contents of *Feature_name*, *Description*, *Issue_and_decision*, *Rel_feature*, and *Allocated_to_subsystem* elements are marked-up with part-of-speech XML tags (XML POS-tags) indicating their grammatical role in the sentence. For instance, the word "Text" is marked-up with element <NN1>, denoting that "Text" is a singular common noun; the word "Messages" is marked-up as <NN2>, denoting a plural common noun; the word "edit" is marked-up as <VVI>, denoting an infinite verb. The XML POS-tags are created by using CLAWS tagger [14] and a converter that we have developed, as explained in Subsection 3.1.

2.1.2 Use Cases

We propose to represent functional requirements of product members in natural language as use-cases, based on a variant of the template proposed in [11]. Figure 3 shows an example of a use case for *Sending a Message* from a mobile phone for one of the product members (PM1) of our mobile phone case study represented in our template.

```
<Use_Case  UseCaseID="UC1" System="MobilePhone" Product_Member="PM1" >
 <Title> Sending a Message </Title>
 <Description> The phone is able to send a text message. The user can specify an address of a receiver of the
                message by selecting the address from a list of contacts </Description>
  <Level> User Goal </Level>
  <Preconditions> The user selects the address of the receiver of the message </Preconditions>
  <Postconditions> A confirmation sign is shown on the screen </Postconditions>
  <Primary_actor> The user </Primary_actor>
  <Secondary_actors/>
  <Flow_of_events>
     <Event> The system shows an editor for writing a message. </Event>
     <Event> The user types in the phone number of a receiver or selects the phone number of a
              receiver from a list of contacts. The user can send a text message to multiple receivers by
              selecting multiple mobile phone numbers. </Event>
    <Event> The system displays the phone number(s). </Event>
    <Event> The user types the message. The size of the message is limited by a maximum size.</Event>
    <Event> The system displays the message. </Event>
    <Event>The user confirms sending the message. </Event>
    <Event>The system establishes the connection for sending the message.</Event>
    <Event> If the connection is properly set, the system sends the message and displays a
            confirmation sign. Otherwise, the system displays a faulty sign.</Event>
    <Event> After completion the system undo the connection.</Event>
    <Event>The log of messages is updated.</Event>
 </Flow_of_events>
 <Exceptional_events/>
 <Superordinate_use_case/>
 <Subordinate_use_case/>
</Use_Case>
```

Figure 3: Extract of XML representation for Use Cases

As shown in Figure 3, a use case is represented by element (<Use_Case>) and contains: a unique identifier (*UseCaseID*), information about the product line domain (*System*), and a product member identifier (*Product_Member)*. It has also a title (<Title>)*;* a brief textual description (<Description>)*;* information

about the level of functionality that the use case describes in a system (<Level>); pre- and post-conditions that must be satisfied before and after the execution of the use case respectively (<Preconditions> and <Postconditions>)*;* primary and secondary actors describing the users of the use case (<Primary_actor> and <Secondary_actors>); flow of events denoting the events that trigger the use case and the specification of the normal events that occur within it (<Flow_of_events>); exceptional events describing the events that not always occur when the use case is executed (<Exceptional_events>)*;* and superordinate and subordinate use cases (<Superordinate_use_case> and <Subordinate_use_case>). As in the case of feature model, the words in the textual parts of a use case are annotated with XML POS-tags denoting their grammatical roles. In Figure 3, we do not represent the XML POS-tags.

2.1.3 Subsystem Model

In FORM [35], a subsystem model is used at the product line level to represent the main functional groups of a system (internal subsystems), subsystems outside the scope of the system (external subsystems), and how the various subsystems relate to each other in terms of data and control flows. In our approach, we propose to represent subsystem models as an XML textual specification template, as shown in Figure 4. For simplicity, in Figure 4 we do not present the XML POS-tags of the words in the textual parts of the model.

```
<Subsystem_Model>
  <Subsystem>
   <Subsystem_name> Operating </Subsystem_name>
   <Description> This subsystem provides facilities for performing basic tasks such as control of the interaction
                 with all devices, software, and data; support of the interaction between internal applications
                 (e.g. games, multimedia, and PC connective), recognition of internal hardware (e.g. screen,
                 keypad, and Bluetooth) and different types of input data (e.g. air signal, keystroke, screen
                 touch, voice); response to different types of output data (e.g. air signal, screen-display, voice).
    </Description>
   <Type>internal</Type>
</Subsystem>
<Subsystem>
 <Subsystem_name> Messaging </Subsystem_name>
 <Description> This subsystem manages the exchange and manipulation of messages. It supports two services:
               short message service (SMS) for textual messages, and multimedia message service (MMS) for
               multimedia messages. The services are based on a store and forward protocol. The subsystem
               interacts with short message service centers (SMSC) or multimedia message service centers
               (MMSC) to receive an incoming message and to forward an outgoing message.
  </Description>
  <Type>internal</Type>
</Subsystem> …
<Flow flow_id = "c1" flow_type = "control_flow" sender = "Operating" receiver = "Messaging"/>
<Flow flow_id = "d2" flow_type = "data_flow" sender = "Messaging"  receiver = "Mobile Internet"/>  …
</Subsystem_Model>
```

Figure 4: Extract of XML representation for Subsystem Model

2.1.4 Process Model

FORM [35] proposes to use a graphical diagram called process model to represent the dynamic behaviour

of each subsystem in a subsystem model and messages exchanged between various processes and data shared by a process (e.g. database, reports, and files). Figure 5 shows our XML specification for a process model of *Messaging subsystem* in Figure 4 without the XML POS-tags.

```
<Process_Model ProcessModelID = "P1" Subsystem_name = "Messaging">
 <Process>
  <Process_name> Short Messaging Service (SMS) Control </Process_name>
  <Description> This process performs delivery and receive of a short message to a short message service
                center (SMSC). The SMSC is connected to the telecommunication network (e.g. GSM,
                HSCSD, and EDGE) through the short message service gateway mobile switching center (SMS
                GMSC). This process also attaches extra information about SMSC in a short message.
   </Description>
   <Activity>multiple</Activity>
   <Type>resident</Type>
 </Process> …
 <Process shared_data = "d1">
  <Process_name> Edit </Process_name>
  <Description> This process performs the composition of a short message.  The short message contains a
                receiver's address and context. The process provides a list of contacts and a set of template
                short messages. The process supports two editing modes i.e. alpha mode and predictive mode.
                The alpha mode accepts alphanumeric. The predictive mode predicts a word from an input
                 keystroke.
   </Description>
   <Activity>single</Activity>
   <Type>resident</Type>
</Process> …
<Message message_id="m7_trigger"    message_type="closely-coupled"
         sender="Short Messaging Service (SMS) Control"   receiver="Notification"/>
<Message message_id="m8_response"    message_type="closely-coupled"
          sender="Notification"    receiver="Short Messaging Service (SMS) Control"/> …
<Shared_data data_id="d1" type="database"/>
</Process_Model>
```

Figure 5: Extract of XML representation for Process Model

2.1.5 Module Model

In FORM [35], each process in a process model is further refined into a module model [40]. A module model represents a hierarchical structure of the various modules composing a process and their interactions. Figure 6 presents an example of the XML specification of a module model for process *Short Messaging Service (SMS) Control* in Figure 5 without the XML POS-tags.

```
<Module_Model ModuleModelID = "MM1"  Process_name = "Short Messaging Service (SMS) Control">
 <Module>
  <Module_name> Short Messaging </Module_name>
  <Description> The maximum length of a text message is 160 characters, numbers, or any alphanumeric
                combination. This module also supports for non-text based short messages such as binary
                format which, is used for ring tones and logos services. (…) </Description>
  <Type> precoded </Type>
 </Module> …
 <Link type="inherit" source="Short Messaging" destination="Messaging Edit"/> …
</Module_Model>
```

Figure 6: Extract of XML representation for Module Model

2.1.6 Class, Statechart, and Sequence Diagrams

The design aspects of the product members in a product line system are described in UML class, statechart, and sequence diagrams. In our approach, these diagrams are represented in XMI format [52] with the words in their textual parts marked-up with XML POS-tags. Due to their large popularity and use, we do not explain these diagrams in this paper. Examples of these diagrams and their XMI representations for the mobile phone case study used in our work can be found in [83].

2.2. Traceability Relations

Based on our study and analysis of the mobile phone domain, our study and experience with software traceability [66], the types of traceability relations proposed in the literature [5][48][53][60], the semantics of the documents of our concern, and the various tasks associated with product line engineering, we have identified nine different types of traceability relations for different elements in the various documents used in our approach. These traceability relations are classified in six different groups, as follows.

Group 1: Relations between elements in documents in the product line level and elements in documents in the product member level (e.g., feature_model vs. use_case).

Group 2: Relations between elements in documents of the same type for different product members (e.g., PM_1_class_diagram vs. PM_2_class_diagram)[4].

Group 3: Relations between elements in documents of different types for the same product member (e.g., PM_1_use_case vs. PM_1_class_diagram).

Group 4: Relations between elements in documents of different types for different product members (e.g., PM_1_use_case vs. PM_2_class_diagram).

Group 5: Relations between elements in documents of the same type for the same product member (e.g., PM_1_use_case_UC1 vs. PM_1_use_case_UC2).

Group 6: Relations between elements in different documents in the product line level (e.g., feature_model vs. subsystem_model).

Each of these groups can assist software development from different perspectives. For instance, relations in group 1 assist with the identification of reusable components; relations in group 2 and group 4 support comparisons between the various product members; relations in group 3 and group 6 assist with better understanding of each product member and the product line system itself, respectively; and relations in group 5 allow for the identification of evolution aspects in a product member.

We define below the various traceability relation types in our approach and illustrate these types through examples shown in Figure 7.

---

[4] PM_1 and PM_2 represent two different product members.

R1 - Satisfiability Relation: In this type of relation, an element e1 *satisfies* an element e2 if e1 meets the expectation and needs of e2. An example of this relation exists between the module *Short Messaging* and the feature *Text Messages,* as shown in Figure 7.

R2 - Dependency Relation*:* In this type of relation, an element e1 *depends on* an element e2 if the existence of e1 *relies on* the existence of e2, or if changes in e2 have to be reflected in e1. An example of this relation exists between the subsystem *Messaging* and the feature *Text Messages*. This is because the feature is allocated to subsystem *Messaging* and, therefore, changes in the feature have to be reflected in the subsystem, as shown in Figure 7.

R3 - Overlaps Relation: In this type of relation, an element e1 *overlaps* with an element e2 (and an element e2 *overlaps* with an element e1) if e1 and e2 refer to common aspects of a system or its domain. As shown in Figure 7, an example of this relation exists between operation *takePhoto():void* in the sequence diagram and the description of use case UC4 since this description contains the name of the operation and the name of the class of the object of this operation is in the sequence diagram.

R4 - Evolution Relation: In this type of relation, an element e1 *evolves to* an element e2 if e1 has been replaced by e2 during the development, maintenance, or evolution of the system. An *evolves* relation occurs between documents of the same type for the same product member. In Figure 7, an example of this relation exists between two state diagrams that have different parameters for the same signals.

R5 - Implements Relation: In this type of relation, an element e1 *implements* an element e2 if e1 *executes* or *allows* for the achievement of e2. As shown in Figure 7, examples of this relation exist between operation *takePhoto():void* of class *Camera* and use case UC3 and operation *takePhoto():void* of class *CameraZoom2X* and use case UC4, since these operations execute the functionalities of these use cases.

R6 - Refinement Relation: This type of relation associates elements in different levels of abstractions. A refinement relation identifies how complex elements can be broken down into components and subsystems, or how elements can be specified in more detail by other elements. Thus, in this type of relation, an element e1 *refines* an element e2 when e1 specifies more details about e2. In Figure 7, an example of this relation exists between the subsystem *Messaging* and the process model *P1*, since a process model describes the behaviour of a subsystem.

R7 - Containment Relation: In this type of relation, an element e1 *contains* an element e2 when e1 is a document, or an element in a document, that uses an element e2, or a set of elements from a different document. In Figure 7, examples of this relation exist between use case UC1 and feature *Text Messages* and use case UC2 and feature *Text Messages*, since the words in the titles of the use cases (or their synonyms) appear in the description of the feature.

