JOURNAL OF AERONAUTICS AND SPACE TECHNOLOGIES
JUNE 2007 VOLUME 3 NUMBER 2 (25-36)

USING A META-LANGUAGE TO BRIDGE THE GAP BETWEEN NATURAL LANGUAGES AND COMPUTER LANGUAGES

Selim TEMİZER

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Artificial Intelligence Laboratory
Cambridge, Massachusetts, USA
temizer@alum.mit.edu

ABSTRACT
Natural languages have vast vocabularies, complex grammars and inherent ambiguities that make them difficult to be processed directly by computers, even with state-of-the-art technology. Therefore, in order to communicate with computers we need to 'develop software', which is actually the very process of translating our problem statements, data and solution algorithms from the languages we speak to the languages that computers speak. But software development and maintenance are costly, time consuming and have many major challenges of their own. In this document we present a group of techniques and tools, collectively named as Temizer Description System, that aim to bridge the gap between natural languages and computer languages by enabling computers to understand the logical structure of natural language texts. The main idea is to tag texts piece by piece in order to make them semantically meaningful to the computers. Once computers start figuring out the meaning of text chunks, they can also use the same chunks to talk back to us and we demonstrate how this new and effective way of communication could be used to automate (i.e. eliminate) many tedious and error-prone aspects of developing and maintaining software.

Keywords: *Natural language processing, meta-language, verification/validation, requirements, DO-178B*

1. INTRODUCTION

Contrary to the fact that we humans built computers to aid us in almost infinitely many ways, we have not yet been able to teach them the way we communicate. We make statements, describe problems, and in general *speak* in *Natural Languages* (NL) like Turkish, but we need to translate our problem statements, data and solution algorithms to some *Computer Languages* (CL) before they are processed by computers.

It would be great if computers were able to decipher NL and we could communicate with them directly, but NL have vast vocabularies, very complex grammars and inherent ambiguities that make them practically unsuitable for computers. To remedy this situation, various computer programming languages such as *HyperTalk*, *Lingo*, *AppleScript*, *SQL* and *Inform* have been designed that resemble NL, and programs written in one of these languages may roughly be understood by a person that has no prior knowledge about the language [1]. However, this does not mean that writing programs in these languages are easy since compilers and interpreters usually have low tolerance to alternative sentence structures, synonyms, etc. We therefore have two sides, namely NL and CL, and although there is no trivial solution, it is highly beneficial to bring them as close to each other as possible. The situation is depicted in Figure 1: At the top, there is the humans' realm where NL are spoken. At the bottom we have the computers speaking CL. When we make a statement in NL and translate that statement into CL, our goal is to make sure that the two have exactly the same meaning. In other words, we want *maximum traceability* between them.

On the NL side, the simplest forms of expressions are *verbal descriptions*. Usually they are cast as *formal requirements* to be more manageable and easier to translate to CL. These requirements might sometimes be organized hierarchically going from less detailed to more detailed such as *system level requirements*, *high level requirements* and *low level requirements*. In that case, traceability among these levels must also be ensured. Verbal descriptions are also usually packaged as *use cases* which describe the functionalities that the customer expects on an item by item basis. To make them official, these use cases are usually signed by both the customer and the contractor responsible for translating them into CL. There are also other methods available that shape up raw verbal descriptions and move them closer to the CL side.

