A Tool for Assisting the Understanding and Formal Development of Software

Salwa K. Abd-El-Hafiz Victor R. Basili
Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Abstract

This paper presents a program understanding tool which documents programs by generating predicate logic annotations of their loops. The tool is based on an analysis by decomposition approach which utilizes a knowledge base of plans in recognizing the abstract concepts in programs. Using data flow analysis, the decomposition encapsulates closely related statements in events which can be analyzed individually. The first order predicate logic annotations of loops are synthesized from these individual analysis results. A summary of the results of a case study, performed on a pre-existing program of reasonable size, is given. The loops in this study, which are used as test data to the tool, serve to validate our analysis approach. Finally, different applications of the tool are discussed. This discussion focuses on how the tool can assist in the formal development of software using VDM and Z.

1 Introduction

Automated program understanding can assist software engineers in tasks which require reverse engineering such as maintenance and reuse. It can also assist many development tasks such as inspections and code reviews. These important applications have motivated considerable research on the topic. Consequently, many encouraging and useful results, which demonstrate the feasibility of automatic program understanding, are available. A significant amount of this research, however, produces informal natural language documentations which can be ambiguous and uses toy programs to validate proposed approaches.

In this paper, we present a knowledge-based program understanding tool, LANTeRN. LANTeRN is based on an analysis by decomposition approach which documents programs by generating first order predicate logic annotations of their loops. The advantage of predicate logic annotations is that they are unambiguous and have a sound mathematical basis which allows correctness conditions to be stated and verified, if desired. Another unique and important advantage is that they can be used in assisting the formal development of software using VDM and Z.

To validate our analysis approach and study its effect on the size of the knowledge base, we have performed a case study on a real and pre-existing program of some practical value. The loops in this program are used as input data to LANTeRN.

Section 2, of this paper, briefly reviews some related research on program analysis and understanding. It explains some of the issues addressed in our current work. Our loop analysis tool, LANTeRN, is presented in Section 3. Its analysis approach, structure, and analysis results are described using an example program. The different applications of LANTeRN are discussed in Section 4. The focus is on the application of assisting formal software development using VDM and Z. Examples are used to explain when and how LANTeRN can play its assistance role. In conclusion, Section 5 discusses the characteristics and limitations of both our approach and its current implementation.

2 Program analysis and understanding

Considerable research has been performed on the automation of program understanding. Many different approaches, which demonstrate the feasibility and usefulness of the automation, resulted from this research. Some of these approaches are: graph parsing [19], top-down analysis using the goals a program is supposed to achieve as input [13], heuristic-based object-oriented recognition [7], transformation of a program into a semantically equivalent but more abstract form with the help of plans and transformation rules [17, 24], and decomposition of a program into smaller more tractable parts using proper decomposition [8].

Most of these approaches [7, 8, 13, 17, 19] produce program documentation which is, more or less, in the form of structured natural language text. Such informal documentation gives expressive and intuitive descriptions of the code. However, there is no semantic basis that makes it possible to determine whether or not the documentation has the desired meaning. This lack of a firm semantic basis makes informal natural language documentation inherently ambiguous.

Some of these approaches rely on user-supplied information which might not be available at all times. For instance, the goals a program is supposed to achieve [13] or the transformation rules that are appropriate for analyzing a specific code fragment [24] are not always clear to the user. Others have difficulty in analyzing non-adjacent program statements [17]. In addition, a significant amount of program understanding research has used toy programs to validate proposed approaches. Realistic evaluations of the approaches used, which give quantifiable results about recognizable and unrecognizable concepts in real and pre-existing programs, are needed. Such evaluations can also represent a basis for empirical studies and future comparisons with other approaches [21].

To address the above mentioned drawbacks, we present an approach for automating program understanding which is motivated by the idea of analysis by decomposition [5, 25]. It is based on the earlier work of Waters [25] which analyzes programs by decomposing them into smaller parts using data flow analysis. Even though Waters' approach does not address the issue of how to use this decomposition to mechanically annotate loops, it is especially interesting because of its practicality.

Another possible decomposition technique was suggested by Hausler *et.al.* [9]. They suggested the use of program slicing [26] to decompose loops and to allow the abstraction of loop functions one variable at time. Since no detailed investigation or discussion of this idea was offered, it is not clear how it would affect the size of the knowledge base. Even though the resulting loop slices are independent, each slice must include all the statements affecting the modification of the current variable. This can result in large loop slices which make plan design and identification more difficult. A large knowledge base might be needed to compensate for this problem.