R8 - Similar Relation: This type of relation occurs between elements of documents of the same type for different product members. This relation assists with the identification of common aspects between various product members. A *similar* relation between elements e1 and e2 depends on the existence of a relation between e1 and another element e3 and a relation between e2 and element e3. For example, a use case uc1 is *similar* to a use case uc2 if both uc1 and uc2 hold a *containment* relation with a feature f1. An example of this relation exists between use case UC1 (*Sending Message*) and use case UC2 (*Transmitting Messages*), since there are containment relations between UC1 and feature *Text Messages* and UC2 and feature *Text Messages,* as shown in Figure 7.

R9 - Different Relation: This type of relation also occurs between elements of documents of the same type for different product members. This relation assists with the identification of variable aspects between various product members. A *different* relation between an element e1 and e2 depends on the existence of a relation between e1 and another element e3, and a relation between e2 and another element e4, where e3 and e4 are variants of the same variability point (e.g. subclasses of the same superclass, sibling features of the same parent feature). For example, a use case uc1 is *different* from a use case uc2 when there are two subclasses c1 and c2 of the same parent class c, where c1 *implements* uc1 and c2 *implements* uc2. An example of this relation exists in Figure 7 between use cases UC3 (*Taking Photo*) and use case UC4 (*Taking Picture*), because of the implements relations that exist between UC3 and operation *takePhoto():void* of class *Camera* and UC4 and operation *takePhoto():void* of class *CameraZoom2x*, and because these classes are subclasses of class *CameraApplication*.

The above traceability relations are important to support different scenarios of product line engineering when using a feature-based object-oriented methodology (see Section 5). Our experience has shown that these relations are complete for the documents and scenarios covered in our work. Although the approaches in [5][36][48] have suggested other classifications for traceability relations in the scope of product line, our classification tackles both feature-based and object-oriented document types. Moreover, the proposed relations are not mutual exclusive and different types of relations can be used to relate the same elements.

Table 2 presents a summary of the traceability reference model being proposed. In the table, each cell contains the different types of traceability relations that may exist between the documents described in the row and column of that cell. In the table we do not represent the exact elements that are related in the different documents, but represent the types of the documents. The direction of the relation is represented from a row *[i]* to a column *[j]*. Thus, a relation type *rel_type* in a cell *[i][j]* signifies that "*[i] is related to [j] though rel_type*" (e.g. "subsystem model *satisfies* feature model"). The traceability relations that are bi-directional appear in two correspondent cells for that relation (e.g., "subsystem model overlaps feature model" and "feature model overlaps subsystem model").

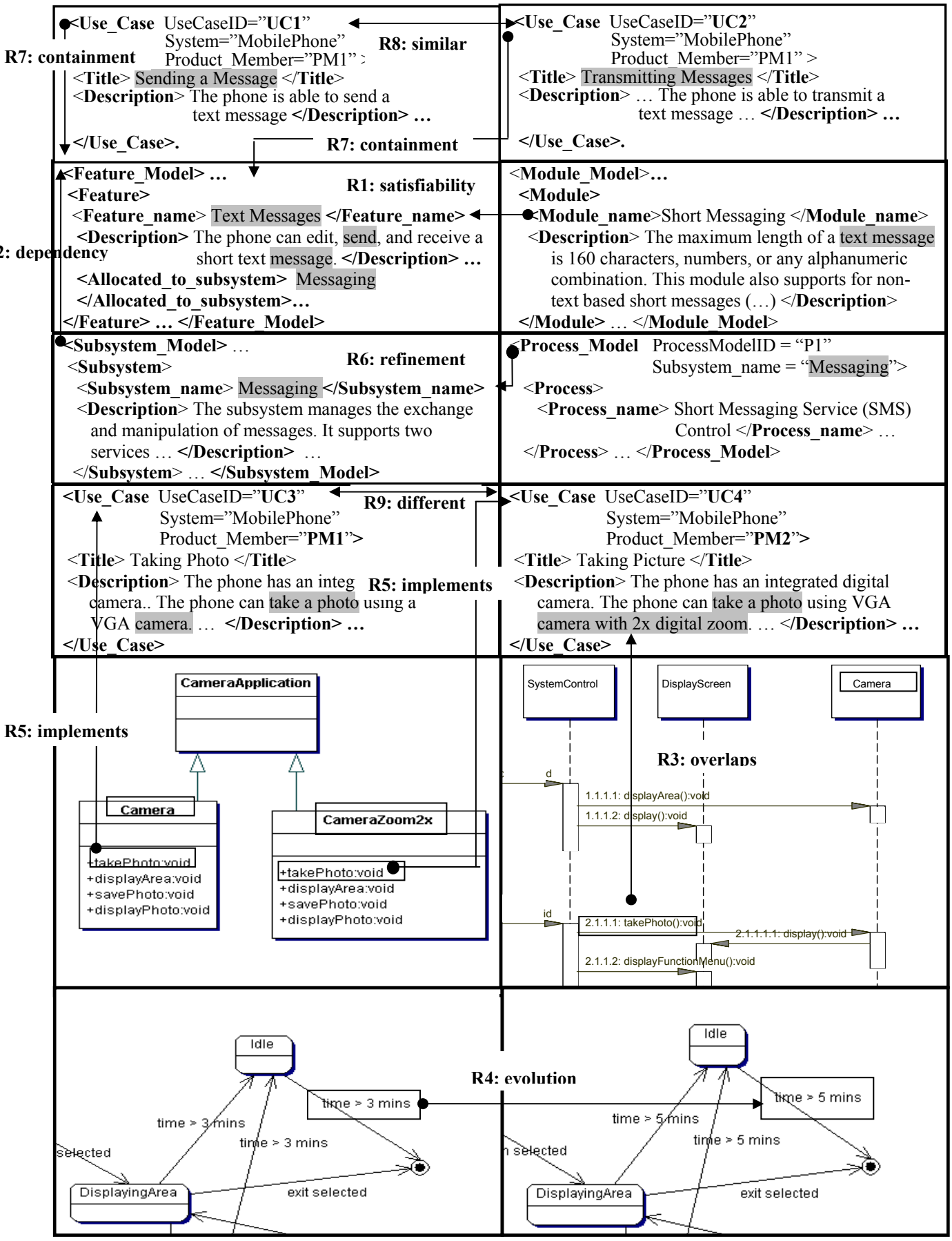

Figure 7: Examples of traceability relations

Table 2: Traceability Reference Model

	Feature Model	**Subsystem Model**	**Process Model**	**Module Model**	**Use Case**	**Class Diagram**	**Statechart Diagram**	**Sequence Diagram**
Feature Model		*Overlaps*	*Overlaps*	*Overlaps*		*Overlaps*	*Overlaps*	*Overlaps*
Subsystem Model	*Satisfies Depends_on Refines Overlaps*					*Contains*		
Process Model	*Satisfies Depends_on Refines Overlaps*	*Refines*				*Contains*	*Contains*	*Contains*
Module Model	*Satisfies Depends_on Refines Overlaps*		*Refines*			*Contains*		
Use Case	*Contains Depends_on*				*Similar Different Evolves*	*Overlaps*	*Overlaps*	*Overlaps*
Class Diagram	*Satisfies Depends_on Overlaps Implements*	*Refines Depends_on*	*Refines Depends_on*	*Refines Depends_on*	*Satisfies Depends_on Overlaps Implements Refines*	*Similar Different Evolves*	*Overlaps*	*Overlaps*
Statechart Diagram	*Satisfies Depends_on Overlaps Implements*		*Refines Depends_on*	*Refines Depends_on*	*Satisfies Depends_on Overlaps Implements Refines*	*Depends_on Overlaps Contains*	*Similar Different Evolves*	*Overlaps Refines*
Sequence Diagram	*Satisfies Depends_on Overlaps Implements*		*Refines Depends_on*	*Refines Depends_on*	*Satisfies Depends_on Overlaps Implements Refines*	*Depends_on Overlaps Refines Contains*	*Overlaps*	*Similar Different Evolves*

3. Traceability Approach

3.1 Overview

As discussed in Section 1, in our approach the generation of traceability relations is based on the use of rules. In general, rules assist and automate decision making, allow for standard ways of representing knowledge that can be used to infer data, facilitate the construction of traceability generators for large data sets, and support representation of dependencies between elements in the documents. In addition, the use of rules in our approach allows for the generation of new relations based on the existence of other relations, supports the heterogeneity of documents being compared, and supports data inference in similar applications.

We use an extended version of XQuery [82] to represent the rules. XQuery is an XML-based query language that has been widely used for manipulating, retrieving, and interpreting information from XML documents. Apart from the embedded functions offered by XQuery, it is possible to add new functions and commands. We have extended XQuery to support representation of consequence part of the rules, i.e. the actions to be taken when the conditions are satisfied, and to support extra functions to cover some of the traceability relations being proposed. Examples of these functions are *findSynonym*, to identify a set of words that have the same meaning of a given word, and *checkDistanceControl*, to identify if two words are associated in a textual paragraph, depending on how distant the words are in a sentence. In the approach,

we assume that closer words in a sentence are more likely to be related to each other. Other extra functions are described in Subsection 3.2.

Figure 8 presents an overview of the traceability process, which is composed of three main stages, namely:

(a) Annotation of textual sentences in the documents with part-of-speech (POS) assignments (Grammatical Tagging), using CLAWS C7 [14].

(b) Creation of documents in XML format (XML Creation), based on the XML schemas and the POS tags generated by CLAWS. The POS tags generated by CLAWS are converted into XML tags as shown in the example in Figure 1. The conversion of the POS tags into XML POS-tags is done automatically by using a converter that we have implemented. In some situations, CLAWS tagger suggests more than one part-of-speech tag for a certain word. In this case, the conversion process uses the first part-of-speech tag suggested for the word.

(c) Generation of direct and indirect traceability relations (Traceability Generation), based on traceability rules and extra functions.

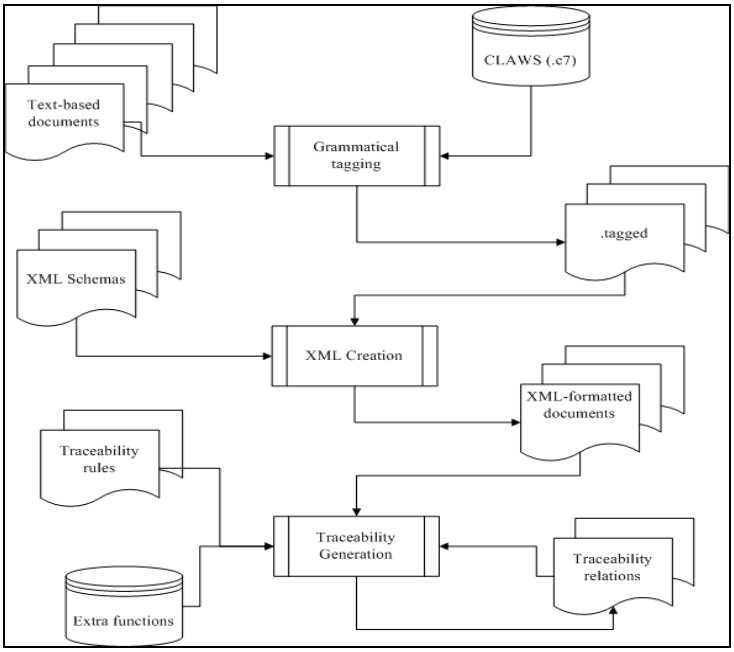

Figure 8: Overview of traceability process

Figure 9 presents a more detailed view of the traceability generation process. More specifically, traceability relations are generated by a *Traceability Generator* component that we have developed which is formed by two sub-components: (a) *rule inference* and (b) *rule parser*. The *rule inference* sub-component is responsible for (i) identifying the traceability rules related to different types of documents to be traced and different types of traceability relations to be generated, and (ii) instantiating placeholders for document types in the identified rules with the names of the documents to be traced. The information about which traceability documents to be traced and traceability relations to be generated are given by the user (see Section 4). The *rule parser* sub-component is responsible for executing the rules. It uses the XML-

formatted documents, extra functions that we have implemented to cover some of the traceability relations, and WordNet [80] to assist with the identification of synonyms. The direct and indirect traceability relations resulting from the execution of the rules are represented in XML documents (Direct_Trace_Rel.xml and Indirect_Trace_Rel.xml, respectively). The document with direct traceability relations is used as input to the rule parser for generating indirect traceability relations.

