CDA: A System for Understanding the Dynamic Properties of Data Processing Programs

William E. Howden
Guangming Shi

Software Engineering Laboratory
University of California at San Diego

Abstract

During software maintenance, it is of critical importance for maintenance staff to understand how a system works and when they make a change to part of the system, what effects this change will have on other parts of the system. It is our observation that much of the program understanding process revolves around dynamic properties such as the states, state sequences and state transition operations. Hence, it is necessary to support the understanding process when the staff is reasoning about what could take place when the program is in operation and what could happen if part of the code is modified. CDA is a system for understanding the dynamic properties of large data processing programs. It allows the user to incrementally document their understanding of a program as working hypotheses and abstract operations with CDA comments. It then verifies these hypotheses to determine if they are justified. The justified hypotheses and abstract operations form an incremental specification of the program. Over time, the documentation improves in quality and completeness as new comments are added to justify and check the assumptions underlying new changes made to the code.

1. Introduction

Software re-engineering is used to create abstract views of a system of the kind that might be produced during the requirements analysis and design phases of a development process. These documents help the programmer understand how a system works and are of critical importance during software maintenance. If they are not available or are no longer accurate, re-engineering can be used to reproduce them.

Most abstract system views are static in the sense that they describe relationships derived from the structure of the code. Examples include subroutine call trees and flow charts. We analyzed the reasoning process that a programmer engages in when understanding and maintaining a program and concluded that much of this revolves around dynamic properties, such as the states, state sequences and state transition operations. Static view documents are important for getting an overall understanding of a program's components and their relationships, but we found that it was also necessary to support the understanding process when the programmer is reasoning about what could take place when the program was in operation.

We observed that during the program reasoning process, when programmers are working in one part of the code, they often construct working hypotheses about effects produced or actions occurring in previous sections of code, in other modules, or in later sections of code. In addition, they may make a note that a given section of code performs some abstract operation, or establishes some object property or relationship. This kind of information is referred to as a fact. This kind of information is particularly important to the understanding and maintenance of large programs such as those found in data processing applications. We are building a program understanding tool which includes a documentation language that allows the user to record facts and hypotheses in the form of program comments. It also includes an analyzer that can be used to verify hypotheses. Although the technology can be used with different kinds of programs in different application areas, we chose the important, but often neglected area of large data processing systems. In order to allow the application of the analysis tool to existing systems, it has been designed for use with COBOL programs. The tool is called CDA (Cobol Dynamics Analyzer).

One of the biggest problems faced by maintenance staff is the worry that when they make a change to part of the code, the change will cause some unexpected failure in the system. When making changes, it is necessary to form hypotheses about the unaffected parts of the system which, if true, ensure that the change is safe. CDA allows the user to document these hypotheses with CDA comments. The analyzer then determines if they are justified. If they are not,
CDA tells the user where the problem lies. The user can then examine the relevant parts of the code. If it is possible to determine that the related parts of code are such that the hypotheses can be justified, the user can document the justifying facts in the CDA comments language so that future analyses will not report the hypotheses as being a problem.

In addition to its use in helping the user keep track of, and determine the consistency of his reasoning about the dynamics of a program, the CDA approach results in a well documented program. As a previously undocumented program is maintained, over time the documentation improves in quality and completeness as new comments are added to justify and check the assumptions underlying new changes made to the code.

CDA hypotheses can be thought of as "incremental specifications" and the CDA approach can also be used during program development. An initial set of requirements or specifications for a system will normally have to be refined and expanded during system development. During programming users will incrementally construct partial and local specifications that correspond to CDA hypotheses. CDA facts can then be put in which "paraphrase" sections of code at levels of abstraction, and in terms that are used in specification increments.

2. Dynamic Program Property Models

The program properties about which a user can reason using CDA are based on a program model, or point of view, called the Operator State Model (OSM). An OSM is a graphical state model with operations at the nodes and states along arcs. The OSM for a program is derived from a program and its fact comments.

An OSM model for a program is constructed by executing its Abstract Program Model (APM). The structure of an APM is derived from the program which it models. The operations in an APM are of two kinds. The first are derived from fact comments. For example, a comment may assert that at some point in the program an object X acquires the property "validated". The second kind of operation is derived from program source code. Of these there are two basic kinds. The first corresponds to an assignment statement. In this case, the value property of an object becomes modified. The second corresponds to a conditional statement branch. For example, if a branch in a program corresponds to the false outcome of the statement "if cond then S1 else S2", then this branch in the APM leads to an operation which establishes the property "¬cond".

