An integrated approach for Developing Semantic mismatch free Commercial Off The Shelf (COTS) components

Developed by:

Muhammad Summair Raza
MS (Software Engineering)
Fall 2005

Supervised by:

Dr. Hamid Abdul Basit
Lahore University of Management Sciences (LUMS)

DEPATRTMENT OF COMPUTER SCIENCE
FACULTY OF BASIC & APPLIED SCIENCES
INTERNATIONAL ISLAMIC UNIVERSITY ISLMABAD

08/07/10

MA/MSc
005.14
RAI

1- Commercial products
2- computer software- Quality control.

T-06114 C2

In the Name of

ALLAH

The Most Merciful

The Most Beneficent

PROJECT IN BRIEF

Project Title:	An integrated approach for developing semantic mismatches free Commercial off the Shelf (COTS) component.
Organization:	International Islamic University Islamabad, Pakistan.
Objective:	The objective of the project is to fulfill the degree requirement of MS in Software Engineering.
Undertaken By:	Muhammad Summair Raza 89-FAS/MSSE/F05
Supervised By:	Dr. Hamid Abdul Basit Assistant Professor, Lahore University of Management Sciences, Lahore
tarted On:	
Completed On:	
Research Area	Component Based Software Engineering (CBSE)

International Islamic University, Islamabd
Faculty of Basic & Applied Sciences
Department of Computer Science

Dated: 28 March 2009

FINAL APPROVAL

It is certified that we have read the thesis, entitled "An Integrated Approach for developing semantic mismatch free Commercial Off The Shelf Components", submitted by Mr. Summair Raza (89-FAS/MSSE/F05), it is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University Islamabad for the award of MS degree in Software Engineering.

PROJECT EVALUATION COMMITTEE:

Supervisor
Dr. Hamid Abdul Basit
Assistant professor
LUMS, Lahore.

[illegible]
14 April 2009

External Examiner:
Dr. M. Afzal
Director KICSIT
KRL Kahuta Rwp.

[illegible]

Internal Examiner:
Shahbaz Ahmed
Asst. Prof DCS,
FBAS, IIU, H-10
Islamabad.

[illegible]

A thesis submitted to the Department of Computer Science,

Faculty of Basic & Applied Sciences, International Islamic University, Islamabad

Pakistan as a partial fulfillment of the

Requirements for the Award of the Degree of

MS in Software Engineering

To

MY DEAREST PARENTS & RESPECTED TEACHERS

Their efforts and guidance

Made me able to achieve this endeavor,

Without

Their prays and support

This dream could have never come true

Declaration

I hereby declare and affirm that this thesis neither as whole nor as part thereof has been copied out from any source; I have provided proper references and citations wherever required. It is further declared that I have completed this thesis on the basis of my personal efforts, made under the sincere guidance of my supervisor. If any part of this report is proven to be copied out or found to be a reproduction of some other, I shall stand by the consequences. No portion of the work presented in this report has been submitted in support of an application for other degree or qualification of this or any other University or institute of learning.

Muhammad Summair Raza

89-FAS/MSSE/F05

ACKNOWLEDGEMENTS

First of all I express my sincere thanks to ALLAH and bestow all praise and appreciation to Almighty Allah, The most Merciful and Compassionate, The Most Gracious and beneficent, whose bounteous blessings enabled me to pursue and perceive higher ideals life, who bestowed me good knowledge to complete my work successfully. And Especially I am grateful to His Holy Prophet Muhammad (SAW) who enabled me to recognize my creator and provided me a true path to follow for success in this world and for hereafter.

Secondly I wish to express my profound gratitude to my supervisor **Dr.Hamid Abdul Basit**, whose suggestions led me throughout this thesis. This thesis would not have been possible without the kind support, the trenchant critiques, the probing questions, and the remarkable patience of my research advisor.

Finally I am thankful to my parents who ever provide me warm encouragement, love and moral support during my entire academic career.

Muhammad Summair Raza

89-FAS/MSSE/F05

Table of Contents

List of Tables:

List of Figures:

ABSTRACT

Software in the modern age are mostly developed by the integration of pre fabricated COTS components as it is the simplest way to develop systems quickly consuming lesser cost as compared to the traditional development approaches. The promising features of CBSE have introduced a new idea of assembling software rather than building them.

Assembling software in this way alternatively results in rapid development, lesser cost with quality software assembled from pre tested COTS components. However the task is not as easy as it appears apparently. Assembling software from the existing components presents other challenges among which the "integration time mismatches" is the one.

Various strategies have been proposed to overcome these mismatches each requiring some external mechanism outside the component to solve them.

This dissertation is an endeavour to provide an integrated approach for resolving integration time semantic mismatches. It enables COTS components to detect semantic mismatches and resolve them by themselves, thus letting the COTS component itself to participate in mismatch resolution process.

The external mediation, in this way, will be reduced up to maximum, resulting in a smooth integration process with a cut-down in integration cost. The proposed approach will further enhance the fault tolerance capabilities of COTS components as it is used more and more.

Chapter-1

Introduction to COTS and CBSE

1.1 Motivation for CBSE:

Efforts have been made, right from emergence of software engineering to improve software development process with special emphasis on design to develop more significant notations to confine and capture the proposed functionality of system, along with encouraging the development of systems by reusing already developed components rather than developing from initial. Each success in these endeavours helped organizations to maintain and improve the quality, maintainability, and flexibility of complex and critical systems developed for broad category of domains.

However organizations, having large-scale and complex applications development, still face a lot of problems, especially while testing and updating the systems. So until systems are not designed carefully, they may be costly, in terms of cost and time to enhance the functionality further, and to test the systems updations effectively and efficiently. Furthermore, the ever-growing demand for unprecedented complex software has made software engineers seriously think in terms of code reusability, as the software sizes in excess of 10 million lines are a practical reality. Such system may have even a decade of development period, alternatively facing the challenges of changing requirements and different other parameters, obviously demanding a clear shift towards "Assembly of code" instead of "building from scratch" approach and using COTS products is one way to implement this strategy, because software development then becomes the process of "simply" integrating COTS components.Engineering practices to support code reuse – CBSE - is the need of time.

So Component Based Development (CBD), now, has been widely accepted as one of the core technologies both in academia and industry. It is successful model for developing software by integration of already developed components, resulting in reduction of efforts, cut of time to market [1] and generating a remarkable attraction due to the development of plug-and-play reusable-software, which lead to the concept of 'commercial off-the-shelf (COTS) components [2], developed by different vendors and probably tested and validated by their use in some other applications. Yet another capability, the Component based development comprises, is the easy reconfiguration of components or integration of newer versions to cater the desired changes in organizational business processes. CBD reduces complexity by offering high level of abstractions, separation of concerns and encapsulation of functionality. It involves designing a system so that it can be reused in other development efforts [3].

There is clear deferment of processes from develop-from-scratch approach towards build-from-components by their assessment, customization and finally integration [4]. According to Brooks [5], the best way to construct software is that never construct it.

Economic aspects are compelling software development to focus on the challenges and opportunities associated with COTS products [6]. Benefits of COTS-based-systems include reduced costs of development, quick deployment, and low maintenance costs [6]. These benefits compel organizations to get their software systems from pre-built COTS components [6]. A clear trend is there now to assemble software rather than building it. Frequently, components from different providers are used to assemble component-based applications.

Different studies conducted on reuse have resulted that about 40%-60% of SLOC is reusable among applications, about 60% of design can be reused in business software, about 75% of functionality is common in no. of programs, however only 15% code was only found to be unique to a particular application; the potential and the actual reuse rates range from 15%-18% [7].

"Software reuse benefits have been substantially discussed in literature e.g in software engineering encyclopedia, in NATO software reuse standards, PhD dissertations, software reuse books, journals and software engineering books etc" [7].

Following measures are taken from [7]

- An empirical study of 25 projects (with range 3,000-112,000 LOC) conducted at NASA software production environment resulted that about 32% of software has been reused /modified from pre-existing systems.

- In Motorola reuse is seen as an alternate-option to improve quality and productivity.

- At HP Inc. a survey (of two reuse projects) about reuse-assessment indicated reduction of defect density, with range 24%-76% and productivity increase of 40%-57%.
- IBM has developed a Reuse Support Center, which involves almost 30 sites worldwide. They have reported the millions of dollars savings, with reuse range 20%-30% in software.

However despite the benefits of reuse, there are certain factors that effect reuse success and failure both directly and indirectly. These factors may be technical, theoretical, organizational, management-related, economy-related etc [7]. Integration time mismatches are just one to name. Even, when we have selected the appropriate components according to requirements, some other fundamental issues may still prevail like one that chosen parts do not fit together well. The integration of such components results in no. of critical faults and inconsistencies [8] called "Architectural Mismatches" [9]. While leading to grave consequences, the mismatches also require some intermediate mediation mechanism, the Wrapper/Glue Code, to overcome the problem. This mediation mechanism is required as many times as the component is reused, not only for the same component while integrating in different applications, but also for its later versions. This entail the first attention of the researchers to come with the solutions of the number of problems adhere to CBSE.

This thesis is a consequence of motivation for this obligation and provides the solution for solving one of the integration time mismatches problem.

1.2 Component Base Software Engineering:

According to [10] CBSE is primarily concerned with three functions:

- Software development from pre-produced parts.
- Reuse of these parts in other software.
- Customization and maintenance of these parts to develop new software.

"CBSE emphasizes the design and development of software using reusable-software-components" [11].

CBSE includes much more than means of neatly organizing an inclusive reuse program. For constant development of components both processes and methodologies are also provided. It is the ultimate anthology of proficiencies ever accumulated on this emerging technology.

CBSE emerged decades ago. Production of software components was proposed in 1969 by McIlroy. Since then, a number of things e.g. pluggability and granularity, of components have significantly changed.

In contrast with the traditional development methodologies that adhere to be development centric, Component-Based Software Engineering (CBSE) aims to come with integration-centric approach. It uses the same Engineering principles as those of OOP to the entire designing and creating process of software systems. Among others, it spotlights on *reusing* and *adapting* already developed components, rather than developing from scratch, letting organizations to "assemble the software" rather than "building it". As a result, it gives several potential advantages, such as developing large systems, comprising of pre-developed components, and hence cutting down the overall development time and costs as the common functionality in applications may be developed only once and may be used again and again in different contexts, instead of re-inventing the wheel. It lets organization to maintain productivity and quality along with augmenting interoperability and portability due to the reason that components can be easily added and replaced.

Initially reusability was adhoc-based because, as a whole the industry was lacking any accepted standard and without standard developers could not be motivated to design and develop for reuse.

The ever growing demand for complex systems imposes limitations on industry to come up with tools to ensure their availability (development) with in shorter time. The "buy, don't build approach" is the first candidate solution, organizations are striving for, to cater the needs. As a result reusability has been gaining substantial importance to fulfil industrial requirement. CBSE ensures safer code – the code gets refined and becomes more reliable as it has already been tested in no. of contexts.

