Using Critiquing Systems for Inconsistency Detection in Software Engineering Models

Cleidson R. B. de Souza[1,2] Hamilton L. R. Oliveira[2] Cleber R. P. da Rocha[2] Kleder M. Goncalves[2] David F. Redmiles[1]

[1]School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA

[2]Informatics Department
Federal University of Pará
Belém, PA, Brazil

Abstract

Many approaches have been proposed for consistency management of software engineering documents and specifications. A few others have been proposed to check consistency among software engineering models. For example, abstract state machines, knowledge-based approaches and so on. In this paper, we apply a different technique that uses critiquing systems. A critiquing system monitors user's actions and triggers a signal when one of those actions activates pre-specified rules, called critics. Because critics are small, we argue that they might be used to address two open issues in inconsistency detection, namely efficiency and scalability. An example of this approach is presented to check domain engineering models (feature diagrams) and application engineering models (class diagrams). Feature diagrams are used to abstractly and concisely express commonality and variability across a domain. These diagrams are used as source of information in the generation of critics in UML class diagrams. We present an environment, called DAISY that uses three different critiquing systems to demonstrate the feasibility of our approach.

***Keywords**: Inconsistency detection, consistency management, critiquing systems, domain engineering, application engineering.*

1. Introduction

Developing software systems is a complex task where participation and collaboration of a large number of stakeholders (e.g. customers, users, analysts, designers, and developers) is necessary to succeed. Often, these stakeholders create different models of the software being developed in order to properly model, understand, design, and evaluate it. However, these software models can be inconsistent with each other since they describe the system under different points of view, and these perspectives reflect the interests, background and skills of these different stakeholders. Although inconsistent models can have positive effects in software development [21], in general it is not desirable to preserve this inconsistency. Indeed, different solutions have already been proposed to this problem, such as model-checking, knowledge-based approaches and so on. In this paper we propose the use of critiquing systems to deal with this problem. Critiquing systems help detect suboptimal design situations and possible errors. They monitor the user's actions and activate a signal when any one of those actions violates a rule [5]. In general, a critic is defined by a single rule, or a small set of rules. For example, a kitchen floorplan design environment might contain a "sink-not-in-front-of-window" critic that has the sole job of identifying situations in which the designer has located a sink anywhere but in front of a window [5]. Because critics are small, we argue that they might be used to address two open issues in inconsistency detection, namely efficiency and scalability [21].

In order to demonstrate the feasibility of our approach, we present a critiquing system that is able to check the consistency of models created during domain and application engineering. In other words, constraints defined in the domain diagrams might be used as critics to evaluate the class diagram. This approach reminds designers about domain characteristics that could be forgotten during application engineering. In this paper we also describe DAISY, an environment that supports the construction of domain engineering and application engineering models. It supports consistency checking of these models through critiquing systems. During the development of these models, three critiquing systems are used to identify problematic situations: the first one searches for constructions that violate heuristics about object-oriented design [20]; the second supports the construction of domain models and checking for problems in its construction; finally, a third critiquing system detects potential inconsistencies and other mistakes which might occur in the mapping between these two models.

Section 2 briefly presents the concepts of critics and critiquing systems. The next section describes our approach. After that, we described the concepts of domain and application engineering, and features diagrams in order to be able to present our prototype in section 4. Section 5 describes the DAISY environment, its critiquing systems as well as some implementation details. In the following section, related works are discussed. Finally, section 7 concludes the paper and presents suggestions for future work.

2. Critiquing systems

A critiquing system monitors user's actions and triggers a signal when one of those actions activates the rules of "bad design". For example, in the object oriented modeling, one may have a critic such as: "*A class should not have any public attributes*". In this case, this critic will fire when the user creates a class that has a public attribute. At the time of detection, the critiquing system should present the justification for triggering the critic [5]. This can be done through the presentation of arguments that helps the user to understand the problem and make his decision. The goal in this case is to present a rationale to the user leading him or her to reflect on the problem and change the design accordingly.

