Verifying the design of a Cobol system using Cognac

Andy Kellens
Kris De Schutter
Theo D’Hondt
Context

- Industrial case study with inno.com
- Fairly large Cobol system (500 KLoc)
- Verify design
Context

• Industrial case study with inno.com

• Fairly large Cobol system (500 KLoc)

• Verify design
Context

- Industrial case study with inno.com
- Fairly large Cobol system (500 KLoc)
- Verify design
Context

• Industrial case study with inno.com
• Fairly large Cobol system (500 KLoc)
• Verify design
Cognac

• Tool to verify design wrt. Cobol code
• Built on top of IntensiVE
• Technically:
  • Island-based parser
  • Library of logic predicates
  • Basic static analyses
IntensiVE in one slide

• Tool to express/verify structural regularities

• Basic concepts:
  • Group entities in *intensional views*
  • Defined by means of a logic meta program (query)
  • Impose *constraints* over these intensional views
Island-based parsing

- Cobol:
  - Lots of dialects
  - Lots of language constructs

- Island-based parser: extract only necessary data

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
     ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
    FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
     01 LOGFILE-RECORD PIC X(2048).
  WORKING-STORAGE SECTION.
    01 LOGFILE-STATUS PIC 9 VALUE ZERO.
     88 LOGFILE-IS-OPEN VALUE 1.
  LINKAGE SECTION.
    01 LOGFILE-ENTRY.
     05 LOGFILE-VERB PIC X(12).
     05 LOGFILE-NAME PIC X(32).
     05 LOGFILE-ENTRY PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
  IF NOT LOGFILE-IS-OPEN
    OPEN EXTEND LOGFILE
    SET LOGFILE-IS-OPEN TO TRUE.
    MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
    WRITE LOGFILE-RECORD.
    GOBACK.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
     ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
    FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
     01 LOGFILE-RECORD PIC X(2048).
  WORKING-STORAGE SECTION.
    01 LOGFILE-STATUS PIC 9 VALUE ZERO.
     88 LOGFILE-IS-OPEN VALUE 1.
  LINKAGE SECTION.
    01 LOGFILE-ENTRY.
     05 LOGFILE-VERB PIC X(12).
     05 LOGFILE-NAME PIC X(32).
     05 LOGFILE-ENTRY PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
  IF NOT LOGFILE-IS-OPEN
    OPEN EXTEND LOGFILE
    SET LOGFILE-IS-OPEN TO TRUE.
    MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
    WRITE LOGFILE-RECORD.
    GOBACK.
```
## Logic predicates

### Structural reification

<table>
<thead>
<tr>
<th>Programs</th>
<th>Source code relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>?program isProgram</td>
<td>Calling relationships</td>
</tr>
<tr>
<td>?program isProgramWithIdentifier: ?identifier</td>
<td>?call callWithTarget: ?string</td>
</tr>
<tr>
<td><strong>Sections</strong></td>
<td></td>
</tr>
<tr>
<td>?section isSectionWithName: ?name</td>
<td>?call transitivelyCallsProgram: ?program</td>
</tr>
<tr>
<td>?section isSectionInProgram: ?program</td>
<td>?call callUsingField: ?field</td>
</tr>
<tr>
<td>?section isSectionWithName: ?name inProgram: ?program</td>
<td>?program programUsingField: ?field</td>
</tr>
<tr>
<td><strong>Paragraphs</strong></td>
<td></td>
</tr>
<tr>
<td>?paragraph isParagraph</td>
<td>?section sectionPerformsSection: ?callee</td>
</tr>
<tr>
<td>?paragraph isParagraphInSection: ?section</td>
<td>Embedded SQL</td>
</tr>
<tr>
<td><strong>Statements</strong></td>
<td></td>
</tr>
<tr>
<td>?call isCallStatementInSection: ?section</td>
<td>?exec execStatementUsesTable: ?table</td>
</tr>
<tr>
<td>?perform isPerformStatementInParagraph: ?par</td>
<td>?exec execStatementWritesToTable: ?table</td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>Move information</td>
</tr>
<tr>
<td>?field isFieldInProgram: ?program</td>
<td>?field fieldIsSenderOfMove: ?move</td>
</tr>
<tr>
<td>?field isFieldInLinkageSection: ?linkage</td>
<td>?field fieldIsReceiverOfMove: ?move</td>
</tr>
<tr>
<td>?linkage isLinkageSectionInProgram: ?program</td>
<td>Field aliasing</td>
</tr>
<tr>
<td></td>
<td>?field mayAliasWith: ?aliasField</td>
</tr>
<tr>
<td></td>
<td>?field mayTransitivelyAliasWith: ?aliasField</td>
</tr>
</tbody>
</table>
Basic static analyses

• Call resolution
  CALL ‘ProgramX’ using PARAM
  CALL LOGGING using PARAM

  01 ROUTINE-NAMES.
    05 LOGGING PIC X(08) VALUE ‘ProgramX’

• Field aliasing