The techniques for analyzing dependency relations between program parts were originally developed for optimizing and parallelizing compilers. Of particular interest is the research reported in [4], which performs automatic recognition of recurrence relations.

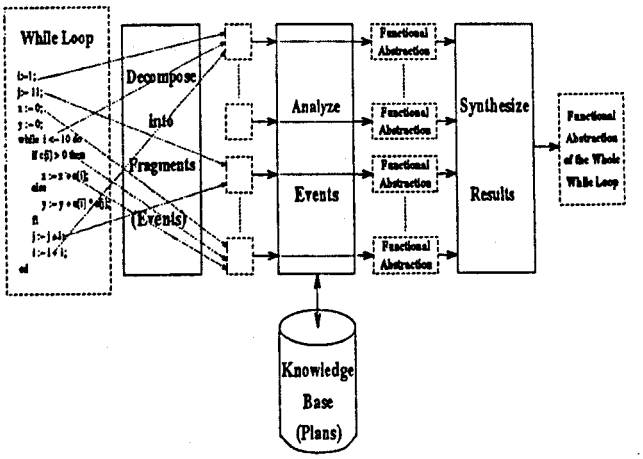

Fig. 1. Overview of the analysis approach.

In this research, the effect of a single loop iteration, on each of the variables computed in it, is summarized in what is called an abstract interpretation. This abstract interpretation only stores information about identifiers and arrays whose net changes in one loop iteration have the forms $identifier := expression$ and $array[index] := expression$, respectively. Recurrence relations in the loop are then identified from the abstract representation and replaced with closed forms using predefined patterns. Since this approach only identifies limited patterns that are useful for parallelizing compilers, it avoids dealing with many complications concerning the loop decomposition and the design and identification of patterns.

The next section starts by giving a brief overview of our loop analysis approach. It then describes the loop analysis prototype and gives a summary of the case study results.

3 A loop analysis tool

Our analysis approach annotates loops with predicate logic annotations in a step by step process as depicted in Fig. 1. The analysis of a loop starts by decomposing it into fragments, called *events*. Each event encapsulates the loop parts which are closely related, with respect to data flow, and separates them from the rest of the loop. The resulting events are then analyzed, using plans stored in a knowledge base, to deduce their individual predicate logic annotations. Finally, the annotation of the whole loop is synthesized from the annotations of its events [1].

An important feature of this approach is its mechanical generation of first order predicate logic annotations. To emphasize this aspect, we explain the

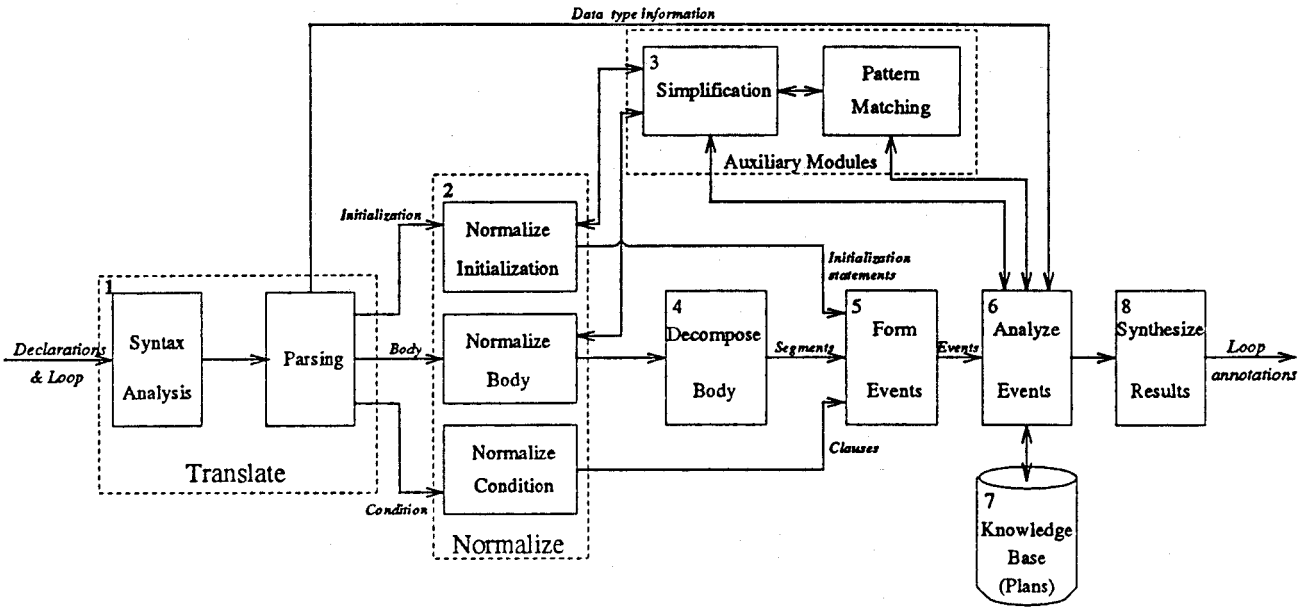

Fig. 2. Structure of LANTeRN.

