LINGUSQL : A VERIFICATION AND TRANSFORMATION TOOL FOR DATABASE APPLICATION

Rikky Wenang Purbojati[1], I.S.W.B Prasetya[2], Heru Suhartanto[1], Sirajuddin Maizir[1], Budiono Wibowo[1], Ade Azurat[1]

[1]Faculty of Computer Science, University of Indonesia
Kampus Baru UI Depok, Depok, Jawa Barat, Indonesia.
Email : {wenang, heru, sirajuddin, budiono, ade}@cs.ui.ac.id
[2]Institute of Information and Computing Sciences, Utrecht University
Netherlands.
Email : wishnu@cs.uu.nl

ABSTRACT

Lingu is an experimental abstract language for programming database scripting. This language includes verification and validation as an integral part of its programming. The verification and validation ensures the application is capable to be used in critical operation. The programming of Lingu itself does not produces a working code, instead it produce an abstract code that describes program logic. This abstract language is transformed into a concrete language, such as C or Java, intended to be used in real operation. This paper discuss about the development of LinguSQL. LinguSQL is a tool for verifying and transforming Lingu script. Currently the development of LinguSQL is in prototyping phase. We hope that the prototype will provide a clear guidance in further development.

Keywords : Abstract Language, Software Verification, Theorem Prover, Software Testing, Language Transformation.

1. INTRODUCTION

Numerous attempts to use formal method to verify a program has been made in the last decades. These attempts include technique such as model checking, theorem prover, etc. The implementations of such formal verification method sometimes become too difficult and unpractical. This is most likely because the code and specification tends to grow more complex than what is expected from when the project started. The complex program implies that the verification of such program will be much more complex and difficult to be done.

One of the possible solutions for this problem is to use a domain specific abstract language to program the code of given cases. The use of abs* tract language will greatly simplify the verification process in a sense that the programmer only has to deal with a script which describes the basic flow of program, not the whole detail program itself. The abstract language has to provide simple yet comprehensive method of specifying what the program should or should not do. The specification on abstract language should guarantee that the program will be running as intended by the design. Afterwards the abstract language can be transformed into a concrete language such as Java, C, VB, etc.

The use of abstract language simplifies application development. Abstract language is easier to understand. The simplification comes from the fact that the programmer only has to focus on how the business process works, and not burdened by miscellaneous work that does not concern with the main process. Simplification will also leads to a more straightforward and clearer code. These, in the end will results in a more credible program.

Besides simplification of the problem, abstract language also increases application portability. Some of the work done in developing an application is to port an old or obsolete system to a newer system. The newer system more than often is made from an entirely different platform than the old one. The work often is done from scratch, which is cost consuming and time consuming.

* Partly Funded by RUTI-Menristek 2003 - 2005

Abstract language can eliminate the need to build the system from scratch. Porting the application from one language to another is done by simply transforming the abstract language into the language required by the owner. This will save a lot of time and money.

Lingu [1] is a lightweight language that is based on the abstract language idea above. Furthermore Lingu is a domain specific language. Lingu is used specifically only to program application that deals with database transaction. It is not meant to be used as a general purpose language such as Java or C. Lingu consists of program specification and validation script. These parts will be used in generating the program code and test cases or scenarios. Lingu used theorem prover as the tool for verifying whether the test cases conform to the specification described on the script.

This paper discuss about LinguSQL as a tool for verifying and transforming Lingu script, which is an abstract language, into domain specific language in a database environment. LinguSQL's main responsibility is to feed on Lingu script and gives its verification result to programmer. If all of the scenarios are valid, LinguSQL then transform Lingu script into concrete language.

LinguSQL consists of 3 different engines. First is transformation engine. It is responsible for the transformation of Lingu script into concrete language. Second is verification engine. This engine will try to reduce the specification in Lingu script into sets of mathematical formula. These formulas then will be solved by the engine using theorem prover. Currently, the use of theorem prover still needs an expert assistance in the event that the automatic solver did not quite succeed in solving the formula. And the last is testing engine. The engine will try to execute the scenario given on the Lingu script and compare the result to the specification. This is done using dummy database generated by the engine. Finally, the result will be brought to user representing its validity value.

The rest of this paper is organized as follows: section 2 presents a brief introduction of Lingu; section 3 presents the development of LinguSQL; and section 4 presents concluding remarks.

2. LINGU

This section will give a brief explanation about Lingu. Lingu is a lightweight language to program data transformations on database. Even though the language is small, Lingu provide enough expression to program a large class of useful data transformations. Lingu embraces validation and testing as an integral part of the programming. This is reflected by integrating validation script inside Lingu. The validation script validates program specification that is described previously in Lingu script.