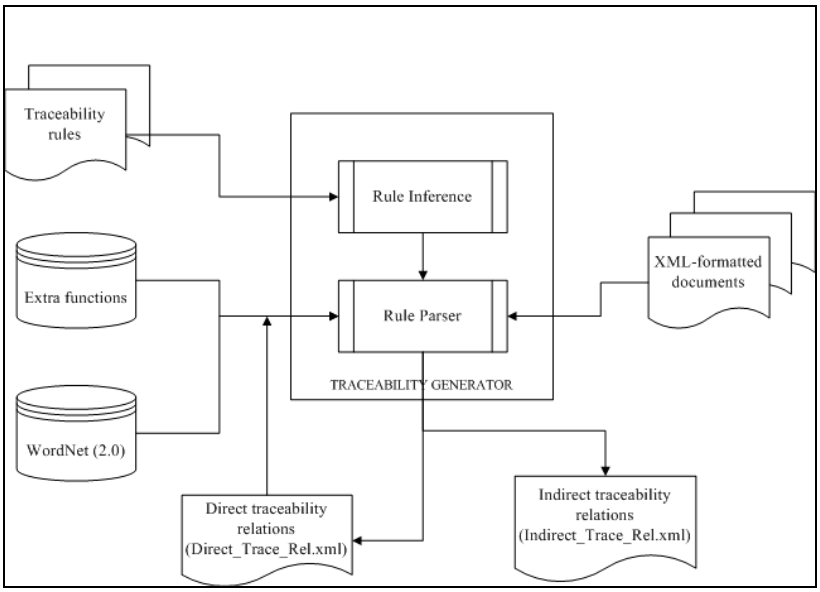

Figure 9: Overview of traceability generation process

3.2 Traceability Rules

The traceability relations described in Subsection 2.2 can be automatically generated in our approach by the use of traceability rules. The traceability rules used in our work have been created based on the following aspects:

(i) the semantics of the documents being compared,

(ii) the various types of traceability relations in the product line domain,

(iii) the grammatical roles of the words in the textual parts of the documents, and

(iv) synonyms and distance of words being compared in a text.

As an example of case (i) above, a rule for comparing feature and use case models takes into consideration the fact that a feature model specifies requirements at the product line level, while use cases describe requirements for product members which may be more specific. Therefore, it may be necessary to traverse the hierarchy of a feature and investigate if one or more children of a feature appear in the use case. Similarly, a sequence diagram describes the order in which messages are exchanged between various class objects and, therefore, rules for comparing operations in classes and sequences of messages in a sequence diagram should be used.

Regarding case (ii), the types of traceability relations also play an important role in the various traceability rules. For example, there is no need to create traceability rules for identifying evolution

relations between elements in feature models and class diagrams, feature models and sequence diagrams, or feature models and statechart diagrams since such relations do not exist between the above documents (evolution relations exist between documents of the same type for the same product member).

Considering case (iii), it is a common approach that names given by software engineers for the main elements in class, sequence, statechart, subsystem, process, and module diagrams do not contain certain types of words such as articles, co-ordinating and subordinating conjunctions, singular and plural determiner, comparative and superlative adjectives, etc. Therefore, when comparing descriptions of use cases and feature names, or flow of events in use cases with elements in the above diagrams (e.g. classes, messages, operations, transitions, process, subsystem, modules), those types of words do not need to be considered.

```
TRACE_RULE    RuleID     = R_ID
              RuleType   = Rule_Type
              DocType1 = DocTypeName
              DocType2 = DocTypeName
  QUERY
    [DECLARE Namespace]
    [DECLARE Functions]
    [DECLARE Variables]
    for $variable_name1 in doc(DocType1Placeholder)//XPathExpression
        $variable_name2 in doc(DocType1Placeholder)//XPathExpression
    where
        fi(fi+1…(fi+j(●))…)
  QUERY_END
  ACTION
    RELATION   RuleID     = R_ID
               Type       = Relation_Type
               DocType1 = DocTypeName
               DocType2 = DocTypeName
    ELEMENT  Document = DocName    [ElementType1] $variable_name1[/XpathExpression]  [ElementType2]
    ELEMENT  Document = DocName    [ElementType1] $variable_name2[/XpathExpression]  [ElementType2]
    [RelationType {XpathExpression} {XpathExpression}]
    [RelationType {XpathExpression} {XpathExpression}]
  ACTION_END
TRACE_RULE_END
```

Figure 10: Traceability rule template

With respect to case (iv), the multiplicity of stakeholders participating in the development of the system, the different phases of product line engineering (domain analysis vs. domain design), and the different levels of specialisation of the system (product line vs. product members) may lead to the use of different words to represent the same thing (i.e., synonyms). Furthermore, the existence of two or more words in a paragraph description does not imply that the paragraph is concerned with these words, in particular when the words appear in different sentences in the paragraph or in different phrases in the same sentence. As an example consider the description of subsystem *Messaging* in Figure 4 and operation *exchange_service()*. In this case, although the paragraph contains the words “exchange” and “service”, the text in the paragraph is not concerned with the “exchange of services”, but with the “exchange and manipulation of messages” and the support for two types of message services (SMS and SMSC). If the distances of the words in the paragraph were not considered, the operation would have been incorrectly related to the description of the subsystem.

The traceability rules can be (a) *direct,* when they support the generation of traceability relations that do not depend on the existence of other relations such as satisfiability, dependency, overlaps, evolution, implements, refinement, and containment relations; or (b) *indirect*, when they support the generation of traceability relations that depend on the existence of other relations such as similar and different relations.

```
<TraceRule RuleID="R1" RuleType="containment" DocType1="Use Case" DocType2="Feature Model">
 <Query>
   declare namespace s="java:synonym.s";
   declare namespace d="java:distanceControl.d";
   for $item1 in doc("file:///c:/UseCase_UC1.xml")//Use_Case,
       $item2 in doc("file:///c:/Feature_MP.xml")//Feature_Model/Feature
   where
     d:checkDistanceControl($item2/Description,
        s:setof(s:findSynonym($item1/Title/VVI),s:findSynonym($item1/title/VVB),
                s:findSynonym($item1/Title/VV0), s:findSynonym($item1/Title/VVG)) ,
        s:setof(s:findSynonym($item1/Title/NN0), s:findSynonym($item1/Title/NN1),
                s:findSynonym($item1/Title/NP0),s:findSynonym($item1/Title/NN2)))
</Query>
<Action>
  <Relation RuleID="R1" Type="containment" DocType1="Use Case" DocType2="Feature Model">
     <Element Document="file:///c:/UseCase_UC1.xml"> {$item1/Title} </Element>
     <Element Document="file:///c:/Feature_MP.xml"> {$item2/Feature_name} <Description/> </Element>
  </Relation>
</Action>
</TraceRule>
```

Figure 11: Example of *containment* traceability rule

```
<TraceRule RulID="R2" RuleType="similar" DocType1="XML-Based-Rel" DocType2="XML-Based-Rel">
  <Query>
    for  $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@type="containment"],
         $item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@type="containment"]
    where
         $item1/@DocType1="Use Case" and $item1/@DocType2="Feature Model" and
         $item2/@DocType1="Use Case" and $item2/@DocType2="Feature Model" and
         string($item1/Element[2]) =  string($item2/Element[2]) and
h        $item1/Element[1]/@Document != $item2/Element[1]/@Document
  </Query>
  <Action>
    <Relation RuleID="R2" Type = "similar" Term ="use case contains feature model">
        <Element>{$item1/Element[1]/@Document} {$item1/Element[1]/Title} </Element>
        <Element>{$item2/Element[1]/@Document} {$item2/Element[1]/Title} </Element>
        <Containment>{$item1/Element[2]/@Document}{$item1/Element[2]/Feature_name} </Containment>
    </Relation>
  </Action>
</TraceRule>
```

Figure 12: Example of *similar* traceability rule

Figure 10 shows a general template for direct and indirect traceability rules. In the template, elements between square brackets ("[" ,"]") are optional, and fi(fi+1…(fi+j(●))…) are embedded XQuery functions or extra functions that we have developed. The XML Schema for our traceability rules can be found in [83]. Both types of traceability rules are composed of three main parts described below. An example of a traceability rule for a *containment* traceability relation between use cases and feature models is shown in Figure 11 and an example for a *similar* traceability relation is shown in Figure 12. Examples of the results

of the traceability rule in Figure 11 are shown in Figure 13, while Figure 14 shows an example of the results of the traceability rule in Figure 12. We explain below the different parts in a rule.

RULE_IDENTICATION: This part is concerned with the identification of the rule and the documents participating in the rule. It contains a unique *RuleID*, a description of the type of the rule (*RuleType*), and descriptions of the types of documents associated with the rule (*DocType1*, *DocType2*). The rule type is the same as the type of traceability relation generated by the rule. In our approach, there are various traceability rules for a certain type; i.e., there are various rules that support the generation of the same type of traceability relation. In the case of direct traceability rules, attributes *DocType1* and *DocType2* contain the names of the different types of documents used in our approach (Use Case and Feature Model in Figure 11); while for indirect traceability rules, attributes *DocType1* and *DocType2* refer to the XML_Base_Relationship document that contains the results of previously identified relations (XML-Base-Rel in Figure 12).

QUERY: This part is concerned with the conditions of the rule. It is represented by element <Query> and consists of XQuery statements. It is composed of three other subparts, as described below.

The first subpart (*declare*) is optional and contains declarations of namespaces, variables, or extra functions used in the rule. In our approach, the extra functions that we have developed are either implemented as XQuery statements (viz. XQuery_functions) or as Java classes (viz. Java_functions). The XQuery_functions are declared as *function*. The Java_functions are represented as Java packages and declared as *namespace.* Figure 11 shows an example of these declarations for Java functions. The example in Figure 12 does not make use of any declaration. The extra functions allow for the identification of specific elements in the documents, identification of words that are synonyms, or textual comparisons. Table 3 presents a list of these functions and their descriptions. The code of these functions is out of the scope of this paper, but can be found in [83].

The second subpart (*for*) identifies elements of the documents (*DocType1* and *DocType2*) to be compared and binds these elements to variables *$item1* and *$item2*, respectively (*$variable_name1* and *$variable_name2* in Figure 10). Initially, the elements to be compared are described in XPath [81] expressions associated with placeholders that represent the types of documents to be traced. The placeholders for the documents to be traced are automatically substituted by specific document names (file names) after the user has indicated these documents through our traceability tool (see Section 4). The examples in Figure 11 and Figure 12 show the values for *$item1* and *$item2* already instantiated with the document names (UseCase_UC1.xml and Feature_MP.xml in Figure 11 with the XPath expressions for the respective elements, and Direct_Trace_Rel.xml in Figure 12 with XPath expressions for relations of type *containment*). In the case of indirect traceability rules, *$item1* and *$item2* refer to Direct_Trace_Rel.xml. However, the type of the relation given by the XPath expression (\\Relation[@type=" "]) differs depending

on the rule type.

The third subpart (*where*) describes the *condition* part of the rule that should be satisfied in order to create a traceability relation. The condition part can use a sequence, conjunction, or disjunction of XQuery in-built functions (e.g., *some*, *contains*, *satisfies*), or of the extra XQuery or Java functions that we have implemented. Depending on the rule, the condition part also takes into consideration the XML POS-tags in the textual parts of the documents.

