In the Proceedings of the Tenth Knowledge-Based Software Engineering Conference, Boston, Massachusetts, pages 79 - 87. IEEE Computer Society Press, November 1995.

An Interface between Different Software Development Environments*

Greger Lindén and A. Inkeri Verkamo

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
e-mail: {linden, verkamo}@cs.helsinki.fi

Abstract

Information sharing between two well-defined development environments demands data transformation between the tools or the underlying repositories of the environments. We give a solution to the problem of making tools in different environments cooperate without modifying the tools or their environments. In building an interface between a KBS environment and a selected CASE tool we have used a transformation generator called ALCHEMIST. This has reduced the work of respecifying the environments for each transformation. We have encountered several problems in the transformation that we have solved, including, e.g., incompatible views of the tools, multiple views, and different ways of representing graphical information. Additional problems can be addressed through user interaction before, during, or after the transformation.

1 Introduction

Even though the literature handling CASE tools has for some time recommended using integrated project support environments (IPSEs) in systematic large scale software production (see, e.g., [4]) the practitioners on the field are mostly involved with much less advanced practical problems in tool cooperation. Only recently, a study of the state of the practice in CASE integration described that it is difficult to find industry experience of long time operational use of integrated CASE tools [11]. Among the most important problems reported in the study is the inadequacy of the interfaces for actual integration, which is contrary to claims often made by CASE vendors.

Several ways have been suggested to ease CASE tool interfacing. Among these are common data presentation frameworks such as PCTE (adopted by ECMA [6]) and IRDS (approved by ISO and ANSI [7]), and tool connection using message passing [12]. So far, none of these standards or methods have been unanimously accepted by tool vendors. Within tool integration, tool developers concentrate on making their own tools work together, using one of the many de-facto industry standards, while cooperation of tools from different origin is considered less important or even commercially disadvantageous. However, the users' best interest is in truly open software development environments that include cross-environmental tool cooperation, whether it is based on common standardization activity or market-driven industry standards [17].

In the present situation software developers who want to make tools of different origin cooperate are more or less left on their own in building the interface. In a previous paper, we have discussed the specific types of problems that must be solved to build such an interface [16]. In this paper, we describe our experience in implementing an interface between the conceptual modeling tools of a development environment for building knowledge based systems and the analysis and design tools of a commercial CASE environment. This work was done in a research project, VITAL, for building a full scale workbench for KBS development [13].

The paper has the following structure. In section 2, we present the two environments that are to be interfaced. In section 3, we describe the requirements for the interface and the solution strategy, based on a transformation generator. The following sections describe how we solved the most essential practical problems in the interface, such as handling different source and target views (section 4), information collection (section 5), and user interaction during the transformation (section 6). We close by a brief conclusion of our experiences and some suggestions for future improvements.

2 The environments

2.1 The source environment

The source environment is the VITAL workbench, a development environment for knowledge based systems, produced in the VITAL research project, [13]. We are specifically looking at the conceptual modeling tool of VITAL, the OCML (Operationalizable Conceptual Modeling Language) editor [5]. The OCML editor includes capabilities for constructing diagrams for domain data, domain knowledge, and problem solving behavior. The OCML editor is the VITAL tool for the conceptual modeling phase and even part of the design

*This research was supported by the Technology Development Center (TEKES) and in part by the Commission of the European Communities under ESPRIT-II project 5365 VITAL. This paper reflects the opinions of the authors and not necessarily those of the VITAL consortium.

phase of the KBS development. The OCML editor, as well as the entire VITAL workbench, runs in Sun/Unix workstations and uses a commercial Lisp environment.

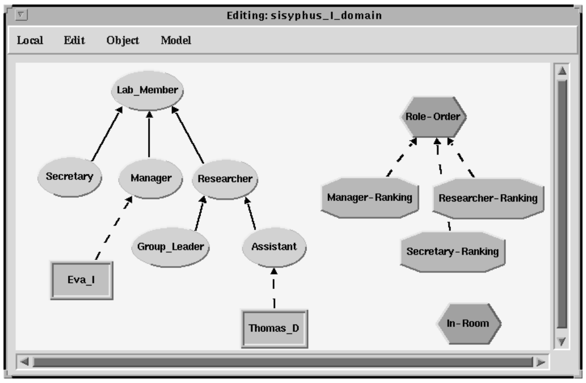


Figure 1: Example of an OCML domain diagram.

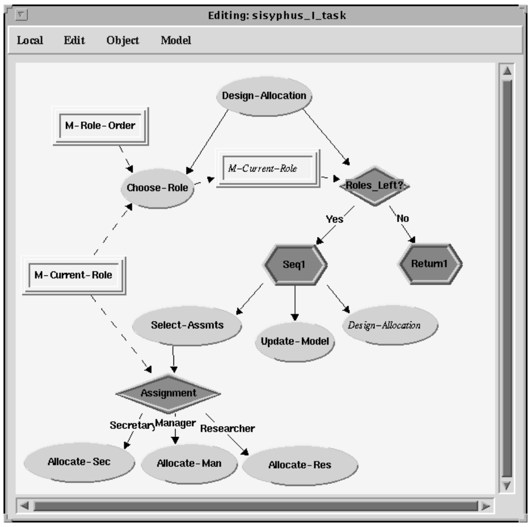


Figure 2: Example of an OCML task diagram.

