Cognac:
a framework for documenting
and verifying the design of Cobol systems

Andy Kellens
Kris De Schutter
Theo D’Hondt

Luc Jorissen
Bart Van Passel
Context

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

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

- Industrial case study with inno.com
- Average-sized Cobol system (500 KLoc)
- Verify design

1. outsourcing

2. iterations
Context

- Industrial case study with inno.com
- Average-sized Cobol system (500 KLoc)
- Verify design
Challenges

• Analyzing Cobol code
  • Parsing
  • More than structural information
• Performance
• Extensibility
  • Various kinds of design rules
Cognac

• Declarative meta programming 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 design rules
- Written in Smalltalk

Basic concepts:

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

- Cobol:
  - Lots of dialects
  - Lots of language constructs
- Island-based parser: extract only necessary data

```cobol
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-DATE 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.
```

```cobol
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-DATE 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

<table>
<thead>
<tr>
<th>Structural reification</th>
<th>Source code relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programs</strong></td>
<td><strong>Calling relationships</strong></td>
</tr>
<tr>
<td>?program isProgram</td>
<td>?call callWithTarget: ?string</td>
</tr>
<tr>
<td>?program isProgramWithIdentifier: ?identifier</td>
<td></td>
</tr>
<tr>
<td>?program programIncludesCopybook: ?copybook</td>
<td></td>
</tr>
<tr>
<td>?call callWithTarget: ?string</td>
<td></td>
</tr>
<tr>
<td>?call callsProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?call programUsingField: ?field</td>
<td></td>
</tr>
<tr>
<td>?call transitivelyCallsProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?call callUsingField: ?field</td>
<td></td>
</tr>
<tr>
<td>?program programIncludesCopybook: ?copybook</td>
<td></td>
</tr>
<tr>
<td><strong>Sections</strong></td>
<td>?section sectionPerformsSection: ?callee</td>
</tr>
<tr>
<td>?section isSectionWithName: ?name</td>
<td></td>
</tr>
<tr>
<td>?section sectionPerformsSection: ?callee</td>
<td></td>
</tr>
<tr>
<td>?section isSectionInProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?section sectionPerformsParagraph: ?par</td>
<td></td>
</tr>
<tr>
<td>?section isSectionWithName: ?name inProgram: ?program</td>
<td></td>
</tr>
<tr>
<td><strong>Paragraphs</strong></td>
<td><strong>Embedded SQL</strong></td>
</tr>
<tr>
<td>?paragraph isParagraph</td>
<td>?exec isExecStatementInProgram: ?program</td>
</tr>
<tr>
<td>?paragraph isParagraphInProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?exec isExecStatementUsesTable: ?table</td>
<td></td>
</tr>
<tr>
<td>?paragraph isParagraphInSection: ?section</td>
<td></td>
</tr>
<tr>
<td>?exec isExecStatementWritesToTable: ?table</td>
<td></td>
</tr>
<tr>
<td><strong>Statements</strong></td>
<td><strong>Move information</strong></td>
</tr>
<tr>
<td>?move isMoveStatementInProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?call isCallStatementInSection: ?section</td>
<td></td>
</tr>
<tr>
<td>?field movesTo: ?field</td>
<td></td>
</tr>
<tr>
<td>?call isCallStatementInSection: ?section</td>
<td></td>
</tr>
<tr>
<td>?perform isPerformStatementInParagraph: ?par</td>
<td></td>
</tr>
<tr>
<td><strong>Field aliasing</strong></td>
<td><strong>Field aliasing</strong></td>
</tr>
<tr>
<td>?field mayAliasWith: ?aliasField</td>
<td></td>
</tr>
<tr>
<td>?field isFieldInProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?field movesTo: ?field</td>
<td></td>
</tr>
<tr>
<td>?field isMoveStatementInProgram: ?program</td>
<td></td>
</tr>
<tr>
<td>?field mayTransitivelyAliasWith: ?aliasField</td>
<td></td>
</tr>
<tr>
<td>?field isFieldInLinkageSection: ?linkage</td>
<td></td>
</tr>
</tbody>
</table>

3.3. Extracting static information

Although a Cobol parse tree offers a wealth of information, certain kinds of information are not directly accessible from such parse trees. We implemented the following static analyses in order to complement the information retrieved from the parse tree:

- **Call resolution**
  - One interesting source of information in Cobol programs are the various calling relations between Cobol programs. In order to retrieve this information from the source code, we need to analyse the `CALL` statements. For example, the statement `CALL 'Example' USING CALL'PARAM` indicates a call to the program named `Example` using the data field `CALL'PARAM` as an argument. While the first argument of the `CALL` in the simplest case is a string indicating the program name that gets called, it can also be a data field (e.g. `CALL PROG'SUB USING CALL'PARAM`). In this case, it is not certain which program will get called, since the value of `PROG'SUB` can be altered at runtime. Cognac implements a simple static analysis that, for `CALL` statements where the callee is stored in a data field, identifies possible programs by looking at data field initialisers (e.g. the field `PROG'SUB` might be initialised to the value `'Example'`) and the allocation of string literals to fields.

- **Field aliasing**
  - While the call resolution we discussed above allows us to give a coarse-grained approximation of the control flow in a Cobol system, Cognac also implements a field aliasing algorithm that offers a light-weight analysis of the data fields within the application. This analysis will collect for each data field in the system a set of other data fields which may possible alias with that particular field. For example, the usage of a `MOVE THIS TO THAT` statement, which moves the contents of one data field (`THIS`) to another data field (`THAT`) introduces an alias between the two involved fields. Similarly, the arguments of a call of a program result in that two different data fields are possibly pointing to the same piece of memory. Note that we take a conservative approach to calculating the aliases of a particular field: if a field is in the alias set of another field, this does not necessarily mean that at runtime they will get used for the same data.
Basic static analyses (I)

• Call resolution
  CALL ‘ProgramX’ using PARAM
Basic static analyses (I)

- Call resolution
  CALL 'ProgramX' using PARAM
  'PMKTR351'
  'QMMCL498'
  'PMMQ304'
  ...

Friday 27 March 2009
Basic static analyses (1)

- Call resolution