The nodes in an OSM contain APM operations. The arcs in an OSM are labeled with the properties or relationships that are true along those arcs. These properties are established by the APM operations. Depending on the operations in the APM, an OSM may or may not be finite. The CDA hypothesis language can be used to describe expected properties of a program's OSM. Different kinds of hypothesis languages are appropriate for different kinds of OSMs. The verification of hypotheses may require different kinds of verifiers depending on the hypothesis language and the complexity of program OSMs.

3. QDA System

The first tool for program analysis and maintenance of the kind described above was the QDA (Quick Defect Analyzer) system [1,2]. This tool was built for analyzing programs written in CMS2 and assembler. It was built in order to analyze the code for an avionics system that had been in production use for a number of years. QDA is a simple system in which the APMs that are executed to create program OSMs contain no operators derived from source code. The only operators used are those derived from fact comments inserted by programming and maintenance staff. The APM consists of the original program structure plus inserted fact comments. The facts language is restricted to conjunctive propositions about simple objects. Typical embedded fact comments are of the form

reinitialized(ground-speed-radar).

When this comment appears as a fact it results in an APM operation establishing this property at the associated location in the program. If it were to appear as an hypothesis, it would be an assumption that would have to be proved. In the case of QDA, where the only facts known about a program are those documented by fact comments, this hypothesis would have to be supported by matching comments appearing on all paths leading to the hypothesis.

QDA has several additional important features. One is the capability of using object names as properties. This makes it possible to deal with program properties involving pointers and indirection. A distinction is also made between static and dynamic object properties. Static properties are established in data base modules, and are object properties which do not change, similar to types. Dynamic properties are established by fact comments appearing in the control flow, and can change during program execution. They look a little like "dynamic types" and were in earlier descriptions referred to as "flavors".
In addition to facilities for establishing properties of objects, QDA has facilities for “clearing” an object. This will result in the erasure of all dynamic properties established for the object. Static properties are not affected.

During the evolution of QDA, it was found to be important to include “rules”. These describe static relationships between object properties. For example, the rule

\[
\text{set(dataword.bit_2) \to \text{off(radar_control)}}
\]

indicates that whenever data word.bit_2 is set, radar_control is off. One of the primary uses of rules is to allow the description of levels of abstraction associated with a system under analysis. Often, operations will occur in a program level that correspond to more abstract operations occurring at the “documentation level”. Through the use of rules, users can simultaneously reason about both the concrete, implemented program properties, and corresponding more abstract properties.

In order to deal with large programs, QDA has facilities for modularizing an analysis. The QDA language contains input and output statements that can be used for summarizing expected input object properties and established output properties. These are used in the following way. Suppose that a procedure P were to be analyzed, whose input and output comments were IN and OUT. The QDA analyzer would begin by treating IN as an APM operation and would “execute” it to create a program state in which the IN properties were true. The paths through P, and the embedded fact comments would then be executed. The final states along paths would be verified against OUT. Alternatively, suppose that P were called from another routine R. Then at the point in the APM code of R where P is called, the state of R is compared with IN. It must logically imply IN. In addition, at the point of call of P, OUT is treated as a fact comment operation and is “executed” to change the current state of R at that point, reflecting the effects of the call to P.

QDA is simple enough that it is possible, for each routine in a program under analysis, to execute its APM and generate a complete OSM, against which all hypotheses can be checked. Implicit in the possibility is the circumstance of all OSM models being finite. This is because all QDA APM fact operations do nothing but establish a simple object property.

Even with a very simple fact language, simple propositional hypotheses, and a simple model checking verifier, QDA proved to be a very powerful analysis tool. In particular it was an ideal tool for detecting the occurrence of “decomposition errors”. These are errors that occur as a result of the necessity, in working in one part of a system, of making hypotheses about what the interacting other parts of the system will do.

4. Complex Program Understanding

When a programmer is reasoning about the properties of some local state in a program, he is concerned with verifying properties expected to be true in that state. QDA is limited to this kind of knowledge and reasoning. The QDA analyzer, by executing the program APM, constructs all possible program states. Each program state is associated with some program location and can be thought of as point in time knowledge that occurs along alternative possible program flows or executions. In the QDA approach, programmers reason about what can be true at some point in time.