Critiquing systems are well suited for complex domains in which the traditional expert systems or automated design approaches only had limited success. Furthermore, they are also appropriate to problem domains with the following characteristics [8][5]: (a) the knowledge about the domain is incomplete and evolving; (b) the requirements of the problem can only be partially specified; and (c) the necessary project knowledge is distributed among many project members. In other words, critiquing systems are a good match for many software development activities. In fact, some critiquing systems to object-oriented design have already been created (see section 6.2).

Critiquing systems have often been embedded in domain-oriented design environments, or DODE's [3]. DODE's contain other components such as catalogs with collections of pre-stored designs in the domain; an argumentative hypermedia system, which contains issues, answers, and arguments about the design domain; among other components. In this case, when the user does not understand the critic, he may browse other designs that do not violate the critic, or access the argumentative system in order to learn about the critic.

Furthermore, a critiquing system should have different intervention strategies that determine *when* and *how* the design should be criticized. This is necessary because critics should trigger at the right time to not disturb users [5]. It is also desirable that the critiquing system allows the user to add or modify critics, decide about their behavior, and even disable the ones that he considers unhelpful. Another important characteristic of critiquing systems is that the knowledge that they encode is expansible and modifiable, since critics are usually independent of each other.

Formally, critics are composed by groups of rules or procedures to evaluate different aspects of a product or design. Although simple and sometimes empirical, the knowledge encoded in critics can help in the detection of: possible design mistakes, problematic solutions, inconsistencies between the design and its specifications, and violations of standards among other problems. This can improve the software development process by eliminating errors and also by helping designers to develop better solutions to their problems.

It is important to note that a critic does not need to be formally correct, i.e., it can just point to a possible error. In fact, according to Hägglund [8] critics may even be used to keep a knowledge base with alternative and conflicting solutions for a problem. For instance, in an object oriented design in which an association exists between two classes A and B with "many-to-many" multiplicity, there are two possible implementations: (i) the addition of a link attribute to model the association properties [14]; or (ii) the addition of a new class C, which has two associations: one with the class A and other with B [1]. In the later case, the properties are added as attributes of the new class. Then, this critic may be implemented with a condition clause that identifies "many-to-many" associations, and, in its argumentation, the two possible solutions are described and evaluated. Therefore, new critics can be added to the system at any time and by different authors without the need for checking the consistency of the knowledge base.

Note also that the user also does not need to agree with the critic: he may not accept the solution presented, but he will be aware of the implications of his action.

Finally, it is important to distinguish a critiquing system from other approaches. In a critiquing system, the most important *input* is a solution for a problem (like a kitchen floorplan design [5] or UML class diagrams [16]), where in expert systems and expert advisory systems this solution is computed and reported to the user as the *output* [17].

3. Consistency Checking using Critiquing Systems

As mentioned before, our main goal is to support consistency management in software engineering models. In this paper, we argue that critiquing systems are useful to deal with the main issues during the detection of inconsistencies, namely: efficiency and scalability [21].

We believe that efficiency can be improved by using critiquing systems because, in general, critics are defined by a single rule or a small set of rules. In addition, these rules are small because they only focus on specific parts of the domain supported. For example, one critic could be created to check that in UML class diagrams, the graph formed by inheritance relationships is acyclic. This means that this critic will only need to be checked, if modifications are performed in components associated with this graph. If one changes an aggregation relationship in the model, this critic will not need to be checked. This is an important characteristic that makes critiquing systems more efficient than other knowledge-based approaches, and more appropriate to deal with inconsistencies that occur when a software model evolves.

Because of their limited size and because they are mostly independent of each other, efficiency is not a problem as difficult as in other approaches for detection of inconsistencies such as model checking. This same characteristic is important to minimize scalability problems because the checking effort is proportional to the number of aggregation relationships rather than the overall size of the model.

Finally, because most critics are independent of each other, new knowledge might be encoded in the critiquing system as the models evolve, as new knowledge about the domain is discovered, or as the requirements evolve. Moreover, this new knowledge might be inserted by different software developers. As mentioned before, critiquing systems provide support to alternative and conflicting solutions, as well as, provide recommendations of actions to the user, but they leave the final decision-making to the user. By doing that, critiquing systems might also be used when one is deciding how to handle the detected inconsistencies [21].

