NASA-IVV-97-017

NASA IV&V Facility, Fairmont, West Virginia

An Automated Method for Identifying Inconsistencies within Diagrammatic Software Requirements Specifications

Zhong Zhang

November 10, 1997

This technical report is a product of the National Aeronautics and Space Administration (NASA) Software Program, an agency wide program to promote continual improvement of software engineering within NASA. The goals and strategies of this program are documented in the NASA software strategic plan, July 13, 1995.

Additional information is available from the NASA Software IV&V Facility on the World Wide Web site http://www.ivv.nasa.gov/

This research was funded under cooperative Agreement #NCC 2-979 at the NASA/WVU Software Research Laboratory.

An Automated Method for Identifying Inconsistencies within Diagrammatic Software Requirements Specifications

THESIS

Submitted to the Eberly College of Arts and Sciences
of
West Virginia University

In Partial Fulfillment of the Requirements for
The degree of Master of Science

By
Zhong Zhang
Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, West Virginia 26505

This research was supported by NASA-IV&V Cooperative Research Project agreement,
NASA-IV&V Facility at Fairmont, WV.

Acknowledgements

First and foremost I would like to express my gratitude to my supervisors Dr. Steve Easterbrook and Dr. John Callahan. In particular, I wish to thank Dr. Callahan for his granting me a chance to join Software Research Laboratory (SRL) at NASA/WVU. And I wish to thank Dr. Easterbrook for his guidance, patience, and most important for his giving me freedom to pick up the subject that I am interested in. Dr. Easterbrook and Dr. Callahan always refer some good papers and books to me, and provide me chances to attend conferences. Their continuous advice and guidance make this thesis possible. I am indebted to them.

It is one of the happiest periods in my life during staying with SRL. I enjoyed two years peaceful, fruitful, fun, exciting time in Morgantown. My thanks to Dr. Swarn Dhaliwal, my teammate, for his sharing Java knowledge during the TCM translation process. Many thanks go to Ms. Reshma Khatsuriya, my officemate, and Mr. Mani Kancherla for their kind help and valuable discussion. I thank Mr. Todd Montgomery and Mr. Nicholay Gradetsky for their helps and suggestions in solving problems in my research. Many thanks to Dr. Wu Wen, Dr. Frank Schneider, Dr. Ralph "Butch" Neal, and Mr. Edward Addy for their interest in my various presentations and their useful discussions and insights. Also, I would like to thank Ms. Vivian Jenab for her help whenever I needed.

Many thanks go to the faculty members of CSEE for their instruction and providing friendly environments to grow my knowledge. Especially, I want to thank Dr. Frances VanScoy, my graduate study committee member, for her comments on drafts of my thesis and valuable advice. Also, my thanks go to Dr. Frank Dehne at Vrije Universiteit at Amsterdam, the Netherlands, for his generous providing C++ source code of TCM and his comments on TCMJAVA.

Many thanks to my roommates and friends, Mr. Daming Li, Mr. Jesse Wang, Ms. Haijing Wang and Ms. Lei Xu for their helps and friendly companionship. My thanks to the friends in CSEE and the friends residing in the building on Willey Street for their helps.

Finally, my gratitude goes to my parents for their love and consistent support. I am very grateful to my girl friend Wei, for her continuous encouragement and support.

To my parents and wei

Abstract

The development of large-scale, composite software in a geographically distributed environment is an evolutionary process. Often, in such evolving systems, striving for consistency is complicated by many factors, because development participants have various locations, skills, responsibilities, roles, opinions, languages, terminology and different degrees of abstraction they employ. This naturally leads to many partial specifications or viewpoints. These multiple views on the system being developed usually overlap. From another aspect, these multiple views give rise to the potential for inconsistency. Existing CASE tools do not efficiently manage inconsistencies in distributed development environment for a large-scale project. Based on the ViewPoints framework the WHERE (Web-Based Hypertext Environment for requirements Evolution) toolkit aims to tackle inconsistency management issues within geographically distributed software development projects. Consequently, WHERE project helps make more robust software and support software assurance process.

The long term goal of WHERE tools aims to the inconsistency analysis and management in requirements specifications. A framework based on Graph Grammar theory and TCMJAVA toolkit is proposed to detect inconsistencies among viewpoints. This systematic approach uses three basic operations (UNION, DIFFERENCE, INTERSECTION) to study the static behaviors of graphic and tabular notations. From these operations, subgraphs Query, Selection, Merge, Replacement operations can be derived. This approach uses graph PRODUCTIONS (rewriting rules) to study the dynamic transformations of graphs. We discuss the feasibility of implementation these operations. Also, We present the process of porting original TCM (Toolkit for Conceptual Modeling) project from C++ to Java programming language in this thesis. A scenario based on NASA International Space Station Specification is discussed to show the applicability of our approach. Finally, conclusion and future work about inconsistency management issues in WHERE project will be summarized.

Chapter One Introduction

The WHERE (Web-Based Hypertext Environment for Requirements Evolution) project is concerned with the communication and coordination problems faced on large, geographically distributed projects. It aims to support the process of collaborative development of requirements specifications with tools to manage incremental changes to large specifications [1]. The WHERE project is based on the ViewPoints framework. It strives for: annotating and tracing the relationships between different pieces of requirements specifications; managing inconsistency in requirements specifications in distributed environments; and supporting group collaboration via the WWW. This thesis is one part of an effort to tackle "inconsistency management" issues in the WHERE project. In this thesis the motivation behind the WHERE project is described. Also, a framework based on graph grammar theory for inconsistency management is delineated, and a port of an implementation based on this framework is described. Finally, a case study of the valid space station modes and state transitions requirements from the NASA space station specification is presented with application of our inconsistency-checking engine.

The long-term goal of the WHERE project is to tackle the "inconsistency management" problems. Large specifications are likely inconsistent for most of their lifetimes, because the development of large-scale composite software in a geographically distributed environment is an evolutionary process. Often, in such evolving systems, striving for consistency is complicated by many factors, because development participants have various locations, skills, responsibilities, roles, opinions, languages, terminology and different degrees of abstraction they deploy. This naturally leads to many partial specifications or viewpoints, which are usually overlapped. From another aspect, these multiple views give rise to the potential for inconsistency. The problem of inconsistency management is to detect inconsistencies, to keep track of them, and to manage them, rather than to prevent or avoid inconsistencies [2]. None of existing CASE tools could efficiently manage inconsistencies in a distributed development environment. The delivery of the WHERE project greatly helps information acquisition, group communication and error finding in the software development process. Also, the performance of inconsistency management supports innovative thinking and exploration of alternatives. Consequently, the WHERE project helps make more robust software and support software assurance process.

The detection of inconsistency needs an editing CASE tool to handle different notations encountered in requirements specifications. An editing toolkit called TCMJava has been developed in this group in order to meet the need of WHERE project. The exploring of inconsistency issues in this thesis is based on TCMJava toolkit. In the course of investigation of different approaches for inconsistency detection and management, we find that Quasi-classical logic [3] is difficult to implement based on the TCMJAVA implementation, while the application of graph grammars [4] in implementations is feasible. In this thesis a framework based on graph grammars for inconsistency detection is established.

In the complex system development process, requirements specifications usually are expressed by various graphical notations. The internal representation of most software documents is a directed attributed graph. Sometimes even the same piece of a specification document can be represented using different graphical notations. Our approach focuses on graphical requirements specifications. The encoding of a piece of diagrammatic or tabular specification into a graph system is explicit and direct. After the conversion of specifications from graphical notations to graph systems, a user defines mapping functions to detect if there exists a graph homomorphism between the two graph systems. The finding of homomorphism helps detection and analysis of inconsistencies. The subgraph queries and mappings make use of the basic operators INTERSECTION, UNION, DIFFERENCE, and SELECTION. In our implementation, method users define mapping functions via user friendly GUIs.

In this thesis, the application of graph transformation in inconsistency management is also investigated. Graph rewriting rules store the information of a specification evolving process. The recording of all graph rewriting rules helps backtracking to a previous state in development process, supports the exploration of alternatives, and supplies valuable help in searching for an inconsistency resolution. As a data type, rules are stored and manipulated in our approach. In the searching for and replacing of subgraphs for the writing rules, the established static operations are used for queries and replacements.

In order to support group coordination and communication work we adopt the networking programming language Java as the implementation language and CORBA as the information access and communication infrastructure. Before experimenting with our approach, we need a set of diagrammatic and tabular notation editing tools to build graphical specifications. We translated a graphical conceptual modeling toolkit called the Toolkit for Conceptual Modeling (TCM) [5] from C++ into the Java programming language. The difficulties encountered in the translation process are described as a part of the thesis work, and the graphical shape data structure is discussed in detail in order to map a shape class in TCM to a graph system in graph grammar theory.

We experiment with our approach by applying our TCMJAVA and inconsistency-checking engine on the valid space station modes and state transitions requirements from NASA space station specification as a part of the Independent Verification and Validation (IV&V) process [6]. First, in order to help readers to understand this example, the IV&V process is briefly outlined. The piece of NASA space station specifications in this case study is represented using a state transition diagram and a state transition table in the original document. We described how to use our inconsistency-checking engine in the detection of inconsistencies between those graphical notations.

We conclude that the implementation TCMJAVA fits well a graph grammar system. Our approach is intuitive and feasible. Overlap and inconsistency are inevitable and acceptable in distributed composite system development environment. We have to learn to live with them and learn from them. Our framework is the first implementation of graph grammars on inconsistency management for requirements specification. From empirical study, our framework helps solve the inconsistency management problems.

Chapter Two The Necessity for Inconsistency Management

This chapter describes the problems the WHERE project face. First, from a broad perspective the goals of the WHERE are described. Later, we focus on inconsistency management issues in this thesis. The motivation of pursuing inconsistency management in requirements specifications is delineated. Before describing what is inconsistency management, we introduce the concepts of overlap and inconsistency in section 2.2. At first glance, it is difficult to see the need for inconsistency management. Therefore, in section 2.3, a scenario [7] is described to elaborate how inconsistencies are generated and how different inconsistency handling actions are used to mange inconsistencies [2]. In section 2.4, we summary the problems with inconsistency management, and enforce the argument that the principal mechanisms used for preventing or avoiding overlap and inconsistency are impractical, inconsistency (and perhaps overlap) is managed rather than simply prevented could lead to more robust specification.

2.1 The WHERE Project Overview.

The development of large-scale composite software system is an evolutionary process. The WHERE project concerns the group coordination and communication problems in such a system [1]. Basically three goals are pursued in the WHERE project: the requirements traceability and annotation to support group coordination in evolving development processes, information access and communication among group collaboration work, and inconsistency management in an evolving development process. We describe each of these goals first, and later focus on the inconsistency management issues in this chapter. We start with the first goal of the WHERE.

In a large-scale project development, managing an evolving requirements specification is a significant problem, because any small changes to one part of the specification may have impacts throughout the whole system specification documentation. These impacts are often difficult to track and reason. Therefore, it is difficult to know that all implications resulted from a small change. Based on the ViewPoints framework, the WHERE project aims to solve these problems with the support of annotation methods [8]. Dr. Al-Rawas discussed how multiple-perspectives of software requirements are annotated and what the communication channels in multiple-perspectives environments are in his Ph .D. dissertation. We will not discuss these issues in this thesis.

World Wide Web (WWW) has emerged in recent years as an effective way to provide access to information in a uniform and standardized manner. One of the cornerstones for setting up a collaborative infrastructure is providing the ability for any member of a geographically distributed team to access information. Therefore, the WWW is the ideal candidate for group collaboration. The WHERE is implemented using the networking programming language Java, and the communication and information access infrastructure is constructed on CORBA (common object request broker architecture). The support of group collaboration via WWW from the WHERE project is discussed in reference [9].

The development of a composite system is an evolving process. In such a system striving for consistency is complicated by many factors. Because development participants have various skills, responsibilities, roles, opinions, languages, terminology and different degrees of abstraction they deploy, this naturally leads to many partial specifications or ViewPoints. These multiple views on the system being developed are usually overlapped, by which we mean they refer to common aspects of the system domain under development. From another aspect, these multiple views give rise to the potential for inconsistency [2]. ViewPoints are inconsistent when they assert properties of these aspects that conflict. Therefore, as a consequence of multiple perspectives, overlap is inevitable; inconsistency is acceptable, because inconsistency supports innovative thinking, deferment of commitments, and exploration of alternatives [2][10]. In any case the principal mechanisms used for preventing or avoiding overlap and inconsistency are impractical, and are not cost-effective. For "global" repositories and "universal" schemas, they are difficult to design, difficult to extend, lead to centralized architectures, and are not scaleable [11]. The consequence is that inconsistency (and perhaps overlap) is "managed" rather than simply prevented or avoided. The idea of "managing inconsistency" helps information acquisition and error finding [2][10]. The goal of using the ViewPoints framework is to pursue partial consistency, and inconsistency management, even in "the final product" and even in the development of "safety critical systems". Before describing the inconsistency management, we first introduce what is inconsistency.

2.2 What is Inconsistency?

Figure 1 shows the "interference" relationships between overlap, consistency rules, and inconsistency [12]. Overlap is self-explained, which means ontological overlap. We employ the following definition for inconsistency [13].

"An inconsistency is any situation in which two sets of objects do not obey some relationship that should hold between them."

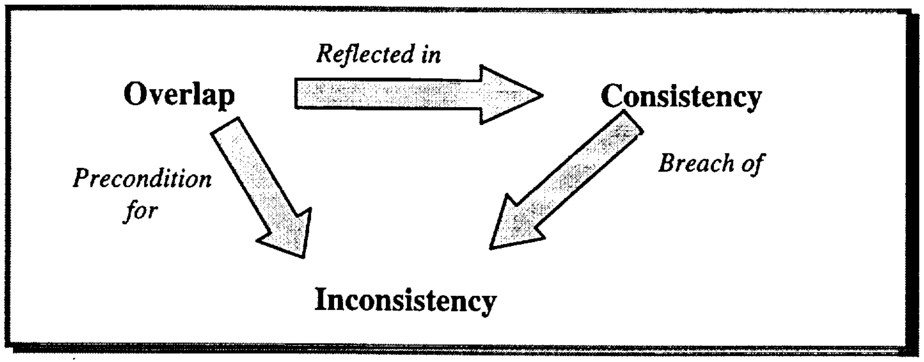


Figure 1. Overlap with Inconsistency.

The objects may be any of the artifacts generated during the system development process, including specifications, models, code, test cases, user documents, process definitions, development plans, process and product metrics, and so on.

The relationships may refer to both syntactic and semantic properties of the objects. Of course, we can not detect inconsistencies unless the relevant relationships have been identified. Therefore, the quality of any inconsistency analysis exercise depends on how well the consistency relationships have been defined. These relationships are expressed as consistency rules, and they are discussed in the next section.

We described the goals of the WHERE project and the concepts of overlap and inconsistency. This thesis focuses on the inconsistency management issues in the WHERE project. In following section we present a scenario to explain how conflicts or inconsistencies happen and what the strategies to handle inconsistencies are. This scenario describes a hierarchically-related development process [14], in which tasks are delegated to separate members of a team to develop.

2.3 An Inconsistency Scenario

This section skips the details about how to define rules, and the notations about rules. The example described below gives an overview about inconsistency management issues in the ViewPoints framework. This example attempts to summary all of the issues regarding inconsistency management, not just a rephrasing of paper [14]. Data flow diagrams (DFDs) is used as the example notation. A node in a graph represents a process, and may be decomposed in a separate diagram. When this happens, the set of inputs and outputs to that process should be shown on the decomposition.

As described in Figure 2, Anne has created a ViewPoint for a top-level data flow diagram of the system. Two processes have been created, process A and process B. Process B is selected for decomposition, and its state is changed to a non-primitive type. Then the job is delivered to Bob, as represented by Activation Policy in Figure 2.

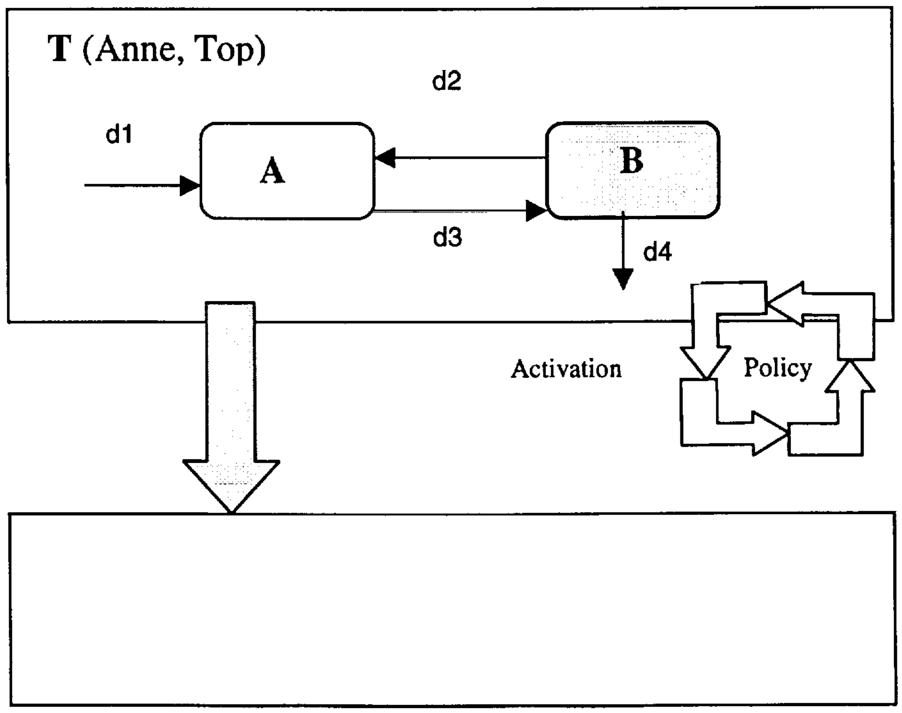


Figure 2. Creation of a New Viewpoint

2.3.1 Force

In Figure 3, Bob creates a new ViewPoint to represent the decomposition. He gives the new ViewPoint a suitable name (B) to indicate it is a decomposition of process B. Bob takes the input flow (d3) and output flows (d2, d4) connected to process A in the parent, and then adds them to the decomposition as its context. His definition of the processes that comprise the decomposition has to be restricted to have the same input and output flows as in the parent regardless of the flows between the sub-processes generated.

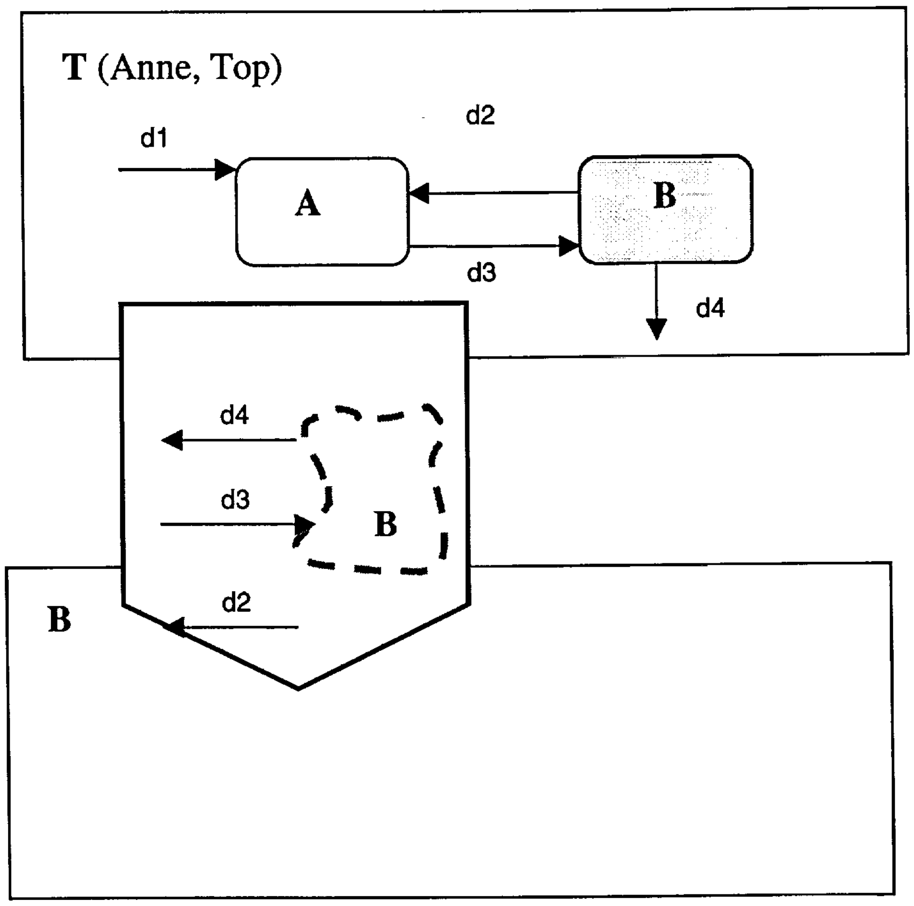


Figure 3. Constraints in Creation of New ViewPoints

2.3.2 A Consistent State:

In Figure 4, we assume that at a certain stage of the development, the specification, which consists of the data flow diagrams, reaches a consistent state. Diagram B is a decomposition of node B in diagram T, and it has the same input and output flows as the node B in its parent diagram.

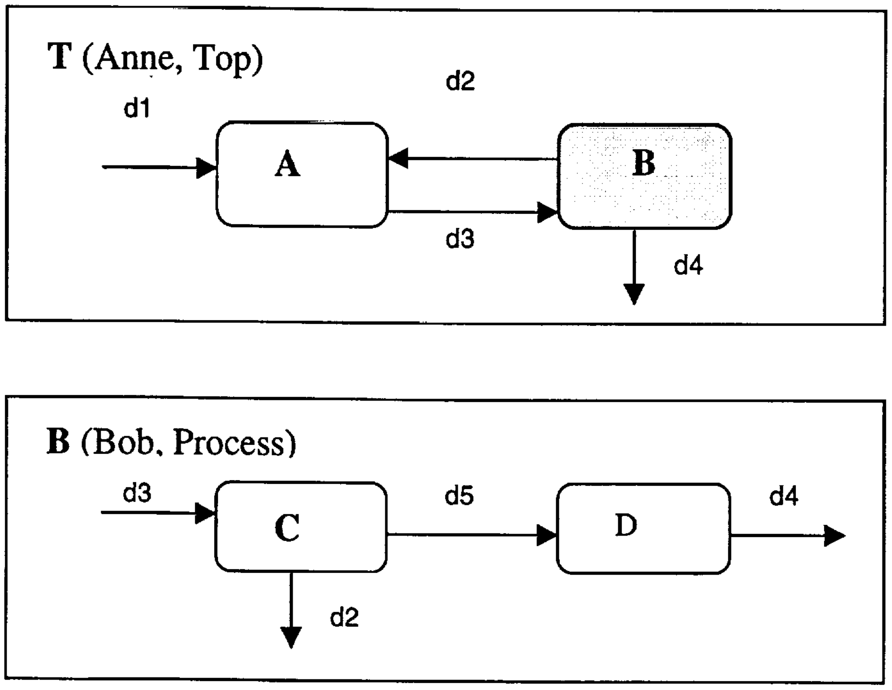


Figure 4. ViewPoint T is Consistent with the Decomposition.