In addition, programmers may want to reason about what can occur during some time interval. For example, a user may wish to hypothesize that if “data is valid” in some state, then “data is valid” holds until a state is reached in which “new_data is available”. This facility was not available in QDA, but it was found that in studying COBOL programs that it would be desirable to allow some form of interval temporal program reasoning like this. An example is given below that includes a description of such a situation. CDA includes facilities for both point and interval temporal reasoning.

In QDA, it was assumed that the abstract program model (APM) was fixed. There are several ways in which an APM could vary. One way is through the occurrence of subjunctive program knowledge. In this case the programmer is not reasoning directly about a program, its APM and the resulting OSM, but about what the OSM would look like if the APM were to contain some operation. This is like a program test in which the programmer reasons about what output state would occur if some input state were to be enforced. The establishment of an input state corresponds to the insertion of variable assignment operations at the appropriate point in the program.

QDA was limited to facts and hypotheses about object properties. In addition, it is also possible to express program knowledge in the form of relationships between objects. QDA rules were also limited to propositions. In a more complex system, it might be desirable to have rule schenata, or rules in which there are variables. There is also the consideration of different kinds of restrictions that might be put on rules. In QDA, complex theoretical and theorem proving issues were avoided by requiring that objects appearing on the left of a rule could not also appear on the right. More general rules could be allowed by relaxing this
restriction.

In order to establish a fact at some point in time, QDA users inserted a fact comment of the form

\[ \text{!initialized(radar)} \]

The exclamation mark identifies the comment as a fact, or APM operation, to be executed when constructing the program OSM. There are several possible ways in which program knowledge, and its corresponding APM operations might be automatically derived from code. The first is the inclusion of assignment operations that are derived from program assignment statements. An assignment of the form

\[ X := Y \]

is interpreted as being a statement of fact that “all of the dynamic properties possessed by \( Y \) at this point in time, including its value property, now become the dynamic properties of \( X \)”. In some cases, such as those involving indexed data structures and pointers, it may not be possible to accurately interpret an assignment statement without actually executing the program so that restrictions on automatic derivation of facts from assignments may be necessary. For example, automatic fact/operation identification could be restricted to assignments with a single variable on the right, assignments involving only integers or booleans, or assignments involving only simple arithmetic expressions.

In addition to assignments, we may also wish to automatically recognize APM facts associated with conditional branches, like those described in the earlier discussion of the source of APM fact comments. As in the case of assignments, we may wish to restrict such recognition to simple arithmetic relations or booleans.

The verifier in QDA with a few simple exceptions, was limited to simple pattern matching. More complex program understanding systems may have more complex verification capabilities. When assignment fact/operations are included, and arithmetic is allowed, it is no longer possible to simply generate a complete OSM for an APM as a prelude to determining the validity of a hypothesis. The OSM may not even be finite, for example. Even if it were, the sizes of the states, and the number of possible states may quickly become too large. One possibility is to use a “backwards symbolic evaluation” approach that will accommodate the use of assignments, relations, and integer arithmetic.

Regardless of the level of sophistication of the verifier used in a fact/hypothesis program understanding system, it is desirable to restrict its shortcomings to false negative errors. For example, in attempting to verify some hypothesis, we are much more concerned if our verification approach tells us that it is true, when it is not, than if it tells us that it is false, when it is true. This is because we can always fix up a false negative by inserting an appropriate fact/operation in the program to compensate for missing verifier deductive or fact recognition capabilities. In the following section, in our discussion of the basic features of CDA, we propose levels of verifier capabilities, all of which preserve this property.

One of the thorny issues that needs to be faced is disjunction. Disjunction in hypotheses is not a problem. Disjunction in facts is different. Suppose that at some point in a program an analyst wishes to state that at that point either \( P_1 \) or \( P_2 \) becomes true. What can this mean? We interpret this to mean that in some cases \( P_1 \) is true and in other cases \( P_2 \) is true. This implies that the OSM should be constructed having two states/nodes corresponding to these two alternatives. It is as if two little paths occurred in the source code, along which each of these held. This raises the possibility of the analyst wanting to state in an hypothesis associated with a program location not that \( P_1 \) or \( P_2 \) is true in the states at that location, but that in some states \( P_1 \) is true and in some \( P_2 \) is true, and in all states \( P_1 \) or \( P_2 \) is true. This will require some form of quantification. In QDA this was accomplished by changing the semantics of disjunction to conform to this interpretation. There were situations, however, in which traditional disjunction was desirable, so that in a more complex understanding system it would be desirable to deal with quantification explicitly.