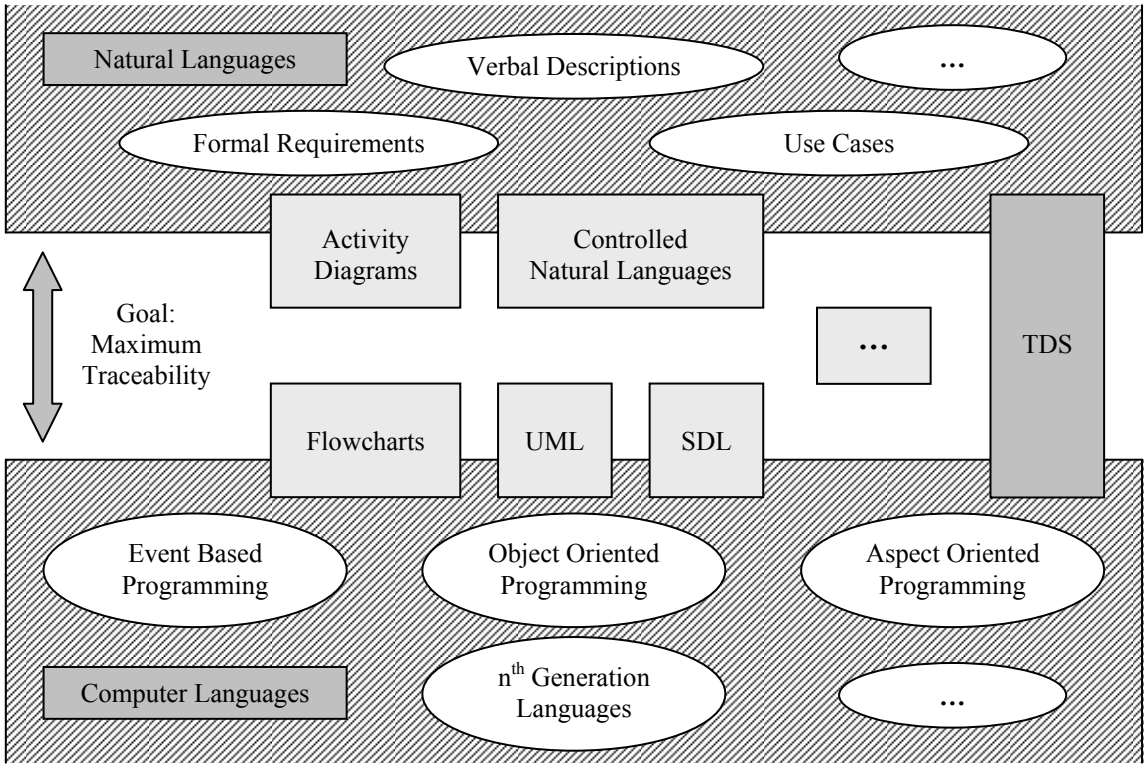

Figure 1. Natural languages, computer languages and workarounds to reduce the gap between them

On the CL side, basic means of speaking to computers are *5 generations of programming languages* which are listed below with some examples:

1. Machine language
2. Assembly language
3. High-level programming languages (C, C++, Java)
4. Languages closer to NL than typical high-level languages (Lingo, SQL)
5. Languages used for artificial intelligence and neural networks (Lisp, Prolog)

On top of these languages (and sometimes mingled within their grammar) are some paradigms such as *event based*, *object oriented* and *aspect oriented programming*, that aim to provide additional structure to these programming languages in order to make them more comprehensible and natural to humans and thus to nudge them closer to NL side. There are also other techniques and paradigms for that same purpose that we have not mentioned here.

Aside from the efforts within the NL and CL sides, there are external workarounds to shorten the distance between them. For example, we use *activity diagrams* (together with state and interaction diagrams) to organize NL statements into forms that resemble CL constructs. Also there are scientific studies to restrict grammars and dictionaries of NL in order to reduce or eliminate ambiguity and complexity (for example at Macquarie University, Australia [2]). These subsets of NL are called *controlled natural languages* and they serve as much better candidates to be processed by the computers. Some examples of controlled natural languages are Attempto Controlled English (ACE) [3], PENG (Processable ENGlish) [4], Common Logic Controlled English (CLCE) [5] and The KANT Project [6].

To reduce the gap between NL and CL, we also have techniques and tools that extend from CL to NL side. For example, we use *flowcharts* to turn textual computer programs into graphics and as we all know 'a picture is worth a thousand words'. We might also design our models independent of any programming language in easy to use specification languages like *Unified Modeling Language (UML)*. In that case, we might employ various tool suites that take our UML specifications and generate associated program code in the programming language of our choice. There are also some graphical programming languages such as the *Specification and Description Language (SDL)* which let us visually design our programs and free us from most of the remaining chores of programming.

In the rest of this document we will present a group of techniques and software tools, collectively named as Temizer Description System (TDS), that aim to bridge the gap between NL and CL. At the center of TDS lies a simple but an extremely powerful meta-language that is called Temizer Description Language (TDL). In a nutshell, TDL is used to tag natural language texts and the software tools are used to parse and process those texts in various different ways.

In the following sections, we will first talk about the motivation behind TDS. Then we will describe all aspects of the problems that we would like to solve. After that, a formal definition of our solution, namely TDS, with extensive implementation details will follow, and we will conclude our discussion after an assessment of TDS.

2. MOTIVATION BEHIND TDS

The original problem that TDS was designed as a personal hobby to address was a reverse engineering project that involved analyzing, debugging and documenting a large amount of previously developed software for a foreign avionics system. In addition, the documentation was expected to be very detailed in order to meet criteria recommended by the DO-178B [7] specification (Federal Aviation Administration of USA, FAA, accepts use of DO-178B as a means of certifying software in avionics).

In such projects and generally in every software development project, documenting software is very tedious and highly prone to errors. As an example, let us assume that we are given a function written in C programming language that describes the behavior of a student depending on the state of the school library and the amount of money that s/he has. The function is shown in Figure 2 without any details (definitions of enumerations, invoked functions, etc.).

```
Student (char library, int money)
{
  if (library) checkout(MATHBOOK);
  else borrow(MATHBOOK);

  if (money > 50) eatAt(RESTAURANT);
  else eatAt(HOME);
}
```

Figure 2. Sample function ‘Student’

Aside from irrelevant implementation details such as types of local variables, the explanation that describes how the function works and that shall be documented about this function is shown in Figure 3.

```
If   { Library is open      }
Then { Check out math book  }
Else { Borrow friend’s book }

If   { There is enough money     }
Then { Have dinner at restaurant }
Else { Cook dinner at home       }
```

Figure 3. Explanation of function ‘Student’

There are various ways to formalize that explanation. One way is to draw *activity diagrams* just like the one shown in Figure 4.