above analysis steps by describing a prototype tool which implements them. LANTeRN, which stands for "Loop ANalysis Tool for Recognizing Natural-concepts", runs on a Macintosh IIci and is being developed using Macintosh Common Lisp. Fig. 2 depicts the structure of LANTeRN and, at the same time, represents a break down of the blocks in Fig. 1. The loop decomposition into events is performed by modules 1-5 of LANTeRN. The remaining blocks of Fig. 1 have a one to one correspondence with the rest of LANTeRN modules.

In the next subsection, we will describe the function of each LANTeRN module and explain how it contributes to the objective of finding loop specifications. Using the example in Fig. 3, some intermediate, as well as final, analysis results will be shown.

3.1 The analysis technique

The input to the current version of LANTeRN is in the form of a loop to be analyzed, and its declarations, written in Pascal. It is assumed that the input Pascal program has been previously compiled successfully. An example input loop, adapted from [22], is shown in Fig. 3. Throughout this paper, all variable names are written in upper case to distinguish them from other identifiers.

1. Translate: The input is converted into a language independent form. The loop initialization and body are converted into a set of lisp function calls. The loop condition, however, is left in its predicate form. Data type information is also extracted and saved so that it can be accessed during the event analysis. After this conversion, the rest of the prototype can be used to analyze loops independent of the imperative programming language used.

2. Normalize: The loop representation is converted into a standard form which abstracts away from implementation variations such as how a variable is modified during a single loop iteration or how the individual predicates of the condition are connected.

The loop condition is converted into conjunctive normal from [20] and the clauses of the normalized condition are produced. The loop condition in Fig. 3 is already normalized and has the single clause: `J <= HWM`.

The initialization and body are symbolically evaluated. This is performed by symbolically evaluating each execution path in the input individually and then simplifying the result of this evaluation. Finally, the various execution paths are merged to give the net modification performed on each variable.

The output of the symbolic evaluation gives the modifications performed on each variable in the form of a guarded command set which is a set of one or more guarded commands separated by a vertical bar. Every guarded command has a boolean expression as an antecedent of an implication sign and a concurrent assignment as its consequent [3, 6]. When the boolean expression is satisfied , the modifications performed on a variable are given by the concurrent assignment.

For example, the normalized initialization and body

```
var
  NAME: name_type;
  DATE, TODAY: date_type;
  NCARDS, J, HWM : integer;
  DATES: array [1..max] of date_type;
  CARDLIST, NAMES : array [1..max] of name_type;
begin
  NCARDS := 0; J := 1;
  while J <= HWM do begin
    if DATES[J] = TODAY then begin
      NCARDS := NCARDS + 1;
      CARDLIST[NCARDS] := NAMES[J]
    end;
    J := J + 1
  end
end
```

Fig. 3. Example of LANTeRN's input.

of the loop in Fig. 3 are as follows:
Normalized initialization
`TRUE ==> J := 1`, and
`TRUE ==> NCARDS := 0`.
Normalized body
`(DATES[J] = TODAY) ==>`
`NCARDS := (NCARDS + 1)`,
`(DATES[J] = TODAY) ==>`
`CARDLIST[(NCARDS + 1)] := NAMES[J]`, and
`TRUE ==> J := (J + 1)`.

3. Auxiliary modules: Since pattern matching and simplification are central for the efficient and successful operation of different LANTeRN components, they are encapsulated in two separate auxiliary modules. Both modules have been adapted from [18]. The simplification of arithmetic expressions is performed by converting the input expressions into an internal canonical form for polynomials, manipulating them, and converting them back to their external form. Predicate simplifications, however, are limited. They are performed using rule-based translation with the rules being some logical identities.

4. Decompose body: The referenced and identified variables are found for each guarded command set of the normalized loop body. The body is decomposed into ordered minimal segments of code by recursively identifying and isolating the minimal number of guarded command sets which do not have data flow going to other parts of the loop body [25]. The resulting segments are ordered such that the ones identified first are analyzed last. These segments can be characterized as follows:

- Each segment gives the net modification done to some variables(s) in one iteration of the loop.
- The set of variables defined in a segment are not defined in any other segment.
- The ordering relation is an irreflexive partial order in which a segment defining a variable is analyzed before the ones referencing it. This enables the utilization of the analysis results of lower order segments in the analysis of higher order ones.

