2008

Visual language representation for use case evolution and traceability

Coretta Willis Douglas
Louisiana State University and Agricultural and Mechanical College

VISUAL LANGUAGE REPRESENTATION FOR USE CASE EVOLUTION AND TRACEABILITY

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
Requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Coretta Willis Douglas
B.A., Mississippi State University, 1974
M.S., Louisiana State University, 1998
May 2008

Acknowledgements

I am grateful to my parents, Mr. and Mrs. Elward S. Willis, who encouraged me to study by their example and inspire me still with their curiosity to learn. To my friends at Louisiana State University, Elias Khalaf, Guillermo Tonnsman, Steve O'Neal, and Nigel Gwee who inspired me to continue my formal studies, I am grateful for their example too. Dr. Kraft was especially persistent in his encouragement and support; it is particularly because of him that I was motivated to undertake this challenge and have persisted. To my friends and my family (especially my husband, Mel) whom I have often neglected because of my academic pursuits and research endeavors, you have been so patient with me; thank you. My committee, Dr. Doris Carver, Dr. Jianhua Chen, Dr. Donald H. Kraft and Dr. Ye-Sho Chen, has been ever-supportive by guiding me, applying a needed push to get me back on track, and commending me on my efforts just when I needed it most. I have been most fortunate to have Dr. Carver as my major professor. She is a patient teacher with a gentle character; she astounds me with her knowledge of the field and her thorough and consistent analysis of our research. I have learned greatly under her tutorage having been more challenged by her well-spoken challenging questions than in many lectures. She propels me to study, to improve and to progress. Thank you, especially Dr. Carver, and all who have upheld me on this journey.

Table of Contents

List of Tables

List of Figures

Abstract

The primary goal of this research is to assist non-technical stakeholders involved in requirements engineering with a comprehensible method for managing changing requirements within a specific domain. An important part of managing evolving requirements over time is to maintain a temporal ordering of the changes and to support traceability of the modifications. This research defines a semi-formal syntactical and semantic definition of such a method using a visual language, RE/TRAC (Requirements Evolution with Traceability), and a supporting formal semantic notation RE/TRAC-SEM. RE/TRAC-SEM is an ontological specification employing a combination of models, including verbal definitions, set theory and a string language specification RE/TRAC-CF. The language RE/TRAC-CF enables the separation of the syntactical description of the visual language from the semantic meaning of the model, permitting varying target representations and taking advantage of existing efficient parsing algorithms for context-free grammars. As an application of the RE/TRAC representation, this research depicts the hierarchical step-wise refinement of UML use case diagrams to demonstrate evolving system requirements. In the current arena of software development, where systems are described using platform independent models (PIMs) which emphasize the front-end design process, requirements and design documents, including the use cases, have become the primary artifacts of the system. Therefore the management of requirements' evolution has become even more critical in the creation and maintenance of systems.

1. Description of the Problem

There are many contexts where a generalized approach is used at the onset of an endeavor, and then additional ideas or concepts are gradually introduced to further incorporate detail and difficulty. A challenging concept is commonly introduced in its simplest form, and then qualifications and/or exceptions are introduced in a logical manner to facilitate comprehension. An example is in the area of requirements engineering in software development where functional requirements are first loosely described and then customized over time by incrementally adding to or constraining the description of the system. As the evolution of requirements progresses, a means of tracing the transformation is imperative. Additionally, the process of refinement of requirements is compounded because of the variability of expertise among the participants. Foremost in the specification of the requirements is the contribution of the stakeholder who may, but often does not, have a technical background.

The hypothesis of this research is that a formally defined system for the depiction of the refinement and dependencies of requirements based on a formal representation benefits the stakeholder by increasing understandability, providing traceability and improving the quality of the representation. This research defines a visual model, RE/TRAC (Requirements' Evolution with Traceability) [DouCar2006], to represent the refinement of requirements and to enable the tracking of evolutionary improvements. A RE/TRAC model enhances the evolutionary process by providing a symbolic abstract depiction of change over time. Because RE/TRAC may be useful in many areas by people from diverse, perhaps non-technical backgrounds, a primary goal is to support a step-by-step refinement process that is comprehensible and easy to employ. System developers will benefit from utilizing RE/TRAC because it facilitates the customization of a set of core requirements by a non-technical stakeholder.

This research also describes an ontological specification, RE/TRAC-SEM, for describing the semantic information in the requirements evolution process with limited reliance on the graphical model [DouCar2007]. The formal semantic representation, i.e. the ontological specification, employs a combination of models, including set theory, a string language specification RE/TRAC-CF, and verbal definitions. The ontology enables the syntactical description of the visual language (VL) to be separated from the semantic meaning of the model, permitting varying target representations and integration with other system views.

The hierarchical representation and step-wise refinement method has many application domains; however, this research currently focuses on requirements specification within the requirements engineering domain.

1.1. Domain Modeling

An enterprise obtains a competitive advantage when it is the first to adopt innovative technologies; however, as new technology employed in a software application becomes commonplace, those who first employed the novel technology lose their competitive edge. Attention then shifts to enterprise goals and implementation strategies in order to regain a leadership position among competitors. Software developers today recognize that these fluctuations driven by changing technology requirements will persist in the future. One approach to managing changing technologies is the use of platform independent models (PIMs) that separate the implementation details, which rely on technology decisions, from the representation of the business processes. The separation of the implementation from the analysis and design of a problem solution allows developers to better focus on identifying the high level goals, refining the goals, and implementing the goals [Foreman]. The output of the goal refinement stage is a set of system requirements.

Before the goals are identified for a software development application, the domain must be identified, and the scope of the domain must be delineated. Pender defines a domain as the description of basic elements common to all systems in the same subject area and their relationships [Pender]. A primary activity of domain engineering is to identify commonality and variability within the domain under study and to plan for reusable components [Pender], [Sutcliffe]. For a specific application within a domain there may be elements in the domain that are optional or excluded.

An application within a domain is developed using the artifacts described by the domain engineer, taking advantage of the commonality and variability documented within the domain [Foreman], [Morisio et al]. The artifacts in each stage are often UML documents and text [Foreman]. The domain model, which is a conceptualization of the entities and their relationships, is embodied in these artifacts. The set of requirements for an application is represented in the domain model.

The responsibilities for the domain engineer have shifted to reflect the emphasis on PIMs. As auto-generation of code from the domain model and from design documents has become a reality, analysis and design phases of domain engineering have become more significant. The analysis and design (front-end) artifacts have become the primary representation of the system, rather than the low-level executable code. Whenever requirements change, only the analysis and design artifacts are modified.

1.2 Motivation

Once a domain model has been clearly described, the requirements for an individual application can be specified. There are several reasons that the domain engineer or stakeholder in the system would want to directly manage the specification of the requirements:

1. Assuming auto-generation of code from the requirements' documents, the enterprise stakeholders can react instantaneously to changes in the environment by altering requirements documents.
2. There is a continuing trend in enterprise development towards implementation of "best practices"; however, the enterprise stakeholders may consider their goals private and may distrust an outside domain engineer who might duplicate successful operation strategies in other applications within the domain. With auto-generation of code from the requirements specification, some of the enterprise's business strategies can conceivably be implemented and maintained privately.
3. Stakeholders can make adjustments to the requirements based on anticipated business expansion such as in employee organization and product development.
4. One-of-a-kind software systems are expensive to develop and maintain. A cost effective alternative to application-specific software is commercial off the shelf (COTS) software. But often customers cannot modify the software to incorporate their own processes and implementation strategies. More often customers must amend their individual operations to conform to an inflexible software product.

COTS software can solve these problems by permitting the user to specify requirements in order to generate or alter the implementation. A framework of core generic requirements provided explicitly for the domain defines the problem space and thereby simplifies the

requirements specification process. Requirements that are structured in natural language-like form can be easily understood and altered by the domain expert. However, the management of large quantities of interconnected textual documents can become difficult. A global composite view of the documents is helpful. A partial solution to the problem is to employ a visual diagram or graphical depiction of the information to support the design effort. The fundamental aspects of such a software development method are described in this dissertation.

RE/TRAC when applied to the evolution of requirements enables traceability of changes. The ability to trace system development over time is important. Structuring the changes by temporal order is important for sequencing the altering events. As evolutionary changes are made, a log should be maintained of the deviations so that the linear history can be queried in both a forward and backward order. Tracking enables evaluation of progress during development; provides documentation; provides an audit trail for error and fault resolution; and serves as a pattern for future developments. When problems occur in the verification and validation of the requirements, tracing the alterations in a backward manner enables a rollback to a viable system description. The current state of the system requirements in the evolutionary process is observable in the documentation. Subsequent systems development may take advantage of commonalities in a requirements specification already in practice and points of delineation in the history may be marked.

Therefore, the motivation for this research is to empower the non-technical stakeholder in requirements specification by facilitating requirements configuration management. The quality of the method is sustained because all views are consistent with the formal representation of the current and past states of the requirements specification. By enabling the tracing of the systematic evolution to the front-end documents, alterations noted in discrete change events may

be verified and validated in a temporal order. The rules and constraints of the formal representation enforce a uniform and coherent interpretation of all views of the requirements evolution.

1.3 Summary

Requirements engineering is the most important phase of software development. If the requirements are not well understood and described, the result is likely to be a failure. Today's emphasis related to application development is on domain engineering to analyze, design, and implement a set of related systems. Requirements within a domain are described in a manner to account for variability and commonality across various implementations. Platform independent models (PIMs) enable the domain engineer to focus acutely on realizing the enterprise goals and representing the requirements within a system regardless of technology and platform. More emphasis is placed on the accuracy and completeness of the front-end artifacts from the analysis and design phases, as they form the main artifacts of the system from which implementation-specific requirements will be added and code auto-generated.

Enterprises are concerned about securing their strategic goals and requirements as reflected in the corresponding implementation. In addition, the viability of the enterprise may depend on immediate reaction and response to changes in a competitive market. The avoidance of custom application software reduces the dependence on a domain expert outside of the enterprise organization and lowers the expense of one-of-a kind system development. An enterprise could therefore benefit from employing a software development method for requirements engineering that is domain-specific but permits requirements to be customized in a flexible manner. Because the stakeholder may not be a technical expert, the approach should be simple, intuitive, and robust. This research provides a method that supports the elaboration of

the core requirements within a specific domain in the form of a set of natural language-like entities. The use of a visual diagram facilitates the design process and eases the work of the stakeholder by providing a simple abstraction of the work. The evolution of changing requirements is recorded in a temporal order, enabling changes to be traced in forward and reverse directions. The separation of the visual diagram from the text string language with the accompanying formal semantic description permits varying target representations of the requirements.

RE/TRAC can be applied to diverse applications. Where entities are elaborated and there are relationships between entities, the effects of change are compounded because of the possible cascading impact on other entities that are involved in integrated relationships. A method such as presented in this research is needed to facilitate consistency between the entities, including the various versions over time as well as consistency between various views depicting the entities.

2. Modeling with UML

2.1 Requirements Engineering

A system is built to satisfy the needs of a client taking into account possible constraints such as cost, time, and resources. The client accepts the system if it exhibits the desired features or requirements that the client has expressed as essential for a successful software implementation. Requirements engineering is concerned with defining the requirements for a system under construction prior to design and implementation. This entails two stages, requirements elicitation and requirements analysis. Ordinarily, requirements elicitation involves dialogue between the client(s) who are the domain expert(s) and the developer(s) to obtain a description of the work that the system should accomplish. The developers then analyze the descriptions of the requirements for feasibility and resolve any ambiguity in the specification of the system.

Requirements are described as functional or non-functional and should be traceable to the system goals. Functional requirements pertain to the interactions between the system and its environment. Non-functional requirements pertain to system implementation aspects such as usability, reliability, and performance. This research focuses on the representation of functional requirements and how the representation enables the evolution of the requirements specification.

The requirements should be validated as complete, consistent, unambiguous and correct. All aspects of the system should be represented, including exceptional behavior. The descriptions should be clear in meanings, and there should not be contradictions between the requirements. The system should be represented accurately, according to a client's needs. When the system is implemented, repeatable tests should verify that the system fulfills the requirements specification; furthermore, the requirements should be described in the specification in a manner

so that requirements may be verifiable in the implementation [BruDut]. The requirements specification becomes the foundation for all subsequent documents and artifacts in the system modeling process. This research does not detail the preparation of the formal requirements specification document, but recognizes that any representation of the requirements must embody the character and purpose of such a document.

2.2 Overview of UML

The Object Management Group (OMG) formally developed and approved a standard for a modeling notation and also for several modeling techniques. The Unified Modeling Language (UML) has emerged as the de facto standard for modeling object-oriented software. UML permits developers to specify and document models in a graphical or visual manner. The language provides extension mechanisms so that the language may be used to customize the models to a particular technology or platform.

The architecture of UML is based on the Meta-Object-Facility (MOF) which is the foundation for creating other modeling languages. The MOF defines standard formats for the modeling elements so that pertinent facts about the models may be shared or converted from one modeling language to another. The Extensible Markup Language (XML) is such a language, and the XML Metadata Interchange (XMI) facilitates the sharing of various model elements [MOF], [OMG].

The Unified Modeling Language (UML) [OMG] is a language employed to describe the system processes and structures of the business and the resulting software. UML is especially expressive in that the system can be described from numerous views; however, the combination of multiple views is needed to completely describe the system. A diagram depicts a view, or, more often, a combination of diagrams of various diagram types (diagrams) is used to depict a

particular view. The motivation in using multiple diagram types is to keep a single diagram, which describes a view(s) in a graphical manner, uncluttered, clearly dedicated to a particular aspect of the system and therefore easy to understand. There is some repetition in the information stored between diagrams, and additionally there are diagrams that span views. Keeping all diagrams cohesive so that ambiguity and inconsistencies are checked is a daunting task. Furthermore, most of the diagrams, while intuitive to a software developer, have fairly complex syntax and semantics that can intimidate non-technical stakeholders. This research concerns the specification of the use-case view.

2.2.1 The Use Case View

The use-case view that depicts the system from the external actors perspective is the basic building block of this research. Delineation of the system provides boundaries between the actors and the system. The detailed functionality of the system is not emphasized at this time. So the system is often referred to as a black box or gray box. The user (stakeholder) should be the most comfortable with the use case view because it focuses on human interaction with the system and avoids implementation details. The use-case view is the foundation of all other views, and ultimately the set of use cases will describe the functionality of the working system. The use case view is composed of use cases, each of which narrates a complete and specific set of actions that are closely related. While use cases may join other use cases and incorporate other use cases, duplication of functionality should be avoided.

According to [Eriksson et al] the main purposes for use cases are:

1. To refine and describe the functional requirements of the system,
2. To provide an unambiguous and consistent description of what the system will do, and later upon implementation, the working functionality

of the system,

3. To provide the basis for carrying out tests that verify that the system performs as expected and to validate the system's capabilities as requested, and
4. To provide the means to trace functional requirements to the classes and operations in the system.

[BruDut] describes how a use case is developed during requirements elicitation:

1. Actors identified. The users (actors) of the system are identified.
2. Scenarios identified. The users are observed and a set of detailed responsibilities or scenarios emerges for the users. The developers rely on the scenarios to communicate what the system is to do.
3. Use cases identified. The scenarios are grouped according to their functionality into use cases. The use cases will define the scope of the system.
4. Use cases refined. Each use case is specified in detail, including exceptional behaviors.
5. The relationships between use cases are identified. There may be commonality between the use cases or dependencies among the cases that when identified, may simplify the system specification.
6. Non-functional requirements identified.

When a use case has been thoroughly investigated, A UML use case diagram with characteristic stick figures will be supplanted by natural language (NL) text descriptions. The

descriptions are formalized by using a template that shows the sequential progression of actions in a structured manner for a use case (see Appendix A [Cockburn1997], [Cockburn1998]).

Use case approaches should address relationships such as generalizations, extensions, and inclusion. [Pender] describes a refinement of actors in a use case similar to the way that classes are generalized. Generalizations are called the “is-a” relationship in a simple context. For example, a project manager is an employee. A project manager is a special case or refinement of its parent class, employee. Likewise, the use case diagram can depict generalization among actors, which eventually will be represented as a class.

The include relationship pertaining to use cases is used to support the identification of common features that may be used between objects. [Pender] describes the behavior as a call to another use case. The calling use case incorporates or includes the called use case in a nesting relationship. The extend relationship describes optional behavior of another use case. The use case that provides the extension functionality is only inserted into the base use case if a discrete behavior necessitates the additional functionality.

Scenario diagrams (scenarios) help to isolate specific functionality in a use case. A scenario shows a single task as a sequence of actions that will produce a final result. A scenario will enact a single path in a use case and will end in some final conclusive state. Scenarios are useful for requirements gathering and validation of the system functionality because they are depicted from the users’ or stakeholders’ perspective [AntPot]. Testing of a use case often involves testing of each scenario in the use case.

An activity diagram is a dynamic depiction of the sequential flow of events such as the general process workflow. Creating activity diagrams in concert with the use cases is helpful for defining operations, discovering and refining use cases, and describing workflow between use

cases. Flow in the activity diagram occurs upon completion of an event or action. Control mechanisms and conditionals are used to show the response to triggers from external events or from time dependent constraints. It can show parallel events as well. The activity diagram can be used to identify the objects in the system that will be used to support the static behavior of the modeled business. [Pender].

2.2.2 The Logical View

The use-case view describes what the system should do. On the other hand, the logical view describes how the system's functionality is depicted inside the System [Eriksson et al]. This research uses "system" to denote the application development in its entirety, whereas "System" is used to designate the internal system with which the external actors will interact. The use-case view represents the System as if it were a black box, whereas the logical view describes the System as a glass box. The internal workings of the System must be described and defined in a detailed manner. The logical view shows both the static and dynamic behavior of the System essential for later code generation. From the use case identification, other UML diagrams are employed to provide the necessary information needed to capture all of the system requirements in the logical view. The static structure is depicted by class and object diagrams, whereas, the dynamic behavior is modeled using state, interaction, and sequence diagrams.

The logical view is intended primarily for designers and developers. The information contained in the logical view is essential for the implementation but is difficult for the non-technical stakeholders to comprehend. The research aims to shield the stakeholders from the complexity of the logical view, but acknowledges that the logical view is reliant on the information content of the use-case view.

The class diagram represents the things (classes) that are represented in the system and how they are related. Classes are associated to one another, may be dependent, and may be a specialization of a different class. Classes may be grouped together or packaged as a unit for a depiction at a higher level of abstraction. An object diagram is very much like the class diagram except class instances, called objects, are shown instead of the more abstract classes.

State machines, interaction diagrams, and activity diagrams show how the objects will interact during execution. A state machine complements the class diagram by illustrating all the states that an object can have and the events that cause the states to change. A movement from one state to another due to an event is called a transition. Sometimes an event may be caused or triggered by another object interacting with the object being described. A transition from one state to another is depicted in the diagram as a directed line with the behavior noted that describes the action occurring during the transaction.

There are several types of interaction diagrams: sequence diagram, communication diagram, and interaction overview diagram. Interaction diagrams are so named because they demonstrate the interaction between objects during execution. A UML sequence diagram shows the interaction of objects in a scenario and how the scenario unfolds over time [AntPot]. The sequence diagram shows an ordering of messages communicated between objects and also has vertical lifelines to depict the time frame of an interaction sequence.

The communication diagram is similar to the sequence diagram but does not include timelines. The communication diagram is sometimes called a context diagram because it is used to depict the classes and their relationships for a single scenario within a use case. Messaging is indicated on the connecting lines between the classes. The use of the communication diagram is significant to verify that the class diagrams are complete for a use case. Eventually the class

diagrams relevant to all use cases must be integrated to provide a static view of all classes with complete data elements and methods.

2.2.3 The Ontological Specification

We assume a single source of information or repository for a system. The system dictionary is the central repository for the system documentation, artifacts and system constraints, and it is vital to ensure consistency between the use case view and the logical views. A configuration area for global reference containing common information across multiple views avoids duplication of information in the system, helping ensure a correct system that is consistent in all views. The dictionary contains a glossary of terms relevant to a particular domain. As use cases are described, a dictionary of vocabulary and facts from the domain will be referenced and updated to later facilitate the integration of class diagrams and the behavioral model. A diagram (model) is generated based on the syntax and semantics of the UML metamodel and on information pertinent to the diagram found in the dictionary.

The notation for requirements traceability used in this research can be described as a particular view within the use case view for the representation of requirements. Data concerning the requirements (structured as use cases), including all versions over time, will be archived in a common repository. By querying the repository, a tracing of a use case including its derivation history can be found. Constraints on use case behavior will be located in the repository also.

To facilitate code generation, use case descriptions are semantically complete and consistant such that each use case description in the use case view is unambiguous and in compliance with the dictionary. This research acknowledges that use of a controlled natural language and the supporting dictionary provide a precise detailed description of the requirements specification needed to facilitate code generation. This research also recognizes the importance

of issues related to completeness of requirement specifications, however, small-grain technical issues necessary for code generation are not addressed. We maintain consistency within the active versions and archived use cases only in this research.

The common repository or dictionary is often represented by a specification, herein called an ontology. The dictionary may in fact utilize multiple ontologies. The use of an ontology can support the refinement process by providing a formal structure for coherence of the related requirements' documents for a particular domain application by simplifying the representation of the information and by automating traceability. An ontology is most often quoted as a "formal, explicit specification of a shared conceptualization" [Gruber]. Names and definitions are given to terms and relationships in the domain to represent an abstract model. The abstract model is depicted in a formal manner in order to remove ambiguity and at times to provide flexibility in the manner of presentation. The ontology can be used reliably as a specification by providing documentation, supporting maintenance and enabling reuse of knowledge. A system may employ various ontologies to structure the information and unify the information in the repository for searching and for viewing during all phases of development [UschJas]. We employ such an ontology, RE/TRAC-SEM [DouCar2007], to support requirements evolution within an application domain.

2.3 Use Case Evolution

This domain-independent research focuses on artifacts used to directly represent requirements. This research relies on natural language descriptions of the requirements in the form of use cases. We build on the knowledge that the domain has been defined and sufficiently analyzed such that a core set of use cases common to the domain have been defined in a planned

and rigorous manner and that the functional requirements are sufficiently embodied in the form of use cases. Use cases are represented using templates in a plaintext form for understandability.

We demonstrate a view of the domain model that depicts how a set of use cases are related and integrated over time. The refinement of use cases, as well as the dependency relationships between the use cases, is prescribed following a step-by-step method specification which incorporates traceability of changes. The notations described in this research relate to use cases using a compact representation of the natural language descriptions. RE/TRAC addresses the abstract representation such that generalizations (refinements) are applied to the base set of use cases. Use cases may be related to other use cases via the dependency relationships of inclusion, and extension. Dependent use cases may also be refined.

2.3.1 Hierarchical Structure of Use Cases

Hierarchical models are intuitive and support the goal of a simple and understandable portrayal of the system via use cases. The hierarchical depiction of use cases is a graduated presentation of the specification detail. Higher levels typically will be of coarser granularity while lower more recent levels in the use case hierarchy will be of a finer granularity. The hierarchical nature of the representation enables top-down, step-wise refinements to the general or base form of the use cases for customizing of the use case model. In use-case modeling, this concept is known as generalization, which is the relationship of a child use case to a parent use case. Usually the child case adds more detail to the parent use case description by further specializing of behavior and characteristics of the parent. Also at any level, new requirements may be introduced that are not part of the previous level's base requirements. Each level in the hierarchy represents an evolutionary change to the base use case over time. A record of change is

therefore captured as a means of documentation and enables tracking of the transformations in both a forward and backward manner.

In RE/TRAC, the child use case does not necessarily encompass more detail than the parent. This research uses the term generalization of the use cases to refer to refinement that optionally incorporates more information. When a modification transpires to create a child use case, this research describes the child as a generalization of the parent. The child use case exists independently from the parent case, but the relationship is recorded. This definition differs substantially from the understanding of generalization in the context of classes. This research uses the term child to mean that it is a more current generation or version of the previous parent use case. A use case from the initial base set has no parent.

2.3.2 The ATM Banking Example

The example of a banking system providing automated teller services is well known and is used as a rudimentary introduction to the refinement of a use case as defined in this research. An automated teller machine (ATM) provides an interactive user interface for banking transactions such as withdrawals, deposits and account queries. In this example a single use case will undergo several evolutionary changes. The use case is presented in template form and based on the ATM example use case in http://www.lv.psu.edu/cad18/ist240/ucn%20sect1%20ex.htm. The template depicts actor/user actions in the left-hand column and the System responses in the right-hand column. The English language format is unstructured with abbreviated sentence constructs. Appendix B Version 1 contains the rudimentary base case describing the banking transaction, “Withdraw Cash from an ATM”.

Referring to Appendix B Version 1 and event numbered 2, the method of identification is not explicit. The use case actions numbered 2 and 3 can be altered to specify the way in which

the customer is identified such as via pin number, fingerprint or eye scan. A refinement to the original use case is made and a revised version of the use case is created and replaces Version 1 as the current active use case (see Appendix B Version 2). Boldface type denotes evolutionary changes to the use case shown in Appendix B. The action statement:

(Actor Actions: Version 1) 2. Customer Id's self to ATM

is revised to:

(Actor Actions: Version 2) 2. Customer activates the ATM.

The System response to action statement number 2 is revised from:

(System Response: Version 1) 3. Verifies valid customer
Constraint:
If invalid customer ID,
stop transaction.

to:

(System Response: Version 2) 3. System checks specific ATM verification method.
(System Response: Version 2) 4. Directions for verification method displayed.

After the verification method is determined, the customer will be prompted for identification in the user action response re-numbered 5 from its original Version 1 statement number of 2. All other enumerations are altered accordingly.

2.3.3 Use Case Extension and Inclusion

The extend relationship between use cases allows the user to customize a use case by describing optional behaviors. This relationship can be used to complete generic high-level use cases that are functional but not comprehensive. As requirements change, a use case may be extended to incorporate additional use cases. Refinement by incorporating extensions supports the component-based implementation of the system as well.

In the ATM example, assume customers with pre-approved credit limits can instigate an instant loan when funds are insufficient for withdrawal. System action statement 14 in Version 2

is extended to verify the instant loan feature and reply with the pre-approved loan. See Appendix B Version 3 for the revised use case with the extension. The original statement 14 in Version 2,

(System Response: Version 2) 14. Bank Info System returns request status
Note:
If status insufficient funds,
return "Insufficient Funds"

incorporates the new functionality in Version 3,

(System Response: Version 3) 14. Bank Info System returns request status
Note:
If status insufficient funds,
return "Insufficient Funds"
check instant loan approval,
return instant loan approval amount.

After that, the user is given the option of accepting the instant loan terms if s/he is pre-approved. Statement 15 is altered from

(Actor Actions: Version 3) 15. Customer views request status

to

(Actor Actions: Version 4) 15. Customer views request status
Constraint:
If Customer approved for instant loan
Customer accepts or declines

The System reacts by dispensing the amount based the users request. If the user selects the instant loan feature, then the System actions are extended in statement 16 Version 2 to incorporate the additional functionality of the use case, "ATM Instant Loan". System statement 16 is changed from originally,

(System Response: Version 2) 16. System dispenses desired amount
Constraint:
If status insufficient funds,
do not dispense.

to

(System Response: Version 3) 16. System dispenses desired amount

Constraint:
If status insufficient funds or
(status instant loan approved and customer declines),
do not dispense.
If (status sufficient funds and customer accepts),
instigate Use Case: ATM Instant Loan

Moreover, a use case may be incorporated to facilitate the include relationship between use cases. While a generalization refines at the child level, the include relationship allows nesting of use cases, which supports a component-based organization. The use case(s) that is included is always essential for all scenarios in the use cases as opposed to describing optional behavior of the extend relationship. A use case example of inclusion in the ATM textual example may be found in Appendix B Version 4. By grouping common actions into a composition of events, the use case is simplified. The Customer Verification is modeled as a separate use case which can be included in the "Withdraw Cash from an ATM" use case. Steps 3, 4, 5, and 6 are common in Versions 2 and 3:

(System Response) 3. System checks for specific ATM verification method.

(System Response) 4. Directions for ID verification method displayed.

(Actor Actions) 5. Customer Id's self

(System Response) 6. Verifies valid customer
Constraint:
If invalid customer ID,
stop transaction.

The statements are therefore combined and separated into a single use case, "System Verifies Customer". The statement 3 is revised to

(System Response: Version 4) 3. Instigate use case: System Verifies Customer

and the subsequent statement enumeration updated.

An application may elect to totally exclude the functionality of a use case. Employing the option to literally delete is discouraged because the functionality may be needed at a later time. Similarly, derived use cases that have once been referenced are archived, as they are required for traceability.

2.4 Summary

The Unified Modeling Language (UML) defines a notation using object-oriented concepts. The notation, which is graphical, describes the language syntax which embodies the rules for depicting various models or diagrams. A model is a simplified depiction of the system with a goal or perspective in mind. The UML is a meta-model because it explains how each artifact of a system may be put together or composed. A developer will describe a particular system using models that adhere to the syntax and semantic rules of the meta-model.

Because UML offers many advantages in software development and because it was designed expressly for documenting object-oriented systems [Pender], it has become prevalent in object-oriented development environments. In recent surveys, the object-oriented paradigm methodology is considered superior to the classical paradigm that relies on structured programming techniques using modules. While not without problems, the object-oriented paradigm continues to grow in acceptance as its successes are repeatedly documented [Schach]. A few of the advantages UML offers are: a graphical notation fairly easy to understand by developers; a meta-model depicted using an object-oriented representation that is familiar and has been a successful modeling strategy; and extensibility mechanisms that offer creative and flexible modeling solutions. UML enables the creation of specifications that are independent of the programming language selection and development processes [Pender]. Finally, UML supports view-oriented development of which the use case view is prominent.

The ATM example presented in section 2.2.2 demonstrates the need for a method to manage evolution of requirements documents. Even in this uncomplicated example of a single use case, several document versions were created and dependent relationships created. As the number of changes increase, the complexity of sequencing the document versions and of managing the dependent relationships grows also. To simplify this problem and to maintain the integrity of the system development, this research describes a graphical representation, RE/TRAC, that facilitates the depiction of system requirements within the UML use case view. RE/TRAC visually shows a use case and its relationships with other use cases. After refinements are made to a use case, the graphical depiction incorporates the changed use case and its relationships, thereby enabling traceability.