The OCML diagrams for domain data and knowledge representation are *domain diagrams*, and the diagrams for problem solving behavior are *task diagrams* (Figures 1 and 2). The domain diagrams consist of graphical elements for *concepts*, *relations*, *concept instances* and *relation instances*. The task diagrams contain graphical elements for *subtasks* and *roles*, as well as elements for defining the control structure, such as *sequencing* of subtasks, *choice* between subtasks, or *return* from a subtask. Part of the information of the elements (e.g., the *properties* of concepts) is not presented in the diagram, but it can be manipulated through the menus of the OCML editor.

There is no global repository for the objects processed in the VITAL workbench. The persistent representation of OCML is Lisp (Figure 3). OCML is operationalizable in the sense that the persistent representation of the elements may include Lisp code representing low-level control, knowledge and data. The persistent representation consists of two main parts. The *graphical part* (starting with `(dale:defgraphics` in Figure 3) contains graphical information of all elements. The *declaration part* (starting with `(def-relation role-order` in Figure 3) contains a definition for each element, and describes its connections to other elements (`kldesign-instance-of-link` in Figure 3), and its properties (`:slots` in Figure 3). As much as possible of the information in the persistent representation should be included in the transfer.

```
;;; -*- Mode: Lisp, Design Domain
Layer; Package: DL -*-

(dale:defgraphics dale:coordinates
  dale::instance
  (thomas_d (278 304 358 354)
  eva_i (42 244 122 294))
dale::concept (lab_member (119 9 199 59)...)

(def-relation role-order NIL ((:schema role-
  order ?role ?order) (:inter-diag-alias)))
(def-concept lab_member NIL ((:allocated?)
  (:group . yqt)(:works-with)(:hacker)(:smoker)
  (:project) (:slots project smoker works-with
  hacker group allocated?)(:inter-diag-alias)))
(def-instance eva_i ((|:::| manager dale::
  kldesign-instance-of-link dale::concept))
  ((:smoker)(:hacker) (:works-with thomas_d
  monika_x ulrike_u) (:slots smoker hacker
  works-with))) ...
```

Figure 3: Example of the persistent representation (Lisp) of an OCML domain diagram.

2.2 The target environment

The target environment is a commercial CASE environment, FOUNDATION[1] by Andersen Consulting [1], which is a typical example of advanced commercial CASE environments. FOUNDATION runs in PCs and uses the OS/2 operating system. FOUNDATION uses a repository that is built on the commercial database system DB2/2[2]. Since FOUNDATION supports object import and export [3] building an interface does not require changing the environment itself. The FOUNDATION toolset for the analysis and design phases, FOUNDATION Design, contains several tools [2], including diagramming tools for *entity-relationship diagrams* (ERD), *data flow diagrams* (DFD), and *procedure diagrams* (PD) (Figures 4, 5, and 6). Since the repository is common to all phases of software development it also contains objects that are used in later phases of development, such as *records* and graphical user interface objects.

The repository uses an entity-relationship model. Diagrams and their elements are presented in the

[1]FOUNDATION is a registered trademark of Andersen Consulting.

[2]DB2/2 is a registered trademark of IBM.

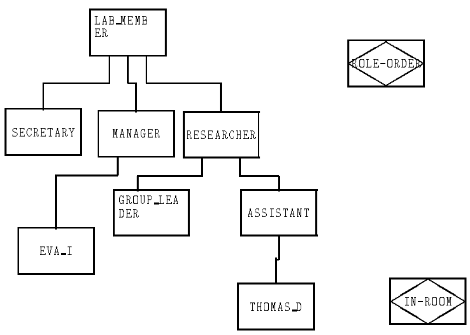


Figure 4: Example of a FOUNDATION Design ERD.

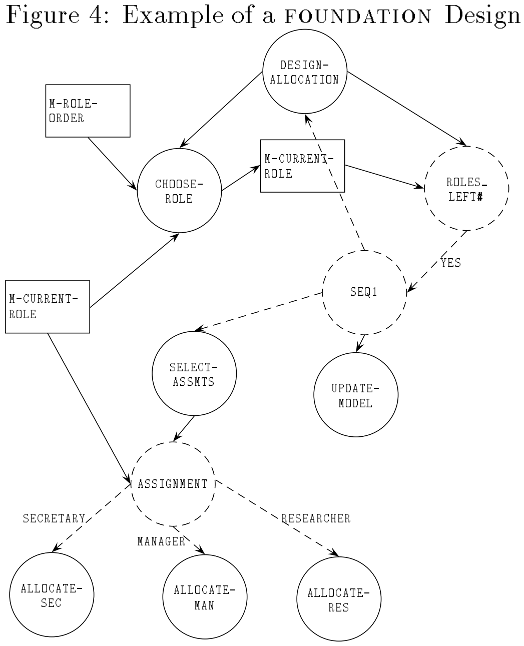


Figure 5: Example of a FOUNDATION Design DFD.