5. User Interface

Both batch and interactive styles of analysis may be desirable. For example, a user may want to take a completely commented routine and analyze the embedded hypotheses, and to have this done without any form of interaction. Alternatively, an analyst/maintainer may need a more interactive style of usage in which it is possible to pose “what if” questions such as

- What if I change this program statement—will it affect the validity of any of the documented hypotheses?
- What if I insert this new fact, are all hypotheses still ok?

Regardless of the level of sophistication of the verifier
Is it the case that this proposed hypotheses is correct at this point in the program?

Both styles of usage will be available in the CDA system.

In addition to the kind of interaction mentioned above, a user may also wish to "browse". In QDA the user could identify a program statement and ask to see what the states (i.e. the properties of all objects) were at that point. This is only feasible in cases where a system operates by constructing a complete OSM. In systems where this is not done, a compromise might be to allow the user to ask to see all the properties of a selected set of objects in the OSM states identified by a particular program location.

6. CDA (Cobol Dynamics Analyzer)

While studying COBOL programs, it became apparent that a more sophisticated fact and hypothesis language than that used for QDA was going to be needed. In this section we outline some of the basic features of CDA. The following section contains the grammar for the CDA program understanding specifications language. This is followed by an example of its use.

Constructing an OSM analyzer involves tradeoffs between the expressibility of the hypothesis and fact languages, and the complexity of the associated OSM verifier. Since there is a research component to this project we decided to propose several levels of expressibility/complexity, and to build a prototype in such a way that it could be enhanced to become more powerful. These levels are discussed below. The first level is that of QDA. The next is the minimal CDA level, at which the prototype will be built. The following two levels will be implemented if desired, and resources permit.

CDA0 At this level, the only fact/operations are those that are documented by comments, not operations are derived automatically from code. Facts are limited to simple, unary predicates about objects, and to rules which relate one object/predicate to another. Hypotheses are limited to propositional expressions whose elements are also simple unary predicates. Disjunction in facts is implemented with a "path split" operator which sets up paths and states for each of the alternatives. Facilities are included for modularization in the form of input and output comments. There are no interval or subjunctive hypotheses. At this level it is possible to implement an hypothesis verification procedure which generates, for each subroutine or module, a complete OSM. This makes possible the construction of a browser like that built for QDA.

CDA1 This is the lowest level at which fact/operations in the APM are automatically derived from source code. At this level we identify simple, scalar assignment statements, in which there is only one item on the right. This is the only APM operation derived automatically from code, and is used to recognize the manipulation of Boolean flags. Although we do not recognize relations established by conditional branching statements, users are allowed to include relations in fact comments and hypotheses. This contrasts with CDAO in which only unary predicates are allowed. The verification mechanism, as in CDA0 is limited to pattern matching. As in the case of CDAO, the OSM is still finite, so it could be completely constructed, and verification done directly off of its states.

CDA2 At this level we allow automatic derivation of APM operation/facts from assignments where there are expressions on the right hand side that have plus and minus arithmetic operations. Everything else is the same as CDA1, including no automatic recognition of facts establishing relations corresponding to conditional statement branches. Note that in this case, the OSM may not be finite, so that hypothesis checking is going to be more complicated. The more automatic fact recognition we allow, the greater the danger of an exploding state space in the OSM. This implies that we will have to restrict the generation of the OSM, and the the execution of the corresponding APM operations, to those parts that are necessary to prove hypotheses. One possibility is to use some kind of backtracking symbolic evaluation approach. Another is to do a dependency analysis, to figure out which fact/operations to "execute". In CDA2 we do not do deduction, only pattern matching and simplification of constant expressions.

CDA3 This level will include automatic recognition of branching relations and will have a more complex theorem prover. The verifier will be constructed to do deductions in which an hypothesis is proved from a single, non-matching fact. The next level of complexity would be to allow deductions from a collection of facts (or previously established hypotheses) using, for example, knowledge about transitivity of arithmetic relations. Apart from this, everything else at this level will be the same as CDA4.

CDA4 At this level we introduce rule schema, i.e. rules which allow variables, and allow more complex deductions, like that mentioned in CDA3.

CDA+. Other things to be considered at higher levels include non-arithmetic fact/operations and subjunctive hypotheses.
7. CDA Knowledge Language

The following annotated grammar summarizes the CDA knowledge language (CDAKL). A technical report about CDA [3] contains a more complete description, including rules defining the ways in which facts and hypotheses can be embedded in source code programs.

