A FRAMEWORK OF A LOGIC-BASED TRANSFORMATION SYSTEM

Jeffrey J.P. Tsai, Ray-Yuan Sheu and Bing Li
Department of Electrical Engineering and Computer Science
University of Illinois, Chicago, IL 60680

tsai@bert.eecs.uic.edu

Abstract

In spite of advances in various transformation systems [2, 3], the transformation of a nonmonotonic logic-based requirements specification into a procedural (imperative) language program has not been investigated. This paper presents a framework for a transformation system which can transform a nonmonotonic logic-based specification language, FRORL, into various kinds of procedural language programs. We discuss how to handle nonmonotonic inheritance in FRORL. We establish a matrix-based data flow and dependency analysis mechanism to find out all the possible data transformation paths in a logic-based specification. We also build an algorithm to adjust the execution sequence of a logic-based specification so that the functions included in the logic-based specification can be represented by a procedural language program.

1. Introduction

The transformation of high-level or logic-based languages has been studied by many researchers [2, 3] in different aspects. Some researchers are interested in the transformation of a logic theory into another logic theory based upon some criteria such as specialization or optimization. Some researchers are interested in the transformation of high-level specification languages into detailed implementations in different languages. Some other researchers are interested in the transformation of logic languages into functional languages based upon lambda calculus. However, the transformation of a nonmonotonic logic-based requirements specification into a procedural (imperative) language program has not been investigated. In this paper, we use the techniques from knowledge engineering to facilitate the transformation of a nonmonotonic logic-based specification, represented in FRORL [5, 6], into a procedural language program.

The application of knowledge engineering to software engineering has the possibility of improving the productivity in software development with folds of magnitude. One of the ultimate goals of applying knowledge engineering to software engineering is to have a tool of automatic programming. Automatic programming would allow users to specify what a software system have to do and then generate program codes automatically. To tackle the problem of automatic programming, researchers in this area have cut the scope of the problem down in several ways. Basically, they can be divided into at least five approaches [4]: (i) deductive program synthesis, (ii) synthesis by program transformation, (iii) knowledge-based automatic programming, (iv) high-level abstract specification, (v) other approaches such as intelligent assistance. In this paper, we present a framework of a logic-based transformation system which can be considered as knowledge-based program transformation with the combination of (i) high-level abstract language, (ii) program synthesis by program transformation, and (iii) knowledge-based automatic programming. The goal of our transformation approach is to generate target codes in procedural programming languages based upon the requirements specification represented in FRORL.

The organization of the paper is arranged as follows: Section 2 gives an overview of our transformation system. Section 3 discusses different ways to handle nonmonotonic inheritance structure in FRORL specifications. Section 4 describes how to adjust the specification clause by pre-transformation to facilitate further analysis. Section 5 describes a matrix-based mechanism to decide the data flow and dependency information within a logic-based specification. Section 6 studies the execution sequence adjustment based on the data flow information. Finally, we give a conclusion and future research in Section 7.

2. FRORL-Based Transformation System

In this section, we give an overview of our transformation system as shown in Figure 1. The input to our transformation system is a FRORL specification, and the output is the target code written in a procedural programming language. There are five main steps in our transformation system. The first step is nonmonotonic inheritance processing, in which a FRORL specification is transformed into a Horn-clause logic-based specification. The pre-transformation, which includes Equal-Introduction, Equal-Substitution, Decomposition and Simplification, is used to transform a logic-based specification into some entailed form but still within logic specification category in order to make the data flow and execution sequence analysis easier to apply. These operations often need to be applied again and again to make necessary adjustment to a logic-based specification along with the progression of a transformation process. The data flow and dependency analysis is used to automatically deduce mode information of
clauses in a logic-based specification without the users' input. These mode information is essential to make adjustment to the execution sequence of a logic-based specification so that the execution sequence of the adjusted specification can be represented by a procedural language program which is normally sequential. The last step of our transformation system is to model back-tracking control mechanism and generate target codes. This paper will discuss activities involved in the first four steps.

Figure 1. System Organization of the Transformation System.

3. Nonmonotonic Inheritance Processing

The transformation of FRORL into Horn-clause logic is straightforward since the theoretical foundation of FRORL is based upon Horn-clause logic. On the other hand, the transformation of FRORL into imperative(procedural) programming languages will involve deep analysis of semantic interpretation of FRORL. Since a pure logic system does not have any preferred control mechanism over the order of choosing clauses to perform unification and resolution, the transformation system for different control mechanisms will vary accordingly. The operational semantics of FRORL is the first issue to be clarified and defined in order to address the FRORL transformation. Next one is to investigate the nonmonotonic inheritance reasoning (NMIR) of FRORL. FRORL provides the nonmonotonic inheritance reasoning mechanism to ultimately support the code-sharing and the reusability of existing codes. To transform a FRORL specification into target codes, we need to design a specialized inference procedure for NMIR in FRORL.