repository tables as entities (Figure 7). Information such as "Entity E belongs to diagram X" is presented in the repository as relationships between the corresponding entities. For representing graphical information, FOUNDATION Design uses two strategies. The graphical information of the ERDs is presented in the relationships between the elements of the ERD and the ERD itself. The graphical information of the DFDs and PDs is included in separate binary coded graphical files (*lod* files, see Figure 9) which are referenced by the corresponding repository entities (e.g., the repository entity for a DFD contains the name of the *lod* file defining the graphical structure of the DFD). Using a different strategy in tools of the same environment is by no means an ideal solution but it is the practical result of a changing environment. CASE tools, like any other software, are developed in slow steps. Although the look-and-feel of the toolset is fairly unified, the structures beneath originate from versions constructed by different developers.

Importing and exporting repository information is based on using import/export files or *exp* files (Fig-

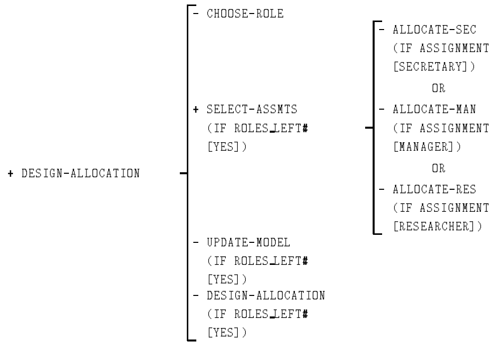


Figure 6: Example of a FOUNDATION Design PD.

ure 8). In addition to the *exp* files, the *lod* files referenced by the *exp* files must be available for producing DFDs and PDs. The import routine reads the *exp* files and the *lod* files and stores the corresponding elements in the repository.

3 Strategy for building the interface

3.1 Design

A natural basis for tool cooperation between OCML and FOUNDATION Design is to have several transformations to support the different aspects of conceptual modeling. For the domain data and knowledge representation purposes, domain diagrams are transformed to ERDs since they contain elements that correspond fairly well. For the task diagrams, there is no single representation in FOUNDATION Design that could contain all the information involved. To describe the data flow contents of the task diagram, the task diagrams are transformed to DFDs. To describe the control information of the task diagram, the task diagrams are transformed to PDs. In both cases, only part of the source information can be presented by the target representation.

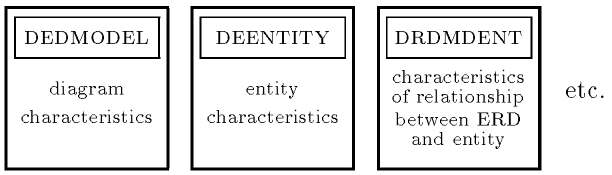


Figure 7: Examples of FOUNDATION repository tables.

In the domain diagram, concepts and instances are transformed into ERD entities. We decided to include the instances also in the target diagrams since the OCML representation relies heavily on also describing instances, relevant information that would be lost if the instances were discarded in the transformation. Links between concepts are transformed into ERD *is-a* relationships, while links between concepts and instances are transformed into ERD *is-instance-of* relationships. All relations are transformed into ERD associative entities. This seems to be the best solution, while OCML relations do not correspond very well to

ERD relationship types. OCML relation instances are discarded altogether.

```
HEADER              ...
0                   ...
2DEDMODEL         2sisyphus
2DEENTITY         2ROLE-ORDER
...
2DERELSHP         2ROLE-ORDER: 1
...
2DEATTR           2LAB_MEMBER:PROJECT
...
3DEDMODELDEENTITY4sisyphus
 ... 0000100000 ACR  THOMAS_D
...
3DEDMODELDERELSHP4sisyphus
 ... 0000100000 ACR  sisyphus:is a 1
...
```

Figure 8: Example of an *exp* file of an ERD.

In task diagrams, OCML tasks are transformed into DFD processes and/or PD procedures. Choice and sequence tasks are transformed into DFD control processes. In PDs they are represented by control structures. Return tasks are discarded in both DFDs and PDs. OCML roles are transformed into DFD external entities, and discarded in PDs. The control links in OCML diagrams (between tasks) are transformed into control data flows, and OCML data links into data data flows in DFDs. In PDs the control is shown through the hierarchical structure of the PD. No data flows or data storages are presented in PDs.

```
F   F   O   R   M   X   X   X   U 050 049 052
 032 032 032 032   J   B   B 032 032 032 032 032
 ...
   D   A   T   A 032   F   L   O   W 032   D   I
   A   G   R   A   M 032 045 064
 ...
 001 000   D   F   S   Y   M 032 032 032 046   D
   W   B 028 000
 ...
 001 032 001 020 000 003 000 000 000 000 000 000
   D   R   D   O   C   D   O   C 126 052 045 059
 071 034 103 059 000 000 000 000 000 000 000 000
 000 000 000 000 000 000 000 000 000 000 000 000
 000 000 000 000 018 000   D   E   S   I   G   N
 045   A   L   L   O   C   A   T   I   O   N 013
 ...
```

Figure 9: Example of a *lod* file of a DFD.