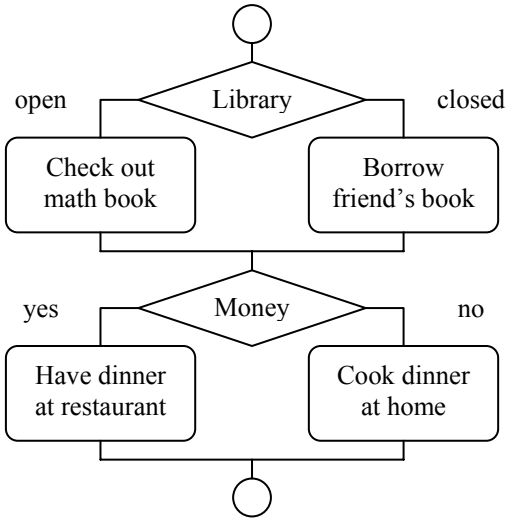

Figure 4. Activity diagram for the function ‘Student’

Another way is to treat the function like a *finite state machine (FSM)* and analyze it thoroughly to identify all possible *conditions* and *actions*, and then document all possible execution paths, or *transitions*, that could be taken by an invocation of the function. An example of such a formal analysis is shown in Figure 5.

List of all conditions
C1. Library is open
C2. Library is closed
C3. There is enough money
C4. There is not enough money

List of all actions
A1. Check out math book
A2. Borrow friend’s book
A3. Have dinner at restaurant
A4. Cook dinner at home

List of all possible transitions
T1. If C1 and C3 hold then take A1 and A3
T2. If C1 and C4 hold then take A1 and A4
T3. If C2 and C3 hold then take A2 and A3
T4. If C2 and C4 hold then take A2 and A4

Figure 5. List of transitions for the function ‘Student’

Unfortunately, if there are hundreds of functions to be dealt with, then drawing activity diagrams becomes an extremely tiring solution. And an FSM analysis is absolutely not practical for especially long functions because the number of possible transitions increases exponentially with each branching within the function body. Actually, it is not very uncommon to have over 1000 possible transitions for a function that has just 20-25 lines of code (for example a dispatcher function that just controls the flow of execution based on some conditions might have many cascaded branches). And what doubles the pain of documenting some software

is *maintaining* and *updating* the documentation in parallel as the code evolves over time.

The foundations of TDS were laid upon the following question: 'If we were to manually translate a computer program from some CL to NL piece by piece (locally) without worrying about the overall (global) structure and semantics of the whole program code, and quickly scribble something like the one shown in Figure 3, can we then use this translation to automatically generate activity diagrams, transition lists or any other forms of documentations that we want?'

As we present in the following sections, the answer to the above question turns out to be positive. In fact, the research that started as an automatic documentation generation tool ended up as a meta-language and a tool suite that could be used to automatically or semi-automatically perform many software engineering tasks such as generating software requirements, code templates, test scenarios and test code stubs, setting up traceability between verbal requirements and pieces of code, documentation and even helping in structural coverage analysis of the code. In the next section, we describe the problems that TDS tackles, in details.

3. PROBLEMS

At the beginning of the life cycle of any software, we need to have a set of software requirements from the customer and we want them to be crisp, clear and contain no ambiguities. This is a package that is hard to get at once, and we usually need a few rounds of meetings with the customer and/or some prototyping before we can reach a complete mutual agreement.

Then we need to go from NL to CL and translate those requirements into a programming language. While doing this, we need to keep in mind that we might need to set up traceability between the requirements and the code later in the life cycle.

When the coding phase is over and the software is up and running, verification and validation phases are in order, if required by the customer. We need to carefully design test scenarios that cover all aspects of the code, develop our test cases, and exercise the code against the tests to make sure that the developed code functions exactly as the customer wants it to. If the code developed will be deployed in an airborne system or in general it is categorized as safety-critical, then *structural coverage analysis (SCA)* should also be performed on the developed software. Depending on the safety-critical level of the code, SCA requires one or more of statement coverage, decision coverage, condition coverage, condition/decision coverage, modified condition/decision coverage and multiple condition coverage tests to be conducted. SCA not only makes sure that the code does what it is supposed to do, but it also makes sure that the code does not do anything more, the test cases are actually enough to test all aspects of the code, and there are no missing requirements in the requirement set.

Although there are other techniques, SCA is usually conducted on *instrumented* code. *Instrumentation* is a technique where programs such as *VectorCAST™* take the software and inject additional software inside. When we run the tests against the instrumented code, the injected code pieces generate reports that tell us which parts of the software were executed and more importantly which parts were not.

Depending on the safety-critical level of the code, verification and validation activities also usually entail the very difficult and highly time consuming task of setting up one-to-one traceability between the verbal requirements (NL) and pieces of the software (CL). *Traceability matrices* or sometimes ad hoc registering methods are usually used to document such data.

Finally, after verification and validation, it is usually required by the customer that some user manuals or other documentation about the software be prepared.

All the above tasks and difficulties inherent in them are for regular forward engineering applications where we go from NL to CL. Let us also check out some of the tasks when we are up against a reverse engineering application, where we go mostly from CL to NL.

We are handed out a huge amount of previously developed software and it needs to be analyzed, debugged, and documented. In such projects, it is usually easy to look at small pieces of code locally and pretty much understand what they do, but it is difficult to visualize how the small pieces fit together to construct the big picture. In such cases, it would be great if we could feed our local understandings and findings into a system, and that system would help us figure out the overall functionality.