  CALL ‘PROGRAM5’ using FIELD1.
  MOVE FIELD1 to TEXT1.
Example 1: Section layers

• Control-flow in programs encoded in the section names

• Top-level sections call 2nd level, and so on.

• Indicated by beginning letter of section name

• Regularity:
  • A section can only perform sections that have section name with the same or later beginning letter
Section layers

Sections with callees

?section isSectionInProgram: ?program,
?section sectionPerformsSection: ?callee
## Section layers

<table>
<thead>
<tr>
<th>Program</th>
<th>Section</th>
<th>Call</th>
<th>Source</th>
<th>Call</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMKTL385</td>
<td>B0204-ADDRESS-OF-NOTARY</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTC372</td>
<td>B0002-COMM-CHECK</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTL378</td>
<td>B0031-DEC-P-ACTN</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTL339</td>
<td>B0011-LOG-ABEND</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTL352</td>
<td>B0028-GET-AMOUNT-DETAILS</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTL355</td>
<td>B0011-LOG-ABEND</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTR838</td>
<td>B0100-MQS-PROCESS-LAYOUT-03</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTR832</td>
<td>B0023-GET-BEGIN-START</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTR837</td>
<td>B0023-GET-CREDIT-START-DATE</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMKTL330</td>
<td>B0202-CHECK-CREDIT-VERSION</td>
<td>E9000-INIT-ABEND-VELDEN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section layers

Sections with callees

?section isSectionInProgram: ?program,
?section sectionPerformsSection: ?callee
Section layers

Sections with callees

\[\forall \text{?entity} \in \text{Sections with callees} : \text{?entity.section isSectionWithName: ?callerName,} \]
\[\text{?entity.callee isSectionWithName: ?calleeName,} \]
\[[(\text{?callerName at: 1}) \leq (\text{?calleeName at: 1})] \]
Figure 2. Feedback provided by our tool suite

ease this correspondence, a copybook is used that contains the data definition and that should be included in the link-age section of the called program as well as in the calling program. We express the above design rule by means of a binary intensional relation. First, we create an intensional view named Called programs with as intension:

\[ ?\text{program} \text{programCalls:} ?\text{call}, \] 
\[ ?\text{call} \text{callCallsProgram:} ?\text{calledProgram} \]

The above query retrieves all pairs of \( ?\text{program} \) and \( ?\text{calledProgram} \) between which there exists a possible calling relationship.

Note that the above predicates make use of the call resolution analysis that was discussed in Section 3.

Next, we create a second intensional view Program with copybook that groups all programs together with the copybook that defines their linkage section data definitions. The intension for this view is:

\[ ?\text{program} \text{programWithCopyStatement:} ?\text{copy}, \] 
\[ ?\text{copy} \text{copyStatementInLinkageSection}, \] 
\[ ?\text{copy} \text{copyStatementIncludesCopybook:} ?\text{copybook} \]

This intension consists of three parts. The first condition retrieves all the copy statements in Cobol programs. Copy statements are used to include a particular copybook. In the second condition, this set of copy statements is limited to those that are contained within the linkage section of the program. Finally, the third condition binds the logic variable \( ?\text{copybook} \) to the actual copybook that was included in the linkage section.

Using the above two intensional views, we can now express the design rule as the following binary intensional relation:

\[ \forall ?\text{caller} \in \text{Called programs}: \exists ?\text{corresponding} \in \text{Program with copybook}: \] 
\[ ?\text{caller}.\text{program equals:} ?\text{corresponding}.\text{program}, \] 
\[ ?\text{caller}.\text{program programIncludesCopybook:} ?\text{corresponding.copybook} \]

The above relation verifies that for all called programs, the corresponding copybook in the linkage section of the callee is included by the caller.