The ordered segments of the loop in Fig. 3 are as follows:

```
1. TRUE ==> J := (J + 1)
2. (DATES[J] = TODAY) ==>
     NCARDS := (NCARDS + 1)
3. (DATES[J] = TODAY) ==>
     CARDLIST[(NCARDS + 1)] := NAMES[J]
```

5. Form events: The clauses of the condition, the segments of the body, and the normalized initialization are taken as input. The *Basic Events* (BE's), which constitute the control computation part of the loop, are formed by combining the clauses of the loop condition with the segments of the body responsible for the data flow into them (i.e., the segments which affect their truth value). The *Augmentation Events* (AE's) are the remaining segments of the loop body. In addition, each event includes the initialization of the variable(s) defined in it. The advantage of this decomposition into events is that it is based on the structural dependencies, rather than the physical location, of the different loop parts. The ordered events of the loop in Fig. 3 are shown in Fig. 4 as part of LANTeRN output.

6. Analyze events: The resulting events are analyzed using plans, stored in a knowledge base, to deduce their individual predicate logic annotations. The simplest form of a plan is like an inference rule with an antecedent part and a consequent part [7]. First, the loop event is matched with a plan's antecedent. The identified plan's consequent is then used in producing predicate logic annotations of this event. To improve readability, the resulting annotations are simplified whenever possible. The events which cannot be matched, due to an insufficiency in the plan knowledge base, can serve as a guide for its enhancement and evolution.

LANTeRN output for the program in Fig. 3 is shown in Fig. 4. It lists each event along with its generated predicate logic annotations and the name of the plan it matches. The predicate logic annotations are in the form of Hoare-style annotation [11]. That is, they consist of a pre-condition, a loop invariant, and a post-condition. The mathematical notations used are given in appendix A. Those which are necessary for the understanding of Fig. 4 are:

```
EVENTS AND THEIR GENERATED SPECIFICATIONS
-----------------------------------------

1. BE (order 1)
     Clause:          (J <= HWM)
     Segment:         true ==> J :=  (J + 1)
     Initialization: true ==> J :=  1

   The matched plan is DBP-1. The matching result is:
     Pre-condition:  (HWM >= 0)
     Invariant:      (1 <= J <= (HWM + 1))
     Post-condition: (J = (HWM + 1))

2. AE (order 2)
     Segment:         (DATES[J] = TODAY) ==> NCARDS :=  (NCARDS + 1)
     Initialization: true ==> NCARDS :=  0

   The matched plan is SLAP-1. The matching result is:
     Pre-condition:  true
     Invariant:      (NCARDS = card { IND : (1 .. J - 1) | (DATES[IND] = TODAY) })
     Post-condition: (NCARDS = card { IND : (1 .. HWM) | (DATES[IND] = TODAY) })

3. AE (order 3)
     Segment:         (DATES[J] = TODAY) ==> CARDLIST[(NCARDS + 1)] :=  NAMES[J]
     Initialization: -

   The matched plan is SLAP-2. The matching result is:
     Pre-condition:  true
     Invariant:      (forall IND_1 : IND_1 in (1 .. J - 1) and (DATES[IND_1] = TODAY) :
                      CARDLIST[((card { IND : (1 .. IND_1 - 1) | (DATES[IND] = TODAY) }) + 1)] = NAMES[IND_1])
     Post-condition: (forall IND_1 : IND_1 in (1 .. HWM) and (DATES[IND_1] = TODAY) :
                      CARDLIST[((card { IND : (1 .. IND_1 - 1) | (DATES[IND] = TODAY) }) + 1)] = NAMES[IND_1])

SYNTHESIZED SPECIFICATIONS
--------------------------

Pre-condition:  (HWM >= 0)

Invariant:      (1 <= J <= (HWM + 1)) and
                (NCARDS = card { IND : (1 .. J - 1) | (DATES[IND] = TODAY) }) and
                (forall IND_1 : IND_1 in (1 .. J - 1) and (DATES[IND_1] = TODAY) :
                 CARDLIST[((card { IND : (1 .. IND_1 - 1) | (DATES[IND] = TODAY) }) + 1)] = NAMES[IND_1])

Post-condition: (J = (HWM + 1)) and
                (NCARDS = card { IND : (1 .. HWM) | (DATES[IND] = TODAY) }) and
                (forall IND_1 : IND_1 in (1 .. HWM) and (DATES[IND_1] = TODAY) :
                 CARDLIST[((card { IND : (1 .. IND_1 - 1) | (DATES[IND] = TODAY) }) + 1)] = NAMES[IND_1])
```

Fig. 4. LANTeRN's output for the program in Fig. 3.