In order to demonstrate the feasibility of our approach, we developed a critiquing system that implements the functionalities described above. This prototype, called DAISY, was built on top of another one called ABCDE-Critic [20]. Both will be described in section 5.

4. Domain and Application Engineering

4.1. Domain Engineering (DE)

The intent of domain engineering (DE) is to identify, construct, classify and disseminate a set of software components that have applicability to existing and future software in a particular domain [13]. DE encompasses a set of methods and procedures for software reuse that has been developed since the late 80's. It deals with the analysis and modeling of a given application domain that will provide the scope for future software systems.

In this paper a domain is defined as a knowledge area characterized by a group of problems with similar techniques, operational and functional specifications [9]. Examples of domains are telephony, banking, Internet browsers, and airline ticket reservation, among others. Usually, a domain presents a set of well-defined and coherent concepts and functions. Software products in this domain can be built based on these concepts and functions.

DE includes three major activities: (a) domain analysis, (b) domain design, and (c) domain implementation. In this paper, we are more concerned with domain analysis and domain design because the diagrams used by the critiquing systems are built during these activities. During domain analysis common characteristics from similar systems are generalized, objects and operations that are found in systems within the same domain and that vary from system to system are identified, and a domain model is defined to describe their relationships. This model is established to serve as a unified source of definitions, a repository of shared knowledge, and a basis for standardized reusable components [10]. In general, this model includes feature diagrams to represent the domain characteristics (features) and its similarities and differences. These diagrams are discussed in section 4.3.

Domain analysis takes into consideration the specifications of a family of systems and predicts possible changes that may occur in these specifications. This must be done in such a way that the created models can encompass target systems and be able to evolve. By doing that, it provides several benefits: it aids understanding of domain concepts and functions by the development team members; it creates a common vocabulary among the several stakeholders; it facilitates maintenance; and, it identifies similar characteristics among different products in the same domain, therefore supporting reuse [6].

After domain analysis, the domain design activity may begin. It is the process of developing a design model from the products of the domain analysis (the features diagram for example). It produces a design model that represents the generic architecture created for the analyzed domain and provides the framework for the development of reusable components during the next activity [18]. Usually, software architecture diagrams and class diagrams are used to represent this design.

The last activity of DE is the domain implementation. Using the domain knowledge gathered during domain analysis and the generic architecture developed during domain design, domain engineers acquire and, when possible, create reusable assets. These assets are catalogued into a component library to be used by application engineers. These reusable components, as well as application generators and domain languages, are the main outputs of this activity [18].

4.2. Application Engineering (AE)

Application engineering (AE) is the complementary process of domain engineering. This process generates software products from software assets created during the domain engineering [18]. Usually, an application is created by "slicing" the domain model according to individual requirements of the application, i.e., the specific application requirements are identified and the components that implement these requirements are selected to serve as a framework to the application development.

During AE, the necessary components of the application are chosen in a high level of abstraction, through the domain model, gradually descending in abstraction levels to reach the domain semi-developed or implemented components.

4.3. The Feature Diagram

One of the most important tasks during domain analysis is the creation of feature diagrams. The key utility of these diagrams is to characterize in an abstract and concise way, the commonality and variability among the applications of a domain [18]. In addition, they describe other characteristics that could be added, if needed (optional features). In general, features can be defined as functional abstractions that are implemented, tested and maintained. They can be implemented as classes or any kind of reusable components. An example is presented in Figure 2.

Feature diagrams should be general enough to be valid for different applications in the same domain. In fact, in order to test these diagrams, one should create several applications of the domain, which were not used in the definition of the feature diagram. In other words, feature diagrams show the architectural composition of software features, indicating, for instance, which one is optional, their definition, and rules that define how they can be combined.

It is important to note that according to the adopted DE process, the number of diagrams and relationships among them can be different. In this paper, we deal with feature diagrams and class diagrams.

4.4. Feature Diagram Notation

There are several notations to represent feature models such as FODA [9], FODACom [22] and [12]. In this paper, we used the notation proposed by Miler [11]. This notation is based on existing concepts from other notations, and provides additional concepts such as restrictions and associations. There are six different types of features:

(i) *Essential*: they model domain's fundamental characteristics. These features are intimately linked to the domain essence. They describe characteristics that represent the model's functionalities and concepts.