2.3.3 Concurrent Editing

In Figure 5, Anne now does some more work on the parent. In the process of delegating decomposition of other processes, she realizes that one of the outputs of process B is missing, and so she adds it (d7). Meanwhile, Bob has noticed the omission, and adds it too, but using a different label (d6). An inconsistency is generated because of using different label names (d7 vs. d6). In following sections, we describe the strategies for inconsistency detection and management in a high level of view.

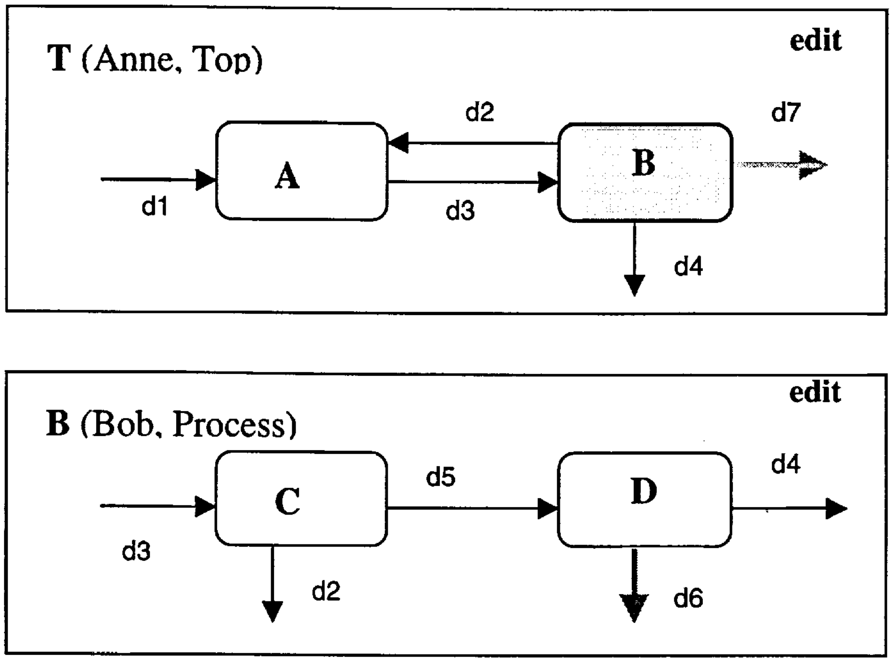


Figure 5. The Situation of Current Editing.

2.3.4 Establishing Overlaps

The requirement for inconsistency management varies. In certain cases it might be loose, that is simply identifying ontological overlaps and inconsistency, in other cases it might be tight, involving the integration of the specification and the resolution of their inconsistencies. Sometimes a meta-method is needed which lies between loose and tight inconsistency management. We will discuss them separately; in any case the detection of ontological overlaps is a prerequisite for detecting inconsistencies.

In Figure 6, inconsistencies have arisen by using different name labels. Overlaps have been identified by underlining the labels, process B and data flows d2, d3, d4, d6, d7 are labeled to indicate that they are the overlap parts in two DFD diagrams. Different approaches can be employed to detect overlaps. We will discuss these issues in following sections.

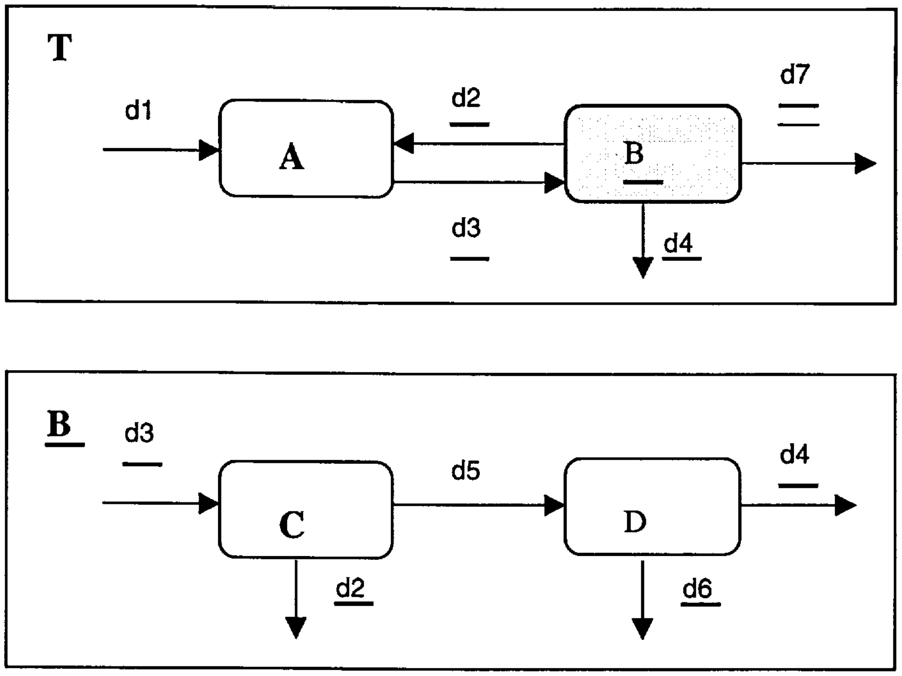


Figure 6. Establishing Overlaps

2.3.5 Building Consistency Rules

Overlaps have been identified; consistency rules can be defined to check the consistencies between the development objects. The relationships between development objects should be completely understood before rules are derived. In our approach, we are looking for a method that does not require a lot of effort to encode a specification in formal logic. Rules should be readable and easy to define, and method users should have the privilege to enforce and remove rules. Also, the method should be automated at some points. We will look at two promising approaches, Quasi-classical and graph grammars in more detail in next section.

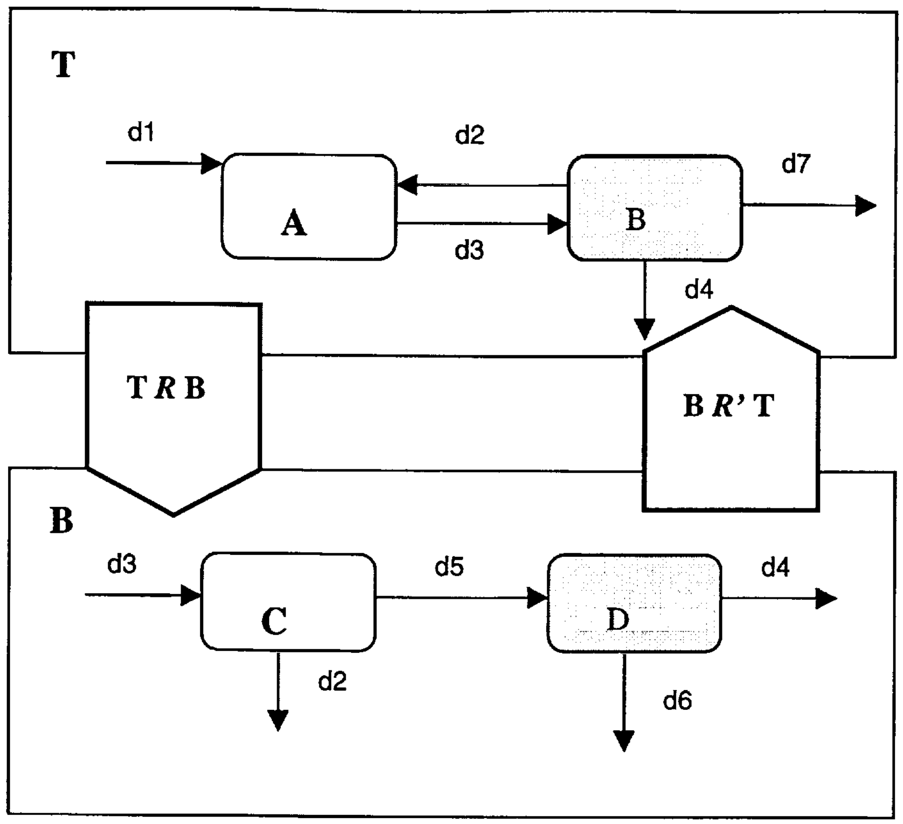


Figure 7. Building Consistency Rules.

2.3.6 Different Strategies for Inconsistency Management.

Different strategies can be employed to manage inconsistencies [10]. We will talk about them respectively in the following sections. Apply analysis and diagnosis of inconsistencies, then solve them locally; tolerate inconsistencies, ignore inconsistencies or circumvent them; trace inconsistencies, postpone resolving inconsistencies; ameliorate and rectify inconsistencies for increasing possibilities of future resolution. The question of which inconsistencies should be resolved at any point is an engineering question involving an assessment of resource availability and risk. Figure 8 and Figure 9 illustrate these different approaches for manage inconsistencies.

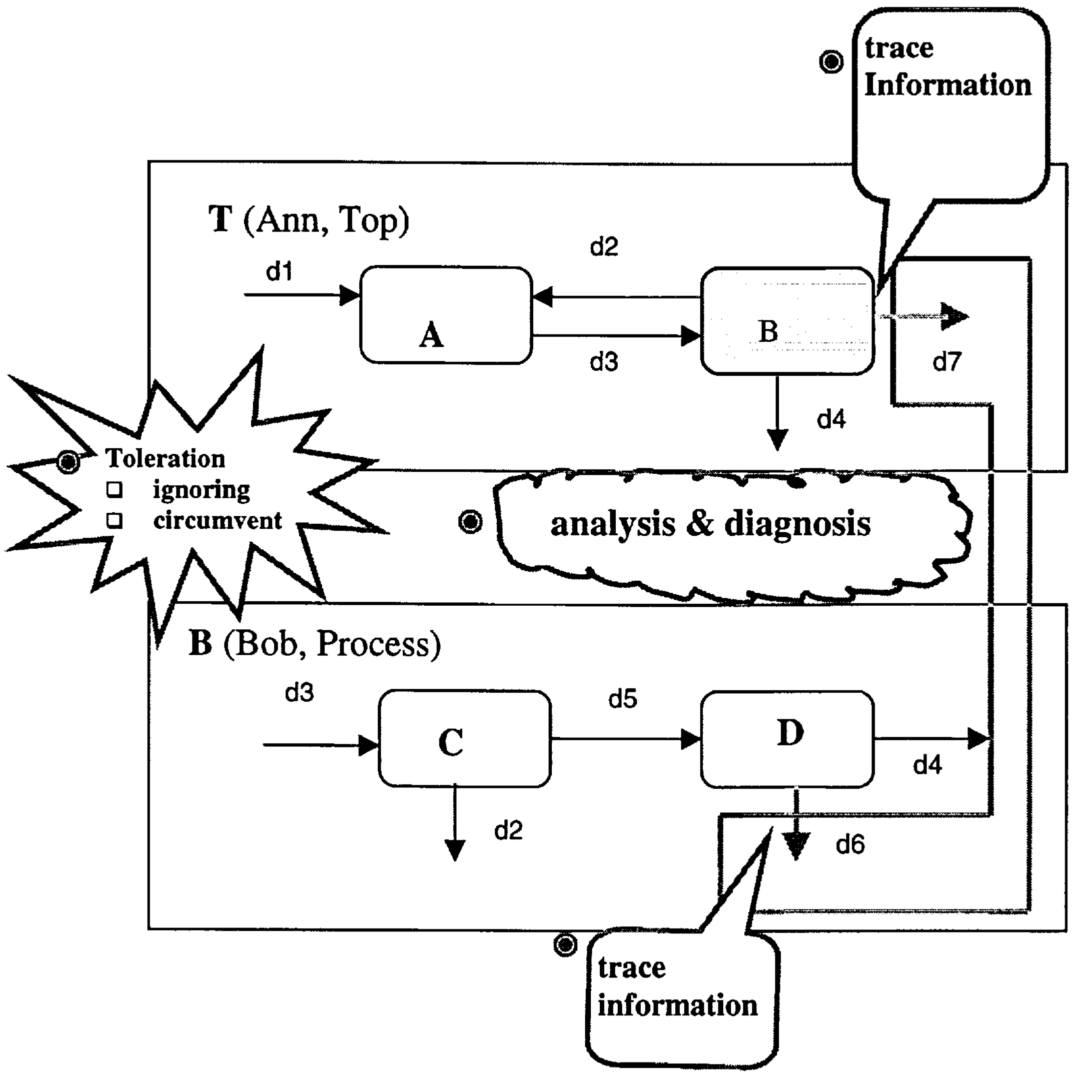


Figure 8. Different strategies in inconsistency management.

Analysis and diagnosis usually is the pre-requirement of inconsistency management. In this step, the sources resulted into inconsistencies are reasoned, and the priorities of inconsistencies are evaluated.

One of the strategies for inconsistency toleration is ignoring inconsistencies completely and continuing development regardless. This may be appropriate in certain circumstances where the inconsistency is isolated and does not affect further development, or prevent it from taking place.

Circumventing the inconsistency and continuing development is the other of the strategies for inconsistency toleration. This may be achieved by marking inconsistent portions of the system or by continuing development in certain directions depending on the kind of inconsistency identified. It is appropriate in situations where it is desirable for the inconsistency to be avoided and/or its removal deferred.

Resolving the inconsistency altogether by correcting any mistakes or resolving conflicts is another method of inconsistency management. This depends on a clear identification of the inconsistency and assumes that the actions required to fix it are known. Achieving consistency completely may be difficult, and is quite often impossible to automate completely without human intervention.

Ameliorating and rectifying inconsistencies is an approach for inconsistency management. This approach performs actions that improve and increase the possibility of future resolution (this may involve ignoring or circumventing inconsistencies). This approach is attractive in situations where complete and immediate resolution is not possible (perhaps because further information is required from another development participant), but where some steps can be taken to fix part or some of the inconsistent information. This approach however requires techniques for analysis and reasoning in the presence of inconsistency.

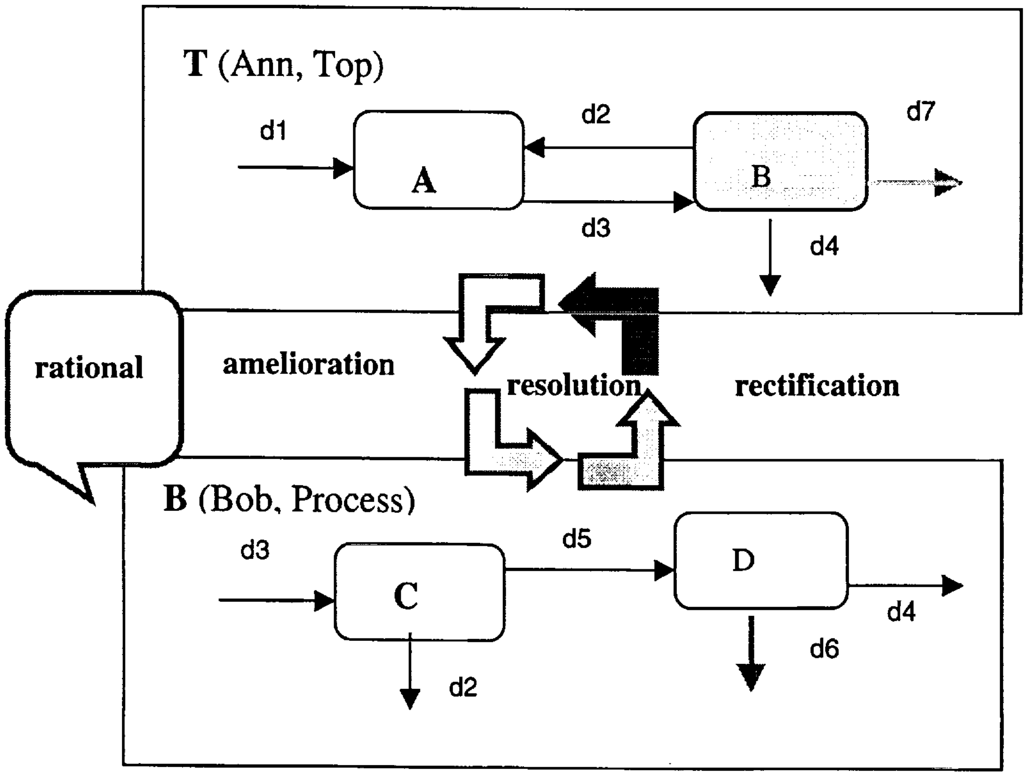


Figure 9. Rational and Resolution.

2.4 Summary of Inconsistency Management

In this chapter we describe the problems we face. In a large-scale composite software development process, participants have various locations, skills, responsibilities, roles, opinions, and languages. This naturally leads to many partial specifications or ViewPoints. Communication and coordination between participants are big problems. This thesis explores inconsistency management issues in such a system. Multiple views from participants are likely overlapped, and also give rise to the potential for inconsistency. We argue that the principal mechanisms used for preventing or avoiding overlap or inconsistency are impractical. Overlap and inconsistency are inevitable and acceptable. We have to learn to live with it and learn from it. Also, we describe the inconsistency handling actions. First, overlaps have to be identified. A clear understanding of relationships between objects helps building consistency rules. After inconsistencies are analyzed and diagnosed, different actions are adopted to manipulate inconsistencies; toleration, tracking, resolution, enforcement, rational provision.

The management of inconsistency needs the ViewPoints framework to trace an evolving development process, to support the communication of information, and to annotate multiple partial specifications. The detection and analysis of inconsistencies need a theory foundation based on a rules-based language or logic theorem. In the next chapter we discuss various related work, ViewPoints framework, Quasi-classical logic, Graph Grammar, WHERE project implementation language Java, Various CASE tools, and various approaches for inconsistency management.

Chapter Three   Literature Review

This chapter contains various topics, all of which are the background introduction and theoretical foundation of the WHERE project. The WHERE project is concerned with large systems that display a complex structure and with many interlocking constraints on their construction and behavior [1]. Working with such systems necessitates multiple ViewPoints for complexity. Therefore, the WHERE project is based on ViewPoints framework. During the analysis and design of such systems, the multiple ViewPoints naturally arise out of differences of opinion, varying goals and mistakes or errors. Working on ViewPoints has commonly been linked to work on requirements engineering. The problem is especially acute during the early stages of development (particularly elicitation where many diverse client views are prevalent). In section 3.1 we give an overview of the existing CASE tools, and further explain that a tool like WHERE is highly desirable. In section 3.2 we describe what is the ViewPoints framework, and how it works for inconsistency management. In section 3.3, we briefly introduce existing models for inconsistency management. In section 3.4 and section 3.5, we discuss the two different approaches for inconsistency detection and reasoning: Quasi-classical logic and graph grammar. They are the foundations for our approach introduced in next two chapters.

In next section we discuss some existing Computer-Aided Software Engineering (CASE) tools, and outline their lack of support for the requirements engineering processes.

3.1 Case Tools

Two kind of CASE tools are discussed in this section, requirements traceability tools and visualization editing CASE tools. We point out the deficiencies of these existing tools, and enforce the argument of the need for the creation a tool such as WHERE.

Requirements traceability tools DOORS [15] and CORE [16] help to some extent by recording links between requirements at different levels, and between requirements and test cases, design objects, and so on. However, existing traceability tools only record links, they do not record any other information about the relationship expressed by the link. Such tools encode a simple process model based on flow down of requirements through different levels. They do not capture any knowledge about the methods and notations being used, and hence do not provide any active support for the development and evolution of specifications. Also, CASE tools for design and analysis requirements are short of the relationship tracking between different pieces of requirements specifications. Because there are a lot of similarities between Rational Rose [17] and the WHERE editing toolkit, e.g. both support the visualization of graphical editing, we describe this tool in more detail.

Rational Rose is a Computer Aided Software Engineering (CASE) tool that supports the design and implementation of object oriented systems by supporting the Object Modeling Technique (OMT) and Booch methods. The Booch method is documented in "Object-Oriented Analysis And Design With Applications" [18], was developed by Grady Booch, a Rational Chief scientist. The OMT methodology, documented in "Object Modeling And Design" [19] was developed by James Rumbaugh, also a Rational fellow. For both, Rational Rose supports multiple views of the underlying model, including object and class diagrams and scenario diagrams for object-oriented development.

The main purpose of Rational Rose is to aid in the design and implementation stages of a piece of software. In broad terms, Rational Rose achieves this by enabling the user to construct a representation of the system being developed using one of the two supported methodologies and then generating the outline for an implementation (in C++) based on the design. Rational Rose allows a user to analyze, design and implement systems in a way that makes them easy to visualize and communicate.

Rational Rose provides a method-based environment where the software development method may be either Booch or OMT. As such, Rational Rose supports these methods by provides a collection of representation notations (such as the module and process diagrams) and a strategy/heuristics for using them. As with many CASE tools, Rational Rose goes about translating the method to the users desktop by providing a number of graphical editors – a collection of editors which facilitate drawing and object annotation (i.e. relation specification). But, Rational Rose does not support group coordination development process, and it does not have inter-notations consistency checking functions.

There are also some freeware or commercial CASE tools for graphical editing. For example, the MetaEdit [20] Toolkit. But, none of these tools supports consistency checking between notations or methods. The WHERE project aims to realize these features. We will explore the inter-notations consistency checking issues in next two Chapters. After the survey of existing CASE tools, we conclude:

- ❑ The existing requirements traceability tools do not capture all of the information in an evolving development process.
- ❑ None of the existing CASE tools supports the consistency and completeness checking for a same piece of specification in different notations.

The ViewPoints framework [21] may give us the solution for these problems. In the next section, we introduce the ViewPoints framework, and show that the ViewPoints framework supports requirements traceability and inconsistency management.

3.2 ViewPoints Framework.

The WHERE project builds upon the ViewPoints framework. An outline description for a ViewPoint can be considered as a set of chunks of a specification [21]. Each chunk has a specific owner and representational style. The idea is that each ViewPoint implicitly is a self-contained specification-editing tool for one particular notation, and is complete with knowledge about how the contents of the ViewPoint are related to other ViewPoints. The editing tool on its own can be thought of as a ViewPoint template. Because a ViewPoint template can be defined for any notation, the framework is independent of any particular method. In fact, different methods can be implemented in the framework by constructing an appropriate set of templates. Most importantly, the framework supports inconsistency management. Each ViewPoint template includes a set of rules for checking consistency with other ViewPoints, together with guidance for resolving inconsistencies. As responsibility for consistency is devolved to the individual ViewPoints, local decisions can be made about whether to ignore or tolerate various inconsistencies.

3.2.1 What is a ViewPoint?

There are various definitions for ViewPoint. In this thesis the definition by Finkelstein in his The ViewPoints FAQ is used [21].

"The combination of the agent and the view that the agent holds is termed a ViewPoint."