This research employs a large-grain graphical method to manage system requirements hierarchically to permit the user to straightforwardly understand the functionality of the system, to support refinement of the system and to offer traceability. Use cases are described during requirements evolution using the concept of generalization. A use case at a higher level in the hierarchy will usually be coarser grained than a use case lower in the hierarchy. The stakeholder should be able to enhance use cases to build a system specification in a step-wise manner. The include and exclude relationships between use cases are also shown in a hierarchical manner that depicts the integrated relationships of the use cases and supports modularity. A visual language is described for presentation of the model to the stakeholder. A related supportive grammar is defined using formal methods in order to uphold efficiency and reliability aspects of the visual model.

3. Related Research

RE/TRAC is inspired by the paper "Generating Product-Lines of Product-Families" [Batory et al], which describes primitive components that are the core constructs for features by using an intuitive graphical depiction. A component has a hierarchy of levels or layers that are the features, and it may represent refinements of the primitive form. The notion of feature refinement is extended to describe the relationship between multiple classes based on the information contained in the various collaboration diagrams for a system model. In UML 2.0 [OMG] the collaboration diagram was renamed and is now called a communication diagram. As described earlier, the communication diagram is a dynamic depiction without timelines showing the relationship of objects and the exchange of messages between the objects.

Communication diagrams are developed in isolation of one another; therefore the diagrams likely overlap in content, that is, they share common features that enable reusability [Batory et al]. The union of all the communication diagrams provides the complete representation of the classes and their intercommunication. Figure 1 (see Appendix C, [Batory et al]) demonstrates how refinements have been added to a constant set (i) of classes. Levels in the diagram indicate refinements to a class such as the addition of new data members to a class, of methods to the parent class, or of method overrides of the parent class. The diagram does not depict the communication between the classes, only the refinement as a particular collaboration diagram is considered at each level.

In Figure 1 the complete set of classes is described by the initial constant set i. Each class has data members and methods pertinent to the constant i. Noted at these levels are the functions i, j, and k. [Batory et al] refers to a level as a function because it denotes the functionality needed

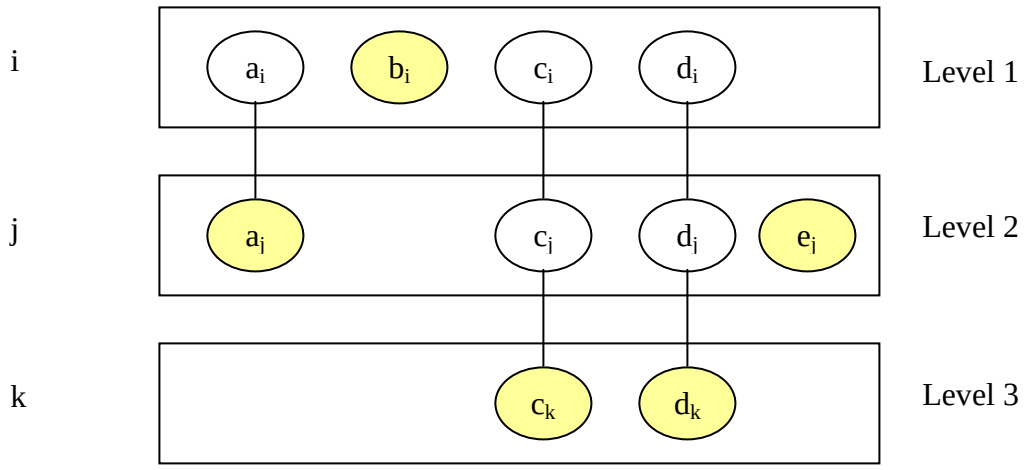


Figure 1 Hierarchical Refinement of Classes [Appendix C, Batory et al]

by a class in a particular collaboration diagram. So function j adds functionality from a particular collaboration diagram, which brings in more content to a_i. In the same manner c_i and d_i are expanded. Also at Level 2, a new class, e_j, is defined. Likewise, at Level 3 another collaboration is considered and the functionality of k is incorporated into the classes, yielding c_k and d_k. The bottom-most class (leaf) of a refinement chain (depicted by the connecting lines) constitutes the class that is instantiated. A leaf class implements all of the roles assigned to it via the totality of the collaboration diagrams. So in the above example, the application will need the classes, a_j , b_i , c_k , d_k , and e_j.

A similar concept of a hierarchy of levels is employed in this research to show levels of refinement of use cases for a particular application. In analysis and design of system software development specific core or base requirements formulate the most generic description of a system within a domain. As the system requirements evolve for a specific application instance, refinements (generalizations) are made to the core in a step-wise manner by formulating revisions to the core use cases.

The class diagrams depict dependencies with lower level classes inheriting characteristics of the upper level classes in the hierarchy. The leaf classes of a class diagram are instantiated but so are all the classes above it upon which it depends. This research differs from the typical class diagram in that with each refinement step a new version of the previous step is created that replaces the instance above, and this research incorporates interconnectivity of the parts.

In [BatO'Mal], an acyclic call graph is described to depict reusability of components. The edges denote the call relations between the components. Figure 2 (see Appendix C, [BatO'Mal]) is an example of the graphical hierarchical notation.

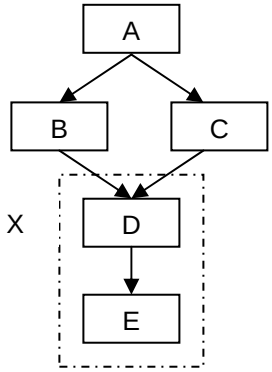


Figure 2 Hierarchical System H Showing Nesting of Components [Appendix C, BatO'Mal]

The hierarchical system, H, in Figure 2 is noted by the expression:

H = A[B[X], C[X]]

The subsystem, X, composed of components, D and E, is described as:

X = [D[E]]

The following expressions representing software systems, a[b[c]] and d[b[q]], reveal that component b is reused. The common use of b in the sub expressions indicates that b is used in two different systems. Batory notes that components may have input and output

parameters such that there is one instance of the code for subsystem X (see Figure 2), but B and C may use X differently based on the input parameters. Similarly, components in [BatO'Mal] as in RE/TRAC are denoted by rectangular symbols. This research also graphically represents the grouping of components in a nesting depiction for the include relationships between use cases. This research differs from [BatO'Mal] by additionally employing a horizontal access dotted line for extension and a component visual primitive for condensing portions of the diagram. The component subsystem expression noted by [BatO'Mal] is not incorporated into the expressional representational grammar for RE/TRAC but is a function of the graphical transformation from the expression. This research also uses a visual diagram that is top-down in interpretation, but the levels in RE/TRAC denote change over time. Levels in RE/TRAC indicate refinements to previous components symbolizing use cases.

3.1 An Incremental Method for Specification

E. Astesino and G. Reggio [AstReg2002a][AstReg2002b] describe how to organize and represent requirements specification artifacts. Similarly, this research employs a multi-view use-case driven approach to requirements specification and describes a representation of requirements that depicts a separation of the domain model from the System model. This research differs in that while the separation will simplify the presentation of the System via the use case view to the stakeholder, the stakeholder may not always address the System as a black box. The stakeholder as the domain expert must know the internal structure of the System, and, therefore, the System's operability must be explicit to the user, if only upon request. The research assumes that the System may not be totally separated from the domain model. One approach to requirements specification uses diagrams described using the UML notation whereas this research does not used the UML notation [AstReg2002a], [AstReg2002b].

[AstReg2002a] describes a method for initially capturing the requirements and incrementally adding to the specification while maintaining consistency among various views of the system. It is use case directed to describe a way to capture the requirements and specify the requirements. This research does not focus on the initial capture of requirements. It will assume that the base set of use cases has been elicited and the set is represented in a declarative manner so that the stakeholder can make additions, deletions, and refinements to the core set of use cases.

The method presented in [AstReg2002a] provides insight on how to initially set up and organize the information about the core set of use cases from a multi-view viewpoint. This method also shows the need for a method to be prescribed to guide the stakeholder in the building of the system by incrementally merging use cases in a multi-view context. Our research defines a step-by-step procedure of the transformation from one set of entities (initially the core set) to another set of entities as evolutionary changes occur, rather than the small-grain specification of individual use case documents.

3.2 Graphical Representations of Requirements' Behavior

The notation of Message Sequence Charts (MSCs) [Reniers] is a behavioral diagram that is a graphical formalism used to capture system requirements during the early stages of design. MSCs are particularly useful to describe domains such as telecommunications where message passing is significant. An extension of MSCs, called Live Sequence Charts (LSCs), depicts the behavior noted in the requirements explicitly, and therefore contributes more than MSCs toward code generation [Harel]. LSCs enable the additional differentiation and depiction of possible, necessary and forbidden behaviors.

Amyot describes the User Requirements Notation (URN) for visualizing and analyzing requirements and he argues the need for a formally defined notation for capture and analysis of

requirements [Amyot2003]. URN is actually two notations: Goal-oriented Requirement Language (GRL) for describing goals and non-functional requirements and Use Case Maps (UCMs) for scenarios. GRL is used to define business or system goals and to evaluate alternatives for achieving the goals with explicit rationales for choices. A main focus of the RE/TRAC research is on the formation and maintenance of use cases by the stakeholders to capture and describe the behavioral qualities of the system rather than on goal discovery.

Amyot notes that there are many notations useful for describing the behavioral qualities of a system and most are variations of Message Sequence Charts (MSC). UML defines a similar notation (syntax and semantics) for its Sequence Diagrams. Many of these notations employ messaging and inter-component interactions, which may be too detailed for requirements engineering. Amyot proposes that Use Case Maps (UCMs) are practical for illustrating operational scenarios and functional requirements [Amyot2003].

UCMs (see Appendix D [Amyot2005]) are relevant to the research because of their simplicity and inherent understandability by the stakeholder. UCMs avoid expressing component interactions as message exchanges. Moreover, UCMs enable incremental development and integration of scenarios for customizing the core set of requirements for a specific application within the business domain. UCMs are similar to some UML diagrams in that they can express forks, joins, conditionals, as well as concurrency and partial ordering of responsibilities. Additionally, UCMs enable the representation of software components in an abstract and generic manner [Amyot2003].

Amyot states that URN –FR (functional requirements) / UCMs can replace UML use case and deployment diagrams. Rather than supplant use case diagrams, the research is based on the hypothesis that the stakeholder will benefit by having another simplified graphical representation

to substantiate the textual representation of use cases. For business applications, the UCM provides a similar functionality, as do workflow diagrams. The UCM and the corresponding use case declarative description must be consistent [Amyot2003].

Message Sequence Charts (MSC) and Use Case Maps (UCM) employ easy to understand graphical representations of requirements. The semiformal graphical notations are presented in a simple manner and the semantics is often intuitive. Both notationally depict use cases as components, and depict alternative paths in the execution of various scenarios. We too describe a semiformal graphical notation. Like MSCs and UCMs, the research uses a large-grain approach to formal method specification. Large-grain refers to the size of the atomic parts in the depiction rather than the size of the system. Small-grain methods are used at the lower level of statements and in small programs. Huge-grain methods are used to depict very large systems whose components may be systems themselves [LuqGog]. However, the research differs from MSCs and UCMs; it is more general and therefore more versatile in the application of the notation than in the representation and specification of requirements. This research has a larger grain approach that enables quicker comprehension of the dependencies between modules or components. Additionally this specification approach addresses traceability.

3.3 Lightweight Behavioral Notations and Diagrams

In comparision to the use of UCMs, Dumas and Hofstede [DumHof] have employed a type of activity diagram that is simplified and incorporates similar functionality. Fickas, Beauchamp and Mamy [Fickas et al] represent requirements as event trees in a similar manner to the workflow specification. Anton and Potts [AntPot] describe several specification formalisms used for task analysis to depict Human Computer Interaction (HCI): Operational Sequence Diagrams (OSD) which use a visual language similar to flowgraphs, GOMS (Goals, Operators,

Methods, Selection Rules) for user/task oriented hierarchal descriptions of the methods needed to attain a goal and User Action Notation (UAN) which uses a tabular notation for describing human-computer dialogues. UML state machines (state charts), interaction and activity diagrams were discussed earlier. There are numerous notations for modeling the behavioral nature of a software system. All of the above in this section, including UCM, might be called lightweight behavioral diagrams because they often lack the detail needed for code generation. The notation developed and illustrated in RE/TRAC is not dependent on the detailed activities within the use cases and is not classified as a behavioral diagram. The depiction of dependencies between entities (use cases) as based on include relationships, extend relationships, and replacement (from refinement) relationships is the focus of this research.

3.4 Requirements State Machine Language (RSML)

The Requirements State Machine Language (RSML) was first described for the specification of safety-critical embedded control systems [Lev et al]. RSML was designed for readability and understandability by the users of the system in requirements specification and not by computer professionals, as is a goal in this research. RSML is similar to statecharts in that it supports parallelism, hierarchies, and guarded transitions. Like RE/TRAC, RSML is described using a static syntax and a semantic description of the next-state mappings. The hierarchical structure of the RSML state machine relates to the ordering of the states, identification of common parents, maintaining the global state, and state changes. RSML defines a component-based hierarchy as does RE/TRAC, but primitives in RSML model events and relationships represent next state mappings. In RE/TRAC the next-state mapping is always a result of a revision of a use case either by refinement and/or changes in associated use case relationships. Therefore RE/TRAC while presenting a record of change is less of a behavioral diagram than

RSML. The RSML graphical notation begins with an initial global state and as transitions occur the global state changes to depict the input and output histories of the machine. Similarly, RE/TRAC diagrams contain a history of the change to the initial state of the use case.

Heimdahl and Keenan use a RSML specification to generate executable code [HeiKee]. Changes are made to the specification during refinement and not to the source code for regeneration of the executable code. The low-level RSML specification needed for code generation necessitated an auxiliary tabular representation for guard conditions. RE/TRAC on the other hand is not intended as a specification for code generation but is useful for management of use case documents in a larger-grained specification of requirements.

3.5 Natural Language to Depict Use Cases

In this research, we assume the stakeholder is a non-technical domain expert. As a result, the requirements will be described using natural language. However, the use of natural language (NL) will create problems of ambiguity, and NL is complex in terms of syntax and semantics. For instance, homonyms produce lexical ambiguity, and structural ambiguity occurs when a sentence has two or meanings based on the sentence structure trees.

A formal language may be used in place of a natural language to alleviate some of these problems by using variables (V_0, V_1…) , logic symbols(¬, Λ…), function symbols and predicate symbols. Not only is a formal language difficult to understand by the stakeholder, but also documents written in a formal language are not acceptable as contractual documents. A solution is to use a controlled language that restricts the natural language in order to reduce the size of the language, the complexity, and inherent ambiguity. The control language will constrain the grammar, style, and lexicon, while providing the benefits of a readable text.

Li constructs use cases according to similar constraints in a controlled language. The controlled language is similar to the structural constructs used in programming languages including if statements, while statements, and currency directives [Li]. Thirteen patterns or syntactic structures of simple sentences describe interactions and responses. A parser is employed that identifies static information including classes, objects, and attributes. The parser gleans dynamic information from the textual accounts, including operations and message sends. It separates the sentence parts and matches the sentence to a sentence type with predetermined parts of speech in order to deduce subjects, verbs, and objects. A message is determined, for example, if the sentence is established to have a subject and verb but no object.

RE/TRAC, as described in this dissertation, relies on previous work in the area of the declarative representation of the use cases, such as using restricted natural language and describing the domain entities used in the textual use cases in the dictionary.

3.6 Model Grammars, Prepositional Connectives and Prepositional Calculus

The deficiencies of a sound semantic basis for such methods as Booch's object-oriented design are noted [AchSch]. The method described, Fox, takes a formal approach to modeling and notation specification [AchSch] and is based on Object-Z which uses an object oriented approach to the formal specification Z. A specification written in Z employs the form of mathematical proofs [Ince]. Object-Z's logic is based on sequent calculus where constraints on inference rules are given using meta-functions [Smith]. A meta-function returns information from the specification text. A description of the transformational semantics permits notations in Fox to be extended [AchSch]. This research employs a graphical model as well as textual means for specification. Likewise this research examined a formal representation of the constraints in the local environment of a use case and also the global environment. [AchSch] demonstrate formal

methods for developing good analysis and design specifications. Their research assumes that the requirement documents are complete. We base our research on a combination of formal and semiformal means to specify the requirements as described in use cases. These works in formal specifications influenced this dissertation work.

First-order logic can be applied to validate hierarchical product line models [Mannion], which has influenced this research. The product-line requirements are added gradually and therefore in an incremental manner. Each requirement in the product line is represented as an atom, and each relationship between requirements is portrayed as a logical expression. In totality, a logical expression can be developed for representing a product-line. A variety of relationships between the entities are expressed: mutual exclusion, choose one or more from a list, optional (zero or more chosen). [Mannion] uses prepositional connectives and propositional calculus to represent subgraphs, dependency relations, and option discriminants (variations in requirements). Given a product-line graphical model, the logical expression can be derived from the model and then evaluated for validity. Rather than expressing the graph as a logical expression, this research uses a string textual grammar to describe a valid graph. In textual grammars, the only relationship between the objects is “immediately precedes” [Mar et al], which corresponds to the top-down ordering of the entities in the proposed hierarchical diagram. In addition, the phrase-structure of a textual grammar can be used to represent the include and extend relationships in a succinct manner, and existing efficient parsing algorithms are available for parsing of textual strings.

A feature model that is a hierarchically arranged diagram set of features [Mannion] has been described [Batory]. After describing the model using a graph and a grammar, productions in the grammar are associated with prepositional formulas [Batory]. Our research is similar to this work in that a graphical notation of a hierarchical representation is represented with a diagram

and with a grammar. This research differs in that each refinement in the diagram is a replacement of a previous entity. Batory distinguishes relationships of: and, alternative or mandatory, and optional [Batory]. Our research initially addresses the refinement, include, and extend relationships.

3.7 Traceablity

The ability to record changes to entities over time is often needed. Most definitions of requirements traceability (RT) include in the definition that the requirements are tracked from the development and specification through deployment and use. RT also refers to the tracking of the changes to a requirement in both a forwards and backwards direction, as well as the ability to describe the requirement adequately [GotFin].

Timeline demarcations are noteworthy, and the following factors are important: the status of the requirements before acceptance, the status of the requirements upon acceptance (the baseline), and the status of the requirements as evolutionary changes occur after acceptance. RE/TRAC supports a representation depicting the temporal order, and it maintains a record of the requirement version.

By restricting the application of the method to a narrowly described domain, the core set of requirements will have been described from the onset. This description reduces problems that occur because of lack of support for pre-requirements specifications (pre-RS)]. Pre-RS involves the elicitation and formulation of requirements prior to requirements specification which culminates in the writing of the requirements specification document. Inadequate or delinquent Pre-RS traceability results in most of the problems with poor overall RT according to [GotFin]. The post requirement specifications (post-RS) traceability involves a forward and backward tracing to the initial baseline requirements specification. This research supports RT during all

time intervals and also permits the forward and backward traceability of the requirement changes as embodied in use cases. The core set of use cases developed for commonality and variability within a domain are considered the baseline of the system under study.

3.7.1 Overview of RT Techniques

There are various techniques for providing RT [GotFin]:

1. cross referencing schemes – the ability to store links to other requirements and documents
2. key phrase dependencies – the selecting of requirement data via information retrieval techniques
3. templates – essentially the use of forms for maintaining textual requirements written in natural language for documentation
4. RT matrices (RTM) – used to associate requirements with software development artifacts
5. matrix sequences – maintains RTM in a temporal order
6. hypertext – links in requirements to related documents
7. integration documents – merging related documents
8. assumptions-based truth maintenance networks – knowledge-based design support systems using artificial intelligence techniques and
9. constraint networks – restructures the documentation based on constraints.

Based on the broad techniques, our research applied to RT would be considered a cross-referencing scheme that links use cases described using templates and depicts how the use cases

are related and integrated over time. An implementation based on the method defined in the research would generate tracings automatically.

3.7.2 Formal Methods for RT

[Pinheiro] describes the formal and informal features of requirements tracing. Likewise we examined the use of both formal and informal techniques. [Pinheiro] provides a brief overview of specification languages useful for requirements specification but notes that many are not designed primarily for requirements tracing. The model introduced in our research is not described as a fully detailed specification language because it is for modeling information at a more abstract level. RE/TRAC is focused on the logging of the occurrence of change rather than the actual changes that occur. It maintains the documentation of an evolutionary process. Because this research is restrictive in its application, the method is best described as a tracing mechanization as opposed to a language for specification. Expressed concisely, this research method supports a specification process.

RE/TRAC employs a formal language, BNF meta-language, to describe the textual notation using a context-free language which differs significantly from database query languages. [Pinheiro] also uses a similar language called TOOR to describe module structures and depict patterns in their relationships as regular expressions.

[Pinheiro] utilizes an object-oriented approach. RE/TRAC does not use an object-oriented approach for the graphical model of the visual diagram, and the grammar for the textual representation. [Pinheiro] uses a relations class identifier, Derive, to denote how the objects are related. The language described in [Pinheiro] is used for linking a broad range of documents related by shared references to a requirement. Users of TOOR may assign custom definable relations of the derive and refine types. Our research restricts all relationships to the include,

extend, and refine relationships between entities, and is therefore explicit and restrictive in the allowable relationships. This restraint is in accordance with our goal of a simple comprehensible depiction. The use of abstract relationships depicted by the derive and refine types of TOOR are not seen as useful in the context of use cases at this time. [Pinheiro] uses the term "trace" to mean that a requirement is traced between various documents. This research uses the term to describe the linkage of an older form of an entity and a newer one, including the relationships that the entity has with other entities at a point in time. This research relies on previous work in information retrieval to identify the actual differences.

TOOR enables the recording of peoples' views or opinions concerning the requirements. Such added functionality adds to the complexity and contributes to the difficulty in understanding the language. TOOR would be especially useful for requirements elaboration. However, our research describes a simpler language, useful when the entities are very nearly specified as in a domain model and the evolution of the entities is uncomplicated. The tracing of the entities in our research are automatic as changes are made to an entity. Like [PinGog], the research supplements the grammar to describe the entities and their relationships with additional assumptions, definitions and propositions.

[DorFly] define ARTS which presents a hierarchical structure for linking requirements and permits searching based on the attributes of requirements. The method described in our research, while hierarchical in structure, does not describe searching. Our research method is general-purpose and useful where information needs to be stored in a hierarchical manner with the before mentioned relationships noted over time. In our research, querying the use cases is made easier by the proposed structuring of the integration of the use cases.

3.8 Miscellaneous Diagrams

RE/TRAC shares some similarity to other static diagrams such as the configuration model which depicts modules and/or subsystems model and to the cooperation diagram [ChoReg]. In [ChoReg] the use case diagram has been amended by using dotted connector lines to indicate the include relationship but the diagram is not hierarchical. The requirements specification described in [ChoReg] is written in both diagrammatic and textual means with a formal specification as a foundation based on the specification language CASL (Common Algebraic Specification Language). They do not address traceability and are not specific to the use case view but to the context view. A hierarchical depiction of use case generalization is shown in [Dion et al] but the depiction does not address traceability; the generalization is not a refinement. A footprint graph is introduced in [EgyGrn] to trace requirements which overlap in different views and models. However the footgraph is modeling individual requirements rather than the tracing of individual documents in which the requirements are embedded. Finally, in UML use case diagrams textual annotations denote dependent use cases. These are called stereotypes and are denoted as <<include>> and << extends>>. RE/TRAC simplifies the cluttered appearance by using simple shapes and line patterns instead of textual labels.

3.9 Summary

RE/TRAC presents a simplified notation and approach representing evolution of documents. The work described in [Batory et al] is inspirational in that it suggests a hierarchical representation for classes that may be adapted for the representation of use cases or other entity types. A difficulty in any requirements specification method is ensuring that requirements are complete, consistent, correct and non-ambiguous. Issues of completeness and ambiguity are especially difficult to address in natural language representations. Our research addresses

consistency issues that arise during the evolutionary process that are related to version control and active status.

Astesino and Reggio proposed an incremental method for specification of requirements. While the textual description of the use cases serves as the essential representation of the requirements, this research presents a visual language, described with a syntactic and semantic language specification, to facilitate requirements specification by the stakeholder. The notation is used to describe use cases and their relationships at a higher level of abstraction than textual descriptions. Likewise, this notation is straightforward to be easily understood by the non-technical user. This research describes an incremental method including allowable actions and constraints for recording the changes to a set of entities, which may be applied to the refinement of use cases. See table 1 for a comparison of some of the most common graphical representations of requirements. In table 1, diagram features are characterized as hierarchical (H), using components (C), and employing traceability (T). The degree of behavior is described as static (S), having limited dynamics (L), or descriptive of behavioral actions (B). The complexity rating is applied based on a clear and intuitive interpretation of meaning and on the number of primitives and connector types, and labeling. The complexity range is from 1 to 5 with 1 being low complexity and 5 highest complexities in the comparison.

Message Sequence Charts, UML Sequence Diagrams and Use Case Maps are some of the better-known notations for requirements specification. There are others that warranted study such as Operational Sequence Diagrams, GOMS (Goals, Operators, Methods, Selections Rules) and the User Action Notation (UAN). While the purpose of each notation is quite different from this research, the semi-formal to formal descriptions of the languages describing the notations influenced RE/TRAC.

Table 1: Comparison of Common Graphical Methods for Requirements Specification

Diagram	**Diagram Features**			**Degree of Behavior**	**Complexity**	**Usage**
Hierarchical Refinement of Classes [Batory et al]	H		T	S	1 does not label connectors	Depicts refinement of classes. Refinements are related to class generalizations. Traceability is viewed as levels when classes are refined via generalization of the class. Vertical, top-down ordering.
Acyclic Call Graph [BatO'Mal]	H	C		S	1 does not label connectors	Depicts function or method call relationships. Diagrams are read top-down, left-to-right for sequence interpretation of the calls. Vertical, top-down ordering.
[AstReg2002a] [AstReg2002b]				S	5	Structures and represents requirements specification artifacts in general, multi-view, use-case driven, UML-based (object-oriented). Systematic approach developed using UML.
Message Sequence Charts (MSCs) [Harel]	H	C		B	2	Visual formalism for capturing systems requirements as scenarios. Similar to UML sequence diagrams. Useful in system requirements capture.
User Requirements Notation: Use Case Maps (UCMs) [Amyot2003]	H	C		B	3	Standard visual notation used for specifying functional and non-functional requirements. Used for use case formulation, high-level architectural design and test case generation. Notation uses start points (pre-conditions), connectors and end points (post-conditions). Connector lines (paths) may be labeled with responsibilities.
Requirements State Machine Language (RSML) [Lev et al]	H	C	T	B	2	Uses a graphical hierarchical RSML specification which describes dynamic behavior defined by transitions and events.
Use Case Diagram (amended) [ChoReg]				S	1	Diagram and user-friendly notation that uses a NL-like language for specification of the use cases.

There are numerous techniques for traceability of requirements, however we found no techniques that supported version control of use cases including traceability. A formal BNF grammar facilitates the internal structuring and tracking of the requirements evolution over time. Employing a BNF grammar will enable varied depictions based on the language. Using a diagrammatic notation as an interpretation dependent on a sentence from the grammar will benefit

the non-technical stakeholder by providing an alternative but coherent viewing of the tracing information. Consistency is maintained between meaning in a RE/TRAC-CF sentence, its diagrammatical interpretation of the sentence and the evolutionary changes to a use case. Clearly the use of grammars to describe diagrams is not novel, but it is a useful formal method in this research which uses a unique combination of semi-formal and formal means to depict RE/TRAC diagrams.

4. The RE/TRAC and RE/TRAC-SEM Models

4.1 The Static Syntax

Static syntax describes all acceptable visual sentences in a language and the rules that enable a decision of either accept or reject. Visual sentences or individual diagrams are assembled using a vocabulary consisting of a set of visual primitives; visual dimensions such as shape, color, and juxtaposition; and relations between the primitives. Sentences in the visual language (VL) are described by the static semantics and correspond to states in the application domain [NarHüb].

A goal in developing the RE/TRAC visual language [DouCar2006] is to be able to create diagrams with an uncluttered appearance, few annotations, and whose meaning was easily inferred. The primitives used in the RE/TRAC visual model are shown in Figure 3. The base case which is the head of a diagram serves as an introduction to a particular use case hierarchy and is symbolized by the oval (Figure 3a). The oval and square (Figure 3b) are model symbols representing documents containing meta-data information such as the active status, name, unique document identification label, creation date and link to the NL textual document. A version instance is shown by a square.

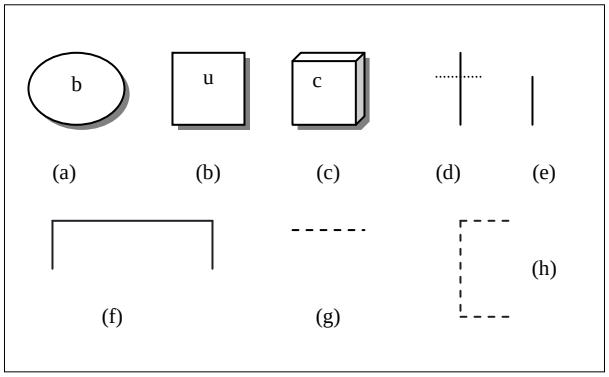


Figure 3 RE/TRAC Visual Language Primitives

The vertical refinement connector is shown in Figure 3d. When a single case is refined by two use cases, the refinement connector attaches to the bridge (Figure 3f) to represent a split. If there are more than two child cases, then the bridge is expanded and vertical connector lines (Figure 3e) are inserted. Use cases related via the extension dependency are connected with a horizontal dashed line (Figure 3g), and the dashed bridge (Figure 3h) is used with the extension connector for depicting multiple extension relationships.

