TUG: A Multi-Paradigm Language for Software Development

Abstract
The quality of software depends mainly on the effectiveness of the software development paradigm. Existing software development paradigms are not flexible and practical enough for developing a system that needs the mix of existing paradigms used in the production of that system. This paper presents a formal specification language, called TUG, to support a system to be developed through an integration of conventional software development, operational specification, rapid prototyping via software transformations, software reuse, and analysis of specifications and programs via testing and proofs. The software development paradigm with the aid of the TUG specification language improves the software development process by delivering an operational prototype early to the user and increasing the role of automation. The pursuit of these improvements contributes to the production of reliable and reusable programs.

1. Introduction
The conventional software development paradigm has been criticized as not appropriate for evolutionary software. Errors are frequently not detected until the testing phase. Even worse, several errors may still remain in the system after the system is delivered. Alternative software development paradigms such as rapid prototyping, operational specification, software transformations, and software reuse respond to the issues of the conventional software development paradigm. However, these existing paradigms are not flexible and practical enough for developing a system that needs the mix of existing paradigms used in the production of that system. A flexible and practical software development paradigm is a software development model in which the model can not only allow for the mix of existing paradigms to a given system but also can be customized to one of existing paradigms [Agr86]. For example, the application of the model to a system that has a poorly understood user interface and decision support functions provides an example of a mix of rapid prototyping and evolutionary development. However, if the system has fully understood user decision support functions, the considerations of a poorly understood user interface derive the model into an equivalence to the rapid prototyping paradigm. This paper presents a formal specification language, called TUG (Tree Unified Grammar), to support a system to be developed through an integration of conventional software development, operational specification, rapid prototyping via software transformations, software reuse, and analysis of specifications and programs via testing and proofs. The combinations of existing paradigms provides the ability to be applied to a system that needs to be developed in a mix of existing paradigms and also achieves the strengths of each paradigm on the system.

2. Importance of TUG in Software Development
The development of a language plays a key role in supporting this integration of existing software development paradigms. Informal languages are not appropriate to support this task due to noise, silence, overspecification, contradiction, ambiguity, forward reference, and wishful thinking [Mey85]. Formal languages make this integration of existing software development paradigms possible. Unfortunately, most of existing formal
specification languages only support partial integration, none of them supports the whole integration as TUG does. Programming languages are formal languages. However, they are unsuitable for writing user requirements because the premature decisions need to be made in the front-end of the software development paradigm.

The formality of TUG supports the operational specification paradigm in which TUG specifications can be executed by a language processor. Although the TUG specification language supports the operational specification paradigm, the language still lacks notation for specifying non-functional properties of software systems such as user interface and performance. To alleviate this problem, TUG allows a prototype to be automatically built and then code for non-functional properties of the system can be added manually to the prototype for rapid prototyping purpose. The language supports the construction of a prototype via software transformations [Chi04b]. The underlying mathematical principles of DCGs (Definite Clause Grammars) provide a vehicle for developing a set of transformation rules used to generate a prototype in Prolog from a TUG specification. The resulting prototype can be exercised for user feedback under the Prolog environment. Rapid prototyping via software transformations also provides support for program modifications. The process bypasses the difficulty of having to modify code that has been poorly structured since changes are made to the specifications. TUG supports prototype evolution by avoiding complete retransformation of the prototype whenever a change is made in the specification. To avoid complete retransformation, a CRS (Change Request Script) is written and used to update the prototype only in response to minor changes to the specification involving nodes to be modified, extended, relaxed or refined. If a major change is needed, the specification may need to be rewritten and a new prototype may be derived from the beginning. A major change may require the structure of the specification to be modified.