(ii) *Organizational*: these features are created with the purpose of organization. They do not really represent domain features.

(iii) *Actors*: These features are entities of the real world that act on the domain. They can, for instance, expose the need for an interface or control procedure.

(iv) *External*: Features that belong to other domains. They may be defined in the model or not. They show the boundaries of the domain.

(v) *Undefined*: features already identified in the domain, but that have not yet been defined.

(vi) *Additional*: Important additional characteristics for understanding the domain.

While features are used to represent the most important concepts and functions involved in a domain, relationships are elements that make it possible to express the way that these features interact. The types of relationships are:

(i) *Composition*: Relationship in which a feature is composed of several others. It is a relationship where a feature is a fundamental part of another, so that one does not exist without the other.

(ii) *Aggregation*: Relationship in which a feature represents the whole, and the others represent the parts. It is similar to the composition without the dependence relationship among its members.

(iii) *Inheritance*: This relationship is similar to inheritance in object-oriented programming languages, i.e., there is a super-feature where the common characteristics are placed and sub-features that are special types of the super-feature. Sub-features also inherit super-features' characteristics.

(iv) *Association*: It is a simple association between two features. It denotes some relationship among its members. It can be named indicating a specific type of connection.

The previous set of relationships is intentionally similar to UML's relationships. There are also relationships based on other DE methods (for example [7]), such as:

(i) *Exclusiveness*: Type of connection where the sub-features can not be used at the same time. It can denote variations or incompatibility problems.

(ii) *Optional*: denotes a domain's non-mandatory feature.

(iii) *Restriction*: It expresses the need of combined use of two features, or that problems occur if two features are used together. In this case, the features do not have a direct relationship between them as in the exclusiveness relationship.

An example of a feature diagram created according to this notation, is presented in section 5.

5. The DAISY[1] Environment

DAISY provides support to the domain and application engineering modeling activities by integrating three different critiquing systems. The first one evaluates the feature diagram. Therefore, domain-engineering activities are improved. A second critiquing system is used during application engineering to evaluate the class diagram according to object-oriented design heuristics. The main goal of these two systems is to check their models against well-formedness rules, i.e., rules that must be satisfied by the models for them to be legitimate models of the language in which they have been expressed [21]. However, as mentioned, the second critiquing system is also used to check object-oriented design heuristics.

Finally, the third critiquing system is used to improve the application engineering activities (in this case the construction of class diagrams) by using the information collected from the domain engineering models (feature diagrams). In short, this critiquing system is used to check development compatibility rules, i.e., "rules which require that it must be possible to construct at least one model that develops further two or more other models or model elements and conforms to the restrictions which apply to both of them" [21]. Specifically, class diagrams are evaluated according to the feature diagram because relationships between features are seen as either constraints that must be met, or recommendations that must be taken

[1] Domain and Application engineering using Integrated critiquing SYstems.

into consideration. Generally, the underlying idea in our approach is to use information expressed in a diagram as input for critics in other diagrams. This has been explored by other authors such as [19] and [2]. It is important to note that our approach is dependent of the notation used in the feature diagrams. If a rich notation is used, more information can be extracted from the diagram and, therefore, be used for checking class diagrams.

By using three different critiquing systems, our environment provides support to the development of feature and class diagrams, as well as reminds designers about characteristics identified during DE that could be forgotten during AE activities. In other words, the third critiquing system is the more important, because it implements the inconsistency detection. The other two systems were only used to improve the overall quality of the models created.

5.1. DAISY's Architecture

DAISY was developed using Java and is composed of about 40 classes. It has two main subsystems: the FeatureEditor and the ClassEditor. These subsystems were developed based on the critiquing system for UML class diagrams ABCDE-Critic [FSW00]. By extending ABCDE-Critic, we were able to easily develop two different critiquing systems: one for feature diagrams and another one for class diagrams. Figure 1 presents DAISY's architecture.