Organisations, while developing new components, need to develop them from reuse point of view, which means a careful design, that-if not considered may lead to huge loss of development time and cost.

The paragraph from [3] explains the clear trend which organisations seem to follow:

"The commercial off-the-shelf (COTS) products are becoming increasingly popular. They cause shrinking of budgets and accelerate the COTS augmentation. Both in development and maintenance there is a shift from custom-development to COTS-based systems. The proper use will establish a solid modernization practice".

CBSE now has become a constant roadmap for rapid software development and is a dominant research area. Massive research has already been conducted in this field and a lot of research is still in progress.

1.3 Components:

According to [13]:

"A Component is an opaque-implementation of functionality which Subjects to third-party composition, conforming with a component-model"

"*Reusable software components are self contained and clearly distinguishable artifacts which depict and/or perform explicit functions and have clear interfaces along with appropriate documentation and a defined reuse status*" [7].

Components are clearly identifiable and integrate able artifacts in software systems. They have well defined interfaces and encapsulate internal details so that they may interact with each other for the final component based software, even without knowing the internal details of each other. Each component has its own documentation.

Using of a component in various projects, result in the fact that it has to be maintained only once e.g. bugs need not be fixed only once, documentation is written only once, and by spreading the similar code over many locations we can avoid many inconsistency problems [7].

The "self-contained" in the above definition means that component should be self-sufficient, i.e. it should by itself perform the intended functionality without the help of or including the other components. However, in case a component needs the presence of other components then the entire group will be considered as a single component, with one component working as the interface for the intact group. According to this definition a single function may be a component as far as it does not need the presence of other functions, so the different libraries having set of functions and modules are considered as a single component.

"Identifiable" means, the implementation of a single software component must be precise which could easily be identified i.e. the code of component must not be intermixed with other components or scattered in all files of application, but must be in a single module or in case of more than one modules, it must be properly packaged. Components details should be properly documented and they should be developed by following standard CBSE practices, in order to maximize and facilitate their reuse.

1.4 Types of component reuse:

1.4.1 Black Box Reuse:

By black box reuse we mean that we use the component but we cannot see, know or modify he internal functionality of component [7]. The component user only has single or a set of interfaces provided by the component to interact with it. Internal details are hidden and cannot be modified. User only has the knowledge of "what" functionality a component implements with out knowing "how" it is being implemented. This alternatively facilitates the replacement of components with the newer ones.

1.4.2 White Box Reuse:

It is still another form of reuse. It means that the internals of a component can be changed for the purpose of reuse [7]. User has the source code and other implementation details of the component; this facilitates the component user to modify the component according to requirements. However, on the other hand, it has its own negatives. Customization of the internal of a component will be regarded as a new component, which will require a thorough testing and fixing of the bugs.

1.4.3 Gray Box Reuse:

Gray box reuse means that (components are used as the black box components but) the user can see the internals of the component which provides the user with the information about internal functionality of component without giving him any ability to change it [7]. So the user can get some idea about, how component works.

Glass box reuse may have its own limitations. Changing the internals of component may become fatal as it may lead to malfunctioning of component.

1.5 Benefits of Software Reuse:

Software reuse always has positive impacts on quality, productivity, performance and reliability providing application consistency and reducing risks.

Quality: as the component gets reused with the passage of time, the number of errors and bugs get fixed and the quality of component increases, particularly this would not be the case for a newly developed component.

Productivity: reuse leads to increase the productivity, as using the existing code save a lot of time and effort for designing, developing and testing the same code again. This alternatively benefits in the form of a sharp cut down in overall cost.

Performance: with each no. of reuse a component gets performance wise maturity, as it gets refined with the implementation of new algorithms, and ultimately saves the time for optimization of the newly developed code.

Interoperability: Components present standard interfaces to use their services, which lets the augmented interoperability of components in component based systems, leading to lesser no of interoperability errors.

Reduced Risks: if components are used then there is less chance of risks using these components as they get already tested and verified in various no. of applications contexts, which enhances the confidence of the application developer. However this is not the case with the newly developed code which has, so far, not tested in real domain.

Standard compliance: due to the presence of standards, such as interface standards, the use of components leads to enhanced compliance among applications, e.g. if Menus are implemented in the form of component and all applications use the same menu component, all applications will have same menu format which will increase the dependency of systems as user are likely to make less mistakes in case of similar interface.

1.6 Challenges of software reuse:

Despite the benefits of reuse, it is still not an easy paradigm, one might assume. There are many challenges that contribute towards the failure of software reuse. The next section presents some of the challenges.

Initial cost: software reuse saves the cost for further software development; however, it requires an initial investment, which of-course needs support from organizations top management.

Inadequate organizational structure: Organizations tending to exploit reuse need a bit change in their structure. For example, they may need separate teams responsible to gather, maintain and provide reusable-components [7].

Not-invented here: people may be hesitant to use other's software. They may have lack of confidence on the software being reused or may be confident to develop a better and efficient code by themselves instead of using someone else's code.

Difficult to find reusable components: before reuse the first step is to search for an appropriate component. Normally components come with incomplete documentation that does not fully explain the internals of component which leads, not only, towards the difficulty of finding the appropriate components, but also makes it a tough task to reuse a selected component.

Non-reusability of components: after a component is found, it may still be challenging to reuse the component, because the software are seldom written in a way to ease their reuse, although not intentionally, it requires solid expertise and skill with all the possible (future) reuses in mind.

1.7 Component interfaces:

Component interfaces are the connecting points to use the component's services. This is normally the user view of the component. They hide the implementation details. The successful interoperability of components depends upon interface specification; these specifications provide a basis for the development, management and use of software components [14]. Jun Han in [14] defines a framework for interface specification discussing the five aspects given below:

Signature: are the basis tools for the component to interact with external environment. They include all the necessary methods or elements which help in this interaction i.e., attribute operations and events.

Configurations: component interface may have no. of configurations according to use contexts; each configuration may comprise of no. of ports according to the functionality, the component provides in Component Based system.

Semantics: semantics of signature elements mean to describe the semantics of the all signature elements, to capture their precise behavior.

I-constraints: Interaction constraints provide the complete details of how to get the intended functionality of component.

Qualities: characterizes the non-functional or quality attributes, e.g. attributes related to reliability, security and performance.

So interfaces need to follow the implementation standards. Components may have multiple interfaces to fulfill multiple customer needs e.g. they may implement separate interface for configuration, yet another interface for initialization and domain functionality etc.

Two important aspects of interaction with component through interfaces are the function calls (to use the component services) and the data exchange between the component and the target application. The next section will explain that any mismatch between component's interface and the interface of target application causes serious integration time or execution time errors which may lead to the failure of the application.

1.9 Architectural mismatches:

Despite all the efforts, software development from reusable components remains an elusive goal, even, when we have selected the appropriate components according to requirements, some other fundamental issues may still prevail like one that chosen parts do not fit together well [9]. The integration of such components results in no. of critical faults and inconsistencies [8] called "Architectural Mismatches" [9]

Architectural mismatches, also called "Integration anomalies" [8], arise when a component makes some false assumptions (pre-assumptions) about the architecture of the target application [16]. These assumptions often contradict the assumptions of other components/target application and lead to a number of variation points (variability). A variation point means the place where there is some conflict among same family members [18]. Different classifications of these anomalies exist in literature e.g. [17, 19], however the main are:

- **Syntax at component interface level:**
  Mismatches in representation of data imported/exported.
- **Semantics at component interface level:**
  Occur when the data with different semantics is exchanged among different components.
- **Application-specific properties at system level:**
  Local functionalities of the components do not accurately represent the target application context.
- **Pragmatic properties of system environment:**
  These inconsistencies are related to the computational environment of the components e.g. access-policies, concurrency-constraints, timing requirements or underlying architectural constraints etc.

As COTS often come in black box form (binary form), therefore it is not possible to modify the source code or re-link the object code with library [25]. So we use the Wrapper/Glue code to negotiate or overcome the anomalies that arise. A recent research [8] emphasizes to separately implement fault tolerant strategies for these inconsistencies in Wrapper/Glue Code.

Chapter - 2

Literature Survey

A smooth and error free interaction of components is critical to the success of component based systems. For these systems architectural mismatches have always been an issue and the situation becomes worst critical in case of large, complex applications having high dependability requirements. It is essential to include different ways to muddle through software faults [21]. So the mismatches have been focus of the research with passage of time.

2.1 Terminologies:

CBA	Commercial-Off-The-Shelf Based Applications
CBS	Commercial-Off-The-Shelf Based Systems
CBSE	Component Based Software Engineering
COTS	Commercial Off-The-Shelf
GOTS	Government Off-The-Shelf
GUI	Graphics User Interface
LOC	Lines of Code
KLOC	Kilo-Lines of Code
NATO	North Atlantic Treaty Organization
NASA	National Aeronautics and Space Administration
IBM	International Business Machines Corporation.
HP	Hawvelet Packard
NEC	NEC Software Engineering Laboratory
GTE	GTE Data Services Inc.
UDS	Universe Defense Systems
OOP	Object Oriented Programming
SEI	Software Engineering Institute
OSS	Open Source Software

2.2 COTS Integration:

COTS based systems are composed of components at different levels, integrated either statically at CBS's design time or dynamically at runtime. The interoperability among the components takes place in the form of sharing / exchanging data and services. Message passing (through message busses) may be used, in case; a component wants to establish a communication session with other components.

2.3 Integration anomalies: classification:

Architectural mismatches or integration anomalies are inevitable part of COTS integration process. They stem when a reusable component makes false assumptions about the architecture of target application [16]. These assumptions often diverge with the assumptions of other components/target application and lead to a number of variation points (variability). A variation point means the place where there is some conflict among same family members [18].

Kevin et all. in [22] argue:

"The components that apparently seem to be compatible, architectural mismatches may hinder their integration to develop and application. Software engineers can face subtle problems to get the components work together even if they are developed using the same programming language and run on the same platform. Engineers have to write the wrapper code, customize the component and some time re-implements the functionality to overcome the architectural mismatches".

The resulting system may be unable to meet the performance requirements.

So, it has been the focus of the research, right from the first day, to comprehensively and systematically detect and classify such mismatches, develop sufficient approaches and tools to cope with them.

The term "Architectural Mismatch" was first coined by Garlan et. all in [9]. While developing AESOP, they came across six main difficulties during integration of four existing software subsystems (components) into a new coherent system: bulk of code, poor performance, extensive modification required to make components work together smoothly, the necessity of reinventing existing functionality to meet the intended use, redundant complexity of resulting application, and a complex, error-prone system development process [16]; Root causes of these mismatches were divided in four broad categories: Nature of the components, Nature of the connectors, Global architectural structure, and Construction process [16]. Four guidelines were proposed by them [16] to overcome these mismatches

- Architectural assumptions should be made explicit.
- Large pieces of software should be developed from orthogonal subcomponents.
- Techniques should be provided for reconciling mismatches.
- Sources for architectural design guidance should be developed.