Reuse is believed to be a key to improving software development productivity and quality [Big87]. The reuse of software allows developers to spend less time in creating new software. To facilitate software reuse, the TUG specification language allows a developer to write strongly typed and weakly typed specifications [Chi03b]. In a strongly-typed specification, each terminal node is declared to at most one type. In contrast to a strongly typed specification, a weakly typed specification allows for a terminal node belonging to more than one type. The application of weak typing, such as overloading, provides a way of constructing reusable specifications in the language. In addition, often there is a specification for reuse, but it is not in exactly the right form. TUG allows the applications of some modifications to the specifications. TUG achieves such goals by providing a renaming operator to replace an old name by new name. The language also provides the extending operator to expand some portions of the base specifications.

A specification in TUG can be analyzed for detecting errors due to the formality of the language [Chi02]. Errors such as type inconsistency, redundancy, and conflict can be detected by analyzing the specification via inspection or review. In addition, TUG allows the detection of errors either by directly executing a specification in the language with test data or by proving a specification against user claims. Executing a specification with
test data produces results only for a given test of input data satisfying the specification’s input. A set of test cases can be generated from a TUG specification [Chi03a]. Unfortunately, testing is not reliable because one cannot have test cases having all the input values. TUG was designed to provide another way of proving a specification for correctness against the corresponding user claims by first translating the specification into a set of Horn clauses. The set of Horn clauses with a user claim are then proved for correctness via theorem proving. Proofs on a TUG specification demonstrate the consistency of the specification, and also show that the specification satisfies the user claims. However, proofs are of little help in some areas, such as user interface suitability [Boe84, Hal90]. Therefore, TUG emphasizes testing and proofs. A combination of testing and proofs enhances software quality and increases software reliability.

TUG supports the conventional software development paradigm. A specification is written in TUG. The design process operates in two phases, each of which transforms the input of one representation into another. First, the TUG specification is mapped into a structured design in which each structure diagram corresponds to a specification module. Second, the structured design is then mapped into a set of structured programs in which each program module corresponds to a structured diagram. The mapping is performed from specification, design to coding according to the structuring notation in the TUG specification. A structure diagram or structured program is composed of a set of blocks. A block may denote a structuring notation, a single statement, a set of statements, or a complete program. A set of mapping guidelines is provided to guide developers in developing structure diagrams and programs. The structured diagrams and programs can be proved for correctness using the axioms and the loop theory [Chi01, Chi97]. The proofs can be complemented with the testing method in which test cases are automatically generated from the specifications [Chi03a].

3. Overview of the TUG Language

TUG is based on a data structuring method and the concept of recursion. The language describes the expected input data, the output produced, and the functional relationships that exist between them. If the input data are considered as a string of characters, then the rules for creating the input data can be defined. Therefore, we can parse the input data based on the rules described in the specification. The output of the syntactic analysis is a parse tree where pattern matching is performed to generate the required output. In addition, the matched parts of the input data can be combined into new internal input data and then fed back into the analysis so that further output can be generated.

The TUG specification language consists of three parts: a name part where the title with input/output parameters are given, an analysis part where the input data is defined, and an anatomy part where the output data is generated. The name part contains a module or scheme title with input/output parameters. The input/output parameters are enclosed in parentheses. The analysis part contains the rules for analyzing the input data and then produces a parse tree for the input data. Each rule expresses a possible form for a nonterminal, as a sequence of terminals with optional constraints on the terminals and nonterminals. Nonterminal nodes in uppercase indicate constituents. A terminal node in lowercase indicates a token that must occur in the input data. A terminal node can be a
literal which is any string enclosed in a pair of quotes. A constraint wrapped in braces places the conditions such as type checking on a terminal node. The anatomy part contains the rules describing the output to be generated. The tree from the analysis part is fed into the anatomy part and the output data is then synthesized through pattern matching on the tree. The new input data can also be generated and sent back to the analysis part for further output generation; otherwise, the execution terminates. The language syntax and semantics are defined in [Chi04a].