For corresponding diagrams, see Figures 1 – 2 and 4 – 6. The domain diagram of Figure 1 corresponds to the ERD of Figure 4, and the task diagram of Figure 2 to the DFD and the PD of Figures 5 and 6. The figures present diagrams for the Sisyphus I room allocation problem [9] which is used in the examples throughout the paper. A solution of the Sisyphus I problem using OCML has been described in [10].

3.2 Implementation

We have built the three interfaces using the ALCHEMIST transformation generator [14]. ALCHEMIST requires describing the source and target representations as context-free grammars and defining a mapping (correspondency rules) between the productions of the grammars. The specification of the mapping is based on the notion of TT-grammars [8]. From the transformation specification ALCHEMIST generates transformation code which is compiled into an executable transformation program.

The transformation process consists of three modules: preprocessing, transformation, and postprocessing (Figure 10). The preprocessing module edits the source file by rearranging the elements to simplify the grammatical structure and by removing unnecessary lines such as comments. The transformation module builds a tree representation using the preprocessed source file, transforms the source tree into a target tree according to the mapping, and finally writes the result into a target file. The transformation module also produces a *log file* containing performance statistics and additional information about the transformation. Finally, the postprocessing module edits the results of the transformation by doing simple character replacements (necessary because of the different character sets used by the environments) and, if necessary, by separating the *exp* file and the *lod* file.

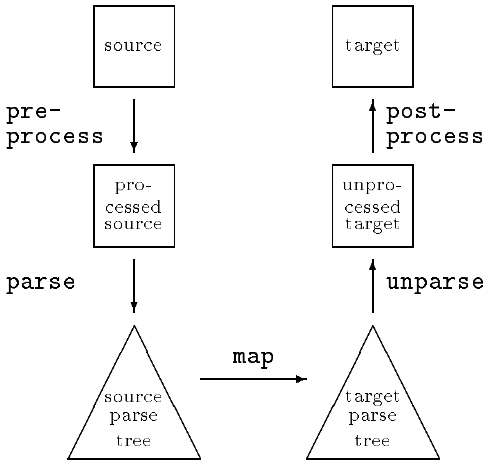


Figure 10: The transformation process.

The ALCHEMIST approach of separating the definition of the two grammars and the mapping between them, produces elements that are potentially reusable when building new interfaces. When building CASE interfaces, it is advantageous to use this approach instead of hand-made end-to-end transformations. In our situation, instead of one interface we needed a group of interfaces between the source and target environments. In all three interfaces, the source representations are either the same or fairly similar, and the target representations also have much in common (e.g., the overall structure of the *exp* file). In two cases, the same source grammar can be used in building both interfaces. As the grammars for presentations of a common environment are often fairly similar, the effort needed to define the grammars is reduced.

If a standard representation is available, this approach supports generating interfaces that use the standard as an intermediate representation. This permits easy building of interfaces between any pair of tools having an interface with the standard representation.

The most time consuming task in implementing the transformations is to deduce the grammars of the source and target representations. In neither case did we have complete grammars from the beginning. This is information that is usually not provided by the tool developer. Both OCML and FOUNDATION grammars were constructed partly based on documentation, partly by testing the tools and examining their persistent representations. When the grammars are correct, building the mapping is fairly easy. Preliminary results can be reached within a day, a complete transformation of this size within a week. The OCML domain and task grammars contain about 85 and 80 productions, respectively. The FOUNDATION *exp* and *lod* grammars contain about 300 and 130 productions, respectively. The mappings between the grammars are based on some 30 rules designating corresponding parts in the source and target representations.

The test runs were performed on a Sun IPX workstation with Solaris 2.2 and a main memory of 48 MB. Typical time requirements for the transformation range from 5 to 20 seconds depending on the size of the diagrams. Transforming the example domain diagram of Figure 1 into the *exp* file of the corresponding ERD takes about 8 seconds. Transforming the task diagram of Figure 2 into the *exp* and *lod* files of the corresponding DFD or PD takes about 10 seconds. The transformation time is in any case much less than the time required to import the target files into the FOUNDATION environment; on our FOUNDATION running on a Pentium PC under OS/2 with 40 MB of main memory, the import takes from 30 seconds to 2-3 minutes depending on the size of the diagrams.

The biggest OCML files we used are about 30 kB corresponding to a diagram of some 250 symbols (concepts/relations or tasks/roles). The corresponding target files can be up to about 160 kB. The example domain and task files were 2.7 kB and 3.9 kB, respectively. The *exp* file of the corresponding ERD was 47 kB, the *exp* file and the *lod* file of the corresponding DFD were 35 kB and 4.2 kB, respectively, and the *exp* file and the *lod* file of the corresponding PD were 2.8 kB and 2.4 kB, respectively.