In fact study of Garlan et. all provided a solid base for further research to deal with architectural mismatches.

In [17] Yakimovich et el. presented two major causes of COTS interaction incompatibilities: syntax and semantic-pragmatic.

Syntax defines the syntactic rules, where as the functional interaction specifications are defined by semantic-pragmatics. Syntactic differences among the components result in syntactic incompatibilities. Semantic-pragmatic incompatibilities, on the other hand, can arise out of the conflict in components interaction. The semantic pragmatic incompatibilities are further classified as:

- 1-order semantic-pragmatic incompatibility or internal problem: caused by a single component. E.g. it may be that this component does not fulfill the required functionality.
- 2-order semantic-pragmatic incompatibility or a mismatch: incompatibility occurred due to interaction of two components.
- N-order semantic-pragmatic incompatibility: incompatibility caused due to interaction of several components.

In [19] Reussner e.t al. present enhancements of two already existing classification schemas for component-interoperability-errors to carry component alteration in order to avoid mismatches.

Firstly they modify the “Interface classification schema”, and present a more systematic classification of incompatibilities based on two dimensions. One dimension differentiates between functional and non functional perspectives and the second is concerned with granularity of interface description. By using these two dimensions they resulted in a classification matrix shown in Table-2.

Secondly, they derive a novel classification of component heterogeneities from UnScom framework [20]. It differentiates between different component perspectives, i.e. it differentiates between different developments perspectives, corresponding to main steps of component development process, also distinguishing between three design views, on the other hand as shown in Table-3

	Methods	**Interfaces**	**Domain**
Functional	Signature	Protocols	Domain Objects
Non-Functional	Method Specific Quality	Interface Specific Quality	Domain Constraints

Table-2: A Classification of Interface Models.

	Functionality / Concepts (Domain-related)	Architectural Design / Interfaces (Logical)	Implementation / Quality (physical)
Static View (Structure)	Information Objects (Entity Model)	Type Declarations, Properties (Attributes)	Usability, Maintainability, Portability
Operational View (Effects)	Functions (Operations Model)	Evens, Methods, Exceptions, Assertions	Functionality
Dynamic View (Interactions)	Process (Flow Model)	Interaction Protocols	Reliability, Efficiency

Table- 3: The UnSCom Classification Schema.

In [8] Sglietti et al. present an approach to detect and tolerate architectural inconsistencies. They categorize architectural mismatches in different classes and tend to implement a wrapper that separately handles these mismatches.

They mainly identify integration anomalies in four classes:

Syntactic inconsistencies: inconsistency in the representations of data imported/exported to/from component.

Semantic inconsistencies: inconsistency of semantic nature e.g. exchanging the data with different semantics

Application-based inconsistencies: may occur if the local functionalities of components fail to accurately represent the global application context.

Pragmatic inconsistencies: pragmatic inconsistencies are concerned with component's computational environment. E.g. access policies, timing-constraints and other architectural limitations etc.

These classes are further classified in sub-classes as shown in fig 1.

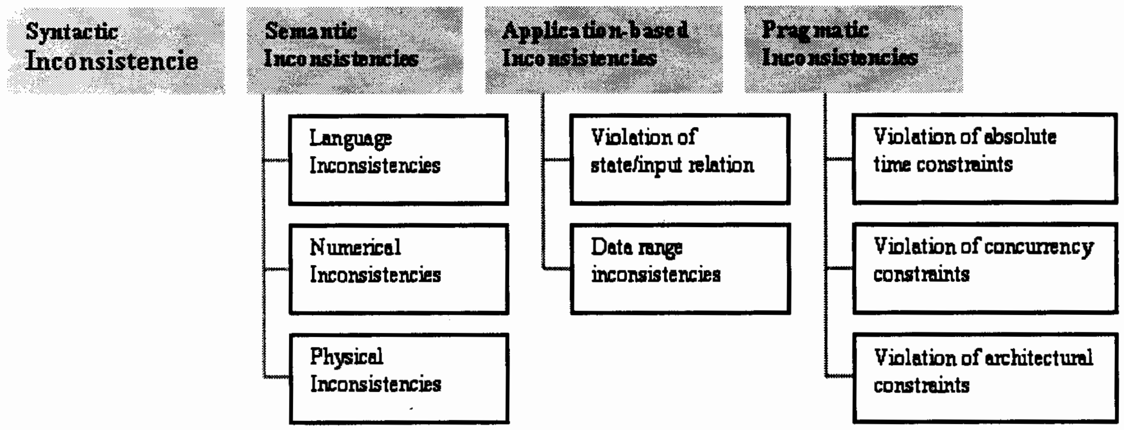

Syntactic Inconsistencie
Semantic Inconsistencies
Application-based Inconsistencies
Pragmatic Inconsistencies
Language Inconsistencies
Numerical Inconsistencies
Physical Inconsistencies
Violation of state/input relation
Data range inconsistencies
Violation of absolute time constraints
Violation of concurrency constraints
Violation of architectural constraints

Figure 1: Architectural inconsistencies and sub-classification.

2.4 Integration anomalies: negotiation and reconciliation approaches

Keshav et. al [23] discuss their initial conclusions from architectural-style-integration analysis. They form an integration-taxonomy comprising of three main functional integration elements: Translator, Controller and Extender.

A **Translator** translates both the data and the functions between different formats; however contents of the data are not changed. It does not need the information about source/destination of data

A **Controller** based on some predefined decision making process, a controller synchronizes and negotiates information exchange between different components. It does however need to know the exact identities of components for which decisions are being made.

An **Extender** adds new features/functionalities, and hence augments component‘s capability. Extender may or may not need to know about the component identity with which it interacts, depending on the particular application.

Authors also propose to combine these basic integration elements to make possible, the interaction of different components combinations.

Robert DeLine in [24] provides the catalog of tools and techniques to negotiate packaging-mismatches that are organized with respect to underline architectural-commitment. All the techniques presented are explained with the help of an example system consisting of two components A and B that exhibit incompatibilities while interacting with each other.

On-line bridge: in using on-line bridge a new component (i.e. Br - bridge component) is inserted between the interacting components A and B. Br implements a separate interface for each of the components involved in interaction.

Off-line bridge: is a special version of on-line bridge except that the component-B is in form of some persistent data. Now the bridge component Br reads data of B and transforms it to be compatible to A. This component transforms component B to B' which then interacts with component A.

Wrapper: in this method the bridge Br and the component B are wrapped together to form a new component B', this component than interacts with component A.

Mediator: in this method the connector C can support several alternatives (interfaces) for a given commitment. Components A and B can use any of the provided commitment to interact with each other.

Intermediate Representation: just like Mediator technique, but the only restriction is that the mismatches between component A and B are caused by data representation. Connector C can support several alternatives (interfaces) for a given commitment regarding data-representation.

Unilateral negotiation: In this technique a single component (A) supports multiple commitments, and if the other component's (B's) commitment is supported by it (i.e by A) then A is specialized to match B's commitment and they are integrated.

Bilateral negotiation: in this technique both component support a set of alternative commitments and agree upon a protocol to select one of the alternatives with negotiation with each other.

Component extension technique: In this technique a component provides mean for its extension. It defers some of the commitments about interaction by assigning these commitments to a set of modules integrated while the component is initialized at runtime

Eun Soo Cho et. al. in [1] present a methodology for component development. During "Identify variability" (DA3) phase they tend to model the existing variability of the domain, to design components that let component users to customize variation points. They tend to identify the attribute and logic variability by examining function descriptions and use case descriptions. They argue to consider function semantics while comparing functions. Similarly they identify work flow variability by examining use case descriptions. They further argue to specify the scope of each variation point. If variation point has a closed set of known variants, it is marked as "Predictable", else "Non-predictable" which means that the scope is open for future additions.

In [25] Soo Dong Kim et al. present a comprehensive set of techniques to comprehend variability in blackbox components and present effective interface-based customization schemes:

Selection Technique: it provides classes and a customize interface for component users in order to select one of the component variants, realized in it. After selection of a specific variant, it is stored in order to use it as references in further invocations.

Plug-in technique: in this technique we pass references of objects to components using customize interface and call the functions of these objects (whose references are passed) by using these references inside the component. In this way we can provide the application specific functionality to component and the component can be customized for each application.

External profile technique: this technique an external file such as XML file, describing the customization variants, is used to allocate an external customization variant to a variation-point. In this way, however, to change a variant the external profile needs to be changed.

Chapter - 3

Definition of Problem

All of the integration mismatches require a permanent solution in order to generate plug and play COTS.

This thesis is an endeavor that lets us develop such intelligent COTS components which lead towards minimum external mediation required for resolving semantic mismatches. Semantic mismatches occur due to misinterpretation of data exchanged between components, sequencing critical system failures, in case they are not resolved e.g. failure of NASA's Mars Climate Orbiter mission [26], where a component exported its data which was representing physical force in British units (pound-force), while the other component expected the measure in metric units (Newton). Similarly, the cost of programming-errors in component interoperability is estimated to be $16 Billion in capital facilities industry in US alone. The primary reason for such high cost is due to fixing of errors in incorrect data that is exchanged between different components [27].

So all these facts lead towards the need of an active research to cope with semantic mismatches as to avoid system failures, reduce development costs, and cater the obstacles to provide smooth and easily integrate able COTS having Plug and play characteristics.

3.1 Problem definition:

Almost all the negotiation strategies, to resolve architectural mismatches (including semantic errors), tend to provide reconciliation methodologies external to components; however an important scenario that is left unaddressed, is the internal design of the component, especially from the perspective of its role in resolving integration mismatches, and particularly resolving semantic mismatches as in context of this thesis.

All the preceding facts discussed so far lead to the following question that frames the problem area for this research:

"How can we develop such intelligent Commercial off the self (COTS) components that could themselves participate in fault tolerance mechanism to resolve the semantic mismatches, in order to reduce the traditional external mediation mechanism required?"

The above statement clearly implies that the successful implementation of proposed solution will reduce, up to maximum extent, the intermediate mediation mechanism required to negotiate the semantic mismatches.

3.2 Research Questions:

This thesis tends to answer the following questions:

- How can we reduce intermediate Reconciliation work required to negotiate the integration mismatches?
- How can we develop intelligent COTS components which themselves participate in mismatch reconciliation process, (to reduce reconciliation work)?
- What will be the role of COTS internal design in achieving the goal?
- What enhancements will be required in COTS internal design?