Also, when documenting either our own code or code prepared by some others, a very important aspect that we call *capturing software layers* is almost always overlooked. This phenomenon could be explained as follows: Large code pieces are usually prepared by teams of programmers rather than individuals. Usually different programmers are specialized in different areas and they take turns to work on the same piece of software (or at least to review their peers' work to find bugs, make enhancements, etc.). As an example, let us assume that a team starts working on a graphical game played over a network. First, a graphics programmer goes in and constructs the graphics framework. Then an audio specialist injects code that is responsible for game music and effects. A network specialist goes over the code and makes it network enabled. Finally an experienced software engineer checks the code from beginning to end and inserts his own code that

ensures that shared resources are accessed in critical sections, semaphores and mutexes are used properly, there are no memory leaks and possible deadlocks, etc. Even if the whole game was developed by a single programmer, it is evident that different chunks of code serve different purposes.

Although the game software has many *layers* (i.e., there are different groups of statements that relate to different game features), capturing, identifying and documenting those layers is usually not easy. Most programmers put their initials, a date and sometimes the purpose of the code as a comment right above the code piece that they inject, but this is usually not enough (especially in big projects). One can also go by investigating the CVS check-in logs and trying to identify the layers, but this is a very tedious task. In this case, some external system or tool that could keep track of layers in the code for us would be very useful.

Now that we have gone over some problems inherent in both forward (NL to CL) and reverse (CL to NL) engineering applications, we are ready to present our solution, TDS, in the following section.

4. THE SOLUTION: TDS

Temizer Description System consists of two pieces: a meta-language and a suite of software tools that parse, process and exploit the language as much as possible.

4.1 TEMİZER DESCRIPTION LANGUAGE

The meta-language is called Temizer Description Language (TDL) and its main purpose is to describe generic processes and *units* in a structured fashion. The units usually correspond to functions, procedures and/or methods of some software, but it is also possible to define a unit to be something at a higher level than the function level. Hence, for example, we may first describe the functionality of some system in TDL, then hierarchically describe sub-functionalities to any desired level of detail, also in TDL.

The description text itself is in a natural language of our choice. Therefore, the first step to create a TDL description of a unit is to prepare a description of the unit in our preferred natural language. After that, in order to turn this description into a TDL description, we tag the text piece by piece using TDL statements. In other words, in order to make our NL description understandable by computers TDL statements are used to assign semantic meaning to all chunks. Hence the resulting TDL description is actually a mixture of TDL statements and regular text.

Before proceeding any further, let us give a simple example that shows how a TDL description looks like. A TDL description for the sample 'Student' function given above is shown in Figure 6.

```
Student ( Library Money )
{
  Branch
  {  Condition [ Library is open ]
     Action    [ Check out math book ] }
  {  Condition [ Library is closed ]
     Action    [ Borrow friend's book ] }

  Branch
  {  Condition [ There is enough money ]
     Action    [ Have dinner at restaurant ] }
  {  Condition [ There is not enough money ]
     Action    [ Cook dinner at home ] }
}
```

Figure 6. TDL description of function 'Student'

The information that it conveys is as follows: The TDL description in Figure 6 is for a unit named 'Student'. The behavior of the unit is dependent on two parameters, 'Library' and 'Money'. There are two sequential two-way branches within the unit body. Each branch has its own condition and the associated action taken in case the condition holds. Text pieces enclosed within square brackets are the tagged natural language pieces and they can be any expression of any length that we want.

The formal definition of TDL in Backus-Naur Form (BNF) notation is given in Appendix A. TDL is a complete language (provides sequential execution, branching and looping constructs) and it has only a handful of carefully designed and self explanatory statements. Therefore it is almost instantaneous to learn TDL and it stays out of the way as much as possible when applying it to tag regular text pieces.

Currently, there are only six statements (a total of nine keywords) in TDL. Although not shown in the BNF grammar, each TDL keyword also has a 1 or 2 letter abbreviation (acronym) in order to make the language even easier to use. Below are the statements, their acronyms and the nature of text chunks that should be tagged with them:

- ***'Action'***, ***'A'***: Some actual work, an action to be taken. Could pretty much be anything depending on the context.
- ***'Branch'***, ***'B'***: Denotes branching. After this statement a list of one or more conditional statements that describe a different branch of execution follow. Each conditional statement has a ***'Condition'***, ***'C'***, and a list of statements. The condition describes when that branch is to be taken, and the statements tell what happens in that case. For one-way branches (such as an else-less if statement in C programming language), the only conditional statement might also contain an

optional ***'NegativeCondition'***, ***'NC'***, indicating when the branch should not be taken.

- ***'Goto'***, ***'G'***: Denotes an unconditional jump to a labeled statement within the same unit.
- ***'Invoke'***, ***'I'***: Calling another function or procedure could be described by this statement. Usually the number of arguments passed to an invoked unit and the number of parameters of the invoked unit are expected to be the same, and this could be automatically verified when the TDL description is processed by the tools in TDS.
- ***'Return'***, ***'R'***: Very much like the *return* statement in C programming language. Could be used with or without a data parameter.
- ***'Exit'***, ***'E'***: Like 'Return', denotes the end of execution within a unit. Could also be assigned special meanings such as the end of all processing within the whole system.

In addition, each unit could be annotated with a ***'Note'***, ***'N'***, list and these correspond to any number of arbitrary notes, very much like *commenting* in a programming language.