Most of the transformation processing is done in main memory. Due to complex source and target grammars, the internal parse trees of the source and target files contain about ten times as many nodes as there tend to be symbols in the files. Every node demands about 600 bytes. A large source file of 30 kB corresponds to about 260 lines and 2700 internal source nodes (1.6 MB); a large target file of 160 kB corresponds to about 800 target lines and 20000 target nodes (12 MB).

The following sections handle the most essential practical problems that we encountered in building the three interfaces.

4 Source and target views of the structure

Much of the problems that we faced in building the interface originate from the different views that the source and target representations have in the structure to be presented. In this section we describe these differences and their implications.

4.1 Handling different source and target views

The OCML persistent representation is based on describing the *logical* structure of the diagram. There is a definition for each element describing the characteristics of that element (such as its properties, see Figure 11). A link connecting two elements is defined within the definition of the element where the link starts. There is no information about the link elsewhere, e.g., in the element where the link ends.

```
(def-instance eva_i ((|:::| manager dale::
 kldesign-instance-of-link dale::concept))
 ((:smoker)(:hacker) (:works-with thomas_d
 monika_x ulrike_u) (:slots smoker hacker
  works-with)))
```

Figure 11: Properties and links of an OCML element.

The graphical structure of the diagram is described in the graphical part containing only the coordinates of the elements (see Figure 12). The placement of the links is generated automatically by the tool.

```
(dale:defgraphics dale:coordinates
 dale::instance (thomas_d (278 304 358 354)
 eva_i (42 244 122 294))
 dale::concept (lab_member (119 9 199 59)
 group_leader (144 203 224 253) manager (128
 118 208 168) researcher (219 119 299 169)...)
```

Figure 12: Graphical structure of an OCML diagram.

The FOUNDATION repository contains an entity for each of the elements of the diagram, including also the links. In addition, information about the connections between the elements is presented as repository relationships. In some cases (ERDs) information about the graphical placement of each element is also presented as a repository element. In other cases (DFDs and PDs) the graphical information is placed in the corresponding *lod* file which is also part of the repository. Thus, the repository representation is based also on describing the *physical* structure of the diagram.

As a result of the difference in source and target views, all information required to produce the target representation is not immediately available when it is needed. E.g., the need to produce an ERD relationship is observed, when handling a concept that has a link to another concept. This leads to producing the necessary ERD relationships. As the graphical placement of the ERD relationship depends on both of the elements being connected it cannot be computed without knowledge of the element where the link ends.

4.2 Combining or producing multiple views

In some cases, the source and target representations separate or combine information in a different way, thus requiring either the division of information into several elements, or the combination of information from several source elements in order to obtain one target element.

At the highest level this happens when entire files have to be divided or combined. Transforming a task diagram into a DFD (or a PD) requires producing an *exp* file and a *lod* file. ALCHEMIST produces transformations from single file to single file. Our solution is to produce a file with a separator between the two parts, which the postprocessing module then divides into two files. The other direction (combining information from several files into one file) did not occur in our situation, but a similar approach (file catenation with a separator in between) would be applicable. E.g., if we wanted to perform a reverse transformation to combine the information in a DFD and a PD to obtain a task diagram we would have to deal with this problem.

At a lower level, the need to produce several elements from one element occurs, e.g., when we have an OCML concept to be transformed: we need to produce an entity in the repository and also several repository relationships to describe the placement of the entity and its connections to other elements. ALCHEMIST supports having multiple results by allowing us to specify explicitly all parts that must be produced from a single source object.

Combining several elements to produce one element is needed, e.g., because the declarations of the elements and the graphical information about them are separated in the OCML persistent representation (see Figure 11 and 12). To obtain the repository relationship of the placement of an entity we must combine information from these two parts of the source.

Another example of combining entire elements is handling an *alias* (or duplicate): a task diagram can contain several aliases of a task, while FOUNDATION Design does not allow process aliases. This requires combining the information of all aliases (e.g., all links starting or ending at an alias) to the original (see Figure 13). In practice, this often leads to cluttered target diagrams as the main reason for using aliases is to clarify the diagram.

A third example of the need to combine information from several elements occurs when a task diagram with control elements is transformed into a PD (see Figure 14). Choice and sequence tasks are not transformed into separate PD items. Instead, all subtasks of a choice or a sequence become subtasks of the supertask (of the suppressed task) in the PD, with the choice involved presented as conditional structures (`IF` and `ELSE`). In this case, the transformation is necessarily non-local and complex. Sometimes it is necessary to apply semantic checking of the entire source file when producing a single target object.