3.3 Main Objectives in Order of Priority:

The objectives of the research are:

1) To analyze the COTS integration process.
2) To analyze semantic mismatches, their categories and root causes.
3) To analyze the fault tolerance strategies.
4) To probe out the way to involve the COTS in fault tolerance mechanism in order to reduce the intermediate mediation mechanism for resolving the semantic mismatches.
5) To explore, how to minimize the dependency of component-based developers on COTS specifications, regarding semantics of data exchange (mismatches) between components

3.4 Outputs of the proposed research

The outputs of the proposed research are:

- A simple integration process resulting minimum integration time semantic faults and consequently reducing mediation efforts (Wrapper/Glue Code).
- An enhanced COTS design that will let the COTS to be intelligent enough to itself participate in fault tolerance process.
- Reduced dependency of COTS integrators on COTS design specification, regarding integration faults.

3.5 Benefits of the Research

Some of the benefits of the proposed research are:

- A simple and smooth COTS integration approach.
- Minimum integration-time architectural mismatches.
- Reduced cost of COTS integration.

3.6 Justification:

Wrapper/glue code, bridges, mediators, extenders and controllers etc. are all intermediate mediation mechanisms provided by different research approaches, however all these mechanisms are external to components and are required as many times as component is integrated, not only for same component while integrating in different applications but also for its later versions. This requires a clear shift in the internal design of component that could enable the smooth integration of COTS with minimum no. of integration time mismatches and required customization process.

As COTS often exist in black box (binary) form, therefore it is prohibited to modify the source code or re-link the object code with library. So the Wrappers/ Glue wares remain the only mechanism to overcome the problem.

Although mediation approaches reconcile the inconsistencies including integration time and runtime semantic mismatches, however they suffer the following drawbacks:

- Developing intermediate mediation mechanism (Wrapper/glue code etc.) requires in-depth details of COTS functionality and hence not an easy job as, the COTS specifications are normally incomplete [28] and do not provide much detail about the COTS internal functionality.
- All the intermediate code remains the separate entity, external to the components, especially in case of Black Box Components.
- In case of porting the same component to some other (target) application, most likely the same type/no. of anomalies will be flagged again and the same amount of intermediate code will have to be re-written in context of the new target.

The above facts clearly show that the intermediate mediation mechanism is just a temporary solution that does not add to enhance anomaly-free COTS portability and does not provide any solution to minimize the amount of Wrapper/Glue Code. Also, it is really a challenge to customize black box components without any access to internal design and source code of these COTS [25]. However, this is the only solution so far to overcome the problem.

The proposed integrated approach will be a step ahead towards providing a complete base for developing glue ware independent, anomaly-free, automatically plug-and-play COTS (regarding semantic mismatches) with increased portability and target domains. An important feature of Such COTS components will be the role of a component itself in resolving semantic mismatch, consequently minimizing the dependency of component-based developers on COTS design specifications.

Finally, this solution will also help the COTS users, who can get rid of consulting the COTS specifications to resolve the integration mismatches; as such specifications are mostly incomplete [28] in providing an in-depth detail of the COTS internal functionality.

3.7 Research Method

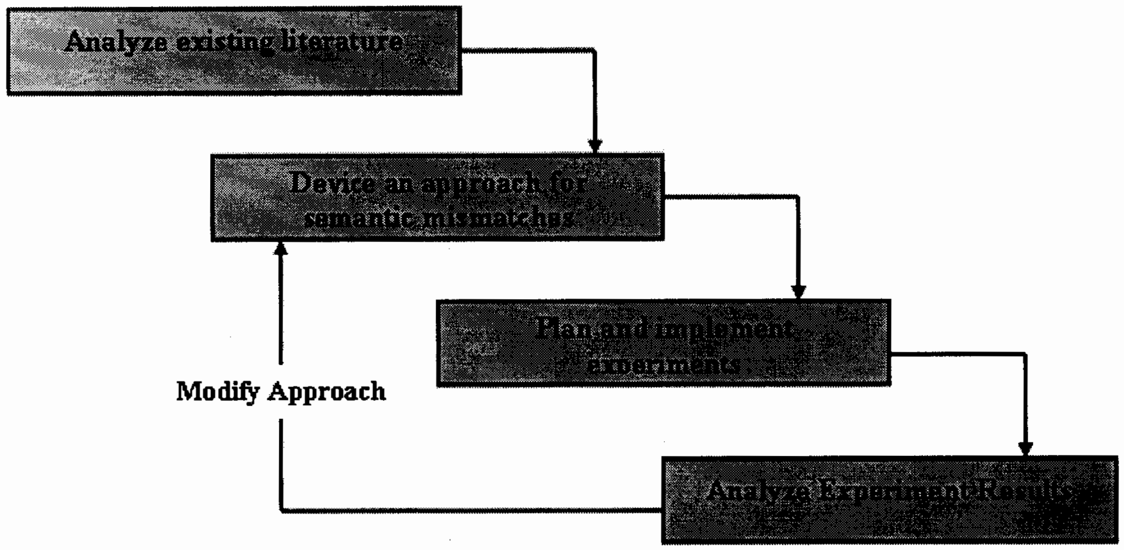

Analyze existing literature
Device an approach for semantic mismatches
Plan and implement experiments
Analyze Experiment Results
Modify Approach

Figure 2: Research Method

The main question of research was: how to develop a methodology to solve the semantic mismatches while reducing the external mediation mechanism required.

The apparent starting point to answer the question was to study a large body of knowledge associated with resolving architectural mismatches. As the research tends to modify the existing design of COTS, so different approaches for designing COTS were also studied and a new slightly enhanced design of COTS was proposed.

Experiments were then carried out to verify the methodology and results were analyzed and compared with other approaches to confirm the validity of the proposed solution. Entire process is shown Figure 2.

3.8 Scope:

Although all integration mismatches require an active research, but this thesis provides a solution for semantic mismatches that occur at interface level. A design based approach is presented to solve the problem i.e. a slight enhancement is made to traditional design of COTS, in addition to the interface and domain objects of a COTS component.

The approach enables COTS to itself identify a semantic mismatch and then reconcile it. The approach will minimize the amount of external work required to heal semantic errors, alternatively increasing portability and target domains while reducing CBA developer's dependability on COTS specifications.

The approach also motivates to make the COTS intelligent enough to not only solve semantic mismatches but also other classes of errors discussed in previous sections. Finally the approach is equally applicable to all types of COTS either they are black-box, white-box or gray-box.

Chapter - 4

Proposed Solution

The proposed research solves the semantic mismatches in a way that the COTS component itself participates in error reconciliation process without requiring mediation code being needed. This chapter will provide a brief detail of the proposed solution.
I argue that the errors will continue to occur and the intermediate reconciliation will be required many times during integration until and unless we involve the COTS itself in the negotiation process.

4.1 COTS Semantic Mismatch Reconciliation using Semantic Thesaurus (SMiReST):

Semantic Mismatch Reconciliation using Semantic Thesaurus (SMiReST), for its implementation, requires two enhancements in the traditional COTS development process:

- **Enhanced domain analysis**
- **Enhanced COTS internal design**

Following section discusses both steps in detail

4.1.1 Enhanced domain analysis:

During this phase the target domain of component is analyzed to understand the problem and component requirements. SMiReST requires an extra consideration to be made to traditional domain analysis phase. It emphasizes to identify and model all the possible semantics of data that the component will import and export during its interaction with target application. E.g. in case, the parameters of a certain service call include the price of an item, we need to consider all the possible currencies, the target application may be deemed to use.

The process of modeling all the possible data semantics will enable the COTS to be enriched semantically so that it may be capable to negotiate semantic mismatches during interoperation, in case any semantic mismatch is detected.

This requires a comprehensive analysis and thorough study of all possible target contexts. A history of other COTS from the same family and some possible target applications will be more helpful in this regards.

4.1.2 Enhanced COTS internal design:

COTS components are a collection of domain objects that interoperate to provide the intended functionality. The functionality is provided by using external interfaces. In fact an interface is an invoking point of a COTS component. Figure 3.

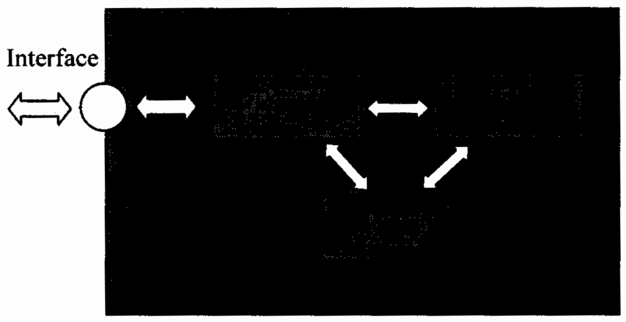


Figure 3: traditional COTS design

Each object in set of objects is dedicated for a specified task. Additionally each object may further aggregate sub-objects, and hierarchy may continue up to n-level.

The proposed COTS design suggests enhanceable semantic thesaurus (EST) as an essential part of each component that is developed. Now the COTS component along with Central Control Unit (CU) consists of two sub components: Enhanceable Semantic Thesaurus component – the component which implements the EST, and Main Functionality Component (MFC) – set of domain objects which perform the domain functionality as shown in figure 4.

In fact all the interaction of component is controlled by central control unit (CU), that intercepts all the calls to and from the components, it then verifies the semantics of data and in case any mismatch is found EST component is request to resolve it and then the call is delegated to MFC. Similarly before returning the data to calling application semantics are validated and made compatible to those of calling application and finally data is returned.

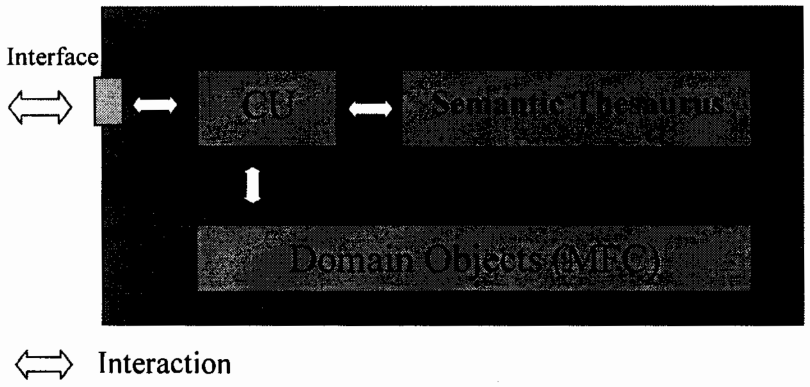


Figure 4: Enhanced Cots Design

4.2 Enhance able Semantic Thesaurus (EST):

Enhanceable Semantic Thesaurus (EST) is basically an implementation of semantic dictionary that holds all the possible semantics (that were identified during enhanced domain analysis phase) of the data, the component operates on.