The transformation of a FRORL specification into a Horn-clause logic-based specification can be divided into two different approaches based upon the characteristics of nonmonotonic inheritance. The first approach is to expand all the frames to their ultimate sets and therefore drop out the inheritance mechanism used in FRORL. The second approach will preserve the computational mechanism of nonmonotonic inheritance in FRORL during the transformation into Horn-clause logic form. At a glance, it seems that the first approach is much easier to achieve and the query will be more efficient. However, there exists a serious drawback over it when a FRORL specification contains large and deep frame hierarchies. The expansion of a frame can be exploded in exponential ratio. This problem will be further complicated by having multiple inheritance existing among the frames. There is another problem which is the maintenance of the expanded clauses. Whenever there is a change over some parent frame, we will have to regenerate its child frames and descendent frames in order to maintain the consistency. Considering the second method, it will be natural to keep the inheritance mechanism in the transformed logic-based specification by using some kinds of inheritance meta interpreters. But this approach has its own problems, too. That is the efficiency of execution during query of goals since the interpreter will have to perform the inheritance inference every time. This problem will make the already slow execution of the interpreter much worse. It is very clear that the choice between these two approaches is the trade-off between computing time and storage space. We have developed algorithms to convert a FRORL specification into a Horn-clause logic-based specification.

4. Pre-Transformation

The transformation from a logic-based specification into a procedural program cannot be done directly using the transformation rules because the highly encrypted semantics of a logic-based specification sometimes may involve with terms or definitions which are totally different from those used in a procedural programming language. For example, the most commonly used list construct function in a logic-based specification language, 

\[ [I] \]

contains several procedures to be executed in certain argument-dependent execution sequence and this sequence is not determined until we interpret the logic semantics by transforming it into some entailed logic form but still within Horn-clause logic semantic domain. The sequence of execution may change from input to input. This kind of abstraction is the characteristics of a logic-based specification language and also the main problem during the transformation of a logic-based specification into a procedural program. It
implies that there should be some (pre-)transformations to be done before the actual transformation into a procedural program. We call this kind of transformation the equivalence-preserving transformation because it is from a logic-based specification into another logic-based specification which can preserve the "equivalence" in terms of equivalent logic semantics and equivalent input/output.

During the equivalence-preserving transformation, we need to obtain all the implicit and explicit constraints from the underlying logic-based specification language features to eliminate impossible or useless interpretations. An example of implicit constraints exists within the literals of the same clause body, such as the same variable exists in two predicates of a clause. The explicit constraints could be the information supplied by users or defined by the features of the underlying logic-based specification language such as built-in operations, functions, or user-defined predicates.

In this paper, we identify four basic operations to transform a logic-based specification into an equivalence-preserving form which is suitable for applying matrix based flow analysis and we will call them as "pre-transformations". These four basic operations are listed as follows:

Equal-Introduction: This operation is to introduce a new variable for each argument of the literal in the head of a clause and replace the argument with the new variable, then make the new variable equal to the argument in the clause body. The purpose of this operation is to rename arguments of the literal in a clause head to unify the names of all "equivalent" arguments and simplify the arguments of the literals in a rule clause.

Equal-Substitution: This operation is to substitute all the occurrence of the original arguments with the new variable names which are "equal" (=) to them. The purpose of this operation is to create chances to compress a clause into a compact form.

Decomposition: This operation is to decompose a built-in predicate or function in the body of a clause by expanding the predicate or function into some simpler forms to make the flow analysis easier to apply.

Simplification: This operation is to simplify the representation of a logic-based specification. The operation can be categorized into two different types. One is to adjust a clause form, and the other is to apply partial evaluation to a predicate or function if all of its parameters are bounded. The first group includes Folding Literals and Clause Pruning. The second group includes Partial Evaluation.

These operations for pre-transformation often need to be applied over and over again since one transformation may explore some opportunities for other operations to become applicable. Some operations depend on certain mode information to invoke a specific transformation among multiple choices. As a result, these operations are applicable only when the mode information of the related predicates has been obtained to a certain degree. We have developed rules to support various pre-transformation operations.

5. Data Flow and Dependency Analysis

To transform a logic-based specification, we first need to perform data flow and dependency analysis, the execution control sequence can be obtained from the result of the data flow and dependency analysis.

In this section, we present an approach that uses a matrix representation to perform the data flow and dependency analysis based upon the four-valued mode lattice. The matrix approach uses a two dimensional array in which the labels on the top row represents the arguments of all literals including the literal in the clause head of a given rule clause being analyzed, and the labels on the first column represent the predicate names of all literals including the one the in clause head. The entries of a matrix are based upon the four-valued mode domain, (empty, input, output, both). For each parameter of a literal, we adopt the following notations:

- Use "+" to mean an "input" mode,
- Use "-" to mean an "output" mode,
- Use "?" to mean both (don't know), and
- Use "_" (empty) to mean empty value.