Here, agent means participant or actor involved in the construction of a complex description or model. Agents hold different perspectives or views of the artifact or system they are trying to describe or model because of different responsibilities or roles assigned to the agents. A ViewPoint can be thought of as a combination of the idea of an actor, knowledge source, role or agent in the development process, and the idea of a view or perspective that an actor maintains. In software terms, ViewPoints are loosely coupled, locally managed, coarse-grained objects which encapsulate partial knowledge about the system and domain; ViewPoints are specified in a particular, suitable representation scheme, and encode partial knowledge of the process of development. The study of ViewPoints includes the investigation of relations between views, between views and agents, and between agents. We will focus on the relations between views in Chapter Three. As Figure 3 shows, each ViewPoint consists of five slots according to the framework of ViewPoints [2][11]:

- the <u>style</u> slot, the scheme and notation of the ViewPoint representation is described,
- the <u>work plan</u> slot, the set of development actions, process and strategy of the ViewPoint are included,
- the <u>domain</u> slot, the area of concern of the ViewPoint with respect to the overall system under development is defined,
- the <u>specification</u> slot, which specifies the ViewPoint domain, and is developed using the strategy described in the work plan slot,
- the <u>work record</u> slot, which contains an annotated history of actions performed on the ViewPoint. It is the vehicle by which traceability may be achieved, and some form of development rationale may be recorded.

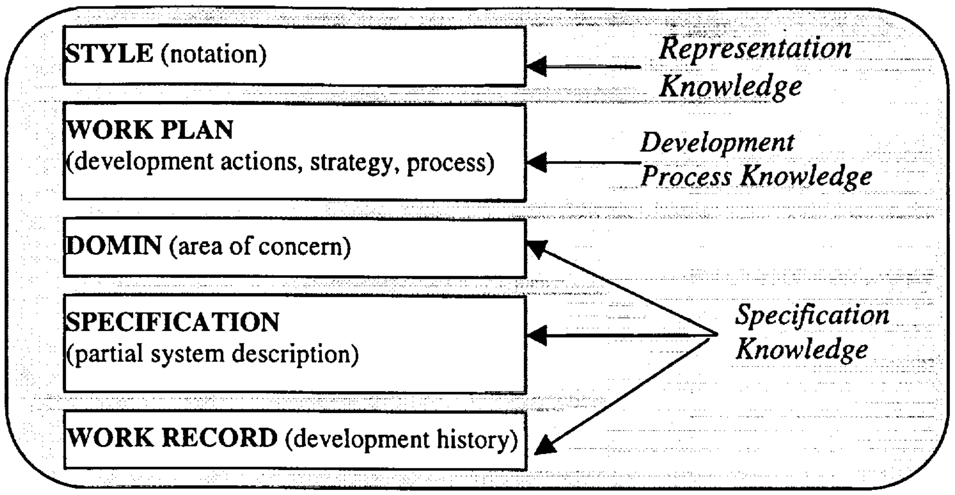

STYLE (notation)
WORK PLAN (development actions, strategy, process)
DOMIN (area of concern)
SPECIFICATION (partial system description)
WORK RECORD (development history)
Representation Knowledge
Development Process Knowledge
Specification Knowledge

Figure 1. The five slots of a ViewPoint.

The development participant associated with any particular ViewPoint is known as the ViewPoint owner. The owner is responsible for developing a ViewPoint specification using the notation defined in the style slot, following the strategy defined by the work plan, for a particular problem domain. A development history is maintained in the work record.

After introducing the concept of a ViewPoint, we will briefly discuss why the ViewPoints framework supports requirements traceability and inconsistency management. Examples will be given in Chapter five of the thesis.

3.2.2 ViewPoints Framework Supports Requirements Engineering.

Conventional system development methods do not recognize the existence of ViewPoints as user-definable entities. Instead they provide rigid structuring schemes and strict control on both the diversity of ViewPoints and the relations between them. By contrast, ViewPoint-oriented methods make ViewPoints first class objects so they can be defined by method users, the relationship between them can be established. The objective of ViewPoint-oriented methods is to strike a balance between preservation of multiple perspectives during system development, and the demands for consistency and coherence arising out of group work. As ViewPoints allow you to collect system information from multiple perspectives, you will inevitably collect a lot of information that must be managed. This is both a strength and a weakness of the approach. It is a strength because you are much less likely to miss information that is critical to the success of the system; it is a weakness because somehow you have to manage this information and eliminate redundancy from it.

The essential problem that ViewPoints present is of consistency or coherence [10]. Given that ViewPoints may overlap if the agents have a shared goal or their goals are potentially interfering, the consistency of their ViewPoints must be established. This consistency need only be partial, i.e. sufficient to achieve the goals. Consistency can be achieved by integrating the ViewPoints, or by locally resolving inconsistencies as they arise.

The ViewPoint framework tolerates inconsistency, with no requirement for changes to one ViewPoint to be consistent with other ViewPoints Finkelstein, et al., [2][10][11]. Consistency checking is performed through a set of inter-ViewPoint rules, defined by the method, which express the relationships that should hold between particular ViewPoints. These rules define partial consistency relations between the different representation schemes. This allows consistency to be checked incrementally between ViewPoints at particular stages rather than being enforced as a matter of course. A protocol is provided for applying consistency checks between ViewPoints, with the checking process being initiated by either ViewPoints owner. A fine-grained decentralized process driven model in each ViewPoint provides guidance for the resolution of inconsistencies as shown in Figure 2 [22].

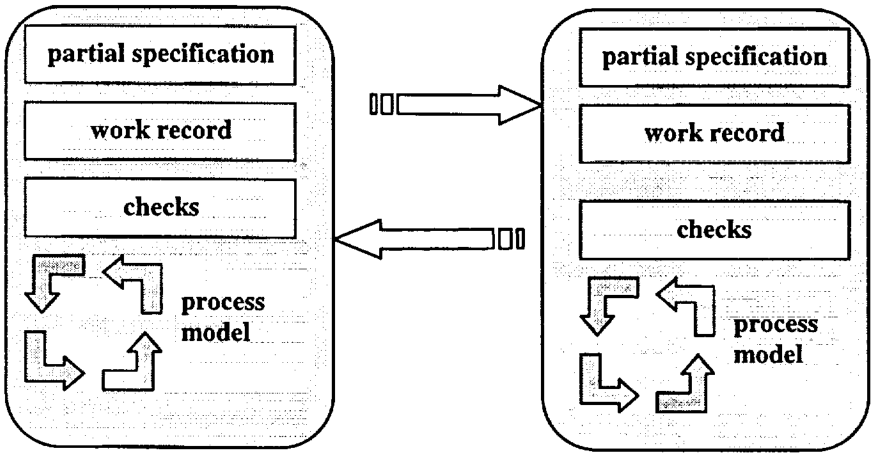


Figure 2. Decentralized Process driven Consistency Checking.

One important factor in the success of inconsistency management is consistency rule defining. We try to granulate consistency rules. They are shown in next section.

3.2.3 The Granularity of Consistency Rule.

Basically we can categorize consistency from three different levels. From the aspect of ViewPoint we can say there are two kinds of consistency checking [14]

- In-ViewPoint checking is concerned that a ViewPoint specification is locally (syntactically) consistent. Such syntactic checks partially define the semantics of a ViewPoint's representation. The work on SCR [23] is an example of such an activity.
- Inter-ViewPoint checking is concerned with the consistency between overlapping or interacting specifications residing in different ViewPoints. The relationships between such ViewPoints are described by inter-ViewPoint rules. In some software design methods, for example, Booch method, every method uses several notations such as Entity Relationship Diagram, Class Diagram and so on. Every notation can be considered as one ViewPoint, and the relationships between two different notations can be expressed as inter-ViewPoint rules. Also, a piece of specification can be expressed by different methodologies from different aspects, for example, the requirements for Fault Detection Isolation and Recovery on the space station can be tested by SCR method or Spin method [23]. These two different methods can be considered as two individual ViewPoints. When those two methods are used, the specification has to be translated into method dependent logical objects and relationships. This translation process is non-trivial, and failure of a proof does not necessarily help to precisely characterize the specification. If we could define some relationships to check the consistency between two different methods, this will increase the credence of the translation process. The relationships between two different methods can be thought of as inter-

ViewPoint rules. Usually these relationships between different methods are very difficult to reason about.

- Handling global consistency is a problem in a fully distributed environment, in which there is no central database. In the ViewPoints framework, global consistency checking can be performed by transforming global checks into in- and inter-ViewPoint checks. For example, if a particular method requires that some consistency conditions hold for all ViewPoints, the method designer might define a ViewPoint to contain a representation of all the other ViewPoints. The global check then becomes an in-ViewPoint check for this new ViewPoint.

It should be noticed that at the inter-ViewPoint level all the rules at the in-ViewPoint level do not necessarily hold. There may be some circumstances under which a user may wish to perform an inter-ViewPoint check, without resolving local inconsistencies. One example is in a hierarchical decomposition of ViewPoints with parent and children [14]. It is not always necessary to apply the in-ViewPoints rules to children, then apply the inter-ViewPoints rules to parent and children. Sometimes, just inter-ViewPoint checking does make sense.

We described the concept of ViewPoints, categorized consistency rules. Due to the interest in requirements specification from multiple ViewPoints, the detection of ontological overlap and the resolution of the inconsistencies are big concerns in requirements engineering research. Several models have been proposed to manage inconsistency based on ViewPoints frameworks. Some of the approaches focus on the detection of particular types of inconsistencies between specifications expressed in specific representation models, while others are concerned with inconsistency in general. We briefly describe several popular models in the next section.

3.3 Different Approaches for Inconsistency Handling

In this section, we introduce several approaches for inconsistency handling. This section does not cover all of the approaches regarding inconsistency management. However, these approaches are the most popular ones in Requirements engineering research area. As a part of related work of this thesis, this section describes three kinds of approaches: logic based approach, reconciliation approach, and meta-model based approach.

3.3.1 A Logic Based Approach to Inconsistency Handling:

In this approach rules that specify how to act in the presence of inconsistency are explicitly specified. This approach includes following procedures [2]:

- Partial specification knowledge in each ViewPoint is translated to first order classical logic;
- Logical inconsistencies are identified;
- Temporal logic (meta-level) rules are combined with the inconsistencies identified to specify inconsistency handling actions.

In this approach classical logic is used to detect and identify inconsistencies. After the identification of inconsistencies, the meta-level axioms specify how to act according to the context of the inconsistency (Figure 3). This context will include the history behind the inconsistent data being put into the ViewPoint specification- as recorded in the ViewPoint work record-and the history of previous actions to handle the inconsistency. The meta-level axioms will also include implicit and explicit background information on the nature of certain kinds of inconsistencies, and how to deal with them. Action-based meta-language is based on linear-time temporal logic. We will not discuss them here.

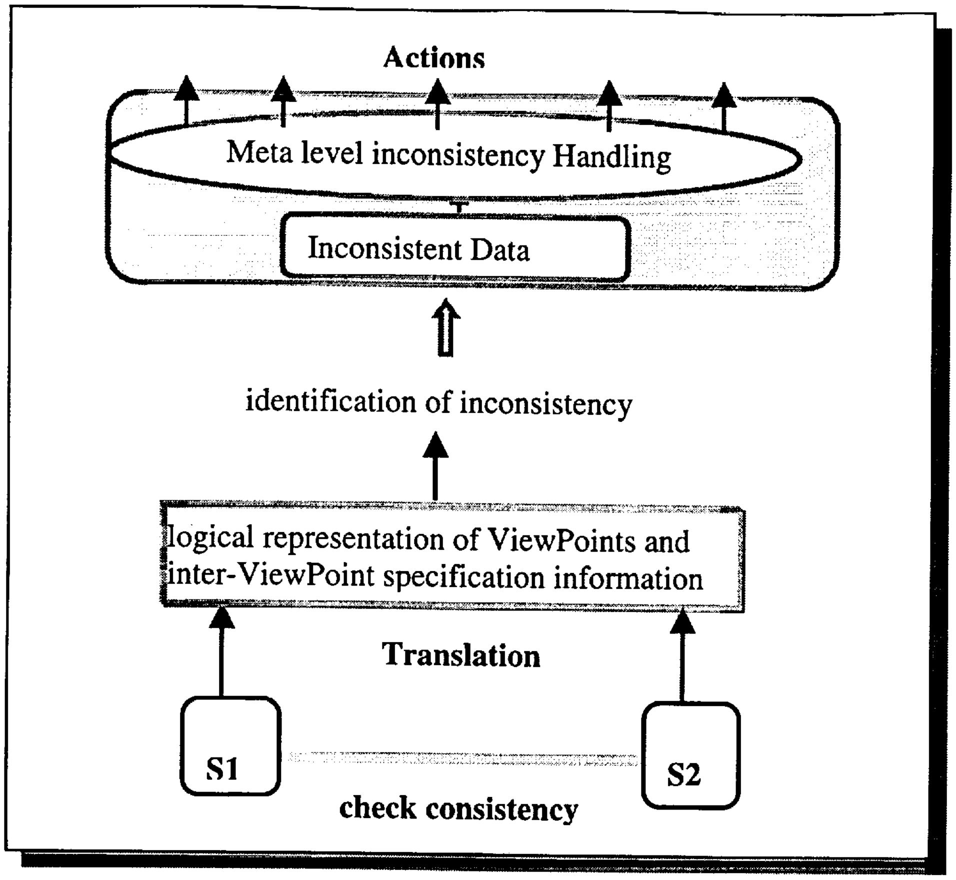

Actions
Meta level inconsistency Handling
Inconsistent Data
identification of inconsistency
logical representation of ViewPoints and inter-ViewPoint specification information
Translation
S1
S2
check consistency

Figure 3. A logic based approach for Inconsistency handling in the ViewPoints Framework.

Research on the detection and resolution of logical inconsistencies usually takes for granted the detection of ontological overlap [24]. There has been some work on the detection of ontological overlap based either on the generation of canonical representations of specifications in a common underlying language or on elaborating analogies between specifications. Such elaboration has been possible either by matching specifications with classes of requirements engineering problems or by heuristically identifying analogies from the annotation of specifications with terms in domain-specific dictionaries. The reconciliation approach is such a kind of model as described in the next section.

3.3.2 Reconciliation Approach to Interference Management.

Reconciliation was proposed by Spanoudaskis and Finkelstein [24]. It lies between loose and tight inconsistency management (Figure 4). It supports the detection, verification and tracking of ontological overlaps. It is amenable to the application of heuristic techniques, for example, inexact-matching mechanisms and models of computer-supported negotiation, while inconsistency detection may require theorem proving and sophisticated formal frameworks. It uses a meta-model to compute the similarities between ViewPoints. Such process needs to translate the knowledge of ViewPoints into a non-trivial domain-independent meta-model based on Telos objects [25] as described in next section.

This approach includes two basic stages; analysis and revision. Ontological overlaps are detected using a computational model of similarity and a classification of specification components with respect to a meta-model of domain-independent, semantic modeling properties-analysis [25].

It also supports the remodeling of specification components, so that the results of similarity analysis and ViewPoint owner assessment of overlaps is a convergent revision. The goal of this process is to ensure that the modeling of specifications is consistent with the human assessment of ontological overlaps between them and to establish a shared understanding among specification owners of the potential for inconsistency.

In the analysis stage, a specification is treated as an aggregation of "specification components", classified using a meta-model, which expresses general, domain independent, semantic modeling properties. Both this meta-model and the specifications are described in Telos, an object-oriented knowledge representation language, supporting the semantic modeling abstractions of classification, generalization and attribution.

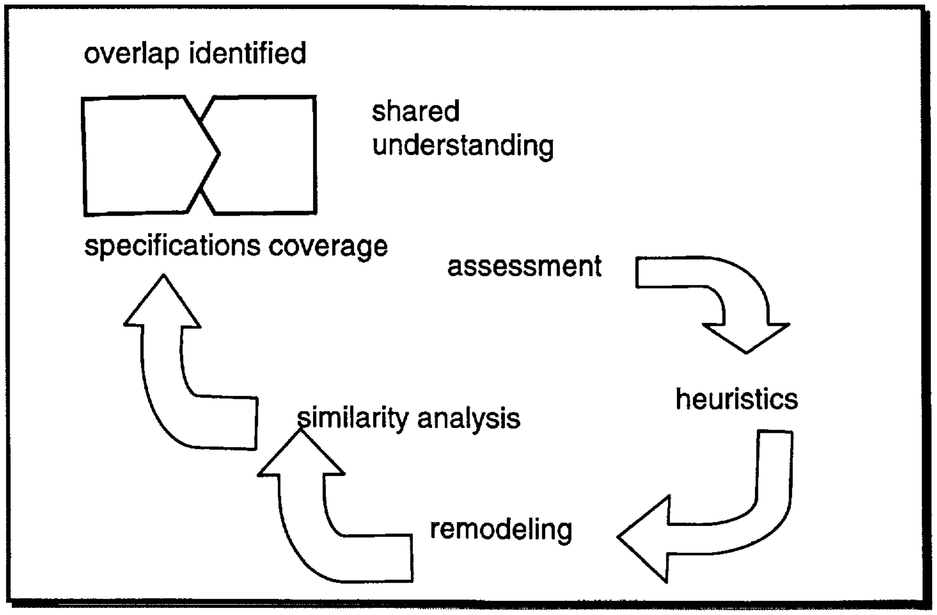


Figure 4. Reconciliation Approach to Interference Management.

The similarities between ViewPoints can be computed by meta-model. We discuss meta-model in next section.

3.3.3 Meta-model

The meta-model [26] consists of a kernel and a set of extensions. The kernel provides the key, domain-independent, properties of semantic modeling schemes, whereas the extensions provide additional properties for modeling established specification languages.

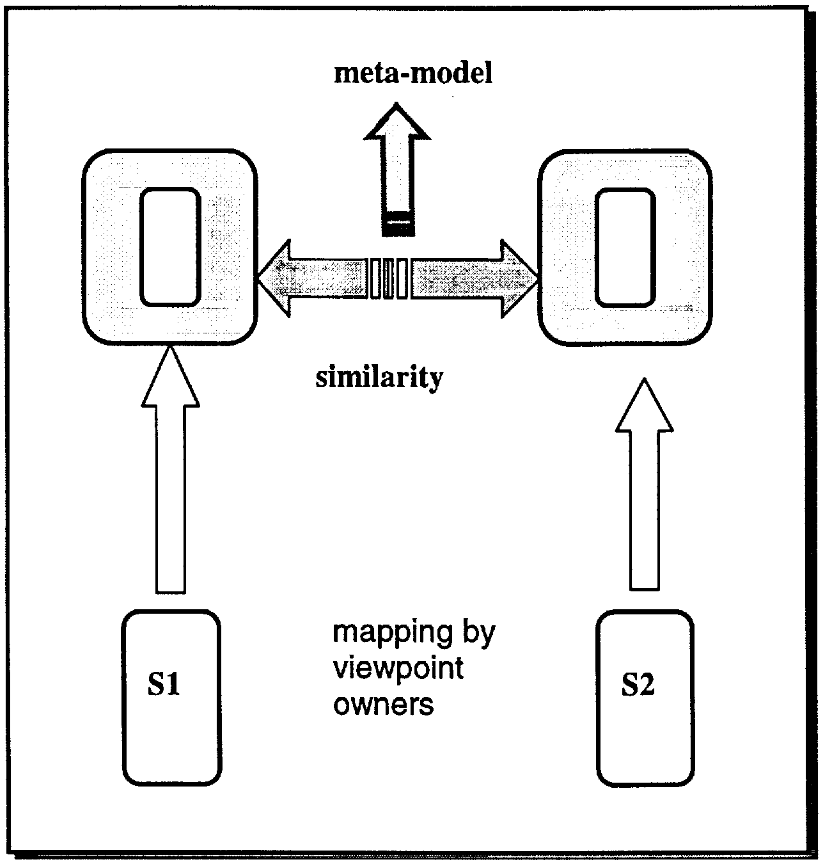


Figure 5. The Illustration of Meta-Model.

The kernel part is organized as a generalization of taxonomy of classes, with specification components classified by the properties they possess. In particular, components are distinguished into those representing entities and those representing relations. Entity representing components are further specialized into natural, nominal, place, event, activity, state, agent and physical quantity components. Components representing relations are initially distinguished by their arity. Binary relationships are further specialized according to: cardinality constraints (for example 1:1, N:M, total and onto relations); mathematical properties of relations (for example symmetric, transitive and set-inclusion relations); existential dependencies between related items or other constraints between them, including their temporal coexistence, physical separability and substance homogeneity.

The extensions to the meta-model comprise classes of additional properties for modeling established specification languages.

In essence, the meta-model enables the enrichment of the semantic content of specifications by asserting domain-independent properties about them and supports their representation with respect to a common set of structuring constructs.

The similarity analysis of two specifications results in

- their classification, generalization, attribution and overall distance measures
- a graph isomorphically mapping semantically homogeneous components at the successive levels of the structural closures of two specifications;
- a list with their common and non common classes, each weighted by its importance;
- a list with their common and non common superclass, each weighted by its importance.

The isomorphism between the components of two specifications is likely to reflect their ontological overlaps. However, flaws, incompleteness or lack of an adequate semantics in the modeling of specifications might force similarity analysis to generate a mapping that is incorrect. In such cases, specification owners can propose a different isomorphism between specification components, which in their opinion correctly reflects these overlaps.

We described different process models for inconsistency management. In the next two sections we describe two kinds of representations of partial specifications. The encoding of requirements specifications into these representations allows us to identify, analyze and manage inconsistencies between them. Let us start from graph grammars.

3.4 Graph Grammar

In this section we introduce the basic concept of a graph. Our framework for inconsistency management is built on graph grammar [4]. The exploration of graph grammars is explained in Chapter five. A graph is a universal data model to describe static properties of objects and the structure of their relationships. It plays an important role within many areas of applied computer science, and there exists an abundance of visual languages and environments that have graphs as their underlying data model. Rule-based languages and systems have proven to be well-suited for the description of complex transformation or inference processes on complex data structures. Although graphs and rule-based systems or grammars are quite popular among computer scientists, their combination in the form of graph rewriting systems or graph grammars is more or less unknown. One reason is, many people believe that modeling with graphs and graph rewriting systems leads to inherently inefficient implementations due to the NP-completeness of many graph algorithms. This situation changed gradually with the appearance of the first graph rewriting system implementation like PAGG [27], GOOD [28], and PROGRES (PROgramming Graph REwriting System) [29]. In our approach, we use graph grammar as a framework to manipulate graph data rather than to explore the NP-completeness problems in graph theory. The application of graph grammars is performed as controlled rather than non-deterministically. As graphs are the basic data model for our approach, we first introduce what is graph.

Definition (graph)

A (directed, labeled) graph is a 6-tuple containing system as Figure 6 shown. It is represented as M=(V, E, s, t, l, m), where

(i) V is a finite set of nodes,

(ii) E is a finite set of edges,

(iii) s, t : E → V are mappings assigning to each edge $e \in E$ a source s(e) and a target t(e).

(iv) l: V→ Σ_V is a mapping, called the node labeling,

(v) m: E→ Σ_E is a mapping, called the edge labeling.

Note, we always assume that two alphabets Σ_V and Σ_E (of node and edge labels respectively) are available.