Figure 4 shows the allowable relationships. For brevity, only refinements (Figures 4a, 4b) of one child and single extensions (Figure 4c) are provided. A base use case has no parent as shown in figure 4a. Dependent relationships except for refinement are not permitted for the base case in a diagram instance. The base use case is refined by zero or more use cases at the next level (level 1). Likewise, any singular use case (Figure 4b) may be refined by zero or more use cases at each subsequent level; any use case (Figure 4c) may also be extended by zero or more use cases. A refinement can only be applied to a leaf use case in its hierarchy because a leaf represents a current active use case version. All other versions are frozen. The nesting of primitive squares (Figure 4b) depicts an inclusive relationship. Only a single nested use case primitive is shown (Figure 4d). A use case may include zero or more use cases. According to the rules described, a dependent use case may have dependent relationships and also have refinements. Obviously, a use case cannot include or extend itself or versions of itself.

In a diagram instance, the size and juxtaposition of the primitives may vary for visual appeal. In compound version instances that result from successive nesting and/or repeated refinements to dependent use cases, a component (Figure 3c) cubic symbol is used to compress the presentation. The component relationships are omitted in Figure 4 but component primitives are substitutable for all version squares except in Figure 4d. A use case may include a

component, but a component cube cannot have a dependent use case nor component. Active leaf nodes are depicted in yellow.

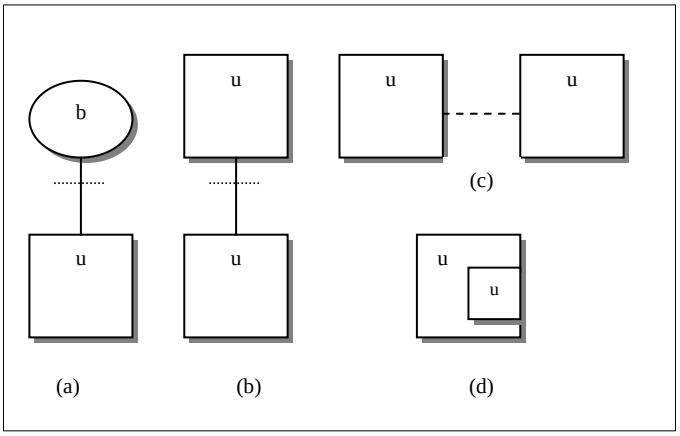


Figure 4 RE/TRAC Visual Language Relationships

A use case label is used for uniquely identifying both active and archived versions in the system. As shown in the example diagrams (Figures 5a, 5b, 6) the superscripts identify the base cases of b^1, b^4 and b^9; subscripts denote levels; and concatenated sets of colons (':') with enumeration values denote siblings. The level zero subscript is optional. Use cases related via refinement always share the same superscript. Use cases are described as either core (fundamental) use cases or dependent use cases to a core use case. A dependent use case is a core use case serving in context as in a supplementary use case to a core use case. Refinement labels for a core use case must have ordered, unique and consecutively numbered levels. Level labels for refinements depicted in dependencies are not necessarily consecutive. A component label for a component primitive adopts the superscript and subscript string of the use case for which it is replacing. The component primitive labeled c^9_2 in Figure 6 replaces use case primitive u^9_2 and all of its dependencies.

In the RE/TRAC diagram for use case hierarchy b^1 (Figure 5a) there are two levels. Base case b^1_0 was first described by u^1_1 and then at a later time, u^1_1 was refined by a split into versions $u^1_{2:1}$ and $u^1_{2:2}$. Use case $u^1_{2:1}$ is related to u^{43}_1 via inclusion. At the current state shown, b^1 is described by the union of the set *CoreLeafActive*^{1} of active leaf use case versions in the b^1 core and the set *dom TotalActiveDependenciesLeafCoreLeaf* ^{1} of all active leaf case versions of all dependencies in the set *Dependent* of b^1 where *~(Core*^{1} ∩ *Dependent* ^{1}*)*. In this example,

CoreLeafActive ^{1} = {$u^1_{2:1}$, $u^1_{2:2}$},

TotalActiveDependenciesLeafCoreLeaf ^{1} ={ u^{43}_1 , $u^1_{2:1}$ },

CoreLeafActive ^{1} ∪ *dom TotalActiveDependenciesLeafCoreLeaf* ^{1} = { $u^1_{2:1}$, u^{43}_1, $u^1_{2:2}$ }

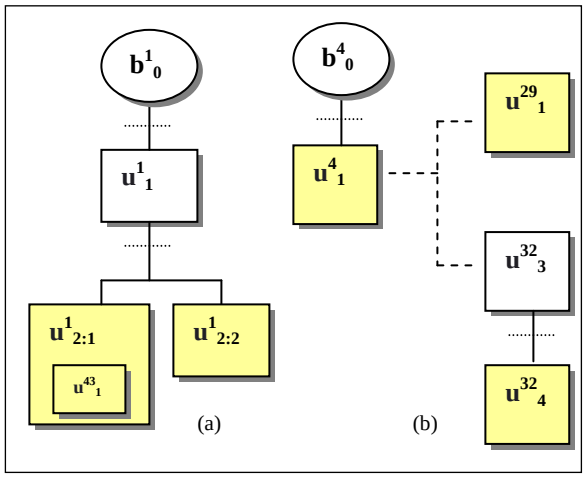


Figure 5 RE/TRAC Diagram Examples

The RE/TRAC diagram for b^4 (Figure 5b) shows that the core was first described by a version u^4_1 that referenced two use cases u^{29}_1 and u^{32}_3 both via extension. At an even later time, u^{32}_4, a newer version of u^{32}, was selected. Note that in this case, updating the dependency did not require a new version of u^4_1. The resulting *CoreLeafActive*^{4} and *TotalActiveDependenciesLeafCoreLeaf*^{4} are:

$CoreLeafActive^4 = \{u^4_1\}$,

$TotalActiveDependenciesLeafCoreLeaf^4 = \{ (u^{29}_1, u^4_1) (u^{32}_4 , u^4_1) \}$

$CoreLeafActive^4 \cup dom\ TotalActiveDependenciesLeafCoreLeaf^4 = \{ u^4_1, u^{29}_1, u^{32}_4\}$

Figure 6 indicates that b^9 has had two successive refinements, versions u^9_1 and u^9_2. The later revision included the updated version level 3 of u^2 which was even later updated to version level 5. The use of the component primitive simplifies the diagram instance. The resulting $CoreLeafActive^9$ and $TotalActiveDependenciesLeafCoreLeaf^9$ are:

$CoreLeafActive^9 = \{ u^9_2 \}$,

$TotalActiveDependenciesLeafCoreLeaf^9 = \{ (u^2_5 , u^9_2) (u^{44}_8 , u^9_2)\}$

$CoreLeafActive^9 \cup dom\ TotalActiveDependenciesLeafCoreLeaf^9 = \{ u^9_2, u^2_5, u^{44}_8\}$

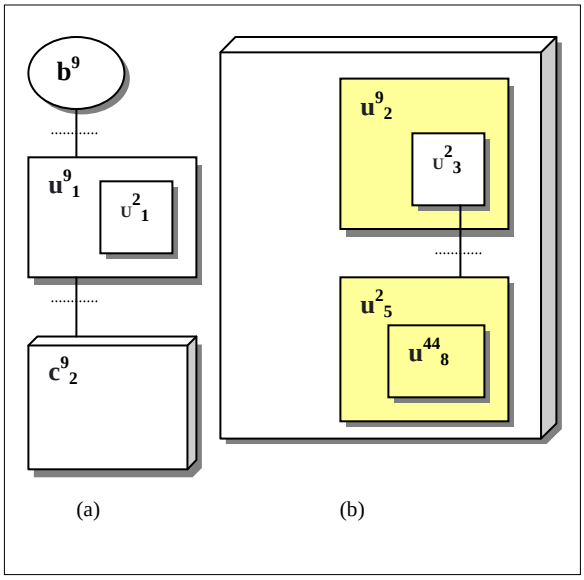


Figure 6 Diagram and Detailed Component Diagram

The set representation provides the foundation for the definition of the dynamic semantics of RE/TRAC-SEM given in 4.2.

Figure 7 shows a Parallel RE/TRAC Diagram modeled after [Batory et al] for a hypothetical domain, D. All RE/TRAC diagrams for D are shown simultaneously including core use cases and dependent use cases. The example diagram consists of core use cases 1, 2, and 3 and dependent use cases of 4 and 5. All base use cases have been refined at level 1; use cases 1 and 2 have been refined to a maximum level 2 and use case 3 refined to a maximum level of 3. The yellow coloring indicates that leaves in all use cases including related dependencies are active.

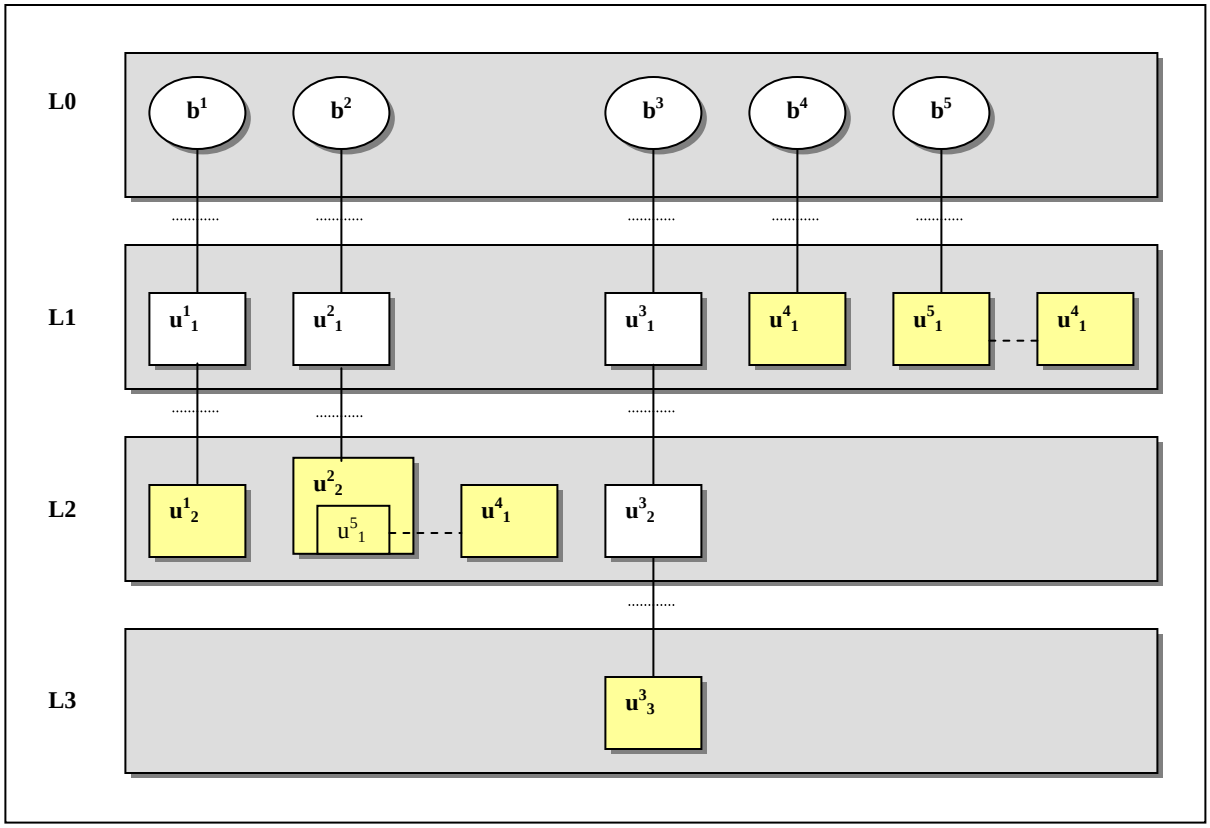


Figure 7 Parallel RE/TRAC Diagram

The active state of the domain is:

$$ActiveState = CoreLeafActive^{1} \cup CoreLeafActive^{2} \cup CoreLeafActive^{3} \cup ActiveDependentChildLeafCoreLeaf^{2} \cup dom\ dom\ ActiveDependentToDependentChildLeafCoreLeaf$$

$$= \{ u^{1}_{2} \} \cup \{ u^{2}_{2} \} \cup \{ u^{3}_{3} \} \cup \{ u^{5}_{1} \} \cup \{ u^{4}_{1} \}$$

$$= \{ u^{1}_{2}, u^{2}_{2}, u^{3}_{3}, u^{5}_{1}, u^{4}_{1} \}$$

Levels in the collective diagram depict discrete time events (refinements) in the use case specification process for an individual use case. While a Parallel RE/TRAC Diagram may not be practical for complex domain applications, its use here shows a universal view including interrelationships and depicts the consistencies between the versions. In other words, a use case cannot include or extend a use case version that does not exist, and a use case may only refine the most current use case in a base use case hierarchy.

4.2 The Semantic Model RE/TRAC-SEM

Several specification techniques were explored in this research for describing the semantic model for RE/TRAC. While formal specifications are beneficial for precise and unambiguous descriptions they are difficult and often impractical to use. So the ambition of the research was to leverage the advantages of formal specifications where the benefits could best be achieved yet construct a specification that was easily understandable. The semantics were first described using UML class diagrams which quickly became cluttered and chaotic with numerous constraints and cardinality specifications. The use of UML object oriented class diagrams for the semantic specification appeared more of a design specification and therefore compromised the predilection for a platform independent model. Likewise the uses of prepositional calculus, predicate calculus, or the specification language Z were found to be cumbersome for the problem domain of use case document management with traceability.

A combination of specification models is used to describe the supporting ontology RE/TRAC-SEM for RE/TRAC, including set theory, a grammatical formalism, and informal verbal definitions [DouCar2007]. As a hybrid model, the language representation uses the relevant model features of each to best describe the ontology [Kal et al]. The grammatical formalism, RE/TRAC-CF, is a string language specification and provides a strong formal

structure to the ontology. RE/TRAC-CF describes the relationships of the use cases independently of the justification and relationships of the RE/TRAC primitives in order that the visual interpretation may vary. Because the use case documents can be grouped by the core identification, the use of set theory was a logical choice for describing the constraints between the use cases.

Feature models are employed to describe members of a product-line, where a feature is a distinctive characterized aspect or quality of a system. A product-line denotes related marketable goods with variability in features such as size, color, or other qualities. A feature diagram (FD) is a graphical way to represent a feature model that has a tree-like structure with primitive features as the leaves and compound features as interior nodes [Kang et al]. In [JonViss] the advantages of converting a feature diagram to a grammar are described. Similar advantages are perceived in this research which employs syntax tools for the specification of requirements.

This research also takes advantage of existing string grammar theory. While the string grammar approach may be restrictive, the domain of use case modeling is narrow also. The RE/TRAC graphical depiction of relationships between the primitives for a single core use case can be described succinctly at a higher level of abstraction using a context-free grammar. Efficient parsing algorithms exist for LL(k) or LR(k) type grammars like RE/TRAC-CF which employ top-down and bottom-up parsing respectively and can be made to work deterministically by looking ahead k symbols. Parsing a sentence, S, in RE/TRAC-CF determines if S has adhered to the syntactical rules and permits the extraction of the meaning within S. This formalism requires an initial lexical analysis phase to recognize the relationships between the terminal symbols and subsequent top-down generation of a graph instance [Mar et al].

4.2.1 RE/TRAC-CF

A context-free grammar consists of a quadruple (T, N, S, R) where: T is a finite set of terminal symbols, N is a finite set of nonterminals where N and T are disjoint, S is a unique starting symbol ($S \subset N$) and P is a finite set of productions of the form, $A \rightarrow \beta$, where A is a non-terminal and β is a non-empty string of symbols. A sequential textual sentence, S, that conforms to the RE/TRAC-CF grammar, describes the history of change including dependencies for a core use case and can be used to generate its corresponding graphical representation, S'. The grammar, shown in Figure 8, is represented in Baccus Naur Form (BNF); a detailed description of the production rules is provided in table 2.

1.) S::= b | bA
2.) A::= <Z>
3.) Z ::= ZZ | (T) | (TA)
4.) T ::= u | uE | uG
5.) G ::= I | IE
6.) I ::= [Z]
7.) E ::= {Z}

Figure 8 RE/TRAC-CF Grammar

The language is recursive as use cases may refine uses cases, use cases may include or extend other use cases which can then include use cases which can be refined. When used in an instance, S, the '<' terminal begins a refinement relationship and the '>' terminal ends the refinement relationship. The left and right parentheses group a use case term including all of its subsequent refinements and its dependencies. The left and right square brackets set apart a use case term including all of its subsequent refinements and dependencies to be included; curly braces denote a use case term related to its parent via extension. The use case identification superscripts and subscript levels are not noted in the grammar associated with the terminals b

and u. References to the numeric children identifiers are omitted also in order to keep the grammar simple and context-free.

Table 2 Explanations for the RE/TRAC-CF Production Rules

	RULE	**EXPLANATION**
1a	S ::=b	b is base case in a core
1b	S ::=bA	Base my be refined
2	A ::=<Z>	'<' indicates refinement; '>' ends refinement
3a	Z ::= ZZ	Each use case term may be refined by one or more use case terms – a list
3b	A ::= (T)	'(' indicates a use case term follows
3c	A ::= (TA)	Use case term is refined
4a	T ::= u	Term is "simple"
4b	T ::=uE	Term is "compound" with a dependent extension relationship E
4c	T ::= uG	Term is "compound" with a dependent relationship possibly inclusion or both inclusion and extension
5a	G ::= I	Term is "compound" with a dependent inclusion relationship
5b	G ::= IE	Term is "compound" with both a dependent inclusion and extension relationships. Rule forces inclusion as higher precedence.
6a	I ::= [Z]	'[' indicates inclusion relationship. Z may be one or more included terms
6b	E ::= {Z}	'{' indicates extension relationship. Z may be one of more included terms

The following are the RE/TRAC-CF representations for the corresponding graphical representations from the previous graph models:

Figure 5a:
$\mathbf{S^1 = b^1_0 < (u^1_1 < (u^1_{2:1}\ [(u^{43}_1)])\ (u^1_{2:2}) >) >}$

Figure 5b:
$\mathbf{S^4 = b^4_0 < (u^4_1\ \{(u^{29}_1)\ (u^{32}_3 < (u^{32}_4) >)\}) >}$

Figure 6:
$\mathbf{S^9 = b^9_0 <(u^9_1[(u^2_1)]<(u^9_2[(u^2_3<(u^2_5[(u^{44}_8)])>)])>)>}$

Figure 7:

$S^1 = b^1_0<(u^1_1<(u^1_2)>)>$

$S^2 = b^2_0<(u^2_1<(u^2_2[(u^5_1 \{(u^4_1)\})]) >)>$

$S^3 = b^3_0<(u^3_1<(u^3_2 <(u^3_3)>)>)>$

$S^4 = b^4_0<(u^4_1)>$

$S^5 = b^5_0<(u^5_1\{(u^4_1)\}) >$

Additional RE/TRAC-CF examples with their corresponding RE/TRAC depiction can be found in Appendix E.

4.2.2 Dynamic Semantics

Changes in the application domain motivate changes to RE/TRAC-CF sentences. The dynamic syntax describes how a sentence is transformed. The dynamic semantics describes the conditional changes to the domain's objects, attributes and relationships and their mappings to conditional changes in the representation [NarHüb]. The dynamic semantics is partially embodied in meta-restrictions which describe the pre or post conditions associated with changes to a RE/TRAC-CF sentence. Meta-restrictions describe the dynamic semantics by defining invariants that must be upheld to preserve the integrity of the system [Rodri et al].

The semantics of RE/TRAC is described by definitions of terms, set definitions and meta-restriction descriptions. The term parent is used to describe the head of a refinement or dependent relationship. A child is described as a descendant of a use case and is associated to its parent via refinement. A parent is the older version of a child use case related by refinement. Multiple children of a parent are siblings. A dependent use case is related to its core parent via inclusion or extension. A use case that has not been refined is called a leaf. A use case is part of a dependency legacy if it is part of an inheritance chain including dependencies that leads upwards in the hierarchy to a dependent use case in a core, $Core^k$. In Figure 5b, u^{32}_3 and u^{32}_4 compose an extension dependency legacy of the $Core^4$ use case u^4_1. In Figure 6b, u^2_3, u^2_5 and

$u^{44}{}_8$ compose the inclusion legacy for *Core*^{9} use case $u^9{}_2$. The set definitions applicable for a given domain and application instance are given in table 3.

Table 3: Set Definitions for RE/TRAC-SEM

1	$U = \{b_0 \cup u_i \mid i{:}N\}$ where b_0 and u_i are use cases. Archived use cases included.
2	$Core^k = \{ b^k{}_0 \cup u^k{}_{v\{:m\}*}{:}U \mid k, m, n, v, z{:}\ N \wedge \exists k{:}1 <= k <= n \wedge \exists v{:}1 <= v <= z,\}$ where k denotes the core identification of n number of use cases, v denotes the version or level instance, ":m" is the child identification string and z is the maximum number of refinement levels of a use case $b^k{}_0$.
3	$Dependent^k = \{ u^j{}_{v\{:m\}*} : U \mid n{:}N, \exists j{:}N\ 1 <= j <= n \wedge j \neq k\}$ Every u in $Dependent^k$ is part of a dependency legacy of one or more parent use cases in the $Core^k$.
4	$\sim(Core^k \cap Dependent^k)$
5	$DependentChildCore^k{}_{v\{:m\}*} = \{ u^j{}_{v\{:m\}*} : Dependent^k, u^k{}_{v\{:m\}*} : Core^k \mid \exists u^j{}_{v\{:m\}*} : u^j{}_{v\{:m\}*}\ \underline{Child}\ u^k{}_{v\{:m\}*} \cdot (u^j{}_{v\{:m\}*}, u^k{}_{v\{:m\}*})\}$ If $u^j{}_{v\{:m\}*} \in dom\ DependentChildCore^k{}_{v\{:m\}*}$ then u is a dependent child of the parent use case $Core^k{}_{v\{:m\}*}$. Subscript identifier, $_{v\{:m\}*}$, is the particular level identifier for the j^{th} or k^{th} use case and does not indicate equality of the expression.
6	$CoreLeaf^k = \{ u^k{}_{v\{:m\}*} : Core^k \mid \exists u^k{}_{v\{:m\}*}, \neg\exists u^k{}_{x\{:m\}*}\ where\ x\{:m\}* > v\{:m\}* \}$ $CoreLeaf^k$ is the set of leaf use cases in $Core^k$. The comparison operator, >, means there are no refinements of $u^k{}_{v\{:m\}*}$.
7a	$Status = \{"active", "inactive"\}$ $CoreStatus^k = Core^k\ X\ Status$
7b	$CoreStatusActive^k = \{ u^k{}_{v\{:m\}*}{:}\ dom\ CoreStatus^k \mid CoreStatus^k\ \underline{Status}\ "active"\}$
7c	$CoreStatusInactive^k = Core^k - CoreStatusActive^k$
8a	$DependentStatus^k{}_{v\{:m\}*} = DependentChildCore^k{}_{v\{:m\}*}\ X\ Status.$
8b	$DependentStatusActive^k{}_{v\{:m\}*} = \{ (u^j{}_{v\{:m\}*}, u^k{}_{v\{:m\}*}) : dom\ DependentStatus^k{}_{v\{:m\}*} \mid DependentStatus^k{}_{v\{:m\}*}\ \underline{Status}\ "active" \cdot (u^j{}_{v\{:m\}*}, u^k{}_{v\{:m\}*})\}$
8c	$DependentStatusInactive^k{}_{v\{:m\}*} = DependentChildCore^k{}_{v\{:m\}*} - DependentStatusActive^k{}_{v\{:m\}*}$

(Table 3 continued)

9a	$CoreLeafActive^{k} = CoreStatusActive^{k} \cap CoreLeaf^{k}$ The set of active use cases within the set of leaves of $Core^{k}$.
9b	$CoreLeafInactive^{k} = CoreLeaf^{k} - CoreStatusActive^{k}$ The set of inactive use cases within the set of leaves of $Core^{k}$.
10a	$DependentRelation = \{\text{“include”}, \text{“extend”}\}$ $DependentRelationChild^{k}{}_{v\{:m\}^*} = DependentChildCore^{k}{}_{v\{:m\}^*} \; X \; DependentRelation$
10b	$DependentIncludeChildCore^{k}{}_{v\{:m\}^*} =$ $\{ (u^{j}{}_{v\{:m\}}, u^{k}{}_{v\{:m\}^*}) : dom\, DependentRelationChild^{k}{}_{v\{:m\}^*} \mid$ $DependentRelationChild^{k}{}_{v\{:m\}^*} \; \underline{DependentRelation} \; \text{“include”} \cdot (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) \}$
10c	$DependentExtendChildCore^{k}{}_{v\{:m\}^*} =$ $\{ (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) : dom\, DependentRelationChild^{k}{}_{v\{:m\}^*} \mid$ $DependentRelationChild^{k}{}_{v\{:m\}^*} \; \underline{DependentRelation} \; \text{“extend”} \cdot (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) \}$
11a	$DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} =$ $\{ (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) : DependentChildCore^{k}{}_{v\{:m\}^*} \mid$ $\exists (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}), \neg \exists u^{j}{}_{x\{:m\}^*} \; where \; x\{:m\}^* > v\{:m\}^*$ $\wedge \; \exists u^{k}{}_{v\{:m\}^*} \in CoreLeaf^{k} \cdot (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) \}$
11b	$DependentChildLeafIncludeCoreLeaf^{k}{}_{v\{:m\}^*} =$ $\{(u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) : DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} \mid$ $DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} \cap DependentIncludeChildCore^{k}{}_{v\{:m\}^*} \}$
11c	$DependentChildLeafExtendCoreLeaf^{k}{}_{v\{:m\}^*} =$ $\{ (u^{j}{}_{v\{:m\}^*}, u^{k}{}_{v\{:m\}^*}) : DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} \mid$ $DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} \cap DependentExtendChildCore^{k}{}_{v\{:m\}^*} \}$
12a	$ActiveDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} =$ $DependentStatusActive^{k}{}_{v\{:m\}^*} \cap DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*}$
12b	$InactiveDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} =$ $DependentStatusInactive^{k}{}_{v\{:m\}^*} - ActiveDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*}$
13	$DependentToDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} =$ $\{u^{q}{}_{v\{:m\}^*} : Dependent^{k},$ $u^{j}{}_{v\{:m\}^*} : dom\, DependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*},$ $u^{k}{}_{v\{:m\}^*} : CoreLeaf^{k} \mid$ $\exists((u^{q}{}_{v\{:m\}^*}, u^{j}{}_{v\{:m\}^*}), u^{k}{}_{v\{:m\}^*}), \; u^{q}{}_{v\{:m\}^*} \; \underline{Child} \; u^{j}{}_{v\{:m\}^*} \wedge$ $\exists(u^{q}{}_{v\{:m\}^*}, u^{j}{}_{v\{:m\}^*}), u^{k}{}_{v\{:m\}^*}), \; u^{j}{}_{v\{:m\}^*}, \underline{Child} \; u^{k}{}_{v\{:m\}^*} \cdot ((u^{q}{}_{v\{:m\}^*}, u^{j}{}_{v\{:m\}^*}), \; u^{k}{}_{v\{:m\}^*}) \}$
14a	$StatusDependentToDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} =$ $DependentToDependentChildLeafCoreLeaf^{k}{}_{v\{:m\}^*} \; X \; Status$

(Table 3 continued)

14b	$ActiveDependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} =$ $\{ ((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}) :$ $dom\ StatusDependenttoDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \mid$ $StatusDependentToDependentLeafCoreLeaf^{k}_{v\{:m\}^*}\ \underline{DependentRelation}\ \text{“Active”}$ $\cdot ((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*})\}$
14c	$InactiveDependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} =$ $DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} -$ $ActiveDependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$
15	$DependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*} =$ $\{((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}) :$ $DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \mid$ $\exists((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}),\ u^{q}_{v\{:m\}^*}\ \underline{Child}\ u^{j}_{v\{:m\}^*}$ $\wedge\ \exists((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}),\ u^{j}_{v\{:m\}}\ \underline{Child}\ u^{k}_{v\{:m\}^*}$ $\wedge\ \exists((u^{q}_{v\{:m\}^*} u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}), \neg\exists u^{q}_{x\{:m\}^*}\ where\ x_{\{:m\}^*} > v_{\{:m\}^*}$ $\cdot ((u^{q}_{v\{:m\}^*}, u^{r}_{v\{:m\}^*}),\ u^{k}_{v\{:m\}^*})\}$
16a	$ActiveDependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*} =$ $ActiveDependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \cap$ $DependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*}$
16b	$InactiveDependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*} =$ $InActiveDependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \cap$ $DependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*}$
17	$TotalActiveDependenciesLeafCoreLeaf^{k} =$ $ActiveDependentChildLeafCoreLeaf^{k} \cup$ $\{ u^{q}_{v\{:m\}^*} \in dom\ dom\ ActiveDependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*},$ $u^{k}_{v\{:m\}^*} \in ran\ ActiveDependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*} \mid$ $((u^{q}_{v\{:m\}^*} u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}) \in$ $ActiveDependentToDependentLeafChildLeafCoreLeaf^{k}_{v\{:m\}^*} \cdot (u^{q}_{v\{:m\}^*}, u^{k}_{v\{:m\}^*}) \}$ $TotalActiveDependenciesLeafCoreLeaf^{k}$ is the union of all active leaf dependencies in *CoreLeaf*^{k}.
18	$ActiveState = \{ u^{k}_{v\{:m\}^*} \in CoreLeafActive^{k},$ $u^{q}_{v\{:m\}^*} \in dom\ TotalActiveDependenciesLeafCoreLeaf^{k} \mid k:N, q:N \cdot 1 <= k <= n,$ $1 <= q <= n\}$ The current active state of the use case specification, S^{k}, is defined by the union of the active leaf cases in the core and the active leaf cases in the legacies of use cases in *CoreLeafActive*^{k}.