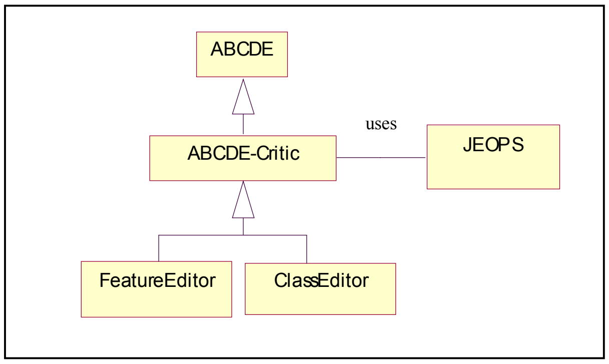


Figure 1: DAISY's Architecture.

In our work, we used feature diagrams to represent the concepts of the domain; therefore we created a graphic editor to support the development of these diagrams. It is called FeatureEditor and supports the development of diagrams created according to the notation described in section 4.4. As discussed in the previous section, this editor was created based on the ABCDE-Critic, thus using a critiquing system to evaluate its diagrams. The critics for this critiquing system are very simple and based on the notation used. Figure 2 presents a screenshot of the FeatureEditor with an example of feature diagram in the telephony domain (model extracted from [11]). We used UML stereotypes to express the feature types.

The model presented on Figure 2 describes a telecommunication domain where a telephony system is "composed" of several stations, and has more than one charging mode and possible use. There are two types of stations: PABX or individual stations, but one station can not be both at the same time. Furthermore, the charging mode can be based on the receiver or the sender, and it is connected with an organizational feature (bank). This connection expresses that a financial institution, the bank, needs to exist so that the telephony system is able to charge its customer.

An example of critic used by the FeatureEditor is described below:

External Features should not be refined

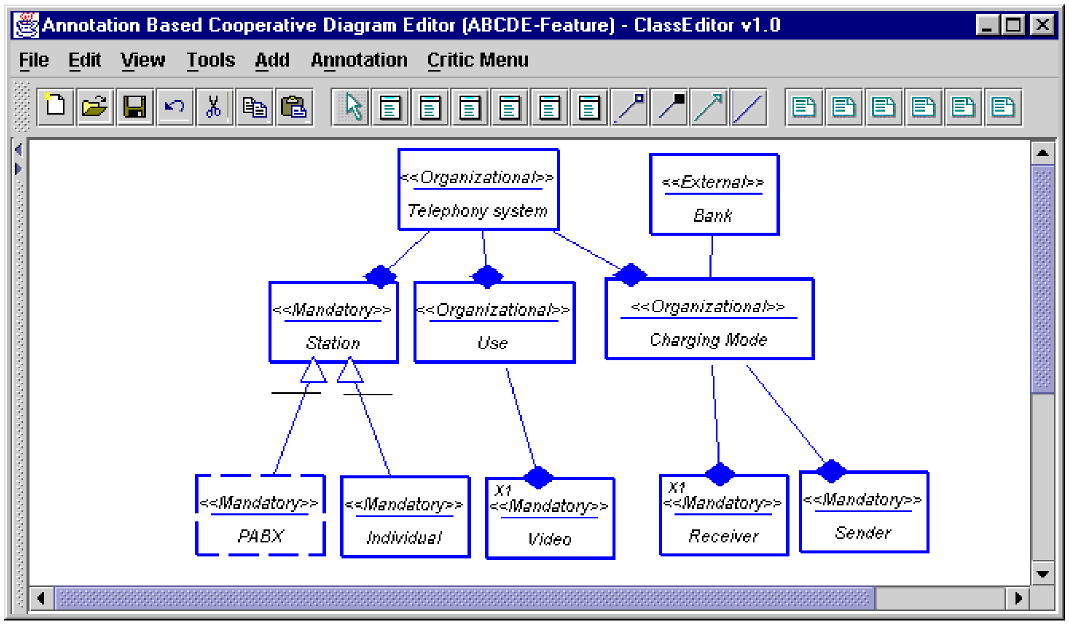
External features are those that belong to other domains [11]. They model the boundaries of the domain. In other words, these features do not belong to the domain of the problem, therefore, they do not need to be refined, because they are not really important in this domain. If these features are refined, they may add an unnecessary complexity to the model, turning it error-prone. For example, a designer could connect an essential feature to a sub-feature of an external feature.