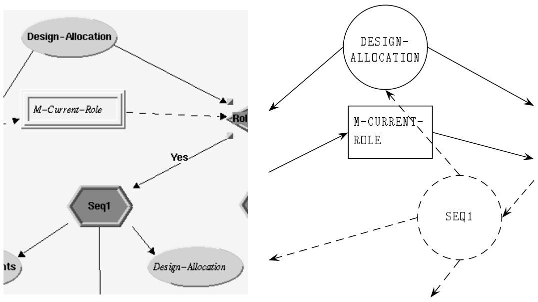


Figure 13: Transforming a task diagram with aliases.

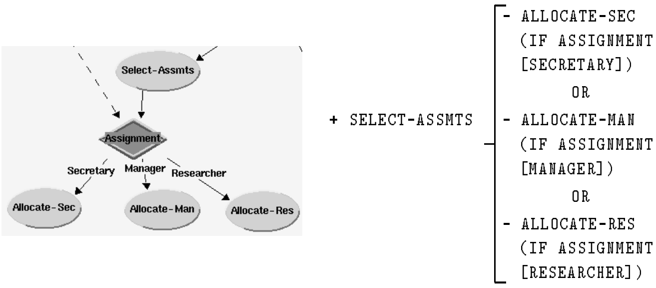


Figure 14: OCML choice tasks transformed into PD processes.

4.3 Graphical information

One of the points where the source and target environments most often differ is in representing graphical information of the diagram. This ranges from different coordinate systems (e.g., coordinates starting from upper left or lower left corner, or having different unit sizes) to a completely different view of how to describe the structure of the diagram (e.g., placement defined by coordinates or relative to other parts of the structure).

As an example of an easy modification, producing the graphical information of an element involves merely simple arithmetic computations. The OCML persistent representation gives upper left and lower right coordinates for each element, while FOUNDATION Design requires upper left coordinates, element width and height for boxes and midpoint coordinates and a size code for circles; computing these is straightforward. The unit sizes of the environments are somewhat different but that can be handled by a constant factor. On the other hand, the default fonts of the two environments are also different which means that the text that fits in the default size box in one environment may not do so in the other.

A more complex graphical information modification is required for the placement of the data flows in a DFD. As we mentioned in Section 4.1, the persistent representation of the task diagram contains no graphical information for the links (see Figure 3). To compute the coordinates for a data flow, the coordinates of the two elements connected by the data flow are first obtained. The starting point and the ending point of the data flow are placed on the midpoints of some side of the corresponding element, depending on the relative positions of the elements (see Figure 15). This computation is necessary to avoid the data flows crossing the elements themselves. Avoiding crossing other elements as well would require even more complex heuristics (finding the placement of all other elements, etc.) which we did not consider worth while: the user can easily rearrange the data flows in the target environment if he does not consider their placement pleasing.

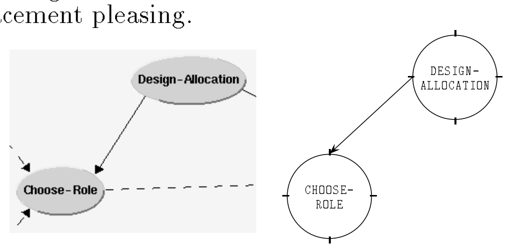
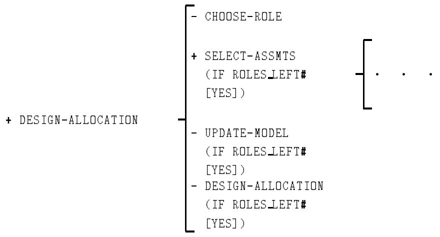


Figure 15: Possible starting and ending points for the data flow.

The most difficult computation for graphical information is required for producing the PD, which is a formatted textual representation of the control structure of the diagram. The graphical structure of the PD is defined using *zone* (logical column) and *line* information (see Figure 16). Thus, to know the placement of a certain item (task) we need to know its level in the task hierarchy starting from the root node (= its zone) and how many lines are required by the preceding nodes on the same level (= its line). The hierarchical level of the node in the task diagram is not always the same as its zone in the PD since some of the control elements in the task diagram are presented directly by the sequencing and conditional structures of the PD. The number of lines required by each node is affected by the number of its children and the length of each text. Therefore, computing the coordinates of each element requires parsing the entire task diagram in a hierarchical order. The OCML persistent representation structure does not support such parsing in any way.

5 Information collection during the transformation

During the transformation, information is collected from the source representation, to be used during the transformation and provided to the user for possible evaluation and later use. Some global information not explicit in the source representation must also be computed (e.g., the number of elements in the diagrams). An essential structure for collecting and arranging information is the *symbol table* that contains a record for each logical element of the source diagram. For further use, e.g., a reverse transformation, the information in the symbol table is placed in a *log file*. In addition,



Element	elem type	node zone	start line	end line	Conditional text
DESIGN-ALLOCATION	pro	1	1	14	
CHOOSE-ROLE	pro	2	2	2	
SELECT-ASSMTS	ifp	2	3	7	(IF ROLES_LEFT# [YES])
UPDATE-MODEL	ifp	2	8	10	(IF ROLES_LEFT# [YES])
DESIGN-ALLOCATION	ifp	2	11	13	(IF ROLES_LEFT# [YES])
...					