Meta-restrictions on the set entities prescribe limitations on use case refinement, identification, and status alteration in context when there is a change to S^k. There are restrictions on the identification numbering system for refinement levels, for dependent use cases, and for differentiation between siblings. All use cases in $Core^k$ are related via a refinement hierarchy. Refinement labels for core use cases in $Core^k$ must have ordered, unique and consecutively numbered levels. Operations or actions permitted by the dynamic semantics are described as commands.

A RE/TRAC diagram usually expands vertically as refinements are made. Changes can only be made to documents represented at the lowest levels in a diagram: to a use case $\in$ $CoreLeaf^k$ or to the most current dependent use case associated with some use case $\in$ $CoreLeaf^k$. The addition of a dependent relationships may only be linked to a use case $\in$ *CoreLeaf.* Child dependent use cases to the new dependency are automatically associated.

The following parameterized commands are necessary for maintaining consistency among the use case versions and preserving the integrity of the requirements specification:

(1) REFINE_CORE_SINGLE (Figure 9),

(2) REFINE_DEPENDENT_SINGLE_RELATION (Figure 10),

(3) CORE_ADD_RELATION (Figure 11),

(4) SET_CORE_ACTIVE (Figure 12a),

(5) SET_CORE_INACTIVE (Figure 12b),

(6) SET_DEPENDENT_STATUS (Figure 13),

(7) REFINE_CORE_MULTIPLE (Figure 14),

(8) REFINE_DEPENDENT_MULTIPLE_RELATION (Figure 15) and,

(9) UNION_DEPENDENT_DEPENDENT (Figure 16).

The commands make possible the refinement of a core by a single use case or a split into multiple children. A core use case may have a dependent use case added. A core leaf may be set to active or inactive status. Likewise dependents may be refined singly or by multiples and may have their statuses altered. The command to join dependent use cases to another dependents use case is UNION_DEPENDENT_DEPENDENT. The support command, SET_HISTORY, is incorporated to maintain a log of the active/inactive status a use case relative to its core parent. The function, TIME, is employed in conjunction with a history recording.

The commands REFINE_CORE_SINGLE and REFINE_DEPENDENT_SINGLE_RELATION are defined for a refinement of only one child use case. When a core leaf case is refined, the level number is increased by one (i+1) from the parent level (i). The REFINE_CORE_SINGLE command is defined in Figure 9. The newly added core leaf case is set to active status with the command SET_CORE_ACTIVE in Figure 12a.

Pre-Condition: $u^k_{i+1\{:m\}*} \notin CoreLeaf^k$
Command = REFINE_CORE_SINGLE ($u^k_{i\{:m\}*}$, $u^k_{i+1\{:m\}*}$) $\Rightarrow$
$(u^k_{i\{:m\}*} \in CoreLeaf^k \vee CoreLeaf^k = \{\} \Rightarrow$
$(CoreLeaf^k)' = CoreLeaf^k - \{u^k_{i\{:m\}*}\}$
$\wedge (CoreLeaf^k)' = CoreLeaf^k \cup \{u^k_{i+1\{:m\}*}\})$
$\wedge$ SET_CORE_ACTIVE $(u^k_{i+1\{:m\}*}))$.
Post-condition: $u^k_{i\{:m\}*} \notin CoreLeaf^k$

Figure 9 Refine Core Single

Similarly dependent use cases can be refined. However the new level label is not necessarily consecutive but will be ordered and unique. Refinements to $u^j_{v\{:m\}*} \in dom\ DependentChildLeafCoreLeaf^k_{v\{:m\}*}$ maintain the same relationship to the parent core as its

immediate parent. The command REFINE_DEPENDENT_SINGLE_RELATION is described in Figure 10 for the relationships include and extend. The parameter list includes the new dependent use case, its dependent parent, the core parent, and the relation either "include" or 'extend". To enforce traceability, only leaf dependencies directly related to a core leaf parent may be refined. When a dependent use case is refined, the newly added use case is set to the active default status.

Pre-condition: $(u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}) \notin DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$
Command = REFINE_DEPENDENT_SINGLE_RELATION
$(u^{j}_{i\{:m\}^*}, u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}, relation) \Rightarrow$
$((u^{j}_{i\{:m\}^*}, u^{k}_{v\{:m\}^*}) \in DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$
$\wedge x > i \wedge relation \in \{\text{"include"}, \text{"extend"}\} \Rightarrow$

$((DependentChildCore^{k}_{v\{:m\}^*})' = DependentChildCore^{k}_{v\{:m\}^*} \cup \{(u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*})\}$
$\wedge (DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*})' =$
$DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} - \{(u^{j}_{i\{:m\}^*}, u^{k}_{v\{:m\}^*})\}$
$\wedge (DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*})' =$
$DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \cup \{ (u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}) \}$

$\wedge$ UNION_DEPENDENT_DEPENDENT $(u^{j}_{v\{:m\}^*}, u^{k}_{v\{:m\}^*})$
$\wedge$ SET_DEPENDENT_STATUS $((u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}), relation)$

$\wedge (DependentRelationChild^{k}_{v\{:m\}^*})' =$
$DependentRelationChild^{k}_{v\{:m\}^*} \cup \{(u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}), relation))).$
Post-condition: $(u^{j}_{i\{:m\}^*}, u^{k}_{v\{:m\}^*}) \notin DependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$

Figure 10 Dependency Refined

A dependent use case $u^{j}_{v\{:m\}^*}$ is associated with a core use case via the command CORE_ADD_RELATION described in Figure 11.The status of the added dependency (include or extend) is set to active. Use cases may only be included or extended to use cases in *CoreLeaf* ^{k}. All other versions are frozen. When a dependent use case $u^{j}_{v\{:m\}^*}$, is associated with a leaf

core use case $u^{k}_{v\{:\,m\}*}$ by refinement commands or commands to add dependencies to the core, all dependents $u^{q}_{v\{:\,m\}*} \in dom\ DependentToDependentLeafChildLeafCoreLeaf^{\,j}_{v\{:\,m\}*}$ are also associated with $u^{k}_{v\{:\,m\}*}$. The command to associate dependents to a dependent child leaf is UNION_DEPENDENT_DEPENDENT described in Figure 16.

Pre-condition: $(u^{j}_{v}, u^{k}_{v\{:\,m\}*}) \notin DependentChildLeafCoreLeaf^{\,k}_{v\{:\,m\}*}$
Command=CORE_ADD_RELATION $(u^{j}_{v\{:\,m\}*}, u^{k}_{v\{:\,m\}*}, relation) \Rightarrow$
$(u^{k}_{v\{:\,m\}*} \in CoreLeaf^{\,k} \wedge$ relation $\in$ {"include", "extend"} $\Rightarrow$
$((DependentChildCore^{k}_{v\{:\,m\}*})' =$
$DependentChildCore^{\,k}_{v\{:\,m\}*} \cup \{(u^{j}_{v\{:\,m\}*}, u^{k}_{v\{:\,m\}*})\}$
$\wedge (DependentRelationChild^{\,k}_{v\{:\,m\}*})' =$
$DependentRelationChild^{\,k}_{v\{:\,m\}*} \cup \{(u^{j}_{v\{:\,m\}*}, u^{k}_{v\{:\,m\}*}, relation\}$
$\wedge (DependentChildLeafCoreLeaf^{\,k}_{v\{:\,m\}*})' =$
$DependentChildLeafCoreLeaf^{\,k}_{v\{:\,m\}*} \cup \{(u^{j}_{v\{:\,m\}*}, u^{k}_{i\{:\,m\}*})\}$
$\wedge$ UNION_DEPENDENT_DEPENDENT $(u^{j}_{v\{:\,m\}*}, u^{k}_{v\{:\,m\}*})$
$\wedge$ SET_DEPENDENT_STATUS $((u^{j}_{x\{:\,m\}*}, u^{k}_{v\{:\,m\}*}), "active"))$.
Post-condition: $(u^{j}_{v}, u^{k}_{v\{:\,m\}*}) \in DependentChildLeafCoreLeaf^{\,k}_{v\{:\,m\}*}$

Figure 11 Dependency Added to Core Leaf

Because use case versions may not be deleted in order to enforce traceability, the inactive or active status is used to denote that it is currently or was previously a viable version within a hierarchy. Each use case in a core hierarchy $Core^{k}$ and $Dependent^{\,k}$ has a status and a status history is maintained. The status from active to inactive or vice versa may only be changed for leaf use cases both in the core ($u^{k}_{i+1\{:\,m\}*} \in CoreLeaf^{\,k}$) or dependencies to the core ($(u^{j}_{v\{:\,m\}*}, u^{k}_{v\{:\,m\}*}) \in DependentChildLeafCoreLeaf^{\,k}_{v\{:\,m\}*}$). A newly added use case $u^{k}_{i+1\{:\,m\}*}$ in the core is set to the active default status. The actions of the SET_CORE_ACTIVE and SET_CORE_INACTIVE commands are provided in Figure 12a and Figure 12b respectively.

(a)
Pre-Condition: $(u^k_{v\{:m\}*}$, *"active")* $\notin$ *CoreStatus*^{k}
Command = SET_CORE_ACTIVE $(u^k_{v\{:m\}*})$ $\Rightarrow$
$(u^k_{v\{:m\}*} \in$ *CoreLeafInactive*^{k} $\Rightarrow$
((*CoreStatus*^{k})' = *CoreStatus*^{k} - {$(u^k_{v\{:m\}*}$, *"inactive")*
$\wedge$ (*CoreStatus*^{k})' = *CoreStatus*^{k} $\cup$ {$(u^k_{v\{:m\}*}$, *"active")*
$\wedge$ *SET_HISTORY* (($u^k_{v\{:m\}*}$) , *"active", TIME()*))).
Post-condition: $(u^k_{v\{:m\}*}$, *"inactive"*) $\notin$ *CoreStatus*^{k}

(b)
Pre-Condition: $(u^k_{v\{:m\}*}$, *"inactive")* $\notin$ *CoreStatus*^{k}
Command = SET_CORE_INACTIVE $(u^k_{v\{:m\}*})$ $\Rightarrow$
$(u^k_{v\{:m\}*} \in$ *CoreLeafActive*^{k} ,
(($u^q_{v\{:m\}*}$, $u^j_{v\{:m\}*}$), $u^k_{v\{:m\}*}$) $\in$
DependentToDependentLeafChildLeafCoreLeaf$^k_{v\{:m\}*}$ $\vee$
DependentToDependentLeafChildLeafCoreLeaf$^k_{v\{:m\}*}$ = {} $\Rightarrow$
((*CoreStatus*^{k})' = *CoreStatus*^{k} - {$(u^k_{v\{:m\}*}$, *"active")*
$\wedge$ (*CoreStatus*^{k})' = *CoreStatus*^{k} $\cup$ {$(u^k_{v\{:m\}*}$, *"active")*
$\wedge$ SET_HISTORY (($u^k_{v\{:m\}*}$) , *"inactive", TIME()*)
$\wedge$ SET_DEPENDENT_STATUS (($u^j_{v\{:m\}*}$, $u^k_{v\{:m\}*}$) , "inactive"))).
Post-condition: $u^k_{v\{:m\}*} \notin$ *CoreLeafActive*^{k}

Figure 12 Commands: SET_CORE_ACTIVE (a) and SET_CORE_INACTIVE(b)

Similarly for a use case immediately dependent on a core use, $\{(u^j_{v\{:m\}*}, u^k_{v\{:m\}*})\} \in$ *DependentChildLeafCoreLeaf*$^k_{v\{:m\}*}$) $\wedge$ $j \neq$ k, the status may be changed by applying the command, SET_DEPENDENT_STATUS, as shown in Figure 13. The dependents to a dependent (($u^q_{v\{:m\}*}$, $u^j_{v\{:m\}*}$), $u^k_{v\{:m\}*}$) $\in$ *DependentToDependentChildLeafCoreLeaf*$^k_{v\{:m\}*}$) $\wedge$ $j \neq$ k) are initially set to the default status of their associated status in their respective core hierarchies. For a use case version serving in a dependency role, additionally, there will be a current status in the hierarchy in which it appears; use cases in dependent relationships retain a status history within the context of the dependency. When the status of a core use case is made inactive (command SET_CORE_INACTIVE) or an immediate dependency is made inactive

(SET_DEPENDENT_STATUS), all leaves in its legacies are automatically updated to status inactive.

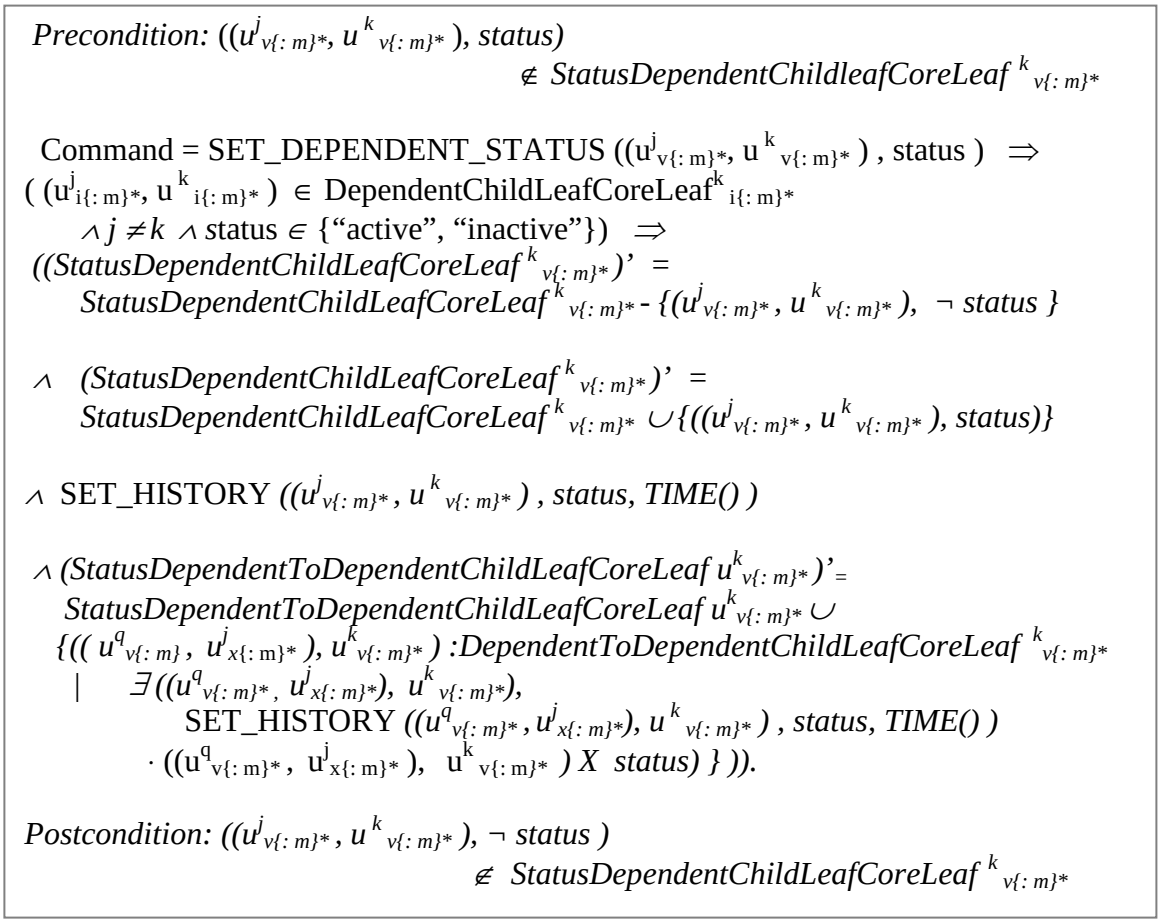


Figure 13 Command SET_DEPENDENT_STATUS

In the case of a refinement split, multiple children are created from a single parent and an enumerated child label extension string, denoted "*: m*" above, is appended to the identification level subscripts for each sibling. Child labels are omitted if there is only one child. To create a level identification label for a refinement to a use case in *CoreLeaf*^{k}, the level number is increased by 1 as described above and the child suffix extension string, if one exists, beginning with ':' of the parent is concatenated to the child's level. If there are multiple children at the new

level, suffixes of ':1', ':2', … to ":m" are finally composed and concatenated. In Figure 14 the command to refine a core leaf use case by multiple children is shown. The command to refine a dependency by multiple siblings, REFINE_DEPENDENT_MULTIPLE_RELATION, is shown in Figure 15. All siblings are set to the default status of active.

Pre-Condition: $u^k_{i+1\{:m\}^*:p+0}, u^k_{i+1\{:m\}^*:p+1}, \ldots, \wedge u^k_{i+1\{:m\}^*:p+(n-1)} \notin CoreLeaf^k$

Command = REFINE_CORE_MULTIPLE $\Rightarrow$
$(u^k_{i\{:m\}^*}, u^k_{i+1\{:m\}^*:p+0}, u^k_{i+1\{:m\}^*:p+1}, \ldots, u^k_{i+1\{:m\}^*:p+(n-1)}) \Rightarrow$
(REFINE_CORE_SINGLE $(u^k_{i+1\{:m\}^*:p+0})$
REFINE_CORE_SINGLE $(u^k_{i+1\{:m\}^*:p+1})$
:
REFINE_CORE_SINGLE $(u^k_{i+1\{:m\}^*:p+(n-1)})$ *)).*

Post-condition: $u^k_{i\{:m\}^*} \notin CoreLeaf^k$

Figure 14 Command to Refine a Core Leaf Use Case by Multiple Children

Pre-condition: $(u^j_{x\{:m\}^*:p+0,}, u^k_{v\{:m\}^*}), (u^j_{x\{:m\}^*:p+1,}, u^k_{v\{:m\}^*}), \ldots, \wedge$
$(u^j_{x\{:m\}^*:p+(n-1)}, u^k_{v\{:m\}^*}) \notin DependentChildLeafCoreLeaf^k_{v\{:m\}^*}$

Command = REFINE_DEPENDENT_MULTIPLE_RELATION
$(u^j_{i\{:m\}^*}, u^j_{x\{:m\}^*:p+0}, u^j_{x\{:m\}^*:p+1}, \ldots, u^j_{x\{:m\}^*:p+(n-1)}, u^k_{v\{:m\}^*}, relation) \Rightarrow$
(relation $\in$ *{"include", "extend"}* $\wedge$ $u^j_{i\{:m\}^*} \in Dependent^k_{v\{:m\}^*} \Rightarrow$
REFINE_DEPENDENT_SINGLE_RELATION
$(u^j_{i\{:m\}^*}, u^j_{x\{:m\}^*:p+0}, u^k_{v\{:m\}^*}, relation)$
REFINE_DEPENDENT_SINGLE_RELATION
$(u^j_{i\{:m\}^*}, u^j_{x\{:m\}^*:p+1}, u^k_{v\{:m\}^*}, relation)$
:
REFINE_DEPENDENT_SINGLE_RELATION
$(u^j_{i\{:m\}^*}, u^j_{x\{:m\}^*:p+(n-1)}, u^k_{v\{:m\}^*}, relation)$ *)).*

Post-condition: $(u^j_{i\{:m\}^*}, u^k_{v\{:m\}^*}) \notin DependentChildLeafCoreLeaf^k_{v\{:m\}^*}$

Figure 15 Command to Refine a Dependency by Multiple Children

Use of the RE/TRAC-SEM commands restricts changes to a sentence in RE/TRAC-CF. By stipulating that all changes must be made to leaves in which there are no further refinements, traceability of the requirements evolution is preserved. An organization of use cases, *U*, is initially described and then at discrete advances in time, variability is introduced via refinements and associations. The dynamic semantics constrains those changes so that consistency is maintained in the evolutionary change process. An application example follows in section 4.3.

Precondition: $(u^{q}_{v\{:m\}^*}, u^{j}_{x\{:m\}^*}), \; u^{k}_{v\{:m\}^*}) \notin DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$

Command= UNION_DEPENDENT_DEPENDENT $(u^{j}_{x\{:m\}^*}, u^{k}_{v\{:m\}^*}) \Rightarrow$
$(j \neq k \Rightarrow$
$((DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*})' =$
$DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*} \cup$
$\{(u^{q}_{v\{:m\}^*}, u^{j}_{x\{:m\}^*}) : DependentChildLeafCoreLeaf^{j}_{x\{:m\}^*},$
$u^{k}_{v\{:m\}^*} : CoreLeaf^{k}_{v\{:m\}^*}$
$| \; \exists((u^{q}_{v\{:m\}^*}, u^{j}_{v\{:m\}^*}), u^{k}_{v\{:m\}^*}), u^{q}_{v\{:m\}^*}$ Child $u^{j}_{v\{:m\}^*}$
$\wedge \exists(u^{q}_{v\{:m\}^*},$
UNION_DEPENDENT_DEPENDENT $(u^{q}_{v\{:m\}^*}, u^{k}_{v\{:m\}^*})$
$\cdot ((u^{q}_{v\{:m\}^*}, u^{j}_{x\{:m\}^*}), u^{k}_{v\{:m\}^*})\} \;))$.

Postcondition: $(u^{q}_{v\{:m\}^*}, u^{j}_{x\{:m\}^*}), \; u^{k}_{v\{:m\}^*}) \in DependentToDependentChildLeafCoreLeaf^{k}_{v\{:m\}^*}$

Figure 16 Command: UNION_DEPENDENT_DEPENDENT Use Cases

4.3 Example

As a rudimentary application of RE/TRAC-SEM, consider a software development company that maintains an online conference registration system for use by various professional organizations. The following are base or core cases *U* in the example domain, D:

b^1 – Inquire available conferences
b^2 – Register for a specific conference and tally of registration fees
b^3 – View specific attendee's registration information
b^4 – Change attendee's registration
b^5 – Delete attendee's registration
b^6 – Verify attendee
b^7 – Debit attendee's account
b^8 – Credit attendee's account
b^9 – Join organization
b^{10} – Lookup discount for organization members

Recall that b^k represents the k^{th} base case, $u^k{}_1$ represents a more detailed description of the base case, and subsequent versions of u^k indicate the evolution of the base case over time. A sentence S^k depicts all use cases related to the k^{th} base case. Given the domain D, an application instance numbered 3, represented as D^3, is represented by the sentences, $\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8,\}$. After an initial selection of sentences 1-8 and a first refinement incorporating the inclusion of several dependent use cases, the following base cases at time 1, denoted $D^3{}_1$, are described:

$$S^1 = b^1<(u^1{}_1)>$$
$$S^2 = b^2<(u^2{}_1[(u^7{}_1)])>$$
$$S^3 = b^3<(u^3{}_1[(u^6{}_1)])>$$
$$S^4 = b^4<(u^4{}_1[(u^6{}_1)(u^8{}_1)(u^7{}_1)])>$$
$$S^5 = b^5<(u^5{}_1[(u^6{}_1)(u^8{}_1)])>$$
$$S^6 = b^6<(u^6{}_1)>$$
$$S^7 = b^7<(u^7{}_1)>$$
$$S^8 = b^8<(u^8{}_1)>$$
$$S^9 = b^9<(u^9{}_1)>$$

Use Case 2 describes the registration of a conference attendee and includes Use Case 7 which records the participant's fees as paid in full. This example assumes that payment for the conference is due upon registration. Use Case 3 to view information about an attendee must first verify that the customer has previously registered as an attendee. The added functionality is provided by the inclusion of Use Case 6 in Use Case 3. If a customer requests a change to his registration as portrayed in Use Case 4, the registration is first verified (Use Case 6), then the old

fees are credited to his account (Use Case 8), the new fee calculated and his account debited (Use Case 7) with the corrected fee tally. If a customer elects to withdraw from a conference (Use Case 5), the attendee's registration is verified (Use Case 6) and the relevant account is credited (Use Case 8). Inclusion of embedded use cases in RE/TRAC-CF sentences 2, 3, 4 and 5 are denoted by square bracket pairs ([ ]) as noted in table 2. The relevant RE/TRAC_SEM sets at D^3_1 are shown in Figure 17.

The client decides that if a registrant must cancel a registration within one week of the conference, there will be a refund of only 50%. The organization is striving to increase memberships, so even if an attendee withdraws from a conference, he may be interested in joining the organization. So before an account is credited (Use Case 8), the client would like a prompt for the user to become an organization member (Use Case 9). So now at time D^3_2, u^8_1 is extended by u^9_1 and there is a change to u^5_1 to include a test for one week before invoking u^8_1. The commands,

(3) CORE_ADD_RELATION (u^9_1 , u^8_1, "extend")

(9) UNION_DEPENDENT_DEPENDENT (u^9_1 , u^8_1) - no action taken

(6) SET_DEPENDENT_STATUS (u^9_1 , u^8_1 , "Active")

(1) REFINE_CORE_SINGLE (u^5_1, u^5_2)

(4) SET_CORE_ACTIVE (u^5_2)

(3) CORE_ADD_RELATION ((u^6_1, u^5_2), "include")

(9) UNION_DEPENDENT_DEPENDENT (u^6_1, u^5_2) - no action taken

(6) SET_DEPENDENT_STATUS (u^6_1, u^5_2 , "active")

(3) CORE_ADD_RELATION ((u^8_1 , u^5_2), "include")

(9) UNION_DEPENDENT_DEPENDENT (u^8_1 , u^5_2) - u^9_1 added,

(9) UNION_DEPENDENT_DEPENDENT (u^9_1 , u^8_1) – no action taken

(6) SET_DEPENDENT_STATUS (u^8_1 , u^5_2 , “active”).

are employed generating the following evolutionary changes to S^8 and S^5:

$S^8 = b^8<(u^8_1\{(u^9_1)\})>$

$S^5 = b^5<(u^5_1\,[(u^6_1)(u^8_1)]<(u^5_2\,[(u^6_1)(u^8_1\{(u^9_1)\})])>)>$

Pertinent set changes are shown in Figure 18.

The client chooses to provide a discount to the organization members attending a conference at time D^3_3, so u^2_1 is refined to include the test for organization membership and then subsequently to invoke the lookup for the discount. Commands

(1) REFINE_CORE_SINGLE (b^{10} , u^{10}_1)

(4) SET_CORE_ACTIVE (u^{10}_1)

(1) REFINE_CORE_SINGLE (u^2_1 , u^2_2)

(4) SET_CORE_ACTIVE (u^2_2)

(3) CORE_ADD_RELATION ((u^7_1 , u^2_2) , “include”),

(9) UNION_DEPENDENT_DEPENDENT (u^7_1 , u^2_2) - no action taken

(6) SET_DEPENDENT_STATUS ((u^7_1 , u^2_2), ”active”)

(3) CORE_ADD_RELATION ((u^{10}_1, u^2_2), “extend”)

(9) UNION_DEPENDENT_DEPENDENT (u^{10}_1 , u^2_2) - no action taken

(6) SET_DEPENDENT_STATUS ((u^{10}_1 , u^2_2), ”active”).

are applied causing the sentences S^{10} and S^2 to be altered:

$S^{10} = b^{10}<(u^{10}_1)>$

$S^2 = b^2<(u^2_1[(u^7_1)]<(u^2_2\,[(u^7_1)]\,\{(u^{10}_1)\})>)>$

A partial listing of the sets is provided in Figure 19.