4. A Case Study: the Library Problem
In this section, we are illustrating the use of TUG to support the writing of user requirements and rapid prototyping through the execution of TUG specifications. TUG is different from the programming languages for specifying the system behavior. TUG offers the formality of a programming language, but without the perspective aspect whereby every detail of data structures and algorithms must be provided. Abstraction in TUG describes the system properties from unnecessary details such as implementation decisions or overspecification caused by programming languages.

Rapid prototyping provides a practical approach to building evolutionary software and managing the changing user requirements in a cost-effective way [Luq92]. A prototype is built and exercised to explore the user requirements at the requirements and specification phase of the software development life cycle. There are two distinct types of software prototypes: throwaway and evolutionary prototypes. A throwaway prototype needs to be built quickly and cheaply. An evolutionary prototype is built in a quality manner. A throwaway prototype is discarded after the desired user requirements are obtained. However, an evolutionary prototype is successively elaborated toward the final product. This section is to explore how user requirements and specifications can be formulated and executed to support rapid prototyping through operational specification.

4.1 The Problem Description in Natural Language
This problem was originally developed by Kemmerer and was restated at the Fourth International Workshop on Software Specification and Design on Page ix. “Consider a small library database with the following transactions:
1 – Checkout a copy of a book/Return a copy of a book;
2 – Add a copy of a book to/Remove a copy of a book from the library;
3 – Get the list of books by a particular author in a particular subject area;
4 – Find out the list of books currently checked out by a particular borrower
5 – Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4 and 5 are restricted to staff users, except the ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:
1 – All copies in the library must be available for checkout or to be checked out.
2 – No copy of the book may be both available and checked out at the same time.
3 – A borrower may not have more than a predefined number of books checked out at one time.”
4.2 The Questions Inherited in the Library Problem

The formality of TUG allows to answer questions not answered by reference to the library problem in natural language.

We begin with the questions about initialization of the library database, then a question about error/exception handling, and finally a question about missing operations.

- Initialization: The library starts out with no books, but a set of staff user id’s.
- Error Handling: The informal requirements do not state what should happen if an error is encountered. Some of the errors we address include:
  - Error for checking out: The input data are incorrect, so that the book record cannot be located by the system. An error message is displayed to the staff user.
  - Error for returning: The input data are incorrect, so that the book record cannot be located by the system. An error message is displayed to the staff user.
  - Error for adding: The book being added already exists. This error may occur in two situations, namely when the staff user types in the incorrect data and when the book has more than one copy. If the book has several copies, a unique book id should be given to differentiate the copies.
  - Error for removing: The book being removed does not exist or is already checked out. This error may occur in two situations, namely when the staff user types in the incorrect data and when the book is already checked out. A staff user cannot remove the book that is already checked out until the book is returned to the library.

Type errors should also be detected in the language. If an argument, input data, or result of an operator is of the wrong type, the specification contains an inconsistency.

- Missing Operations: Four operations are needed to add a new staff user, remove a staff user, add an ordinary borrower, and remove an ordinary borrower. Since we assume that if a person is not a staff user, then the user is an ordinary borrower. In this specification, two operations are specified to add a new staff user and remove a staff user. The errors for these two operations are:
  - Error for adding a new staff user: The staff user being added already exists.
  - Error for removing a staff user: The staff user being removed does not exist.

4.3 A Formal Specification in TUG for the Operational Specification Paradigm

This section is to demonstrate how TUG supports the Operational Specification Paradigm in which a TUG specification can be executed to explore user requirements at the requirements and specification phase.