Program specification is the part where business logic and process is described. The structural form resembles ordinary object oriented code. Figure 1 is a sample code taken from midterm report of RUTI project [4].

```
type Agenda = Dbase {|
        Old  :: Table Appointment
      ; Main :: Table Appointment  |}

type Appointment = Record
  {| Date     :: Date
   ; Priority :: Integer
   ; Note     :: String   |}

class AgendaUtility (a :: Agenda)

method cleanup (d::Date) :: ()
 do
 { insertAll x<-a.Main where
         x.Date<=d to a.Old ;
  delete   m<-a.Main where
         x.Date<=d
 }

method emptyOld () :: ()
 do
 a.Old := <| |>
```

Figure 1 Study Case Sample Code

The code describes two data type and a class. Data type is analogous to struct in C language or data object in Java. It only contains a collection of data and no method. Since Lingu is intended for database environment use, the structure of the language is created as similar as possible to the structure of database programming. There are two keywords in the data type constructor that express this; the keywords are *Record* and *DBase*. *Record* symbolizes a row in a database table. Data type *Appointment* with *Record* as its keyword means that it represents a single row with *Date*, *Priority*, and *Note* as its data in table *Appointment*. The second keyword, *Dbase* symbolizes a database which consists of collection of table. A table inside *Dbase* represents a collection of *records*. The sample code shows *Agenda* has the type of *Dbase*. This means *Agenda* consists of tables *Old* and *Main*, each is a collection of *record* of data type *Appointment*.

Class of *AgendaUtility* describes methods that can be applied to *Agenda* a. This class has two methods, *cleanup* and *emptyOld*. Basically *cleanup* is a method that moves all appointment data in table *Main* that is older than the

date given and moves it to table *Old*. After it is moved to table *Old*, the appointment data is deleted from table *Main*. The second method *emptyOld* has a very simple function, which is to delete all appointment data from table *Old*.

Validation script is an important part in Lingu. This script describes specification of the validation of program code. Validation script takes form of formal specification. This script then will be broken into small mathematical formulas called verification conditions. With proper tools, like a theorem prover, the conditions can be verified. Figure 2 is a sample of a validation script.

```
validation test1 (i::Integer; d::Date)  :: Bool
  local
  x     :: Appointment ;
  b1,b2 :: Bool
  do
  { x := a.Main!!i ;

    call cleanup(d) ;

    b1  := find  x' <- a.Main
           where x' = x
           found T otherwise F ;

    b2  := find x' <- a.Old
           where x' = x
           found T otherwise F
  }
  return (b1 \/ b2)

  pre  0 <= i  /\  i < #a.Main

  post return=T

scenario
  test1(0,currentDate()) ;
  test1(#a.Main-1,currentDate()) ;
  test1(#a.Main,currentDate())
```

Figure 2 Validation Code

The script describes a validation called test1 which accept an integer and a date as its parameter, and returns a Boolean value. In short, the script above tries to ensure that method cleanup only moves item from Main to Old, that the method will not delete the item from database. After this script is run, the results will be checked according to precondition (pre) and postcondition (post) script. Note that a.Main!!i means i-th element of a.Main and #a.Main means member count of a.Main.

Scenario is a set of validation script calling with different parameters or conditions. The above example shows the test1 calling with different range of parameters. Each result of the scenario is checked against the expected value defined on validation script. The validity value then will be shown to programmers.

When verification steps are done, then the Lingu script will be transformed into conforming concrete language. The transformation process follow predefined rule in order to guarantee compatibility with intended application platform. Figure 3 are the result of transformation Agenda data type from Lingu to Java.

```
public class Agenda {
        private Appointment[] Old;
        private Appointment[] Main;

        public void setOld(
              Appointment[] Old)
      {
                this.Old = Old;
        }
        public void setMain(
              Appointment[]Main)
      {
                this.Main = Main;
        }

        public Appointment[] getOld()
      {
                return this.Old;
        }
        public Appointment[] getMain()
      {
                return this.Main;
        }
}
```

Figure 3 Lingu to Java Transformation Result

3. DEVELOPMENT

LinguSQL transforms and verifies Lingu scripts. This is done in 3 steps. The first step is to verify the script itself. The verification engine will try to verify the script logics using theorem prover. Although this step is ideally done in an automatically fashion, currently the engine still requires expert assistance to help in solving the verification problem.

The second step is to test the execution result according to the scenario described in the script. The scenario will try different values on validation script and

execute it with a working database. This engine will also automatically populate the database with random inputs according to the script.

The last step is to transform the Lingu script into intended domain specific language such as C, Java, or VB. The transformation is done following certain rules rule to achieve consistency in executing the script.

