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

Muhammad Summair
Raza
International Islamic University, Islamabad
summairpk@gmail.com

Shafqat Hussain
Majoka
International Islamic University, Islamabad
shafqat.majoka@gmail.com

Ahmad Mohsin
National University of Computer & Emerging Sciences, Lahore
ahmadspm@gmail.com

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 component-based-software-engineering (CBSE) have introduced 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 paper 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.

KEYWORDS
Components, COTS, CBSE, semantic mismatches, enhanceable semantic thesaurus (EST)

1. INTRODUCTION
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. However despite the benefits of reuse, there are factors that directly or indirectly influence the success or failure of reuse. These factors may be of technical, conceptual, organizational, managerial, economic, psychological, or legal nature [1]. 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 [2] called “Architectural Mismatches” [3]. Resulting from the mismatched assumptions a reusable part makes about the structure of the target system arise due to the mismatched assumptions a reusable part makes about the structure of the target system [4], these mismatches not only lead towards grave consequences, but also require some intermediate mediation mechanism, the
2. Architectural Mismatches: Classification

The term “Architectural Mismatch” was first coined by Garlan et. all in [3]. While developing AESOP, they came across six main difficulties during integration of four existing software subsystems (components) into a new coherent system: extensive code size, poor performance, extensive modification required to get reused subsystems to work together, the necessity of reinventing existing functionality to meet the intended use, redundant complexity of applications built from reused systems, and a complex, error-prone system development process [4]; 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 [4]. Four guidelines were proposed by them [4] 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 [5] Yakimovich et al. 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 [2] 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 to external resources, timing requirements and other architectural constraints etc.

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

<table>
<thead>
<tr>
<th>Semantic</th>
<th>App-based</th>
<th>Pragmatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. language</td>
<td>1. violation of state/input relation</td>
<td>1. violation of absolute time constraints</td>
</tr>
<tr>
<td>2. numerical</td>
<td>2. data range</td>
<td>2. violation of concurrency constraints</td>
</tr>
<tr>
<td>3. physical</td>
<td></td>
<td>3. violation of architectural constraints</td>
</tr>
</tbody>
</table>

Keshav et. al [6] discuss their preliminary findings 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 functions between different component formats; however contents of the information are not changed. It does not need knowledge of source/destination of data.

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

An **Extender** adds new features and functionality, and hence augments component capability. Whether or not the Extender needs to have the knowledge of component identity with which it interacts, depends upon specific application.

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

Robert DeLine in [7] provides a catalog of tools and techniques to negotiate packaging mismatch, organized according to underline architectural commitment: on-line and off-line bridges, wrappers, intermediate representations, mediators, unilateral and bilateral negotiation, and component extension. All these techniques 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 this method a new component (Br - bridge component) is introduced 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 some form of 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 (often about data representation). Components A and B can use any of the provided commitment to interact with each other.

**Intermediate Representation**: just like Mediator technique, with the restriction that the mismatch between A and B is caused by data representation. Connector C can support several alternatives (interfaces) for a given commitment about 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 for selecting one of the alternatives by negotiating 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.

3. Enhance able Semantic Thesaurus (EST)

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 [8] 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 same porting the 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 challenging to be able to customize black box components without accessing their internal design and source code [9]. However, this is the only solution so far to overcome the problem.

Resolving semantic mismatches using Enhance able Semantic Thesaurus (EST) 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.

3.1 Enhanced domain analysis:

During this phase the target domain of component is analyzed to understand the problem and component requirements. EST implementation 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.

3.2 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 1.

![Figure 1: traditional COTS design](image)

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 2.

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.
4. Features of 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.

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.

5. 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.
Main parts of EST component Structure are (Figure 3):

5.1 Semantic Thesaurus:
It is basically the repository of all the semantics that were identified during Enhanced domain analysis phase (figure 4). 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.

5.2 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.

5.3 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.

5.4. 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 4) given below.
6. Data collection and analysis

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.

6.1 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.

6.2. Case Study 1

In first case study a mathematical component “Extreme Optimization Numerical Library (EONL) version 3.0” by Extreme Optimization [10] was integrated with MathType. Table-2 shows the list of classes developed corresponding to each type of semantic mismatch.