TDL has many advanced features and one of them is *statement grouping*. Each statement can optionally be assigned a user defined group. For example, we may designate a certain tag like 'Safety' and assign it to all safety related statements within a TDL description of some unit as shown in Figure 7.

```
…
Action <Safety> [ Acquire semaphore ]
Action [ Read data from a shared resource ]
Action <Safety> [ Release semaphore ]
Action [ Use the data just read ]
…
```

Figure 7. Statement grouping in TDL

In order to be practical, tagging pieces of texts in NL should be as easy, quick and natural as possible. It should never get in the way, and devour our attention. Therefore, in addition to providing acronyms for each keyword, TDL also comes in a few flavors. Currently there are four *dialects* of TDL that we have developed and been experimenting with:

Mini TDL (mTDL) contains the smallest set of statements ('Action' and 'Branch') that are necessary and sufficient for cause-effect type of descriptions. It is very interesting to observe that with only two statements one can actually describe the behavior of many systems and generate software requirements (using the tool suite) that conform to many standards and recommendations such as DO-178B. It should be noted that *mTDL* is not a complete language since there are no statements to effectively describe iterations (loops). However, it is still possible to hack the language by describing the loops in a natural language and putting them within the text portion of 'Action' statements.

TDL is the regular language as defined in Appendix A. On top of *mTDL*, it also has statements that signal the end of computation within a unit, unconditional jump statements (to make iterations possible) and statements that are explicitly aware of invocation of other units by the described unit. Note that you can describe unit invocation in *mTDL* inside the text of 'Action' statements, but the special invocation-aware statements in regular *TDL* also make it possible for some non-trivial verifications about interactions between different units.

Extended TDL (xTDL) primarily adds statements that could make it easier to describe loops and repetitions inside the processes.

C-Like TDL (cTDL) slightly renames, modifies and extends *xTDL* statements to make them have same or similar names and syntax to C/C++ statements. For example, in addition to 'Branch' statement, *cTDL* also has 'If' and 'Switch' statements to make it easier to prepare descriptions of C/C++ programs. Note that with *xTDL* and *cTDL*, there is no additional power injected into the regular language (*TDL*), rather, only some *syntactic sugar* is added.

This document describes features of TDL in general, and *TDL* refers to all TDL dialects (not just the regular TDL dialect) unless otherwise specified.

TDL descriptions could also be embedded within a program code as comments, thereby allowing the code and its TDL description to be prepared, kept and updated (preferably simultaneously) in the same file. In fact, if a little care is taken to position the TDL statements carefully within the program code, the tool suite that we will describe can instrument the code for structural coverage analysis. To give an example, in Figure 8, we have a function in C programming language that computes the quotient of its parameters. The function also indicates whether the operation was valid or not. In Figure 9, we have a TDL description of the function. Note that the last TDL statement is an 'Action' which mentions about returning a value from the function only in the text part. We could instead use a 'Return' statement and it would be more appropriate, but this actually shows how flexible TDL is, and how powerful *mTDL* could be. And in Figure 10, we see how both the function and its TDL description could properly reside in the same file. By proper, we mean carefully positioned to make instrumentation possible. If instrumentation is not desired, then TDL statements could be positioned anywhere in the file that the programmer and/or the documenter wants.

```
float divide ( float numerator,
               float denominator ) {
  float result = 0;

  if ( denominator == 0.0 )
  {
    printf( "Division by 0 error\n" );
  }
  else
  {
    printf( "Valid operation\n" );
    result = numerator / denominator;
  }
  return result;
}
```

Figure 8. Sample function 'divide'

```
divide ( numerator  denominator )
{
  Action [ Assign zero to local variable, result ]

  Branch
  {  Condition [ Denominator is equal to 0.0 ]
     Action    [ Print error message ] }
  {  Condition [ Denominator is not equal to 0.0 ]
     Action    [ Print valid operation message ]
     Action    [ Store answer in result ] }

  Action [ Return value stored in result ]
}
```

Figure 9. TDL description of function 'divide'

```
/*  This is a regular comment. Comments that contain TDL statements start with  */
/*  a special string such as '>' to be easily extractable.                       */

float divide ( float numerator, float denominator ) {
/*> divide ( numerator denominator ) {                                           */

    /*> Action [ Assign zero to local variable, result ]                         */
    float result = 0;

    /*> Branch                                                                   */
    if ( denominator == 0.0 )
    {
      /*> { Condition [ Denominator is equal to 0.0 ]                            */
      /*>   Action    [ Print error message ] }                                  */
      printf( "Division by 0 error\n" );
    }
    else
    {
      /*> { Condition [ Denominator is not equal to 0.0 ]                        */
      /*>   Action    [ Print valid operation message ]                          */
      printf( "Valid operation\n" );
      /*>   Action    [ Store answer in result ] }                               */
      result = numerator / denominator;
    }

    /*> Action [ Return value stored in result ]                                 */
    return result;

/*> }                                                                            */
}
```

Figure 10. TDL embedded in source code as comments (highlighted in gray)

4.2 TDS SOFTWARE TOOLS

Once a piece of description is prepared in TDL, it could be processed by TDS tools to generate various data. In this section, we will briefly describe some of the tools that we have developed and experimented with. Our tools and the type of data that we were able to generate are summarized in Figure 11. Since TDL is very flexible and powerful, it is possible to create various other tools that process the language in many other ways to generate data for many purposes for both forward and reverse engineering applications.