The library system is designed to be an interactive system. In addition, the system is specified to be a set of specification modules. Operations for a staff user and an ordinary borrower are specified in different modules. This separation helps developers to maintain the system since the operations can be added, modified, or deleted without affecting each other. Users are divided into two categories – staff users and ordinary borrowers. The id’s of staff users are stored in the library database. For each type of user, an appropriate menu is used for operations. Staff users can perform all transactions and ordinary borrowers can perform only Transaction 4. Each operation is specified in one module only.
We now demonstrate some functions of the library. A main module is specified to accept the user identification number. The id is authenticated and the status is classified in the authenticate_id module. Once the user status (a staff user or an ordinary borrower) is obtained, an appropriate menu is displayed for operations. The following module authenticates the id and returns the status of the user.

```plaintext
MODULE authenticate_id(in: LIBRARY_DATABASE; out: STATUS)
  ANALYSIS
    LIBRARY_DATABASE|
      STAFF_USER_FOUND&
      Id
        {integer(id)}
      list1
      staff_user_id
        {integer(staff_user_id),
         equal_to(staff_user_id, id)}
      list2
      ORDINARY_BORROWER_FOUND&
      Id
        {integer(id)}
      list3
  END OF ANALYSIS;
  ANATOMY
    library_database|
      staff_user_found&
      STATUS = 'staff_user'
    ordinary_borrower_found&
      STATUS = 'ordinary_borrower'
  END OF ANATOMY;
END OF MODULE authenticate_id.
```

The input id and the library database is stored in LIBRARY_DATABASE and the output status will be returned in STATUS. The id is checked against LIBRARY_DATABASE whether the user is a staff user. If not, the user is an ordinary borrower. In the anatomy part of the module, if a staff user is found, the module returns ‘staff_user’; otherwise, returns ‘ordinary_borrower’ to the caller. The following module specifies how a staff user can check out a book for an ordinary borrower.

For a staff user to check out a copy of a book for an ordinary borrower, the copy should not be checked out yet. An ordinary borrower cannot exceed the book limit. The book limit is 10. An ordinary borrower should not have the same copy of a book. In the check_out_a_copy_of_a_book module, a copy of a book is available for checkout if borrower is set to 0. A copy is updated using book_number and borrower_id. The borrower node is set to borrower_id after the copy is checked out. If the borrower is not equal to 0, the copy is already checked out. The invalid_check_out_input path specifies what happens when a staff user types in the wrong input data. An error message is displayed to the staff user. The staff user needs to retype the input data.

```plaintext
MODULE check_out_a_copy_of_a_book(in: LIBRARY_DATABASE; out: NEW_LIBRARY_DATABASE)
  ANALYSIS
    LIBRARY_DATABASE|
      VALID_CHECK_OUT_INPUT&
      book_number
        {integer(book_number),
         equal_to(size(book_number), 5)}
      borrower
        {integer(borrower)}
      book_number
        {integer(book_number),
         equal_to(size(book_number), 5)}
    invalid_check_out_input
      borrower
        {integer(borrower)}
      book_number
        {integer(book_number),
         equal_to(size(book_number), 5)}
  END OF ANALYSIS;
END OF MODULE check_out_a_copy_of_a_book.
```
In the `check_out_a_copy_of_a_book` module, the input values contain `book_number` to be checked out, borrower’s id and the library database. The output contains a updated library database that contains the information indicating the book has been checked out. The `size` operator returns the number of digits in an integer. The `#` symbol returns the occurrences of the `book_id` node being evaluated. For instance, if a user checks out five books, the `book_id` in the `VALID_CHECK_OUT_INPUT` path will be evaluated five times during the course of the analysis. In the library problem, a user is not allowed to check out more than 10 books at one time.

To support the execution of TUG specifications, an interpreter was developed in C that runs under the UNIX environment. Besides the interpreter, a scanner was developed to detect any illegal token in the specification. A parser was developed to detect any illegal statement that does not satisfy the syntax of the language.

5. The Library Problem Revisited

Software reuse should not be limited to code alone. The design rationale of how software should be developed is too often ignored. Software reuse should be engineered as early as
possible in the software development life cycle. The earlier reusability considerations are put into software development, the larger the payoff can be obtained from reuse. In this section, we are going to revisit the library problem and take software reuse into account at the requirements and specification phase.