`card S`	cardinality of the set S
`(forall x: p1: p2)`	for all x values which satisfy p1, p2 is true

An advantage of these annotations is that they have a sound mathematical foundation. A theorem prover can be used, if desired, to verify the correctness of the annotations.

7. A knowledge base of plans: To increase the efficiency of searching for plans which match specific loop events, the plans are divided into eight different categories. This division is based on the kind of events being analyzed (BE's or AE's), loop control computation (similar to *for* loops or not), and complexity of the loop body (nested or not). Within each plan category, the plans used in the case study are stored in a linear list.

The structure of a plan can be generalized to enable (1) the analysis of similar events whose specifications vary depending on their environment (e.g., data types, control computation of the loop, ..., etc) and (2) the detection of cases which have loop specifications that are simpler and more concise.

The generalized structure has a single general antecedent and several consequents organized in one, or more, tree structures. The more general the consequent, the closer it is to the root of its tree. In order to select a specific general plan, a match with the antecedent should occur first. Then, conditions asso-

ciated with the edges of the tree(s) guide the search for the suitable consequent. The instantiation of the information in the selected consequent represents the contribution of this plan to the loop specifications.

Using this generalized structure can lead to a reduction in the size of the knowledge base since several similar plans can be combined together in a larger one having a unique antecedent. Because of space limitations, no plans are shown in this paper. For a detailed description of the plans and their design and structure, refer to [2].

8. Synthesize results: In case of unnested loops, the synthesis is performed by just taking the conjunction of the individual analysis results (see Fig. 4).

The analysis of nested loops is performed by recursively analyzing the innermost loop and replacing it with a sequential construct that represents its functional abstraction [1]. These sequential constructs are in the form of built-in function calls. Outer loops are analyzed using the same steps we have described. However, the synthesis step adapts the inner loop annotations so that they can be proven using Hoare verification rules [11]. For a more detailed description of this synthesis step, refer to [1].

3.2 Case study

A case study was performed, manually, prior to the implementation of LANTeRN. The case study results are, thus, not affected by implementational limitations. The objectives were to validate the analysis techniques and to judge the effect of the decomposition method and the generalized plan structure on the size of the knowledge base. Complete specification of the whole program was not one of our objectives. This case study also served to package our experience in the design and utilization of plans in a specific application domain. All of the designed plans and analyzed loops are gradually being included in LANTeRN's plan knowledge base and test cases data base, respectively.

The case study includes a set of 77 loops in a program for scheduling university courses [12]. The program has 1400 executable lines of code. The loops analyzed have the usual programming language features such as pointers, procedure and function calls, and nested loops.

Out of the 77 loops, we have completely analyzed 65 and partially analyzed 12. We decided not to specifically design plans for the analysis of 12 loops because they were very complicated and specific to the extent that they would not have had a good chance for reuse. The 65 completely analyzed loops contained 213 events. To analyze the 213 events, only 48 plans were designed. These results support the hypothesis that the decomposition method has a positive effect on the size of the knowledge base. The generalized structure of the plans was also useful since only 10 generalized plans covered 139 events.

The results indicate that if we focus on a specific application domain, there is bound to be a kernel of events which can be captured by a relatively reasonable number of plans. On the other hand, there will also be plans which, as in our study, may be used just once. The emphasis should be on the design of the plans that cover the kernel.

4 Applications

Software maintenance and reuse are two well-known and important applications of program understanding research. In this section we briefly explain how LANTeRN can assist both activities. We then focus on an application that is not as traditional. We explain how LANTeRN can be used in assisting formal software development using VDM and Z. This application is possible because LANTeRN's output is in the form of predicate logic annotations.

4.1 Assisting the maintenance and reuse of software

Maintenance comprises several activities such as correcting, enhancing, and adapting existing programs. Since many programs are undocumented, underdocumented, or misdocumented, a major part of the maintenance task is spent in recognizing and understanding abstract programming concepts. Automation of program understanding can, thus, contribute to maintenance tools and methods and provide high-level support for various maintenance activities.

Program understanding is also crucial for code reuse since the reuser must be aware of what a code component does and how to utilize and modify it. Understanding reusable code components can be achieved by augmenting them with a precise and clear description of their functionality. If these descriptions are in the form of formal specifications, they can be further used in generating test cases and assessing the correctness of the implementation. Automation of program understanding is needed to facilitate the quick and efficient population of a reuse repository with well-documented components. Being able to manipulate the documentations automatically can also assist in their identification by designing a retrieval mechanism which utilizes them [15].