Notation

In the following, \( t \) represents an item that can appear in the denoted class of expressions.

\( [t] \) selective, suppose \([a \mid b] \), \( t \) can be \( a \) or \( b \)

\( [t]\text{prop} \) expression in elements of \( t \), \( \text{and or} \Rightarrow \)
where \( a \Rightarrow b \Rightarrow \sim a \) or \( b \)

\( [t]\text{expr} \) expression in elements of \( t \), \( ( ) + - * / \)

\( [t]\text{andexpr} \) \( t \text{ and } t \text{2 and} \ldots \) where \( t 1, t 2, \ldots \) are elements of \( t \)

\( [t]\text{complist} \) \( t \text{1; } t \text{2; } \ldots \) where \( t1, t2, \ldots \) are elements of \( t \)

\( [t]\text{altest} \) \( t1; t2; \ldots \) where \( t1, t2, \ldots \) are elements of \( t \)

Facts

1. Object properties

This part of the language is used to describe properties of, and relationships between objects.

\( \text{object \_ property} \ ::= \text{predicate} \mid \text{relation} \)
\( \text{predicate} \ ::= \text{pred(object)} \)
\( \text{relation} \ ::= \text{rel([object, constant, expr]}\text{complist)} \)
\( \text{pred} ::= \text{user \_ defined} \)
\( \text{rel} ::= \text{predefined} \mid \text{user \_ defined} \)
\( \text{predefined} ::= <1 \mid l \mid l <1 \mid l >1 \mid l < > \)
\( \text{object} ::= \text{name \_ string} \)
\( \text{constant} ::= \text{number} \mid \text{Boolean} \mid \text{string} \)
\( \text{expr} ::= \text{[object, constant]}\text{expr} \)
\( \text{object \_ property \_ expr} ::= \text{[object \_ property]}\text{prop} \)

2. Operational facts

\( \text{operational \_ fact} ::= \)
\( !([\text{object \_ prop \_ expr, assignment, clear, operation \_ name}\text{complist}) \)
\( \text{assignment} ::= \text{object} ::= \text{expr} \)
\( \text{clear} ::= \text{clear(object)} \)

operation \_ name ::= \text{string}  

Facts are used to assert the occurrences of object properties and relationships. The clear operator is used to delete object properties and relationships. The user can also describe the occurrence of an abstract, undefined operation with the notation, "!operation".

3. Declarative facts

\( \text{declarative \_ fact} ::= !\text{declaration} \)
\( \text{declaration} ::= \text{object \_ property \_ expr} \Rightarrow \)
\( \text{[object \_ property]}\text{andexpr} \)

Declarations are used to define rules, like those described above for QDA.

4. Facts

\( \text{fact} ::= \text{operational \_ fact} \mid \text{declarative \_ fact} \)

Hypotheses

Hypotheses are working assumptions that must be proved.

1. Flow descriptions

In the hypothesis language, the executed program is described in terms of program flows. These are modeled as paths through the program state graph, and the hypotheses refer to arcs and nodes along these paths. The following notation is used to describe hypotheses about different kinds of program flows. Arcs correspond to program states and nodes to operations performed in the program. In the following specifications, a number of keywords are used.

Keyword | Explanation
---|---
AF | For all flows through a node of a program
EF | Exist a flow through a node of a program
I | Input arc of a node
O | Output arc of a node
IN | Incoming node of a node
ON | Outgoing node of a node
IP | Incoming path of a flow
OP | Outgoing path of a flow
V | Value of a data object
PC | Path condition

\( \text{flow \_ description} ::= \text{[AF\mid EF]}(\text{node \_ expr}) \)
\( \text{iarc} ::= \text{I(object \_ property \_ expr)} \)
\( \text{outc} ::= \text{O(object \_ property \_ expr)} \)
inode ::= IN(node-expr)
onode ::= ON(node-expr)
ipath ::= IP(node-expr since node-expr)
opath ::= OP(node-expr until node-expr)
node-expr ::= [label-expr, operational-fact, iarc, oarc, inode, onode, ipath, opath]prop
label-expr ::= label = location

In a flow description, AF is the default and can be omitted. For example, I(a(x)) is the same as AF(I(a(x))). In addition, if there is no I, O, IN, or ON prefix, I is assumed. Hence, a(x) is the same as AF(I(a(x))).