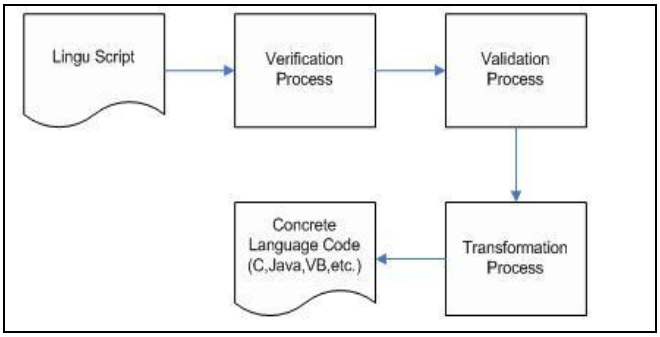

Figure 4 LinguSQL Flow

3.1 Verification Process

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description and specifications [2]. The specifications that are needed to be fulfilled in this case are precondition and postcondition of a method or operation. Precondition is a state that is needed to be fulfilled before an operation is executed, while postcondition need to be fulfilled after an operation is executed. This specification follows Hoare logic in {P}S{Q} form [3], even though it is written in Lingu grammatical form. For example test1 state what is in Hoare logic would look like this [4]:

for all i and M:
$\{0 \leq i \Lambda i < \#a.Main \Lambda a.Main = M\}$
a.cleanup()
$\{M!!i \in a.Main \ V \ M!!i \in a.Old\}$

Figure 5 Case Study Hoare Form

The process will break the script into a set of mathematical prepositions called verification conditions. These verification conditions then are sent to a Higher-Order Logic (HOL) theorem prover. HOL theorem prover tries to verify these conditions automatically. Because of the nature of HOL is undecidable, sometimes HOL theorem prover cannot solve the problem on its own. It needs an expert assistance to guide and tell HOL backend what tactics or formula to be used. If all of these conditions are solved then the scripts are safe to be used on validation process.

3.2 Validation Process

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model [2]. The validation process takes a scenario and executes it with data and in an environment that resembles the real world. The scenario is a set of validation script execution with different inputs and parameters. Inputs that are given ideally must cover all the possibility. When it cannot be done then at least the inputs are given within range of possible data. These inputs will be executed and returns a value. The value from the execution of the scenario will be compared with the expected value given in postcondition script.

The execution of scenario demands an environment that resembles real world. This includes the data that are being used. To accommodate the needs, we also develop an automatic test generation. The automatic test generation will generate a collection of data that resembles the actual data. This is done by using a constraint for the data that are being generated. For instance, if we will generate a salary data, then some constraints are needed to make the salary data looks real. Such as salary data cannot be negative or zero, or that the sum of all salary cannot exceed the corporation salary budget. Ignorance of this issue will degrade the credibility of the verification and potentially will create problem when it is implemented in actual use.

When all of the results obey the rules stated in postcondition script, then the transformation into concrete language can be done.

3.3 Transformation Process

The process transforms Lingu script to a domain specific language. Transformation process takes place when Lingu script has been verified and validated. This guarantee that the transformation result will inherit the same nature as the abstract language it is derived from. The expected result is that the domain specific code is safe and conforms to the specification stated in Lingu script.

Each language has its own characteristics and uniqueness. System design that tries to handle all transformation of the language in one big application is inefficient. This leads to the design that support plug-in style module for each different language. System supports that covers all language is pretty much a big task to do.

That is why currently we only develop widely used concrete language such as Java.

4. CONCLUDING REMARKS

In this paper we have presented LinguSQL, a tool for verifying and transforming Lingu scripts. Currently the work is in the development phase of LinguSQL prototype. For the development we use the case study of SPMB or SET (Student Entrance Test). We manage to identify critical points that need to be considered in SET process. These critical points are in simple mathematical forms. The problem can be easily solved with available theory and formula on HOL theorem prover.

Even though the language itself is still under heavy revision, we are designing a prototype that is robust enough to accommodate changes in grammatical or structural language of Lingu. This prototype hopefully will provide guidance in further development of LinguSQL.

REFERENCES

[1] I.S.W.B Prasetya, H. Suhartanto, Y. Stefanus, A. Azurat, Jimmy, S. Aminah, "Lingu : Concept Document" draft ,July 2004.

[2] D. Coughlin, "An Integrated Approach to Verification, Validation, and Accreditation of Models and Simulations", Proceedings of the 2000 Winter Simulation Conference, 2000.

[3] C. A. R. Hoare. "An axiomatic basis for computer programming". *Communications of the ACM*, 12(10):576-585, October 1969.

[4] H. Suhartanto, B. Widjaja, L.Y. Stefanus, S. Aminah, Jimmy, I.S.W.B. Prasetya, A. Azurat, "Midterm Report of RUTI Project Year 2004", Report of RUTI project chartered by UI team and Utrecht team. July 2004.