4.2.1 EST Features:

Enhance able:

An important feature of EST, as the name indicates, is its ability of being enhance able. This means that if the semantics (considered by target application) of any data are not defined in EST (one of the situations leading to semantic mismatch), they can be added using a simple interface (explained in next session). In fact this feature enables the COTS itself, to participate in mismatch resolution process which otherwise requires intensive external mediation work.

Mismatch elimination with passage of time:

Probability of semantic mismatches reduces as much as the component gets used. This is because EST gets populated with more and more semantics, which alternatively `augments the component's fault tolerance capability and increases component strength to participate in mismatch resolution process.

It also means that the later versions of COTS component will less error prone as compared to earlier versions

4.3 EST Component: Implementation of semantic thesaurus:

In any component, EST can be implemented either by applying a single object or a set of objects each having a dedicated functionality. However the best approach to implement EST is to implement it as a sub component, of the main component.

TH6556

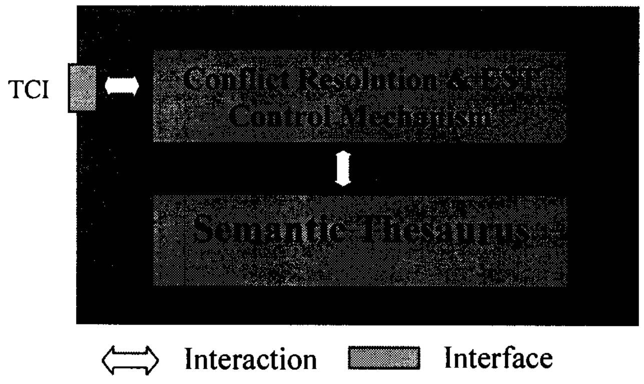


Figure 5: EST Component

Main parts of EST component Structure are (Figure 5):

4.3.1 Semantic Thesaurus:

It is basically the repository of all the semantics that were identified during Enhanced domain analysis phase. It can be implemented either by using database or simple flat files, storing the semantics of the data. Any new semantic that is defined is stored in this repository for future use. In fact this repository gets enriched as new semantics are defined, with the use of the component.

4.3.2 Conflict Resolution & Thesaurus Control Mechanism:

This feature enables the EST component to resolve the mismatch and negotiate it, on the basis of the terms defined in Semantic Thesaurus.

All the thesaurus related processing is performed by this sub-component, e.g resolve the mismatch, add new semantics in Thesaurus, delete semantics from thesaurus etc.

It is the key feature of EST component that enables the main COTS component to participate in mismatch detection and resolution process, and hence eliminates the external mediation mechanism required to resolve the mismatches otherwise.

4.3.3 Thesaurus Control Interface (TCI):

All the functionality of EST component can be invoked through this interface. It is used by CU, which after detecting any mismatch, uses this interface to forward the requests for negotiating it.

Similarly the Semantic Thesaurus management tasks e.g. its customization by addition of more semantics, by modification or deletion of existing semantics etc. can be performed through this interface.

4.4 Interaction of EST supported Components:

All the calls from the target applications are first interpreted by CU. The data accompanied by these calls is verified, and if semantic mismatch is detected, CU requests EST component to negotiate it and after the mismatch is resolved by EST component, CU forwards the call to MFC, which after performing the requested functionality returns the results to CU. CU, if needed, now again requests the EST component to make the semantics compatible with the calling application, and the results are finally forwarded to calling application.

Interaction of EST supported components is explained with the help of diagram (Figure 6) given below.

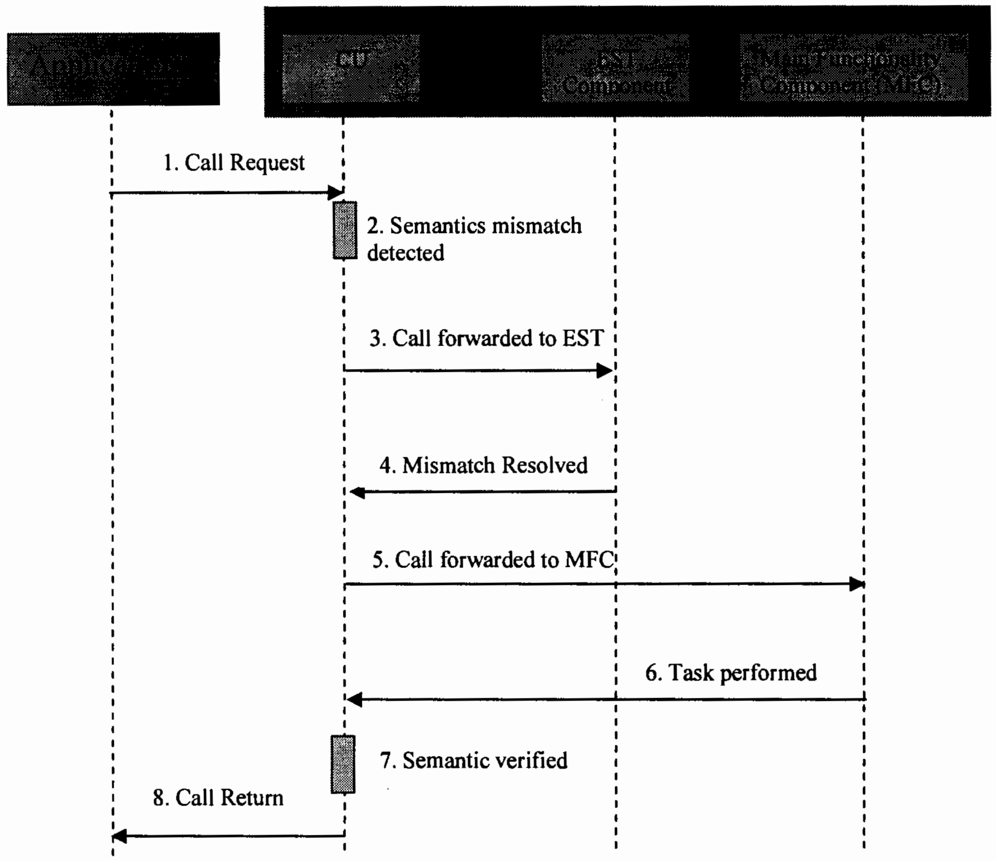


Figure 6: Interaction of EST supported component

Chapter - 5

Case Studies

To prove its solidity, every new concept and idea requires concrete experiments before its practical implementation. In order to prove the proposed solution three case studies were implemented.

In each case study one separate mathematical component was integrated to a custom build application. Each component was black box in nature and to keep the consistency among results the same custom build application was used.

Finally an in-house developed math component, "eCalculus", developed according to proposed solution was integrated and the results were analyzed on the basis of the results. The integration of same nature of components also proved the drawback of current development approaches regarding the semantics of data exchanged.

5.1 MathType:

A mathematical application "MathType" is developed to prove the results. It is a custom build application meant for providing functional support to mathematicians, scientists and engineers regarding the basic math domains e.g. trigonometry, statistics, natural number manipulation etc.

Implementation of MathType was completed in two steps:

- A Command Line Interface (CLI) was developed to get the input commands from user and to show the results.
- The CLI was then integrated with three mathematical components (COTS) and in-house developed eCalculus.

Each component was responsible to complete the command, and provide the results back to the CLI.

5.2 Wrapper code development:

In order to resolve the semantic mismatches among CLI and the integrated components, the reconciliation mechanism was implemented. Object Oriented approach was used to develop the wrapper code, and separate wrapper classes were implemented for reconciliation of data with different semantics just for "separation of concerns".

Finally the total no. of wrapper classes, developed in each case (during integration of COTS and eCalculus), lead us to make conclusions about the size of the wrapper code required and ultimately to the validity of proposed solution.

5.3 Case Study – 1:

In first case study a mathematical component "Extreme Optimization Numerical Library (EONL) version 3.0" by Extreme Optimization [29] was integrated with MathType. *"EONL is the most innate numerical-library intended for the .NET framework. It has its use in large number of applications. This library basically provides object oriented approach without suffering any cost regarding the performance"*. [29].

5.3.1 Concerns:

The intention to integrate this component was:

- To find out the no. of semantic mismatches and calculate the size/effort to write the wrapper code for resolving these mismatches.
- To analyze the portability of the component by verify about either the component permits or not, the easy enhancement of semantics regarding data exchange

A full 60 days trial version of EONL for .NET was used for this purpose.

5.3.2 Basic Course of actions: Narrative style

MathType needs to deal with different semantics of data for different mathematical domains e.g. for trigonometric functions some possible semantics were Degree, Radian and Grads Etc. Also sometime user needs to define its own semantics e.g Base-5, Base-20 number systems etc.

However for such issues, no support was found in EONML, so the traditional approach was followed to negotiate the mismatches. i.e. Wrapper code was developed as shown in figure 7.

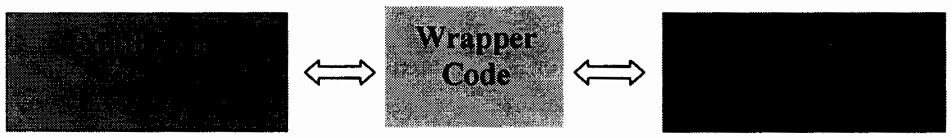


Figure 7: EONL Integration: Abstract Level

Table-4 below shows the complete list of data and possible semantics that were concerned with MathType

Table-4: Semantics of data exchanged

Data Exchanged	Data Semantics	Description
Number System	Binary	Binary number system includes 0 and 1
	Octal	Octal no. system includes numbers from 0-7
	Decimal	Has base-10 and includes numbers from 0-9
	Hexadecimal	Has base-16 and includes decimal numbers 0-9 and A, B, C, D, E, F
	N-Number System	It represents a number system with any base provided the base is less than 35. It allows the user to define its own semantics i.e. user can define any number system according to its own requirements. This option was not provided in any component
Angle Mode	Degree	Unit of angle equal to 1/360 of circle circumference.
	Radians	Unit of angle equal to 180/п degree
	Gradians	Unit of plane angle, equivalent to $\frac{1}{400}$ of a full circle.
	N-Angle Mod	It shows the user defined angle mod i.e. user can define his own semantics depending on his requirement. This option was also not provided in any component

5.3.3 Wrapper Code for EONML:

In order to overcome the mismatches the wrapper code was developed, as explained already, Object Oriented approach was used to overcome the issue, a separate wrapper class was implemented to negotiate each type of semantic mismatch.