4.2 Assisting the formal development of software

Predicate logic plays an important role in the formal development of software using VDM and Z [14, 23, 27]. Since our loop analysis technique produces predicate logic annotations, it can assist such formal development methods. Before explaining how to provide such an assistance, we will briefly explain when it can be provided.

The formal software development methodology in VDM and Z starts by defining, in an abstract notation, the objects manipulated by the system and the invariant relationships that should be maintained. Operations to be performed on the data objects as well as relationships between their input and outputs are also defined.

After specifying the system in a high-level abstract notation, the development proceeds by moving from abstract specifications to more implementation oriented ones. This process involves both *data refinement* (or *data reification*) and *operation refinement* (or *operation decomposition*).

In data refinement, abstract objects are refined onto concrete data types which are available in the implementation language. In operation refinement, implementations for operations are developed in terms of the primitives available in the programming language used [14]. Often, data refinement will allow some of the program control structures to be more explicit, and this is achieved by one or more steps of operation refinement [23].

As the description of the system moves from an abstract level to a more concrete level, proof obligations are generated. These obligations are used in proving that one specification actually implements another more abstract one. They are also used in proving the properties of a given specification.

Because LANTeRN is a reverse engineering tool, it can provide assistance in the last development stage which moves from operation specifications to imperative programming languages implementations. That is, our loop analysis technique can help in showing that the proof obligations generated during the operation refinement process are satisfied. It should be noted, however, that the mathematical notations used in VDM, Z, and our plans are not the same. To transform one mathematical notation to another, simple syntactic variations need to be performed. In order to focus on more important issues, we will assume that the notation in appendix A is a unified notation.

```
Pre-condition:
  (HWM >= 0)
Post-condition:
  (forall IND_1 : IND_1 in (1 .. HWM) and
   (DATES[IND_1] = TODAY) : CARDLIST[((card
   { IND : (1 .. IND_1 - 1) | (DATES[IND] =
   TODAY)}) + 1)] = NAMES[IND_1])
```

Fig. 5. VDM operation Specification.

4.2.1 Assisting the operation refinement process in VDM

In the VDM framework, it is the case that after a series of operation refinement steps, during the design phase, the last refinement step actually implements the specified operations. Since the specifications and proof techniques are closely linked to Hoare-style [11] annotations and proof rules, they are also very close to the specifications produced by our analysis method. To be convinced that a specific implementation actually satisfies the given specification, two alternatives have been suggested [14]. The first alternative requires coming up with an invariant and using proof rules to establish that the implementation matches the specification. The second alternative uses the while loop proof obligations to stimulate program design steps and, in parallel, come up with loop invariants during the code development.

In practice, however, such alternatives are not generally followed because the time and effort needed to perform them are not justifiable. Moreover, carrying out correctness proofs requires a deeper knowledge of the underlying mathematics than that required of a normal user of VDM. That is why we suggest using our loop analysis method as a more practical alternative for validating loop implementations.

Given an operation specification, the software engineer can directly implement it. LANTeRN can then be utilized to produce predicate logic annotations of the implementation. A comparison of LANTeRN's output and the given VDM specification can increase the confidence in the implementation or reveal some mistakes.

For example, if we have the VDM operation specification shown in Fig. 5, a loop construct similar to the one in Fig. 3 can be implemented. To validate this implementation, it can be analyzed by LANTeRN. By direct comparison of LANTeRN's output (Fig. 4) and the VDM specification (Fig. 5), it is clear that the VDM post-condition is implied by LANTeRN's post-condition and pre-conditions are the same. Hence, the implementation in Fig. 3 actually satisfies the VDM

specification.

If the implementation were inconsistent with the VDM specification, a comparison with LANTeRN's output, on the other hand, could have revealed the inconsistency. For instance, if the statement that updates the array CARDLIST in Fig. 3 is changed to: CARDLIST[NCARDS+1] := NAMES[J], LANTeRN would have given the following predicate, which indicates the existence of a fault, as part of the post-condition:

```
(forall IND_1 : IND_1 in (1 .. HWM) and
 (DATES[IND_1] = TODAY) : CARDLIST[((card
 { IND : (1 .. IND_1 - 1) | (DATES[IND] =
 TODAY)}) + 2)] = NAMES[IND_1])
```

4.2.2 Assisting the operation refinement process in Z

To assist the operation refinement process in Z, the same discussion presented for VDM applies. However, the Z methodology is less related to our analysis technique than VDM methodology because of the following reasons:

1. One of the main features of Z is that it describes the properties of an implementation without constraining the implementor's choice of an algorithm. Operation refinement to the code level receives little attention [10, 23]. The focus is on moving from specifications to high-level designs.

