In Proceedings of the Seventh International Conference on Software Engineering and Knowledge Engineering, June 1995, pages 429 -- 437.

Problems in Interfacing Tools of Different Development Environments*

A. Inkeri Verkamo Greger Lindén

University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 Helsinki, Finland

Abstract

Integrating two separate development environments requires information transfer between the tools or the underlying repositories of the environments. This information transfer has usually been solved by applying ad hoc transformations to integrate the tools within a single development environment. We study the problem of making tools in different environments cooperate without modifying the tools or their environment. This requires building a permanent interfacing mechanism between the two environments. Problems particular to this context are incompatibility of the source and target languages, multiple views used within the environments, different naming conventions, and different ways of representing graphical information.

1 Introduction

The production of knowledge based systems (KBS) has long been performed using prototyping and ad hoc type methods instead of systematic development similar to more conventional software development, e.g., in building information systems. One of the reasons for this may be the different backgrounds of the people building KBSs and those in software engineering: KBS developers tend to be experts in AI and social sciences, but not necessarily in software engineering. As large embedded KBSs are being developed, there is an increasing need of systematic support for the KBS development. Some projects aiming at building KBS development environments are already under way (e.g., KADS and KADS II [21], and VITAL [16]). Presently, however, much of the methods developed for the special needs of KBS development, such as classification methods or other problem solving methods for knowledge acquisition, are implemented in separate tools that are in no way integrated to any software development environment.

A similar problem is faced by a team of software developers with a plethora of useful but separate tools available for their purposes. More often than not they have neither the time nor the means to switch into a fully integrated project support environment (IPSE) [12]. Recently reported experience indicates that the operational use of CASE integration is still far from straightforward [15]. A pragmatic solution is to use whatever tools are necessary to build the product. If the tools are not a part of an integrated environment, information must be transferred from one tool to another. Typically, the source tool and the target tool do not fully match, since the basic reason for using both tools lies in their different abilities. Even though an IPSE should be open to external tools, commercial CASE environments are often not really open in this respect. The cost involved with openness may not be justified if most of the users are satisfied with the toolset offered by the environment per se. It may even be considered a commercial disadvantage to offer a system that makes it too easy to switch over to using the competitor's product. In some cases the vendors offer an import/export mechanism, e.g., based on a standard database interface (SQL), if the tools use reasonably similar representations, but in more complex situations this is usually not the case.

Several studies have considered the requirements for integrating tools of different developers into a common environment (see, e.g., [11, 20]). When designing an IPSE, the tools may be modified in order to better adjust to the environment. However, if we want to make use of a non-integrated set of commercial tools, the environments involved may not be changed. In this paper we consider the problems involved in

*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 consortium. The e-mail address of the authors is verkamo@cs.Helsinki.FI

building an interface between tools in two independent environments, in order to transfer the necessary information. — Related problems in the context of interfacing mixed-language software components during execution time have been treated, e.g., in [14].

The paper has the following structure. In section 2, we describe some characteristics of the tools and development environments that affect the interfacing mechanism. In section 3, we present two tools that we use as a source of examples in the rest our presentation. The tool interface is part of a project developing a KBS development environment [16]. Section 4 describes the types of problems involved, supported by our example. In section 5 we propose an interfacing strategy to facilitate solving these problems. We close by a brief conclusion of the problem area.

2 Tool and environment characteristics

In this section, we discuss briefly the tool and environment characteristics that may affect the information transfer from one tool to another.

The source and target tools may handle the same or different *levels of abstraction.* The transformational approach to software development [13] concentrates on transformations from a higher level of abstraction to a lower level of abstraction, such as transformations from analysis to design models (see, e.g., [1, 6]). However, we may also want to interface source and target models on similar levels of abstraction.

If we want to move entire software artifacts from an outdated tool to a new target environment (defined as *immigration* in [17]) only *unidirectional* information transfer is adequate. Similarly, if only a small part of the development requires (local) use of a non-integrated tool we only need transfer from the primary environment to the tool. If the tools represent different views of the same system development phase [11] *bidirectional* information transfer is needed. This is the case in an embedded KBS where we want to use both environments in parallel, KBS tools for the KBS part and CASE tools for the rest of the application [19].