Table-5 shows the list of classes developed corresponding to each type of semantic mismatch

Table-5: List of wrapper classes

Data Exchanged	Data Semantics	Wrapper Class	Explanation
Number System	Binary	CBin	Converts data with binary semantic to any other semantic
	Octal	COct	Converts data with Octal semantic to any other semantic
	Decimal	CDec	Converts data with Decimal semantic to any other semantic
	Hexadecimal	CHex	Converts data with Hexadecimal semantic to any other semantic
	N-Number System	CNmN	Converts data with N-Number system semantics to other semantic
Angle Mode	Degree	CDeg	Converts Degree mod to any other mod
	Radians	CRad	Converts Radian mod to any other mod
	Grad	CGrad	Converts Grad mod to any other mod
	N-Angle Mod	CAmN	Converts N-Angle mod to any other mod

Totally 10 wrapper classes were written to negotiate all the mismatches between MathType and EONML where 9 classes were handling semantic mismatches and one class "CDataConv" was written to convert data structures to make them compatible with those of EONML data structures.

5.4 Case Study – 2:

During this case study a second mathematical component "Scinet Math" by "OBACS" [30] was integrated with "MathType".

"Scinet Math" is intended to perform basic mathematical operations in order to fulfill the need of numerical-software-components for basic mathematical operations or highly advanced scientific or engineering problems [30]. Some of its features include:

Numerical Differentiation
Numerical Integration
Interpolation
Optimization
Root Finding
Ordinary Differential Equations
Real/Complex Vectors
Real/Complex Matrices
System of Simultaneous Linear Equations
LU Decomposition
QR Decomposition
Singular Value Decomposition
Eigenvalue Decomposition
Fractional Numbers
Complex Numbers
Quaternions
Bessel Functions [First/Second Kind]
Chebyshev Polynomials [First/Second Kind]
Legendre Polynomials
Laguerre Polynomials
Hermite Polynomials
Gamma Function
Permutations
Combinations
Normal Distribution
Chi-Square Distribution
Hyperbolic Trigonometric Functions
Sorting Algorithms

5.4.1 Concerns:

The intention to integrate this component was the same:

- To explore the integration time semantic mismatches and develop the wrapper code for them
- To analyze the portability of the component by verify about either the component permits or not, the easy enhancement of semantics regarding data exchange

A full 30 days trial version of fully functional "Scinet Math" was used for this purpose.

5.4.2 Basic Course of actions: Narrative style:

"Scienet Math" was integrated in the same as the case of EONL, with "MathType". (Figure 8)

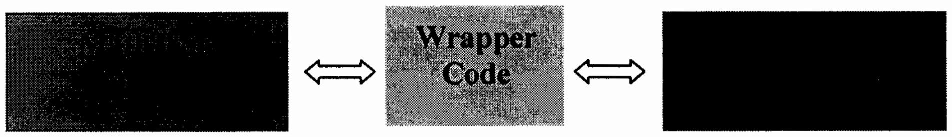


Figure 8: Scinet Math Integration: Abstract Level

The same no. and type of errors were encountered except the there was some support for the "Angle Mode" parameters in a way that Scinet Math supported Degree and Radian semantics, however the semantic mismatches again occurred while dealing with "Grads" semantic. Also there was no support for user to define its own semantics. So the wrapper classes were written in the same way as in case of EONL.

5.4.3 Wrapper Code for Scinet Math:

Almost same amount of wrapper code was required to overcome the semantic mismatches. For this purpose the same wrapper classes that were written for EONL were reused except with some modification due to some extent of semantic support provided in Scinet Math.

However "CDataConv" class was not required as there were no data format mismatch.

5.5 Case Study – 3:

In this case study the final Off the Shelf component "NMATH" by "CenterSpace" [31] integrated with MathType.

"The NMath Suit, intended for .Net platform, provides the basic building-blocks for engineering, financial, mathematical, and scientific applications. Matrix and vector classes, random number generators, statistics, linear algebra, multiple linear regression, numerical integration methods, optimization, interpolation, biostatistics, analysis of variance (ANOVA), and object-oriented interfaces are some of the features included" [31].

5.5.1 Concerns:

NMath was the final case study that was implemented, to highlight the semantic mismatches, regarding Commercial Off the shelf components and the results that were analyzed. A no. of semantic mismatches (Table-2) were occurred for which wrapper code was required. Also no support for defining the semantics by user was found in the component.

5.5.2 Wrapper Code Generated

The same amount of wrapper code had to be re-written however as we had already developed the wrapper code so those wrapper classes were used again. All the classes defined in Table-3 were reused. Also no "CDataConv" class was required as there was no data type mismatch.

5.6 eCalculus:

eCalculus was the in-house developed Off-The-Shelf mathematical component, integrated to verify the proposed solution. The architecture of this component was based on the model that was complied with the proposed design.

Possible semantics were integrated as subcomponent with in "eCalculus", in order to avoid semantic mismatches, and consequently minimize the external mediation.

Further more the scope of semantics with in eCalculus was kept open to enhance the semantic thesaurus with minimum effort.

5.6.1 Application of Proposed Solution:

5.6.1.1 Semantic Thesaurus:

Semantic thesaurus was implemented using flat files, all the possible semantics were saved in thesaurus using (Key, Value) structure.

e.g. in case of angle mode the thesaurus had following entries:

0	1	2	3	4	5
AM_DEG	AM_RAD	AM_GRAD	AM_CUST_	AM_CUST_	AM_CUST_
1	0.0174532	1.111	-	-	-

AM_DEG key at index "0" represents the default mode of the component i.e. component performs all the trigonometric calculations in degree mode. AM_RAD key at index "1" represents the value of Radian in one degree, so if any parameter is received in Radian mode, it will be converted to default angle mode using the Keys "AM_DEG" and "AM_RAD" by EST component's Conflict Resolution and EST control mechanisms. Same is the case with AM_GRAD which shows the angle mode in Gradients. For the purpose of adding new semantics user only needs to use CU's user interface to add the semantics before using it, as all other responsibility is that of the EST component to add those semantics and update the semantic thesaurus.

"AM_CUST_" keys at indexes 3, 4, 5... indicate the open scope of thesaurus where new semantics can be added by the user. However the semantics must be defined in terms of default angle mode of component i.e. any value of custom semantic defined by user must be equal to 1 degree.

As eCalculus uses MOD_DEG as the default angle mod for trigonometric functions, so all the parameters for trigonometric functions that will be in "degree mode", they will require no conversion in their mode. However, all the other angle modes will be first converted in "degree mode" before applying any trigonometric function on them. But one thing the worth noting is, that any parameter received in angle mode other than "degree" will not be considered as a semantic mismatch, unlike the mathematical components used in previous case studies, so no external code will be required as the EST component's mismatch resolution mechanism will resolve the conflict by converting it to "degree" mode by itself, after reported by CU.

Similarly the semantic thesaurus for Number Systems was defined:

0	1	2	3	4	5
NS_BIN	NS_OCT	NS_DEC	NS_HEX	NS_CUST_	NS_CUST_
2	8	10	16	-	-

Here the key "NS_BIN" represents the binary number system and the value of "2" of this key represents the base. Similarly the key "NS_OCT" represents the octal number system and the value "8" of this key represents the base.

The keys "NS_CUST_" at indexes 4, 5... represent the open scope of thesaurus where user can add its own semantics.

5.6.1.2 Thesaurus Control Interface:

Component's Control Unit (CU) uses this interface to perform the thesaurus control tasks. E.g. addition of new semantics, resolution of conflicts et.c

Some of the sample calls by CU are:

```
TCI::sem_Add ("AM_CU_MAM" , "0.33476");
TCI bool::sem_IsSemanticMismatch ("AM_GRAD")
TCI double::sem_Resolve ("AM_GRAD" , "2.34");
```

Sem_Add (...) function of TCI interface is used to add new semantics to semantic dictionary. The parameters are sent in the "Key, Value" notation. Here the new semantic "AM_CU_MAM" is defined, which shows the new angle mode "My Angle Mode" and the value "0.33476" shows that 1 degree = 0.33476 MAM.

sem_IsSemanticMismatch (...) is used by CU to confirm that whether the parameter received will cause a semantic mismatch or not. And in case the true value is returned the further call to "sem_Resolve" is made which resolves the mismatch by taking as input the value of parameter and converting it to default angle mode, CU then passes the parameter to MFC for required processing.

5.6.1.3 Component Control Unit (CU):

Component's Control Unit (CU) is the traditional sub-component that intercepts all the services requests to component. It validates all the data exchanged to/from component. In case any mismatch is detected it then passes the data to EST component and after receiving the validated data from EST component it then forwards the call to MFC.

Similarly before returning the data to the calling application, CU again request EST component to change the semantics w.r.t. calling application.

Along with semantic validation CU performs all the other validations like NULL values, extreme values etc.

For all the calls to component CU provides external interface, which can be used by other application to invoke all the functionality provided by the component. CU internally links to EST component and MFC.

5.6.1.4 Main Functionality Component:

MFC is the main component that provides the domain functionality, however the user cannot pass the service request directly to MFC rather MFC receives all the requests by CU. MFC always operates in its default mode regarding semantics, however any service call with data other than the default semantics is first forwarded to EST component and then to MFC, which makes sure that MFC operates on data that is semantics mismatch free. Similarly after performing the intended functionality all the results are returned to CU which again after validating the semantics (w.r.t target application) returns to calling application.