At the moment, our environment is being "seeded" [3] with new critics to feature diagrams. As mentioned before, users might add critics. Therefore, a domain expert can also encode his knowledge about the domain as critics. It other words, besides critics about the notation used to create feature diagrams, the environment supports domain-specific critics. For example, in a telephony domain, one could define that a Station must have at least three sub-features. This constraint can not be expressed using the notation itself, but it can be expressed using the language used to express critics.

5.3. ClassEditor

ClassEditor is the most important component of our environment. It implements a UML class diagram integrated with two critiquing system. Critics already implemented in ABCDE-Critic based on heuristics for object-oriented design were reused.

However, the ClassEditor goes beyond that by adding new critics. These critics were created from relationship among features in the features diagram. This is possible, because the relationships in the feature diagram are seen as (i) either constraints that must be met or (ii) recommendations that must be taken into consideration during the design of the class diagram. Then, we are assuming that the class diagram is the "implementation" of the feature diagram. Examples of such critics will be described later in the paper.



Annotation Based Cooperative Diagram Editor (ABCDE-Feature) - ClassEditor v1.0
File Edit View Tools Add Annotation Critic Menu
<<Organizational>> Telephony system
<<External>> Bank
<<Mandatory>> Station
<<Organizational>> Use
<<Organizational>> Charging Mode
<<Mandatory>> PABX
<<Mandatory>> Individual
X1 <<Mandatory>> Video
X1 <<Mandatory>> Receiver
<<Mandatory>> Sender

Figure 2: FeatureEditor with a feature diagram about telephony.

In order to support the integration between the feature diagram (description of the domain) and the class diagram (that implements this domain), we created a trace relationship among features and classes. This relationship allows the connection of any number of features with any number of classes. In other words, a feature can be implemented as several classes, as well as several features can be mapped as only one class. The environment also supports bi-directional navigation among the models: from a class, it can identify the feature(s) that it implements or, given a feature, it is possible to check the class that it implements.

In the rest of the section, two examples of critics that implement development compatibility rules [21] are presented.

Essential Features should be mapped in the Class Diagram

Essential features indicate concrete concepts and functions in the domain, i.e., characteristics of the domain that should exist in all applications. According to that, these features must be mapped in classes in the class diagram. This critic just checks if all essential features have classes that implement them.

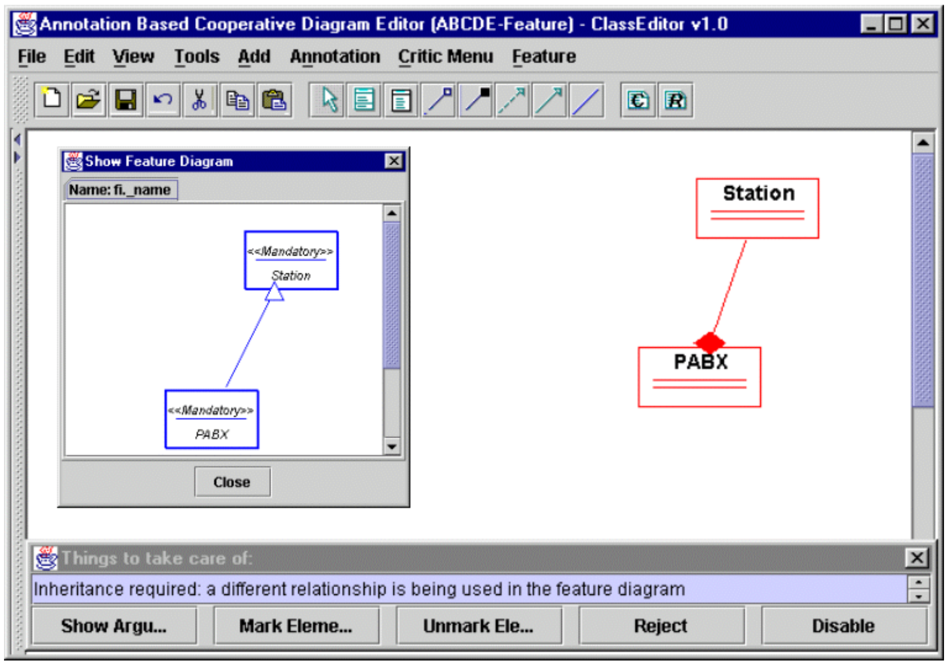
The critic above is an example of a critic that should be passive, because it only needs to evaluate the diagram when the application engineer asks for it. Otherwise, this critic would trigger when one essential feature were not mapped into a class. By doing so, this critic would be more a disturbance than a help to the designer [5].