```
CALL 'ProgramX' using PARAM
CALL LOGGING using PARAM

01 ROUTINE-NAMES.
   05 LOGGING PIC X(08) VALUE 'ProgramX'
```

...
Basic static analyses (2)

- Field aliasing

```plaintext
01 WORKING-DATA.
   05 FIELD1 PIC 9(6).
   05 TEXT1   PIC X(6).
```
Basic static analyses (2)

• Field aliasing

CALL ‘PROGRAM5’ using FIELD1.
MOVE FIELD1 to TEXT1.

01 WORKING-DATA.
  05 FIELD1  PIC 9(6).
  05 TEXT1   PIC X(6).
Basic static analyses (2)

- Field aliasing

CALL 'PROGRAM5' using FIELD1.
MOVE FIELD1 to TEXT1.

01 WORKING-DATA.
  05 FIELD1 PIC 9(6).
  05 TEXT1 PIC X(6).
Example design rules
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
- Design rule:
  - A section can only perform sections that have section name with the same or later beginning letter
Sections with callees

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

Sections with callees

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

Sections with callees

∀ ?entity ∈ Sections with callees :
  ?entity.section isSectionWithName: ?callerName,
  ?entity.callee isSectionWithName: ?calleeName,
  [(?callerName at: 1) <= (?calleeName at: 1)]
Section layers

If a section performs another section, this section should have a first letter that comes later in the alphabet than the calling section.

Programs and callees:

- Program PMKTR607, Section D1201-FETCH-TC0001, Callee Section B9999-ARR-LIMIT-END-ABEND
- Program PMKTR635, Section D1301-FETCH-TC0011, Callee Section B1302-GET-AMT-DET-WITHOUT-AL
- Program PMKTR484, Section F001-LAY02-OST-0019-UTILITY, Callee Section E9001-LOG-ABEND
- Program PMKTR889, Section D0501-GET-KT-HEADER, Callee Section B0502-GET-AND-LOCK-CREDIT
- Program PMKTRM77, Section D0009-LOG-ATCH-END-INST, Callee Section B0007-CALL-ATCH-UTILITY
- Program PMKTR635, Section D0672-FETCH-TC0020, Callee Section B0702-GET-LST-REGSTRD-ADJ
- Program PMKTR315, Section P0000-GET-CREDIT-NO, Callee Section E9006-INIT-ABEND-VELDEN
- Program PMKTR278, Section D0008-LOGICAL-END-ATCH-INST, Callee Section B0007-CALL-ATCH-UTILITY

Not in domain

- Program PMKTR607, Section D1201-FETCH-TC0001, Callee Section B9999-ARR-LIMIT-END-ABEND
- Program PMKTR635, Section D1301-FETCH-TC0011, Callee Section B1302-GET-AMT-DET-WITHOUT-AL
- Program PMKTR484, Section F001-LAY02-OST-0019-UTILITY, Callee Section E9001-LOG-ABEND
- Program PMKTR889, Section D0501-GET-KT-HEADER, Callee Section B0502-GET-AND-LOCK-CREDIT
- Program PMKTRM77, Section D0009-LOG-ATCH-END-INST, Callee Section B0007-CALL-ATCH-UTILITY
- Program PMKTR635, Section D0672-FETCH-TC0020, Callee Section B0702-GET-LST-REGSTRD-ADJ
- Program PMKTR315, Section P0000-GET-CREDIT-NO, Callee Section E9006-INIT-ABEND-VELDEN
- Program PMKTR278, Section D0008-LOGICAL-END-ATCH-INST, Callee Section B0007-CALL-ATCH-UTILITY
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’

Design rule: A caller should include the copybook used in the linkage section of the callee
Copybook -linkage

**Called programs**

`?program programCallsProgram: ?calledProgram`

**Program with copybook**

Called programs

∀ ?program programCallsProgram: ?calledProgram

Program with copybook

∃ ?program programWithCopyStatement: ?copy,
    ?copy copyStatementInLinkageSection,
    ?copy copyStatementIncludesCopybook: ?copybook

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

∀

If a program calls another program, it should also include the corresponding copybook of that program.

Program with copybook

∀

Friday 27 March 2009
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

Programs writing to tables
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>
<tr>
<td>Design rule verification</td>
<td></td>
</tr>
<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
Cognac:
a framework for documenting
and verifying the design of Cobol systems

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