Definition (graph match)

A graph match g from a definition graph N into an image graph M is a structure preserving mapping between these graphs (Figure 6). It assures that the source and target of each edge in graph N is mapped consistently onto nodes in the target graph M such that the graph structure of N is mapped onto a proper subgraph of M (predicates g_v, and g_E). Further, it may be checked if nodes and edges, respectively, match by comparing their labels. This is done using attribute match predicates p_v and p_E for node and edge labels, respectively.

If the mapping predicates g_v, g_E, p_v, p_E exist between graph M, N, then M and N are called homomorphic and M→N is a homomorphism. If the underlying mappings are bijections, M→N is said to be an isomorphism, and the graphs M and N are called isomorphic.

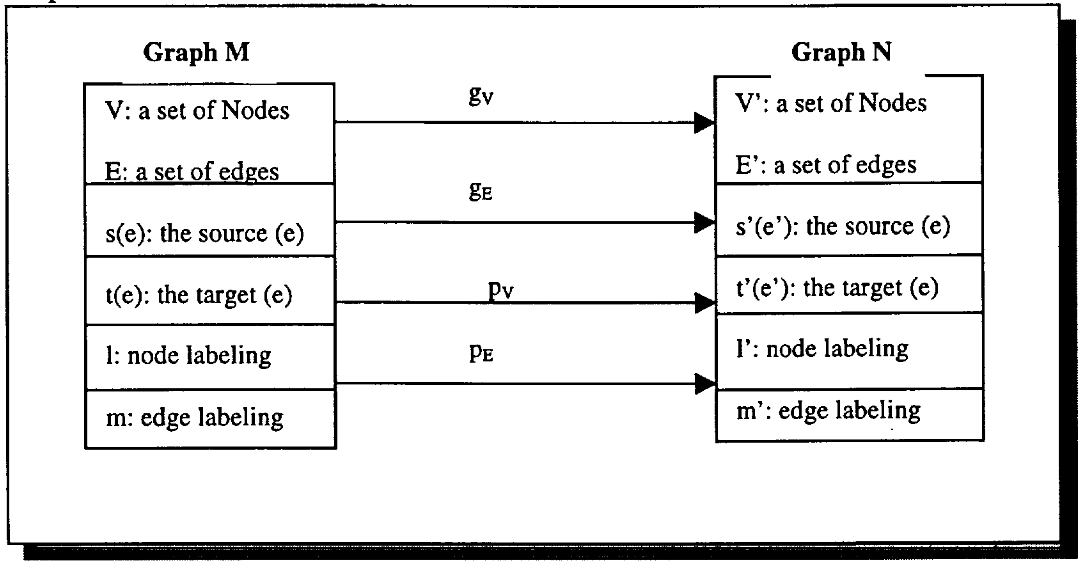


Figure 6. Graph match, g_v, g_E, p_v, p_E are defined by method users

Dynamic aspects can be represented by graph rewrite rules that identify a subgraph in a source graph and replace it by a new subgraph. Types of graphs and rewrite rules can be encapsulated in graph grammars and graph grammar modules.

Definition (graph rewrite rule)

A graph rewrite rule r = (L, R, K) is a triple of graphs, where graph L is left hand side of the rule, graph R is right hand side of the rule and graph K is called gluing graph. K ensures that no dangling edges appear in the new graph after applying r to the old graph, hence, K identifies anchor elements which have to remain unchanged by the modification and is subgraph of L and R as well.

Definition (modification of a graph)

A modification of a graph M into a new graph N by applying a graph rewrite rule r = (L, R, K) is realized by the following two principal steps as in Figure 7 shown.

1. Search for a subgraph, termed redex, in a given host graph that matches the rewriting rule's left-hand side.

2. Replace the selected redex by a copy of its right-hand side, but preserve all those nodes (and their context and attribute values) which are shared among its left-hand and right-hand side.

The application of a graph rewrite rule r on a source graph M yielding the target graph N is called a graph rewrite step and noted as $M \xRightarrow{r} N$ (Figure 7).

We introduced the basic concepts of a graph grammar. Based on graph match and graph rewriting rules our approach is explained in Chapter five. In next section, we introduce another representation of partial specifications Quasi-classical logical.

3.5 Quasi-classical (QC) Logic

Quasi-classical logic [3] is an adaptation of classical logic. It allows continued reasoning in the presence of inconsistency. The adaptation is a weakening of classical logic that prohibits all trivial derivations, but still allows all resolutions of assumptions to be derived.

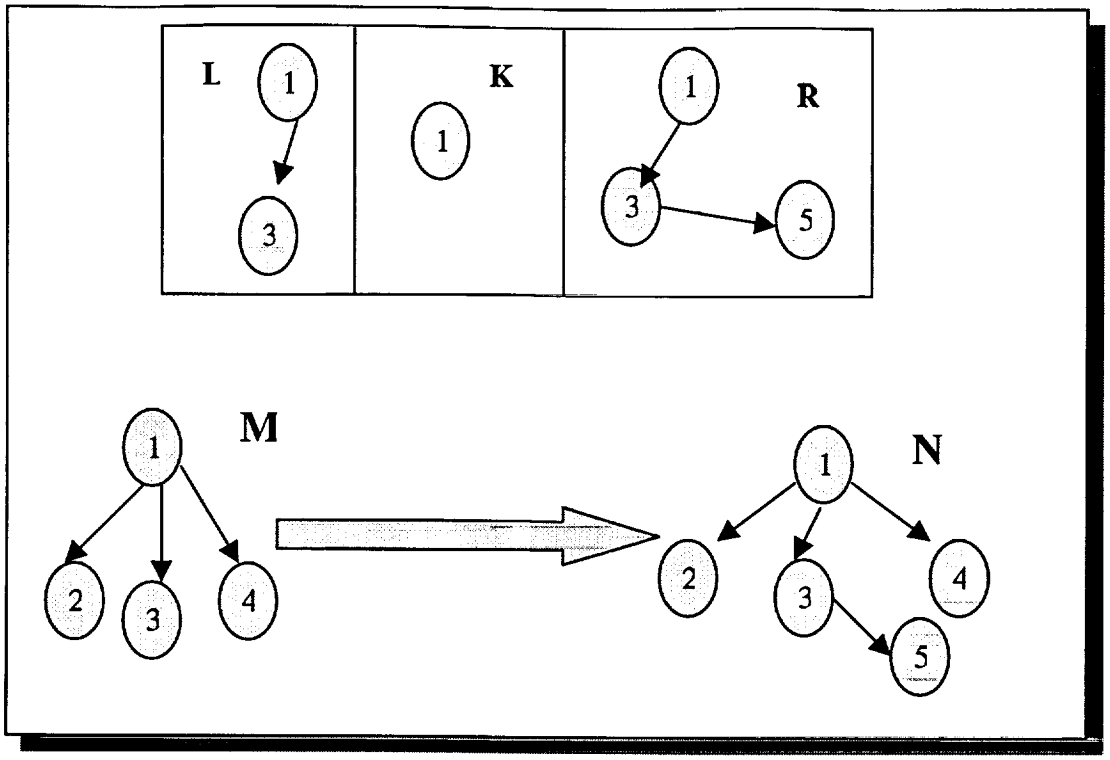


Figure 7. The application of graph rewriting rule

An inconsistency in logic results from the simultaneous assertion of a fact α and its negation, $\neg\alpha$. Using this definition, translating software specifications into logic facilitates detection of inconsistencies and allows us to concentrate on reasoning about these inconsistencies. However, in practical reasoning, it is common that there is inconsistent information in a specification (e.g., multiple contradictory requirements of a system). Classical logic is trivialized because, by the definition of logic, any inference follows from inconsistent information. For classical logic, trivialization renders the specification useless, and therefore classical logic is obviously unsatisfactory for handling inconsistent information. QC-logic weakens classical logic by prohibiting all trivial derivations. A language called labeled QC logic is developed to record and track assumptions used in reasoning. This allows each item of development information to be identified, and each inference to be labeled. Labels can be used to differentiate different types of development information and in particular they can indicate the source of the information.

We covered various topics in this chapter. They are the foundation of understanding our approach. We recapitulate our introduction and discussion in the next section, and move to explanation of our approach.

3.6 Summary

This chapter extensively introduced several important concepts for our inconsistency management framework. We surveyed existing CASE tools, and found that none of them captures all of the information in an evolving development process, and none of them supports the consistency and completeness checking for the same piece of specification in different notations. We talk about the ViewPoints framework, which is the theoretical foundation of the WHERE project. Also, we surveyed related work, the ViewPoints based inconsistency management process models. In section 3.4 and section 3.5 we introduced two kinds of representations that could represent partial specifications, graph grammars and QC-logic. We will explore these two kinds of representation in chapter five further. From an application view of point, an ideal CASE tool based on ViewPoints should have some similarities to the tool described below.

The tool can be used to provide complete support for the entire requirements specification process, or as a partial aid to some aspects of it. The partial mode allows a gradual introduction of the tool into existing projects. For example, existing specification documents can be loaded into the ViewPoint reviewer, and annotated without much effort. New ViewPoints can then be created using the ViewPoint editor, with defined relationships to the existing documentation. A typical use of the tool would then be to model a portion of an existing specification in a new notation, whilst explicitly recording the relationships between the existing specification and the new model.

Some tools about the ViewPoints framework have been developed as research prototypes. For example, a tool called viewer was developed at Imperial College, London [11], but for inconsistency management the internal implementation is not available yet. We do not know of any currently available commercial tools in this area. In next two Chapters, we describe our approach for inconsistency management based graph grammar and TCMJAVA infrastructure. In chapter four, we first describe our approach for inconsistency management. Then we explain how our framework could be implemented.

Chapter Four An Approach for Inconsistency Management Based on Graph Rewriting System

This Chapter describes our framework for inconsistency management. This approach is based on graph rewriting rules system. Graph is a universal data model to describe static properties of objects and the structure of their relationships. Dynamic aspects can be represented by graph rewrite rules that identify subgraphs in a source graph and replace it by a new graph. Our approach is described from two aspects, the static graph manipulation and dynamic transformation. And the application of these two aspects on inconsistency management is outlined. In section 4.1, we first describe different relationships between ViewPoints in general. In section 4.2, we describe the overview of our framework, and discuss the application domains of our approach. In section 4.3, we revisit the data structure of shape object in TCMJAVA, in order to help the explanation of our implementation for this framework. In section 4.4 we elaborate the basic graph manipulation operations, UNION, DIFFERENCE, SELECTION, INTERSECTION. And how we can query or manipulate graphs based on these basic operations. In section 4.5, we explain the need for graph module. In section 4.6, we describe graph transformation, and how graph production can be used in inconsistency management. In section 4.7, we evaluate our approach, explain the deficiencies. In section 4.8, we recapitulate our achievement. Our approach is an intuitive and easy to implement, because the infrastructure for graph systems are already established. In next section we first describe the different relationships that may exist between ViewPoints in general.

4.1 The Relationships Between ViewPoints

There are four kinds of inter-ViewPoint relationships that may exist between ViewPoints in general [11]. We give an overview of these different relationships, and explain the application domain of our approach in next section.

As Figure 1 shown, the first kind of relationship (1.1) is two ViewPoints may be completely independent. Two ViewPoints have no overlaps, and they are not related. For example, a ViewPoint describes university faculty income; and another ViewPoint describes the gardening facilities at that university. Those two ViewPoints may not have any relationships. There exist some existential relationships between two different

ViewPoints in the second kind of case (1.2). The existence of one ViewPoint may depend upon the other ViewPoint in some sense. For example, In a university community, a ViewPoint describes the whole organization of this university. There is a separate ViewPoint to describe every unit in the organization. The third case (1.3) is that two ViewPoints may partially overlap. A partial specification in a ViewPoint is related to a partial specification in the other ViewPoint. This kind of relationship may include the case that one ViewPoint is a subset of the other ViewPoint. For example, a ViewPoint describes the faculty publications at a university. The other Viewpoint describes the graduate student publications at the university. The last case (1.4) is that two ViewPoints may totally overlap. In another words, two ViewPoints describe the same domain or system. In this case, two ViewPoints may use two different representations for the same domain or scheme, or the two ViewPoints may be required exact same, just from different people; there should not have any conflicts, discrepancies or inconsistencies existing between these ViewPoints.

We described the different relationships that may exist between two ViewPoints. In next section we delineate our approach based on graph grammar, and discuss the application scope that our approach can be applied.

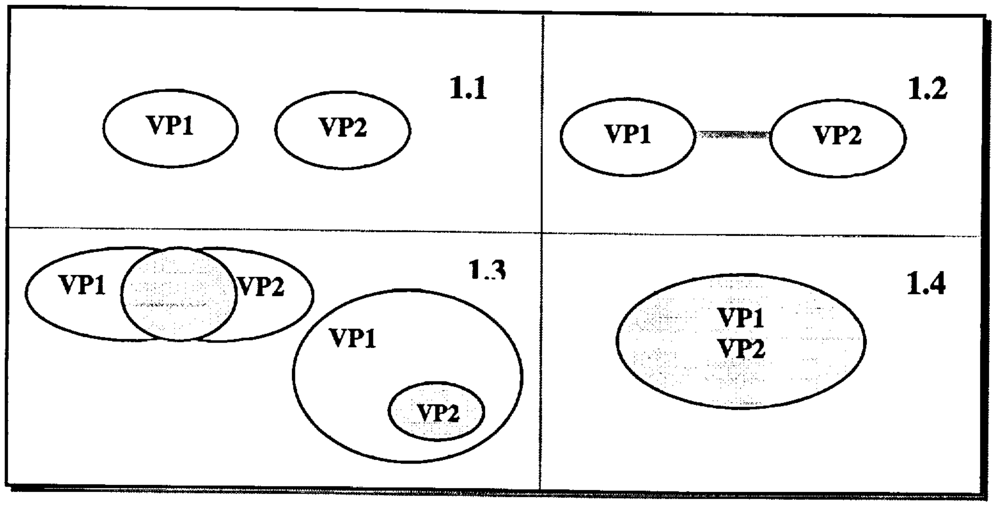


Figure 1. Different Inter-ViewPoint relationships. Shaded areas represent overlaps between ViewPoints.

4.2 An Overview of our Approach

Figure 2 summaries our experimental inconsistency handling approach. The steps in this approach can be viewed as following,

- Partial specification knowledge in each ViewPoint is encoded into a separate graph.
- Overlaps are identified using graph grammar matching, or defined by method users.
- Inconsistencies can be detected using graph match or Quasi-classical logic. The adoption of method depends on the application domain or system. For graphical notations, graph match is more likely to be employed. The overlap part are represented by two partial specifications, if a homomorphism exists between those two partial specifications, two ViewPoints (specifications) are inconsistent. And vice versa. The seeking of homomorphism also supplies useful information for inconsistencies.
- For ViewPoints represented in graphical notations, graph rewriting rules are very useful to express the transformations of graph systems. Transformation information is recorded using graph rewriting rules. The reversing application of rewriting rules supports "UNDO" operation, and consequently supports inconsistency management.
- Temporal logic (meta-level) rules are combined with the inconsistencies identified and human support to specify inconsistency handling actions.

Our approach has following characteristics.

- This approach is intuitive for ViewPoints represented by graphical notations.
- The detection of overlaps involves a fairly easy and straightforward process, and the non-trivial translation from original specification into formal logic can be avoided for ViewPoints represented by graphical notations.
- In order to manage inconsistencies, the relationships between viewpoints need to be clearly defined. This process is not trivial. Usually the success of defining relationships determines the quality of inconsistency detection. Our approach reduces the work of defining relationships. After an overlap between two ViewPoints has been identified, we can focus on the relationships of the complement part of the overlap part in a ViewPoint to the complement part of the overlap part in the other

ViewPoint, complement parts to overlap, and ignore the relationships inside of overlaps.

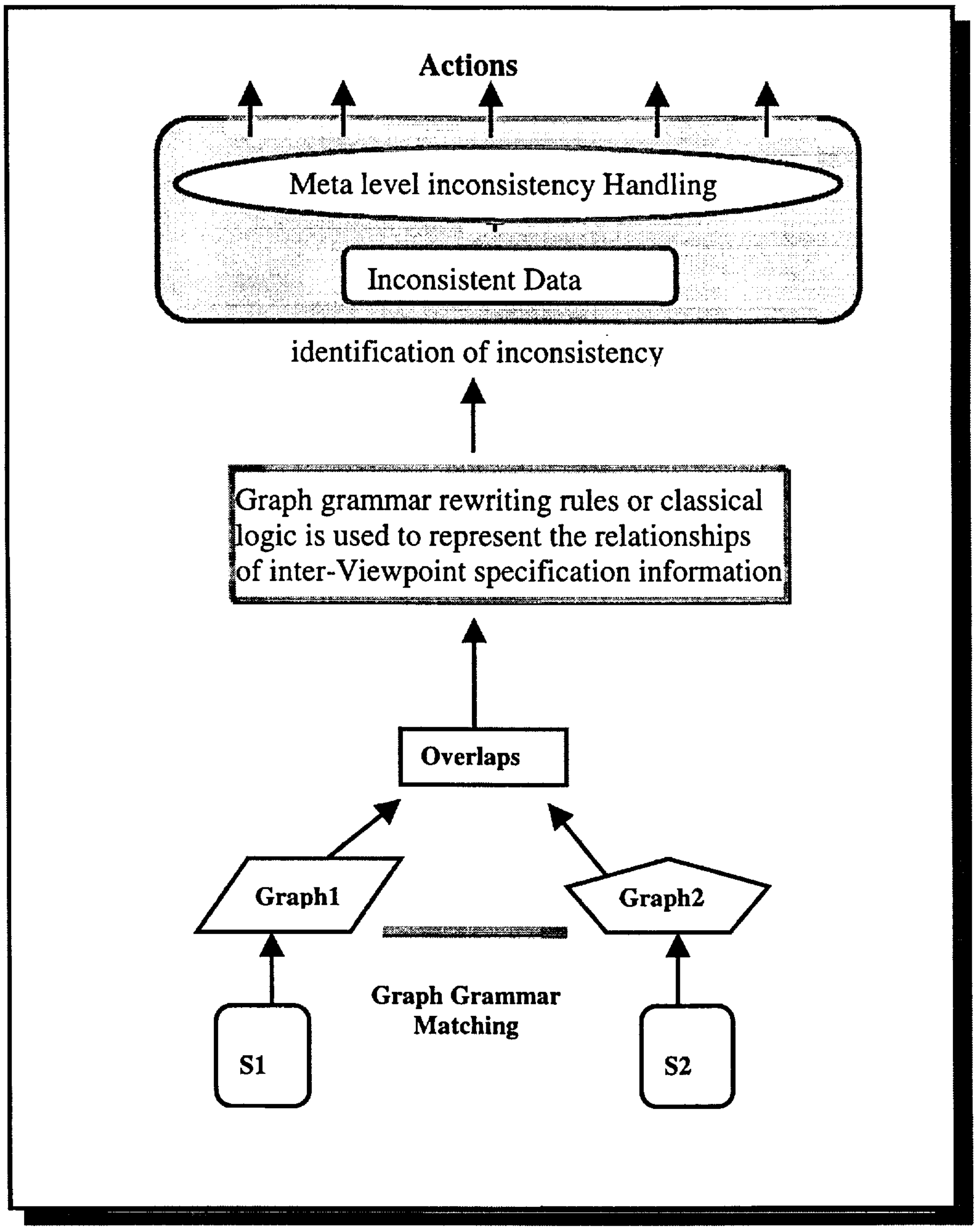

Actions
Meta level inconsistency Handling
Inconsistent Data
identification of inconsistency
Graph grammar rewriting rules or classical logic is used to represent the relationships of inter-Viewpoint specification information
Overlaps
Graph1
Graph2
Graph Grammar Matching
S1
S2

Figure 2. Inconsistency handling in the Viewpoints framework. Each viewpoint is translated into graph, different inconsistency detection methods can be used.

- Depends on the application domain, users have flexible to make selection of graph grammar or quasi-classical logical.
- The detection of inconsistencies can be realized by seeking the homomorphism between two graph systems. Method user defines mapping functions between graph systems using user friendly windows step by step.
- TCMJava already has a class called Shape which record all of the information about graphical node or edge attributes, and a class called Cell which record all the information of table cell. The translation of diagram or table into a graph will be a selecting and reorganizing process. The infrastructure of graph system is already available.
- The application of graph rewriting rules for ViewPoints represented by graphical notations supports inconsistency management. The recording of graph rewriting rules supports tracing information, ameliorating and rectifying inconsistencies, and helping a seek an effective way to resolve inconsistencies.

We described our approach for inconsistency detection. And in section 4.1 we explained that there are four inter-ViewPoints relationships between two different ViewPoints. A question is what is the application domain that our approach can be applied to? Our approach can be utilized to detect the inconsistencies for the case **3** and case **4** described in section 4.1 for ViewPoints represented in graphical notations. One ViewPoint may be partially overlapping with another as Figure 1 case 3 shown. The overlap can be detected by subgraph querying from the two ViewPoints, or can be defined by method user. After the detection of overlap, if a homomorphism exists between a partial specification in a ViewPoint and a partial specification in the other ViewPoint, and those partial specifications refer to the overlap, then no conflicts or inconsistencies should exist between those two ViewPoints. The querying of subgraph from a graph system (ViewPoint) will be discussed in section 4.4. If two ViewPoints describe the same domain as Figure case 4 shown, in this case, seeking a homomorphism is the goal. If a homomorphism exists between these two ViewPoints, the two specifications represented by those ViewPoints are consistent. And vice versa. Also, in this case, the two ViewPoints is totally overlapping, the process of finding homomorphism supplies useful information for the inconsistencies existing between the two ViewPoints, and later helps resolve inconsistencies. Our case study described in Chapter Six employs such an example to explain how our approach is utilized.

In next section, we revisit the implementation of TCMJAVA, especially Shape class. This helps us understand the basic static manipulation of graph system.

4.3 Visiting Shape class in TCMJAVA

The Shape class in TCMJAVA is a very huge class. The class hierarchy is complex, and partially described in **Figure 3**. Before talking about our implementation for inconsistency management based on Graph Grammars, we should mention several important attributes for the Shape class in the implementation of TCMJava. The Shape class is characterized by; the type of the Shape class, e. g. BOX, CYCLE, LINE; two positions; the topleft position and the central position of the Shape class; the name of the Shape, e.g. "my node". A node of a diagram in TCMJava is mapped to a node in a graph system. A name of a node in TCMJava is mapped to a label of node in a graph system. An edge of a diagram in TCMJava is mapped to an edge in a graph system. A name of an edge in TCMJava is mapped to a label of edge in a graph system. Edge shape in TCMJAVA has two node shapes as its attributes. They represent the source and the target of an edge shape. These two node attributes of an edge shape are mapped to the source and the target node of an edge in a graph system. A diagram in TCMJava may include more than one node or edge. Edge shape can not exist independently; there exist two nodes associating with an edge in TCMJava. The storage of a group of shapes in TCMJava is described in following paragraph.

In a diagram, all drawing shapes are stored in a Vector (java.util.Vector) according to the sequence of drawing. Therefore, the manipulation of shapes in a diagram is just operations on the Vector. For example, the delete operation on a subgraph selected is just an operation that deletes several elements form a Vector. Java Vector class supports enough methods on Vector manipulation, they are very useful in our study.