The following special notation is used to define restrictions on node-expr.

node-expr_1 ::= [iarc, oarc, inode, onode, ipath, opath]prop
node-expr_2 ::= label-expr => node-expr_1
node-expr_3 ::= [label-expr, operational-fact]prop

If ?node-expr_1 appears at a statement with label=L, it is a shorthand notation for something of form AF(node-expr_2), i.e.,
AF(label=L => node-expr_1)

In node-expr_3, only label-expr and operational-fact are allowed. This restricted node expression is used in global hypotheses (defined below) to make it easier to locate the nodes that satisfy conditions. In the prototype of CDA, node-expr in IP and OP expressions will be limited to node-expr_3.

2. Hypotheses

hypothesis ::= local_hypothesis | global_hypothesis
local_hypothesis ::= ?flow_description
global_hypothesis ::= ?if node-expr_3 then node-expr

The if_then clause in a global hypothesis is not equivalent to => in a propositional expression. It denotes a hypothesis over the whole program. It states that for all paths through an OSM, if a path satisfies the condition in node-expr_3 at some node, then node-expr must also be satisfied at the node. A local hypothesis is equivalent to a global hypothesis with an implied label expression in the if_part. The label expression states the current location at which the hypothesis is placed.

Procedure Interface Specifications

A called subroutine or procedure can be thought of as single OSM node. Its input/output behavior corresponds to possible pairs of input and output arcs for the node. We include a special notation in CDA to describe such procedure properties. It has the form
spec(input(object-property-expr),
     output(object-property-expr);
     ...).

This is interpreted to mean "there exists a flow through this node whose input and output arcs to this node have the specified properties 1 and 2, and there exists another path whose input and output arcs have the specified properties 3 and 4, and so on, and all paths satisfy at least one of these input/output descriptions". The node in this case is the paragraph or procedure called or performed from the node. We also allow a shorthand notation in cases such as the above where the two input expressions (1 and 3) are the same. In this case it is possible to write
spec(input(object-property-expr),
     output(object-property-expr);
     object_property_expr,
     ..).

Input and output interface specifications are formally defined in the following grammar.

proc_interface_spec ::= proc_spec | proc_hypothesis
proc_spec ::= spec([input_output_pair]aist)
proc_hypothesis ::= spec(proc-name)([input_output_pair]aist)
proc-name ::= name_string
input_output_pair ::= input_spec, output_spec
input_spec ::= nil | input([object_property_expr]aist)
output_spec ::= nil | output([object_property_expr]aist)
nil ::= 

Procedure interface specifications can be used to modularize program analysis as described above for QDA.

Queries

query ::= query_expr
query_expr ::= [V | PC] ([object]complist)

Queries can be used to determine the symbolic values (V) of a set of variables, or the conditions on paths (PC) restraining the values of a set of variables.

Specifications

specification ::= #(fact | hypothesis)
8. Examples

In the following examples we illustrate the use of CDA in understanding a particular program. The sample program was taken from a well-known COBOL textbook. The power of the CDA approach is indicated by the fact that during our use of CDA in attempting to understand the program, several errors were found. All but one was introduced when the program was transcribed, and is similar to errors that might be introduced during maintenance. One error is a real error, occurring in the original text. The full program, and all of the comments that had been embedded up to the point where the actual error was discovered, is included in [3].

In each of the following examples we describe some level of understanding that was developed using CDA. The examples illustrate how CDA can be used to build up, incrementally, a computer aided understanding of the program and at the same time record this knowledge for future understanding and analysis of the program during maintenance.

The program in the examples is designed to validate the records in a file. It generates a report describing errors in record fields. It also generates a file containing all valid records.

**Example 1** The first thing that was done in understanding the program was to look at how input was handled. Examination of the source code revealed the statement

```
PERFORM 800-READ-SALES-TRANSACTION-RECORD.
```

This paragraph, which is duplicated below, was then examined.

```
800-READ-SALES-TRANSACTION-RECORD
READ SALES-TRANSACTION-FILE
INTO ST-SALES-TRANSACTION-RECORD
NOT AT END ADD 1 TO WS-RECORDS-READ
END-READ.
```