Note that, the "input" mode of an argument of a literal means that the literal will "receive" the "data flow" of the argument coming into it, and the "output" mode means that the literal will "transmit" the "data flow" of the argument out of it. Also, the argument which generates an output data (exports a value for the argument) is said to have an "output" mode and the argument which consumes an input data (needs an imported value to bind the argument) is said to have an "input" mode. "?" means that there is no mode information about the parameters in a predicate. "_" means the argument in a column does not occur in the corresponding predicate of a row, so there is no mode information in the cell. In order to handle multiple mode combination more conveniently, we also introduce a new mode status "B", which stands for "Both", in the matrix. It has a similar meaning as "?". The difference between "?" and "B" is on the mode information contained in them. "?" represents that we have no idea about the mode of a cell; "B" implies that we have already explored the cell and the possible mode in the cell could be both. The actual mode is a random combination of all possible modes.

An illustration of a matrix used for flow analysis is shown in Figure 2. From the matrix we can see that the argument S occurs in the literal v with mode "?" filled in the cell; while arguments S and T do not occur in the predicates v and u respectively because these two cells are shaded; predicates p and q have mode "_+" and "-" in the cells related to the arguments X and Y. These are the five possible modes which may occur in an analysis matrix.
The mode values of the arguments of all the literals in a clause body follow the normal definition of data flow: input with "+" and output with "-". But for a predicate in a clause head, the mode convention for the arguments in it should be made opposite to that of the arguments of the predicates in a clause body. Because a clause head predicate serves as a dual role: one is to unify itself with the calling(invoking) literal in order to pass the data flow in both directions, and the other is to provide communication between the arguments in the clause head predicate and clause body predicates which share the same name. The definition of the modes of parameters in a clause head predicate is based on the first role a clause head predicate plays. But when we consider the mode analysis of a specific clause using a matrix, in order to conserve the balance of the analysis, we interpret the meaning of the mode of a clause head predicate by the second role it plays in the data flow transmission. So the mode convention in clause head is opposite to that in clause body. From here we can see that the mode convention for the arguments in clause head should be made opposite to that of the arguments in clause body.

All the predicate symbols include symbols in a clause. All distinctive argument that in the clause head of a clause:

![Figure 2. An example matrix used for data flow analysis.](image)

Boundary of clause "C1"

\[ S \ni \ldots, R, \]
\[ P(\text{Var}_1, \text{Var}_2, \ldots, \text{Var}_K, \ldots, \text{Var}_N), \]
\[ \quad \text{Output from literal } P \text{ for } \text{"Var}_K" \]

First role a clause head predicate plays

Boundary of Clause "C2"

\[ \ldots, \]
\[ P(\text{Arg}_1, \text{Arg}_2, \ldots, \text{Arg}_K, \ldots, \text{Arg}_N), \]
\[ \quad \text{Output from clause head } P \text{ for } \text{"Arg}_K" \]

Second role a clause head predicate plays

Input to "Arg" of literal "U"

![Figure 3. An illustration of data transformation by a clause head predicate.](image)

If a clause \( C_2 \) with the clause head \( P \) being called by a predicate \( P \) in the clause body of another clause \( C_1 \), then the predicate \( P \) in \( C_1 \) may provide data to the predicate \( P \) in \( C_2 \) by unifying the variables \( \text{Var}_K \) and \( \text{Arg}_K \) in \( C_1 \) and \( C_2 \) together. So in this case the mode of \( \text{Var}_K \) of \( P \) in \( C_1 \) is output, and we have to define the mode of \( \text{Arg}_K \) of \( P \) in \( C_2 \) as input in order to conserve the balance of data flow. On the other hand, while \( \text{Arg}_K \) of \( P \) in \( C_2 \) receives data from \( \text{Var}_K \) of \( P \) in \( C_1 \), \( \text{Arg}_K \) of \( P \) in \( C_2 \) also provides data to the predicate \( U \) in the body of clause \( C_2 \) by sharing the same variable \( \text{Arg}_K \). Since we always perform mode analysis within a clause, we have to let the mode convention of clause head be identical with that of a clause body. So the same sign convention in clause head represents exactly the opposite meaning as that in clause body. From here we can see that the mode convention for the arguments in clause head should be made opposite to that of the arguments in clause body.