Restrictions among Features

Miler [11] defines the concept of restrictions among features (section 2.4). According to his definition, two features connected by a restriction either can not be used together or must be used together. There is no distinction between the two cases.

However, this definition can not be used to generate critics, because it implies two different conditions. Therefore, we decided to create two subtypes of restrictions called of exclusion restriction and inclusion restriction. The first models the first case, where two features can not be used together; while the second means that it is mandatory to use the two features. Now, if two features A and B should be used together (inclusion restriction), a critic could check that both features should have mappings to classes in the class diagram. Or, if these features can not be used together (exclusion restriction), the class diagram should be checked to avoid two mappings at the same time. With this approach, we created two powerful critics without increasing the complexity of the notation and the tool for domain engineers.

It is important to note that any amount of design knowledge embedded in the environment (as critics) will never be complete because real-world situations are complex, unique, uncertain, conflicted, and unstable, and knowledge is tacit, which means that additional knowledge is triggered and activated only by experiencing breakdowns in the context of specific use situations [3]. In this case, the knowledge is triggered during the process of domain and/or application engineering. In the former case, developers can easily update the feature diagram to reflect the new knowledge. In the latter, a domain designer can easily update the feature diagram, and the critics will be automatically enabled in the class diagram.



Inheritance required: a different relationship is being used in the feature diagram

Figure 3: A critic is triggered pointing out that there is an incorrect relationship among the classes.

6. Related Work

6.1. Consistency Checking

Several approaches have already been proposed to support consistency checking in software development. For example, it is possible to check consistency of software engineering documents and specifications. It is also possible to ensure consistency among the several UML diagrams using abstract state machines [19], knowledge-based approaches [23], and so on. Our approach goes beyond those because it addresses the main issues in inconsistency detection, namely scalability and efficiency [21].

On the other hand, transformation-based approaches [2] also address the scalability issues by "abstracting" the models to be checked and then checking those new models. Our approach is similar to this one, but brings additional advantages to the efficiency process.

6.2. Critiquing Systems

Only a few critiquing systems to support software development activities have been identified. For example, Robbins *et al.* [15] describe Argo, a critiquing system based on cognitive theories, to support the development of software architecture models. Later, Robbins and Redmiles [16] describe ArgoUML, a tool for object-oriented modeling. This tool supports the edition of diagrams according to UML and detects common mistakes made by designers. In this case, the critics for the critiquing systems are object-oriented modeling heuristics, as well as the UML semantics.

Finally, Souza *et al.* [20] describe an environment called ABCDE-Critic, that uses a critiquing system to check UML class diagrams. ABCDE-Critic allows the users themselves to add critics to the critiquing system, because it uses a first-order production system.

None of these systems allows the construction of domain models. They also do not use critics to support inconsistency detection as in the DAISY environment.

7. Conclusions and Future Work

In this work, we presented the environment DAISY which provides support to domain and application engineering modeling. This is possible because it uses three different critiquing systems. The first one helps the development of feature diagrams and has defined seven different critics. The second critiquing system supports the creation of UML class diagrams using object-oriented design heuristics. It has about twenty critics. Finally, the most important critiquing system to this work uses the relationships defined in the feature diagram as critics in the class diagram. Therefore, constraints defined in the domain model are checked in the application model in order to detect inconsistencies between these models. Right now, there are seven different critics implemented. Despite the small number of critics, the authors believe that this approach is valuable in the process of domain and application engineering.

In the future, we are planning to integrate critics from different UML diagrams into other UML diagrams. For example, one could use the information defined in the sequence diagram to check the class diagram. We also plan to use our environment to create representative systems in order to properly evaluate its features.