Code generators: In forward engineering applications, we could start with software requirements recorded in TDL, and use TDS code generators to automatically generate code stubs and code templates for us in the programming language of our choice. This has many advantages including speed and automatically setting up any required dependencies between source files.

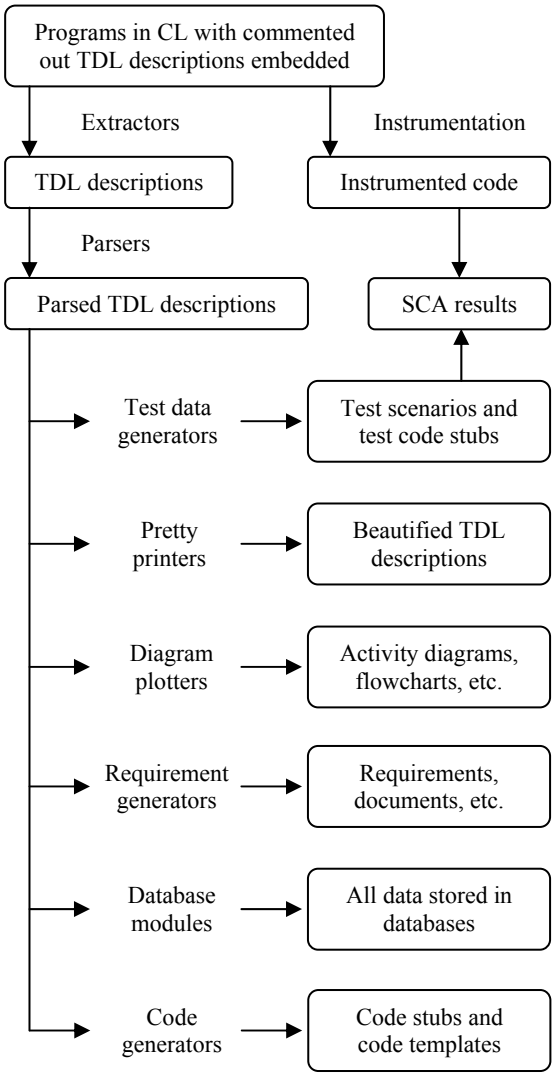

Figure 11. TDS tools and generated data

Extractors: As mentioned, in addition to standalone TDL descriptions, TDL can also be embedded within comments in the same file that contains the source code. Actually, it might be beneficial to prepare TDL descriptions by typing just above the actual source code lines. Then, when the code is modified, it would be easier to keep the TDL description synchronized for the developers. The extractors are tools that extract TDL descriptions embedded in such source files.

Parsers: Parsers are tools that create parse trees from TDL descriptions for further use by other TDS tools. Parsers also perform many non-trivial validations, consistency checks and verifications of integrity such as identifying *dead codes* (the tasks that are never executed due to the way the control flow is set up) within a TDL description.

Pretty printers (Beautifiers): TDL is a free format language just like C, C++ and Java. Beautifiers take the TDL descriptions that are quickly scribbled and format them nicely for later reference. Also, we may use acronyms of statements in our TDL descriptions, and we can instruct the beautifiers to blow them up to full statement names in the output that they produce.

Diagram plotters: Activity diagrams, flowcharts, etc. could easily be generated by TDS tools. For example, the diagram on the left in Figure 12 is automatically generated for the 'divide' function (with additional documentation such as a legend of labels used, as shown in Figure 13), and the diagram on the right is generated for the 'Student' function. Our plotter makes use of Graphviz tool [8], an open source graph visualization software (we generate '*dot*' files and feed them to Graphviz to get the diagrams).

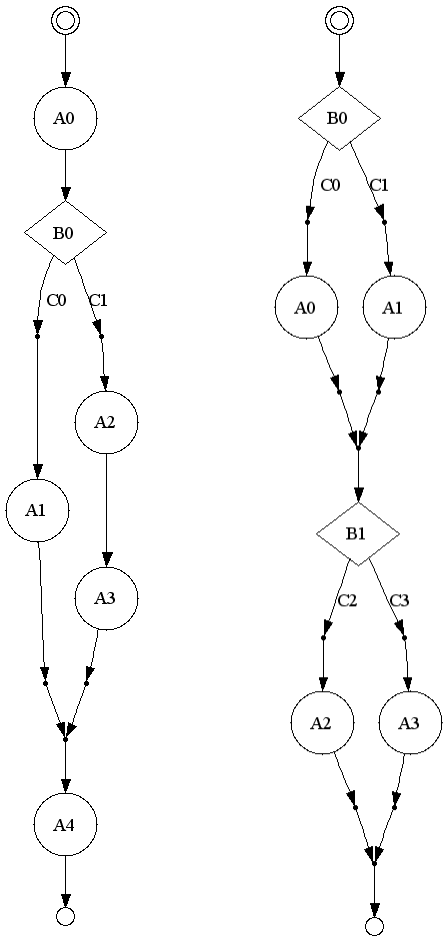

Figure 12. Diagrams of 'divide' and 'Student'

Requirement generators: Descriptions in TDL contain the semantic structure of the described units, therefore just like diagrams, software documentation or software requirements could easily be generated automatically in any desired format such as text, rich text, html, xml, etc. As an example, a TDL description that consists of our samples 'divide' and 'Student' yields the documentation shown in Figure 13 when processed by the respective TDS tools. It is also possible to generate various measures of complexity such as the number of branchings in each unit, and that information is also present in the given example.