Table 3: XQuery and Java functions used in the traceability rules

Functions	Description
XQuery Functions	
getTransitioninState(): item()[5]*	Identifies the set of transitions in a statechart diagram
getStateinState($*transition* as node()): item()	Identifies the state of a transition in a statechart diagram
getMessageinSeq(): item()*	Identifies the set of messages in a sequence diagram
getObjectinSeq($link as node()): item()	Identifies the object of a message or operation in a sequence diagram
getClassObjectinSeq($object as node()): item()	Identifies the class of an object in a sequence diagram
getClassinClass($diagram as xs:string): item()*	Identifies the classes in a class diagram
getParentFeature($child as xs:string): item()	Identifies the parent feature of a feature in a feature model
getChildrenFeature($parent as xs:string): item()*	Identifies the set of children features of a feature in a feature model
getFeatureofSubsystem($Subsystem as xs:string):item()*	Identifies the set of features used by a subsystem in a subsystem model
getOperationinSeq(): item()*	Identifies the set of operations in a sequence diagram
getOperationinClass($object as node()):item()*	Identified the set of operations in a class diagram
getStateofOperationinState($operation as node()): item()	Identifies the state that receives an event when the event represents an operation
getParentofVariantFeatures($one as node(), $two as node()): item()	Identifies the parent feature of two features that are either alternative or optional
getParentofVariantClasses($one as xs:string, $two as xs:string): item()	Identifies the superclass of two classes in a class diagram
getParenClass($child as xs:string): item()	Identifies the superclass of a class in a class diagram
getClassID($name as xs:string)as xs:string	Identifies the identifier of a class in a class diagram
Java Functions	
containsInDistance(Object[6] word1, Object word2, ArrayList[7] word3): Boolean [8]	Determines if word1 contains word2 and word3, or their synonyms, considering their part-of-speech
stringNoSpace(String strInput):String	Returns a string without white spaces
setof(ArrayList s1, ArrayList s2, ArrayList s3, ArrayList s4): ArrayList	Returns a set composed by the parameters
checkDistanceControl(String strInput, ArrayList s1, ArrayList s2):Boolean	Identifies if the set of synonyms of two words (s1 and s2) appears in the same sentence in a textual paragraph (strInput)
findSynonym(String word): ArrayList	Identifies a set of synonyms for a word

In the example of Figure 11, the rule verifies if the words (or their set of synonyms) in element *Title* of UseCase_UC1 appear in the same sentence in the *Description* of a feature in Feature_MP.xml (checkDistanceControl). The rule checks for synonyms, by using WordNet [80], of any possible form of the main verb (VVI, VVB, VVG, VV0) and of any possible form of the noun (NN0, NN1, NN2, NP0) of the verb-phrase in the title of the use case. In Figure 12, the rule verifies if there are two relations of type *containment* in Direct_Trace_Rel.xml document between a use case and a feature model such that the feature names are the same and the use cases are different. The elements representing feature names and

[5] XQuery *item()** implies a sequence of *item()*, which in XQuery can be an XML node (*node()*) , an XML element (*element()*), or atomic values such as string and integer [82].
[6] Java *Object* representing an XML node or an XML element.
[7] Java *ArrayList* representing a sequence of XML elements, XML nodes, or strings.
[8] Variants of this function with other parameter types have also been implemented.

use cases are accessed from Direct_Trace_Rel.xml document by using XPath expressions. These elements are referenced in the XPath expression as Element[2] and Element[1], respectively. They appear in Direct_trace_Rel.xml document as the second and first XML elements <Element> of an XML relation element <Relation>, as shown in the extract of the document in Figure 13.

ACTION: This part describes the *consequence* of the rule and is represented by element (<Action>). It specifies the action(s) to be taken if the conditions in the QUERY part are satisfied. The consequence part describes the type of traceability relation to be created (attribute Type) and the elements that should be related through it in the documents described in the *for* part of the rule (element <Element>). For the case of direct traceability rules, an extra element associated with each element (ElementType2 in Figure 10) may be used to indicate the exact type of elements in the respective documents that were satisfied by the rule, when necessary. The extra element represented by ElementType1 in Figure 10 is used when the content of *$variable_name1* or *$variable_name2* is of type string and it is necessary to represent the XML element that this content represents. For the case of indirect traceability rules, a special element is used to represent how the elements being compared are similar or different (RelationType in Figure 10). The content of element <Action> is used to compose the *return* part of XQuery. The implementation of an action consists of writing the information in the <Action> part in the XML relation document (Direct_Trace_Rel.xml and Indirect_Trace_Rel.xml). As in the QUERY part, the placeholders of the specific documents containing the elements to be associated are instantiated based on the user's input.

```
<Relation_Document>
  <Relation RuleID="R1" Type="containment" DocType1="Use Case" DocType2="Feature Model">
    <Element Document="file:///c:/UseCase_UC1.xml">
      <Title> <VVG> Sending </VVG> <AT0> a </AT0> <NN1> Message </NN1> </Title>
    </Element>
    <Element Document="file:///c:/Feature_MP.xml">
      <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2> <Description></Description>
      </Feature_name>
    </Element>
  </Relation>
  <Relation RuleID="R1" Type="containment" DocType1="Use Case" DocType2="Feature Model">
    <Element Document="file:///c:/UseCase_UC2.xml">
      <Title> <VVG> Transmitting </VVG> <NN2> Messages </NN2> </Title>
    </Element>
    <Element Document="file:///c:/Feature_MP.xml">
      <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2> <Description></Description>
      </Feature_name>
    </Element>
  </Relation>
</Relation_Document>
```

Figure 13: Result of *containment* traceability relation

In Figure 13, a relation of type *containment* is created between the title of use case UseCase_UC1 (first <Element>) and the feature name in Feature_MP.xml document that satisfies the condition part of the rule (second <Element>) represented by XPath expressions. An element <Description> is used to indicate that

the relation is between the title of the use case and the description of the feature. In Figure 14, a relation of type similar is created between the titles of the two use cases (both elements <Element>) together with an extra element representing how the two use cases are similar, i.e., through a *containment* relation with the feature (element <Containment>).

As discussed in Subsection 2.2, an example of rule R1 in Figure 11 exists between use case UC1 entitled *Sending a Message* (Figure 3) and feature named *Text Messages* (Figure 2). A *containment* relation is created since a synonym (send) of verb <VVG> Sending </VVG> and noun <NN1> Message </NN1> appear in the description of the feature in the same sentence; i.e., a sequence of a conjunction of verbs (<VVI> send </VVI> <SC>,</SC>, <CJC>and</CJC>, <VVI> receive</VVI>), followed by a qualifier of the noun *message* (<AT0> a</AT0> <AJ0>short</AJ0> <NN1> text </NN1>), separate the words *send* and *message*. Another example of rule R1 also exists between use case UC2 entitled *Transmitting Messages* and feature *Text Messages*. In this case, a *containment* relation is also created. The results of rule R1 for use cases UC1 and UC2 and feature *Text Messages* are shown in Figure 13.

An example of rule R2 in Figure 12 exists between use cases UC1 and UC2. A *similar* relation is created since there are two *containment* relations between UseCase_UC1 and feature *Text Messages* and UseCase_UC2 and feature *Text Messages*. The result of rule R2 is shown in Figure 14.

```
<Relation RuleID = "R2" Type ="similar" Term = "use case contains feature model" >
    <Element Document="file:///c:/UseCase_UC1.xml">
        <Title>  <VVG> Sending </VVG> <AT0> a </AT0> <NN1> Message </NN1> </Title>
    </Element>
    <Element Document="file:///c:/UseCase_UC2.xml">
        <Title>  <VVG>Transmitting</VVG> <NN2> Messages </NN2> </Title>
    </Element>
    <Containment Document="file:///c:/Feature_MP.xml">
        <Feature_name> <NN1>Text</NN1> <NN2>Messages</NN2> </Feature_name>
    </Containment>
</Relation>
```

Figure 14: Result of *similar* traceability relation

4. Implementation

In order to evaluate and demonstrate our approach, we have implemented a prototype tool called XTraQue. We envisage the use of our tool as a general platform for automatic generation of traceability relations and support for product line engineering. The tool has been implemented in Java and uses Saxon [69] to evaluate XQuery [82]. The XTraQue tool implementation contains 10000 lines of code to support five main functionalities, namely:

(a) specification of the documents to be traced;

(b) specification of the types of relations to be created;

(c) generation of direct and indirect traceability relations based on the input given in (a) and (b);

(d) visualisation of the documents containing traceability relations generated in (c); and

(e) testing of new traceability rules.

For functionality (a), the tool has a sophisticated user interface in which users can select to establish traceability relations between (i) documents of two specific product members, (ii) documents at the level of product line and one specific product member, and (iii) documents at the level of product line and two specific product members.

For any of cases (i) to (iii) above, the user can select to trace all the documents related to the product line and product members, or to specify which documents to be traced based on (i') type of documents (e.g., all use cases, class, statechart, and sequence diagrams of a product member, or all feature, subsystem, process, and module models of a product line); (ii') particular document names; or (iii') types of traceability relations. In this latter case, the types of documents to be traced are selected depending on the elements of the documents that can be associated with a specific relation type. For example, an *implements* relation may exist between elements in class diagrams and feature models or use cases, elements in sequence diagrams and feature models or use cases, and elements in statechart diagram and feature models or use cases. Therefore, documents at the product line domain design level (e.g. subsystem, process, and module models) will not be selected to have their elements traced in this case. Moreover, the tool will not attempt to establish *implements* relations between elements of documents that have been selected but do not hold the relation type (e.g., elements in feature and use case models).

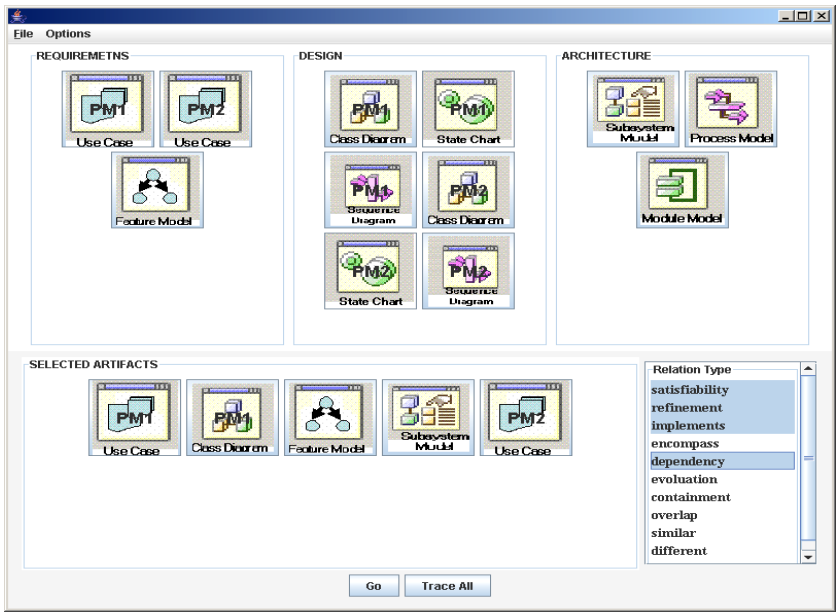

Figure 15: Example of XTraQue interface

In the case that the user selects to trace all documents or documents based on case (i') and case (ii') above, the tool also allows the user to specify the types of relations to be traced. The user can select to trace the documents for all traceability relations for any of (i') to (iii') cases. Figure 15 shows an example of one of the interfaces of XTraQue in which the user has selected the types of documents to be traced and the types of traceability relations.

The generation of direct and indirect traceability relations is executed by the *Traceability Generator*

component. For each pair of documents, the *Traceability Generator* automatically identifies traceability rules associated with the documents, instantiates the placeholders for the document types in the rules, and generates direct relations in XML format and indirect relations based on the direct ones also in XML format.

The XML documents containing the traceability relations can be visualised in the tool. The XTraQue tool allows for the creation of new traceability rules and the execution of these rules in order to verify their correctness. After the user is satisfied with a new rule, this rule can be inserted in the document containing all the traceability rules.

5. Evaluation and Analysis

As discussed in Section 1, there are many activities and difficulties associated with product line engineering. Moreover, as proposed in [37], organisations can develop product line systems in three different ways, namely *proactive*, when an organisation decides to analyse, design, and implement a line of products prior to the creation of individual product members; *reactive*, when an organisation enlarges the product line system in an incremental way based on the demand of new product members or new requirements for existing products; and *extractive*, when an organisation creates a product line based on existing product members by identifying and using common and variable aspects of these products[9]. These approaches are not mutually exclusive and can be used in combination. For instance, it is possible to have a product line system initially created in an extractive way to be incrementally enlarged over time by using a reactive approach. In addition, various stakeholders may be involved in the product line development process ranging from market researchers, to product managers, requirement engineers, product-line engineers, software analysts, and software developers. These stakeholders contribute in different ways to product line engineering, have distinct perspectives of the system, and have distinct interests in different aspects of the product line. For example, a market researcher may be interested in the requirements and features of a new product member to be developed, while a software developer may be interested in the design and implementation aspects of this new product member. Therefore, the stakeholders would be interested in different types of documents and traceability relations that could assist them in their various tasks during system development.