<table>
<thead>
<tr>
<th>Data Semantics</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number System</strong></td>
<td></td>
</tr>
<tr>
<td>Binary Class: CBin</td>
<td>Converts data with binary semantic to any other semantic</td>
</tr>
<tr>
<td>Octal Class: COct</td>
<td>Converts data with Octal semantic to any other semantic</td>
</tr>
<tr>
<td>Decimal Class: CDec</td>
<td>Converts data with Decimal semantic to any other semantic</td>
</tr>
<tr>
<td>Hexadecimal Class: CHexl</td>
<td>Converts data with Hexadecimal semantic to any other semantic</td>
</tr>
<tr>
<td>N-Number System Class: CNmN</td>
<td>Converts data with N-Number system semantics to other semantic</td>
</tr>
<tr>
<td><strong>Angle Mode</strong></td>
<td></td>
</tr>
<tr>
<td>Degree</td>
<td>Converts Degree mod to any other mod</td>
</tr>
</tbody>
</table>

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.

6.3. Case Study-2

During this case study a second mathematical component “Scinet Math” by “OBACS” [11] was integrated with “MathType”. 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 (Table-2) were written in the same way as in case of EONL.

6.4. Case Study-3

In this case study the final Off the Shelf component “NMATH” by “CenterSpace” [12] was integrated with MathType. 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-2 were reused. Also no “CDataConv” class was required as there was no data type mismatch.

6.4. 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.

7. Structure of eCalculus

7.1 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 (Table-3):
AM_DEG key at index “0” represents the default mode of the component i.e. component performs all the trigonometric calculations in degree mode.

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 (Table-4):

**Table-4: Thesaurus for Numeric System**

<table>
<thead>
<tr>
<th>Index</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NS_BIN</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>NS_OCT</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>NS_DEC</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>NS_HEX</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>NS_CUS</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>NS_CUS</td>
<td>-</td>
</tr>
</tbody>
</table>

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_CUS_” at indexes 4, 5, … represent the open scope of thesaurus where user can add its own semantics.

7.2 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 etc.

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.
7.3 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.

7.4 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.

8. Result verification:
In order to verify the result of case studies, the wrapper code written in each case was compared (Table-6). 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.

8.1 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.
Table-5 shows the comparison of wrapper classes (code) that were written for each of the case study.

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 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:

8.2 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 = Pw(c) / A(c)
Where Pw(c) = Writeable properties in component.
The results show the highest value of Pw(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.

8.3 Function Points (FP):
This standard metric was basically used to measure the size of wrapper code. The results obtained are shown in the graph given below (Figure 5):

Table-5: wrapper code comparison

<table>
<thead>
<tr>
<th>Data Semantics</th>
<th>EONL</th>
<th>Scinet Math</th>
<th>NMATH</th>
<th>eCalculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number System</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Class: CBin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Octal</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: COct</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CDec</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CHex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Number System</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CNmN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Angle Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Class: CDeg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radians</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CRad</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grad</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CGrad</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Angle Mod</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class: CAMN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The minimum value of FP obtained for eCalculus clearly shows the minimum effort, required to integrate the components, based on the proposed solution.
Table-6: Table of Metrics

<table>
<thead>
<tr>
<th>Factors and Metrics</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EONL</td>
</tr>
<tr>
<td>Customizability</td>
<td>RCC</td>
</tr>
<tr>
<td>Size of wrapper code</td>
<td>FP</td>
</tr>
<tr>
<td></td>
<td>KSLOC</td>
</tr>
<tr>
<td></td>
<td>WMC</td>
</tr>
</tbody>
</table>

8.4 Thousand Source Line of Code (KSLOC):

Thousand Source Line of Code metric, when calculated, was found to have minimum value “0.10” for “eCalculus”, which is a clear proof that we need minimum intermediate mediation which integrating COTS based on proposed solution.

8.5. 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.

9. 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 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.

9.1 Future plans:

Endeavors are in progress 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. Efforts will be made even to develop mechanisms so that the component could expose its complete documentation by itself without any need for manuals.

References:

