Software Synthesis Shell SOFTEX/S

T. Yamanouchit, A. Sato†, M. Tomobe†
H. Takeuchit, J. Takamuratt and M. Watanabet†

†C&C Systems Research Laboratories
NEC Corporation
4-1-1 Miyazaki, Miyamae-ku,
Kawasaki, 216 JAPAN
‡C&C CASE Engineering Division
NEC Corporation
§NEC Systems Laboratory, Inc.

Abstract

Many research activities have been reported in order to improve the software productivity in many different approaches. However, when current real task software development environment is studied, software development automation is still not sufficient. The authors' goal is to bring techniques for software automation into such software development environment. This paper approaches to this goal by providing a software synthesis shell SOFTEX/S, to support the low cost construction of software model specific synthesis systems.

SOFTEX/S includes a transformation system, based on a term rewriting system, a language DSL/C++ for defining transformation rules and a specification language for a specific software model, as well as a rule verification system which supports developing correct transformation rules. SOFTEX/S has been implemented and a program synthesis system for a state transition model SOFTEX/STM has been developed with SOFTEX/S. Based on the results of this experience, SOFTEX/S has been evaluated to improve the real task software productivity, while maintaining the desired software quality in terms of execution speed and memory usage.

1 Introduction

Many research activities have been reported in order to improve the software development environment, in many different approaches. For example, when they are grouped in terms of the software development phases supported by the approaches, all the phases, including requirement acquisition[1] [2] [3], algorithm design[4] [5], reuse of program constructs[6], program derivations [7]-[13], and maintenance[14], are covered.

When focusing on automatic program derivations, REFINE[7] has provided a general purpose wide spectrum language from which lisp programs are generated with transformation rules. TAMPR[12] takes a LISP program as an abstract input specification and generates a FORTRAN program. Unlike these two systems, FINEX[8] and [9] has tried to obtain efficient programs from high level specifications by limiting the program domains. ELI[11] and DRACO[10] have proposed meta mechanisms for extending existing program derivation systems to specific domains.

Though many proposed ideas and mechanisms have shown their effectiveness in laboratory settings, when actual current software development environment is studied, software development automation is still not sufficient. The authors' goal is to apply these techniques to actual software development environment, in order to improve the real task software productivity.

This paper approaches to this goal by providing a software synthesis shell SOFTEX/S, which enables broad range of software engineers to construct a program synthesis system for each specification representation style and the corresponding program structure. A specification representation style together with the corresponding program structure, is called a "software model" in this paper. The software engineers, con-
structing software model specific synthesis systems, do not have to be specialized in program synthesis systems but be specialized in the software model.

Section 2 discusses a case study on the current software development environment and how SOFTEX/S has been designed, based on the case study results. Section 3 illustrates the generic synthesis mechanism for SOFTEX/S, composed of a language DSL/C++, a transformation system, and a transformation rule verification system. Section 4 explains a state transition model specific program synthesis system SOFTEX/STM as a SOFTEX/S application. Section 5 evaluates SOFTEX/S and SOFTEX/STM. Section 6 compares SOFTEX/S with related work. Section 7 concludes the paper.

2 Motivation

2.1 Case study on software development environment

In order to determine the best approach to improve the current software development environment, cases regarding communication software and operating system development, have been studied. The results are summarized as follows;

(i) Development by teams:
In many cases, a team of 5 to 20 software engineers, specialized in a specific software area, work together. They have software development knowhow.

(ii) Specifications are determined before implementation:
First, precise specifications are determined. Then programs satisfying the specifications are implemented. In research communities, more programs are developed in spiral models. However, in many real task software development environments, fixing specifications before implementation is important, in order to define the interface with other modules, which are developed by other teams.

(iii) Each team has its own software model:
Many teams have their own specification representation styles and the corresponding program structures (software models). This improves communication in the team.

(iv) Different software models exist for different teams:

Different teams have different software models reflecting subtle differences in specialized areas.

2.2 Design philosophy

(i) Productivity:
In order to improve the productivity in the environment studied and reported on in the previous section, the software models have been utilized to enable program synthesis from high level specifications, corresponding to the specification representation styles. Fortunately, since precise specifications are determined before implementation, program synthesis approach is suitable for this environment. However, if a program synthesis system has to be developed for each software model, the cost of developing program synthesis systems may cancel the productivity improvement unless the cost is very low.

In order to lower the cost of developing such program synthesis systems, the following approaches have been considered;

- Provide a software synthesis shell, which includes common modules required by most of the program synthesis systems and functions to support building software model specific parts of the synthesis systems.
- Support customizing (reuse and modify) an existing program synthesis system to construct a new program synthesis system for a similar software model.

(ii) quality of synthesized program:
Among several issues concerning the software quality, the correctness, execution speed and memory usage efficiency for programs are considered in this paper. They are important because, if a developed program is not correct or intolerably inefficient, they cannot be used in real task environment, regardless of the productivity achieved.

The correctness of the programs, synthesized by a program synthesis system, depends on the transformation rule correctness as well as the input specification correctness. The input specification correctness can be verified only in regard to the specific software model. Therefore, it should be guaranteed in each specific program synthesis system. However, some aspects of the transformation rule correctness can be verified, independently from specific software models. Therefore,
the software synthesis shell should have such a capability.

In a program synthesis system, the higher the specification level, the higher the productivity. However, achieving efficient programs becomes more difficult as the specification level becomes higher. In actual software development environment, program efficiency is carefully considered for the parts of the programs where efficiency becomes critical, while other parts are implemented without much regard for efficiency. In order to achieve an efficient program without losing high productivity, critical parts of specification should be written with low level language primitives, while other parts should be written with high level primitives in the same specification language. In this paper, such a specification language, with primitives of various specification levels, is called a wide spectrum specification language.

(iii) Compatibility with the current environment:

Besides the productivity and quality, the reading and writing ease for the input specifications for a program synthesis system is important. Therefore, providing graphical interface for input specifications are beneficial to the software engineers, who are not familiar with formal specification languages.

As the result of the above discussions, the system concept for SOFTEX/S has been determined as shown in Figure 1. SOFTEX/S is a software synthesis shell, which supports the construction of a program synthesis system for each software model. It has a language DSL/C++ to define a specification language for each software model as well as transformation rules, a transformation system based on a term rewriting system, and a transformation rule verification system. Therefore, by defining a specification language, transformation rules, and a specification input interface with SOFTEX/S, a program synthesis system for a specific software model can be constructed.

\footnote{In this paper, the word "specification level" is used in the meaning of a degree of abstraction and/or that of software model utilization.}
3 Generic software synthesis mechanism

3.1 System organization

Figure 2 shows the system organization for a program synthesis system constructed with SOFTEX/S. A user defines input specifications, using a specification input interface. The specification input interface checks the specifications, and the user corrects the specifications interactively. Then, the interface converts the specifications into the language representation in a specification language SL/C++ extended from DSL/C++.

The SL/C++ syntax is analyzed by “parser” and an abstract syntax tree is created as its result. This abstract syntax tree becomes an input term for the term rewriting system. The specifications are then rewritten using term rewriting rules partially and stepwisely. When no more rules become applicable, “printer” generates the C++ syntax for the specifications (programs).

The term rewriting rules are also written with DSL/C++. The rule verification system helps defining rules by checking some aspects of correctness for rules and rule modules. In this figure, DSL/C++ (including parser and printer), the transformation system and the rule verification system are provided by SOFTEX/S, while the specification input interface (including specification checker), the specification language SL/C++ and the transformation rules must be constructed for each software model.

3.2 DSL/C++

DSL/C++ is a language which has the following primitives and C++ programming language primitives. It enables the definition of transformation rules, as well as software model specific wide spectrum specification language SL/C++, as an extension of DSL/C++.

```plaintext
termdef sort(parameter1-sort, ... ) term_name
relation relation_name eql (lhs, rhs)
vardef sort variable_name1, variable_name2, ...
```

The first primitive defines the syntax of terms extending DSL/C++. It enables the high level description of specifications, using the defined terms. By defining a set of terms used frequently in a software model, a wide spectrum specification language specific for the software model can be constructed.

The second primitive defines relations between two terms in specifications. It gives semantics to the terms defined with the first primitive. Relations work as transformation rules in the transformation system. In other words, specifications written with terms extended with the first primitive are transformed by the relations defined with this primitive.

The third primitive defines variables with sorts used in relation definitions. It enables generalization of relations by introducing variables. A variable in a relation unifies with any term with the same sort.

DSL/C++ has parser and printer modules, which convert specifications and the corresponding abstract syntax trees from one to the other. The SL/C++ parser and printer are automatically extended from those for DSL/C++ to accept newly defined terms.

```plaintext
terndef 'statement('statement_list) state_switch;
terndef 'statement_list ('identifier, 'identifier, 'statement_list) state_cell;
terndef 'statement('identifier) next_state;

state_switch([. state_cell
  (ev1, st1, 'statement_list
    [ f1(); f2(); next_state(st3) ]) state_cell
  (ev1, st2, 'statement_list
    [ f3(); next_state(st4) ]) .])

(a) Term definitions and a specification

vardef 'statement_list STL1, STL2;
terndef 'statement('statement_list) state_switch;
terndef 'statement_list('statement_list) state_case;
terndef 'statement_list ('statement_list, 'statement_list) stm_list_append;

relation R1 eql
  state_switch(STL1),
  'statement[
    switch(state) {
      state_case(STL1)
    }]

(b) Variable and rule definitions
```

Figure 3: Examples of DSL/C++ descriptions
Figure 3 shows specification and rule definition examples. Figure 3(a) shows an example for a specification with terms extended with the \texttt{termdef} primitive. A backquote(') shows that the following symbol is a sort name. For example, in the first line, “state\_switch” is defined as a term with \texttt{statement} sort, which takes one argument with \texttt{statement\_list} sort. Terms “state\_cell” and “next\_state” are defined in the same syntax.

Figure 3(b) shows examples of a variable definition with the \texttt{vardef} primitive and a relation definition with the \texttt{relation} primitive. For example, in the first line, “STL1” and “STL2” are defined as variables with \texttt{statement\_list} sort. The relation statement defines a transformation rule \texttt{R1}, which rewrites any term which unifies with “state\_switch(STL1)” into the corresponding switch statement in C++ language.

### 3.3 Transformation System

The SOFTEX/S transformation system accepts an SL/C++ specification and generates a C++ program, both in a term representation. SOFTEX/S has adopted a sorted term rewriting system\[16\] as a base for the transformation system. The sorted term rewriting system is defined by a set of rules and rule variables, where each rule can have exactly one term in its left hand side(lhs) and right hand side(rhs), and all variables in its rhs must exist in its lhs. A rule matches with a term, when the term and the lhs of the rule unify and all variables with \texttt{sorts} are bound to terms with the same \texttt{sorts}. If a rule matches a term, the term is rewritten to the rhs of the rule, after substituting variables with their unified terms.

Transformation rules are tested and applied in the following strategy. Each term in the term represented specifications becomes a candidate to be rewritten in left most out most order. Rewriting rules are tested with the candidate term, in the order that the rules are specified. When a rule does not unify with the candidate term, the next rule is tested with the same term. When a rule unifies with the candidate term, the term is rewritten by its rhs, and both the rule order and the candidate order are initialized (the first rule is tested with the first candidate). The rewriting process terminates, when no more rule matches a term.

A rule compiler is provided by SOFTEX/S for efficient transformation. It generates a lisp function for each set of rules, whose out most term symbols of their lhs’s are identical, and indexes the lisp function with the corresponding term symbol. Then, instead of testing rules for each candidate term, the lisp function, indexed by the same term symbol, is evaluated. In each lisp function, the transformation is determined after comparing the candidate term parameters with parameters for each original rule lhs. By using the rule compiler, the transformation speed becomes a function of the average number of rules with the same out most term symbol, instead of the total number of rules. In experiments, the transformation process has been speeded up from 75 to 95 times. As a result, the transformation speed has become satisfactory for application in practical use.

Another framework for efficient execution, as well as ease of writing and understanding rewrite rules, is the introduction of a “module” for rules. Rules can be divided into modules and the above mentioned strategy is applied to each rule module. The rule modules are applied sequentially in the order they are specified. For more complex module structure, the rule module order is controllable by “process knowledge”. For example, selecting data structures and procedures for optimization, depending on the contents of the input specifications, can be achieved by writing a sophisticated “process knowledge” which switches modules to apply. Currently, “process knowledge” are directly represented in LISP language.

### 3.4 Rule Verification

SOFTEX/S supports the development of correct transformation rules by providing a rule verification system in two correctness aspects.

**Transformation termination:**

A set of transformation rules has this feature, if the transformation by the set of transformation rules is guaranteed to terminate. This feature is important, because bugs in transformation rules, such as infinite loops, can be detected by checking this feature for the transformation rules. Several sufficient conditions for this feature are provided as a theoretical research result of term rewriting systems \[17\]. By checking these conditions, SOFTEX/S supports the development of transformation rules, which are guaranteed to terminate.

**Syntactic equivalence:**

A set of transformation rules has this feature, if syntactically correct specifications are guaranteed to be obtained from syntactically correct specifications, by applying the set of transformation rules. The necessary and sufficient condition for a set of transformation rules \(R\) to have this feature is given as follows;

---

32
For each rule \( l \rightarrow r \in R \),
\((l, r \in \{t \mid t \text{ is a term with variables}\})\)
\(\text{Locations}(r) \supseteq \text{Locations}(l)\)

\((\text{Locations}(t) \text{ is the locations in the language grammar where the term } t \text{ is allowed to appear.})\)

This feature is important in regard to two points. (1) The syntactic equivalence of a rule set can be verified by testing the syntactic equivalence of each rule in the set, instead of testing each sequence of rules in the set. (2) Since the necessary and sufficient condition is given, the output specifications/programs are guaranteed to be syntactically correct, if and only if, this feature holds. By checking this condition, SOFTEX/S automatically detects bugs in transformation rules, which might have generated syntactically incorrect programs.

A set of transformation rules is syntactically complete, if a syntactically correct “program” is guaranteed to be obtained with no part remaining untransformed from a syntactically correct specification by the set of transformation rules. A necessary condition for a set of rules to have this feature is that there is at least one sequence of transformation rules applicable to each extended term and that the sequence of the transformation rules terminates with a C++ program. Currently, theoretical study for such conditions for syntactic completeness is being made, but implementation remain as future work.

Not only the syntactic correctness, but also semantic correctness, such as semantic equivalence, can be defined. For example, a transformation rule is semantically equivalent, if the semantics of the specifications, after the transformation has been completed, is guaranteed to be equal to the semantics of the specifications, before the transformation took place.

The semantic features can be theoretically verified by providing another formal semantic representation of the specification, and comparing the semantics of the specifications, before and after a transformation, in the formal semantic representation (denotational semantics). However, there are difficult problems left unsolved. For example, formalizing another semantic representation and transforming a specification language to the semantic representation are both difficult problems when dealing with complex language such as C++. Therefore, SOFTEX/S does not support verification of semantic features. The semantic correctness must be guaranteed through testing process.

### 4 Program synthesis system for a state transition model

A program synthesis system SOFTEX/STM for a state transition model has been constructed with SOFTEX/S.

#### 4.1 Specification language

SOFTEX/STM generates C++ programs from specifications represented in a State Event Matrix(SEM), as shown in Figure 4. In SEM, a procedure for each event for each state is represented in a state predicate, action functions and a next state name. For example, when a system is in a state, and an event occurs to the system, the state predicate specified in the corresponding cell is evaluated. If it evaluates as being true, action functions in the same cell are executed. Then, the system state transits to the state specified as the next state in the same cell.

Figure 5 shows the specification language representation of SEM. In this representation, the term STATE_TBL_LIST represents a list of matrices and the term STATE_CELL represents a cell in a matrix. The state predicate, action function and next state name are represented by terms PREDICATE, ACTION and NEXT_STATE, respectively.

As can be seen from Figures 4 and 5, correspondence between the table representation and the language representation for the SEM is straightforward. SEM specification language has been constructed by defining eight terms on C++ with the termdef primitive of DSL/C++.

Figure 6 shows the State Event Matrix Editor(SEME) as a specification input interface for SEM specification language. SEME allows users to write specifications by filling the matrix cells. The language representations of the edited specifications are obtained as the output from SEME. SEME can verify the input specifications in several aspects, including isolated states, infinite loops, and lack of state definitions.

#### 4.2 Transformation rules

The eight defined terms in the SEM specification language are transformed into C++ representations by transformation rules. Two sets of transformation rules have been developed to support different implementations of SEM specifications. The rules are defined with relation and vardef statements of DSL/C++. The two implementations are as follows;
Event driven: A C++ function is created for each event. Inside each function, a case statement switches the procedure for each state.

State switch: A C++ function is created for an entire state event matrix. The procedure is switched first by each state, and then by each event.

These two implementations have been adopted from program structures for different software engineer teams. The number of transformation rules, rule modules and intermediate terms used during the transformation are provided in Table 1.

Table 1 Two different sets of transformation rules

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Rules</th>
<th>Modules</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event driven</td>
<td>54</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>State switch</td>
<td>49</td>
<td>5</td>
<td>30</td>
</tr>
</tbody>
</table>

Each of five modules for Event driven implementation has the following role in the transformation.

Module 1: Creates a list of states and a list of events.

Module 2: Divide the table into a list of one dimensional vectors by events.

Module 3: Implement the vectors for events in C++ language.

Module 4: Implement data structures.

Module 5: Implement procedures and data structures.

These modules are applied sequentially in this order. By modularizing rules as above, transformation process has become more understandable. For example, checking the correctness (semantic correctness) of intermediate specifications after each module terminates, is a good strategy for debugging rules.

The last module stores rules which are often customized to adjust the input specification style as well as the output program structure for SOFTEX/STM to users' environment. The typical customization points are, default procedures for blank cells, error handling procedures, and declarations for the types and parameters of member functions, etc.

5 Current status and evaluations

SOFTEX/S has been implemented and its effectiveness in a real task software environment is under
evaluation. SOFTEX/S is evaluated in four steps.

(i) Step 1
In the first step, SOFTEX/STM has been developed with SOFTEX/S by one of the SOFTEX/S project members for a given software model. As shown below, this evaluation step has shown that, if used by a software engineer familiar with SOFTEX/S, it is possible to develop a software model specific synthesis system very efficiently.

**Synthesis systems productivity:**
The development cost for SOFTEX/STM (54 transformation rules and SEM specification language with 8 terms) with SOFTEX/S was 5 person days. Another program synthesis system, SOFTEX/SDL, which transforms SDL specifications into C++ programs, has been developed with SOFTEX/S. It has 280 transformation rules and the development cost with SOFTEX/S was 20 person days. These productivity values have been evaluated to be 30 times higher than the standard productivity for human programmers.

Because the SOFTEX/S transformation system has adopted a stepwise rewriting scheme, transformation rules could be easily customized to adjust the input specification style and output program structure for each user’s environment. A team of software engineers could not use SOFTEX/STM without modification, because their software model was slightly different from that of SOFTEX/STM. However, modifying 6 out of 54 rules has adjusted SOFTEX/STM to their model. This customization was made in only one day by a software engineer not specialized in program synthesis systems. As another customization example, C programs, instead of C++ programs, have been generated from the state event matrix by customizing 10 of the 54 original SOFTEX/STM rules.

The rule verification system has been effective in finding potential bugs in transformation rules especially in syntactic equivalence aspect. Since syntactic equivalence verification checks the necessary and sufficient condition, a rule not satisfying the condition is determined to have a bug.

(ii) Step 2
In the second step, SOFTEX/STM has been evaluated by software engineers specialized in the software model. The evaluation has been accomplished by synthesizing a real task software with SOFTEX/STM, and comparing its output programs with the programs coded by expert programmers from the same specifications. This evaluation step has shown that a program synthesis system, developed with SOFTEX/S, is practical in all the aspects listed below.

**Target software productivity:**

The productivity improvement in the programming phase for the State Transition Model software has been evaluated as being very large. In several teams of software engineers, state event matrices had been written by hand as parts of informal specifications, before SOFTEX/STM was introduced. Thus, writing the specifications with SEME is not extra work for them. On the other hand, C++ programs are obtained which are almost free with SOFTEX/STM, indicating that almost all the entire coding cost can be reduced.

**Output program quality:**

Although the main contributions of the rule verification system is to support developing correct transformation rules, the effect on output program quality is valuable. The SOFTEX/STM output programs have been guaranteed to be syntactically correct, in regard to the grammar summary of [15].

C++ programs, synthesized by SOFTEX/STM, have as good quality as those coded by expert programmers, in terms of both memory usage efficiency and execution speed. This evaluation has been made by a team of software engineers specialized in a state transition model.

**Compatibility with current environment:**

The SEM specification language constructed with DSL/C++ is a wide spectrum language with C++. This feature has helped SOFTEX/STM to be used in combination with conventional programming support tools. For example, an existing library of functions has been utilized in specifying the action functions in the state event matrix.

\*CCITT standard Specification Description Language[18].
(iii) Step 3
Currently the authors are in the third evaluation step. In this step, program synthesis systems are developed with SOFTEX/S by broad range of software engineers. SOFTEX/S has been released to several teams of software engineers, specialized in switching machine software, transmission device software, operating systems, etc. A team specialized in switching machine software has developed a program synthesis system with SOFTEX/S for new service scenarios in switching machine software. It consists of 387 rules in 28 modules. The rules were written in 3 months, by a software engineer who had not known program synthesis systems nor rewriting systems.

The productivity is lower than that of SOFTEX/SEM, however, without SOFTEX/S, developing such a program synthesis system was very difficult. Currently, the specification input interface for this program synthesis system is under development. The quality of the program synthesis system is to be evaluated upon its completion.

The SOFTEX/S generality is also evaluated in this step. Three program synthesis systems have been developed with SOFTEX/S. Two of them have been evaluated to be useful in actual software development environment, one has not yet evaluated. The generality evaluation must wait until more program synthesis systems to be developed and evaluated. SOFTEX/S is being planned to be applied to business software as well as system software.

(iv) Step 4
The last step evaluates the total productivity improvement in the organization, by comparing the cost of developing software model specific synthesis systems and the productivity improvement achieved by them.

6 Related Work
Several systems have been reported in order to support developing program synthesis systems specific for users' problem domains. DRACO[10] supports the construction of domain languages specific for users application domains, by introducing a domain hierarchy. When a domain language for a new domain is necessary, users can develop one by defining (1) a parser, (2) a prettyprinter, (3) transformations (4) components, and (5) procedures. Refinement rules in components rewrite specifications in the new domain language, not directly into the target language, but into already defined domain languages in the hierarchy.

EL[1][11] supports construction of abstract programming languages with high level constructs, by syntactically extending a base language. Then, abstract programs in the extended language are refined to concrete programs by transformation rules and definitions for procedures, types and data objects. Transformation rules replace high level constructs into more concrete constructs in several stages successively.

REFINE[7] has a general purpose specification language, REFINE language, with abstract data structures and operations. Specifications in REFINE language are transformed into lisp programs by a set of transformation rules. REFINE provides a mechanism for defining a domain model by creating object classes, on top of REFINE language. By using this mechanism, users can define even higher level specification language specific for their application domain.

SOFTEX/S is similar to all these systems in design concepts; i.e. specifications should be represented in a language specific for a software model or a domain, from which efficient programs in a target language are synthesized. One difference is that SOFTEX/S transformation system is based on a sorted term rewriting system. This theoretical background enables SOFTEX/S to provide the rule verification system. Though automatic programming based on term rewriting has also been reported[19], there are few attempts made to use such techniques in synthesizing programs in a widely used language such as C++.

However, the main point of this paper is not the development of any component technology for software synthesis. The main point is to combine these technologies to build a software synthesis shell which can be used by broad range of software engineers. In this combination, there is an interesting tradeoff between sophisticated mechanisms which are difficult to use without program synthesis knowledge and relatively simple mechanisms which are easy to use by broad range of software engineers.

7 Conclusions and Future Work
This paper has described the motivations and the generic software synthesis mechanism of SOFTEX/S including DSL/C++, the transformation system, and the rule verification system. While SOFTEX/S is still under evaluation, however, evaluation result obtained so far show that it improves the real task software productivity by enabling low cost development of program synthesis systems for specific software models. They also show that synthesized programs have as
good quality as programs coded by expert programmers.

Steps 3 and 4 in the evaluations are planned for future work, while program synthesis systems, developed during the evaluation, are to be released to real task software development environment.

Acknowledgements

The authors would like to thank Dr. Masahiro Yamamoto, Dr. Toshihiko Yoshimura, Mr. Takeo Ogo, Dr. Satoshi Goto and Mr. Hiroshi Oka for their support for this research. Thanks also go to all members of SOFTEX project for valuable discussions and implementation of the system.

References