In a table of TCMJava, cell class records all the information about cell text, cell position, cell height, cell width, etc. The mapping between a table to a graph system is not very explicit. The mapping relationships may vary. Here we give an example for state transition table, which will be used in chapter six. The cells in the first row or the first column of a state transition table in TCMJava are mapped to the nodes of a graph system. The texts of all cells in the first row or the first column are mapped to the labels of nodes in a graph system. And all cells in the first row of a table should be same as all cells in the first column of the table sequentially. For example, the text in cell (1, 2) should be same as cell (2, 1). This constraint should be enforced inside of a ViewPoint by method engineer. If a cell in a state transition table is empty, then there should not exist a

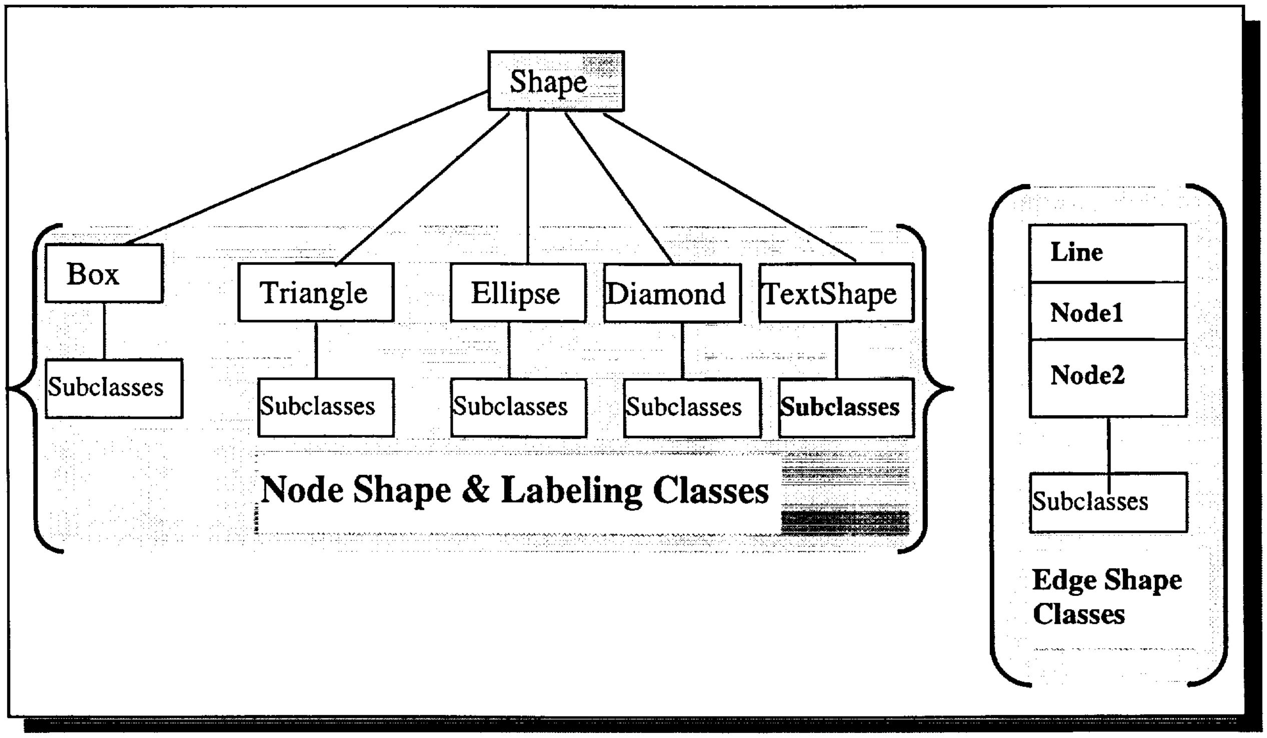


Figure 3. The class hierarchy of TCMJAVA Shape class

transition in the table, or an edge in a graph system. Transition cells are the cells that are not empty and not the cells in the first row or column in a table. Transition cells in TCMJava are mapped to edges in a graph system. Texts in cells are mapped to labels for nodes or edges in a graph system. The "from" nodes and "to" nodes in a table are mapped to source nodes and target nodes in a graph system respectively. CellVector class stores all information in a row or a column in a table.

We described the data structure for shape class in TCMJava and the mapping relationships between TCMJava shape or table class to a graph system. In next section we will discuss the static aspects of graph system, and how they can be implemented based on TCMJAVA.

4.4 Graph query and replacement

In this section we first introduce several basic operations for graph system manipulation based on TCMJAVA implementation. Because graph-rewriting system is also a rule based system, in rule-based systems complex operations are based on several simple basic operations. Complex operations can be derived from the combination of simple basic operations. This claim in rule-based system is sound. In our study, we define four basic operations:

UNION (or): for two sets of shapes A and B, the performance of union operation assumes the two set A and B is disjoint. This operation can be implemented by appending one shapes List (Vector) to the other shapes List (Vector).

INTERSECTION (and): Two vectors store two sets of shapes. The utilization of this operation should find the common shape objects in the two shape vectors. Java programming language supports data persistence, two objects can be compared each other. Then common shape objects can be obtained in a $O(n^2)$ running time.

DIFFERENCE (but not): This operation can be implemented as selecting all of the shape objects in one list of shapes, but not in the other list of shapes. As a complementary operation of INTERSECTION operation, the running time of this operation is $O(n^2)$.

SELECTION: This operation needs two parameters, a list of shapes, and a shape object that need to be identified and selected from the list of shapes. A implementation could be as following. For example, selecting of shapes with a specific shape name. Define a new vector first, then compare every shape object in the list of shapes, if a shape object is found same as the target shape, then put the shape object found in the new vector, At last return the vector.

```
public Vector Selection (String shapename, Vector shapes) {
        Vector p = new Vector();
        for (int I=0; I<shapes.size(); I++) {
                if (shapes.elementAt(I).name.equals(shapename)
                        p.addElement(shapes.elementAt(I));
        }
        return p;
}
```

The subgraph queries, replacement and other complex operations can be implemented using these simple operations. For example, the Merge function can be implemented as following:

```
public vector Merge (Vector shapes1, Vector shapes2) {

        Vector set1 = shapes1 DIFFERENCE shapes2;
        Vector set2 = shapes2 DIFFERENCE shapes1;
        return (set1 UNION set2 UNION (shapes1 INTERSECTION shapes));
}
```

The Merge function basically discards the common objects in two sets, then unions all the objects. Therefore, the static manipulation on graph system is deterministic and can be finished in polynomial running time. In next section, we describe the need for graph module.

4.5 Graph Module

In a decomposition hierarchy based development process, top level task usually decompose into small subtasks. Then assign those small subtasks to other people to develop. One simple example is explained in chapter two (Figure 4 in chapter two). The process Bob performs is a subtask in Ann's process. In order to check the consistency between this two ViewPoints, a graph module is needed to encapsulate the information in Bob's process, and check the consistency between Bob's process (as graph module) to Ann's sub-process B. Based on the implementation of TCMJAVA, the implementation for a graph module is practical. For example, for figure 4 in chapter 2, we can use incidence or outgoing edges as attribute in graph module class, and use a Vector to store other shapes inside the module. The graph match also can be performed on the checking between a graph atomic unit and a graph module. In Figure 4, we show the different comparison cases for ViewPoints, ViewPoint modules and atomic units in a graph system. Case 1 shows a comparison between a graph node with a ViewPoint; Case 2 shows a comparison between two ViewPoints; Case 3 shows a comparison between two ViewPoint modules; Case 4 shows a comparison between a ViewPoint and a ViewPoint module. Graph module encapsulates non-deterministic as well as controlled graph rewriting according to software engineering modularization techniques.

In next section we will discuss, the dynamic aspect of graph, graph rewriting rule, or graph production to represent graph transformation mechanisms.

4.6 The Application of Graph Rewriting Rules

In previous sections we described graph static properties, and the basic operations to manipulate graph. In this section we explain what is graph transformation, and how it supports inconsistency management.

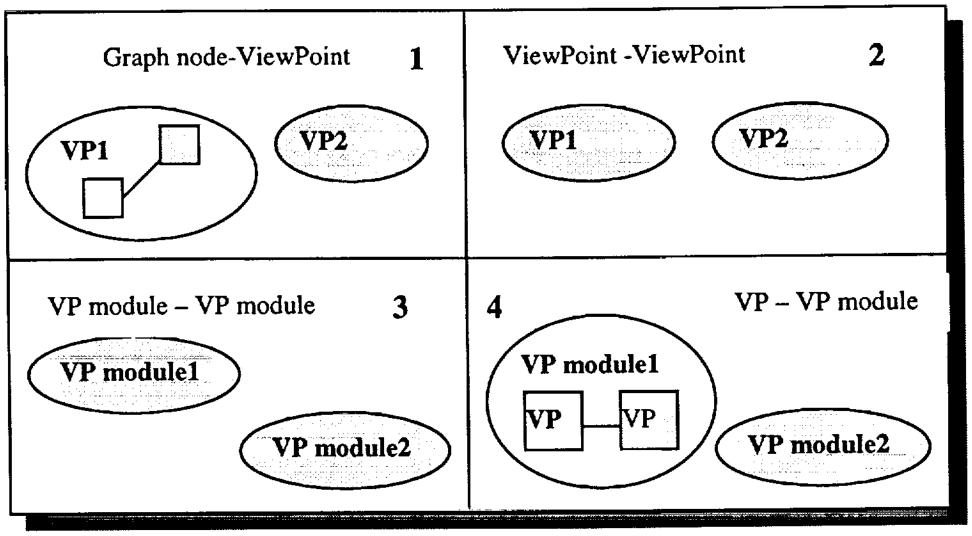
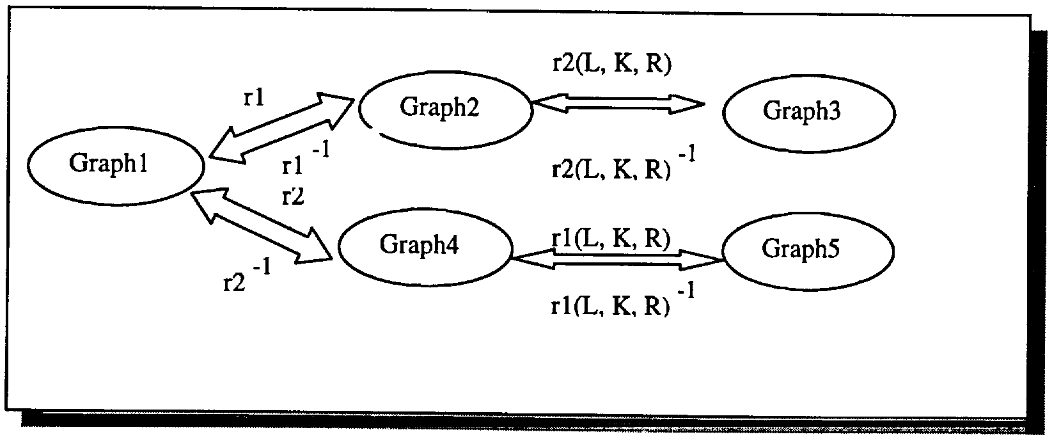


Figure 4. Different comparison cases among ViewPoints, ViewPoint modules, and atomic unit in a graph system. Showed areas represented domains in comparison.

Subgraph tests and queries are the main operations for inspecting already existing graphs, whereas productions is their counterparts for creating and modifying graphs. Productions have a *left-* and a *rigt-hand side* graph patterns as their main components as described in chapter four. Dynamic aspects of graph can be represented by graph rewriting rules. The application of graph rewriting rules keeps any graph transformation information in the development process. Therefore, the reversing application of graph rewriting rules can go back to previous steps in graph transformation process. This supports "Undo" operation in development process. The employment of graph rewriting rules supports method user to explore alternatives, and supplies help to a method user to search for an inconsistency resolution. Hence, the application of graph rewriting rules supports inconsistency reasoning, analysis, diagnosis, tracing information, resolution, amelioration, rational. Therefore, our framework supports inconsistency management. Figure 5 shows an example for the application of graph rewriting rules. There are two graph rewriting rules r1 and r2. The different sequence of applying rules results into different graphs. But a method user can always go back to previous steps by applying graph rewriting reversing rules ($r1^{-1}$ and $r2^{-1}$), in order to find a satisfied state (a consistent state) for development process. The help from a method user in searching for a resolution of an inconsistency can greatly reduce the running time (O((number_of_rules)!)). The implementation issues are discussed in next paragraph.

The data structures of the WHERE and the characteristics of Java programming language determine that the implementation of graph productions is feasible. Graph rewriting rules as data objects can be stored in a Vector data structure (Figure 6). Then, a graph rewriting rule as an element in the Vector can be retrieved and applied to a graph system. The application of graph rewriting rule needs the basic operations as described in the previous section. For example, a query operation is needed to find a the left-hand side of a rewriting rule, then a replacement operation is needed to replace the left-hand side subgraph with the right-hand side subgraph for a rewriting rule. All graph rewriting rules employed for a graph system is stored in the work plan slot of a ViewPoint. In next section we evaluate our approach for inconsistency detection and management based on graph grammar.



rule1	rule2	rule3	rule4	rule4	rule5	rule6	...

Figure 6. The storage for graph rewriting rules applied to a graph system.

4.7 Evaluation of Approach

Our approach is explicit for ViewPoints represented in graphical notations. The encoding from graphical notations into graph systems is straightforward. The manipulation of graph systems is deterministic with the help of method users. If two ViewPoints have no overlap, it does not make sense to check inconsistency. If two ViewPoints have an existential relationship, our approach is difficult to be employed. Also, if specifications are represented in mathematical notations or natural language, and difficult to be converted into graph systems. Our approach is not applicable to these kinds of systems. In our framework, we adopt meta-level rules for "inconsistency implies action", and emphasize that such inconsistency handling actions need not necessarily remove inconsistencies. How does it combine with graph grammar approach is a problem. In our framework, method user defines mapping functions in seeking homomorphism between two graph systems by using a series of windows. Sometimes, the definition of mapping functions are blur, the seeking of mapping functions is not very easy. This is an engineering issue, also is an implementation issue. In next section we summary our achievements.

4.8 Summary

In the complex system development process, requirements specifications usually are expressed as various graphical notations. The internal representation of most software documents is a directed attributed graph. Sometimes even for same piece of specification document, it can be represented using different graphical notations. We discussed different inter-ViewPoint relationships in this chapter, and explained that our approach fits for two ViewPoints partially overlapped or totally overlapped. Our approach focuses on graphical requirements specifications. The encoding of a piece of diagrammatic or tabular specification into graph system is explicit and direct. After the conversion of specifications from graphical notations to graph systems, method users define mapping functions to detect if there exists graph homomorphism between two graph systems. The finding of homomorphism helps detection and analysis of inconsistencies. The subgraph queries and mappings make use of basic operators INTERSECTION, UNION, DIFFERENCE, and SELECTION. In our implementation, method users define mapping functions via user friendly wizard windows.

In this thesis the dynamic graph transformation in support of inconsistency management is also investigated. Graph rewriting rules store the information of evolving process. The recording of all graph rewriting rules helps backtracking to previous state in development process; supports the exploration of alternatives; and supplies valuable help to search for an inconsistency resolution. As a data type, rules are stored and manipulated in our approach. In the search of subgraphs for the writing rules, the established static operators are used for queries and replacements. In next Chapter, we present the applicability of our approach based on International Space Station Specification.

Chapter Five Case Study – International Space Station Specification

This chapter introduces the case study that was conducted to assess the applicability of our inconsistency detection framework. The scenario introduced in this chapter describes the verification and validation activities conducted by an IV&V term. The findings of this case study confirmed that inconsistencies are inevitable, the use of our framework will detect some of the inconsistencies experienced in large software projects.

In section 5.1, we briefly describe the case study document about. In section 5.2, we introduce the concept of Independent Verification and Validation (IV&V). We start by describing traditional software verification and validation processes. Then we introduce IV&V process and describe the roles of the parties involved. Having described IV&V process, section 5.2.3 and 5.2.4 describe the tools and methods of an IV&V term uses, and the documents of an IV&V term faces. In section 5.3, the original valid Space Station Mode and state transitions requirements for space station specification is described. Section 5.4 explains the consistency checking process, and in section 5.5, step by step a working process to apply our consistency-checking engine is described. Finally, section 5.6 recapitulates the applicability of our approach to the International Space Station Specifications.

5.1 International Space Station Specification

The International Space Station (ISS) is a co-operative project to establish an Earth orbiting facility in which scientific experimentation is conducted. This project involves a cooperation effort from six international partners. The piece of requirements specification used in this case study is about automated station control requirements. The automated control capability is intended to provide a structured, well-understood framework for operator control of station activities and operations. To ensure the quality of the ISS software system, NASA has an IV&V term to oversee the entire ISS software development process. The need for verification and validation of software and the concept of IV&V are discussed in next section.

5.2 What is Verification and Validation?

Verification and Validation are two commonly used terms in software engineering. They mean two different types of analysis in the software development process. According to their definitions from ANSI/IEEE:

Validation is the evaluation of software at the end of the software development process to ensure compliance with the user requirements.

"Verification is the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements (ANSI/ASQC A3-1978) [32][33].

In other words, validation answers such question "Are we building the correct system?" and validation is 'end-to-end' verification." Verification answers question "Are we building the system correctly [6]? "

Different activities are involved in verification and validation process, verification includes all the activities associated with the producing high quality software, such as testing, inspection, and specification analysis. In comparison, validation includes the activities to check how well the system proposed addresses a real-world need. For example, activities like requirements modeling, prototyping and user evaluation. But in a real world application, the distinction between these two analyses is not important. V&V is now regarded as a coherent discipline: "Software V&V is a system engineering discipline which evaluates the software in a system context, relative to all system elements of hardware, users, and other software" [34].

5.2.1 Independent Verification and Validation Process

For independent verification and validation (IV&V) process, the customer hires a separate contractor to analyze the products besides the software development contractors. The system under development is usually high-cost and safety-critical. IV&V process is used to overcome analysis bias and reduce development risk. This process is performed in parallel with the development process throughout the software lifecycle, and is independent of any inner V&V performed by the development contractors. The customer relies on the IV&V contractor as an informed, unbiased advocate to assess the status of a project's schedule, cost, and the viability of its product during development. Following is the definition of IV&V process from NASA "Software Assurance Guidebook" [35].

"IV&V is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software. The IV&V agent should have no stake in the success or failure of the software."

The analysis activities involved in IV&V process are various, they are relevant to requirements, design, code, performance, schedule, cost, and testing.

The main purpose of IV&V process is to reduce risk, besides IV&V has other benefits. For example, the earlier errors are found in the development process, the lower the cost to get them fixed. The clearer the requirements specifications used to drive the implementing and testing processes are, the less effort may be needed for error-removal and re-testing. Empirical studies show that the application of IV&V makes significant saving, and the delivered software have fewer defects [6].

Figure 1 shows the relationship between IV&V agent, developer and customer in the ideal model for IV&V process. Different communication channels may exist in IV&V processes. The IV&V agent might report to the program management office within the customer, or directly to the development contractor [6].

5.2.2 The Characteristics of IV&V Process.

In IV&V practice, different degrees of independence are possible. Roughly there are three kinds independence: managerial, financial and technical. We are going to focus on technical independence rather than managerial and financial independence here. We say an IV&V agent is technically independent, we mean an IV&V team gains different perspective with the using personal who are not involved in the development effort, and the understanding of the software and its requirements from IV&V team is not influenced by the development team. Usually the use of different tools or techniques with the V&V team in developer contractor enhances the technical independence of an IV&V team. The experimenting of the emerging tools and techniques should increase the credits of an IV&V team.

The depth and coverage of analysis performed by an IV&V agent may vary depending on resources available, and the criticality of the software being developed. A lightweight or partial specifications analysis on incomplete specifications is very valuable [23].

5.2.3 Methods and tools in IV&V process:

An important aspect of IV&V work is to choose the right methods and tools. Ideally, an IV&V contractor will have access to all the tools and resources used by the development contractor, including the ability to share all project databases. However, in order to address any gaps or weaknesses in the coverage of the developer's scope, the IV&V contractor also needs additional methods and tools. These additional tools need to complement the developer's tools, so that interoperability does not become a problem. The use of these additional tools is an important factor in ensuring that IV&V is truly independent.

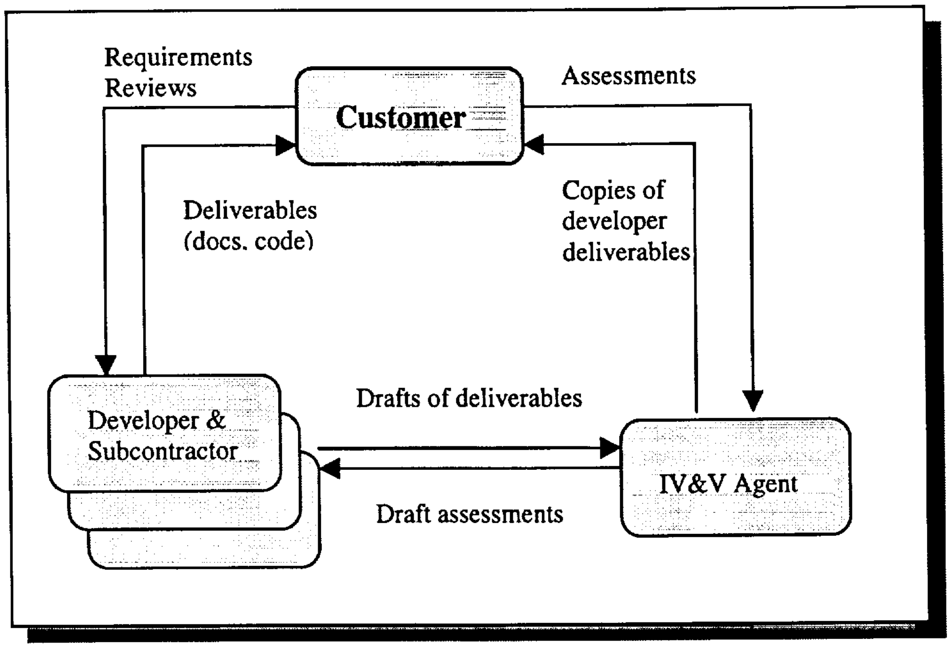


Figure 1: The relationships between IV&V agent, developer and customer.

After the tools and methods used by IV&V team are proved valuable in practice, they are often adopted by developer team. If the developer team knows that their product will be analyzed in a particular way, naturally they get interested in performing the same analysis themselves before releasing their products. IV&V team has more scope for experimenting with new techniques than the developer team [1], because IV&V team is out of the critical path for the software development effort. Hence, in some ways IV&V team plays a role to prove and test new techniques, as well as an agent of process improvement.