Useful common features of generalized systems can be abstracted into reusable schemes (templates). These schemes may be reused across all similar systems. In this section, the library problem can be viewed as a particular instance of a generalized inventory control problem, where an inventory consists of library books. Schemes of the inventory control problem may be instantiated to model the library problem. Consider a small inventory system, with the following transactions:

Transaction 1: dispense an item.
Transaction 2: return an item.
Transaction 3: get the list of items by a particular customer or in a particular item name.

All transactions are restricted to staff users. The system must also satisfy the following constraints:

Constraint 1: All items in the inventory must be available for withdraw or be dispensed.
Constraint 2: No item may be both available and dispensed at the same time.

The system must handle the following errors:

Error for dispensing: The input data are incorrect, so that the item record cannot be located by the system. An error message should be displayed to the staff user.

Error for returning: The input data are incorrect, so that the item record cannot be located by the system. An error message should be displayed to the staff user.

The inventory contains a collection of items. An item is modeled as having an item id, item name, item description1, item description2, customer, and last customer.

5.1 A Formal Specification in TUG for Software Reuse

TUG provides parametric building constructs such as schemes (generic and parameterized modules) to encourage the design of general-purpose, reusable specification templates. A scheme is a template or abstraction of a specification that some portions of the specification remain undefined. These abstract or undefined portions can be either specification sections, types, constants, or nodes. A scheme separates the body and the parameter. With careful instantiation, a scheme can be specialized and refined on different specifications for different functions. In this section, we are going to demonstrate how a generalized inventory scheme can be reused to model a library system and how a check_out_a_copy_of_a_book specification of the library problem can be built through a reusable template of the inventory system.

The following parametric scheme dispense_an_item_scheme captures the most common features of dispensing an item from the inventory described above. One situation is considered to be false, namely, when the item cannot be located by the system due to the incorrect input data.
SCHEME dispense_an_item_scheme(ITEM_TYPE, ITEM_SIZE, CUSTOMER_TYPE, NULL_CUSTOMER_ID)

ANALYSIS

INVENTORY_DATABASE
VALID_DISPENSE_INPUT
item_number
  {ITEM_TYPE(item_number),
    equal_to(size(item_number), ITEM_SIZE)}
  \ \ customer_number
  {CUSTOMER_TYPE(customer_number)}
  \ \ list1
item_id
  {ITEM_TYPE(item_id),
    equal_to(size(item_id), ITEM_SIZE),
    equal_to(item_id, item_number)}
  \ \ list2
customer_id
  {CUSTOMER_TYPE(customer_id),
    equal_to(customer_id, NULL_CUSTOMER_ID)}
  \ \ list3
INVALID_DISPENSE_INPUT
item_number
  {ITEM_TYPE(item_number),
    equal_to(size(item_number), ITEM_SIZE)}
  \ \ customer_number
  {CUSTOMER_TYPE(customer_number)}
  \ \ database

END OF ANALYSIS;

ANATOMY

inventory_database
valid_dispense_input
output nl
output 'Dispense Successful!' output nl
T_L = item_id :: ' ' :: list2
T_L1 = customer_id :: ' ' :: list3
NEW_INVENTORY_DATABASE = list1 <> T_L <> T_L1
invalid_dispense_input
output nl
output 'Wrong Input Data!' output nl
NEW_INVENTORY_DATABASE = database

END OF ANATOMY;
END OF SCHEME dispense_an_item_scheme.