In order to evaluate our work and consider the various documents and traceability relation types used in our approach, we have conducted five sets of experiments related to five different scenarios concerned with product line engineering. More specifically, these scenarios include (a) the creation of a new product member for an existing product line, (b) the creation of a product line system from already existing product members, (c) changes to a product member in a product line system, (d) changes at the product line level,

[9] The proactive approach is also known as top-down approach, while the extractive approach is known as bottom-up approach [67].

and (e) impact of changes at the product line level to a product member. Although these scenarios are not a complete base set for product line system development, they have been chosen since they illustrate the different ways in which organisations can develop product line systems, as discussed above. For each of these scenarios we have identified the stakeholders involved in the process, the types of documents and traceability relations that are related to the scenarios, and evaluated the scenarios in terms of recall and precision measurements, as defined in page 30. Moreover, we have used our mobile phone case study to evaluate the scenarios.

Table 4: List of functionalities of the product members in the mobile phone case study

Functionality	PM_1	PM_2	PM_3
F1: Make and receive calls using GSM 900	X	X	X
F2: Make and receive calls using GSM 1800	X	X	X
F3: Make and receive calls using GSM 1900		X	X
F4: Hold and swap a call	X	X	X
F5: Receive and update voice mail	X	X	X
F6: Display and update time and date	X	X	X
F7: Set alarm and time	X	X	X
F8: Record, display, and manipulate call logs	X	X	X
F9: Play games	X	X	X
F10: Update calendar	X	X	X
F11: Add, delete, and update preferences	X	X	X
F12: Add, delete, and update contacts	X	X	X
F13: Include calculator	X	X	X
F14: Take photos using VGA camera	X		
F15: Take photos using VGA camera with 2x digital zoom		X	
F16: FM radio			X
F17: Email system using SMTP, POP3, or IMPA4	X	X	X
F18: Hand-free speaker	X		X
F19: Send and receive text messages	X	X	X
F20: Send and receive multimedia message	X	X	X
F21: Play RealOne format tunes and video		X	
F22: Play and record MP3 format tunes			X
F23: Record and update video (clips)		X	
F24: Play 3GPP video format		X	X
F25: Play Real Video format		X	
F26: Access Internet using WAP 1.2.1	X		X
F27: Access Internet using WAP 2.0		X	
F28: Access Internet using WAP XHTML		X	X
F29: Connect via Bluetooth transfer data	X	X	X
F30: Connect via Infrared transfer data	X	X	
F31: Connect via USB			X
F32: Play MIDI formatted tunes	X	X	X
F33: Play AMR formatted tunes	X		X
F34: Play AAC formatted tunes			X
F35: Play MP3 formatted tunes			X
F36: Play WAV formatted tunes			X
F37: Play True Tones formatted tunes		X	
F38: Compose and play MIDI formatted ring tones		X	X
F39: Record and update voice messages	X	X	X
F40: Transfer data via SyncML and TCP/IP	X	X	X
F41: Support CLDC Java technology	X	X	X
F42: Support MIPD Java technology	X	X	X
F43: Support Wireless messaging API Java technology		X	X
F44: Support Mobile media API Java technology		X	X

The mobile phone case study has been developed based on study, analysis, and discussions of mobile

phone domains, and ideas in [50][51]. The case study includes a line of systems with different mobile phones. This line of systems is composed of three product members (mobile phones), namely PM_1, PM_2, and PM_3, with common and variable characteristics. Table 4 presents some of the various functionalities of the three product members in our case study.

Table 5: Number of document types used in the mobile phone case study, number of main elements in the documents, and size of the documents

Document Type	Number of Document Type	Element Type	Number of Element Type	Size of XML document
Feature Model	1	Features	130	58 KB
Subsystem Model	1	Subsystems	5	9.24 KB
Process Models	6	Processes	48 (total for all 6 process models)	Ranging from 6KB to 12.5 KB
Module Models	15	Modules	167 (total for all 15 module models)	Ranging from 2KB to 9.2 KB
Use Cases	PM_1 = 4 PM_2 = 4 PM_3 = 5	Events	PM_1 = 37 (total for all 4 use cases)	Ranging from 4.4KB to 7KB
			PM_2 = 36 (total for all 4 use cases)	Ranging from 4.9KB to 8KB
			PM_3 = 44 (total for all 5 use cases)	Ranging from 4.3KB to 5.5KB
Class Diagrams	PM_1 = 1 PM_2 = 1 PM_3 = 1	Classes	PM_1 = 23	
			PM_2 = 25	
			PM_3 = 27	
		Attributes	PM_1 = 26	
			PM_2 = 26	
			PM_3 = 33	
		Methods	PM_1 = 78	
			PM_2 = 82	
			PM_3 = 87	
Sequence Diagrams	PM_1 = 4 PM_2 = 4 PM_3 = 5	Messages	PM_1 = 114 (in total for all 4 seq. diagrams)	
			PM_2 = 82 (in total for all 4 seq. diagrams)	
			PM_3 = 112 (in total for all 5 seq. diagrams)	
		Objects	PM_1 = 22 (in total for all 4 seq. diagrams)	
			PM_2 = 21 (in total for all 4 seq. diagrams)	
			PM_3 = 27 (in total for all 5 seq. diagrams)	
Statechart Diagrams	PM_1 = 1 PM_2 = 1 PM_3 = 1	States	PM_1 = 4	
			PM_2 = 4	
			PM_3 = 4	
		Transitions	PM_1 = 8	
			PM_2 = 8	
			PM_3 = 8	
UML Documents				PM_1 = 1.33 MB
				PM_2 = 1.4 MB
				PM_3 = 1.5 MB
Legend: PM_1, PM_2, PM_3 represent each of the three product members in the case study, respectively				

Table 5 shows a summary of the types and number of documents for each type used in the case study,

the size of the various documents with respect to the number of the main elements in the documents, and the size of the XML files representing the documents. Please note that the class, sequence, and statechart diagrams of a product member are represented in a single XMI file due to the nature of XMI. Therefore, in the table, we present the size of one XMI file for each product member (rows related to UML Documents). Moreover, for the documents representing information of product members (use cases, class, sequence, and statechart diagrams), we present the number of these documents and the number of the main elements in these documents for each product member in the case study. The documents and traceability rules used in the case study can be found in [83]. We describe below each of the five scenarios and the results of our experiments.

Scenario 1: Creation of a new product member for an existing product line

This situation occurs when an organisation wants to enlarge its system and creates a new product member. In this case, traceability relations can be used to support the evolution of software systems and reuse of existing parts of the system. The stakeholders involved in this scenario are (a) market researchers that are responsible for identifying the feasibility of creating a new product and the features that this new product should include from a commercial point-of-view; (b) requirements engineers that specify the requirements of the new product; (c) product line engineers that identify which aspects in the product line level are related to the new product; (d) software analysts that analyse existing product members and identify the commonality and differences between existing product members and the new product; and (e) software developers that design the new product by reusing parts of existing product members and specifying new aspects of the product being developed.

Table 6: Documents and traceability relations for scenario 1

	Feature Model	Use Case (PM_1)	Use Case (PM_2)	Class Diagram (PM_2)	Sequence Diagram (PM_2)
Use Case (PM_1)	Contains		Similar Different		
Use Case (PM_2)	Contains	Similar Different			
Class Diagram (PM_2)		Satisfies Implements Refines	Satisfies Implements Refines		
Sequence Diagram (PM_2)		Satisfies Implements Refines	Satisfies Implements Refines	Refines Contains	
Statechart Diagram (PM_2)		Satisfies Implements Refines	Satisfies Implements Refines	Contains	Refines

For this scenario, suppose the situation in which the product line in an organization contains product member PM_2 and the organization wants to develop product member PM_1 from our case study. Consider that the requirements of PM_1 have been specified in four different use cases, as shown in Table 5. In order to be able to identify the similarities and differences between PM_1 and PM_2, the parts of PM_1 that can be reused from PM_2, and the parts of PM_1 that need to be developed, it is necessary to compare various documents including product line feature model, use cases of PM_1 and PM_2, and class,

sequence, and statechart diagrams of PM_2. The types of documents to be compared and the relevant traceability relations associated with these documents and relevant to the scenario are shown in Table 6. As in the case of Table 2, the direction of a relation is represented from a row [i] to a column [j] and bi-directional relations appear in two correspondent cells for that relation[10].

As presented in the table, the set of use cases of PM_1 and PM_2 need to be compared with the feature model of the product line in order to support the identification of similarities and differences between the use cases of PM_1 and PM_2. In addition, all class, sequence, and statechart diagrams of PM_2 are compared with the use cases of PM_1 to assist with the identification of which elements of PM_2 design models can be reused. It is also necessary to compare all class, sequence, and statechart diagrams of PM_2 with the use cases of PM_2 to assist with the identification of similarities and differences between the use cases of PM_1 and PM_2. Moreover, the class, sequence, and statechart diagrams of PM_2 need to be compared in order to support the identification of the elements that can be reused when designing PM_1.

Scenario 2: Creation of a product line system from already existing product members

In this case, traceability relations can be used to support the identification of variable and common aspects of existing product members in order to create a product line. The stakeholders involved in this scenario are product managers that identify which aspects of the product members should be part of the product line; and product line engineers, software analysts, and software developers that design and develop the documents at the product line level.

Table 7: Documents and traceability relations for scenario 2

	Use Case (PM_1)	**Use Case (PM_2)**	**Class Diagram (PM_1)**	**Sequence Diagram (PM_1)**	**Statechart Diagram (PM_1)**	**Class Diagram (PM_2)**	**Sequence Diagram (PM_2)**	**Statechart Diagram (PM_2)**
Use Case (PM_1)		Similar Different						
Use Case (PM_2)	Similar Different							
Class Diagram (PM_1)	Satisfies Implements Refines	Satisfies Implements Refines				Similar Different		
Sequence Diagram (PM_1)	Satisfies Implements Refines	Satisfies Implements Refines	Refines Contains			Refines Contains	Similar Different	
Statechart Diagram (PM_1)	Satisfies Implements Refines	Satisfies Implements Refines	Contains	Refines		Contains	Refines	Similar Different
Class Diagram (PM_2)	Satisfies Implements Refines	Satisfies Implements Refines	Similar Different					
Sequence Diagram (PM_2)	Satisfies Implements Refines	Satisfies Implements Refines	Refines Contains	Similar Different		Refines Contains		
Statechart Diagram (PM_2)	Satisfies Implements Refines	Satisfies Implements Refines	Contains	Refines	Similar Different	Contains	Refines	

For this scenario, suppose the situation in which an organisation has product members PM_1 and PM_2

[10] This will also be the case for tables 7, 8, 9, and 10.

from our case study and would like to create a product line that composes these two members. The types of documents to be compared and the relevant traceability relations associated with these documents and relevant to the scenario are shown in Table 7. In this case, all the domain analysis and design models of product members PM_1 and PM_2 need to be compared in order to assist with identification of the information represented at the product line level.

Scenario 3: Changes to a product member in a product line system

In this scenario, traceability relations can be used to support the analysis of the implications of changes in the system. The stakeholders involved in this scenario are software analysts that specify changes to be made in a design part of a product member and, together with software developers, identify the effects of these changes in the other related design software artefacts.

For this scenario, supposed the situation in which an organisation has a product line for mobile phones with product members PM_1 and PM_2 from our case study, and that changes are made to product member PM_1. Therefore, it is necessary to evaluate how these changes will affect the other design models of PM_1 and if these changes also affect the other product members in the product line that may be related to the changes (PM_2 in this scenario). The types of documents to be compared and the relevant traceability relations associated with these documents are shown in Table 8. As shown in the table, all the design models of PM_1 and PM_2 are compared in order to assist with the identification of information that may be affected by the changes.