5.7 Sample source code:

Following are some snippets of source code from case studies

5.7.1 CDec class:

The following code shows the "CDec" wrapper class which is responsible for converting the data with decimal semantics to any other number system.

```
Public Class CDec
    Public Function Dec2Bin(ByVal dec As Double) As String
        Dim dval As Integer
        Dim bval As String
        bval = ""
        dval = Normalize(dec)
        Do While dval > 0
            bval = bval & (dval Mod 2).ToString
            dval = (dval \ 2)
        Loop

        Dec2Bin = Invert(bval)
    End Function
    Public Function Normalize(ByVal dec As Double) As Integer
        Dim temp, normval As Integer
        Dim s As String
        s = dec.ToString
        temp = dec.ToString.IndexOf(".")
        If temp > 0 Then
            s = dec.ToString.Substring(0, temp)
        End If
        normval = System.Convert.ToInt64(s)
        Normalize = normval
    End Function

    Function Invert(ByVal str As String) As String
        Dim charstr() As Char = str.ToCharArray
        Array.Reverse(charstr)
        Return charstr
    End Function
    Public Function Dec2Oct(ByVal dec As Double) As String
        Dim dval As Integer
        Dim oval As String
        oval = ""
        dval = Normalize(dec)

        Do While dval > 0
            oval = oval & (dval Mod 8).ToString
            dval = (dval \ 8)
        Loop
        Dec2Oct = Invert(oval)
    End Function
    Public Function Dec2Hex(ByVal dec As Double) As String
        Dim dval, temp As Integer
        Dim hval As String
        Dim es() As Char = {"0", "1", "2", "3", "4", "5", "6", "7",
"8", "9", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
"M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"}

        hval = ""
        dval = Normalize(dec)

        Do While dval > 0
            temp = dval Mod 16
            hval = hval & es(temp)
            dval = (dval \ 16)
        Loop
        Dec2Hex = Invert(hval)
    End Function
    Public Function Dec2NSn(ByVal dec As Double, ByVal radox As
Integer) As String

        If (radox > 35) Then
            Return -1
        End If
        Dim dval, temp As Integer
        Dim nval As String
        Dim es() As Char = {"0", "1", "2", "3", "4", "5", "6", "7",
"8", "9", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L",
"M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"}

        nval = ""
        dval = Normalize(dec)

        Do While dval > 0
            temp = nval Mod radox
            nval = nval & es(temp) 'GetEqualSymbol(temp)
            nval = (nval \ radox)
        Loop
        Dec2NSn = Invert(nval)
    End Function
End Class

Public Class wCOct
    Public Function Oct2Dec(ByVal dec As String) As String
        Oct2Dec = Convert.ToInt32(Long.Parse(dec), 8)  'correct
    End Function

    Public Function Oct2Hex(ByVal dec As String) As String
        Oct2Hex = Dec2Hex(Oct2Dec(dec)).ToUpper()  'correct
    End Function
    Public Function Oct2Bin(ByVal dec As String) As String
        Oct2Bin = Dec2Bin(Oct2Dec(dec)) 'correct
    End Function
    Public Function Oct2NSn(ByVal dec As String) As String
        Oct2NSn = Dec2Bin(Oct2Dec(dec)) 'fake
    End Function
    Public Function Normalize(ByVal dec As Double) As Integer
```

```
        Dim temp, normval As Integer
        Dim s As String
        s = dec.ToString
        temp = dec.ToString.IndexOf(".")
        If temp > 0 Then
            s = dec.ToString.Substring(0, temp)
        End If
        normval = System.Convert.ToInt64(s)
        Normalize = normval
    End Function
    Function Invert(ByVal str As String) As String
        Dim charstr() As Char = str.ToCharArray
        Array.Reverse(charstr)
        Return charstr
    End Function
End Class

Public Class wCBin
    Public Function Bin2Hex(ByVal dec As String) As String
        Bin2Hex = Dec2Hex(Bin2Dec(dec)).ToUpper()  'correct
    End Function
    Public Function Bin2Dec(ByVal dec As Double) As String
        Bin2Dec = Convert.ToInt32(dec, 2)   'correct
    End Function
    Public Function Bin2Oct(ByVal dec As String) As String
        Bin2Oct = Dec2Oct(Bin2Dec(dec))   'correct
    End Function
    Public Function Normalize(ByVal dec As Double) As Integer
        Dim temp, normval As Integer
        Dim s As String
        s = dec.ToString
        temp = dec.ToString.IndexOf(".")
        If temp > 0 Then
            s = dec.ToString.Substring(0, temp)
        End If
        normval = System.Convert.ToInt64(s)
        Normalize = normval
    End Function
    Function Invert(ByVal str As String) As String
        Dim charstr() As Char = str.ToCharArray
        Array.Reverse(charstr)
        Return charstr
    End Function
End Class
```

5.7.2 Sample Interaction with EONL through wrapper:

```
Public Sub doOperation()

 Select Case (lastOperator)

 Case "a"

    temps = wrp.Add(getLastScrValue, getScreeni().ToString,
numberSystem)
    clrScr()
    toScreen(temps)


 Case "s"

    temps = wrp.Subtract(getLastScrValue, getScreeni().ToString,
numberSystem)
    clrScr()
    toScreen(temps)

 Case "m"
    temps = wrp.Multiply(getLastScrValue, getScreeni().ToString,
numberSystem)
    clrScr()
    toScreen(temps)
 Case "d"
    temps = wrp.Divide(getLastScrValue, getScreeni().ToString,
numberSystem)
    clrScr()
    toScreen(temps)


End sub
```

Here "wrp" is the object of main wrapper class which receives the data, and then after negotiating the semantics forwards the request to component, the code of the functional wWrapper::Add(...) below shows how wrapper class first changes the semantics and then forwards the request to EONL:

```
Function Add(ByVal fv As String, ByVal sv As String, ByVal sem As
String) As String
        If (sem = BINARY_SYSTEM) Then
            v1.SetValue(bin.Bin2Dec(Val(fv)), 0)
            v2.SetValue(bin.Bin2Dec(Val(sv)), 0)
            v = Vector.Add(v1, v2)
            Add = (dec.Dec2Bin(v.GetValue(0)))
        ElseIf (sem = OCTAL_SYSTEM) Then
            v1.SetValue(oct.Oct2Dec(Val(fv)), 0)
            v2.SetValue(Oct2Dec(Val(sv)), 0)
            v = Vector.Add(v1, v2)
            Add = dec.Dec2Oct(v.GetValue(0))
        ElseIf (sem = HEXADECIMAL_SYSTEM) Then
            v1.SetValue(hex.Hex2Dec(Val(fv)), 0)
            v2.SetValue(hex.Hex2Dec(Val(sv)), 0)
            v = Vector.Add(v1, v2)
            Add = dec.Dec2Hex(v.GetValue(0))
        Else
            v1.SetValue(getLastScrValue, 0)
            v2.SetValue(getScreeni(), 0)
            v = Vector.Add(v1, v2)
            Add = v.GetValue(0).ToString
        End If
    End Function
```

The first two "bold" lines are changing the semantics of data to decimal and the third line is forwarding the request to EONL. However in fourth line, the data is changed again to make it compliance with the semantics of original application and then results are returned back.

5.7.3 Sample Interaction with eCalculus without wrapper:

```
Public Sub doOperation()

        Select Case (lastOperator)

 Case "a"

   Dim cmp = New CComponent
   Dim ans = cmp.CSum(getLastScrValue(), getNewScrValue(),
numberSystem)
   clrScr()
   toScreen(ans)

 Case "s"
   Dim cmp = New CComponent
   Dim ans = cmp.CSub(getLastScrValue(), getNewScrValue(),
numberSystem)
   clrScr()
   toScreen(ans)


 Case "m"
   Dim cmp = New CComponent
   Dim ans = cmp.CMul(getLastScrValue(), getNewScrValue(),
numberSystem)
   clrScr()
```

```
    toScreen(ans)

  Case "d"
    Dim cmp = New CComponent
    Dim ans = cmp.CDiv(getLastScrValue(), getNewScrValue(),
numberSystem)
    clrScr()
    toScreen(ans)

End Sub
```

Here “cmp” is the object of the component and all the service requests are forwarded to component using the direct methods from interface, as component is itself responsible for handling semantic mismatches so here we have no need of wrapper classes. The following code shows a sample function from “eCalculus” component.

```
Public Function CSum(ByVal firstvalue As String, ByVal secondvalue As
String, ByVal inputmod As Integer) As String

        If (inputmod = HEX_MODE) Then
            Dim fvl = Hex2Dec(firstvalue.ToString())
            Dim svl = Hex2Dec(secondvalue.ToString())
            CSum = Dec2Hex(fvl + svl)
        ElseIf (inputmod = OCTAL_MODE) Then
            Dim fvl = Oct2Dec(firstvalue.ToString())
            Dim svl = Oct2Dec(secondvalue.ToString())
            CSum = Dec2Oct(fvl + svl)
        ElseIf (inputmod = BINARY_MODE) Then
            Dim fvl = Bin2Dec(firstvalue.ToString())
            Dim svl = Bin2Dec(secondvalue.ToString())
            CSum = Dec2Bin(fvl + svl)
        Else
            CSum = (Val(firstvalue) + Val(secondvalue)).ToString()

        End If

    End Function
```

The function itself detects the semantics of the data, converts them to component’s default semantics using thesaurus and then performs the intended functionality, after this the data is again converted to semantics of target application and the results are returned back. This mechanism clearly shows how we have eliminated the intermediate mediation mechanism by shifting the responsibility of semantic mismatch handling to component.

Chapter - 6

Validation

In order to verify the result of case studies, the wrapper code written in each case was compared. The number of "wrapper classes" written was used to conclude about the size of the code and finally this metric was provided as an evidence to prove the validity of the proposed solution.

6.1 Wrapper code size and Customizability:

The number of "wrapper classes" written in each case was used as source to conclude the size of the code. Each wrapper class contained the LOC containing 70 to 100 SLOC. However the wrapper required in "eCalculus" was of minimum and included the only class "CDataConv" to convert the data formats as was required in case of EONL along with an additional parameter to "eCalculus" function calls, including the semantics of parameter in "Key, Value" form, e.g. the following function call shows the request for the Arch Cosine of the specified value.

```
IExtern::tig.cos("0.345", "MOD_RAD");
```

Here the angle value is provided in "Radian" Mod and the key "MOD_RAD" specifies the component to calculate the results in Radians.
The following table (Table-6) shows the comparison of wrapper classes (code) that were written for each of the case study

Table-6: wrapper code comparison

Data Exchanged	Data Semantics	Wrapper Class	EONL	Scinet Math	NMATH	eCalculus
Number System	Binary	CBin	✓	✓	✓	×
	Octal	COct	✓	✓	✓	×
	Decimal	CDec	✓	✓	✓	×
	Hexadecimal	CHex	✓	✓	✓	×
	N-Number System	CNmN	✓	✓	✓	×
Angle Mode	Degree	CDeg	✓	✓	✓	×
	Radians	CRad	✓	✓	✓	×
	Grad	CGrad	✓	✓	✓	×
	N-Angle Mod	CAmN	✓	✓	✓	×

However in case of "Scinet Math" the classes "CDeg" and "CRad" contained less SLOC due to some support provided for Degree and Radian modes in Scinet math.

The lesser no of wrapper classes means the minimum the size of wrapper code. Here the size of wrapper classes was almost same.

Furthermore some state of the art metrics were used to compare and the results in context of the size of wrapper code and customizability of the components based on the proposed solution:

Table-7: Table of Metrics

Factors and Metrics		**Components**			
Factor	**Metrics**	**EONL**	**Scinet Math**	**NMATH**	**eCalculus**
Customizability	RCC	0	0.5	0	1
Size of wrapper code	FP	2.53	2.35	2.53	0.36
	KSLOC	0.7	0.65	0.7	0.10
	WMC	4.5	5	5	2

Rate of Component's Customizability (RCC):

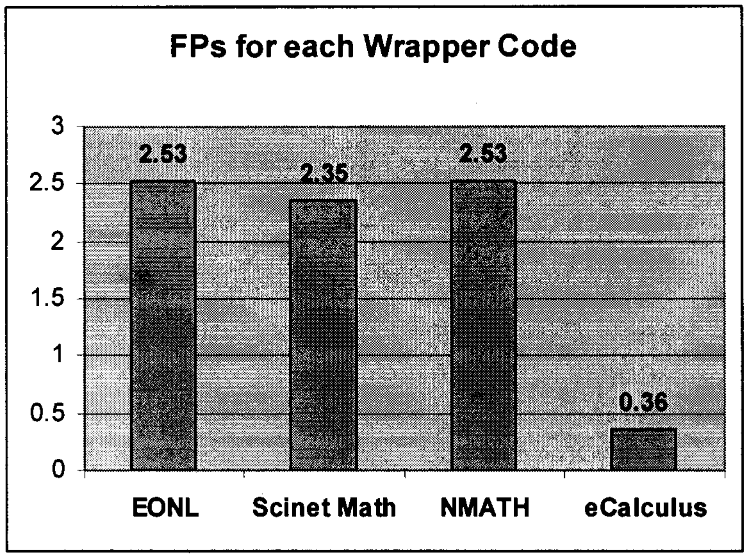
The measure RCC [33] shows, how easy it is to customize a component. It is basically the percentage of writable properties in a component.