$U = \{ b^1, u^1_1, b^2, u^2_1, b^3, u^3_1, b^4, u^4_1, b^5, u^5_1, b^6, u^6_1, b^7, u^7_1, b^8, u^8_1, b^9, u^9_1, b^{10}\}$

$CoreLeaf^1 = \{u^1_1,\}$ $\quad CoreLeafActive^1 = \{u^1_1,\}$
$CoreLeaf^2 = \{u^2_1\}$ $\quad CoreLeafActive^2 = \{u^2_1\}$
$CoreLeaf^3 = \{u^3_1\}$ $\quad CoreLeafActive^3 = \{u^3_1\}$
$CoreLeaf^4 = \{u^4_1\}$ $\quad CoreLeafActive^4 = \{u^4_1\}$
$CoreLeaf^5 = \{ u^5_1\}$ $\quad CoreLeafActive^5 = \{ u^5_1\}$
$CoreLeaf^6 = \{ u^6_1\}$ $\quad CoreLeafActive^6 = \{ u^6_1\}$
$CoreLeaf^7 = \{ u^7_1\}$ $\quad CoreLeafActive^7 = \{ u^7_1\}$
$CoreLeaf^8 = \{ u^8_1\}$ $\quad CoreLeafActive^8 = \{ u^8_1\}$
$CoreLeaf^9 = \{ u^9_1\}$ $\quad CoreLeafActive^9 = \{ u^9_1\}$

$CoreLeafActive = \{ u^1_1 , u^2_1 , u^3_1 , u^4_1 , u^5_1, u^6_1 , u^7_1 , u^8_1 , u^9_1 \}$
$DependentChildCore^1_1 = \{\}$
$DependentChildCore^2_1 = \{ (u^7_1 , u^2_1)\}$
$DependentChildCore^3_1 = \{(u^6_1, u^3_1)\}$
$DependentChildCore^4_1 = \{ (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1)\}$
$DependentChildCore^5_1 = \{ (u^6_1, u^5_1), (u^8_1 , u^5_1)\}$
$DependentRelationChild^1_1 = \{\}$
$DependentRelationChild^2_1 = \{ ((u^7_1 , u^2_1), \text{“include”})\}$
$DependentRelationChild^3_1 = \{((u^6_1, u^3_1), \text{“include”})\}$
$DependentRelationChild^4_1 = \{ ((u^6_1, u^4_1), \text{“include”}), ((u^7_1, u^4_1), \text{“include”}), ((u^8_1, u^4_1), \text{“include”}) \}$
$DependentRelationChild^5_1 = \{ ((u^6_1, u^5_1), \text{“include”}), ((u^8_1 , u^5_1), \text{“include”})\}$
$DependentIncludeChildCore^2_1 = \{ (u^7_1 , u^2_1)\}$
$DependentIncludeChildCore^3_1 = \{(u^6_1, u^3_1)\}$
$DependentIncludeChildCore^4_1 = \{ (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1)\}$
$DependentIncludeChildCore^5_1 = \{ (u^6_1, u^5_1), (u^8_1 , u^5_1)\}$
$DependentChildLeafIncludeChildCore^2_1 = \{ (u^7_1 , u^2_1)\}$
$DependentChildLeafIncludeChildCore^3_1 = \{(u^6_1, u^3_1)\}$
$DependentChildLeafIncludeChildCore^4_1 = \{ (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1)\}$
$DependentChildLeafIncludeChildCore^5_1 = \{ (u^6_1, u^5_1), (u^8_1 , u^5_1)\}$
$ActiveDependentChildLeafIncludeChildCore^2_1 = \{ (u^7_1 , u^2_1)\}$
$ActiveDependentChildLeafIncludeChildCore^3_1 = \{(u^6_1, u^3_1)\}$
$ActiveDependentChildLeafIncludeChildCore^4_1 = \{ (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1)\}$
$ActiveDependentChildLeafIncludeChildCore^5_1 = \{ (u^6_1, u^5_1), (u^8_1 , u^5_1)\}$
$TotalActiveDependenciesLeafCoreLeaf = \{ (u^7_1 , u^2_1), (u^6_1, u^3_1), (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1), (u^6_1, u^5_1), (u^8_1 , u^5_1)\}\}$
$CoreLeafActive \cup dom\ TotalActiveDependenciesLeafCoreLeaf = \{ u^1_1, u^2_1, u^3_1, u^4_1, u^5_1, u^6_1, u^7_1, u^8_1 \}$

Figure 17 Sets at D^3_1

$U = \{ b^1, u^1_1, b^2, u^2_1, b^3, u^3_1, b^4, u^4_1, b^5, u^5_1, u^5_2, b^6, u^6_1, b^7, u^7_1, b^7, u^8_1, b^9, u^9_1, b^{10}\}$

$DependentChildCore^8_1 = \{ (u^9_1, u^8_1)\}$
$DependentRelationChild^8_1 = \{ (u^9_1, u^8_1), \text{“extend”} \}$
$DependentChildLeafCoreLeaf^8_1 = \{ (u^9_1, u^8_1)\}$
$StatusDependentChildLeafCoreLeaf^8_1 = \{ (u^9_1, u^8_1), \text{“active”}\}$

$DependentToDependentChildLeafCoreLeaf\ u^5_2 = \{((u^9_1 , u^8_1), u^5_2)\}$
$Core^5 = \{ b^5 , u^5_1 , u^5_2\}$
$CoreLeaf^5 = \{ u^5_2\}$
$CoreStatus^5_2 = \{ u^5_2, \text{“active”}\}$
$CoreLeafActive^5 = \{ u^5_2\}$
$DependentChildCore^5_2 = \{ (u^6_1, u^5_2), (u^8_1 , u^5_2)\}$
$DependentRelationChild^5_2 = \{ ((u^6_1, u^5_2), \text{“include”}), ((u^8_1 , u^5_2), \text{“include”})\}$
$DependentIncludeChildCore^5_2 = \{ (u^6_1, u^5_2), (u^8_1 , u^5_2)\}$
$DependentChildLeafIncludeChildCore^5_2 = \{ (u^6_1, u^5_2), (u^8_1 , u^5_2)\}$

$DependentChildLeafCoreLeaf^5_2 = \{ (u^6_1, u^5_2), (u^8_1 , u^5_2)\}$
$StatusDependentChildLeafCoreLeaf\ u^5_2 = \{((u^6_1 , u^5_2), \text{“active”}), ((u^8_1 , u^5_2), \text{“active”})$
$ActiveDependentChildLeafChildCore^5_2 = \{(u^6_1, u^5_2), (u^8_1 , u^5_2)\}$
$StatusDependentToDependentChildLeafCoreLeaf^5 = \{((u^9_1 , u^8_1), u^5_2), \text{“active”}\}$
$ActiveDependentToDependentChildLeafCoreLeaf^5_2 = \{((u^9_1 , u^8_1), u^5_2)\}$
$ActiveDependentChildLeafCoreLeaf^5 = \{ (u^9_1 , u^5_2)\}$
$CoreLeafActive = \{ u^1_1 , u^2_1 , u^3_1 , u^4_1 , u^5_2\}$
$CoreLeafInactive = \{ b^1 , b^2 , b^3 , b^4 , b^5 , u^5_1 , b^6, b^7 , b^8_1 , b^9, b^{10} \}$
$TotalActiveDependenciesLeafCoreLeaf^5 = \{ (u^9_1 , u^5_2) , (u^6_1, u^5_2), (u^8_1 , u^5_2)\}$

$TotalActiveDependenciesLeafCoreLeaf = \{ (u^7_1 ,u^2_1), (u^6_1, u^3_1), (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1), (u^6_1, u^5_2), (u^8_1 , u^5_2) , (u^9_1 , u^5_2) \}$

$CoreLeafActive \cup dom$ TotalActiveDependenciesLeafCoreLeaf $= \{ u^1_1, u^2_1, u^3_1, u^4_1, u^5_2, u^6_1, u^7_1, u^8_1 , u^9_1 \}$

Figure 18 Sets at D^3_2

Similarly at D^3_4, the client decides to prompt a registrant to join the organization if not currently a member, resulting in a refinement of u^2_2 to u^2_3. If the customer is not currently a

member, Use Case 10 is invoked conditionally. The following commands are applied,

(1) REFINE_CORE_SINGLE ($u^2_{2,}$ u^2_3)

(4) SET_CORE_ACTIVE (u^2_3)

(3) CORE_ADD_RELATION ((u^9_1, u^2_3), “extend”)

(9) UNION_DEPENDENT_DEPENDENT (u^9_1, u^2_3) - no action taken

(6) SET_DEPENDENT_STATUS ((u^9_1, u^2_3), ”active”)

(3) CORE_ADD_RELATION ((u^{10}_1, u^2_3), “extend”)

(9) UNION_DEPENDENT_DEPENDENT (u^{10}_1, u^2_3) - no action taken

(6) SET_DEPENDENT_STATUS ((u^{10}_1, u^2_3), ”active”)

(3) CORE_ADD_RELATION ((u^7_1, u^2_3), “include”)

(9) UNION_DEPENDENT_DEPENDENT (u^7_1 , u^2_3) - no action taken

(6) SET_DEPENDENT_STATUS ((u^7_1 , u^2_3), ”active”).

resulting in the following revisions to S^2 :

$$S^2 = b^2<(u^2_1[(u^7_1)]<(u^2_2\ [(u^7_1)]\ \{(u^{10}_1)\}<(u^2_3[(u^7_1)]\ \{(u^9_1)\ (u^{10}_1)\})>)>)>$$

A partial listing of the sets at time D^3_4 is provided in Figure 20.

Given the RE/TRAC-CF representation and the constraints provided by the dynamic semantics, a history of change including the corresponding dependent relationships is preserved during requirements evolution of use cases. The history is explicit in the RE/TRAC visualizations in Figure 21 for the sentence S^2 at time D^3_4 and in Figure 22 for S^5 at time D^3_4 . Other RE/TRAC diagrams for domain D^3 are not shown, as they are trivial. Each level in the RE/TRAC interpretation of the RE/TRAC-CF sentence corresponds to a discrete time illustrating that a change has occurred. Commands that are initiated to evoke the evolution have pre-conditions, post-conditions and rules for limiting the relationships, governing the change in

relationships, registering time history, and status maintenance. By following the commands to oversee the evolutionary process, we have correctly maintained sentences in RE/TRAC-CF and generated the corresponding RE/TRAC with correct meaning. This demonstrates that the dynamic semantics specified by RE/TRAC-SEM imposes consistency between the RE/TRAC sentences and its corresponding diagrammatic interpretation.

$U = \{ b^1, u^1_1, b^2, u^2_1, u^2_2, b^3, u^3_1, b^4, u^4_1, b^5, u^5_1, u^5_2, b^6, u^6_1, b^7, u^7_1, b^7, u^8_1, b^9, u^9_1, b^{10}, u^{10}_1 \}$

$Core^{10} = \{b^{10}_0, u^{10}_1\}$
$CoreLeaf^{10} = \{u^{10}_1\}$
$CoreStatus^{10} = \{(u^{10}_1, \text{“active”})\}$

$Core^2 = \{b^2_0, u^2_1, u^2_2\}$
$CoreLeaf^2 = \{u^2_2\}$
$CoreStatus = \{u^2_2, \text{“active”}\}$
$CoreLeafActive^2 = \{u^2_2\}$

$DependentChildCore^2_2 = \{ (u^7_1, u^2_1), (u^7_1, u^2_2), (u^{10}_1, u^2_2), \}$
$DependentRelationChild^2_2 = \{ ((u^7_1, u^2_2), \text{“include”}), ((u^{10}_1, u^2_2), \text{“extend”}) \}$
$DependentIncludeChildCore^2_2 = \{(u^7_1, u^2_2)\}$
$DependentChildLeafCoreLeaf^2 = \{ (u^7_1, u^2_2), (u^{10}_1, u^2_2) \}$
$StatusDependentChildLeafCoreLeaf^2_2 = \{((u^7_1, u^2_2), \text{“active”}), ((u^{10}_1, u^2_2), \text{“active”}\}$
$DependentChildLeafIncludeChildCore^2_2 = \{(u^7_1, u^2_2)\}$

$DependentExtendChildCore^2_2 = \{(u^{10}_1, u^2_2)\}$
$DependentChildLeafExtendChildCore^2_2 = \{(u^{10}_1, u^2_2)\}$
$ActiveDependentChildLeafIncludeChildCore^2_2 = \{(u^7_1, u^2_2)\}$
$ActiveDependentChildLeafExtendChildCore^2_2 = \{ (u^{10}_1, u^2_2)\}$
$TotalActiveDependenciesLeafCoreLeaf = \{ (u^7_1, u^2_2), (u^{10}_1, u^2_2), (u^6_1, u^3_1), (u^6_1, u^4_1), (u^7_1, u^4_1), (u^8_1, u^4_1), (u^6_1, u^5_2), (u^8_1, u^5_2)\}$

$CoreLeafActive \cup dom\ TotalActiveDependenciesLeafCoreLeaf = \{ u^1_1, u^2_2, u^3_1, u^4_1, u^5_2, u^6_1, u^7_1, u^8_1, u^9_1, u^{10}_1\}$

Figure 19 Sets at D^3_3

$U = \{ b^1, u^1_1, b^2, u^2_1, u^2_2, u^2_3, b^3, u^3_1, b^4, u^4_1, b^5, u^5_1, u^5_2, b^6, u^6_1, b^7, u^7_1, b^7, u^8_1, b^9, u^9_1, b^{10}, u^{10}_1\}$

$Core^2 = \{b^2_0, u^2_1, u^2_2, u^2_3\}$
$CoreLeaf^2 = \{u^2_3\}$
$CoreStatus^2 = \{(u^2_3, \text{“active”})\}$
$CoreLeafActive^2 = \{u^2_3\}$

$DependentChildCore^2 = \{ (u^7_1, u^2_1), (u^7_1, u^2_2), (u^{10}_1, u^2_2), (u^7_1, u^2_3), (u^9_1, u^2_3), (u^{10}_1, u^2_3)\}$
$DependentRelationChild^2_3 = \{ ((u^7_1, u^2_3), \text{“include”}), ((u^9_1, u^2_3), \text{“extend”}), ((u^{10}_1, u^2_3), \text{“extend”}) \}$

$DependentIncludeChildCore^2_3 = \{(u^7_1, u^2_3)\}$
$DependentChildLeafChildCore^2_3 = \{(u^7_1, u^2_3), (u^9_1, u^2_3), (u^{10}_1, u^2_3)\}$
$DependentChildLeafIncludeChildCore^2_3 = \{(u^7_1, u^2_3)\}$
$ActiveDependentChildLeafIncludeChildCore^2_3 = \{(u^7_1, u^2_3)\}$
$DependentExtendChildCore^2_3 = \{(u^9_1, u^2_3), (u^{10}_1, u^2_3)\}$
$DependentChildLeafExtendChildCore^2_3 = \{(u^9_1, u^2_3), (u^{10}_1, u^2_3)\}$
$ActiveDependentChildLeafExtendChildCore^2_3 = \{(u^7_1, u^2_3), (u^{10}_1, u^2_3)\}$

$TotalActiveDependenciesLeafCoreLeaf = \{ (u^7_1, u^2_3), (u^9_1, u^2_3), (u^{10}_1, u^2_3)\}$

$CoreLeafActive \cup dom\ TotalActiveDependenciesLeafCoreLeaf = \{ u^1_1, u^2_3, u^3_1, u^4_1, u^5_1, u^6_1, u^7_1, u^8_1, u^9_1, u^{10}_1 \}$

Figure 20 Set Sets at D^3_4

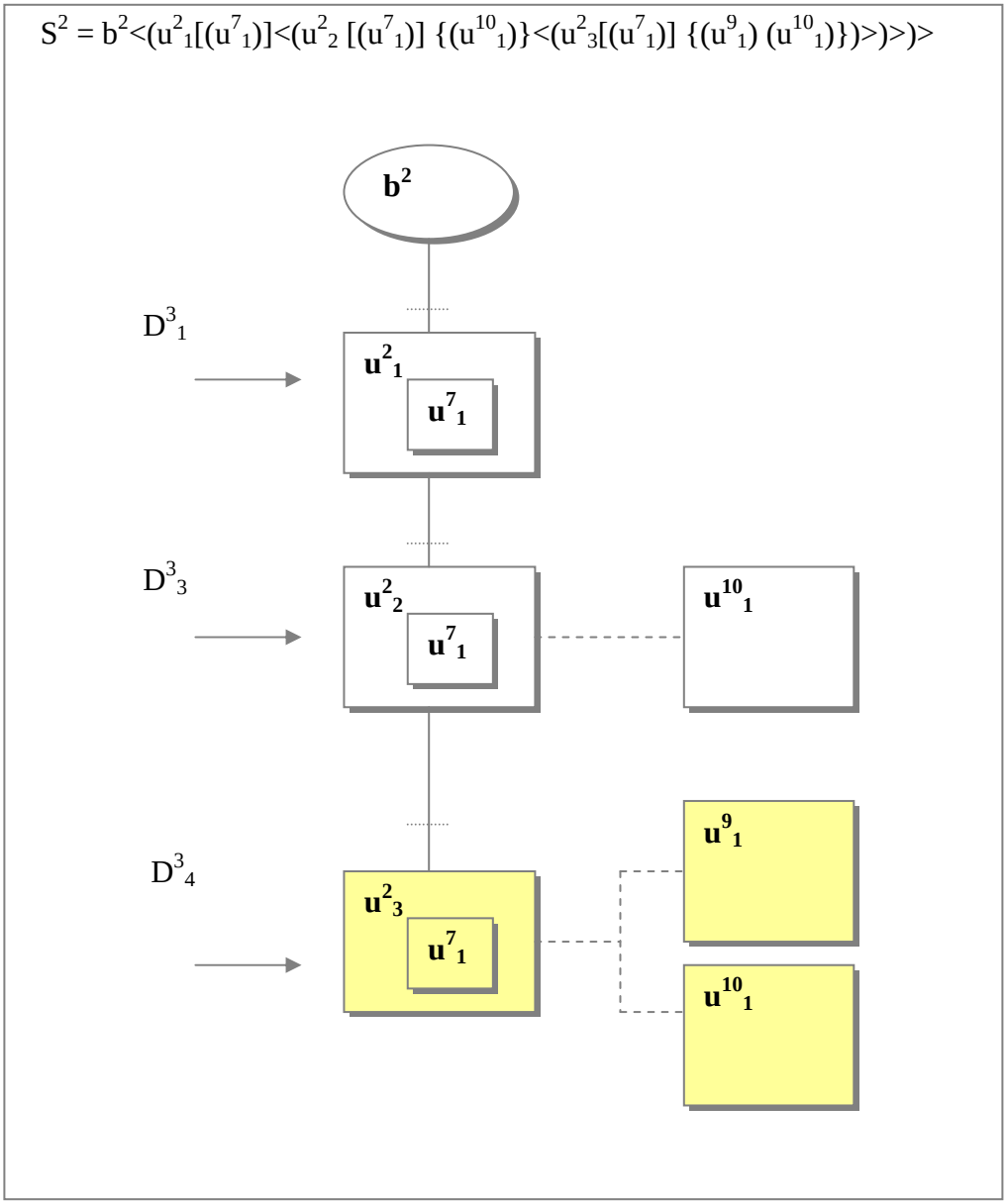


Figure 21 $S^2 = b^2<(u^2_1[(u^7_1)]<(u^2_2 [(u^7_1)] \{(u^{10}_1)\}<(u^2_3[(u^7_1)] \{(u^9_1)\ (u^{10}_1)\})>)>)>$ at D^3_4

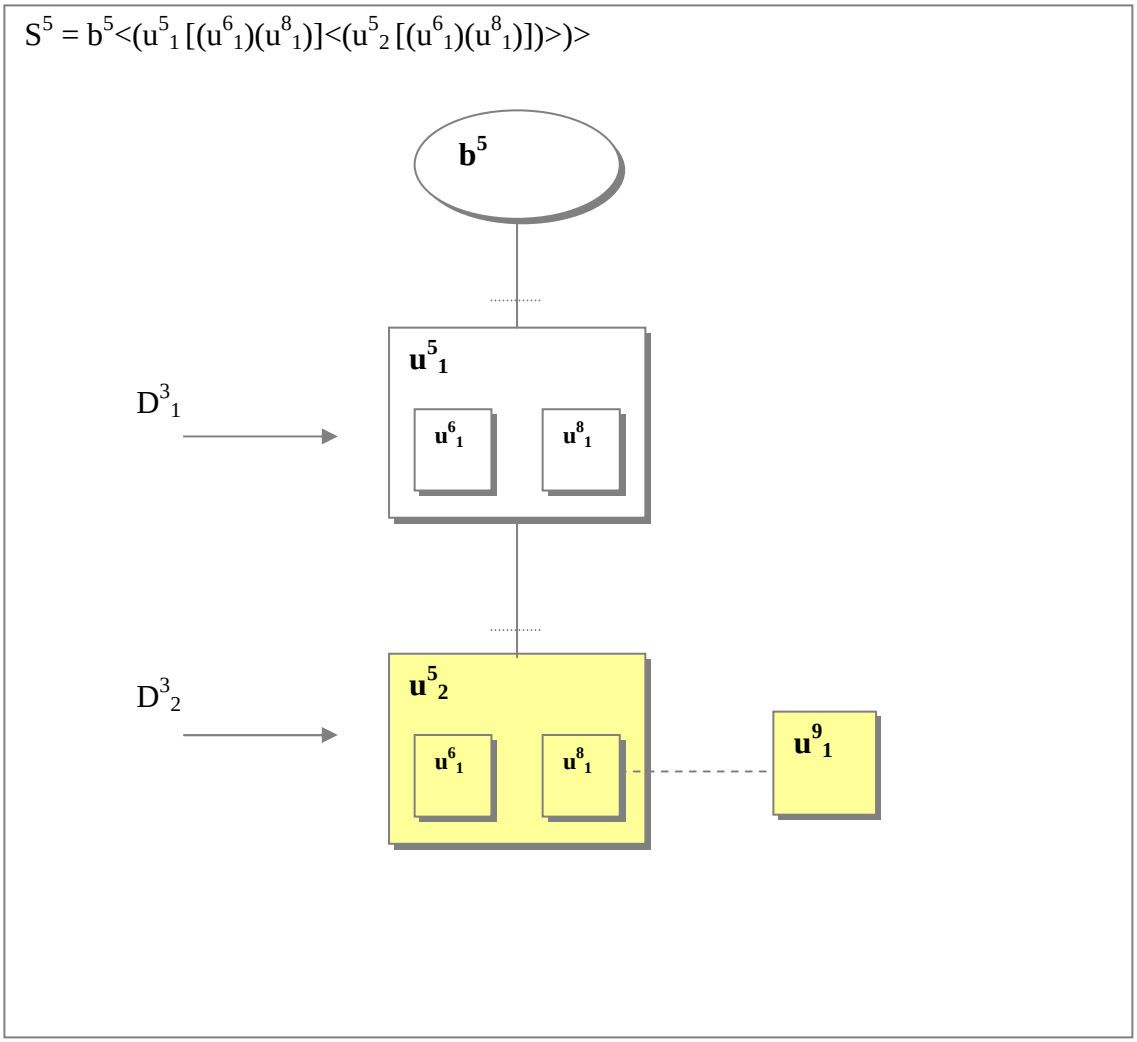


Figure 22 $S^5 = b^5<(u^5_1[(u^6_1)(u^8_1)]<(u^5_2[(u^6_1)(u^8_1\{(u^9_1)\})>)>$ at D^3_2

5. Method Validation

In [Ped et al] the validation of research within the field of engineering design is said to be problematic because is relies on subjective statements as well as mathematical modeling and therefore does not lend itself to traditional scientific validation based on logical induction and/or deduction. [Ped et al] assert " that research validation is a process of building confidence in its usefulness with respect to a purpose" and is founded on whether the design solution is shown to be effective and efficient by employing qualitative and quantitative means respectively. Their research validation process is called the "Validation Square" where the process is divided into two parts: structural validation (a qualitative process steps 1,2, and 3) and performance validation (a quantitative process steps 4, 5 and 6). The validation square is shown in Figure 23 in which detailed steps are numbered (1) through (6).

(1)& (2) Theoretical **Structural** Validity	(6) Theoretical **Performance** Validity
(3) Empirical **Structura**l Validity	(4) & (5) Empirical **Performanc**e Validity

Figure 23 The "Validation Square" [Ped et al]

The validation for RE/TRAC follows the Validation Square process for method validation. In order to show that the method has theoretical or general structural validity, confidence in the validity of the individual constructs (1) of RE/TRAC is demonstrated. Secondly (2), we demonstrate method consistency in the manner in which the RE/TRAC's constructs are integrated. The empirical structural validity (3) is established by explaining the appropriateness

of the example problem in supporting our hypothesis. Next, (4) we describe how RE/TRAC improves the requirements evolution process by addressing the quality of the method. Subsequently we show that the usefulness is linked (5) to using the RE/TRAC method. By showing that the RE/TRAC method is useful beyond some limited purposes (6), then in a more general sense we have confirmed the theoretical performance valid. The sections below are numbered (5.1 – 5.6) according to the numbered components in the Validation Square.

5.1 Accepting the Constructs' Validity

Hierarchical step-wise refinement is similar to a top-down design method. Top-down design is a proven incremental approach where the daunting task of solving a large problem is addressed by first breaking it into smaller and smaller components, each with more detail than its predecessor [KenKen], [PetPed], [Sommerville]. A top-down design approach emphasizes interrelationships and interdependencies of subsystems [KenKen]. Just as the top-down approach facilitates defining of subsystems and modular programming [KenKen], the hierarchical step-wise refinement of requirements' documents lends itself to a component-based organization of the artifacts. Both are divide-and-conquer approaches well proven in military strategies and algorithm development.

Visual languages have been found in varying human cultures and from earliest pre-history until modern times. They may be of a highly abstract form or a very detailed well-defined notation such as a musical notation or an engineer's design blueprints. Visual languages contribute another form of communication besides written or spoken languages. In addition, visual languages are not necessarily constrained by a sequential processing of the symbols as they are read or spoken [Crapo et al], [MarMey]. As demonstrated by a number of researchers, appropriate visualization models can improve the cognitive reasoning and can increase problem-

solving performance. A system is modeled using the cognitive processes of the stakeholder, and external representations, such as visual models, are the primary method of extending working memory by harnessing the parallel processing capabilities of the working brain [Crapo et al].

RE/TRAC provides an alternative highly abstract form of communication of the written requirements artifacts and their interrelationships. It permits another view of the history of the evolution of the artifacts other than written textual lists possibly in an indexed hierarchical display format. There is a similarity to written textual artifacts in that the top-down left to right ordering of the primitives in a RE/TRAC diagram follows the natural reading and comprehension of the English language and many other languages. The nesting of symbols to depict inclusion has its roots in the formalism of Euler circles [Euler] and later Venn diagrams [Venn] which have been beneficial in solving various mathematical problems. The number of symbols defined in RE/TRAC is limited to the purpose so that unnecessary or less important details in the model are not distracting to the viewer. Limiting the constructs toward an explicit representational model has been claimed as an accepted axiom in developing models toward making models more effective [Alabastro], [Crapo et al], [Johnson], [StenGur]. While some users are not comfortable with graphical depictions, others find them simple and often self-explanatory. In this research we take the position that a graphical depiction of interrelationships is quicker to comprehend than detailed written documentation especially if the non-technical stakeholder has little or no prior understanding of component-based design.

Grammatical approaches to visual languages use techniques very close to the specification of string languages, which have a long history. RE/TRAC-SEM employs a generalized context-free grammar, RE/TRAC-CF, to formalize the rules for constructing corresponding correct RE/TRAC visual models. Production rules describe acceptable two-dimensional generalizations of

concatenation, and terminals specify two-dimensional primitive objects. RE/TRAC-CF does not describe the connectors or spatial positioning of the visual primitives. In this way it shares characteristics of positional grammars [Mar et al] where the sequence of the primitives in a RE/TRAC-CF language sentence indicates where the next primitive is relative to the current primitive and what kind relationship it is. According to Chomsky [Chom1957], [Chom1965], a sequential language is described using a (phrase-structure) grammar composed of production rules. While a visual sentence in RE/TRAC may not necessarily be viewed sequentially (top down), the abstract representation is stored sequentially and its RE/TRAC visual representation is constructed sequentially (top down or bottom up).

Based on previous success in the application example section 4.3 and use of the fundamental building blocks of RE/TRAC, the method is shown to have a firm foundation in principles and purpose which are accepted and valued. Hierarchical models have been shown to be useful in many areas of computer science. The use of visual languages has a long history of success in communication. The use of a restrictive VL with limited primitives and relationships enables an abstract diagram that one can readily grasp meaning and association with the domain application. Grammatical approaches are utilized in the area of visual languages with success. RE/TRAC relies on the BNF grammar for generating a diagram, but uses formal set theory to primarily describe the semantics. Both BNF grammars and set theory have been shown to be adequate for modeling domain applications [Mar et al].

5.2 Accepting Method Consistency

In order to build confidence in the RE/TRAC method's internal consistency, several techniques were employed. The programs Lex and YACC were employed to verify that the RE/TRAC-CF language was consistent with its purpose. RE/TRAC-CF constrains the path of

refinement such that it is always linear (top-down) and associates dependencies at each level in the refinement process. RE/TRAC-CF preserves the history of the evolution so that traceability is enabled. Lex is a program that creates a lexical analyzer which is a function that takes a stream of characters as input, and whenever a group of characters (token) (see Figure 24) matches a

```
[b][0-9]+  yylval=(char *)strdup(yytext); return TOKBASECASE;
[uU][0-9]+[\\][0-9]+(:[0-9]+)*
           yylval=(char *)strdup(yytext); return TOKUSECASE;
[<]   return TOKLEFTANGLE;
[>]   return TOKRIGHTANGLE;
[(]   return TOKLEFTPARENS;
[)]   return TOKRIGHTPARENS;
[{]   return TOKLEFTCURLY;
[}]   return TOKRIGHTCURLY;
[\[]  return TOKLEFTSQUARE;
[\]]  return TOKRIGHTSQUARE;
[;]   return TOKEND;
[ \t]+ /* ignore whitespaces */
[\n]  /* ignore new lines */
[a]   yyerror("invalid character\n");
[A]   yyerror("invalid character\n");
[b-tv-wB-TV-W0-9] yyerror("invalid character\n");
```

Figure 24 Descriptions of Tokens

defined key an action is taken. In the case of this research, when a token is matched, a return value indicates what kind of token has been detected. The tokens are described for input to Lex in Figure 24 including error conditions. Note that to process strings of RE/TRAC-CF sentences, a sentence delimiter (‘;’) primitive was added to the grammar for test purposes.

YACC (Yet Another Compiler-Compiler) is a tool for dictating structure on the input to a computer program. It is an LALR(1) (Look-Ahead 1, Left-to-right, Right-most) parser generator. The user of the YACC tool first prepares a specification of the input process. The specification contains user-defined rules describing the input structure, code to be involved when a rule is

recognized, and a routine to handle basic input. Then YACC generates a parser function which when supplied with the tokens provided by the lexical analyzer, organizes them according to the production rules. When a rule is matched the corresponding user supplied action is executed. In this research, a print command is supplied to print the rule with the input primitives. In Figure 25,

```
Z:
   Z Z
   {
      printf("TRACE RULE Z: Z Z | Z Z |\n");
   }
   |
   TOKLEFTPARENS T TOKRIGHTPARENS
   {
      printf("TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | \(
T ) |\n");
   }
   |
   TOKLEFTPARENS T A TOKRIGHTPARENS
   {
      printf("TRACE RULE Z: TOKLEFTPARENS T A TOKRIGHTPARENS |
\( T A ) |\n");
   }
   ;
```

Figure 25 Grammar rule in YACC specification for nonterminal Z

if the non-terminal Z has been determined, then one of three match sequences must be made: `ZZ, TOKLEFTPARENS T TOKRIGHTPARENS,` or `TOKLEFTPARENS T A TOKRIGHTPARENS.` For the RE/TRAC-CF sentence, b46<(U46\1[(U29\2<(U29\4)>)])>;, the ouptut of YACC is shown in Figure 26. No inconsistencies were detected in the test cases (see Figure 27) using the RE/TRAC-CF production rules specified in YACC. In this context, consistency relates to the fact that all erroneous RE/TRAC-CF sentences were detected and no sentence was accepted that was incorrect. An erroneous sentence is one which does not exhibit the hierarchical relational

structure of inheritance nor the component relationships of extension and inclusion. Example executions depicting error detection are located in Appendix F.

```
TRACE RULE T: TOKUSECASE | U29\2 |
TRACE RULE T: TOKUSECASE | U29\4 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
TRACE RULE Z: TOKLEFTPARENS T A TOKRIGHTPARENS | ( T A ) |
TRACE RULE I; TOKLEFTSQUARE Z TOKRIGHTSQUARE |[ Z ]|
TRACE RULE G: I | I |
TRACE RULE T: TOKUSECASE G | U46\1 G |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
ACCEPT TRACE RULE core_expression: TOKBASECASE A | b46 A |
```

Figure 26 Output of YACC with Fired RE/TRAC-CF Production Rules

5.3 Accepting the Example Problems

A tree-like structure is an intuitive depiction of progression from one step to another, whether it is modeling event control and/or object control. Warnier diagrams depict hierarchies using a horizontal tree of textual items connected with braces ({) and special relational symbols [Pfleeger]. Hierarchical dependency diagrams are also used in the depiction of the relatedness of objects used in separate compilation [Meyers] also. Dependency graphs are useful tree structures for showing the order of separate compilation of modules for the building of a Makefile. In a Makefile, script instructions search for the most recent version of a file and oversee recompilation of only those modules that have been changed and then assemble all in a prescribed sequence. The managing of the revision of documents including configuration files and requirement specification artifacts is an old problem of version control. Like source and object files, use cases are revised over time and have related dependencies and the process must be supervised over time.