Table 8: Documents and traceability relations for scenario 3

	Class Diagram (PM_1)	**Sequence Diagram (PM_1)**	**Statechart Diagram (PM_1)**	**Class Diagram (PM_2)**	**Sequence Diagram (PM_2)**	**Statechart Diagram (PM_2)**
Class Diagram (PM_1)		Overlaps	Overlaps	Similar	Overlaps	Overlaps
Sequence Diagram (PM_1)	Depends_on Overlaps Refines Contains		Overlaps	Depends_on Overlap Refines Contains	Similar	Overlaps
Statechart Diagram (PM_1)	Depends_on Overlaps Contains	Overlaps Refines		Depends_on Overlaps Contains	Overlaps Refines	Similar
Class Diagram (PM_2)	Similar	Overlaps	Overlaps		Overlaps	Overlaps
Sequence Diagram (PM_2)	Depends_on Overlaps Refines Contains	Similar	Overlaps	Depends_on Overlaps Refines Contains		Overlaps
Statechart Diagram (PM_2)	Depends_on Overlaps Contains	Overlaps Refines	Similar	Depends_on Overlaps Contains	Overlaps Refines	

Scenario 4: Changes at the product line level

In this case, we are interested in investigating how traceability relations can be used to support evolution and analysis of the impact of the changes at the product line level. More specifically, this scenario is concerned with changes at the product line level due to the addition of new features to the product line system. The stakeholders involved in this scenario are market researchers that identify new features of the

system and product line engineers that identify which aspects in the product line level are related to the new features and the effect of these new features to the other artefacts at the product line level. The types of documents to be compared and the relevant traceability relations for this scenario are shown in Table 9.

Table 9: Documents and traceability relations for scenario 4

	Feature Model	**Subsystem Model**	**Process Model**
Subsystem Model	Satisfies Depends_on Refines		
Process Model	Satisfies Depends_on Refines	Refines	
Module Model	Satisfies Depends_on Refines		Refines

Scenario 5: Impact of changes at the product line and product member levels

In this case, we are interested in investigating how traceability relations can be used to support the impact of changes made at the product line level to product members. This is a small scenario and is concerned with changes at the subsystem of a product line and the impact of these changes in a class diagram. The stakeholders involved in this scenario are product line engineers that identify the changes to be made at the subsystem, and software analysts and developers that identify the effect of these changes to the product member design documents.

For this scenario, consider the situation in which we want to analyse the impact of changes at the subsystem model to the class diagram of product member PM_3 from our case study[11]. The types of documents to be compared and the relevant traceability relations for this scenario are shown in Table 10.

Table 10: Documents and traceability relations for scenario 5

	Class Diagram
Subsystem Model	Contains

We have evaluated the five scenarios by measuring the *precision* and *recall* of the relevant traceability relations generated by XTraQue. We have used the following standard definition of recall and precision given in [22]:

$$\text{Precision} = |ST \cap UT| / |ST| \qquad \text{Recall} = |ST \cap UT| / |UT|$$

where

- ST is the set of traceability relations detected by XTraQue;
- UT is the set of traceability relations which are identified by the user, and
- $|X|$ denotes the cardinality of a set X (viz. $|ST \cap UT|$, $|ST|$, and $|UT|$)

In the experiments we deployed a total of 63 traceability rule templates that have been instantiated depending on the documents used in each experiment and the traceability relations to be identified. These

[11] Although changes at the subsystem model can also have impact on other documents at the product member design level (sequence and statechart diagrams), in this experiment we only analyse the relations between subsystem models and class diagram.

traceability rule templates have been created, by using an evolutionary process, by two software engineers with knowledge of product line systems and mobile phones. Table 11 shows, for each experiment, a summary of the number of documents, number of files, number of (direct and indirect) traceability rule templates, and number of (direct and indirect) instantiated rules[12]. The high number of instantiated direct traceability rules in scenarios 1, 2, and 3 is due to the different types of documents, number of files, and number of traceability relations used in these scenarios.

Table 11: **Summary of documents and traceability rules used in the experiments**

	Scenario 1	**Scenario 2**	**Scenario 3**	**Scenario 4**	**Scenario 5**
No. of documents	15	20	12	6	2
No. of files	10	10	2	6	2
No. of direct traceability rule templates	17	15	11	7	2
No. of indirect traceability rule templates	8	11	5	0	0
Total no. of traceability rule templates	25	26	16	7	2
No. of instantiated direct traceability rules	100	192	80	11	2
No. of instantiated indirect traceability rules	8	11	5	0	0
Total no. of instantiated traceability rules	108	203	85	11	2
Legend: Scenario 1: Creation of a new product member for an existing product line; Scenario 2: Creation of a product line system form already existing product members; Scenario 3: Changes to a product member in a product line system; Scenario 4: Changes at the product line level; Scenario 5: Impact of changes at the product line and product member levels					

Table 12 shows the results of our experiments for each scenario in terms of recall and precision rates, including the number of direct and indirect traceability relations identified by the users and by the tool. The traceability relations generated by the tool in each different scenario were compared against traceability relations manually identified by users with substantial experience and training in software engineering and product line engineering. The results shown in Table 12 provide positive evidence about our approach to automatic generation of traceability relations with a high level of precision and recall.

As shown in Table 12, for scenarios 4 and 5, the cells corresponding to the number of indirect traceability relations detected and the number of valid indirect traceability relations detected by XTraQue contain value zero (0) since these scenarios do not deal with indirect traceability relations (see Tables 9 and 10), and not because the tool cannot generate these types of traceability relations. The high number of traceability relations detected in scenarios 1 and 2 is due to the number of document types and traceability relation types used in these scenarios, as well as the specific documents that are related through the various relation types. For instance, in scenario 1, there are (a) four use case documents for PM_1 and four use case documents for PM_2 that are related in terms of three different types of traceability relations (satisfies, implements, and refines) with one class diagram, four sequence diagrams, and one statechart diagram; (b) four use case documents for PM_1 and four use case documents for PM_2 that are related in terms of contains relations with feature model; (c) four use case documents of PM_2 that are related to four use case documents of PM_1 in terms of similar and different relations; (d) four sequence diagrams of PM_2 that

[12] The number of documents is different than the number of files because of XMI representation for UML diagrams.

are related in terms of refines and contains relation types with one class diagrams and in terms of refines relation with one statechart diagram; and (e) one class diagram that is related in terms of contains relations with one statechart diagram. A similar and more complex situation occurs in scenario 2.

The results in Table 12 show that direct traceability relations have higher precision and recall values when compared to indirect traceability relations for scenarios 1, 2, and 3. This is due to the fact that indirect traceability relations are generated based on direct traceability relations and, in the cases of incorrect direct traceability relations generated by the tool, or missing direct traceability relations by the tool, these will interfere with the precision and recall of the indirect traceability relations. More specifically, the incorrect and missing direct traceability relations will cause a lower precision, while the missing direct traceability relations will contribute to a lower recall.

Table 12: Measure of Recall and Precision Rates

		Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Average
No. of direct traceability relations detected	***By the users*** **|UT|**	519	1076	128	26	6	-
	By XTraQue **|ST|**	525	1090	136	21	6	-
|ST ∩ UT| for direct traceability relations		502	1046	112	17	5	-
No. of indirect traceability relations detected	***By the users*** **|UT|**	333	1412	126	0	0	-
	By XTraQue **|ST|**	341	1418	130	0	0	-
|ST ∩ UT| for indirect traceability relations		282	1208	105	0	0	-
Total no. of traceability relations detected	***By the users*** **|UT|**	852	2488	254	26	6	-
	By XTraQue **|ST|**	866	2508	266	21	6	-
|ST ∩ UT| for all traceability relations		784	2254	217	17	5	-
Precision	***Direct relations***	0.956	0.959	0.823	0.81	0.834	0.876
	Indirect relations	0.827	0.852	0.807	-	-	0.828
	All relations	0.905	0.898	0.816	0.81	0.834	**0.853**
Recall	***Direct relations***	0.967	0.972	0.875	0.654	0.834	0.860
	Indirect relations	0.847	0.855	0.834	-	-	0.845
	All relations	0.920	0.906	0.854	0.654	0.834	**0.833**

Legend: Scenario 1: Creation of a new product member for an existing product line;

Scenario 2: Creation of a product line system form already existing product members;

Scenario 3: Changes to a product member in a product line system;

Scenario 4: Changes at the product line level;

Scenario 5: Impact of changes at the product line and product member levels

The results also show that scenario 4 has the lowest recall when compared with the other scenarios. We attribute this to the fact that this scenario has the lowest number of direct rule templates with respect to the different types of traceability rules used in the scenario when compared to scenarios 1, 2, and 3 (scenario 5 is quite a small scenario to be considered in this case). For example, scenario 1 has 17 direct rule templates for four different types of direct relations, scenario 2 has 15 rule templates for four different types of direct relations, and scenario 3 has 11 rule templates for four different types of direct relations, while scenario 4 has only seven rule templates for three different types of relations. Therefore, the number of direct traceability relations generated by XTraQue for scenario 4 is smaller than the number of traceability relations identified by the users. In the other scenarios we observe an inversion of this situation (i.e.,

number of direct traceability relations generated by the tool is higher than the ones generated by the users). The addition of new traceability rules for satisfies, depends_on, and refines relations for documents at the product line level will cause an increase in the recall measurements.

Overall, the average precision measured (i.e., 85.3%) and average recall measured (i.e. 83.3%) in our experiments are encouraging results. Although the data sets used in our work are different from the data sets used in other approaches that support automatic generation of traceability relations [1][31][45], our precision results are better than the results achieved in those approaches, while our recall results are comparable to the results achieved in those approaches. The results achieved in this paper are also better than the results of our previous work for automatic generation of traceability relations between textual documents representing requirements, use cases, and analysis models of software systems [72]. In order to increase the recall results of our approach, new traceability rules need to be created to support the identification of missing relations.

Although our approach relies on the use of XML documents, our experience has demonstrated that the creation of these documents is not an issue since many application tools use XML as a standard export format to support data interchange among heterogeneous tools and applications. A possible drawback of our work is concerned with the extra effort to mark-up textual parts of the documents with XML POS tag elements. However, this is alleviated by the use of tools like CLAWS [14] and our converter that transforms the POS tags identified by CLAWS into XML elements representing these tags. Our experience has demonstrated that the time spent to mark-up textual parts of the documents is not substantial and once the documents are marked-up they can be used to support different scenarios and situations. Moreover, the POS tags allow the approach to make use of rules that take into consideration the grammatical roles of the words in a textual description and to consider these descriptions when tracing different elements. We believe that the above are important when dealing with documents that contain natural language sentences and descriptions of their elements as in the case of feature models, subsystem models, process models, module models, and use cases.

Another issue of the work is concerned with the creation of traceability rules, which require knowledge of XQuery and understanding of the semantics of the documents, the various types of relations, and the grammatical roles of the words in the textual parts of the documents by the rule editor. Moreover, new traceability rules need to be created for applications that use different types of documents. However, once a set of rules is created, these rules can be used in different applications that use the same types of documents. The XTraQue tool offers support for editing and testing new traceability rules. In addition, in a previous work [71], we have proposed a machine learning algorithm to support the creation of new traceability rules to generate traceability relations that existing rules failed to identify between requirements and object-oriented specifications. We plan to extend this work to support the generation of new traceability rules in the scope of product line engineering. Moreover, changes in the documents require the traceability generation process to be re-executed. However, as explained in Section 4, our tool supports

users to specify which documents to be traced and which traceability relations to be generated, avoiding the process to be executed for all document types and traceability relation types. In addition, only the modified parts of a document will need to be marked-up again with the POS tags. We are currently extending our work to allow for the automatic identification of the parts of the documents that have been changed and the generation of traceability relations for the modified parts.

6. Related Work

The work presented in this paper is a large extended version of our conference paper [32]. In this previous publication we present some initial ideas of the work in which we give an account of the approach, an initial version of the traceability reference model, and few examples of traceability rules. In contrast, our main contributions in the current paper are (a) thorough evaluation of the work in terms of recall and precision based on five different scenarios concerned with product line engineering; (b) description of the approach in details; (c) presentation of a revised and detailed version of the traceability reference model in which we define the various types of traceability relations and present examples and explanation of all document types of our concerned; (d) detailed description of the traceability rules with the various extra functions used in our approach; (e) description of XTraQue tool implementation issues and the mobile phone case study used in our work; and (f) a more complete account of the related work.