RCC = $P_w(c) / A(c)$

Where $P_w(c)$ = Writeable properties in component and A(c) = Total number of properties. The results show the highest value of $P_w(c)$ for "eCalculus", which ultimately proves that "eCalculus" is easy to customize as compared to other, which was one of the objectives of the proposed approach.

Function Points (FP):

This standard metric was basically used to measure the size of wrapper code. Backfiring technique (used to calculate FPs from SLOC) was used to Calculate FPs. The results obtained are shown in the graph given below:



The minimum value of FP obtained for eCalculus clearly shows the minimum effort, required to integrate the components, based on the proposed solution.

Thousand Source Line of Code (KSLOC):

Thousand Source-Line-of-Code metric, when calculated, was found to be having minimum valule "0.10" for "eCalculus", which is a clear proof that we need minimum intermediate mediation which integrating COTS based on proposed solution.

Weighted Methods Per Class (WMC):

As Object Oriented Approach was used to write the wrapper code, WMC shows the average size of each wrapper class. In case of "eCalculus", the average value of WMC is "2", almost half of all the values obtained in each other case. So the size of intermediate wrapper classes, was minimum for "eCalculus" as compared to all other components.

After analyzing the above metrics and their values, we can safely conclude about the proposed approach that it requires the minimum amount of intermediate mediation work and still enhancing the customizability of the component.

6.2 Customization through configuration not through adaptation:

Case studies depict yet another characteristic of the proposed solution i.e. it lets us develop the components that can be customized by configuration and not through adaptation.

In case of white box components we do have the source code and we normally modify this code (or write down the wrapper) in order to make the COTS fit for our application, where as in case of black box COTS the wrapper/glue is the only option.

However the grey box COTS do provide us the limited customization yet defining new semantics is missing. However in case of semantics the proposed solution best provides the way to customize the components through configuration without any external mediation even you can define new semantics through configuration which is not possible in traditional COTS design

6.3 Fault tolerance enhances with more and more reuse:

The more the reuse of EST based COTS components the more the EST enrichment, so the fault tolerant capabilities of EST based COTS components get augmented with passage of time and the probability of semantic mismatches eliminate (Figure 9).

This is true not only for the same version of a COTS component integrated in different domains but also for its later versions.

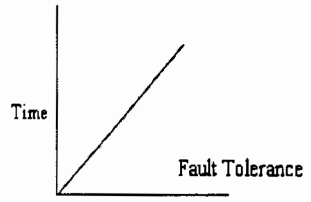


Fig 9: Fault tolerance Vs COTS reusability

6.4 Increase portability:

Portability of the components can be enhanced by implementing EST, as the same component can contain and cope with multiple semantics so the same components can be used in multiple domains with minimum effort. E.g. “eCalculus” is equally useable in many domain (mathematics and other scientific domains) without getting semantic mismatches (Angle Mod and Number System).

6.5 Performance:

An important feature of EST based components is that they provide all the above discussed features without compromising the performance in terms of execution time, i.e. the service time of the COTS will be the same as that of COTS having external wrapper code and adherence of EST will not create any overhead.

However it will not improve the performance further regarding execution time, i.e. the execution time of EST based components and the components without EST (but having wrapper code) will be the same.

6.6 Conclusion:

The proposed solution presents a smooth integrated approach for developing semantic anomalies free wrapper/glue independent COTS components for minimizing integration-time semantic mismatches.

The proposed solution suggests a shift from the traditional COTS design by introducing an Enhanced Semantic Thesaurus (EST) as an essential part of the COTS component. EST stores all the possible semantics of the data exchanged to/from COTS. It enables COTS component to detect the semantic mismatches and automatically resolve it, thus letting the component to participate in mismatch resolution process by itself. Hence the burdensome of developing the wrapper/glue for resolving semantic mismatches shifts from the developer to COTS component. The proposed solution leads us to develop COTS components which could be customized by configuration and not through adaptation. EST also enhances the fault tolerance capabilities of COTS with passage of time due to enrichment of EST, as the COTS is used more and more. Components become highly portable with minimum probability of semantic mismatches, and the most of all is that all of its features are provided without compromising any performance measure.

It provides us further inspiration to develop COTS which could have fault tolerance not only for semantic mismatches but for all types of architectural mismatches highlighted so far.

6.7 Future plans:

In future, I have planned to opt for solving other types of mismatches too so that we could develop such intelligent components which could participate in resolving all types of mismatches (effectively and efficiently without any overhead on component) and hence reducing the intermediate mediation up to maximum. I will try even to develop mechanisms so that the component could expose its complete documentation by itself without any need for paper manuals.

References:

[1] "A domain analysis and modeling methodology for Component development", by E. SOOK C., S. D. KIM and S. Y. RHEW. World scientifific publishing co. international journal of software engineering and knowledge engineering 2004.

[2] "Selecting Software components with multiple interfaces" by L. Iribarne and J. M. Troya and A. Vallecillo. Proceedings of the 28th Euromicro Conference, 2002 IEEE.

[3] "Modernizing Legacy Systems", First addition, By R. C. Seacord, P. Daniel, G. A. Lewis. Addison Wesley 2003. ISBN-10: 0-321-11884-7. Page-304

[4] Y. Yang, J. Bhuta, B. Boehm. and D. Port (2005). "Value-Based Processes for COTS-Based Applications." IEEE-Software Special Issues on COTS-Based Development, Volume 22, Issue 4.

[5] "No Silver Bullet: Essence and Accidents of Software Engineering." By Jr. F. Brooks (1987), IEEE Computer, Volume 20, Issue 4.

[6] "Attribute-Based COTS Product Interoperability Assessment" by J. Bhuta, B. Boehm. Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems 2007 (ICCBSS'07), pp.163-171.

[7] Software engineering with reusable components, by Johannes Sametinger, 1997, ISBN 3-540-62695-6 Springer-Verlag page no. 11

[8] "Supporting Component and Architectural Re-usage by Detection and Tolerance of Integration Faults" by Martin Jung and Francesca Saglietti. Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05) 2005. pp.47-55

[9] "Architectural Mismatch or why it's hard to build systems out of existing Parts" by D. Garlan, R. Allen and J. Ockerbloom. Proceedings of the 17th international conference on Software engineering (ICSE-1995) 1995. Pp. 179.

[10] component based softwrae engineering by George Heineman T. and Wiolliam T. Councill, Addison-wesley, 2001

[11] "Software engineering a practitioner's approach" by R. S. Pressman. 5th edition. ISBN 0073655783. Mcgraw-Hill. Page. 721.

[12] Clements, P.C., "From subroutines to Subsystems: Component based software development," American-programmer, Vol. no. 8, No. 11, November-1995.

[13] Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition by Bachmann et al. Technical report, CMU/SEI-2000-TR-008, ESC-TR-2000-007 (May 2000)

[14] "Temporal logic based specification of component interaction protocols" by J. Han, Monash University: School of Network computing. Proceedings. of the ECOOP-2000 Workshop on Object-Interoperability (WOI'00), June-2000, pages 43–52.

[15] R. Weinreich, J. Sametinger, Component Models and Component Services: Concepts and Principles, Chapter 3 of Component-Based Software Engineering: Putting the Pieces Together, George Heineman, Bill Councill (eds.), Addison-Wesley, pp. 33-48, June 2001.

[16] "Architectural Mismatch: Why reuse is so hard", by D. GARLAN, R. ALLEN, and JOHN O. *IEEE Software*, vol. 12, no. 6, pp. 17-26, Nov. 1995, doi:10.1109/52.469757.

[17] "A Classification of Software Component Incompatibilities for COTS Integration." By Yakimovich D., Travassos G., and Basili V. (1999), Proceedings of 24th Software Engineering Workshop.

[18] "A systematic methodology for developing components frameworks" by S. Choi, S. Chang, and S. Kim, Proceedings of the 7th Fundamental Approaches to Software Engineering Conference, 2004 , LNCS 2984, pp. 359–373.

[19] "Classifying Software Component Interoperability Errors to Support Component Adaptation" by Steffen Becker, Sven Overhage and H. Reussner. Springer-Verlag Verlin Heidelberg 2004.

[20] "UnSCom: A Standardized Framework for the Specification of Software Components" by O. Sven. Weske M. and Liggesmeyer P. (Eds.): LNCS 3263, pp. 169–184, 2004.

[21] "An idealized fault-tolerant architectural component" P. Asterio de C. Guerra, Rogerio de lemos. Workshop on Architecting Dependable Systems, Orlando, FL, May 2002.

[22] "Architectural Mismatch in Service-Oriented Architectures" by B. Kevin, G. Mark and Edy Liongosari S. Proceedings of the International Workshop on Systems Development in SOA Environments 2007. ISBN:0-7695-2960-7. Page 4.

[23] "Towards a taxonomy of architecture integration strategies" by Keshav R. and Gamble R. Third International Software Architecture Workshop, Nov. 1-2, 1998.

[24] "A Catalog of Techniques for Resolving Packaging Mismatch" by Robert DeLine. Proceedings of the 1999 symposium on Software reusability, May-1999, Los Angeles, CA.

[25] "Variability design and customization mechanisms for COTS components" by S. D. Kim et. al. International Conference on Computational Science and Its Applications 2005. LNCS 3480/2005, pp. 57-66, 2005.

[26] Mars climate Orbiter Mishap Investigation Board Phase-1 Report, 1999. Stephenson AG.

[27] "Finding Errors in components That Exchange XML Data" Mark G., Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE-2007), November 5-9, 2007.

[28] "Analysis of Compositional Conflicts in Component-Based Systems" by Andress Leicher et al. 4th international workshop, SC 2005. LNCS 3628, pp. 67-82

[29] http://www.extremeoptimization.com

[30] http://www.obacs.com

[31] http://www.centerspace.net

[32] "Function points languages table" version 3.0, April 2005, by "Quantitative Software Management", Inc. http://www.qsm.com/resources/function-point-languages-table/index.html

[33] "A Metrics Suite for Measuring Reusability of Software Components" by W. Hironori et all., Ninth International Software Metrics Symposium (METRICS'03), 2003. pp. 221.