The tools and methods employed by IV&V team examining a number of properties of requirements specifications, including:

- Clarity (readability) - Are the semantics of the requirements defined clearly enough?
- Consistency-are they free of contradiction?
- Traceability- do requirements trace properly to higher level requirements, to the design, to test cases?
- Correctness – is the system built consistent with its specification?
- Accuracy - do requirements correctly describe the needs for the customer?
- Completeness- are all the modes and actions of requirements covered?
- Testability- is there a way to verify whether these requirements are met in the implementation?

WHERE project focuses on inconsistency detection and management. For existing tools, consistency analysis is conducted by translating the objects in requirements into a suitable logic entity, encoding the consistency relationships between objects as theorems, and attempting to prove these theorems hold. Unfortunately, this approach involves a non-trivial effort to translate the objects into logic, and failure of a proof does not necessarily help to precisely characterize the inconsistency [23].

The facing target of WHERE project is requirements specifications for NASA projects, let us see what these specifications look like.

5.2.4 The Documentation Problem

Most of the requirements specifications are written in natural language, and follow certain convention. For NASA space station specifications, various graphical, tabular, textual, or mathematical notations are involved. But, just few of them can be annotated by using existing CASE tools. Further more, existing CASE tools have some drawbacks to make them hard fitting in with NASA requirements specifications. For example, there is no any CASE tool we found, which has facility to check inter-notations consistencies. For example, the consistency between Entity Relationship Diagram and State Transition Diagram; the consistency between two different diagrams in same notation. These features are imperative in distributed group collaboration projects.

Our approach uses graph grammar as the framework to detect consistency in requirements. This approach has the advantage of removing the need for translation into logic, and make easier to trace inconsistencies when they occur. In next several sections, the applicability of our framework based on graph grammar to International Space Station Specification is described.

5.3 Scenario: International Space Station Requirements Specification

This scenario is taken from International space station Alpha program vehicle management description document. Station modes and state were developed to support the basic mission objectives of payload support, assembly, reboost, external vehicle rendezvous and departure, resupply, and maintenance. The valid Space Station Mode and state transitions requirements specification is expressed as two different notations for clarity; state transition diagram and state transition table. Figure 2 shows the state transition diagram; while Figure 3 shows the state transition table. In Figure 3, 'M' stands for Operator commanded mode transition; "MA" represents Operator commanded plus automatic mode transition; '-' stands for Not applicable; blank cell stands for Not allowed. In next section we revisit our approach for inconsistency checking.

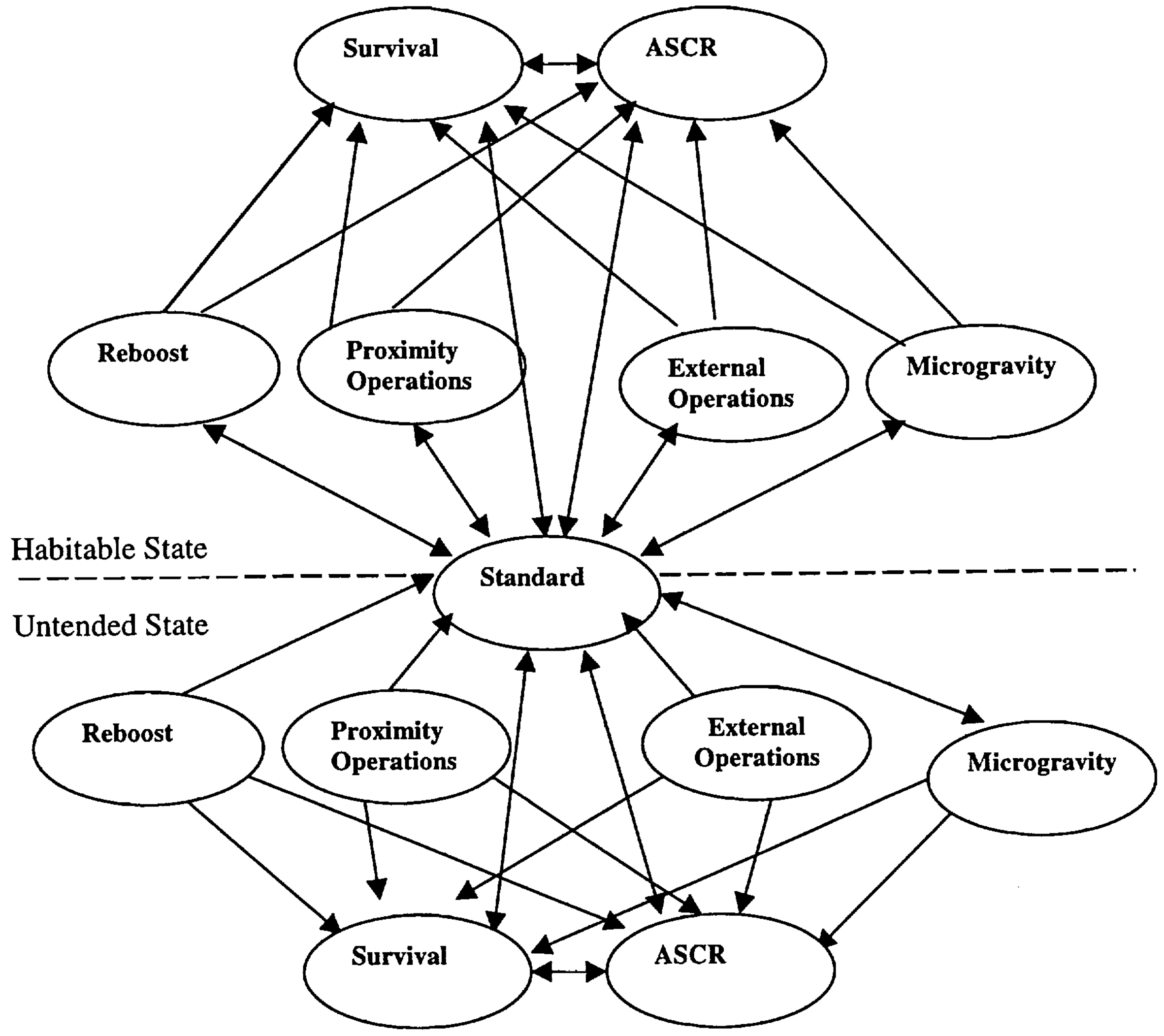


Figure 2. The valid Space station Modes and state transitions requirements for space station specification in State Transition Diagram.

To Mode ← / From Mode ↓		Perform Habitable Mission State							Perform Untended Mission State						
		Standard	Reboost	Microgravity	Survival	Proximity Ops	ASCR	External Ops	Standard	Reboost	Microgravity	Survival	Proximity Ops	ASCR	External Ops
Perform Habitable	Standard	-	M	M A	M	M	M	M	M						
	Reboost	M	-												
	Microgravity	M		-											
	Survival	M A	M A	M A	-	M A	M A	M A							
	Proximity Ops	M				-									
	ASCR	M	M	M	M	M	-	M							
	External Ops	M						-							
Perform Untended	Standard	M							-	M	M A	M	M	M	M
	Reboost								M	-					
	Microgravity								M		-				
	Survival								M	M A	M A	-	M A	M A	M A
	Proximity Ops								M				-		
	ASCR								M	M	M	M	M	-	M
	External Ops								M						-

Figure 3. The valid Space station Mode and state transitions requirements for space station specification in State Transition Table.

5.4 Recapitulation of Inconsistency Checking Approach

In last chapter we described our approach to inconsistency detection and management. In the inconsistency checking for two pieces of requirements specification, first requirements documents are translated into two graph systems. A set of functions for graph mapping is defined by method user. Presumably, user defined the correct mapping functions. If a homomorphism for two graphs is found, at this instance, the two pieces of specification is consistency. If a homomorphism is not found, the two pieces of requirements are conflict. Figure 4 shows how graph match is used to detect structural correspondences. The extraction of homomorphism also reduces the number of relationships between objects to be tracked or managed. Then method users may put more effort to track the relationships outside the homomorphism domain. In this case study, we omit the inconsistency handling actions. At this stage, we are still exploring the meta-model for inconsistency handling. Next section shows slides for the application of our inconsistency-checking engine on the NASA space station specification.

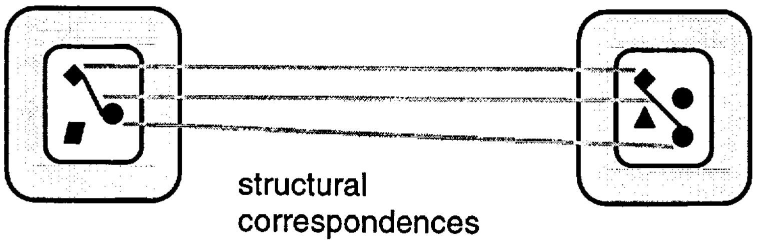


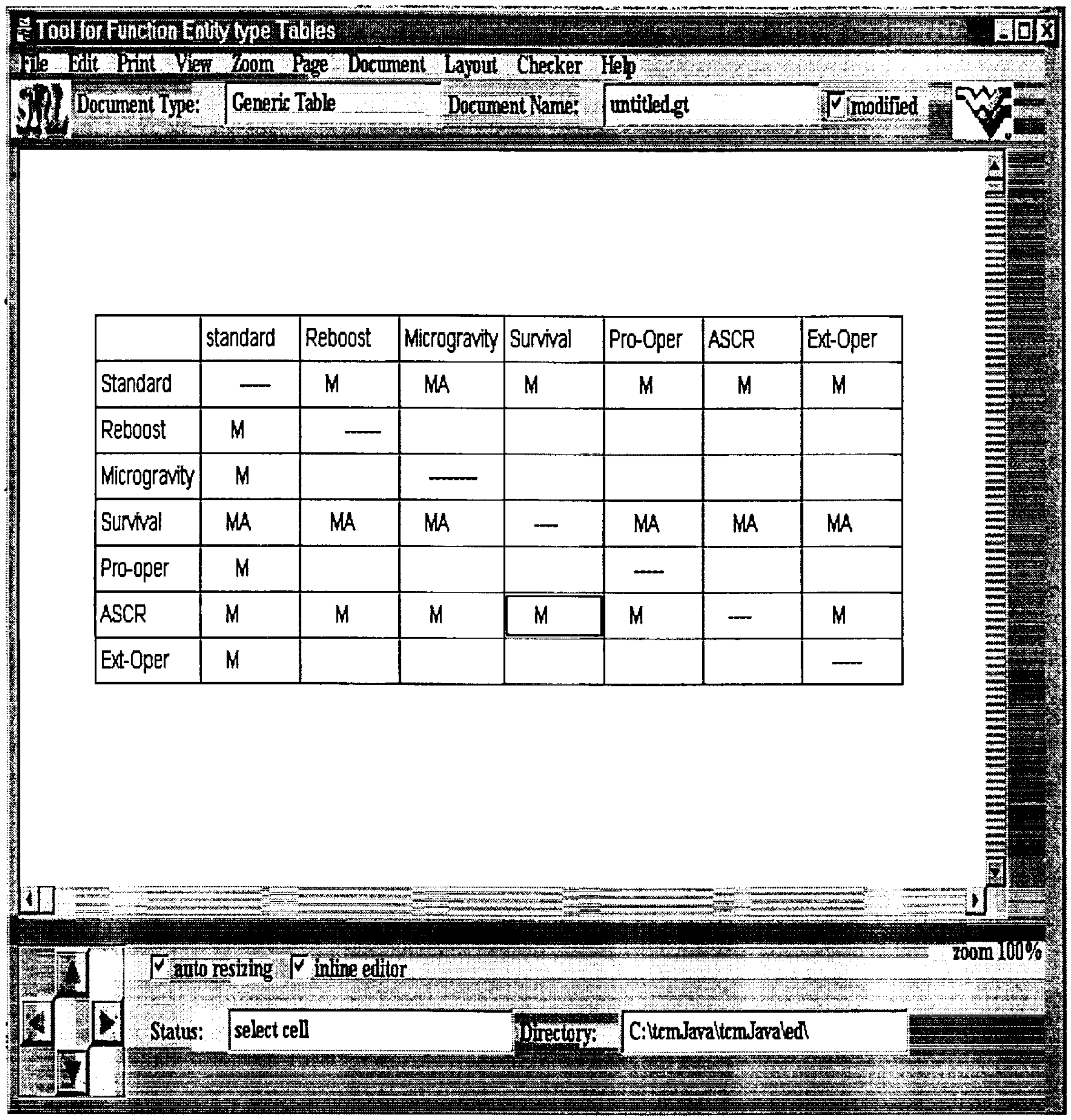
Figure 4. The application graph matching to detect structural correspondences

5.5 The Applicability of Our Approach.

This section describes how IV&V term uses WHERE project to detect inconsistencies in a real requirements specification, and how method user defines mapping functions in graph match. Presuming that developers already developed all of

the requirements specification using our WHERE tool. All documents are store in databases, and shared by Customer (NASA), developer term (Contractor), and IV&V term. IV&V term helps developer term to find error, missing information. In our story, IV&V agent Bob opens the state transition table that was created by developer Anne, and creates a new ViewPoint. He finds that a state transition diagram also expresses the same piece specification. Redundancies at this circumstance helps clarification of specification, avoids losing information. The consistency checking between overlap assures the clarity and correctness of specification. Bob decides to use WHERE inconsistency checking engine to detect the potential inconsistencies. The process is described as following.
Method user Bob uses WHERE project editing toolkit (TCMJAVA) to open the requirements specification in state transition table, and wants to check consistency with the state transition diagram. Figure 5 and Figure 6 show the loaded diagram and table using WHERE editing toolkit. WHERE supplies a series of interactive GUI to ask method user to input user defined information. After Bob inputs the name of the ViewPoints, WHERE inconsistency-checking engine extracts some key information about two ViewPoints (here is the two graphical notations, STD, STT) as Figure 7 shown. Figure 8 shows how method users define mapping functions.

In this CASE study, STD has just one type of node Ellipse and two types of transition edges, double-arrow and arrow to express the direction of the transition. The edge labeling is an empty set in STD. STT has one type of node Cell, one kind of edge type Cell, two kind of edge labeling 'M' and "MA". Method user can define mapping functions using Choice box as shown in Figure 8. The seeking of mapping functions may be a heuristic process.


Tool for Function Entity type Tables
File Edit Print View Zoom Page Document Layout Checker Help
zoom 100%

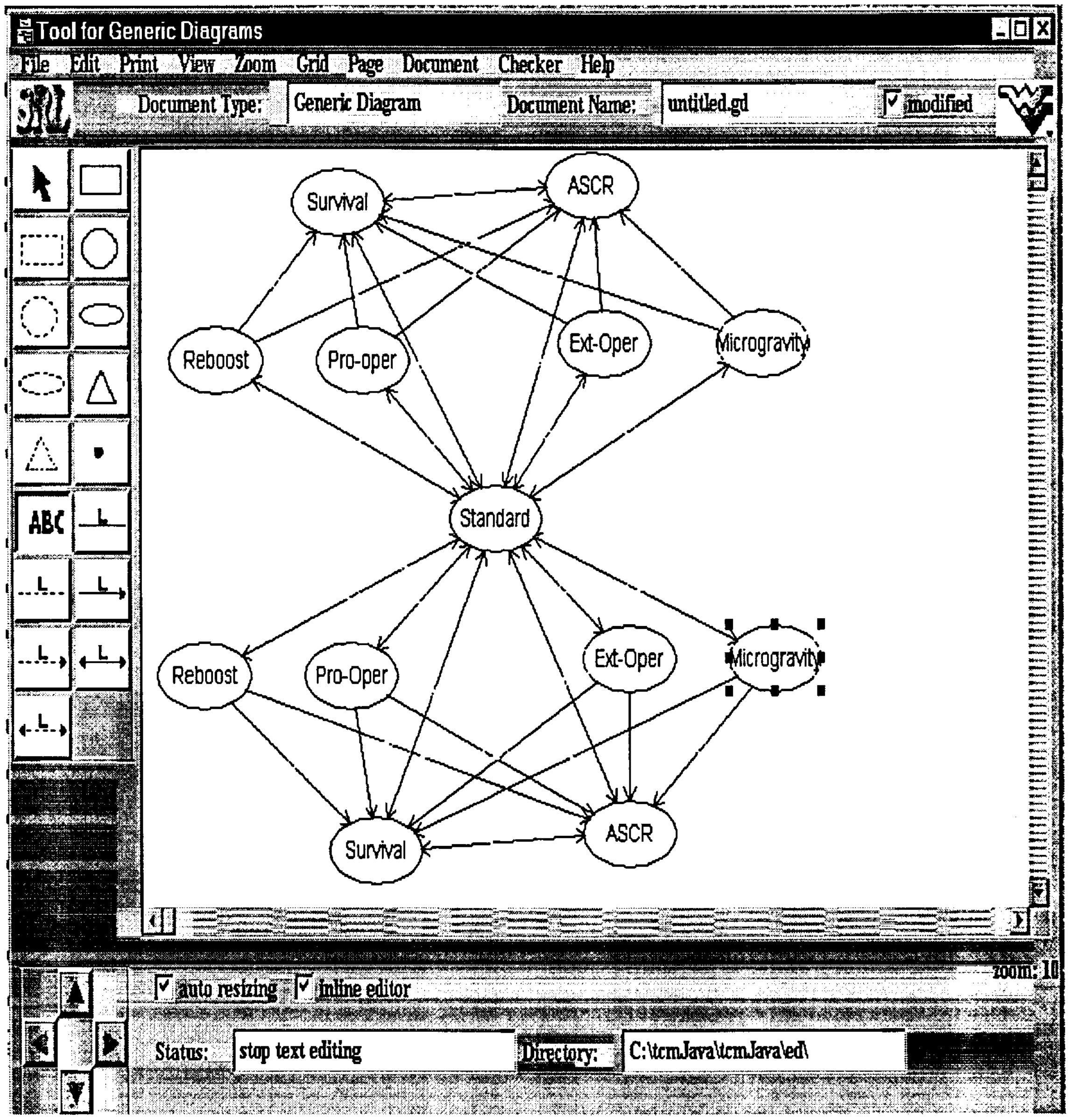
Document Type: Generic Table Document Name: untitled.gt [✓] modified

	standard	Reboost	Microgravity	Survival	Pro-Oper	ASCR	Ext-Oper
Standard	—	M	MA	M	M	M	M
Reboost	M	—					
Microgravity	M		—				
Survival	MA	MA	MA	—	MA	MA	MA
Pro-oper	M				—		
ASCR	M	M	M	M	M	—	M
Ext-Oper	M						—

[✓] auto resizing [✓] inline editor

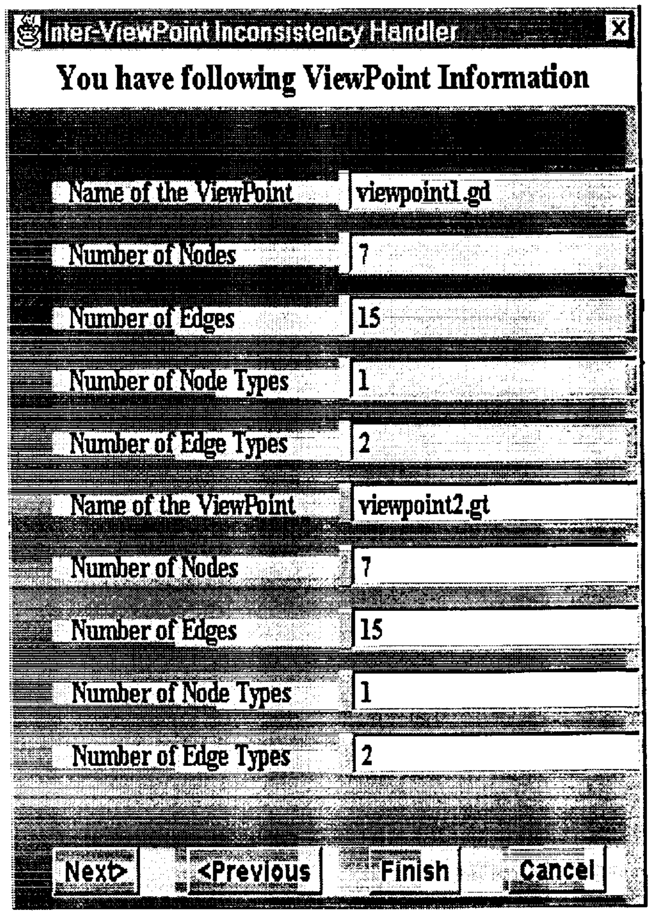
Status: select cell Directory: C:\tcmJava\tcmJava\ed\

Figure 5. The State Transition Table for Allowable Space Station Mode Transitions from NASA Space Station Specification in TCMJAVA.



Document Type: Generic Diagram
Document Name: untitled.gd
✓ modified
✓ auto resizing ✓ inline editor
Status: stop text editing
Directory: C:\tcmJava\tcmJava\ed\

Figure 6. The State Transition Diagram for Allowable Space Station Mode Transitions from NASA Space Station Specification in TCMJAVA.