Acknowledgments

The authors wish to thank Jason Robbins whose comments have helped us to improve this paper. We also thank PIPES and CAPES (grant BEX 1312/99-5) for the financial support. Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Effort also partially funded by the National Science Foundation under grant number CCR-0205724 and 9624846. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

References

[1] Coad, P. and Yourdon, E. *Object Oriented Analysis*, Prentice-Hall International, Second edition, 1991.

[2] Egyed, A. *Scalable Consistency Checking between Diagrams – the VIEWINTEGRA Approach*. In Proceedings of the 16th Conference on Automated Software Engineering, pp. 387-390, IEEE Press, 2001.

[3] Fischer, G. Domain-Oriented Design Environments. International Journal of Automated Software Engineering, vol. 1, pp. 177-203, 1992.

[4] Fischer, G. Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments, International Journal of Automated Software Engineering, 5(4), pp. 447-464, 1998.

[5] Fisher, G., Nakakoji, K. Embedding critics in design environments. The Knowledge Engineering Review, 8 (4); pp. 285-307, 1993.

[6] Fraser, S., *et al.* Patterns, Teams and Domain Engineering. In Proceedings of the International Conference on Software Engineering - Symposium on software reusability, pp. 222 – 224, 1995.

[7] Gomaa, H. *An object-Oriented domain analysis and modeling method for software reuse.*. Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, volume: ii, pp. 46 –56, 1992.

[8] Hägglund, S. *Introducing Expert Critiquing Systems*, The Knowledge Engineering Review, 8(4), pp. 281-284, 1993.

[9] Kang, K., Cohen, S., Hess, J. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[10] Lung, C., Urban, J. *Integration of Domain Analysis and Analogical Approach for Software Reuse.* In: Proceedings of the Symposium on Applied Computing: State of the Art and Practice, pp. 48 – 53, 1993.

[11] Miler Jr., N. *Application Engineering within the context of Domain Model based Reuse.* (in Portuguese) Master's thesis. Universidade Federal do Rio de Janeiro, 2000.

[12] Morisio, M.; Travassos, G.H.; Stark, M.E. *Extending UML to support domain analysis*, In Proceeding of the 15th International Conference on Automated Software Engineering, pp. 321 –324, 2000.

[13] Pressman, R. S. *Software Engineering: A Practitioner's Approach*, Fifth edition, McGraw-Hill, 2000.

[14] Rumbaugh, J., *et al. Object-Oriented Modeling and Design*, Prentice Hall International, 1991.

[15] Robbins, J, Hilbert, D.M. and Redmiles, D.F. Extending Design Environments to Software Architecture Design, *International Journal of Automated Software Engineering*. vol. 5. pp. 261-290, 1998.

[16] Robbins, J. E. and Redmiles, D.F., *Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML*, Proceedings of the International Conference on Construction of Software Engineering Tools, 1999.

[17] Silverman, D. Survey of Expert Critiquing Systems: Practical and Theoretical Frontiers. *Communications of the ACM*, 35(4), pp. 106-127, 1992.

[18] Software Engineering Institute, CMU. Domain Engineering:http://www.sei.cmu.edu/domain_engineering/domain_eng.html, 2001.

[19] Shen, W., *et al. A UML Validation Toolset Based on Abstract State Machines*, In Proceedings of the 16th IEEE Conference on Automated Software Engineering, pp. 315-318, IEEE Press, 2001.

[20] Souza, C.R.B.; *et al.* A group critic system for object-oriented analysis and design, In Proceedings of the Fifteenth IEEE Conference on Automated Software Engineering, pp. 313-316, IEEE Press, 2000.

[21] Spanoudakis, G. and Zisman, A. *Inconsistency Management in Software Engineering: Survey and Open Research Issues.* Handbook of Software Engineering and Knowledge Engineering, S. K. Chang (ed.), World Science Publishing Co., pp. 329-380, 2001.

[22] Vici, A.D.; *et al* .FODAcom: an experience with domain analysis in the Italian telecom industry, In Proceedings of the Fifth International Conference on Software Reuse, pp. 166–175, 1998.

[23] Zisman, A and Koslenkov, A. Knowledge based Approach to Consistency Management of UML specifications. In Proceedings of the 16th Conference on Automated Software Engineering, pp. 359-363, IEEE Press, 2001.