Figure 27 contains example problems to which the RE/TRAC method is applied. Each example has been statically tested for its semantic meaning and the sentence structure verified

```
1. b1;
2. b32<(U32\1)>;
3. b56<(U56\1:1)(U56\1:2)(U56\1:3)>;
4. b49<(U49\1:1<(U49\2:1:1)>)(U49\1:2)(U49\1:3<(U49\2:3:1)(U49\2:3:2)>)>;
5. b49<(U49\1:1<(U49\2:1:1)>)(U49\1:2)(U49\1:3<(U49\2:3:1<(U49\3:3:1:1)>)>)>;
6. b46<(U46\1[(U29\2)])>;
7. b46<(U46\1[(U29\2<(U29\4)>)])>;
8. b46<(U46\1[(U29\2<(U29\4[(U45\5)])>)])>;
9. b46<(U46\1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)])>;
10. b46<(U46\1:1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)]<(U46\2:1:1)>)(U46\1:2)>;
11. b31<(U31\1[(U2\4[(U55\3)])(U5\2)(U6\1)])>;
12. b49<(U49\1[(U35\2)(U43\6)]<(U49\2[(U35\3)])>)>;
13. b67<(U67\1{(U54\2)})>;
14. b54<(U54\1{(U29\3)(U42\2)})>;
15. b59<(U59\1{(U29\3)}<(U59\2:1)(U59\2:2[(U2\1)])>)>;
16. b54<(U54\1{(U29\3)(U42\2<(U42\3:1)(U42\3:2)>)})>;
17. b25<(U25\1{(U4\1[(u3\3)])}<(U25\1{(U4\3)})>)>;
18. b25<(U25\1[(U32\2)]{(U45\3)})>;
19. b67<(U67\1[(U65\3)]{(U54\2[(U29\1)])})>;
```

Figure 27 Test Cases Supporting Version Control and Traceability

using YACC. Each sentence for the base case holds the necessary information for tracing the refinement of the base case, including its dependencies and any refinements to the dependencies.

The data in Figure 27 has been used as the abstract representation from which graphical RE/TRAC diagrams are created using open source graph visualization software, Graphviz. The dot language in Graphviz was used to create the RE/TRAC directed graphs. A C++ program was written to create a .dot file for each example data. The software Graphviz viewer, dotty, was then utilized to display the .dot file in a graphical structure. Dotty was selected because it employs a layout algorithm which aims edges in the same direction (top to bottom, or left to right) and then attempts to avoid edge crossings and reduce edge length. The high-level algorithm developed for generating a .dot file is provided in Figure 28. The source file, written in C++, for generating .dot

files is located in Appendix G. The .dot file generated from the RE/TRAC-CF expression numbered 9 in Figure 27 is provided in Figure 29 and the corresponding graph as drawn in dotty is shown Figure 30.

To generate a .dot file:

1. Build an adjacency table for the RE/TRAC-CF expression
2. Build .dot file
    a. Name nodes for .dot file
    b. Initialize graph
    c. Build the refinement stack top-down left to right
    d. Build the graph following language rules for dot
        For each node in the refinement stack:
        i. Init node cluster
        ii. For each node related via inclusion:
            1. Build the refinement stack (2c)
            2. Build the graph (2d)
        iii. Link to parent
        iv. For each node related via extension:
            1. Build the refinement stack (2c)
            2. Build the graph (2d)

Figure 28 High-Level Algorithm for Generating .dot File

While the dot language and the viewer Dotty were helpful to the research in substantiating that a visual diagram could be generated from a RE/TRAC-CF sentence, the performance of dot with dotty was rather poor. Performance was helpful but for actual use some features were not supported such as composition of the refinement connector, centering refinement connectors over the refinement use case figure, and setting rankings between extend dependencies and the parent. Other graphing softwares were explored such as MagicDraw, MSDN System.Drawing, Essential Diagram for asp.net, yWorks, and GDE—GoVisual Diagram Editor. Some were cost prohibitive and/or were lacking the flexibility needed in providing a diagrammatic programmable language

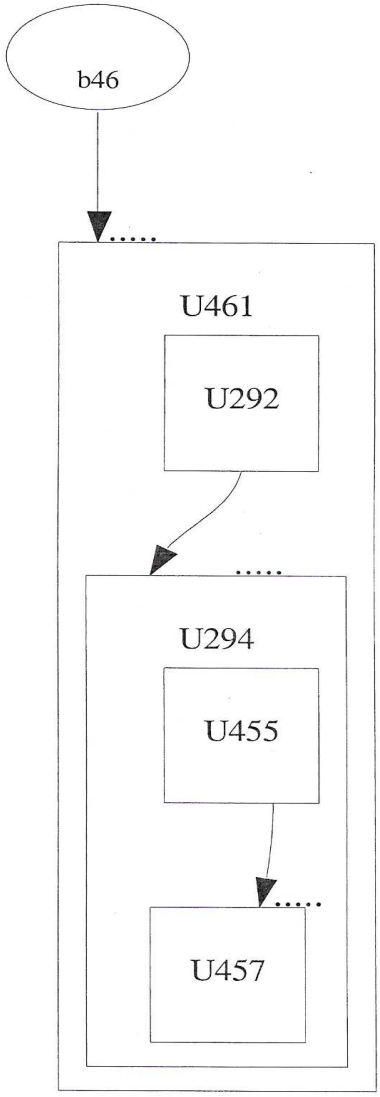
with hierarchical and ranking features for positioning of the primitives. Future work will include development of a better visualization tool.

```
digraph G {
compound=true;
fontsize = 12;
label="b46<(U46\1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)])>;"
ranksep=.5;
node[fontsize=10];

subgraph cluster0{
color=white;
node [shape=box];
label = "   ";
node0[label=b46, shape=ellipse];
} // end cluster0

subgraph cluster1{
label="U46\1";
node1[style=invis, fixedsize=true, height=.09, width=.09];
subgraph cluster2{
label="U29\2";
node2[style=invis, fixedsize=true, height=.09, width=.09];
} // end clustercluster2

subgraph cluster3{
label="U29\4";
node3[style=invis, fixedsize=true, height=.09, width=.09];
subgraph cluster4{
label="U45\5";
node4[style=invis, fixedsize=true, height=.09, width=.09];
} // end clustercluster4

subgraph cluster5{
label="U45\7";
node5[style=invis, fixedsize=true, height=.09, width=.09];
} // end clustercluster5

node4->node5[tailport=s, headport=n, label=".....", ltail=cluster4 ,lhead=cluster5];

} // end clustercluster3

node2->node3[tailport=s, headport=n, label=".....", ltail=cluster2 ,lhead=cluster3];

} // end clustercluster1

node0->node1[tailport=s, headport=n, label=".....", lhead=cluster1];

} // end graph
```

Figure 29 .dot File and Graphviz Dotty View for Expression `b46<(U46\1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)])>;`

Dotty view for expression:

```
b46<(U46\1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)])>;
```



b46<(U461[(U292<(U294[(U455<(U457)>)])>)])>;

Figure 30 Dotty View for RE/TRAC-CF Expression
`b46<(U46\1[(U29\2<(U29\4[(U45\5<(U45\7)>)])>)])>;`

5.4 Accepting Usefulness of Method

Reduction in cost is frequently related to timesaving and/or quality improvement. Overall timesaving of alternative design methods in general cannot be fully determined until the project and the control project(s) have been completed and made operational. A saving of time spent in the requirements analysis phase does not imply a savings overall. If software, especially for a one-of-a-kind small system, is in the maintenance phase, going back and establishing RE/TRAC version control of use cases might be impractical. However, if the software were intended for multiple product-lines or different applications within the same domain, then formalizing the use case manageability would benefit all subsequent applications. As mentioned earlier, some users, where a user is anyone involved in requirements evolution, are more adept at reading abstract visual diagrams than others. Also some users are more proficient at reading abstract visual diagrams than reading and comprehending organized textual documents. RE/TRAC provides an alternative view more easily comprehensible to some than others. This would amount to a timesaving for some users.

While quality is usually associated with the overall quality of the product, in this research we are concerned with the quality of the requirements artifacts in the form of use cases. Increased quality during requirements analysis is related to the overall quality of the end product [Schach]. In particular, we have provided a visual and structural framework for management of use cases. The dynamic semantics of RE/TRAC-SEM improves and preserves the quality of the method by providing a solid framework for the specification of use cases. The rigid yet informal set definitions and meta-restrictions restrict modifications so that the versions must conform to a vertical hierarchy. All information pertaining to a use case is stored in the single physical place in the form of a RE/TRAC-CF expression so that all views are consistent and correct.

5.5 Accepting That Usefulness Is Linked to Applying the Method

This research began with the visual model. The simplicity of the visual model drove the specification of the ontology. For instance, if an older use case could be altered and refined, then the diagram could have multiple versions of versions at various levels in the diagram; this could present a confusing graph. It would follow that current active use cases could be anywhere in the diagram which would reduce the comprehensibility considerably. A RE/TRAC diagram is read top to bottom and left to right as is the English language which follows an intuitive pattern to many people. This research has used a nesting graphical feature as analogous to the meaning of inclusion in textual form which is an intuitive similarity. Likewise to extend something is to supplement or to connect. RE/TRAC uses a connector with the same perceptive visual meaning.

Because a graphical depiction is difficult to store and to alter, the internal representation was simplified using a context-free grammar. But a grammatical representation is more difficult to read and interpret than the corresponding RE/TRAC diagram because of the length of some expressions, and the abstract symbols for refinements (<>), for inclusion ([ ]) and for extension ({}). The dynamic semantics ensures that the grammar representation conforms to the RE/TRAC syntax. The rules must be addressed before a change to a RE/TRAC-CF expression is made. The visual model and the ontology together result in a method that adds usefulness to requirements evolution. Figure 31 shows the synergistic relationship of the parts.

5.6 Accepting Usefulness of Method Beyond Example Problems

To prove theoretical performance validity, induction is based on the following. In section 5.1 we showed that the individual constructs of hierarchical method, visual languages and grammatical approaches are generally accepted for some applications. In section 5.2 we established that the grammar is consistent with the meaning of the refinement of use cases and

their dependencies via Lex and YACC. In section 5.3 we confirmed that graphs could be generated from a sentence in the RE/TRAC-CF grammar. In section 5.4 we demonstrated the usefulness of the method in relation to cost, time and quality. In section 5.5 we confirmed that the usefulness of a VL alone had limitations and that a context-free grammar sentence is a poor visual mechanism for comprehensibility. Therefore it is the method as a whole that generates usefulness to solve the problem of the maintainability of use cases documents.

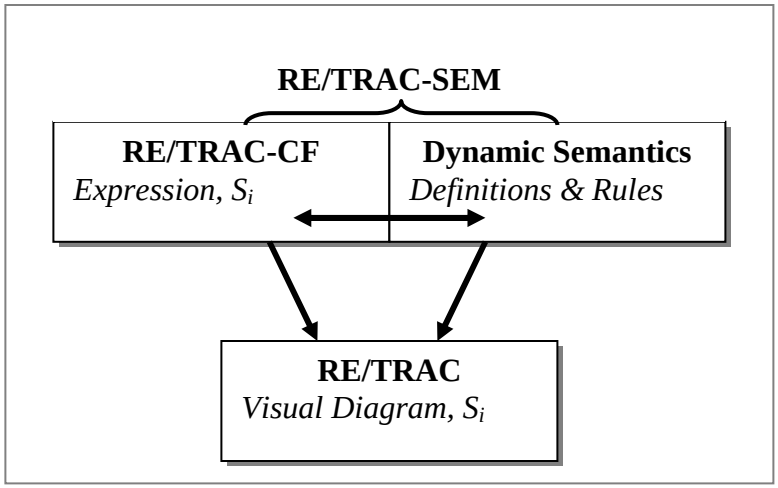


Figure 31 Relationships of RE/TRAC and RE/TRAC-SEM

The software developer and/or stakeholder will benefit from this research in practical use because RE/TRAC:

- Presents a visual model that requires little training for the non-technical stakeholder to understand but is based on formal methods.

  While a sentence in RE/TRAC-CF is difficult to decipher without tool support, a RE/TRAC diagram presents an evocative image of the relationship of use cases to one another. Sentences in RE/TRAC-CF must adhere to rules of the grammar and when changes to use case documents are made, the sentences are consistent with the

semantic rules. RE/TRAC is simple in definition and usage and therefore perhaps more narrowly applicable than requirements specification languages such as MSCs but more specific and suitable to its purpose.

- Incorporates traceability during the evolution of the requirements for tracing of the log history in both forward and backward directions.

  The importance of traceability is described in section 1.2 of this dissertation and is summarized here:

  1. Used to access the progress of the development,
  2. Provides documentation of the requirements specification stage,
  3. Logs changes for error and fault discovery and correction,
  4. Enables rollback to a stable requirements description,
  5. Clearly delineates the current state of the specification.

- Supports domain-independent modeling.

  RE/TRAC is beneficial to requirements specification in instances where a core set of domain requirements have been identified and structured. Example domains demonstrated in the research include the ATM example and the conference registration system. There are other domains that are equally suited to RE/TRAC.

- Works in unison with UML.

  RE/TRAC fits within the use case view of the UML and serves as a first step before generating scenario diagrams and activity diagrams. If the use cases are described using structured English, then RE/TRAC can serve as a bridge to the logical view and can facilitate automatic code generation.

- Supports requirements evolution for any software lifecycle process models.

  RE/TRAC can be employed with any process model such as the waterfall model, evolutionary development, formal systems development, incremental development and especially facilitates reuse-based development.

  RE/TRAC can be used for initial requirements specification, but was developed to target systems for evolutionary specification of a base set of requirements. It is consistent with a platform independent model in that varying target applications may be any programming language paradigm.

- Supports the customizing of requirements within a product-line.

  When a system is specified with consideration for developing product-lines, the requirements for commonality are separated from the variable requirements dependent on the product. RE/TRAC supports the specification of a common set of requirements for product-line development.

6. Summary and Conclusions

The primary goal of this research was to provide an easily comprehensible method for controlled refinement of requirements. The objective to meet this goal was to develop a graphical depiction of the refinement and dependencies of requirements based on formal methods that would benefit the stakeholder by increasing understandability, providing traceability and improving the quality of the representation. The method is described as large-grained for representing entities at a high-level of abstraction and coarse granularity.

To achieve this objective we defined a VL, named RE/TRAC (Requirements Evolution with Traceability), for depicting the evolution of change to use cases. RE/TRAC allows a high-level abstract view of the use case documents in a graphical format. The static and semantic syntax of the language were defined including the language primitives and the allowable visual language (VL) relationships between the primitives. This research presumed that a core set of use cases had been previously identified, named and described in natural language format.

A supporting specification, RE/TRAC-SEM, was described using a combination of specification models. The purpose of the formal specification is to constrain the evolution process in order to ensure that the integrity of the documents is maintained and that a diagram is a correct depiction of a core use case's history and present state. A context-free grammar, RE/TRAC-CF was selected as the underlying formal structure for the VL. The visual language 2-D primitives of the oval, the square, the vertical refinement connector, and the dashed extension connector correspond to terminals in the context-free grammar. Other RE/TRAC-CF primitives such as the '[' or the '{' describe how the 2-D primitives are concatenated. Conditional changes to use cases including relationships to dependent use cases were defined in the dynamic

semantics. Definitions and set definitions were first set forth followed by the meta-restrictions on the definitions constraining the use case refinement.

Validation of the method followed the validation square process. The software tools, Lex, YACC, and Graphviz were employed to validate the RE/TRAC and RE/TRAC-CF models. Section 4.3 presented an example application of the dynamic semantics, RE/TRAC-SEM, to a problem in the application domain. By applying change to a sentence in RE/TRAC-CF according to the commands in the meta-restrictions, correct revised sentences were created from which RE/TRAC diagrams were generated. A static validation check also confirmed that the RE/TRAC-SEM pre and post conditions were sufficiently specified.

6.1 Contributions

This research contributes to the evolution of requirements in the following ways:

- Specifies an ontology that supports component development of a system or a set of systems (domain).

  The ontology for RE/TRAC-SEM describes the rules and constraints for use case relationships. Entities (use cases) are described like components of the system where reuse of components and top-down ordering of components simplify the requirements specification and consequently the design and implementation.

- Provides a method for unifying views within the refinement of use cases.

  A RE/TRAC diagram and its corresponding RE/TRAC-CF sentence are parallel symbolic depictions of the requirements evolution process. Each representation presents a particular view of requirements evolution with traceability. The textual documents form another view that is in accordance with the RE/TRAC and RE/TRAC-CF representations.

- Specifies a supporting ontology to document artifacts in the repository for access in later stages of software design and implementation including automatic code generation.

  The use of RE/TRAC unifies the use case view and the logical view by describing the set of requirements current for a system. From the set of all S in RE/TRAC-CF, activity diagrams may be created/generated and later the class and object diagrams followed by the dynamic models (state, interaction and sequence diagrams) of the system. All views must be consistent.

- Specifies an ontology that constrains the refinement of use case documents in order to minimize points of change and therefore simplify the change process.

  Refinements as specified in the ontology must always occur to use cases that are represented as leaves in the RE/TRAC diagram. In this way, the diagram is always a top-down view of the elaboration of the requirements via the use case versions. Quick analysis of the existing current configuration is practicable, because active current use cases are always located in the leaves of a RE/TRAC diagram.

- Specifies an ontology that provides a trace of the document change sequence and supports the tracking of changes in the evolutionary process.

  The semantics of RE/TRAC does not permit deletion of a use case. In this manner a trace may always be made of the evolution of a use case including its dependencies.

Table 4 is an update of the research comparison table given in table 1 with one additional entry for RE/TRAC. As presented in table 1, diagram features are characterized as hierarchical (H), using components (C), and employing traceability (T). The degree of behavior is described as static (S), having limited dynamics (L), or descriptive of behavioral actions (B). RE/TRAC is

the only method discussed with limited dynamics. A RE/TRAC diagram is a static display, but it depicts discrete time changes in the sequencing of events. The RE/TRAC visual display does not highlight what the particular changes were and does not document the triggers that necessitated the update to the document(s). Therefore RE/TRAC does not strongly exhibit behavioral attributes. Quite a few of the methods are both hierarchical in depiction and support a component based diagram feature: Acyclic Call Graph [BatO'Mal], MSCs [Harel], UCMs [Amyot2003] and RSML [Lev et al]. But none of those that are both hierarchical and component based also depict traceability except for the Requirements State Machine (RSML). The Requirements State Machine Language comes the closest to RE/TRAC in functionality, however it is a graphical language with strong behavioral features. RE/TRAC is less complicated than RSML in terms of intuitive meaning determined from the number of primitives, connector types, and labeling. This research asserts that the non-technical stakeholder better understands a simple diagram, designed for the specific application of requirements modeling.

6.2 Future Work

Future work includes research of the following:

- Incorporating associations and relationships such as inclusive and exclusive conjunctions. A use case version may be associated via the extends relationship with multiple use cases, however only one of the use cases may be exclusively used in any one scenario. In this particular case, the exclusive context may be annotated. In other contexts, all use cases may be activated for a scenario, one or the other or both. In this case the inclusive relationship may be designated.
- Adding concepts such as those used in structure charts, for example illustrating an include relationship that repeats the use within the parent.

A use case version may rely on a dependent use case in an inclusion relationship. However, the dependent use case may be utilized in several instances within the parent use case and/or in varying locations. Some means of indicating the multiplicity could be documented in the RE/TRAC diagram.

- Including temporal information to the visual notation.

  In complex RE/TRAC diagrams where the tree is unbalanced, noting some temporal order could be beneficial.

- Create other views within RE/TRAC at lower levels of abstraction (greater detail).

  As other features are considered for RE/TRAC, keeping the diagram language simple and uncluttered will continue to be a goal in order to promote understandability. Therefore if increasing information is to be displayed, having separate views with alternative information may meet both needs. Another alternative would be to have the tool user select features to be displayed in a RE/TRAC diagram.

- Implement a query-based tool founded on RE/TRAC-SEM.

  Given a grammar to describe entities, such as features or requirements, and their relationships with other entities, a parser could be generated to enforce consistencies between the different versions for configuration control. If there are constraints to be placed on the refinements or upon the other relationships, then a type checker could be added to the parser. Using the grammar, a viewer and an editor could be developed to create an interactive environment in which the hierarchical information can be displayed in a succinct manner and changes validated. Once a structure is described, then queries may be generated such as:

all instances of usage,

which entities are active,

what is the derivation history of a particular entity,

what is the similarities/difference between two versions,

who is the parent.

When a change is made, then consistencies must be checked and enforced. If the entity is used in multiple places, then a change must be validated in all uses in order to maintain consistency.

Table 4: Comparison of Common Graphical Methods for Requirements Specification (including RE/TRAC)

Diagram	**Diagram Features**			**Degree of Behavior**	**Complexity**	**Usage**
RE/TRAC	H	C	T	L	1 does not label connectors	Graphical depiction of requirements evolution represented as use cases. Vertical top-down ordering.
Hierarchical Refinement of Classes [Batory et al]	H		T	S	1 does not label connectors	Depicts refinement of classes. Refinements are related to class generalizations. Traceability is viewed as levels when classes are refined via generalization of the class. Vertical, top-down ordering.
Acyclic Call Graph [BatO'Mal]	H	C		S	1 does not label connectors	Depicts function or method call relationships. Diagrams are read top-down, left-to-right for sequence interpretation of the calls. Vertical, top-down ordering.
[AstReg2002a] [AstReg2002b]				S	5	Structures and represents requirements specification artifacts in general, multi-view, use-case driven, UML-based (object-oriented). Systematic approach developed using UML.
Message Sequence Charts (MSCs) [Harel]	H	C		B	2	Visual formalism for capturing systems requirements as scenarios. Similar to UML sequence diagrams. Useful in system requirements capture.
User Requirements Notation: Use Case Maps (UCMs) [Amyot2003]	H	C		B	3	Standard visual notation used for specifying functional and non-functional requirements. Used for use case formulation, high-level architectural design and test case generation. Notation uses start points (pre-conditions), connectors and end points (post-conditions). Connector lines (paths) may be labeled with responsibilities.
Requirements State Machine Language (RSML) [Lev et al]	H	C	T	B	2	Uses a graphical hierarchical RSML specification which describes dynamic behavior defined by transitions and events.
Use Case Diagram (amended) [ChoReg]				S	1	Diagram and user-friendly notation that uses a NL-like language for specification of the use cases.

References

[AchSch] Achatz, K. and W. Schulte. “A Formal OO Method Inspired by Fusion and Object-Z”. *Proc. of the 10th International Conf. of Z Users on the Z Formal Specification Notation* (April 03 - 04, 1997). J. P. Bowen, M. G. Hinchey, and D. Till, Eds. LNCS 1212, pp. 92-111. Springer-Verlag, London.

[Amyot2003] Amyot, D.. “Introduction to the User Requirements Notation: Learning by Example”. *Computer Networks*, 42(3). pp. 285-301. 21 June 2003.

[Amyot2005] Amyot, D., *Use Case Maps Quick Tutorial Version 1.0*. 12 Mar. 2005 <http://www.usecasemaps.org/pub/UCMtutorial/UCMtutorial.pdf, >.

[AntPot] Anton, A. I. And C. Potts. “A Representational Framework for Scenarios of System Use”. *Requirements Engineering Journal,* 3(3-4). pp. 219-241. 1998.

[AstReg2002a] Astesiano, E. and G. Reggio, G. “Knowledge Structuring and Representation in Requirement Specification”. *Proc. of the 14th international Conf. on Software Engineering and Knowledge Engineering*, Ischia, Italy, (July 15 - 19, 2002), 27. ACM Press, New York, NY, pp. 143-150. 20 Mar 2005 < ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio02a.pdf >.

[AstReg2002b] Astesiano, E., and G. Reggio. “Tight Structuring for Precise UML-based Requirement Specifications”. In *Radical Innovations of Software and Systems Engineering in the Future, Proc. 9th Monterey Software Engineering Workshop*, Venice, Italy, (Sep. 2002). LNCS 2941. Berlin, Springer-Verlag, 2004.

[Batory] Batory, D. . “Feature Models, Grammars, and Propositional Formulas”. In *9th International Software Product Line Conf.* (2005). LNCS 3714, pp. 7-20. 2005.

[Batory et al] Batory, D., R. E. Lopez-Herrejon, and J. Martin. “Generating Product-Lines of Product-Families”. In *Proc. of The 17th IEEE Conference on Automated Software Engineering* (ASE 02). 2002.

[BatO’Mal] Batory, D. and S. O’Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM Transactions on Software Engineering and Methodology*, 1(4), Oct. 1992.

[BruDut] Bruegge, B. and A. H. Dutoit. Object-Oriented Software Engineering Using UML, Patterns, and Java. New Jersey: Pearson/Prentice Hall 2nd ed., pp. 121-169. 2004.

[ChoReg] Choppy, C. and G. Reggio. “Improving Use Case Based Requirements Using Formally Grounded Specifications”. FASE 2004. M. Wermelinger and T. Margaria-Steffen, Eds. LNCS 2984, pp. 244-261. 2004.

[Chom1957] Chomsky, N.. Syntactic Structures. The Hague: Mouton & Co, 1957.

[Chom1965] Chomsky, N.. Aspects of the Theory of Syntax. Cambridge: MIT Press. 1965.

[ChouHuang] Chou, S-C. and C-W. Huang. "A Software Product Model Emphasizing Relationships". In *Proc. 2nd Asia-Pacific Conference on Quality Software* (2001), pp. 417 – 426. 2001.

[Cockburn1997] Cockburn, A.. "Structuring Use Cases with Goals". *Journal of Object-Oriented Programming*. Sep-Oct, 1997 and Nov-Dec, 1997. 17 Mar. 2005 <http://members.aol.com/acockburn/papers/usecases.htm >.

[Cockburn1998] Cockburn, A.. "Basic Use Case Template", Oct. 1998. 18 Mar. 2005 <http://alistair.cockburn.us/usecases/uctempla.htm>.

[Crapo et al] Crapo, A. W., Waisel, L. B., Wallace, W. A., and Willemain, T. R.. "Visualization and the process of modeling: a cognitive-theoretic view". In *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Boston, Massachusetts, United States, August 20 - 23, 2000). KDD '00. ACM, New York, NY, pp. 218-226. 2000. doi: http://doi.acm.org/10.1145/347090.347129

[Crnk et al] Crnkovic, I., P. Funk, and M. Larsson. " Processing Requirements by Software Configuration Management". In *Proc. of the 25th EUROMICRO Conference* 2(1999), pp. 260-265, doi: 10.1109/EURMIC.1999. 794789.

[Dion et al] Dion, F., T. Tran, and A. Abran. " Mapping Processes Between Parallel, Hierarchical, and Orthogonal System Representations". IWSM 2000: pp. 233-244. 2000.

[DouCar2006] Douglas, C. and D. Carver. "Visual Language Support for Use Case Modeling". In *Proc. of the 4th International Conference on Computer Science and its Applications* (ICCSA-2006), eds. P. P. Dey, M. Amin and A. Datta. pp. 299-303. 2006.

[DouCar2007] Douglas, C and D. Carver. "Semantic Representation for Use Case Modeling". Technical Paper 2008-1, Department of Computer Science, Louisiana State University, 2008.

[DorFly] Dorfman, M. and R. Flynn. "ARTS – An Automated Requirements Traceability System". *Journal Systems and Software*, 4(1). pp. 63-74. 1984.

[DumHof] Dumas, M. and A. H. M. ter Hofstede. "UML Activity Diagrams as a Workflow Specification Language". In *Proc. of the 4th International Conf. on The UML, Modeling Languages, Concepts and Tools (2001)*. LNCS 2185. pp. 76-90. 2001.

[EgyGrn] Egyed, A. and P. Grünbacher. "Automating Requirements Traceability: Beyond the Record & Replay Paradigm". In *Proc. of the 17th IEEE International Conf.* (ASA'02). 2002.

[Erikson et al] Eriksson, H., M. Penker, B. Lyons, and D. Fado. UML 2.0 Toolkit. Indianapolis: Wiley Publishing. 2004.

[Euler] Euler, L.. Letters a une Princesse d' Allemangne. vol 2. 1772.

[Fickas et al] Fickas, S., T. Beauchamp and N. Mamy. “Monitoring Requirements: A Case Study”. In *Proc. of the 17th IEEE International Conf. on Automated Software Engineering* (ASA September 23 - 27, 2002). 2002.

[Foreman] Foreman, J.. “Product-line Based Software Development – Significant Results, Future Challenges”. In *Proc. of the Software Technology Conf.*, Salt Lake City, UT. 1996.

[GotFin] Gotel, O. and A. Finkelstein. “An Analysis of the Requirements Traceability Problem”. *1st International Conf. on Requirements Engineering* (ICRE 1994). pp. 94-101. 1994. 4 Dec. 2005 <http://www.cs.ucl.ac.uk/staff/A.Finkelstein/papers/rtprob.pdf>.

[Gruber] Gruber, T.. “Translation Approach to Portable Ontology Specifications”. *Knowledge Acquisition* 5(1993)199-220, doi: 10.1006/knac.1993.1008.

[Harel] Harel, D.. “Message Sequence Charts”. 2003. accessed 23 April 2005 <http://www.comp.nus.edu.sg/~thiagu/public_papers/surveymsc.pdf>.