2. The specification language of Z uses *schemas* which consist of two parts: a declaration of some variables, and a predicate constraining their values. Even though predicate logic is used, there is no requirement that the pre- and post-conditions be explicitly separated. They can be mixed together in the predicate part. This is because the predicate part describes the effect of an operation on the state of the system by using var? and var! to denote the values of a variable var before and after the operation, respectively.

As a consequence of the first reason, an operation specification is more likely to give a liberal description of what is required by the code. In other words, it is more probable to have a Z specification which is less deterministic than that produced by our analysis technique. This is because proof obligations that guide the movement from an operation specification to code are not well emphasized. For instance, consider an example adapted from [23]. The predicate part of the operation specification in this example has the form: {I: 1..NCARDS! . CARDLIST![I]} = {J: 1..HWM | DATES[J]=TODAY? . NAMES[J]}, where {x: S . e} denotes the set of values taken by the expression e as x takes values from S.

The program shown in Fig. 3 represents an implementation of this specification. When we use LANTeRN to validate that this loop actually implements the Z specification, the output shown in Fig. 4 is generated. It is clear that an extra analysis step is needed to show that in all possible states of the system: (1) the abstract pre-condition implies the concrete one, and (2) the concrete post-condition and the abstract pre-condition imply the abstract post-condition [23], where Z specifications are the abstract ones and LANTeRN's specifications are the concrete ones. The concrete pre-condition, in our example, is implied by the fact that HWM is of the natural type in the abstract Z specification. It can also be shown that LANTeRN's post-condition implies the given Z specification.

With respect to the second reason, a predicate part of the form: (exists I: I in 1..HWM : NAME? = NAMES(I) and DATE! = DATES[I]) cannot be directly compared with LANTeRN's output [23]. The Z specification needs to be transformed so that the pre- and post-conditions are clearly distinguishable. For instance, it can be transformed to: (exists IND : IND in 1..HWM: NAME?= NAMES[IND]} and (forall IND : IND in 1..I - 1: NAMES[IND] <> NAME?) and NAMES[I] = NAME? and DATES[I] = DATE!, where the first line corresponds to the pre-condition and the rest corresponds to the post-condition.

4.2.3 Discussion

We have described how our reverse engineering analysis technique can help in the operation refinement processes of VDM and Z. In addition to the syntactic variations that need to be performed to unify the mathematical notation, two important conditions need to be satisfied for the success of this approach:

1. If our plans use any abstract high-level terms, they need to be the same as the ones used in the high-level VDM or Z specifications.

2. The set of plans stored in the knowledge base need to be designed and tailored to analyze a large set of the loops that might be needed to implement the VDM or Z specifications generated in the design phase.

These conditions can be satisfied if the plans are designed in the environment they are going to be used

in and stored in a repository specific to that environment, where they are allowed to improve and evolve over time. In such cases, knowledge of the commonly used specification and implementation templates can be utilized to make the plans more applicable and to present the analysis results using abstract terms similar to the ones used in the VDM or Z specifications.

For instance, if certain VDM specifications are known to be used extensively in a specific application domain, plans that use the same abstract terminology and analyze the different possible implementations of these specifications are good candidates for storage in the knowledge base. That is, the plans in the knowledge base can be used to augment a library of commonly used VDM specifications. While the specification modules can facilitate the generation of formal specifications and their refinement into designs [16], carefully designed and validated plans can be used by our knowledge-based approach to validate that the implementations do match the specifications

5 Conclusion

We have presented a prototype tool, LANTeRN, for the automation of program understanding. LANTeRN is based on an analysis by decomposition approach which documents programs by generating predicate logic annotations of their loops. The main characteristics of our work can be summarized as follows:

1. It generates predicate logic annotations. Predicate logic annotations increase the confidence in the documentation since correctness conditions can be stated and verified, if desired. The use of predicate logic also makes it possible to assist formal software development methods, like VDM and Z.

2. The decomposition method combined with the generalized design of the plans tend to have a positive effect on the size of the knowledge base.

3. The performance of a case study on a real and pre-existing program of some practical value. This helps in validating our analysis approach and in providing a realistic evaluation of the first two characteristics.

4. The analysis method enables partial recognition and analysis of stereotyped loop fragments which have non-adjacent parts.