There have been other approaches and techniques to support software traceability as presented in the survey in [66]. These approaches and techniques can be classified in four main groups: (a) study and definition of different types of traceability relations; (b) support for generation of traceability relations; (c) development of architectures, tools, and environments for representing and maintaining traceability relations; and (d) study of how to use traceability relations to support software development activities.

Among these approaches, various reference models, frameworks, and classifications have been proposed for different types of traceability relations [5][27][53][60][66]. The classifications are based on different aspects, ranging from the types of related artefacts [53], to the use of traceability information in different requirements management activities such as understanding, capture, tracking, evolution, verification, and reuse [5][16][27], to impact analysis [77]. Some approaches have proposed different types of traceability relations that associate requirements specifications [5][20][28][42][48][53][60][72][78], while other approaches suggest relation types between requirements and design specifications [12][13][20][42][60], and between code specifications and requirements and design artefacts [1][20][45][60].

In [53], the authors proposed a classification for traceability relations that include 18 different types of relations for requirements specifications organised in five groups, namely (i) *condition link group*, (ii) *content link group*, (iii) *documentation link group*, (iv) *evolutionary link group*, and (v) *abstraction link group*. In [66], the authors organised all the different types of traceability relations proposed in the literature into eight main types: *dependency, generalisation/refinement, evolution, satisfaction, overlap, conflicting, rationalisation*, and *contribution* relation types. The reference model proposed in [60] is based

on the use of metamodels to represent traceability information including elements to be traced and types of relations between these elements such as *traces-to* links, *manages* links, *documents* links, and *has-role-in* links. Extensions of this reference model have been proposed for workflow management systems [77], product and service families [48], and UML-based systems [42].

Despite the reference models and classifications that have been proposed in the literature, there is still a lack of standard semantic definition for the various types of relations [66]. Many existing tools support the representation of the different types of relations, but the interpretation of these relations depends on the stakeholders. This causes confusion when interpreting relations and difficulties in developing tools for automatic generation of traceability relations. Moreover, few classifications for traceability relations in the scope of product line engineering have been proposed. Exceptions are found in [5][36][48]. However, these approaches do not provide ways of generating traceability relations automatically. The classification of the traceability relations described in this paper contributes to fulfil the existing lack of a more precise semantic for traceability relations, in particular in the product line domain. Moreover, XTraQue provides support for generating the traceability relations automatically and for interpreting the relations.

Approaches to support the generation of traceability relations can be classified in three groups depending on the level of automation, namely (a) manual, (b) semi-automatic, and (c) fully-automatic. The majority of existing commercial traceability tools offer support for manual generation of traceability relations based on the use of sophisticated visualisation, display, and navigability components, such as RETH[33], DOORS[19], RTM[65], and RDT [63]. In these tools, the users are expected to identify and select the elements to be traced. However, manual creation of traceability relations are error-prone, difficult, time consuming, and expensive, resulting in the rare deployment of traceability relations.

In order to alleviate the above problems, approaches that support semi-automatic [15][16][21][25][54] and fully-automatic [1][25][30][45][46][57][59][66][72] generation of traceability relations have been proposed. In the semi-automatic group of approaches, traceability relations can be generated based on previous relations defined by users [15][16][21], or as a result of the software development process [54]. Although the semi-automatic approaches can be considered as an improvement when compared with the manual approaches, the initial identification of the user relations may still be error-prone, time-consuming, and expensive. The software development process-based approaches depend on how systems are developed.

The fully-automatic approaches make use of information retrieval (IR) techniques [1][30][31][45][46], traceability rules [60][72], axioms [58], and special types of integrators [66]. The approach in [1] has been proposed to support overlap traceability relation generation between requirements document and source code when the document matches a query extracted from the source code, based on the use of *probabilistic* and *vector space* IR techniques. The approach assumes that the vocabulary of the source code overlaps various elements in the requirements documents. Experimental results of this work have achieved low levels of precision and reasonable levels of recall. The work in [30] suggests a reduction in the number of

missed and irrelevant traceability relations by using classical vector IR model techniques extended with the use of key-phrase lists or a thesaurus. Experimental studies have demonstrated that the use of key-phrase lists can improve recall, but decreases precision, while the use of a thesaurus increases recall and, sometimes, it also increases precision. An extension of the work in [30] has been described in [31] in which the authors compare the use of IR techniques such as *vector retrieval*, *vector retrieval with thesaurus*, and *latent semantic indexing* to improve the quality of the generation of traceability links between requirements-to-requirements and requirements-to-design traceability. The results of their study have demonstrated that vector retrieval techniques outperformed latent semantic indexing. The work in [45][46] also uses *latent semantic indexing* to support traceability generation between different types of system artefacts and source code. In this approach, a corpus is built based on pre-processing of documents and source code and a traceability relation is established when the semantic similarity measure of the documents is greater than a threshold. The recall and precision results in this approach are better than the results in [1].

Our work does not make use of IR techniques to support traceability generation, but is built upon our previous work in [72], which supports generation of traceability relations between requirements, use cases, and object model documents based on traceability rules. In this previous work, we identify only three different types of traceability relations concerned with object-oriented development called *overlaps*, *requires*, and *realises*. In the work presented in this paper, we focus on automatic generation of traceability relations concerned with product line systems and different types of documents created during the domain analysis and design of such systems. Moreover, in this paper we describe many different types of traceability relations. In [72], the rules are specified in an XML-based language, while in this paper the traceability rules are represented in an extension of XQuery, which has been proposed as a language to manipulate and retrieve information from XML documents. Furthermore, the traceability rules in [72] specify ways of matching syntactically related terms in the textual parts of a requirements statement or use case with related elements in the object model, and matching requirements statements and use cases. The work presented in this paper extends the matching of the traceability rules and uses rules that also take into account the semantic of the documents, the various types of traceability relations in the product line domain, and the distance of the words in a text. The precision and recall measures achieved in the work presented in this paper are higher than the measures achieved in [72].

The work in TOOR [57] uses axioms to support generation of traceability relations between requirements, design, and code specifications and supports the derivation of additional relations from the axioms by transitivity, reflexivitive, symmetry, extraction, and dependency. Like our work, TOOR allows users to specify traceability relations of different types and define axiomatically their semantics.

Approaches to support representation and maintenance of traceability relations range from the use of centralised databases [19][58][65] and software repositories [53], open hypermedia architecture [66], mark-up based documents [43][72], to event-based architecture [16]. As in [43][72], XTraQue represents the

documents to be traced and the generated traceability relations as XML documents, avoiding changes in the original documents. The use of XML documents to represent traceability relations also allows re-use of the approach to support generation of traceability relations based on existing relations; i.e., indirect relations.

Software traceability has been used in different stages of the software development life-cycle to support various activities. Examples of these activities are change impact analysis, system validation and verification, system reuse, and system understanding [5][7][26][17][18][16][49][39][77].

The Goal Centric Traceability approach proposed in [17] supports impact analysis of changes in functional requirements with respect to non-functional requirements by using a probabilistic model. Initial experimental results of this work have demonstrated a recall measure of over 85% for all non-functional requirements represented as softgoals, and precision measures between 40-60%. Another goal-oriented approach to support understanding roles and contributions of stakeholders and their relationships for dynamic adaptive systems (DAS) has been proposed in [26]. This work uses KAOS specification language to define different levels of requirements engineering for DAS and defines three relationship types between elements in these levels, namely *subset refinement*, *scenario refinement*, and *adaptation refinement*.

In [18], the authors proposed a traceability technique named "retrieval by construction" to support verification and validation of UML formalization. More specifically, the approach establishes traceability relations between a UML model and a target model representing UML semantics by using generative procedures. A generative procedure determines elements that should be generated in a target model in order to formalise an element in UML model. In this approach, an inconsistency between the source and target models exists when the endpoints of the relations are unexpected. An extension of this work was presented in [73] in which a graph-theoretic model for formally defining the problem of associating UML models with related code has been proposed. The work in [49] describes ArchTrace, a tool to support automatic evolution of traceability links between architecture models and source code. The approach uses policies to assist with the addition of new links, removal of existing links, and changes in existing links.

The use of traceability to support product line engineering has been advocated in [5][7][36][38][48][55][64][76]. In [5], the authors advocate that traceability is a key technology for product line infrastructure. Similar to our work, the authors share the opinion that giving traceability relations semantic meanings is important to increase the usefulness of traceability and propose traceability support to the PuLSE[6] method based on a metamodel for three stages of product line development namely (a) scoping, involving features, product, and documented product map; (b) architecture, involving components, class, interface, and data; and (c) implementation, involving property, code module, and property file. The authors also affirm that it is important to have ways of identifying the above types of traces in an automated way, although do not specify how this can be achieved. The work in [36] promotes the use of treaceability as the foundation for automating product line engineering process and proposes different types of traceability links and associated mapping rules representing ways of deriving and creating product line artefacts at a certain level of abstraction from other product line artefacts in different levels of abstraction.

In our work, the traceability rules are not restricted to mapping rules between different levels of abstractions, but rules to create traceability relations that can be used to assist with different activities such as identification of common and variable functionality, reduction of inconsistencies between product members, reuse of core assets, maintenance of historical information of the development process, evolution of software systems, validation that a system meets its requirements, understanding of the rationale for certain design and implementation decisions, and analysis of the implications of changes in the system.

In [48], the authors proposed a traceability framework and a knowledge management system for managing traceability of product and service families. The framework is based on an extension of the reference model proposed in [60]. The reference model is mainly concerned with artefacts and traceability relations for supporting the identification of common and variable requirements and includes architectural decisions and configurations resulting from the design objects/components created during development phase. The knowledge management system is based on REMAP[61], a web-based tool for capturing and maintaining traceability information. Another tool for managing traceability of product families was proposed in [38]. This approach is restricted to feature and product component maps and consists of an extension of a commercial software tool to allow the description of features in an unambiguous and manageable way, selection and management of features, tracing of cross-cutting features, and modelling newly developed and COTS components. However, the tool does not offer support for automatic identification of traceability relations.

Based on the above overview of existing approaches to support software traceability, in general, and in product line systems, in particular, the novelty of our work is concerned with the automatic generation of different types of traceability relations. Please note that existing information retrieval techniques to support automatic generation of traceability relations do not allow for the identification of multiple traceability relation types. Moreover, our work supports the generation of traceability relations in various types of documents representing different levels of the development life-cycle of product line systems. Furthermore, the traceability relations introduced by our approach have defined semantics between the artefacts being compared and can be used to support different activities and stakeholders involved in the product line engineering.

7. Conclusion and Future Work

In this paper, we described a traceability reference model and a rule-based approach to support generation of traceability relations between feature-based object-oriented documents, concerned with product line engineering. Our approach can generate nine types of traceability relations including *satisfiability, dependency, overlaps, evolution, implements, refinement, containment, similar,* and *different* relations between feature-based and object-oriented documents created during the development of product line systems such as models, subsystem models, process models, module models, use cases, class, diagrams, sequence diagrams, and statechart diagrams. These traceability relations have semantic meanings and directions instead of being simple links between different elements. Other novelties in our work are the use

of an extension of XQuery to represent traceability rules and the use of rules that take into consideration the (i) semantic of the documents being compared, (ii) various types of traceability relations in the product line domain, (iii) grammatical roles of the words in the textual parts of the documents, and (iv) synonyms and distance of words being compared in a text.

The work has been evaluated in terms of precision and recall in five different scenarios. These scenarios involve different ways of developing product line systems and include (a) the creation of a new product member for an existing product line, (b) the creation of a product line system from already existing product members, (c) changes to a product member in a product line system, (d) changes at the product line level, and (e) impact of changes at the product line level to a product member. The results of these experiments have shown an average precision of 85.3% and an average recall of 83.3%. These results are comparable to other approaches that support automatic generation of traceability relations between requirements specifications and source code[1], requirements, use cases, and object models [72], and requirements and design models [31].

Currently, we are extending the work to support traceability for domain implementation phase. We are also investigating ways of visualising the various traceability relations in order to deal with the scalability of traceability relations and support software engineers with the large number of traceability relations that are generated. Another area of research that we are interested in is concerned with the optimisation of the generation of traceability relations when the documents evolve.