To start the process of the data flow and dependency analysis, we mark all non-empty entries of a given rule clause as "?" because, at this initial step, we really have no knowledge of the mode information yet. The next step is to collect and apply all possible known mode constraints of the literals within the rule clause such as those of user-defined predicates and the mode information for all built-in operators used in this rule clause. Then we are ready to apply the algorithm based on matrix representation to do the data flow and dependency analysis. The result of the analysis is a matrix marked either as "+", "-" or "B", which means that from the analysis we find out what each of the parameter acts in realizing the functions described by the logic clause. This is an important step in the transformation system. The execution sequence determination and the interpretation of logic-based specification in a procedural programming language will base on the result of this matrix.

There are two assumptions for the data flow and dependency analysis algorithm. If we restrict our attention to an argument position in a literal, then we can refer to the literals with the name "producer" and "consumer". If a literal "consumes" the variable value generated by other literals, we call this literal as a "consumer". If a literal "produces" (binds) the value for an ungrounded variable, we call this literal as a "producer". According to this notation, we define these two assumptions in the following:

**Exact-One-Producer Assumption:** This assumption assumes that any argument \( \text{arg} \) with the mode of "+" will demand "exactly" one producer for the value to be consumed by \( \text{arg} \). No more than and no less than one producer should be allowed. This producer could come from a column other than that where mode "+" resides in.

**One-Producer-Many Consumers Assumption:** This assumption advocates that one producer, except in the case of dead end, should have "at least" one consumer. This assumption actually conforms to the exact-one-producer assumption in the sense that they are complementary to each other.
6. Execution Control Sequence Determination

The result of our data flow and dependency matrix can be used to determine the execution control sequence for the literals in a specification clause. We have built an algorithm to determine the execution control sequence. The input to the algorithm is the resulted matrix of the data flow and dependency analysis of a clause, the output is the adjusted execution sequence of the clause from procedural point of view. The main idea behind it is to check a literal in a clause whether it has all of its arguments with the mode "input" been bound with values. If it is satisfied, then the literal is "legal" for next execution sequence. And its arguments with "output" mode can be used for further binding of the arguments in other predicates. The algorithm presented in Figure 5.

INPUT:
1) a rule clause of the given logic specification,
2) mode information of built-in operators,
3) mode information of all known predicates such as user-defined ones.
4) initial matrix with all the cells marked only as either empty or "?".

OUTPUT:
A result matrix with proper mode values filled for the cells.

BEGIN
Update the built-in and user-defined predicates with their mode values;
Push all the built-in and user-defined predicates in Literal Stack;
WHILE (the Literal Stack is not empty) DO
    Pop the literal from the Literal Stack onto "Literal";
    FOR each column "Col" of "Literal" in the matrix DO
        IF "Col" is filled with "+" THEN
            Report "ERROR";
            END IF;
            IF there is only one row with "?" in the same column "Col" THEN
                Update the cell with "?" to "-";
            END IF;
            IF "Col" is filled with "?" THEN
                IF no other row with "?" or "B" in the column "Col" THEN
                    Mark it with "-";
                ELSE
                    Mark it as "B";
                END IF;
            END IF;
        END FOR;
    END WHILE;
Output data dependency matrix;
END; {Algorithm}

Figure 4. Data Flow and Dependency Algorithm.
INPUT:
1) a rule clause of the given logic-based specification,
2) the resulting matrix of data flow and dependency.
OUTPUT:
Generate allowable execution sequence for the rule clause.

BEGIN
Initialize an array, \( A[1.Total\_number\_of\_arguments] \) to empty value;
Initialize an array, \( L[1..Total\_number\_of\_literals] \) to empty;
\( L_{idx}=0; \ A_{idx}=0; \)
FOR each column in the matrix with mode "B" in it DO
    Change the mode "B" in the first row of the column to ";";
    Change all the remaining mode value "B" in the same column to "+";
END FOR;
Mark all literals as unparsed;
Mark the row "Row" of clause head as parsed;
FOR each column "Col" with the mode of ";" on the row "Row" DO
    Update the array, \( L[+L_{idx}] \), with the literal on "Row";
END FOR;
WHILE (there is an unparsed literal in the matrix) DO
    Select a row "Row" from the unparsed literal for each column with mode "+" in the row has been included in \( A/J \);
    Output the row "Row" as the next execution literal;
    FOR every column "Col" with a mode ";" on the "Row" DO
        Mark the array \( A[+A_{idx}] \) with the column "Col";
    END FOR;
END WHILE;
Return array \( L/J \);
END; \{(Algorithm\)

Figure 5. Execution Sequence Determination Algorithm.

...ren's approach [1], which also considered the four-valued mode domain, our method is more powerful in that our method considers multiple-mode combinations and can provide more specific mode information of predicates to users. Currently we are working on the backtracking and code generation problem of the transformation system.

Acknowledgement
The work reported here is partially supported by Fujitsu America, Inc., by National Science Foundation under a Grant CCR-9106540, and by GTECH.

References