Unit name: 'divide'
Parameters: numerator, denominator

Total number of branchings in the unit: 1

List of all conditions (total 2)
C0. Denominator is equal to 0.0
C1. Denominator is not equal to 0.0

List of all actions (total 5)
A0. Assign zero to local variable, result
A1. Print error message
A2. Print valid operation message
A3. Store answer in result
A4. Return value stored in result

List of all possible transitions (total 2)
T0. C0 >> A0, A1, A4
T1. C1 >> A0, A2, A3, A4

Unit name: 'Student'
Parameters: Library, Money

Total number of branchings in the unit: 2

List of all conditions (total 4)
C0. Library is open
C1. Library is closed
C2. There is enough money
C3. There is not enough money

List of all actions (total 4)
A0. Check out math book
A1. Borrow friend's book
A2. Have dinner at restaurant
A3. Cook dinner at home

List of all possible transitions (total 4)
T0. C0, C2 >> A0, A2
T1. C0, C3 >> A0, A3
T2. C1, C2 >> A1, A2
T3. C1, C3 >> A1, A3

Total number of units in translation unit: 2
Total number of transitions in translation unit: 6

Figure 13. Generated documentation

Database modules: All generated data (diagrams, requirements, tests, analysis results, etc.) could be labeled and stored in databases automatically.

Test data generators: TDS tools can generate test scenarios that systematically cover all execution paths within units. Automating such a task removes all possible human errors from this otherwise very difficult task. Given a TDL description, it is easy to generate textual scenarios of the form: '*In order to test transition T0, make sure that the condition C0 holds. Then invoke the unit and observe that actions A0, A1 and A4 are taken in the specified order*'. Furthermore, if proper software code pieces (that set a condition to true or false, and that check if an action is taken or not) are prepared and associated (by the help of TDS tools) with TDL statements, then TDS tools could also easily generate test code stubs for all test cases in the test scenario. There is another method that describes how to attach *hooks* [9] to functions, and that method could also be used in conjunction with TDS tools to vastly decrease the total testing time of big software systems consisting of hundreds of functions.

Instrumentation and analysis modules: During the design of TDL, in addition to many considerations, great attention was also paid to have an instrumentable language. If TDL descriptions are carefully embedded in code as comments, the file can then be augmented automatically with the programming language of our choice, and many tasks such as requirement-to-code traceability and some SCA tasks such as decision coverage analysis could be automated. Furthermore, if the source code is described densely enough in TDL, most other SCA tasks such as statement coverage analysis might also be automated.

5. ASSESSMENT OF TDS

Equipped with TDS, let us go over the same forward and reverse engineering practices mentioned in Section 3 and observe how TDS could change the way that we interact with computers for the better.

During our meetings with the customer, we could record the requirements in TDL rather than in some NL (by just throwing in some TDL statements) and make the requirements semantically meaningful to the computers. Then we could have TDS tools perform some validations (to catch logical errors) and generate diagrams. Quickly going over the diagrams with the customer clears up many potential misunderstandings and problems even at this early stage (requirement specification phase of the software life cycle).

TDS tools then generate code templates for us in the programming language of our choice. Having the template creation done by computers saves us an incredible amount of time and it also has many advantages such as eliminating the possibility of overlooking even the tiniest detail and automatically setting up any necessary dependencies between program files for us with 100% accuracy. The code templates could also contain the very same TDL requirements replicated and commented out for us, and we then just fill in the necessary code to be done with the coding phase.

Then we have TDS tools create test scenarios from the

requirements. If we also feed in code pieces associated with conditions, actions, etc. within the requirements, then TDS tools will also generate a big portion, if not all, of the test code for us. We then just fill in any remaining parts, and our tests are now also ready to be run against the developed software. And we will be sure that not a single test case is omitted and our software will be tested thoroughly. We can even have TDS tools automatically instrument the code and perform most or all of SCA, depending on the level of detail of the TDL requirements.

To our surprise, traceability between the requirements and pieces of software is already set up. There is no need to perform anything else, because this process is inherent when we utilize TDS. With the click of a mouse button, it is possible to generate documents that report which requirement was implemented by which code fragment.

The documentation is also available to us from the very beginning when we use TDL. We just instruct TDS tools to turn and format the requirements into user manuals and/or other necessary documentation. Also, if any information that was not present in the set of requirements is needed, they could be described in TDL and turned into any form of documentation again by the TDS tools easily.

As for the reverse engineering tasks, TDS can help us quickly capture the overall picture that shows how a huge system works. We just need to locally analyze pieces, prepare TDL descriptions for them, and then feed them to TDS. The tools can generate diagrams and reports that help us perceive the interactions between small pieces and their internal workings.

Capturing layers when documenting a piece of software is also taken care of by the *statement grouping* feature of TDL. We can optionally assign a user defined group to some or all TDL statements as shown in Figure 7. TDS tools can then treat statements in different groups in special ways. For example, we can have TDS tools skip 'Safety' related statements in Figure 7 when generating some user manuals, or we can have statements of a certain group be plotted in a different color when generating diagrams, etc.

6. CONCLUSION

Even with state-of-the-art technology, we still do not have the luxury of communicating with computers in our own languages. Contrary to the mysteriously amazing job that the human brain does hundreds of times each and every day (that makes us think how easy verbal communication is), natural languages actually contain ambiguities and are currently not suitable to be processed by computers. Nevertheless, there are and will always be huge scientific efforts to make this dream come true.