Figure 16: Part of a PD and information required for placement.

the log file contains also performance statistics of the transformation.

5.1 Global information

The *lod* files contain some global information about the diagram that must be collected during the transformation. An example is the total number of graphical objects in a DFD or a PD. For PDs we also need the number of text blocks, and the number of text lines occupied by the diagram. Some of these numbers, e.g., the number of graphical objects, can be computed during the transformation. Others, e.g., the number of text lines, can only be computed after the transformation. This means that producing the values requires several passes over the source and target representations.

5.2 Symbol table

The symbol table is an auxiliary data structure used for keeping track of the symbol names occurring in the OCML diagram and their corresponding names in the FOUNDATION diagram. The symbol table contains all OCML object names (source names), their types and their names in the FOUNDATION diagram (target names). Usually, the target names are the same as the source names; occasionally, however, they must be modified if there are differences in what names the two environments use or accept.

The source and target environments may have different naming strategies, both with respect to which elements must have (unique) names and to what the acceptable structure of a name is. In our example, OCML does not require a name for the diagram itself nor any of the links, while FOUNDATION requires all elements to have unique names. Since OCML does not use a repository, OCML names only need to be unique within one diagram but the FOUNDATION names must be unique within all elements of the same type in the repository. The transformation therefore needs to generate (unique) names for those elements that need to have one.

The character sets used for names in the two environments also have some differences, e.g., a question mark (?) is accepted in an OCML name but not in a FOUNDATION name. This means that some characters must be replaced in the transformation. Likewise, too long names have to be truncated and clashing names have to be modified.

The symbol table also keeps track of the coordinates of the OCML elements. The coordinates are used for computing the target coordinates of the elements and the starting and ending points for data flows. For all OCML links, the symbol table also describes which elements the link connects. When producing a PD, the symbol table is needed for reconstructing the diagram, i.e., for removing unnecessary objects in the control structure of the task diagrams and moving their children to the appropriate place in the PD structure (see Figure 16).

5.3 Reverse transformation

To support a reverse transformation the log file must contain several kinds of information. It must contain all *name changes* made by the transformation, including the source name and the target name of the element. It must also describe all elements that are *omitted* in the transformation, and include their source name and type, their properties and their links. Sometimes different source types are transformed into similar target types and therefore the log file must contain information about all *source element types*.

A typical need for reverse transformation involves a file previously imported from environment A to environment B and (slightly) modified in B: we now want to have the original (in environment A) modified in a similar way. This requires the reverse transformation to read the modified file (or files) from B and the log file of the previous transformation to produce a "new original" in environment A. The reverse transformation is non-trivial since the modified diagram may contain updates that are inconsistent with the original source file and its log file.

6 User interaction

In our prototype solution, we used a completely automated transformation with reasonable default values whenever needed. However, an automatic transformation has some limitations. Some differences of the source and target representations are not always adequately solved by using default values. In this section we discuss some features that would improve if the user could interact with the transformation.

There are several types of user involvement that may be useful. Firstly, the user could set new default values to be used for the transformation, or a subset of the transformation. Secondly, he could interact during the transformation answering questions and choosing alternatives. Finally, the user could modify the target diagram in the target environment. Although not implemented in our prototype solution, our approach permits implementing user interaction. We point out some features where the transformations could be made more general through user interaction.

6.1 Naming of elements

As described in Section 5.2, some names need to be generated or modified during the transformation. Our default strategy names the diagram using the (user given) name of the target file. The links (ERD relationships and DFD data flows) are given internal names that include the diagram name, a part describing the type of link, and a running index, to ensure the uniqueness of the names within the repository (e.g., `egram:is a 5` for an *is-a* link in an ERD). As for character set differences, we use a rude replacement strategy and replace all unacceptable characters with a default character (crosshatch, `#`).

All differences in naming strategies are likely ones that the user might want to act on. To implement this, the user should be permitted to examine and modify the source and target names in the symbol table, before the actual transformation takes place. Having user acceptable names is particularly important if the names, once they are given, cannot be changed in the target environment.

6.2 Details of the transformation

We observed some situations where the user might want to omit some part of the source diagram or to modify the transformation strategy.

When transforming a domain diagram into an ERD it is not obvious whether an instance should produce any element in the ERD. This depends on the way the ERD is going to be used: the instances could be represented as entities in the ERD, or completely omitted. This could easily be implemented as a user option.

A similar situation occurs when transforming the roles of a task diagram. A role could correspond either to an external entity (if it is a role external to the task) or to a data store (if it contains task internal information). In our prototype, the roles are always transformed to external entities. In a user assisted transformation, the user could make the decision separately for each role.

Another example are the attribute types. Attributes of concepts in OCML do not have types while FOUNDATION entity attributes always have a type. As a default, the attribute type is always set to "string" as the correct type is impossible to automatically conclude from the attribute values (e.g., `141195` could be a string, an integer, a real, or a date). The user might want to define the types explicitly during the transformation.