Refining a higher level scheme into a lower scheme can also be made reusable. For example, some classes of inventory control systems do not permit the same item to be dispensed to the same customer. These high level design differences can be used to distinguish among variations of inventory control systems as shown in Figure 1.
The `dispense_an_item_without_duplicate_scheme` scheme is an instance of `dispense_an_item_scheme`. The `extend` construct introduces error operations to `dispense_an_item_without_duplicate_scheme` without changing the meaning of `dispense_an_item_scheme`. The properties of the original scheme are preserved. The extension in the `dispense_an_item_without_duplicate_scheme` scheme is considered as a meaning preservance extension. This extension is captured by `same_item_found_in_front` and `same_item_found_in_back`, respectively. The inventory differentiates same items by identifying the item identification numbers. A same item is defined as having the same item name and item descriptions. The customer checks out an item by giving the item id and the customer id. In order to find the item names and item descriptions, the item being checked out must be located first. If the item has been located, the item name and item descriptions with the customer id are used to check whether there is another same item already been dispensed to the customer. If yes, the customer is trying to check out the same item again. An error message is displayed to the customer.

In the `dispense_an_item_without_duplicate_scheme` scheme, `item_number` and `customer_number` are provided by the customer to check out an item. Each item has a unique `item_number`. Two items with the same item names and item descriptions are considered as same items. To check whether a same item has been dispensed to the same customer, the requested item is first located by `item_number`. `Item_name` and `item_description1`, and `item_description2` are obtained to check whether another same item has been dispensed to the same customer or not. The operation of searching another same item in the inventory is split into two cases, namely, when the same item is located before the requested item and when the same item is located after the requested item. These two situations are captured by `SAME_ITEM_FOUND_IN_FRONT` and `SAME_ITEM_FOUND_IN_BACK`, respectively.
SCHEME dipense_an_item_without_duplicate_scheme(ITEM_TYPE, ITEM_SIZE, CUSTOMER_TYPE, NULL_CUSTOMER_ID, ITEM_NAME_TYPE, NULL_ITEM_NAME, ITEM_NAME_SIZE, ITEM_DESCRIPTION_TYPE, NULL_ITEM_DESCRIPTION, ITEM_DESCRIPTION_SIZE)

EXTENSION

Extend INVENTORY_DATABASE of dispense_an_item_scheme with
SAME_ITEM_DISPENSED| SAME_ITEM_FOUND_IN_FRONT&

item_number

{ITEM_TYPE(item_number),
  equal_to(size(item_number), ITEM_SIZE)}

customer_number

{CUSTOMER_TYPE(customer_number)}

list1

same_item_name

{ITEM_NAME_TYPE(same_item_name),
  greater_than(size(same_item_name), NULL_ITEM_NAME),
  less_than_or_equal_to(size(same_item_name), ITEM_NAME_SIZE)}

same_item_description1

{ITEM_DESCRIPTION_TYPE(same_item_description1),
  greater_than(size(same_item_description1), NULL_ITEM_DESCRIPTION),
  less_than_or_equal_to(size(same_item_description1), ITEM_DESCRIPTION_SIZE)}

same_item_description2

{ITEM_DESCRIPTION_TYPE(same_item_description2),
  greater_than(size(same_item_description2), NULL_ITEM_DESCRIPTION),
  less_than_or_equal_to(size(same_item_description2), ITEM_DESCRIPTION_SIZE)}

same_customer_id

{CUSTOMER_TYPE(same_customer_id),
  equal_to(same_customer_id, customer_number)}

list2

item_id

{ITEM_TYPE(item_id),
  equal_to(size(item_id), ITEM_SIZE),
  equal_to(item_id, item_number)}

item_name

{ITEM_NAME_TYPE(item_name),
  greater_than(size(item_name), NULL_ITEM_NAME),
  less_than_or_equal_to(size(item_name), ITEM_NAME_SIZE),
  equal_to(item_name, same_item_name)}

item_description1

{ITEM_DESCRIPTION_TYPE(item_description1),
  greater_than(size(item_description1), NULL_ITEM_DESCRIPTION),
  less_than_or_equal_to(size(item_description1), ITEM_DESCRIPTION_SIZE),
  Equal_to(item_description1, same_item_description1)}

item_description2

{ITEM_DESCRIPTION_TYPE(item_description2),
  greater_than(size(item_description2), NULL_ITEM_DESCRIPTION),
  less_than_or_equal_to(size(item_description2), ITEM_DESCRIPTION_SIZE),
  Equal_to(item_description2, same_item_description2)}