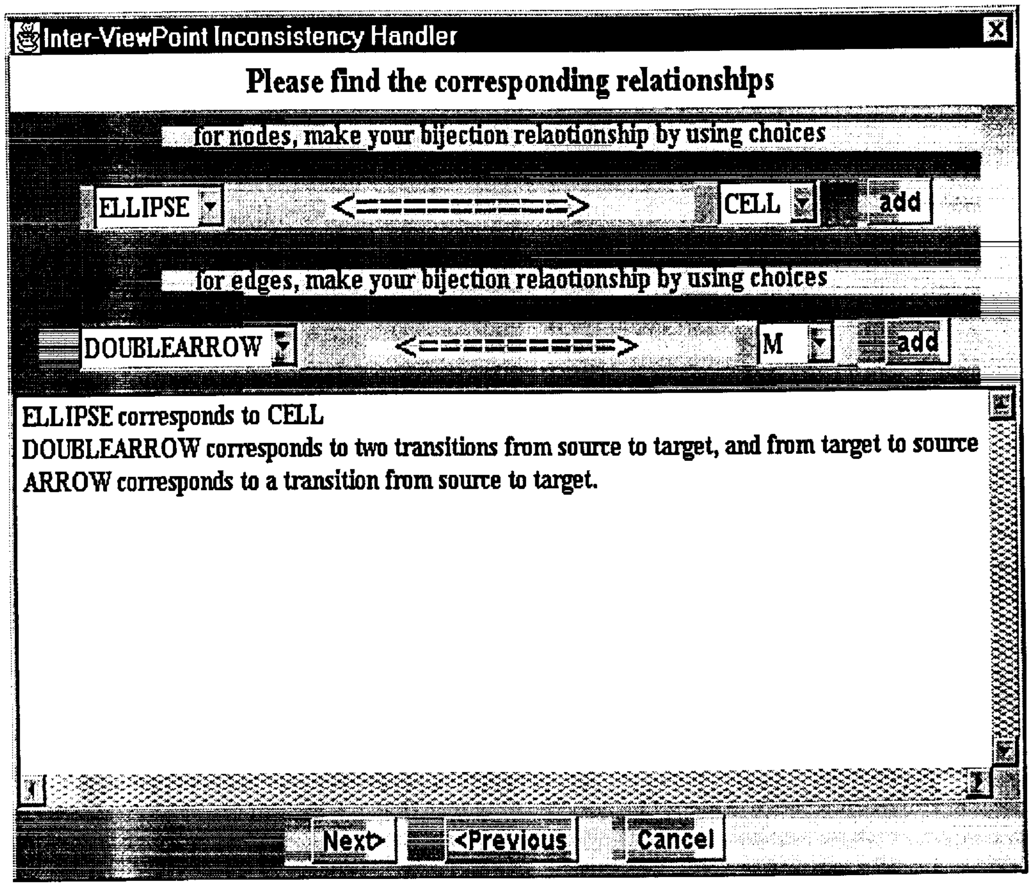


Inter-ViewPoint Inconsistency Handler

You have following ViewPoint Information

Name of the ViewPoint	viewpoint1.gd
Number of Nodes	7
Number of Edges	15
Number of Node Types	1
Number of Edge Types	2
Name of the ViewPoint	viewpoint2.gt
Number of Nodes	7
Number of Edges	15
Number of Node Types	1
Number of Edge Types	2

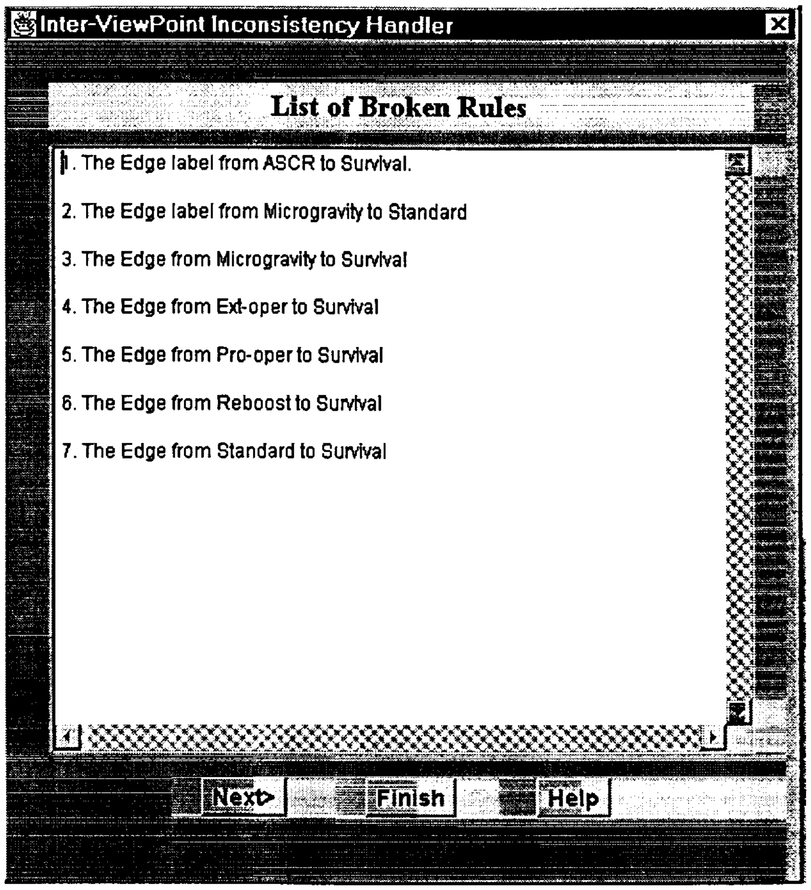
Figure 7. The import attributes summary from two ViewPoints in Comparison.


Inter-ViewPoint Inconsistency Handler
Please find the corresponding relationships
for nodes, make your bijection relaotionship by using choices
ELLIPSE
<==========>
CELL
add
for edges, make your bijection relaotionship by using choices
DOUBLEARROW
<=========>
M
add
Next>
<Previous
Cancel

ELLIPSE corresponds to CELL
DOUBLEARROW corresponds to two transitions from source to target, and from target to source
ARROW corresponds to a transition from source to target.

Figure 8. The interactive environment to define mapping functions for graph match.

After graph match mapping functions are defined, WHERE inconsistency checking engine tries to match the two ViewPoints in different graphical notations. If a homomorphism is found, then these ViewPoints are consistency. Otherwise, the difference between two graphs is shown in the Inconsistency information display window as Figure 9 shown.



Inter-ViewPoint Inconsistency Handler

List of Broken Rules

1. The Edge label from ASCR to Survival.
2. The Edge label from Microgravity to Standard
3. The Edge from Microgravity to Survival
4. The Edge from Ext-oper to Survival
5. The Edge from Pro-oper to Survival
6. The Edge from Reboost to Survival
7. The Edge from Standard to Survival

Figure 9. The inconsistency information display window.

Surprisingly, in our CASE study, the valid space station mode and state transitions requirements in different graphical notation is not consistent. There are seven places are conflict. From another aspect of view, some information in STT is missing in STD. The end of the walking-through leads to the last section of this Chapter, summary.

5.6 Recapitulation

In order to assess the applicability of the framework described in this thesis, we conducted a case study that coves a real, large software project that is most likely to have inconsistencies in its life cycle. This is a good case study, because ISS project is a complex project with participants having different languages and opinions. From the aspect of IV&V term, our framework is applied to detected inconsistencies. The study indicated our framework helps inconsistency detection in the tackling of inconsistency management problems.

Our CASE study is drawn from the Valid Space Station Mode and State Transition requirements for space station specification, represented by a state transition diagram and a state transition table. The method allows us to treat each notation as a separate ViewPoint, the union of the ViewPoints clearly describes all of the modes and state transitions. Such a redundant description is powerful tool to clarify any ambiguities in a requirements specification, especially in a critical section the prevention of any ambiguities is imperative comparing with the trade off redundancy.

We have implemented a prototype tool to check the inter-ViewPoint consistency. We consider two different editors as the frame of two different ViewPoints. The consistency checking between two different notations in TCM method is a kind of activity of checking the consistency of different notations. Our implementation is based on the framework of graph grammar. Method user defines the mapping functions between two graph systems. Then homomorphism seeking is performed. If a homomorphism is found, the two partial specifications are consistent. Otherwise, inconsistencies are shown to tell method user the source of inconsistency. At this stage, how to accomplish inconsistency handling still requires further work. We plan to incorporate many of the conflict resolution strategies and actions within the WHERE, while tolerating inconsistency.

In order to abide by the rule, requirements engineering tools should maximize the ability of automation. We also implemented the parts of the automatic generation of a state transition diagram from a state transition table, and vise versa.

We presented our approach in last two chapters, the implementation of our inconsistency checker is based on TCMJava. In next chapter, we describe the process of porting TCM (Toolkit Conceptual Modeling, from Vrije University at Amsterdam, The Netherlands) project from C++ to Java, and make it being our WHERE project ViewPoints editor.

Chapter Six The WHERE Editing Toolkit: TCMJAVA

This Chapter describes the process of translating TCM (Toolkit for conceptual modeling) project from C++ to Java. The Java port of TCM is called TCMJava. This work was conducted jointly by Zhong Zhang and Swarn D. Dhaliwal. In the initial translating process, C++ files were translated by term members at will. At last stage, Zhong Zhang finished the integration work on TCMJava. This chapter summarizes the experiences we leaned from this translating process. First, in section 6.1, from a broad perspective we describe the architecture of the WHERE project, and in section 6.2 we explain why we choose Java as implementation language. Later, In sections 6.3-6.6, we focus on the difficulties encountered rather than a detail description of the difference of C++ with Java. Also, we talk about the design of TCM project. In future work description, we delineate that the conversion of TCMJAVA to JavaBeans increases the reusability of the TCMJava. We already discussed the goals of the WHERE project, corresponding to these goals the architecture of the WHERE project has been established. In next section, we give a brief description of the architecture of the WHERE project.

6.1 The Architecture of the WHERE Project.

From the functionality aspects, the proposed architecture of WHERE comprises three parts [1], the ViewPoint editor, the ViewPoint reviewer, and the consistency checker. TCMJava provides the basis for the ViewPoint editor, whilst the ViewPoint reviewer and consistency checker are not built yet.

The ViewPoint Editor

The initial design of this tool includes the functionality for editing graphical, tabular, textual and mathematical notations. It is composed of a set of editors. Each editor provides basic syntax and type checking. One editor is not necessary to be able to generate an entire specification. Usually one piece of specification can be broken down into individual ViewPoints. For example, if a specification method calls for three different notations, each notation represents a different ViewPoint, and three different ViewPoint editors are needed, one for each notation.

The ViewPoint Reviewer

This tool allows a user to browser and annotate ViewPoints created by other people. If a ViewPoint to be loaded was developed outside the WHERE environment, the Reviewer provides a very rudimentary parsing of the ViewPoint, so that annotations can be attached to different parts of its structure. The Reviewer does not allow a user to edit the ViewPoint, but it stores any annotations and meta-ViewPoint data as a separate 'view' of the ViewPoint.

The Consistency Checker

This tool defines the relationships between one ViewPoint (source ViewPoint) and other ViewPoints that are included in the developing specification. Each ViewPoint can have three types of relationship with other ViewPoints: method-defined (defined by method engineers), method user-defined, and automatic. The method-defined relationships are laid down as part of the method, and define the relationships that should hold between two ViewPoints of a particular type. User-defined relationships are entered by users in order to record and track non-standard relationships between particular ViewPoints. Automatic relationships are recorded as the result of certain actions on ViewPoints, where the action reveals that a relationship must exist. The consistency checker contains rules for checking the integrity of these relationships.

We decided that use Java networking programming language to implement the WHERE project, because the WHERE supports the group communications and coordination in a large-scale composite system development process. Other than that, we have following good reasons to choose Java as the WHERE implementation language.

6.2 The Implementation of the WHERE Project.

Java as a powerful networking programming language, has being extensively used by many industry companies and academic institutes. With the increasing functionality and applause form industry, Java provides solutions for distributing networking environment computing. In the next year, networking personal computer which will be infused with JavaOS and will based on 100% pure Java will give networking computing market a strike.

The two current networking computing competitors, Sun Microsystems and Microsoft supply different strategies and solutions for networking computing. Corresponding to Microsoft's MFC, ActiveX, DCOM, Sun released JFC, Java Beans, and RMI respectively. These two solutions use different methodologies. It is difficult to judge which one is better. The decision depends on what kind of system under development, what kind of platform the system developed will be used and so on. Although Microsoft solution has a lot advantages over Java solution currently, Java solution distinguishes itself from two aspects, and those two aspects are very difficult for Microsoft solution to catch up. One characteristics of Java is platform neutral, with the application JavaOne Java can be programming once, runs everywhere. The other distinguished characteristic of Java is security empowered. The Java security model includes intrinsic security that protects the user from errors that can result from incorrect type casting or illegal memory accesses. This provides both safety and security. In addition, a Security Manager provides resource-level security that restricts a Java program's access to the disk, to the network, and so forth. As an enterprise computing solution, Java is favored by more and more developers.

The WHERE project faces the communication and collaboration problems of distributed projects. Security and platform independent are the main concerns in this project. Java networking solution will help us at these points with the trade off speed. Therefore the WHERE project will be implemented in 100% pure Java. In next section we introduce TCM project, which is the original port of the WHERE project editing tool TCMJAVA.

6.3 Introduction of the TCM Project

6.3.1 What is the TCM Project?

The TCM project is a suite of editing tools and it aims to support the conceptualizations of the software product in requirements definition and design phase [5]. TCM stands for "Toolkit for Conceptual Modeling". The functionality includes graphical and tabular editors to represent visually the different views of product requirements and product designs. TCM toolkit is composed of 16 editing tools, and implemented in Unix C++ and X/Motif. The hierarchy is showed in Figure 1. TCM project was developed at the Vrije Universiteit at Amsterdam. For use in the WHERE project, the author and his team member Swarn D. Dhaliwal translated the original TCM from C++ into Java programming language. Even though the design of GUI parts was changed considerably in the translating process, the design of application parts was not changed. The design of TCMJava is described in next section.

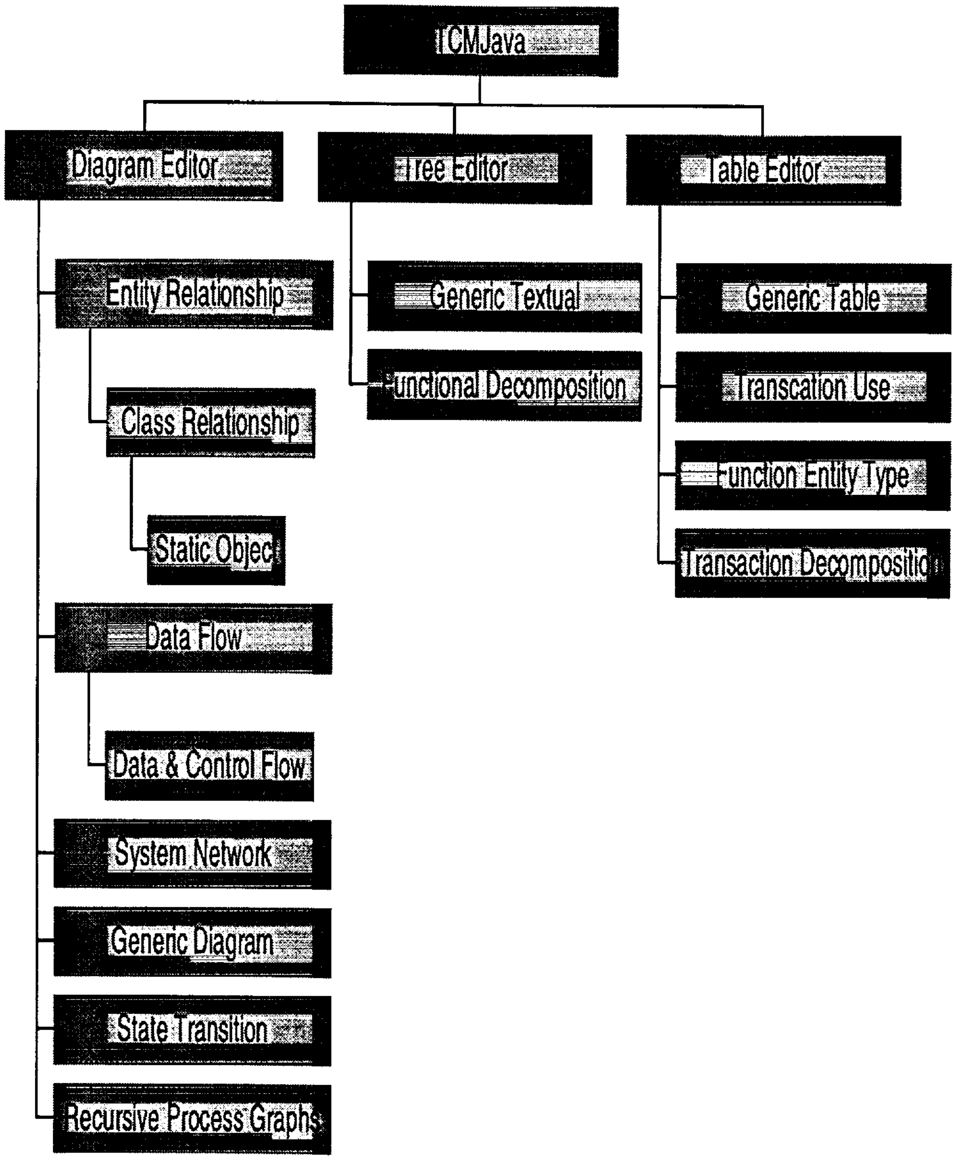


Figure 1. The functional decomposition of TCMJava

6.3.2 The Design of TCMJava

In this section, we discuss the design of TCM project. We try to decompose TCMJava into some subsystems. This decomposition is different with original TCM project because of the different language features of Java with C++. But for the classes except those about GUI classes, such as Shape, Document, we abide by the original design.

We can first decompose TCMJava into Subject area and Graphical User Interface Area, as shown in Figure 2. Note the decomposition does not stand for the inheritance relationships. Also, the organization of source code is shown in Figure 3.

The source code of TCMJava is physical split over several subdirectories, and each subdirectory forms a package. This is done according to the following criteria:

- All code that is global and that is not part of the other areas is collected in the directory gl. This directory is complied into a package called tcmJava.gl. This includes classes like List, string, and some utility classes.
- All code that lies in the editor area but not in one of the subareas, is collected in the directory ed. this code is bundled into a package called tcmJava.ed. This includes classes that are used in all editors and startup tool such as, MainWindow, EditWindow, Viewer, DrawingArea, Grafport, and Startup classes Tcm and TcmDialog.

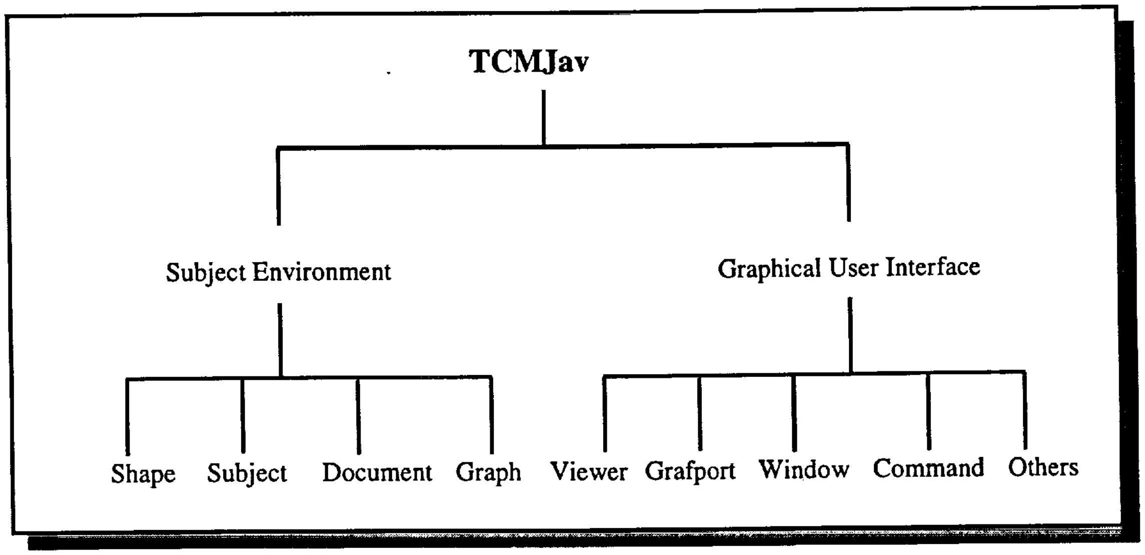


Figure 2. The decomposition of TCMJava into Subsystems.

- All code that lies in the diagram area and that is not part of a specific diagram subareas, is collected in directory **dg**. This code is bundled into a package called tcmJava.dg. This includes classes like Graph, Node, Edge, different graphical shapes and most of the diagram edit commands. This package is used by any diagram editor.
- All code that lies in the table area is collected in directory tb. This code is bundled into a package called tcmJava.tb. This includes classes like TableWindow, TableViewer and Table. Also, most of the tool specific constraints are implemented in these specific table classes.
- All code that is specific for the different diagram editors, is collected in the directory **sd**. This code is bundled a package called tcmJava.sd. Each specific diagram editor is a distinct executable.
- All images that are used in ImageBuuton and logo. These GIF files are converted from bitmaps inherited from original TCM project.

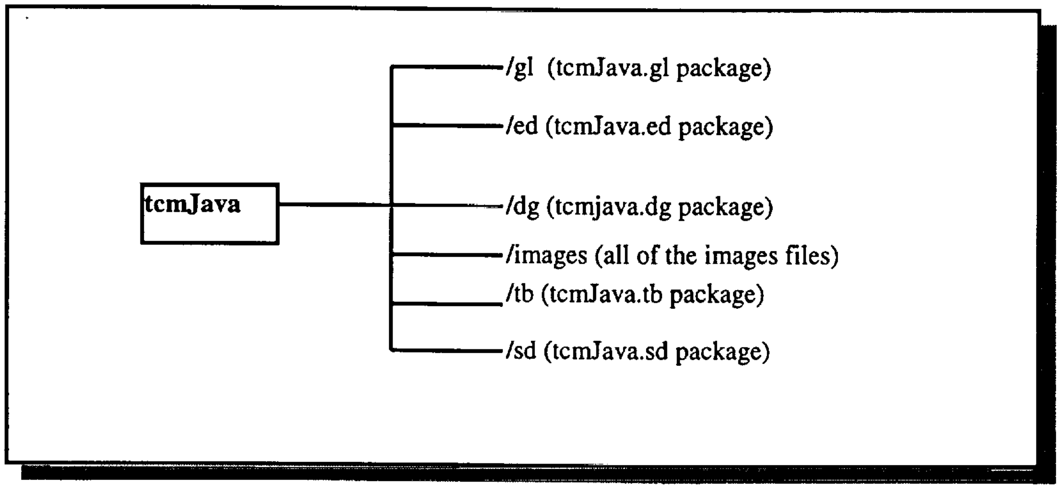


Figure 3. The source organization illusration.

A complete and accurate overview of all classes and their attributes, actions and specialization relationships can be generated automatically by using Javadoc as explained at web site:

http://atlantis.ivv.nasa.gov/projects/WHERE/TCMJava

Overall, the TCM project is well designed. Moreover, the design of TCM intends to avoid some pitfalls of C++ programming language.

e inheritance is avoided. We will discuss the drawbacks of multiple ınce in next section.

ıe place used template. Also the translation problem of template will be ed in next section.

ıext section we describe some differences between C++ and Java programming Although the birth of Java benefits a lot from C++, still the differences should

Difference Between C++ and Java

-+ is a large language that is growing larger with its standardization. Part of this the fact that C++ has grown being extensions from C into being a whole anguage. The draft standard just released by the C++ standards committee has standard library leap from just Isotreams and the C standard library into ten ibraries. In a word, the C++ standard library is now huge.

ꞓing large is not a big problem, especially in the sense of library routines. , the C++ language has also grown increasingly complex. Features like operator ng, multiple inheritance, and templates can generate very complex code that can enhanced or maintained by the person who created it. If that person leaves , the code becomes particularly difficult to maintain.

astly, not only did C++ inherit certain flaws and dangers from C, such as the ssor and pointers, but added its own dangers and redundancies such as multiple ce and references. And all the features to be discussed here do not cover all the ures and ambiguities that are not included in TCM.

he following C++ features were intentionally omitted from Java.

eatures Omitted from C++ in Java

structors. The Java language has changed the concept of an object's destruction its finalization. The reason for this is that object destruction is most often sociated with the explicit freeing of dynamic memory allocated for the object.

a pointer. Much of the requirements of a pointer can be fulfilled with uninstantiated object. Java has no way of implementing two current pointer functions:

1. Pass by reference. Although arrays and objects are passed by reference, a local copy is made in the called function. The need for this is minimized through the use of objects and object-oriented design whereby most methods act on the objects data members and do not need to modify variables in the caller. If it is necessary to modify a caller's variable, the method's return value can be used.
2. Function pointer. Although very useful as callbacks in GUI functions and in parser dispatch tables, some of the benefit of function pointers can be gained through using virtual functions and inheritance. As for the function dispatcher, this can be achieved by using an uninstantiated variable of class Object and run-time type identification.