[HeiKee] Heimdahl, M. and D. Keenan. "Generating Code from Hierarchical State-Based Requirements". *3rd IEEE International Symposium on Requirements Engineering* (RE'97). p. 210. 1997.

[Ince] Ince, D. C.. An Introduction to Discrete Mathematics, Formal System Specification, and Z. New York: Oxford University Press, Inc. 1993.

[Johnson] Johnson-Laird, P.. Human and Machine Thinking. Lawrence Erlbaum Associates; Hillsdale, N.J.. 1993.

[JonViss] de Jong, M. and J. Visser. “Grammars as Feature Diagrams”. Unpublished. Presented at the Generative Programming Workshop 2002, Austin, Texas. 12 Dec. 2005 <http://www.di.uminho.pt/~joostvisser/publications/GrammarsAsFeatureDiagrams.pdf>.

[Kal et al] Kalinichenko, L., M. Missikoff, F. Schiappelli and N. Skvortsov. “Ontological Modeling”. In *Proc. of the 5th Russian Conf. on Digital Libraries*, (2003). 2003.

[Kang et al] Kang, K., S. Cohen, J. Hess, W. Nowak, and S. Peterson. “Feature –Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report, CMU/SEI-90TR-21. November 1990.

[KenKen] Kendall, K. and J. Kendall. Systems Analysis and Design. New Jersey: Prentice Hall. 1999.

[Lev et al] Leveson, N., M. P. E. Heimdahl, H. Hildreth, and J. Reese. “Requirements specification for process-control systems”. *IEEE Transactions on Software Engineering*, 20(9), September 1994.

[Li] Li, L.. “Translating Use Cases to Sequence Diagrams”. 15th IEEE International Conferenece on Automated Software Engineering (ASE’00). pp. 293-296. 2000.

[LuqGog] Luqi and J. Goguen. “Formal Methods: Promises and Problems”. *IEEE Software*, 14(1). pp. 73-85. Jan. 1997.

[Mannion] Mannion, M.. “Using First-Order Logic for Product Line Model Validation”. In *Proc. of the 2nd international Conf. on Software Product Lines* (August 19 - 22, 2002). G. J. Chastek, Ed. LNCS 2379, pp. 176-18. Springer-Verlag, London, 7. 2002.

[MarMey] Marriott, K. and B. Meyer. “Introduction”. In Marriott, K. Meyer, B. (Eds.), Visual Language Theory. Springer-Verlag, New York. pp. 1-4. 1998.

[Mar et al] Marriott, K., B. Meyer, B., and K. Wittenburg. “A Survey of Visual Language Specification and Recognition”. In Marriott, K. Meyer, B. (Eds.), Visual Language Theory. Springer-Verlag, New York, pp. 5-86. 1998.

[Meyers] Myers, A. C.. “Bidirectional object layout for separate compilation”. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (Austin, Texas, United States, October 15 - 19, 1995). OOPSLA '95. ACM Press, New York, NY. pp. 124-139. DOI= http://doi.acm.org/10.1145/217838.217849. 1995.

[MOF] Object Management Group. “Meta Object Facility (MOF) Specification”. Version 1.4, OMG Document: formal/02-04-03. April 2002.

[Morisio et al] Morisio, M., G. H. Travassos, and M. Stark. “Extending UML to Support Domain Analysis”. 15th IEEE International Conf. on Automated Software Engineering (ASE’00), September 11-15, 2000. p. 321. 2000.

[NarHüb] Narayanan, H. and Hübscher, R. (1998) “Visual Language Theory: Towards a Human-Computer Interaction Perspective”. In: Marriott, K. Meyer, B. (Eds.). Visual Language Theory. Springer-Verlag, New York, pp 87-128. 1998.

[PetPed] Peters, and Pedrycz, W. Software Engineering: an Engineering Approach. John Wiley & Sons, Inc. p. 10. 2000.

[OMG] Object Management Group. “UML 2.0 Infrastructure Final Adopted Specification”, OMG document ptc/03-09-15, (2003). 2003.

[Ped et al] Pedersen, K., J. Emblemsvåg, R. Bailey, J.K. Allen and F. Mistree. "Validating Design Methods & Research: The Validation Square". 2000 ASME Design Engineering Technical Conference (Baltimore, Maryland, Sept 10 - 14, 2000). 2000.

[Pender] Pender, T., UML Bible. Indianapolis: Wiley Publishing. 2003.

[Pfleeger] Pfleeger, S.. Software Engineering Theory and Practice. New Jersey: Prentice Hall. 1998.

[Pinheiro] Pinheiro, F.. "Formal and Informal Aspects of Requirements Tracing". 5 Dec. 2005 <http://64.233.167.104/search?q=cache:tJxUdJHCoWIJ:www.inf.puc-rio.br/~wer00/zip/pinheiro.ps+Pinheiro+Formal+Informal+Aspect+requirements+tracing&hl=en >.

[PinGog] Pinheiro, F. and J. Goguen. "An Object-Oriented Tool for Tracing Requirements". *IEEE Software*, 13(2). pp. 52-64. March 1996.

[Reniers] Reniers, M.A.. Message Sequence Chart: Syntax and Semantics. PhD Thesis, Eindhoven University of Technology, June 1999.

[Rodri et al] Rodríguez-Fortiz, M. and P. Paderewski-Rodríguez, L. Garcia-Cabrera and J. Parets-Llorca, "Evolutionary Modeling of Software Systems: its Application to Agent-based and Hypermedia Systems". In *Proc. of the 4th International Workshop on Principles of Software Evolution* (ACM Press, New York, 2001), pp. 62-69, doi: 10.1145/602461.602471.

[Schach] Schach, S.. Object-Oriented & Classical Software Engineering. 7th ed. McGraw Hill Higher Education. 2007.

[Smith] Smith, G.. The Object-Z Specification Language Advances in Formal Methods. Kluwer Academic Publishers. 2002.

[Sommerville] Sommerville, I.. Software Engineering. 6th ed. Addision-Wesley. p. 395. 2001.

[StenGur] Stenning, K. and C. Gurr."Human-formalism Interaction: Studies in Communication through Formalism". *Interacting with Computers*. 9(2). pp 111-128. 1997.

[Sutcliffe] Sutcliffe, A.. The Domain Theory Patterns for Knowledge and Software Reuse. Lawrence Erkbaum Associates Publishers. 2002.

[UschJas] Uschold, M. and R. Jasper. "A Framework for Understanding and Classifying Ontology Applications". In *Proc. of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods* (Stockholm, Sweden, 1999). 1999.

[Venn] Venn J. "On the Diagrammatic and Mechanical Representation of Propositions and Reasonings". Dublin Philosophical Magazine and Journal of Science. 9(59). pp. 1-18. 1880.

Appendix A: Example Use Case Template

Consent

Date: Fri, 22 Apr 2005 09:17:24 EDT
From: ACockburn@aol.com
To: douglas@bit.csc.lsu.edu
Subject: Re: Permission for use of Basic Use Case Template
Parts/Attachments:
1 OK 51 lines Text
2 Shown ~65 lines Text

Of course, you are most welcome.
Alistair

In a message dated 4/21/2005 9:14:38 P.M. Mountain Daylight Time, douglas@bit.csc.lsu.edu writes:

> I am working on my proposal for my dissertation and I am referencing your
> work on Use Case Templates.
> May I use the templates from your website, "Basic Use Case Template" as
> entries in the appendix of my research document.
> Sincerely,
> Coretta Douglas

Template in Table Form http://alistair.cockburn.us/usecases/uctempla.htm

USE CASE #	< the name is the goal as a short active verb phrase>	
Goal in Context	<a longer statement of the goal in context if needed>	
Scope & Level	<what system is being considered black box under design> <one of : Summary, Primary Task, Subfunction>	
Preconditions	<what we expect is already the state of the world>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Condition	<the state of the world if goal abandoned>	
Primary, Secondary Actors	<a role name or description for the primary actor>. <other systems relied upon to accomplish use case>	
Trigger	<the action upon the system that starts the use case>	
DESCRIPTION	Step	Action
	1	<put here the steps of the scenario from trigger to goal delivery,and any cleanup after>
	2	<...>
	3	
EXTENSIONS	Step	Branching Action
	1a	<condition causing branching> : <action or name of sub.use case>
SUB-VARIATIONS		Branching Action
	1	<list of variation s>
RELATED INFORMATION	<Use case name>	
Priority:	<how critical to your system / organization>	
Performance	<the amount of time this use case should take>	
Frequency	<how often it is expected to happen>	
Channels to actors	<e.g. interactive, static files, database, timeouts>	
OPEN ISSUES	<list of issues awaiting decision affecting this use case >	
Due Date	<date or release needed>	
...any other management information...	<...as needed>	
Superordinates	<optional, name of use case(s) that includes this one>	
Subordinates	<optional, depending on tools, links to sub.use cases>	

Appendix B: Use Case Refinement

Course of Events – Version 1:

[1]**Use Case:** Withdraw Cash from an ATM
Actors: Customer, Bank Info System
Purpose: To process customer's request for cash
Overview: Customer arrives at ATM. Customer logs in. ATM gives the Customer options and the customer chooses withdraw cash. The Customer requests funds from desired account and if there are sufficient funds, the ATM processes the request.
Type: Primary and essential

Actor Actions	*System Response*
1. Customer arrives at ATM	
2. Customer Id's self to ATM	
	3. Verifies valid customer *Constraint:* If invalid customer ID, stop transaction.
	4. Displays options
5. Customer chooses withdraw cash	
	6. System displays choice of accounts
7. Customer chooses account	
	8. System asks for amount
9. Customer enters amount desired	
	10. System checks Bank Info System
	11. Bank Info System returns request status *Note:* If status insufficient funds, return Insufficient Funds"
12. Customer views request status	
	13. System dispenses desired amount *Constraint:* If status insufficient funds, do not dispense.
	14. System prints out receipt

1. Based on: http://www.lv.psu.edu/cad18/ist240/ucn%20sect1%20ex.htm

Use Case Refinement

Course of Events – Version 2:

Actor Actions	*System Response*
1. Customer arrives at ATM	
2. Customer activates the ATM	
	3. System checks for specific ATM verification method.
	4. Directions for ID verification method displayed.
5. Customer Id's self	
	6. Verifies valid customer *Constraint:* If invalid customer ID, stop transaction.
	7. Displays options
8. Customer chooses withdraw cash	
	9. System displays choice of accounts
10. Customer chooses account	
	11. System asks for amount
12. Customer enters amount desired	
	13. System checks Bank Info System
	14. Bank Info System returns request status *Note:* If status insufficient funds, return "Insufficient Funds"
15. Customer views request status	
	16. System dispenses desired amount *Constraint:* If status insufficient funds, do not dispense.
	17. System prints out receipt

Use Case Refinement

Course of Events – Version 3:

Actor Actions	*System Response*
1. Customer arrives at ATM	
2. Customer activates the ATM	
	3. System checks for specific ATM verification method.
	4. Directions for ID verification method displayed.
5. Customer Id's self	
	6. Verifies valid customer *Constraint:* If invalid customer ID, stop transaction.
	7. Displays options
8. Customer chooses withdraw cash	
	9. System displays choice of accounts
10. Customer chooses account	
	11. System asks for amount
12.Customer enters amount desired	
	13. System checks Bank Info System
	14. Bank Info System returns request status *Note:* If status insufficient funds, return "Insufficient Funds" **check instant loan approval,** **return instant loan approval amount**
15. Customer views request status ***Constraint:*** **If Customer approved for instant loan** **Customer accepts or declines**	
	16. System dispenses desired amount *Constraint:* If status insufficient funds **or** **(status instant loan approved and customer declines),** do not dispense. **If (status insufficient funds and customer accepts),** **instigate Use Case: ATM Instant Loan**
	17. System prints out receipt

Use Case Refinement

Course of Events – Version 4:

Actor Actions	*System Response*
1. Customer arrives at ATM	
2. Customer activates the ATM	
	3. Instigate Use Case: "System Verifies Customer"
	4. Displays options
5. Customer chooses withdraw cash	
	6. System displays choice of accounts
7. Customer chooses account	
	8. System asks for amount
9. Customer enters amount desired	
	10. System checks Bank Info System
	11. Bank Info System returns request status returns instant loan approval amount *Note:* If status insufficient funds, return "Insufficient Funds"
12. Customer views request status *Constraint:* If Customer approved for instant loan Customer accepts or declines	
	13. System dispenses desired amount *Constraint:* If status insufficient funds or (status instant loan approved and customer declines), do not dispense. If (status sufficient funds and customer accepts), instigate Use Case: ATM Instant Loan
	14. System prints out receipt

Appendix C: Consent for Diagram Use Figures 1 [Batory et al] and 2 [BatO'Mal]

Date: Thu, 26 Jul 2007 09:36:07 -0500
From: Don Batory <batory@cs.utexas.edu>
To: Coretta Douglas <douglas@csc.lsu.edu>
Subject: Re: LSU/ Computer Science - request permission to print

Coretta:

Sure, use whatever you want with citations.
and when you finish, send me a pdf of your thesis!

don

----- Original Message ----- From: "Coretta Douglas" <douglas@csc.lsu.edu>
To: <batory@cs.utexas.edu>
Sent: Thursday, July 26, 2007 9:23 AM
Subject: LSU/ Computer Science - request permission to print

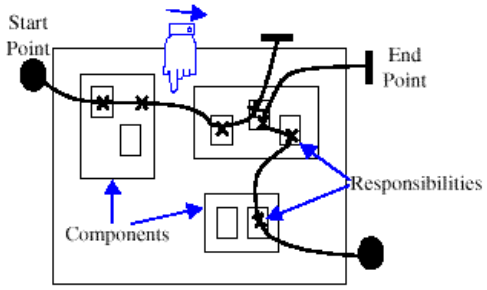
> Hi Dr. Batory,
> I am requesting permission to use
> the following diagrams in my dissertation
> under the direction of Dr. Doris Carver, my
> major professor:
> 1. DIAGRAM p. 7 from
> [BatoryO.Malley] Batory, D. and S. O.Malley. .The Design and Implementation
> of Hierarchical Software Systems with Reusable Components.. ACM Transactions
> on Software Engineering and Methodology, 1(4), Oct. 1992.
> 2. FIGURE 6 from
> [Batory et al] Batory, D., R. E. Lopez-Herrejon, and J. Martin. .Generating
> Product-Lines of Product-Families.. In Proc. of The 17th IEEE Conference on
> Automated Software Engineering (ASE 02). 2002.
>
> Sincerely,
> Coretta Douglas
> Louisiana State University
> Department of Computer Science
> Software Engineering Laboratory
> | Computer Science Department | OFFICE: (225) 578-4359 |
> | Louisiana State University | FAX: (225) 578-1465 |
> | 295 Coates Hall | EMAIL: douglas@bit.csc.lsu.edu |
> | Baton Rouge, LA 70803 | MAIN OFFICE: (225) 578-1495 |
>

Appendix D: Use Case Map Reference Guide
Consent

Date: Thu, 21 Apr 2005 23:12:47 -0400
From: Daniel Amyot <damyot@site.uottawa.ca>
To: Coretta Douglas <douglas@bit.csc.lsu.edu>
Subject: Re: permission to use Use Case Maps Quick Reference Guide

Hello Coretta,
Certainly, please go ahead. What is your proposal about?
Regards,
Daniel
Coretta Douglas wrote:
> I am referencing your work in my proposal for dissertation research. I am
> asking permission to include your
>   Use Case Maps Quick Reference Guide
> in the appendix of the research proposal.
> http://www.usecasemaps.org/pub/UCMtutorial/UCMquickRef.html
>
> Sincerely,
Coretta Douglas

Use Case Map Quick Reference Guide



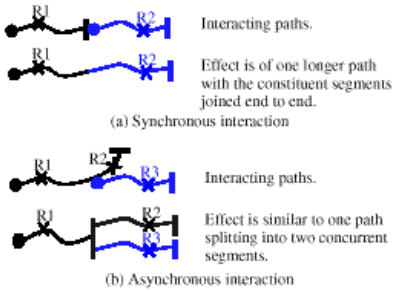
Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

- **start points** (filled circles representing pre-conditions or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing post-conditions or resulting effects).

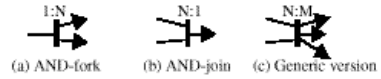
The responsibilities can be bound to **components**, which are the entities or objects composing the system.

A1. Basic notation and interpretation

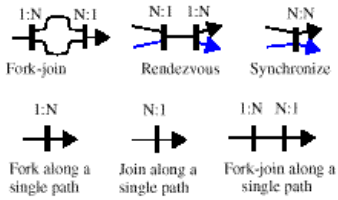
A2. Shared routes and **OR-forks/joins**.



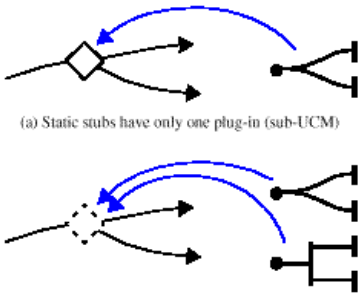
A3. Path interactions.



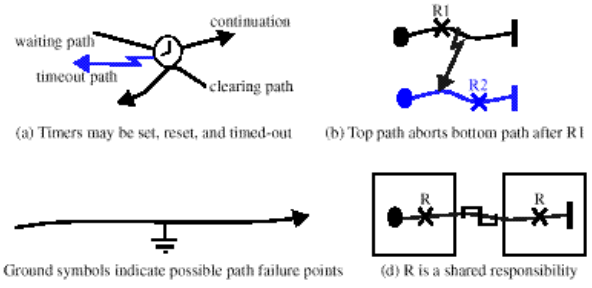
A4. Concurrent routes with **AND-forks/joins**.



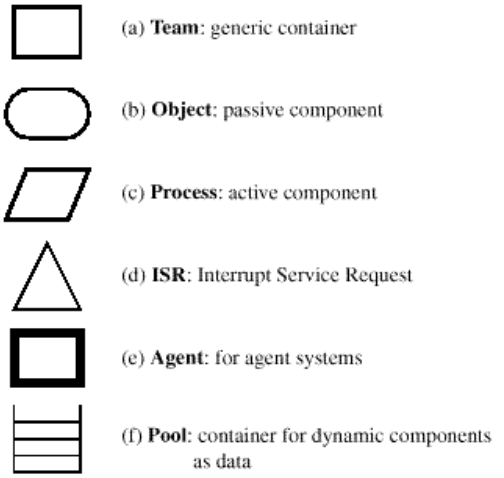
A5. Variations on AND-forks/joins.



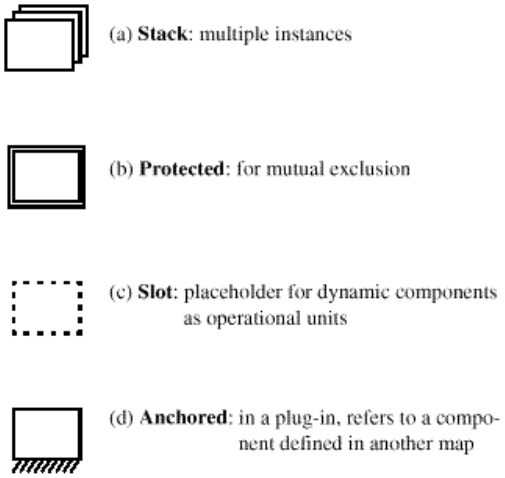
A6. **Stubs** and **plug-ins**.



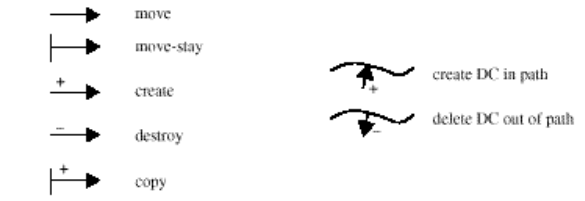
A7. **Timers, aborts, failures**, and **shared responsibilities**.

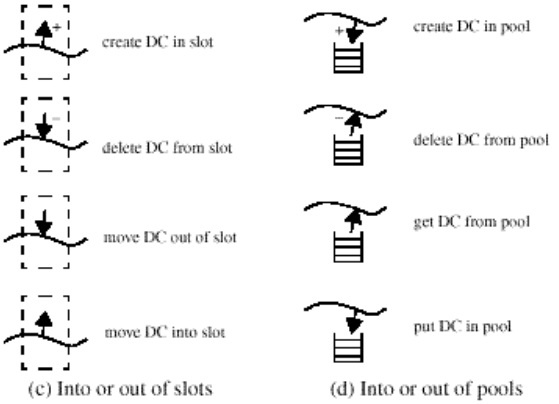


A8. Component types.

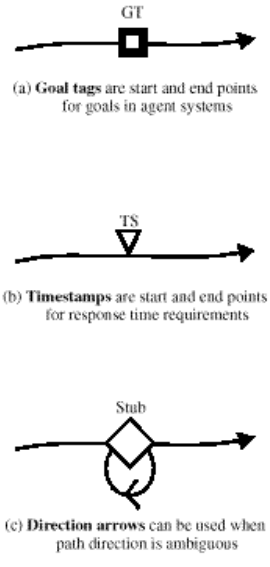


A9. Component attributes.





A10. Movement notation for **dynamic components** (DCs).



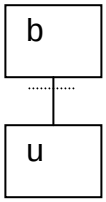
A11. Notation extensions

Daniel Amyot, September 2, 1999 [Amyot2005]

Appendix E Grammar and Diagrammatic Examples

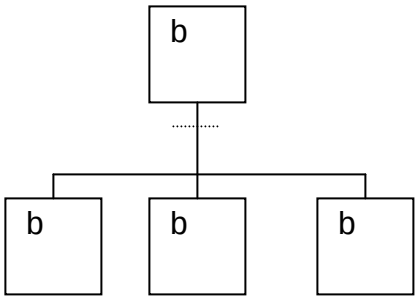
EXAMPLE: 1
DESCRIPTION: base case is replaced
by a single use case.
Sentence:

```
     b<(u)>
        T
       (T)
        Z
      < Z >
        A
     b  A
       S
```



EXAMPLE: 2
DESCRIPTION: base case is replaced by multiple use cases.
Sentence:

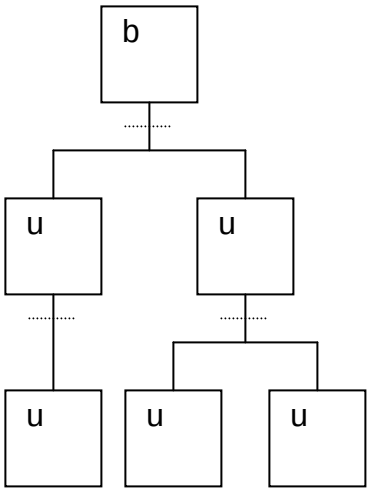
```
     b<(u)(u)(u)>
              T
             (T)
              Z
           T
          (T)
           Z
             Z
       T
      (T)
       Z
          Z
      <   Z    >
          A
     b    A
        S
```



EXAMPLE 3:
DESCRIPTION: base case is first refined into 2 use cases. There is a subsequent level of refinement for both of the use cases. The first is refined by a single use case; the other refined by splitting into 2 use cases.
Sentence:

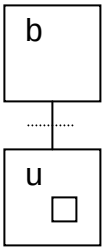
b<(u<(u)>)(u<(u)(u)>)>

```
                    T
                   (T)
                    Z
                  T
                 (T)
                  Z
                    Z
                <   Z  >
                    A
               u    A
               T    A
              (T    A   )
                 Z
        T
       (T)
      < Z >
     u  A
     T  A
    (T  A  )
       Z
            Z
   <       Z           >
           A
   b       A
        S
```



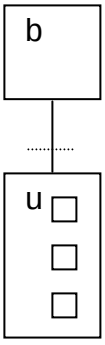
EXAMPLE: 4
DESCRIPTION: The base is refined to a use case which has a single use case accessed via inclusion.
SENTENCE:

```
        B< (u[(u)]) >
              T
             (T)
              Z
            [ Z ]
              I
           u  I
              T
           (  T   )
              Z
         <    Z     >
              A
        b     A
           S
```



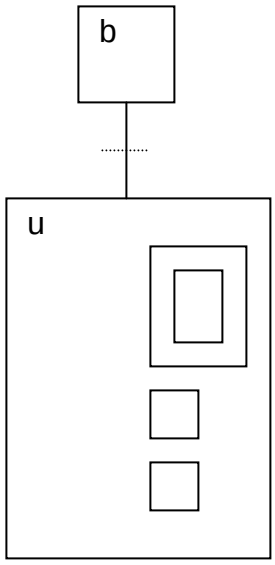
EXAMPLE: 5
DESCRIPTION: A base case is refined by a use case with multiple use cases included.
SENTENCE:

```
       b< (u[(u)(u)(u)]) >
                    T
                   (T)
                    Z
                  T
                 (T)
                  Z
                    Z
              T
             (T)
              Z
                  Z
           [     Z   ]
                 I
           u     I
              T
           (  T       )
              Z
         <    Z         >
              A
         b    A
           S
```



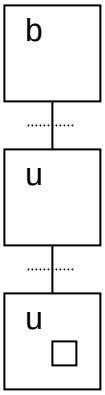
EXAMPLE: 6
DESCRIPTION: a nested inclusion is shown
SENTENCE:

```
       b< (u[(u[(u)])(u)(u)]) >
                          T
                        (T)
                          Z
                       T
                      (T)
                        Z
                        Z
                  T
                 (T)
                  Z
                [ Z ]
                  I
              u   I
                T
             (  T   )
                Z
           [   Z  ]
                I
          u    I
             T
        (     T   )
             Z
                    Z
        <       Z              >
       b         A
            S
```



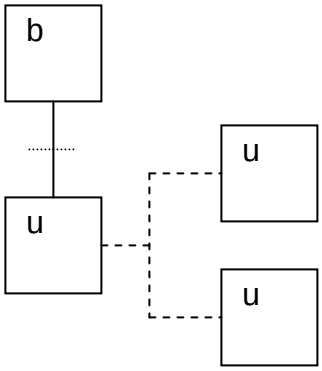
EXAMPLE: 7
DESCRIPTION: The base case is first refined and then the refinement is refined with a use case which has an inclusion.
SENTENCE:

```
      b< (u<(u[(u)])> ) >
                T
               (T)
                Z
               [ Z ]
                I
              u  I
                T
             (  T   )
              <   Z    >
            u    A
            T    A
           (T    A      )
              Z
       <     Z           >
              A
      b       A
         S
```



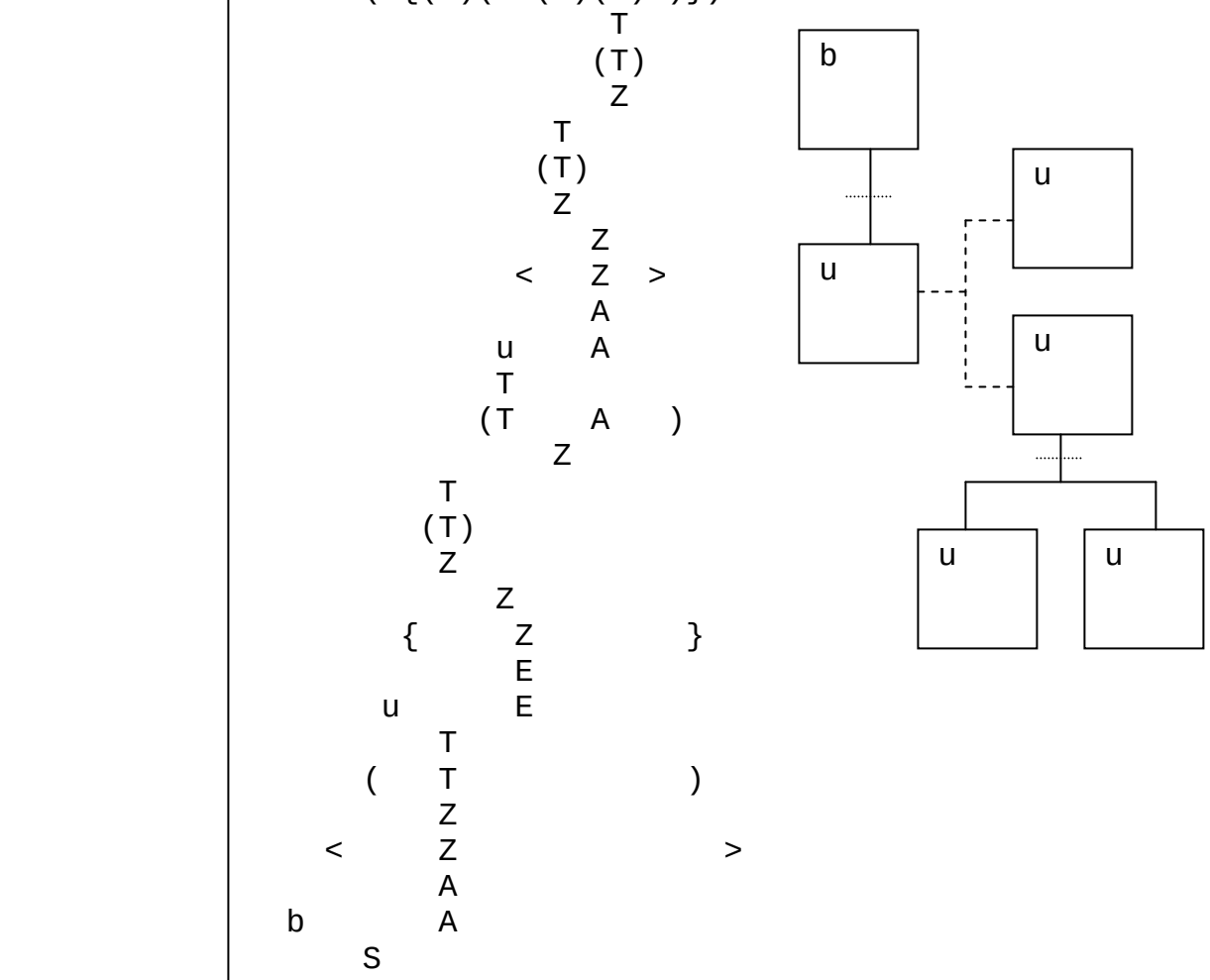
EXAMPLE: 8
DESCRIPTION: The base case is refined with a use case with two extensions.
SENTENCE:

```
       b< (u{(u)(u)}) >
                 T
                (T)
                 Z
               T
              (T)
               Z
                 Z
            {   Z }
                E
           u    E
              T
          (   T    )
              Z
       <      Z       >
              A
       b      A
           S
```



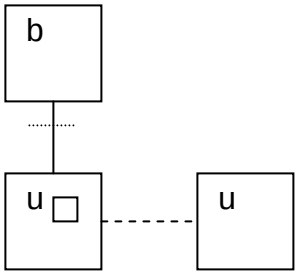
EXAMPLE: 9
DESCRIPTION: the base case is refined by a use case with two extensions. The second extension has been refined by a split into two separate use cases.