The act of *developing software* is actually *talking to computers in their own languages*. If computers were able to speak our language, then there would be no need to develop any software and we would not need any programming languages at all.

In this document, we presented a very promising solution that enables computers to *understand* texts in natural languages. The main idea is to assign semantic meaning *piecewise* by tagging chunks with special marking statements that enable computers to identify how each chunk functions logically within the whole text. Our study shows that by scattering around only a handful of statements and some parentheses, it is possible to let computers discover the structure of natural language texts and construct inferences like 'This text describes behavior of (one or more) units and their interactions. There are certain conditions and some related actions'. And the semantic structure reveals to the computers which actions are taken when certain conditions hold, thereby letting them figure out the flow of ideas within the text.

When we use TDS to speak with computers in NL, all aspects of software development could be eliminated (automated) by the help of a proper set of tools. The quantitative performance that TDS provides could be summarized as having months of work done in only a couple of days without any human errors.

REFERENCES

[1] More information about NL is available at http://en.wikipedia.org/wiki/Natural_language

[2] Centre for Language Technology, Macquarie University, Sydney, Australia. More information about the state of the research is available at http://www.ics.mq.edu.au/~rolfs/controlled-natural-languages/

[3] More information about the Attempto Project is available at http://attempto.ifi.unizh.ch/site/

[4] More information about PENG is available at http://www.ics.mq.edu.au/~rolfs/peng/

[5] More information about CLCE is available at http://www.jfsowa.com/clce/specs.htm

[6] More information about KANT is available at http://www.lti.cs.cmu.edu/Research/Kant/

[7] RTCA DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", December 1992, http://www.rtca.org/

[8] More information about Graphviz is available at http://www.graphviz.org/

[9] Temizer S., "Yazılım Yapılandırma Teknikleri: Temizer Sistemi", İkinci Ulusal Yazılım Mühendisliği Sempozyumu, UYMS'05, pp. 305-313, Ankara, September 22-24, 2005.

APPENDIX A - *Backus-Naur Form (BNF) Definition of TDL*

Below is the syntax and grammar of TDL in BNF notation. Entities in regular font are non-terminal symbols. Italic font denotes optional entities. Items in bold font are terminals. Most terminals are enclosed in single quotes and need no further explanation. There are two non-terminal symbols that we need to define: **IDENTIFIER** is equivalent to a C Programming Language identifier and **TEXT** is a piece of text in a natural language of our choice enclosed in square brackets. The special non-terminal 'TranslationUnit' is the start symbol.

TranslationUnit := Unit
| TranslationUnit Unit

Parameter := **IDENTIFIER**

ParameterList := Parameter
| ParameterList Parameter

Note := **'Note'** **TEXT**

NoteList := Note
| NoteList Note

Label := **IDENTIFIER** **':'**

Group := **'<'** **IDENTIFIER** **'>'**

Statement := Action
| Branch
| Goto
| Invoke
| Return
| Exit

LabeledStatement := Label Statement
| Label LabeledStatement

StatementList := Statement
| LabeledStatement
| StatementList Statement
| StatementList LabeledStatement

Action := **'Action'** *Group* **TEXT**

Condition := **'Condition'** **TEXT**

NegativeCondition := **'NegativeCondition'** **TEXT**

ConditionalStatement := **'{'** Condition *NegativeCondition* *StatementList* **'}'**

ConditionalStatementList := ConditionalStatement
| ConditionalStatementList ConditionalStatement

Branch := **'Branch'** *Group* ConditionalStatementList

Goto := **'Goto'** *Group* **IDENTIFIER**

Argument := **TEXT**

ArgumentList := Argument
| ArgumentList Argument

Invoke := **'Invoke'** *Group* **IDENTIFIER** **'('** *ArgumentList* **')'**

Return := **'Return'** *Group* ***TEXT***

Exit := **'Exit'** *Group*

Unit := **IDENTIFIER** **'('** *ParameterList* **')'** **'{'** *NoteList* *StatementList* **'}'**

VITAE

Selim TEMİZER

In 1999, Selim Temizer received his B.S. degree from the Department of Computer Engineering, Middle East Technical University (METU). In 2001, he received his M.S. degree in Electrical Engineering and Computer Science at Massachusetts Institute of Technology (MIT) and continued his education with his Ph.D. at the same institute in The Artificial Intelligence Laboratory. Among his research interests are artificial intelligence, computer vision, robotics, robotic navigation, map-making techniques for mobile robots and simulation systems. He has taken a leave of absence from his Ph.D. studies since February 2004.

He worked at Meteksan Sistem ve Bilgisayar Teknolojileri A.Ş. and Aydın Yazılım ve Elektronik Sanayi A.Ş., leading teams on various high profile projects such as the design of an OpenGL driver for the Joint Strike Fighter (JSF) F-35 Panoramic Cockpit Displays. He offered a C Programming Language course as an instructor at the Department of Computer Engineering, METU for two semesters.

He recently completed his military service in April 2007, where he had been serving as a reserve officer in The Scientific Decision Support Center (Bilimsel Karar Destek Merkezi, BİLKARDEM, Genelkurmay Başkanlığı).