5. It is a bottom-up analysis approach which does not rely on user-supplied information that might not be available at all times.

6. It focuses on the difficult task of documenting loops. Analyzing complete program modules is not covered.

Even though our analysis technique can handle most of the commonly occurring structured data types, the only structured type being handled in the current version is the array type. The handling of other structured data types is currently being implemented.

Future work includes investigating the use of the resulting specifications in a larger system which performs intelligent analysis of complete program modules and experimenting with the documentation technique in various domains.

Acknowledgements

We would like to thank Professor Marvin V. Zelkowitz and Dr. Sandro Morasca for their valuable contributions to this paper. This research was supported in part by the ONR grant NOOO14-87-k-0307 to the University of Maryland.

References

[1] S. K. Abd-El-Hafiz. *A Knowledge-Based Approach to Program Understanding.* PhD thesis, University of Maryland, College Park, MD 20742, May 1994.

[2] S. K. Abd-El-Hafiz and V. R. Basili. Documenting Programs Using a Library of Tree Structured Plans. In *Proceedings of the Conference on Software Maintenance*, pages 152–161, Montréal, Canada, September 27-30 1993.

[3] S. K. Abd-El-Hafiz, V. R. Basili, and G. Caldiera. Towards Automated Support for Extraction of Reusable Components. In *Proceedings of the Conference on Software Maintenance*, pages 212–219, Sorrento, Italy, October 15-17 1991.

[4] Z. Ammarguellat and W. L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, New York, June 20-22 1990.

[5] V. R. Basili and H. D. Mills. Understanding and Documenting Programs. *IEEE Trans. on Software Engineering*, SE-8(3):270–283, May 1982.

[6] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[7] M. T. Harandi and J. Q. Ning. Knowledge-Based Program Analysis. *IEEE Software*, 7(1):74–81, January 1990.

[8] J. Hartman. Understanding Natural Programs Using Proper Decomposition. In *Proceedings of the 13th International Conference on Software Engineering*, pages 62–73, Austin, Texas, May 13-16 1991.

[9] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner. Using Function Abstraction to Understand Program Behavior. *IEEE Software*, 7(1):55–63, January 1990.

[10] I. Hayes. *Specification Case Studies.* Prentice Hall International, 1987.

[11] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[12] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, 1991.

[13] W. L. Johnson and E. Soloway. PROUST: Knowledge-Based Program Understanding. *IEEE Trans. on Software Engineering*, SE-11(3):267–275, March 1985.

[14] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1990.

[15] S. Katz, C. A. Richter, and K.-S. The. Paris: A System for Reusing Partially Interpreted Schemas. In *Proceedings of the 9th International Conference on Software Engineering*, March 1987.

[16] J. C. Knight and D. M. Kienzle. Reuse of Specifications. In *Proceedings of the 5th Annual Workshop on Software Reuse*, 1992.

[17] S. Letovsky. Program Understanding with the Lambda Calculus. In *Proceedings of the 10th International Joint Conference on AI*, pages 512–514, August 1987.

[18] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, 1992.

[19] C. Rich and L. M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, 7(1):82–89, January 1990.

[20] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill, 1991.

[21] P. G. Selfridge, R. C. Waters, and E. J. Chikofsky. Challenges to the Field of Reverse Engineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 144–150, Baltimore, Maryland, May 21-23 1993.

[22] J. M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, pages 40–50, January 1989.

[23] J. M. Spivey. *The Z notation: A reference Manual.* Prentice Hall International, 1992.

[24] M. Ward, F. W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings of the Conference on Software Maintenance*, pages 307–315, Miami, Florida, October 1989.

[25] R. C. Waters. A Method for Analyzing Loop Programs. *IEEE Trans. on Software Engineering*, SE-5(3):237–247, May 1979.

[26] M. Weiser. Program Slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

[27] J. C. P. Woodcock. Structuring Specifications in Z. *Software Engineering Journal*, pages 51–66, January 1989.

A Notation

`var?`	value of `var` before an operation
`var!`	value of `var` after an operation
`i .. j`	subset of integers from `i` to `j` inclusive
`x in S`	`x` is member of the set `S`
`card S`	cardinality of the set `S`
`{x: S | p}`	set of values of `x` taken from `S` which satisfy `p`
`{x: S . e}`	set of values taken by the expression `e` as `x` takes values from `S`
`(forall x: p1: p2)`	for all `x` values which satisfy `p1`, `p2` is true
`(exists x: p1: p2)`	for some `x` value which satisfies `p1`, `p2` is true