When verifying the above intensional relation, our tool reported on 42 violations of the documented design rule. By manually inspecting each of these violations, we saw that they resulted from the usage of a utility library which the above design rule is not applicable to. In this library, one big copybook is used that includes the data definitions of all fields that are used in the library. Rather than including the specific corresponding copybook for a called program from the library, all clients of the library included this big, more general copybook. Since these invocations are exceptions rather than violations, we documented them as such by declaring them an exception to the intensional relation.

Database modularity

The case study we investigated is designed in a component-oriented fashion. In the system, the various components consist of a top-level program that serves as the component's interface, along with a number of programs to which this top-level program delegates particular requests. Also associated with each component is a set of database tables that contain the persistent data which the module is responsible for. In order not to break this modularity, only programs from within one particular module are allowed writing access to the tables associated with that module. All other programs need to retrieve and manipulate data via the interface program of that module. Preferably also, the number of programs within a module that are allowed to write to the associated tables is limited.

In order to verify this design rule, we opted to use a more pragmatic approach in which we use a visualisation as a means to provide the original designers of the system with feedback concerning the use of database tables in the current implementation. First, we created an intensional view...
Example 2: Copybook-linkage

CALL ‘Program10’ using PARAM
Example 2: Copybook-linkage

CALL ‘Program10’ using PARAM

???Same data definition???
Example 2: Copybook-linkage

CALL ‘Program10’ using PARAM

???Same data definition???
Example 2: Copybook-linkage

CALL ‘Program10’ using PARAM

???Same data definition???

COPY ‘COPYBOOK1’

COPY ‘COPYBOOK1’

COPYBOOK1

01 PARAM

.....

Regularity: A caller should include the copybook used in the linkage section of the callee
Called programs

?program programCallsProgram: ?calledProgram

Program with copybook

?program programWithCopyStatement: ?copy,
?copy copyStatementInLinkageSection,
?copy copyStatementIncludesCopybook: ?copybook
Called programs

∀ ?program programCallsProgram: ?calledProgram

∀ ?caller ∈ Called programs :
  ∃ ?corresponding ∈ Program with copybook :
    ?caller.calledProgram equals: ?corresponding.program,
    ?caller.program programIncludesCopybook: ?corresponding.copybook

Program with copybook

∃ ?program programWithCopyStatement: ?copy,
  ?copy copyStatementInLinkageSection,
  ?copy copyStatementIncludesCopybook: ?copybook
Copybook - linkage

∀ ∈ Called programs : ∃ ∈ Program with copybook : caller.calledProgram equals: corresponding.program, caller.program programIncludesCopybook: corresponding.copybook
Example 3: DB modularity

- Group of programs = module
- Each module contains a set of db tables
  - Only particular programs within a module can read/write db directly
- Rest of programs need to go via interface
DB Modularity

Programs writing to tables

?stat isExecStatementInProgram:?program,
?stat isExecSQLStatement,
?stat execSQLStatementWritesToTable:?table
Programs writing to tables

?stat isExecStatementInProgram:?program,
?stat isExecSQLStatement,
?stat execSQLStatementWritesToTable:?table
Some numbers

<table>
<thead>
<tr>
<th>Cognac operation:</th>
<th>Time (in seconds):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing</td>
<td>377s</td>
</tr>
<tr>
<td>Running the analyses</td>
<td>155s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design rule verification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Section layering</td>
<td>20s</td>
</tr>
<tr>
<td>Copybook - linkage correspondence</td>
<td>23s</td>
</tr>
<tr>
<td>Database modularity</td>
<td>1s</td>
</tr>
</tbody>
</table>

Memory consumption: 73 Mb
Future work

• Encode entire design
• More guidelines
• Rational rose documents
• Convert to intensional views & constraints
• Extend set of predicates
• Implement more analyses
Verifying the design of a Cobol system using Cognac

http://prog.vub.ac.be/
http://www.intensional.be