After reading this, we understood that a record count was being kept, and that it was updated here. In addition, it appeared that this record count must be initialized somewhere else. This piece of understanding was documented by creating a corresponding input hypothesis. In addition we added an output statement and a fact related to the updating of this count, producing the following documented code segment.

```
800-READ-SALES-TRANSACTION-RECORD
#spec(input(initialized(WS-RECORDS-READ)))
#spec(output(updated(WS-RECORDS-READ)))
READ SALES-TRANSACTION-FILE
INTO ST-SALES-TRANSACTION-RECORD
NOT AT END ADD 1 TO WS-RECORDS-READ
#updated(WS-RECORDS-READ)#
END-READ.
```

If we were to run a basic CDA analyzer at this point it would tell us that there are calls of this paragraph at locations where it is not known that the input hypothesis is justified (A more sophisticated CDA might be able to get this information from the code). In the basic CDA case, we would start to examine the code to see if everything is correct. In this case we would notice that there is a call

```
PERFORM 100-INITIALIZE-VARIABLE-FIELDS
```

just before the call on the 800 paragraph. Analysis of this paragraph reveals the statement

```
INITIALIZE WS-PAGE-COUNT,
WS-TOTAL-ACCUMULATORS.
```

Examination of the data division reveals that WS-TOTAL-ACCUMULATORS contains WS-RECORDS-READ as a subfield. If we insert the fact

```
#initialized(WS-RECORDS-READ)#
```

at this point, the input hypothesis for paragraph 800 will now be fully justified, and a CDA analyzer would not report an error. If we had a higher level analyzer that, when seeing an INITIALIZE statement would automatically generate an "initialized" property for both the compound record and all its sub-records, then it would not be necessary for the user to insert the above fact.

Further analysis of paragraph 800 prompted the consideration of end of file conditions. The paragraph performs a special action when it is not the end of file, but does nothing for an end of file. This was puzzling and indicated that a deeper understanding of this part of the code was necessary. Immediately following the call to paragraph 800, the following statement appears

```
PERFORM 200-VAL-SALES-TRANSACTION.
```

Examination of this paragraph revealed that it expected a switch END-OF-FILE to have been set before entry. This was documented with the hypothesis
#?set(END-OF-FILE)#.

If the CDA analyzer were to be run at this point it would report that this hypothesis was unjustified. We could partially fix this by inserting at the NOT AT END location in paragraph 800 the fact

#!set(END-OF-FILE)#.

If the CDA analyzer were to be run at this point, there would still be a problem reported since the hypothesis at paragraph 200 would not be justified in the case where an end of file had been reached. We then realized that there was an error in the program. Examination of a previous version of the source revealed that the following statement had inadvertently been left out

AT END MOVE "YES" TO
WS-END-OF-FILE-SWITCH.

This was put back into the code and the following justifying fact inserted

#!set(END-OF-FILE)#.

The CDA analyzer would now not report an error.

In the above example, we have not included details of some of the additional features that would be desirable in a higher level CDA for Cobol, that are related to specific features of the language. For example, the END-OF-FILE switch is actually a condition variable for WS-END-OF-FILE-SWITCH. It would be desirable to have these automatically linked together. In addition, we might like to have automatic setting of the condition "set" whenever a value is assigned to a variable.

Example 2 Next we decided to try to understand how the output was generated by the program. There appeared to be several kinds of output, including an error report. At the end of the program there are three paragraphs

870-PRINT-REPORT-HEADINGS
880-WRITE-REPORT-TOP-LINE
890-WRITE-REPORT-LINE.

Examination of the calling locations for these paragraphs revealed a call to paragraph 870 preceded by the statement

IF WS-LINES-USED IS NOT LESS THAN
WS-LINES-PER-PAGE

This raised the topic of understanding how the program kept track of lines per page used. In addition, since it appeared to print possibly several lines of invalid transaction error messages for each record, we thought we had better understand this also. We began by documenting paragraph 890 as follows

890-WRITE-REPORT-LINE

#?lessthan(WS-LINES-USED,
WS-LINES-PER-PAGE)#
WRITE AUDIT-ERROR-LINE
AFTER ADVANCING WS-LINE-SPACING
ADD WS-LINE-SPACING TO WS-LINES-USED.