customer_id
With the `dispense_an_item_without_duplicate_scheme`, we can instantiate the `check_out_a_copy_of_a_book` specification of the library problem. The `check_out_a_copy_of_a_book` specification of the library problem is an instance of `dispense_an_item_without_duplicate_scheme` as shown in Figure 2.

```
\textit{NEW_INVENTORY_DATABASE} = \textit{list1} \bowtie \textit{T_L} \bowtie \textit{T_L1} \bowtie \textit{list3}
```

Figure 2: An Instance of Dispensing an Item without Duplicate

The following is considered a two-level nested generic specification that it must be instantiated twice in order to be used. In the first instantiation, `dispense_an_item_scheme` is overloaded. In the second instantiation, `dispense_an_item_without_duplicate_scheme` is refined into the `check_out_a_copy_of_a_book` specification of the library problem with appropriate actual values. The `use` construct is a simple substitution used in schemes or specifications.
MODULE check_out_a_copy_of_a_book(in: LIBRARY_DATABASE; out: NEW_LIBRARY_DATABASE)
INSTANTIATION
  use library_database for inventory_database
  use new_library_database for new_inventory_database
  use same_copy_found_in_back for same_item_found_in_back
  use valid_check_out_input for valid_dispense_input
  use invalid_check_out_input for invalid_dispense_input
  instantiate check_out_a_copy_of_a_book with
    dispense_an_item_without_duplicate_scheme(integer, 5,
      integer, 0,
      word, 0, 20,
      word, 0, 30)
  END OF INSTANTIATION;
END OF MODULE check_out_a_copy_of_a_book.

6. Summary
The development of a flexible software development process with a formal specification language, called TUG, has been described in this paper. The formal method supports a mix of software development activities, such as conventional software development, operational specification, rapid prototyping via software transformations, software reuse, and analysis of specifications, such as testing and proofs. Unfortunately, due to the size of the paper, we cannot exhaustively demonstrate all the paradigms that TUG supports. In this paper, we only illustrated the use of TUG for the operational specification and software reuse paradigms. Readers may refer to [Chi01, Chi02, Chi03a, Chi03b, Chi041, Chi04b, Chi97] for more features of TUG.

TUG has weaknesses. The language based on DCGs provides a logic-based approach to formally specifying and verifying the behavior of sequential software systems. Unlike other existing logic-based approaches, such as those in [Buh91, Mat01, Mos00], TUG has no notations for concurrency, distribution, parallelism, timing, or performance. This domain-specific approach sacrifices suitability to general purposes by focusing on data processing. In addition, Prolog has efficiency limitations, mainly due to backtracking. Thus, prototypes in Prolog are unsuitable for some specific domains. For instance, TUG should not be used for rapid prototyping in Prolog when system performance is a concern.

DCGs have been used in many applications, such as natural language processing [Abr89], language analysis & compilers [Bow93, War80]. We developed the TUG specification language based on the theory of DCGs. However, a specification in DCGs seems difficult to read and understand. Since the main purpose of a specification is to aid the understanding of user requirements, it is better if a specification can be read and understood easily. Therefore, another form of the representation must be acquired. We adopted regular expression notations to syntactically structure TUG specifications in order to improve readability. The theory of the specification language is also hidden from users to improve understandability. We applied TUG to an industrial reverse engineering project at Viasoft in Phoenix. The project aimed to provide reverse engineering tools for analyzing legacy systems in Assembly, PL/1, and COBOL. Program analysis plays a key role in reverse engineering. Since DCGs are more expressive than context-free grammars for describing languages, we used TUG to prototype a subset of the grammars of COBOL to build tree structures in a form of array representations. The purpose of this rapid prototyping was to help us understand and create common data structures needed for analyzing COBOL, Assembly, and PL/1 legacy programs.
References