C++ pointer	**Java references**
Point *point;	Point point;
point = new point();	point = new Point();
point→x = 0;	point.x =0;
*point = *other;	point = new point(other);
Point point;	Point point = new Point();

- References. The reference is a feature that has two purposes: first, to eliminate the need for pointers, and second, to make operator overloading work. Considering the fact that good reasons have been put forth for eliminating both pointers and operator overloading, reference become clearly unnecessary.
- Friend Classes. A feature that allows one class to have access to protected and private data members and methods of another class. This feature is implemented through packages. In a Java package, when you do not explicitly put in an access

specifier, the default access specifier is "friendly", which means that all classes and methods within the package have access to friendly methods and data.

- Sizeof in C, a compile-time operator that returned the size, in bytes, of its argument (a data type or variable). Since the Java language precisely specifies the size of all basic types, there is no need for the sizeof operator.
- extern in C. A data type modifier that tells the compiler that the definition of the global variable or function is found in another compilation unit. In Java, there are no global variables and all external classes are loaded dynamically by the runtime interpreter. In TCMJava we collected some extern variables, and put them in one class file (GlobalName.java) so that all of the classes can access them.

```
public final class GlobalName extends Object{

 public static Vector theHelperVector = new Vector(20);    // Type of Helper
 public static Helper theHelper;
 ......
```

- The C preprocessor processes a C source file before it is passed to the compiler. It generally provides two capabilities: text replacement and conditional compilation. It is important to understand that the functions the preprocessor performs have been criticized as not truly being benefits at all [30]. The primary problem with the a preprocessor can be summed up with the expression, "seeing is not believing." That means that C source code can not be trusted until all header files have been read and all macros expanded. This chore becomes impractical in very large programming projects. The use of interface can partially function as C preprocessor.
- In C++, a virtual function in a base class is dynamically bound at run time based on the pointer value. If the pointer value points to a subclass, the subclass function with the same name will be called. If the function was not declared virtual, no run time checking of the type would occur and whatever type the pointer was declared (i.e., the base class) that is the function that would be run. In Java, all functions of the same name in both the base and derived classes are virtual.

6.4.2 Java New Features

- Basic Types in Java. The treatment of basic types is special in Java, and compiler and runtime system (i.e. VM), are aware of these special types. Variables of type int can not be passed to a method expecting a real object, such as when creating a hash table, where the key and contents are both subclasses of java.lang.Object. In order to allow users to create hash tables with a key based on an integer value, classes like java.lang.Integer have been invented. By definition, *all objects are passed by reference.* Methods that receive an object instance can change its state, assuming appropriate methods are defined. In contrast, *variables of basic types, like int, are passed by value.* In TCMJava, in order to perform the same functionality in C passing by reference (int &), we made a wrapper class which is similar to Integer class, but much simpler.

```
public final class IntRef extends Object{
 private int value;
 public IntRef(){
  this.value = 0;
 }
 public IntRef(int value){
  this.value = value;
 }
 ......
}
In other classes, for example, in Box class

public point GiveSnp(point to, int order, int count, IntRef code) {......
```

- An interface is a list of static variables and method signatures. Classes that implement a given signature basically promise to implement each of these methods. The compiler checks to see if these methods are really implemented in the class definition. Interfaces allow the Java developer to inherit a specification, yet no implemented behavior is inherited. An interface works as a type in the language. For instance, an interface could be used as the type of formal parameter in a given method. In TCMJava we used several interfaces to define constants, and classes that use those constants in an interface implement the interface. In such way the binding of the constants with class files can be finished in compilation time rather than runtime.

```
public abstract interface IconCodes{
 //

 public static final int BOX = 0, ELLIPSE = 1, DOT = 2, COMMENT = 3, ......

}

public class GDIcons extends Icons implements IconCodes {

/* BOX can be directly used rather than using IconCodes.BOX */
}
```

- In Java, a package is primarily used for managing namespaces; however, Java does allow default "friendliness" among all classes inside a package. A Java package can be thought of as a container for related classes and interfaces.
- Communicating through static variables. When classes are loaded into a VM, all the class variables are initialized. When a second application (editor) wants to load the same class, and the VM determines that the class is already loaded, or is still cached in memory, it will not reload the class and also not initialize the class variables. This causes applications that have nothing to do with one another to share the same static variables. In TCMJava, each editor uses several global variables to communicate among classes. If those variables can not be restored at some stages later, when the second editor is activated, the first editor dies, because the global variables of first editor such as theDocoment, theHelper, thePrinter, were overwritten by the second editor. In order to get around the effect, we used separate Vectors to store each global variable, and assign every editor opened a unique number, in this way we can restore global variables successfully.

6.4.3 System Information

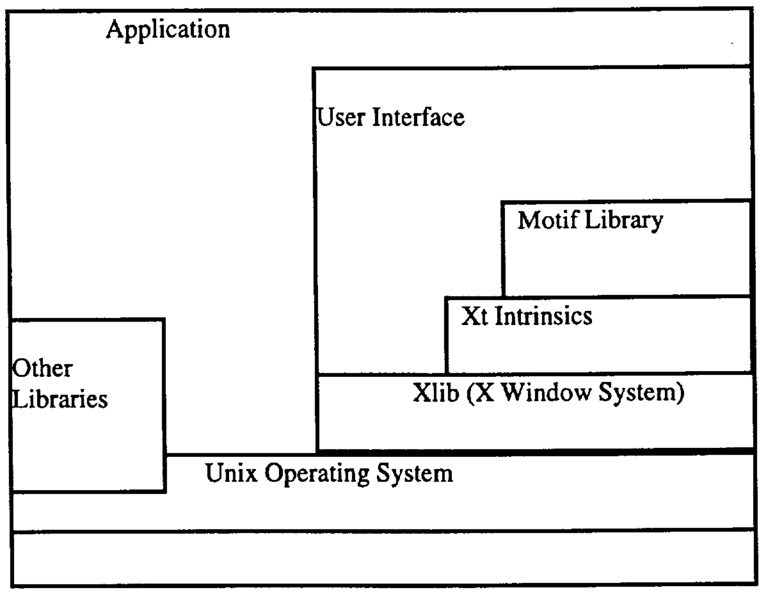
This category classes provides both information form the operating system and access to some operating system services. This is important to realize platform-independent.

- Classloader. This is an abstract class that can be extended to allow the loading of classes either from a file or over a network. This mechanism will allow true dynamic distribution of objects to any machine on the network that has a Java run time. We use Classloader class to load the images for logos and imagebuttons.
- System properties can be obtained by using System.getProperty(key) function in System class. key can be user.name, user.dir, os.name, and etc.

The implementation of GUI part of TCMJAVA and original TCM is complete different. In next section, we describe the some language aspects of X Motif and Java AWT for GUI implementation.

6.5 X Window / Motif and Java AWT Package

The original TCM X/Motif user interface is restricted to that part of the source (classes and functions) that directly uses the Motif, Xt and Xlib libraries. The typical X/Motif application structure is shown in Figure 4.



The Java AWT uses three concepts to implement common functionality/platform-unique look and feel: **abstract objects**, **toolkits**, and **peers**. Every GUI element supported by the AWT package has a class. The AWT GUI objects are platform-independent abstractions of a native GUI, just as Java bytecodes are platform independent assembly language instructions for a virtual machine, AWT objects are platform-independent GUI elements for a virtual operating system display. The toolkit is the platform-specific implementation of all the GUI elements supported by the AWT. Each toolkit implements the platform-specific GUI elements by creating a GUI "peer". A peer is an individual platform-specific GUI element. Since every AWT GUI object is derived from the generic AWT object called a component, every component has a peer. The peer implements the platform specific behavior of the AWT component. The peer is added to the generic AWT object when the object is added to a container that has a peer. The fact is that peers are separate from the AWT object. An AWT object contains a peer. It was implemented using the idea of containment. The peer is not created until there is a physical screen representation (i.e. associated with the setVisible() method).

In the process of porting, because of the irrelevant relationships between X/Motif and Java AWT package, we had to isolate the parts about user interface from source code. Then implemented the GUI parts separately in Java. A lot of classes are related with GUI implementation in TCMJava, such as DrawingArea, DialogManager, EditWindow, and Menu. The X/Motif GUI implementation was ignored, and Java implementation has to be written from scratch.

The most difficult part in the translation process is event handling. Java uses event delegation model to solve the problem; while X Motif uses a lot of callback functions to solve the problem. In next section we describe those two kinds of event handling models.

6.6 Event Handling

Event Handling mechanisms in Java programming language is different with that in X Window Motif. In this section, we describe how C++/X Motif and Java handle events.

6.6.1 X Motif Event-Driven Solution

Programming graphical user interfaces introduced a new style of programming called "event-driven' programming. Although the terminology differs between X

Windows, MS Windows, and the Mac OS, the methodology for event-driven programming is the same. Event-driven programming describes a new paradigm for control flow of a program. In the event-driven paradigm, your program becomes a set of event handlers that are invoked by a user-triggered event. On Macintosh and MS Windows, an application programmer actually writes the small function to retrieve and dispatch events. This main function is the infamous "event loop". In X Windows and Java AWT, the event loop is hidden from you. In X Windows, an application programmer registers function pointers to event handling routines (called callbacks) and then calls a function called XtAppMainLoop(). In the Java AWT events are retrieved and dispatched by Java runtime. Let us see the TCM implementation in more details.

- Create the Widgets.
- Register the event handling for these widgets.
- Go into the main event loop (which is built-in, hidden).
- Activate the appropriate event handler when specific event occurs on a specific widget.
- Return to the main event loop when done.

When a TCM tool is started the widgets that form the main window are created. The class EditWindow and its specialization contain the functions to create the constituent parts of the main window. These widgets have one or more call-back functions. The Widget reacts to a certain set of X events (That set is built-in) and when such an event occurs a user-supplied action is called.

6.6.2 Event Delegation Model in Java

In Java event delegation model, events are generated by event sources. One or more listeners can register to be notified about events of a particular kind from a particular source. Since it allows the programmer to delegate authority for event handling to any object that implements the appropriate listener interface. The new AWT event model lets you both handle and generate AWT events. Event handlers can be instances of any class. As long as a class implements an event listener interface, its instances can handle events.

In our approach of porting, after the GUI parts have been implemented, they were registered for some specific event listeners. Then, in the classes registered corresponding events were handled.

6.7 Summary and Future Work

In this chapter, we discussed the difference of C++ and Java programming languages and the difficulties we encountered during the porting process. An approach of porting process is summarized as following in Figure 5. And the current version of TCMJava can be obtained from SRL web site at

http://research.ivv.nasa.gov/projects/WHERE/TCMJava/index.html

In the future, the WHERE project editing toolkit should be capable of dealing with various notations, including mathematical notations (e.g. PVS) and natural language notations. And user could check the consistencies between different notations. In current version of TCMJava, most editing functions are available, such as copy, cut, paste, zoom in, zoom out, save file, load file and more. However, in future version of TCMJava more features are expected as listed below.

- Multiple steps undo-redo functionality.
- User can customize the background color, and foreground color.
- User can change fonts.
- User can customize different colors for different shapes.
- Printing and Postscript support.

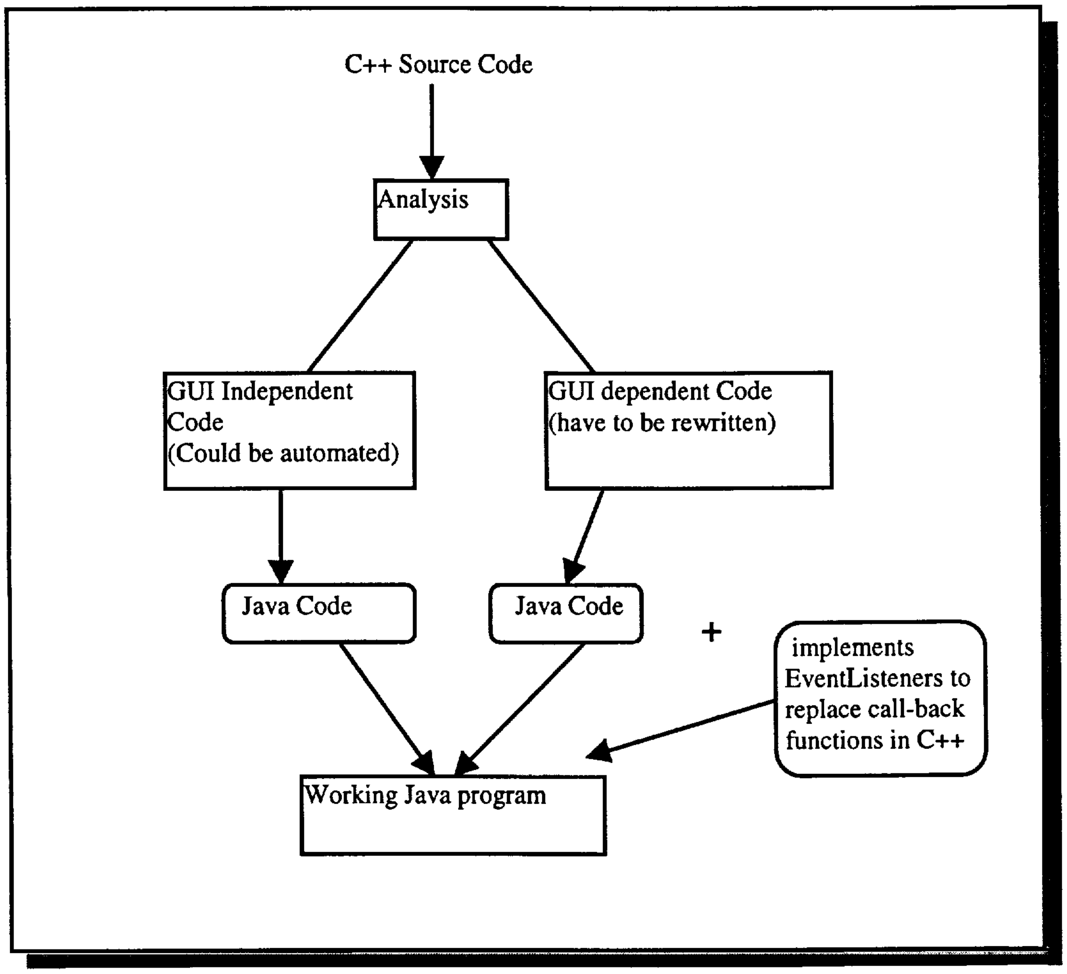

C++ Source Code
Analysis
GUI Independent Code (Could be automated)
GUI dependent Code (have to be rewritten)
Java Code
Java Code
+
implements EventListeners to replace call-back functions in C++
Working Java program

Figure 5. The porting process summary.

With the increasing power and stability of Java programming language, the implementation of these wishes could get more feasible. JavaBeans is a reuse component solution for Java programming language, the conversion of every editor of TCMJAVA into a JavaBeans, will improve the applicability and reusability of TCMJAVA. Other users can use drag and drop mechanism to use any of our editors in a jar file, and tailor with his (her) application to make a new application. The conversion each of 16 editors is also imperative.

In the next chapter we present a summary of the thesis and give an evaluation of its achievements. We discuss remaining problems which this thesis does not address, and outline future work.

Chapter Seven Conclusions and Future Work

In the translation process, we had to redesign and rebuild the GUI parts, event handling parts, and graphics enabling functions from scratch. GUI handling part of Java language is completely different with that in X Motif C++. Fortunately, several classes in the TCM such as shape, subject, document can be reused, still we had to change many places because of the difference of programming languages. We summarized our empirical approach for future projects that need translate C++ into Java in chapter four. A clean separation of GUI parts with underlying functionality implementation is essential. Tossing away the GUI implementation in C++, writing new coding from scratch. Even the coding about GUI parts can be finished beforehand of the translation, just from the appearance of the project translated. Java makes the connection between GUI and application coding easy. Java event delegation model is easy to grasp and simple to implement. The callback features in X motif C++ implementation can be replaced by static stubs functions which use various Java eventlistener classes in our port of TCMJAVA. The translation process of application code (GUI independent) might be automated.

Our translation effort succeeded with the release of beta version of TCMJAVA 1.4. It includes 16 graphical and tabular editors, with 10 diagram editors: "Generic Diagrams", "Entity Relationship Diagrams",

"Class Relationship Diagrams", "Static Object Diagrams",

"State Transition Diagrams", "Process Structure Diagrams",

"Recursive Process Graphs", "Data Flow Diagrams",

"Data and Control Flow Diagrams", "System Network Diagrams",

Four table editors: "Generic Tables", "Transaction Decomposition Tables",

"Transaction Use Tables", "Function Entity type Tables",

And two tree diagram editors: "Generic Textual Trees", "Function Decomposition Trees".

All 16 editors support basic editing functionality, such as copy, cut, paste; viewing functionality, such as zoom in, zoom out, home view; data saving/loading functionality, such as save file, load file, insert file; help facilities, such as help, document info; grid facilities, such as show grid, hide grid. Furthermore, TCMJAVA supply an attractive and user friendly GUI. For more details, please refer to web page at,

http://research.ivv.nasa.gov/projects/WHERE/TCMJava/index.html

The finishing of TCMJAVA gives us a chance to experiment our inconsistency management framework.

Our inconsistency management framework is based upon graph rewriting rules system, ViewPoints framework, and the internal data implementation of TCMJAVA. Graph is a universal data model to describe static properties of objects and the structure of their relationships. It is feasible to encode the diagrammatic or tabular objects in TCMJAVA into graph system, which has six attributes. Because the graph infrastructure is already there, the encoding process just is a graph attribute recognizing and redistributing process. The inconsistency checking between two partial specifications (ViewPoints) can be accomplished to seek if there is a homomorphism between the two graph systems, which correspond to the two partial specifications (ViewPoints). The finding of homomorphism reduces the effort to track relationships inside the homomorphic part of the graphs. The seeking of homomorphism needs query subgraphs in a supergraph. Four basic operations are established in this framework for graph queries and replacements, UNION, INTERSECTION, DIFFERENCE, SELECTION. The non-determined sequence combination of the four basic operations can derive a lot complex operations on graph. The implementation of these four operations is based the List data structure in TCMJAVA, which stores all the shape or table cell objects in a diagram or table respectively. Dynamic aspects of graph systems are represented by graph rewrite rules. The application of graph rewriting rules keeps any information in the development process. Therefore, the recording of graph rewriting rules can reverse graph transformation, supports Undo operations in development process, supports method user to explore alternatives, and supplies active help to the developer to search for an inconsistency resolution. Hence, the application of graph rewriting rules supports inconsistency reasoning, analysis, diagnosis, tracing information, resolution, amelioration, rational. Therefore, our framework supports inconsistency management. In order to check the applicability of our framework, we applied our framework on a large-scale real project, International Space Station project.

In the tomes of NASA Space Station Specification, various notations used make all existing CASE tools not enough to check the consistency of inner-notation or inter-notations. For the valid space station modes and state transitions requirements in

International Space Station Specification, there are two different notations to represent the same requirements, state transition diagram and state transition table. We applied the WHERE to this piece of requirements specification. After method user defines the mapping functions in seeking homomorphism process, inconsistencies are detected and displayed. We found that there are often many inconsistencies even for a simple partial specification. This supports the argument; inconsistency is inevitable. In next section we discuss the limitation of our framework, and what should be done in the future.

7.2 Discussion

We described our framework in previous section and Chapter four. It is a preliminary research work. There should be a lot of places that need to be improved or reconstructed. We outline several deficiencies of the framework; they may not cover all of the holes in this framework.

In our framework, we emphasize that inconsistency handling actions need not necessarily remove the inconsistency. How does meta-level inconsistency handling actions combine with graph grammar approach is still a problem. In our framework, method user defines mapping functions in seeking a homomorphism between two graph systems by using wizard window. Sometimes, the definition of mapping functions are blur, the seeking of mapping functions is not very easy. This is an engineering issue, also is an implementation issue. In next section we will describe what makes a good CASE tool, and a wish list for the WHERE.

7.3 Future Work

In today's webbed world, with using of distributed technology and decentralized control, managing overlap and inconsistency in a software system under development is a big consideration, and is replacing the concern for rigidly enforcing consistency and removing redundancy. The delivery a tool like the WHERE is imperative. However, Before put into a real large-scale project, following implementation needs to be finished.

A granulated analysis and clarification is needed between the relationships of related notations, in order to use our framework better. For example, the static aspect of a partial specification can be represented using ER diagram, and the dynamic aspect can be represented in State transition diagram. A fined analysis of the relationships between these notations is needed.

The automatic conversion between two different related notations is expected, this will greatly unload human effort from tedious work, and concentrate on the creative activity. For example, a state transition diagram can be automatically generated from an ER diagram.

In a long-term, we pursue the computational support for our the WHERE project to support the full spectrum of inconsistency management covering specifications expressed in different languages, with different degrees of abstraction, granularity and formality, deploying different terminology and being at different stages of development or elaboration. We believe that to cope with the diversity of the inconsistency problem, the WHERE project should support multiple reasoning mechanisms and/or methods with different application domains; including Quasi-classical logic based consistency management, graph rewriting rules based consistency management. And the combination of QC-logic and graph rewriting rules may give rise to a potential better solution for inconsistency management. The research in this area is expected.

Bibliography

[1] S. Easterbrook and J. Callahan, "Independent Validation of Specifications: A coordination headache," IEEE Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96) - Workshop on *Requirements Engineering in and for Networked Enterprises*, Stanford, CA, Jun 19-21, 1996.

[2] Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling In Multi-Perspective Specifications", *IEEE Transactions on Software Engineering,* Vol.20, pp. 569-578, August 1994.

[3] Hunter and B. Nuseibeh, "Managing Inconsistent Specifications: Reasoning, Analysis and Action", *Department of Computing Technical Report Number 95/15, Imperial College,* London, UK, October 1995.

[4] H. J. Kreoswki, and G. Rozenberg, "On structured Graph Grammar I", *Journal of Information Sciences*, Vol.52, pp. 185-210. 1990A; "On structured Graph Grammar II", *Journal of Information Sciences*, Vol.52, pp. 211-246. 1990B.

[5] TCM homepage http://www.cs.vu.nl/~tcm/project.html.

[6] S. M. Easterbrook, "The Role of Independent V&V in Upstream Software Development Processes," *Second World Conference on Integrated Design and Process Technology (IDPT-96),* Austin, TX, December 1996.

[7] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh, "Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check", *International Journal on Concurrent Engineering: Research & Applications*, Vol. 2, pp. 209-222, March 1994.

[8] A. Al-Rawas, "A framework for the communication and organization of requirements perspectives", *Ph.D. Thesis,* University of Sussex, 1996.

[9] S. Dhaliwal, "Providing the Persistent Data Storage in a Software Engineering Environment Using Java/CORBA and a DBMS", *MS Thesis,* West Virginia University, 1997.

[10] S. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management,"*Software Engineering Journal*, Vol. 11, (1), 1996.

[11] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification", *IEEE Transactions on Software Engineering,* Vol. 20, pp. 760-773, October 1994.