Running of a CDA analyzer would reveal that there were paths leading to the embedded hypothesis along which the hypothesis is not justified. Along some of these paths, WS-LINES-USED is initialized to zero and incremented by a small integer. If we used a CDA with simple symbolic evaluation capabilities, the hypothesis error message for those paths would automatically be eliminated (i.e. no error would be indicated). Along other paths the situation was more complex. Working backwards along one path, through a hierarchy of paragraph calls, up to a certain point it was possible to see that a simple fact of the form

#?lessthan(WS-LINES-USED + k,
WS-LINES-PER-PAGE)#

where k is some small integer, could be safely stated. It was obvious that the fact was true and simple symbolic evaluation from that point forward would justify the hypothesis in paragraph 890. This fact was posed as an input specification for a paragraph in the calling hierarchy. This has the effect that it can be used as a fact to justify hypotheses appearing inside the paragraph or in called paragraphs. It also has the effect that it would be used as a hypothesis that had to be justified in calling paragraphs.

At a certain point in the analysis, we were reading the code inside a paragraph called 410-IDENTIFY-ERROR-TYPE, where there were multiple calls to paragraphs that were used to validate individual fields in a record. The necessary input hypothesis to this paragraph would be of the form above, but with a large value for k, larger than 30. If this hypothesis were inserted, and the CAD analyzer run, it would direct us to the calling location in the next level up in the hierarchy. At this point it was obvious that there was nothing in the code to indicate that a necessary justifying fact could be stated, and that the code was in error. It appears that the programmer failed to realize that many error report lines could be produced
for a single record, so that it was not logically feasible to include the end of page check at the place where new records are read in.

Example 3 This example illustrates the need for interval temporal hypotheses. After examining the input and output parts of the program, it was decided to look at the parts which do the actual processing of records. Embedded in this part of the code is a paragraph called 410-IDENTIFY-ERROR-TYPE. This paragraph looks at a flag INVALID-SALES-TRANS. If the flag is true, then spaces are moved into the record image area for the error report output line. Apparently this is done so that when a record has more than one invalid field, only the first error output line contains both the record image and the invalid field message. For other fields, only the invalid field message is printed. In order to confirm that our understanding of the program was correct in this point, we wanted to be able to make an hypothesis to this effect. What needed to be said was

"Each time a new record is read in, and it is not the end of the file, then when paragraph 410 is reached for the first time after that, the flag INVALID-SALES-TRANS should be set to false (so that spaces are not moved into the record image part of the invalid record output buffer). Each time 410 is reached for the record, the flag should be true."

This temporal hypothesis is stated in CDA as follows

```cda
#? if !1410-IDENTIFY-ERROR-TYPE
then ((IP(-!(1410-IDENTIFY-ERROR-TYPE)
since !800-READ-SALES-TRANS)
=> ~TRUE(INVALID-SALES-TRANS))
and
(IP(!1410-IDENTIFY-ERROR-TYPE
since !800-READ-SALES-TRANS)
=>TRUE(INVALID-SALES-TRANS)))#
```

This is read as follows

"For all paths through the OSM if you are at a node where the operation !1410-IDENTIFY-ERROR-TYPE is performed, then there are two kinds of sub-paths which can lead you to this node. On the first type of sub-path leading to this node, no operation !1410-IDENTIFY-ERROR-TYPE is performed since an operation !800-READ-SALES-TRANS was performed. On such a path, the input to the !140 node is ~TRUE(INVALID-SALES-TRANS). On the second type of sub-path, the operation !1410-IDENTIFY-ERROR-TYPE does occur since operation !800-READ-SALES-TRANS was performed. On paths having this property, the input to the !1410 node is TRUE(INVALID-SALES-TRANS)."

By setting up artificial flags at key places, and possible other artifacts, the above sequencing information could be expressed in other ways. But it is our opinion that some form of temporal logic specification, like that above, is the most natural.

9. Conclusion

In order to fully understand a program, it is necessary to reason about its dynamic properties. Included in this process is the ability to form and prove hypotheses, and to document dynamic program knowledge not present in the code. We have developed a language, and its accompanying analysis tool that supports this process.

The CDA system allows users to document their knowledge of a piece of code in a way that is important to the maintenance process. It makes it possible for maintenance staff to learn important abstractions used in the original design of the program, and to test with hypotheses their understanding of its dynamics.

The development of the CDA language evolved through many versions. It required the development of the OSM model and the examination of COBOL programs in order to design a language that is sufficiently expressible but does not result in intractable theorem proving problems. The expected effectiveness of the language is based on both previous experience with the QDA system, and with experiments like those partially described in the examples, in which it was applied to actual programs.

References


Acknowledgement

This research was carried out as part of the SMA research project. We are grateful to the Fujitsu company for the support and Professor Isao Miyamoto of the University of Hawaii for helpful advice and suggestions.