References

[1] Antoniol G., Canfora G., Casazza G., De Lucia A., Merlo E., "Recovering Traceability Links between Code and Documentation", IEEE Transactions on Software Engineering, 28(10), 970-983, 2002

[2] ArgoUML. http://argouml.tigris.org/project.html

[3] ASADAL. selab.postech.ac.kr/form/

[4] Atkinson, C., J. Bayer, et al., “Component-based product line development : The KobrA approch”, the first software product line conference, SPLC, Denver, Colorado, USA, 2000

[5] Bayer J., Widen T., “Introducing Traceability to Product Lines”, Software Product-Family Engineering, the 4th International Workshop, PFE 2001, Spain, October 3-5, 2001, appeared in Lecture Notes in Computer Science, Vol. 2290, Springer 2002.

[6] Bayer, J., O. Flege, et al., “PuLSE: A methodology to develop software product lines”, the fifth ACM SIGSOFT Symposium on Software Reusability (SSR’ 99), Los Angeles, CA, USA

[7] Biddle R., Noble J., and Tempero E., “Supporting Reusable Use Cases”. In Proceedings of the Seventh International Conference on Software Reuse, 2002.

[8] Borland Together. www.borland.com/together/

[9] Bosch, J., “Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach, Addison Wesley, 2000

[10] CAFÉ, http://www.esi.es/en/projects/cafe/cafe.html

[11] Cockburn A.,"Structuring Use-Cases With Goals", JOOP, 1997.

[12] Constantopoulos P, Jarke M, Mylopoulos Y, Vassiliou Y, "The Software Information Base: A Server for Reuse", VLDB Journal, 4(1), 1-43, 1995

[13] CORE, http://www.vtcorp.com

[14] CLAWS. http://www.comp.lancs.ac.uk/ucrel/claws.

[15] Cleland-Huang J., Schmelzer D., "Dynamic Tracing Non-Functional Requirements through Design patter Invariants", Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003), Canada, October, 2003.

[16] Cleland-Huang J., Chang C.K., Sethi G., Javvaji K., Hu H., Xia J., "Automating Speculative Queries through Event-based Requirements Traceability", proc. of the IEEE Joint International Requirements Engineering Conference, Essen, Germany, September 2002.
[17] Cleland-Huang J., Settimi R. and BenKhadra O., "Goal-Centric Traceability for managing Non-Functional Requirements", International Conference on Software Engineering, USA, May 2005.
[18] Deng M., Stirewalt R.E.K. and Cheng B.H.C., "Retrieval by Construction: A Traceability Technique to Support Verification and Validation of UML Formalization", International Journal of Software Engineering and Knowledge Engineering, 15(5) 2005
[19] DOORS., www.telelogic.com/products/doors.
[20] Egyed A., "A Scenario-Driven Approach to Trace Dependency Analysis", IEEE Transactions on Software Engineering, Vol.9, No.2, February 2003.
[21] Egyed A., Gruenbacher P., "Automatic Requirements Traceability: Beyond the Record and Replay paradigm", Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE), Edinburgh, UK, September, 2002.
[22] Faloutsos C., Oard D., "A Survey of Information Retrieval and Filtering Methods", Tech. Report CS-TR3514, Dept. of Computer Science, Univ. of Maryland, 1995.
[23] Fantechi, A., S. Gnesi, et al., "A Methodology for the Derivation and Verification of Use Cases for Product Lines", SPLC, pp. 255-265, 2004
[24] FODA. Feature Oriented Domain Analysis. www.sei.cmu.edu/domain-engineering /FODA.html
[25] GCT. Proceedings of the International Symposium of the Grand Challenges for Traceability, Kentucky, March 2007 (http://traceabilitycenter.org/events/TEFSE07).
[26] Goldsby H. and Cheng B.H.C., "Goal-Oriented Modeling of Requirements Engineering for Dynamically Adaptive Systems", 14th IEEE Int. Requirements Engineering Conference, USA, 2006.
[27] Gotel O. and Finkelstein A., "An Analysis of the Requirements Traceability Problem", First Int. Conf. on Requirements, 1994.
[28] Gotel O., Finkelstein A., "Contribution Structures", Proceedings of 2nd International Symposium on Requirements Engineering, (RE '95), 100-107, 1995.
[29] Griss M.L., Favaro J., d'Alessandro M., "Integrating Feature Modeling with the RSEB", Proceedings Fifth International Conference on Software Reuse", 1998.
[30] Hayes J.H., Dekhtyar A., Osborne J., "Improving Requirements Tracing via Information Retrieval", proceedings of the 11th IEEE Int. Requirements Engineering Conference, Monterey Bay, 2003.
[31] Hayes J.H., Dekhtyar A., Sundaram S.K., "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods", IEEE Transaction on Software Engineering, V. 32, No. 1, 2006.
[32] Jirapanthong W. and Zisman A., "Supporting Product Line Development through Traceability", Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005),Taiwan, 2005.
[33] Kaindl H., "The Missing Link in Requirements Engineering", Software Engineering Notes, June 1992.
[34] Kang, K., S. Cohen, et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Pittsburgh, PA, Software Engineering Institue, Carnegie Mellon University.
[35] Kang, K., Kim, S., et al., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Architectures", Annals of Software Engineering 5(1): 143-168.
[36] Kim S.D, Chang S.H., and La H.J., "Traceability Map: Foundations to Automate for Product Line Engineering", 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA05).
[37] Krueger C.W. "Software Mass Customization",http://www.biglever.com/papers/BigLeverMassCustomization.pdf
[38] Lago P., Niemela E., and Vilet H.V., "Tool Support for Traceable Product Environment", In Proc. of the 8th European Conference on Software Maintenance and Reengineering (CSMR), Finland, 2004.
[39] Lavazza L, Valetto G, "Requirements-based Estimation of Change Costs", Empirical Software Engineering - An International Journal, 5(3), November 2000
[40] Lee K., Kang K.C., Chae W., and Choi B.W.,"Feature-based Approach to Object-Oriented Engineering of Applications for Reuse", Software-Practice and Experience, 2000, 30:1025-1046.
[41] Leech G., Garside R., and Bryant M., "CLAWS4: The Tagging of the British National Corpus", In Proceedings of the 15th International Conference on Computational Linguistics (COLING 94), Kyoto, Japan, 622-628, 1994.

[42] Letelier P., "A Framework for Requirements Traceability in UML-based Projects", proceedings of the 1st International Workshop on Traceability for Emerging Forms of Software Engineering (TEFSE'02), Edinburgh, UK, September 2002.
[43] Maletic J.L., Collard M.L., and Simoes B., "An XML Based Approach to Support the Evolution of Model-to-Model Traceability Links", 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05), California, September 2005.
[44] Mannion, M., et al., "Representing Requirements on Generic Software in an Application Family Model", ICSR6, LNCS 1844, pp. 153-169, 2000
[45] Marcus A., Maletic J.I., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", ICSE, 2003.
[46] Marcus, A., Maletic, J.I., Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", International Journal of Software Engineering and Knowledge Engineering, Vol. 15, No. 4, October 2005, pp. 811-836.
[47] Meyer, B., "Object Oriented Software Construction", Prentice Hall, 1998.
[48] Mohan K., Ramesh B., "Managing variability with Traceability in product and Service Families", In proceedings of the 35th Hawaii International Conference on System Sciences, IEEE, 2002.
[49] Murta L.G.P, van der Hoek A. and Werner C.M.L., "ArchTrace: Policy-Based Support for managing Evolving Architecture-to-Implementation Traceability Links", 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Japan, September 2006.
[50] Nokia. http://www.forum.nokia.com/main.html.
[51] OMA. www.omg.org/technology/documents/formal/xmi.htm.
[52] OMG. XML Metadata Interchange (XMI). www.omg.org/technology/documents/formal/xmi.htm.
[53] Pohl K., "Process-Centered Requirements Engineering", John Wiley & Sons, Inc., 1996.
[54] Pohl K, PRO-ART: Enabling Requirements Pre-Traceability, Proceedings of the IEEE Int. Conference on Requirements Engineering (ICRE 1996).
[55] Pohl K. et al., "Product Family Development", Dagstuhl Seminar Report No.304, http://www.dagstul.de/01161/report, 2001.
[56] Poritz A.B., "Hidden Markov Models: A Guide Tour". In Proceedings of International Conference on Acoustics, Speech and Signal Processing, Vol. I, 1998, 7-13, New York, IEEE.
[57] Pinheiro F., "Formal and Informal Aspects of Requirements Tracing", Position Paper in Proceedings of 3rd Workshop on Requirements Engineering (III WER), Rio de Janeiro, Brazil, 2000.
[58] Pinheiro F., Goguen J., "An Object-Oriented Tool for Tracing Requirements", IEEE Software, 52-64, March 1996.
[59] Ramesh B., Dhar V., "Supporting Systems Development Using Knowledge Captured During Requirements Engineering", IEEE Transactions in Software Engineering, June 1992, 498-510, 1992.
[60] Ramesh B. and Jarke M., "Towards Reference Models for Requirements Traceability", *IEEE Transactions on Software Engineering* , Vol. 37, No 1. January 2001.
[61] Ramesh B. and Tiwana A. "Supporting Collaborative Process Knowledge Management in New Product Development Teams", Decision Support Systems, Vol. 27, pp. 213-235, 1999.
[62] Rational Rose. www.306.ibm.com/software/rational/.
[63] RDT, http://www.igatech.com/rdt/index.html
[64] Riebisch M., Plilippow I., "Evolution of Product Lines Using Traceability", OOPSLA 2001 Workshop on Engineering Complex Object-Oriented Systems for Evolution, Florida.
[65] RTM. Integrated Chipware. www.chipware.com.
[66] Sherba S.A., Anderson K.M., and Faisal M., "A Framework for Mapping Traceability Relationships", Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003), Canada, September 2003.
[67] Simpson T.W., "A Concept Exploration method for Product Family Design", in Mechanical Engineering, Atlanta, 1998.
[68] Sinnema, M., et al., "COVAMOF: A Framework for Modeling Variability in Software Product Families", the third international conference, SPLC, 2004
[69] Sourceforge; Saxon: http://saxon.sourceforge.net/
[70] Spanoudakis G. and Zisman A., "Software Traceability: A Roadmap", Handbook of Softawre Engineering and Knowledge Engineering, (V. 3) S.K. Chang, World Scientific Publishing Co., 2003.

[71] Spanoudakis G., Garcez A., and Zisman A., "Revising Rules to Capture Requirements Traceability Relations", 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco, July 2003.
[72] Spanoudakis G., Zisman A., Pérez-Miñana E., and Krause P., "Rule-based Generation of Requirements Traceability Relations", Journal of Systems and Software, Vol 72(2), pp 105-127, 2004.
[73] Stirewalt, K., Deng, M. and Cheng, B.H.C., "UML Formalization is a Traceability Problem", 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05), California, September 2005.
[74] Svahnberg, M. and Bosch, J., "Issues Concerning Variability in Software Product Lines", the third International Workshop on Software Architectures for Product Families, Berlin, Springer Verlag, 2000
[75] Theil, S. and Hein, A.,"Systematic Integration of Variability into Product Line Architecture Design", The 2nd International Conference on Software Product Lines (SPLC2), Springer Verlag, 2002.
[76] Van der Linden, F., "Product Family Development in Philips Medical Systems", Dagstuhl Event 03151, April 2004. www.dagstuhl.de/03151/Titles/index.en.phtml
[77] Von Knethen A., "Automatic Change Support based on a Trace Model", Proceedings of the 1st Int. Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'02), 2002.
[78] Von Knethen A., Paech B., Kiedaisch F., Houdek F., "Systematic Requirements Recycling through Abstraction and Traceability", Proceedings of the IEEE International Requirements Engineering Conference, Germany, September 2002.
[79] Weiss, D. Software Synthesis: The FAST Process, the International Conference on Computing in High Energy Physics (CHEP), Rio de Janeiro, Brazil.
[80] WordNet. http://wordnet.princeton.edu/.
[81] XPath. http://www.w3.org/TR/xpath.
[82] XQuery. http://www.w3.org/TR/xquery/.
[83] XTraQue. XTraQue Project. http://www.soi.city.ac.uk/~zisman/XTraQue.