SENTENCE:
b< (u{(u)(u<(u)(u)>)}) >



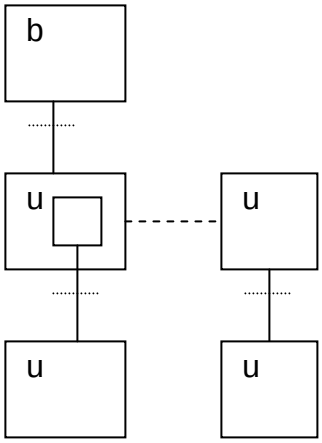
EXAMPLE: 10
DESCRIPTION: A first refinement contains both an inclusion and an extension.
SENTENCE:

```
        b<(u[(u)]{(u)})>
                   T
                  (T)
                   Z
                 { Z }
                   E
              T
             (T)
              Z
            [ Z ]
              I
           u
              I    E
              T
         (    T       )
              Z
        <     Z        >
              A
       b      A
          S
```



EXAMPLE: 11
DESCRIPTION: A base case is refined by a use case with both an inclusion and an extension. The inclusion is then refined as well as the extension.
SENTENCE:

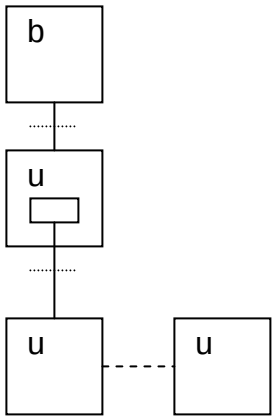
```
   b< (u[(u<(u)>)]{ (u<(u)>)}) >
                         T
                        (T)
                         Z
                       < Z >
                         A
                       u A
                       T A
                      (  T  )
                         Z
                   {     Z   }
                         E
             T
            (T)
             Z
           < Z >
             A
         u   A
         T   A
        (T   A  )
             Z
       [     Z   ]
             I
     u       I          E
             T
    (        T               )
             Z
  <          Z                 >
             A
   b         A
             S
```



EXAMPLE: 12
DESCRIPTION: The base case is refined by a use case with an inclusion. The inclusion is then refined by a use case with an extension.

SENTENCE:

```
    b<(u[(u<(u{(u)})>)])>
                 T
                (T)
                 Z
               { Z }
                 E
              u  E
                 T
               (  T   )
              <   Z    >
            u     A
            T     A
          (     T      )
                Z
        [       Z       ]
                I
       u        I
           T
      (    T          )
           Z
     <     Z           >
           A
    b      A
       S
```

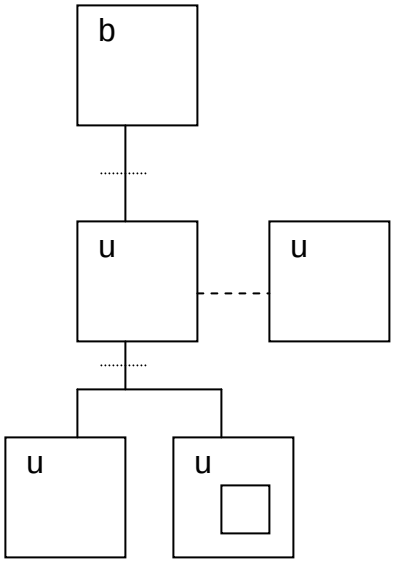


EXAMPLE: 13
DESCRIPTION: The base case is refined with a use case and an extension. The refined use case is then refined with two use cases one of which has an inclusion. Note that the extension in the first refinement is no longer active.

SENTENCE:

```
     b< (u{(u)}<(u)(u[(u)])>) >
```

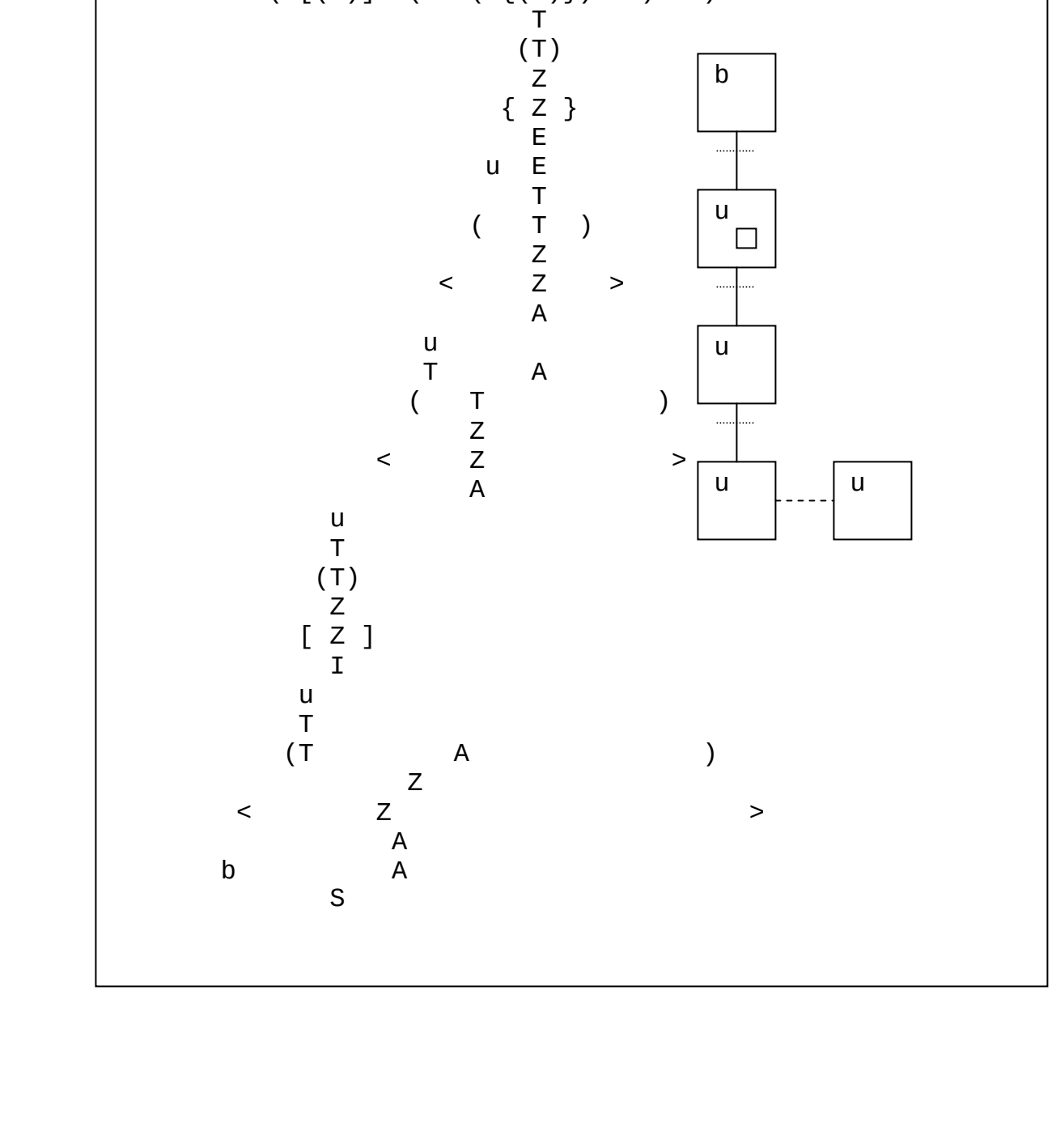
```
                      T
                     (T)
                      Z
                    [ Z ]
                      I
                   u  I
                     T
                  (  T   )
                     Z
               T
              (T)
               Z
                  Z
            <     Z      >
                  A
```




```
          T
         (T)
          Z
        { Z }
          E
       u  E
         T
       (  T       A        )
             Z
      <      Z                >
             A
     b       A
         S
```

EXAMPLE: 14
DESCRIPTION: The base case is replaced by a use case with an inclusion. The next refinement replaces all previous use cases with a use case with an extension.
SENTENCE:

b< (u[(u)]< (u< (u{(u)}) >) >) >



Appendix F: Example Lex and YACC Error Detection

EXAMPLE 1: Left refinement missing

```
Script started on Tue Jul 31 09:51:20 2007
%cat RETRACexp3_header.in

2
b1;
b32(U32\1)>;

%lex re1.l
%yacc -d -v re1.y
yacc: 1 shift/reduce conflict
%cc lex.yy.c ytab.c -o re1
%re1
invalid character

ACCEPT TRACE RULE core_expression: TOKBASECASE | b1 |

syntax error

%exit exit Script done on Tue Jul 31 09:52:09 2007
```

EXAMPLE 2: Right Parentheses Missing on Multiple

```
Script started on Tue Jul 31 09:50:05 2007
%cat RETRACexp3_header.in

3
b1;
b32<(U32\1)>;
b56<(U56\1:1)(U56\1:2(U56\1:3)>;

%lex re1.l
%yacc -d -v re1.y
yacc: 1 shift/reduce conflict
%cc lex.yy.c y.tab.c -o re1
%re1
invalid character

ACCEPT TRACE RULE core_expression: TOKBASECASE | b1 |
TRACE RULE T: TOKUSECASE | U32\1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
ACCEPT TRACE RULE core_expression: TOKBASECASE A | b32 A |

TRACE RULE T: TOKUSECASE | U56\1:1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |

syntax error

%exit exit Script done on Tue Jul 31 09:50:46 2007
```

EXAMPLE 3: Right Parenthesis Missing

```
Script started on Tue Jul 31 09:48:36 2007
%cat RETRACexp3_header.in
4
b1;
b32<(U32\1)>;
b56<(U56\1:1)(U56\1:2)(U56\1:3)>;
b49<(U49\1:1<(U49\2:1:1)>)(U49\1:2)(U49\1:3<(U49\2:3:1)(U49\2:3:2)>>;

%lex re1.l
%yacc -d -v re1.y
yacc: 1 shift/reduce conflict
%cc lex.yy.c y.tab.c -o re1
%re1
invalid character
ACCEPT TRACE RULE core_expression: TOKBASECASE | b1 |
TRACE RULE T: TOKUSECASE | U32\1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
ACCEPT TRACE RULE core_expression: TOKBASECASE A | b32 A |
TRACE RULE T: TOKUSECASE | U56\1:1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE T: TOKUSECASE | U56\1:2 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE T: TOKUSECASE | U56\1:3 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE Z: Z Z | Z Z |
TRACE RULE Z: Z Z | Z Z |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
ACCEPT TRACE RULE core_expression: TOKBASECASE A | b56 A |
TRACE RULE T: TOKUSECASE | U49\1:1 |
TRACE RULE T: TOKUSECASE | U49\2:1:1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
TRACE RULE Z: TOKLEFTPARENS T A TOKRIGHTPARENS | ( T A ) |
TRACE RULE T: TOKUSECASE | U49\1:2 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE T: TOKUSECASE | U49\1:3 |
TRACE RULE T: TOKUSECASE | U49\2:3:1 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE T: TOKUSECASE | U49\2:3:2 |
TRACE RULE Z: TOKLEFTPARENS T TOKRIGHTPARENS | ( T ) |
TRACE RULE Z: Z Z | Z Z |
TRACE RULE A: TOKLEFTANGLE Z TOKRIGHTANGLE | < Z > |
syntax error

%exit exit Script done on Tue Jul 31 09:49:32 2007
```

Appendix G: Source Code to Generate .dot Files

```
Script started on Mon Jul 30 10:40:19 2007
%cat InitGraphTable.h


void   InitGraphTable(GRAPHTABLE graphTable[]) {
   int countNodes;
   int countDeps;

   for (countNodes = 0; countNodes < MAXNODES; countNodes++) {
      graphTable[countNodes].visit = false;
      graphTable[countNodes].gTitleNum = -1;
      graphTable[countNodes].gTitleStr = "----";
      graphTable[countNodes].gParent   = -1;
      graphTable[countNodes].gCtRefinements = 0;
      graphTable[countNodes].gCtInclusions   = 0;
      graphTable[countNodes].gCtExtensions   = 0;

      for (countDeps = 0; countDeps < MAXREFINEMENTS; countDeps++)
         graphTable[countNodes].gRefinements[countDeps] = -1;

      for (countDeps = 0; countDeps < MAXINCLUSIONS; countDeps++)
         graphTable[countNodes].gInclusions[countDeps] = -1;

      for (countDeps = 0; countDeps < MAXEXTENSIONS; countDeps++)
         graphTable[countNodes].gExtensions[countDeps] = -1;

   } // end init

   return;

} // end InitGraphTable



%cat BuildTable.h

void   BuildTable(ifstream& fin,
                  char expression[],
                  int&  nodeCount,
                  GRAPHTABLE graphTable[])

{

   stack<int>  parentSt;
   stack<char> modeSt;
   string      titleStr;
   int         expressionMarker;
   int         parentTop;     // temp
   int         index;         // temp
   char        garbage;

   nodeCount = 0;
```

```
fin.getline(expression, MAXLINESIZE + 1);

// init table and parent stack with base case
expressionMarker = 0;
ParseBase(expressionMarker, expression, titleStr);



// Add base to the graphTable
graphTable[0].gTitleNum = 0;
graphTable[0].gTitleStr = titleStr;

nodeCount++;

//Put base on parent stack
parentSt.push(0);

while(expression[expressionMarker] != ';') {

   //  process expression char by char
   switch (expression[expressionMarker]) {
      case ' ':
         expressionMarker++;
         break;

      case '<':
         modeSt.push('R');
         expressionMarker++;
         break;

      case '>':
         modeSt.pop();
         expressionMarker++;
         break;

      case '(':
         expressionMarker++;
         break;

      case ')':
         parentSt.pop();
         expressionMarker++;
         break;

      case '[':
         modeSt.push('I');
         expressionMarker++;
         break;

      case ']':
         modeSt.pop();
         expressionMarker++;
         break;

      case '{':
         modeSt.push('E');
         expressionMarker++;
```

```
      break;

   case '}':
      modeSt.pop();
      expressionMarker++;
      break;

   case 'U':

   case 'u':

      ParseTitle(expressionMarker, expression, titleStr);
      graphTable[nodeCount].gTitleNum = nodeCount;
      graphTable[nodeCount].gTitleStr = titleStr;
      parentTop = parentSt.top();
      graphTable[nodeCount].gParent   = parentTop;

      switch (modeSt.top())
      {

         case 'R':
            index = graphTable[parentTop].gCtRefinements;
            graphTable[parentTop].gRefinements[index] = nodeCount;
            graphTable[parentTop].gCtRefinements++;
            break;

         case 'I':
            index = graphTable[parentTop].gCtInclusions;
            graphTable[parentTop].gInclusions[index] = nodeCount;
            graphTable[parentTop].gCtInclusions++;
            break;

         case 'E':
            index = graphTable[parentTop].gCtExtensions;
            graphTable[parentTop].gExtensions[index] = nodeCount;
            graphTable[parentTop].gCtExtensions++;
            break;

         default:
             cout << "**********PROBLEM WITH Mode Stack*************\n";
             break;

      } // end SWITCH mode check

      parentSt.push(nodeCount);
      nodeCount++;
      break;

   default:
      cout << "************ INVALID PARSE ************\n";
      break;


} // END SWITCH  character check char by char
```

```
   } // end parse loop

   return;

} // end BuildTable



%cat ParseUseCase.h
void   ParseBase(int& expressionMarker,
                 const char expression[],
                 string& titleStr)

{
   titleStr = "";

   while(expression[expressionMarker] != '<' &&
         expression[expressionMarker]  != ';') {
      titleStr = titleStr + expression[expressionMarker];
      expressionMarker++;

   } // end read of base case
   return;

} /* end ParseBase */



void   ParseTitle(int& expressionMarker,
                 const char expression[],
                 string&  titleStr)

{

   titleStr = "";
   while(expression[expressionMarker] != ')'  &&
         expression[expressionMarker] != '{'  &&
         expression[expressionMarker] != '['  &&
         expression[expressionMarker] != '<' ) {

     titleStr = titleStr + expression[expressionMarker];
     expressionMarker++;

   } // end parsing of use case title

} // end ParseTitle



%cat PrintTable.h

void  PrintTable(ofstream& fTableOut,
                 string    ofName,
                 const char expression[],
                 int   nodeCount,
```

```
                GRAPHTABLE graphTable[])

{
    //Print Headings

    fTableOut << " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - \n";

    fTableOut << "GRAPH_TABLE EXPRESSION: " << expression << endl;
    fTableOut << "See file: " << ofName << endl << endl;
    fTableOut.setf(ios::left);
    fTableOut << "NODE  "
              << "NODE_NAME   "
              << "Parent ";
    fTableOut << "REFS  "
              << "INCLS "
              << "EXTS ";
    fTableOut << "|REF                    "
              << "|INCLS          "
              << "|EXTS           " << endl;
   for (int count = 0; count < nodeCount; count++)
   {

         fTableOut.precision(1);
         fTableOut.setf(ios::right);
         fTableOut << setw(4) << graphTable[count].gTitleNum << " ";
         fTableOut.setf(ios::left);
         fTableOut << setw(10) << graphTable[count].gTitleStr;
         fTableOut.setf(ios::right);

         if (graphTable[count].gParent == -1)
            fTableOut << setw(7) << "X";
         else

            fTableOut << setw(7) << graphTable[count].gParent;
         fTableOut << setw(6) << graphTable[count].gCtRefinements
                   << setw(6) << graphTable[count].gCtInclusions
                   << setw(6) << graphTable[count].gCtExtensions;

         fTableOut << "  |";

         for (int countdep = 0; countdep < MAXREFINEMENTS; countdep++)
         {
            if (graphTable[count].gRefinements[countdep] == -1)
               fTableOut << "   ";
            else
               fTableOut << setw(3) <<
graphTable[count].gRefinements[countdep];

         }

         fTableOut << "|";
         for (int countdep = 0; countdep < MAXINCLUSIONS; countdep++)
         {
            if (graphTable[count].gInclusions[countdep] == -1)
               fTableOut << "   ";
            else
```

```
               fTableOut << setw(3) <<
graphTable[count].gInclusions[countdep];
         }

         fTableOut << "|";
         for (int countdep = 0; countdep < MAXEXTENSIONS; countdep++)
         {
            if (graphTable[count].gExtensions[countdep] == -1)
               fTableOut << "   ";
           else
                fTableOut << setw(3) <<
graphTable[count].gExtensions[countdep];

         }
         fTableOut << endl << endl;

   } // end print of table contents

} // end PrintTable



%cat bdg.cc

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include <string>
#include <stack>
#include <deque>

using namespace std;

const int MAXREFINEMENTS = 8;
const int MAXINCLUSIONS  = 5;
const int MAXEXTENSIONS  = 5;
const int MAXNODES       = 100;
const int MAXLINESIZE    = 90;
const int MAXCHAR        = 500;



struct GRAPHTABLE {
   bool    visit;
   string  gClusterName;
   string  gNodeName;
   int     gTitleNum;
   string  gTitleStr;
   int     gParent;
   int     gCtRefinements;
   int     gCtInclusions;
   int     gCtExtensions;
   int     gRefinements[MAXREFINEMENTS];
   int     gInclusions[MAXINCLUSIONS];
   int     gExtensions[MAXEXTENSIONS];
```

```
}; // end GRAPHTABLE

//***************** PROTOTYPES ****************

void   BuildTable(ifstream& fin,
                  char expression[],
                  int& nodeCount,
                  GRAPHTABLE graphTable[]) ;

void   ParseBase(int& expressionMarker,
                 const char expression[],
                 string& titleStr) ;

void   InitGraphTable(GRAPHTABLE graphTable[]);
void   ParseTitle(int& expressionMarker,
                  const char expression[],
                  string&  titleStr) ;

void  PrintTable(ofstream& fTableout,
                 string    ofName,
                 const char expression[],
                 int   nodeCount,
                 GRAPHTABLE graphTable[]) ;

void   BuildDot(ofstream& fout,
                const char expression[],
                int nodeCount,
                GRAPHTABLE graphTable[]);

void   InitGraphNames(int numNodes, GRAPHTABLE GraphTable[]) ;

void   InitGraph(ofstream& fout,
                 const char expression[],
                 GRAPHTABLE graphTable[]) ;

void   BuildRefinementStack(int beginNode,
                 GRAPHTABLE graphTable[],
                 stack<int>& refStack) ;

bool FindNext (int topnode, GRAPHTABLE graphTable[], int& nextChild) ;

void   BuildGraph(ofstream& fout,

                  GRAPHTABLE graphTable[],

                  stack<int>& refStack);

void PrintStack(stack<int> refStack);


#include "InitGraphTable.h"
#include "BuildTable.h"
#include "ParseUseCase.h"
#include "PrintTable.h"

int main(){
   ifstream fin;       //file for expressions
```

```
ofstream fTableOut; // file for each graph table
ofstream fout;      //file for .dot files, one for each expression

 // build for .dot file names
string ofNameFirst = "graph";
char   ofCount[MAXCHAR];
string ofNameEnd  = ".dot";
string ofName;

int        numExpressions;
char       garbage[MAXLINESIZE];
char       expression[MAXNODES];
int        nodeCount;

GRAPHTABLE graphTable[MAXNODES];

fin.open("RETRACexp3_header.in");
if (fin.fail()) {
   cout << "******** INPUT FILE OPEN FAILURE **********\n";
   exit(1);
}



fTableOut.open("GraphTable.data");
if (fTableOut.fail() ) {
   cout << "******OUTPUT .data TABLE FILE ERROR ****";
   exit(1);
}

fin >> numExpressions;
fin.getline(garbage, MAXLINESIZE+1);


for(int count=1; count <= numExpressions; count++) {
   sprintf(ofCount, "%d", count);
   ofName = ofNameFirst + ofCount + ofNameEnd;
   fout.open(ofName.c_str());
   if (fout.fail()) {
      cout << "*****OUTPUT .dot FILE ERROR ****";
      exit(1);
   }

   InitGraphTable(graphTable);
   BuildTable(fin, expression, nodeCount, graphTable);
   PrintTable(fTableOut, ofName, expression, nodeCount, graphTable);
   BuildDot(fout, expression, nodeCount, graphTable);
   fout.close();

} // build another .dot file

fTableOut.close();
fin.close();

return 0;
```

```
} // end main


void   BuildDot(ofstream& fout,
                const char expression[],
                int numNodes,
                GRAPHTABLE graphTable[])
{
   stack<int> refStack;
   InitGraphNames(numNodes, graphTable);
   InitGraph(fout, expression, graphTable);
   BuildRefinementStack(0, graphTable, refStack);
   BuildGraph(fout, graphTable, refStack);

   fout << endl << "} // end graph";

} // end BuildDot


void InitGraphNames(int numNodes, GRAPHTABLE graphTable[]) {
   // Build node & cluster names
   string nodePrefix = "node";
   char   nodeNum[MAXCHAR];
   string clusterPrefix = "cluster";

   //------------------------
   for (int count = 0; count < numNodes; count++) {
      sprintf(nodeNum, "%d", graphTable[count].gTitleNum);
      graphTable[count].gNodeName    = nodePrefix    + nodeNum;
      graphTable[count].gClusterName = clusterPrefix + nodeNum;
   } // end naming

} // end InitGraphNames

void InitGraph(ofstream& fout,
               const char expression[],
               GRAPHTABLE graphTable[])
{
   // INIT GRAPH
   fout << "digraph G {"          << endl;
   fout << "compound=true;"       << endl;
   fout << "fontsize = 12;"       << endl;
   fout << "label=" << "\"" << expression << "\"" << endl;
   fout << "ranksep=.5;"          << endl;
   fout << "nodesep=.5;"          << endl;
   fout << "node[fontsize=10];"   << endl;

// end cluster

   fout << endl << endl;

} // end InitGraph


void   BuildRefinementStack(int beginNode,
                            GRAPHTABLE graphTable[],
```

```
                              stack<int>& refStack) {

   int topNode;
   int ctRef;
   int nextChild;
   stack<int> tempStack;

   //init stack
   tempStack.push(beginNode);
   graphTable[beginNode].visit = true;

   while (!tempStack.empty())
   {
       topNode = tempStack.top();
       if (FindNext(topNode, graphTable, nextChild))
       {
          tempStack.push(nextChild);
          topNode = tempStack.top();
          graphTable[topNode].visit = true;
       }

       else { // no more children
          refStack.push(topNode);
          tempStack.pop();
       }

   } // end walk to find refinement paths
} // end BuildRefinementStack

bool FindNext (int topNode, GRAPHTABLE graphTable[], int& nextChild)

{
   bool found = false;
   int count;
   count = graphTable[topNode].gCtRefinements;

   for (;count > 0 && !found; count--) {
      nextChild = graphTable[topNode].gRefinements[count-1];
      if (graphTable[nextChild].visit == false)
         found = true;
   }

   return found;

}// end FindNext



void   BuildGraph(ofstream& fout,
                  GRAPHTABLE graphTable[],
                  stack<int>& refStack) {

   int topNode;
   int workingNode;
   int parent;
   int count;
   stack<int> incStack;
```

```
stack<int> extStack;

topNode = refStack.top();

while(!refStack.empty())
{
   workingNode = refStack.top();
   refStack.pop();

   if (workingNode == 0) {

      //BEGIN INIT NODE CLUSTER
      fout << "subgraph ";
      fout << graphTable[0].gClusterName
           << "{" << endl;
      fout << "color=white;" << endl;
      fout << "node [shape=box];" << endl;
      fout << "label = \"   \"; " << endl;
      fout << graphTable[0].gNodeName;
      fout << "[label="
           << graphTable[0].gTitleStr
           << ","
           << " shape=ellipse];"
           << endl;

      fout << "} // end "
           << graphTable[0].gClusterName
           << endl << endl;

   } // end build of first cluster

   else
   {
      //BEGIN INIT NODE CLUSTER
      fout << "subgraph ";
      fout << graphTable[workingNode].gClusterName;
      fout << "{" << endl;
      fout << "label="
           << "\""
           << graphTable[workingNode].gTitleStr
           << "\""  << ";" << endl;

      fout << graphTable[workingNode].gNodeName;
      fout << "[style=invis, fixedsize=true, "
           << "height=.09, width=.09" << "]"
           << ";" << endl;

      // FOR EACH INCLUSION BUILD SUB-TREES
      count = graphTable[workingNode].gCtInclusions;
      for (; count >0; count--)         {

         int incNode = graphTable[workingNode].gInclusions[count-1];
         BuildRefinementStack(incNode, graphTable, incStack);
         BuildGraph(fout, graphTable, incStack);
      }// END PROCESSING EACH INCLUSION
```

```
         // END CLUSTER
         fout << "} // end cluster"
              << graphTable[workingNode].gClusterName
              << endl << endl;

         // LINK TO PARENT
         if (workingNode != topNode)
         {
            parent = graphTable[workingNode].gParent;

            fout << graphTable[parent].gNodeName;
            fout << "->";
            fout << graphTable[workingNode].gNodeName;
            fout << "["
                 << "tailport=s, headport=n" << ", "
                 << "label=\".....\""        << ", ";

            if (parent != 0) {
            fout << "ltail=" << graphTable[parent].gClusterName << " ,";
            }

            fout << "lhead=" << graphTable[workingNode].gClusterName
                 << "]"
                 << ";" << endl << endl;
         }

         // ATTACH EXTENSION USE CASES
         count = graphTable[workingNode].gCtExtensions;
         for (;count >0; count--) {
            int extNode = graphTable[workingNode].gExtensions[count-1];
            BuildRefinementStack(extNode, graphTable, extStack);
            BuildGraph(fout, graphTable, extStack);

            if (count == graphTable[workingNode].gCtExtensions) {
                 fout << "{rank = same; "
                      << graphTable[workingNode].gNodeName << "; "
                      << graphTable[extNode].gNodeName
                      << "}" << endl ;
            }
            fout << graphTable[workingNode].gNodeName;
            fout << "->";
            fout << graphTable[extNode].gNodeName;
            fout << "["
                 << "tailport=e, headport=w" << ", "
                 << "style=dotted"           << ", ";
            fout << "ltail=" << graphTable[workingNode].gClusterName << " ,";
            fout << "lhead=" << graphTable[extNode].gClusterName
                 << "]"
                 << ";" << endl << endl;
         } // end procession extensions
       } // end Build cluster
    } // end recursive build of clusters
} // end BuildGraph
void PrintStack(stack<int> refStack) {
```

```
      int node;

      cout << "\n---------- Print stack pop/print\n";
      while(!refStack.empty()){
         node = refStack.top();
         refStack.pop();
         cout << node << endl;
      }
}

%exit exit Script done on Mon Jul 30 10:41:59 2007
```

Vita

Coretta Willis Douglas was born in Jackson, Mississippi, and received her bachelor of arts degree in computer science from Mississippi State University. Subsequently she worked as a programmer analyst at Mississippi Chemical Corporation in Yazoo City in implementing an automatic re-order purchasing system and in maintenance of other functional systems. She then moved to Baton Rouge and was employed as the online programmer for Capital Bank maintaining a large overlay system for banking peripherals. She returned to academia completing a Master of Science in Systems Science at Louisiana State University in 1998 and thereafter became an Instructor within the Computer Science Department at Louisiana State University. Her research interests include requirements specification, visual languages, ontologies, and risk management.

She is married to Carl L. (Mel) Douglas, Jr. and is the mother of two daughters and the grandmother of three.