6.3 Placement of elements

In the transformation, all elements of the diagram are placed as close to their original position as possible. However, the two environments have a somewhat differing coordinate system, and their boxes, ovals, and circles have different default sizes. The links in an OCML diagram are automatically placed by the editor (connecting the elements with a straight line), while the transformation computes the link placement in a FOUNDATION Design diagram. The source and target environments may prefer different kinds of links (e.g., OCML produces straight links, see Figure 1, but FOUNDATION uses polylines in ERDs, see Figure 4). Therefore the target diagram may not be completely similar to the source.

Since some of the elements of the source diagram are omitted in the transformation (e.g., relation instances of a domain diagram) and others may be combined (e.g., task aliases) the target diagram may differ even more from the source. For these reasons, the user may want to modify the placement of the elements in the diagram. In principle, this could be implemented as a user option. However, the graphics of the target diagram are best judged and modified in the target environment. Therefore we do not consider this a necessary option in the transformation itself.

7 Conclusions and future work

We have described a way of efficiently and easily constructing an interface between two software development environments with the help of the ALCHEMIST transformation generator. All transformations were developed during a six-month period in 1994-5. Experiences from the implementation were used to make improvements to ALCHEMIST as well as to the transformations themselves. The three previous sections described our solutions to some important problems encountered in the transformations. As a general summary, we solved all the problems we encountered and came up with quite efficient transformation modules. Additional problems could be solved by letting the user interact before, during, or after a transformation. We also addressed the problem of reverse transformations.

Our interface supports a situation where software development is started in the source environment and some part of it, e.g., the user interface is transformed into the target environment for detailed design and implementation. Other interesting needs for such transformations are in strong co-operational use of both environments or in a reuse oriented approach [15]. If we aim to use two software development environments in strong cooperation, developing some parts of the application in one and some parts in the other environment, we may need information transfer in both directions. In a reuse oriented approach the direction of information transfer would be mainly from the environment used in the later phases of development (in our case, FOUNDATION), since existing reusable material mostly consists of design structures and code developed in CASE environments.

Acknowledgements

We would like to thank Andersen Consulting, in particular Luis Montero, Angel Luis Muñoz and José Manuel Quesada from Andersen Consulting, Madrid, for helpful comments on our research.

References

[1] Andersen Consulting. *FOUNDATION Application Development, v. 2.0*, 1993.

[2] Andersen Consulting. *FOUNDATION Design, Analyze Application Requirements, v. 2.0*, 1993.

[3] Andersen Consulting. *FOUNDATION Design, Data Administration, v. 2.0*, 1993.

[4] Alan W. Brown and Maria H. Penedo. An annotated bibliography on integration in software engineering environments. *Software Engineering Notes*, 17(3):47 – 55, July 1992.

[5] John Domingue, Enrico Motta, and Stuart Watt. The emerging VITAL workbench. In *Knowledge Acquisition for Knowledge-Based Systems, 7th European Knowledge Acquisition Workshop, EKAW '93*, pp. 320 – 339, September 1993.

[6] ECMA European Computer Manufacturers Association. *Portable Common Tool Environment (PCTE), Abstract Specification*, December 1990.

[7] ISO International Organization for Standardization. *Information Resource Dictionary System (IRDS) Framework*, 1990.

[8] S. E. Keller, J. A Perkins, T. F. Payton, and S. P. Mardinly. Tree transformation techniques and experiences. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 190 – 201, June 1984.

[9] Marc Linster. Problem statement for Sisyphus: Models of problem solving. *International Journal of Human-Computer Studies*, 40(2):187 – 192, February 1994.

[10] Enrico Motta, Kieron O'Hara, and Nigel Shadbolt. Grounding GDMs: A structured case study. *International Journal of Human-Computer Studies*, 40(2):315 – 347, February 1994.

[11] Jock A. Rader, Alan W. Brown, and Ed J. Morris. Operational use of CASE integration: An investigation of the state of practice. *Journal of Systems and Software*, 28:59 – 68, 1995.

[12] Steven Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57 – 66, July 1990.

[13] Nigel Shadbolt, Enrico Motta, and Alain Rouge. Constructing knowledge-based systems. *IEEE Software*, 10(6):34 – 39, November 1993.

[14] Henry Tirri and Greger Lindén. ALCHEMIST — an object-oriented tool to build transformations between heterogeneous data representations. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, vol. II, pp. 226 – 235. IEEE, January 1994.

[15] A. Inkeri Verkamo. Cooperation of KBS development environments and CASE environments. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, pp. 358 – 365, June 1994.

[16] A. Inkeri Verkamo and Greger Lindén. Problems in interfacing tools of different development environments. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pp. 429 – 437, June 1995.

[17] P. Zarrella. CASE tool integration and standardization. Technical Report CMU/SEI-90-TR-14, Software Engineering Institute, Carnegie Mellon University, 1990.