The *scope of the transfer* in software immigration is much larger than the scope of a local transfer. Immigration requires that all the information of the source model is contained in the transfer and that all the structures and attributes of the target environment are produced during the transfer [17]. To support local use of a tool, only a limited part of the source model is transferred to the target environment. Even though limited in scope, the transfer may still be bidirectional.

If we need actual cooperation of the two environments, not merely singular cases of information transfer, we have to address the question of *change propagation.* The transferred part may be changed in either environment and we would like the changes to propagate to the other environment. Full-scale change propagation is practically impossible for ready-made tools since it requires that the tools could handle 'update messages' sent by an external environment. A more practical form of change propagation are *incremental updates*, where only those elements are transferred again that have actually changed. This also preserves any target model modifications outside the updated part.

In a totally non-integrated environment, we may have to account for situations where the two environments run on *different platforms.* The physical and the low level logical differences require some additional complexity from the transformation mechanism.

In [17], a taxonomy of *storage structure forms* is presented, with respect to their usefulness for immigration. The least organized structure is a group of files within a directory system. If the information is stored in simple files the tools must interpret the structure of the files directly. Each tool may even use its own representation format which must be defined anew for each transformation. Integrated CASE environments use a common repository to store all the information handled by the tools. Typically, the repository is built on a commercial database system and it includes all conceptual and diagram elements as objects. Information is transferred between the environment and other tools by import/export routines built on database functions. A repository-based environment thus supports object import and export, which makes it easier to build an interface without changing the tool itself.

Even using a database as the basis of the repository does not guarantee easy import and export. The *granularity* used within the repository affects what elements can be imported and exported. As an example, let us look at an entity-relationship diagram. If each entity and each relationship is stored as a basic repository element such elements can be imported and exported separately. On the other hand, if the graphical information about the diagram itself is stored as a single repository element, the graphical presentation of the diagram can only be imported and exported as a whole.

The purpose and *scope of the tools* in a CASE environment may vary considerably. Often several tools are used during the same phase to model different aspects of the application [12]. In the requirements specification phase, an entity-relationship diagram is used for data modeling, a data-flow diagram for functional modeling and a flow chart for behavioral modeling. Although each fine-grained tool is easier to master a group of separate tools may have inconsistent or confusing notations [5]. Instead of separate tools we might use a single tool with a wider scope, such as high-level Petri nets for all three aspects of the requirements specification [5]. An extended tool is probably more difficult to learn, but this may pay back in the long run, reducing inconsistency and improving productivity [5]. From the point of view of interfacing, if the tools have a small scope it is more probable to find close correspondence between them. On the other hand, the actual work involved in building the necessary interfaces may be smaller for one tool with a wider scope than for a number of separate tools.

What we would like to have is as much *automatic support* for the transfer as possible. In all but the most trivial cases, only semi-automatic information transfer is possible, requiring the user to define what is transferred, what is done with those parts of the source model that cannot be transferred, and how those target language requirements will be fulfilled which are not defined by the source. Compared to completely manual information transfer, any assistance from an automated transfer mechanism is welcome.

3 Introduction of our example environment

Our example environment originates from a research project, VITAL[1], which aims at developing a methodology and a workbench for KBS application development [16, 7]. The openness of the VITAL workbench requires that information can be imported to as well as exported from the workbench. To try out this goal, an interface has been built between the VITAL workbench and a commercial CASE environment. We are specifically looking at the interface between the conceptual modeling tool of VITAL and the analysis and design tools of the CASE environment. We shall briefly describe the essential features of both toolsets.

[1]VITAL is an ESPRIT-II project funded by the European Communities and the Technology Development Center of Finland.

3.1 The VITAL conceptual modeling tool

The conceptual modeling tool of VITAL is the OCML (Operationalizable Conceptual Modeling Language) editor [7]. The OCML editor includes capabilities for presenting coarse level (GDM) models [8] as well as more refined diagrams that include domain data, domain knowledge, and problem solving behavior. As such, it covers the conceptual modeling phase and even part of the design phase of the KBS development. The OCML editor, as well as the entire VITAL workbench, runs in Sun/Unix workstations using a commercial Lisp environment.

The OCML diagrams for domain data and knowledge representation are *domain diagrams* which include graphical elements for *concepts*, *relations*, *concept instances* and *relation instances*. The diagrams for problem solving behavior are *task diagrams* which include graphical elements for *subtasks* and *roles*, as well as elements for defining the control structure, such as *sequencing* of subtasks or *choice* between subtasks (Figure 1).

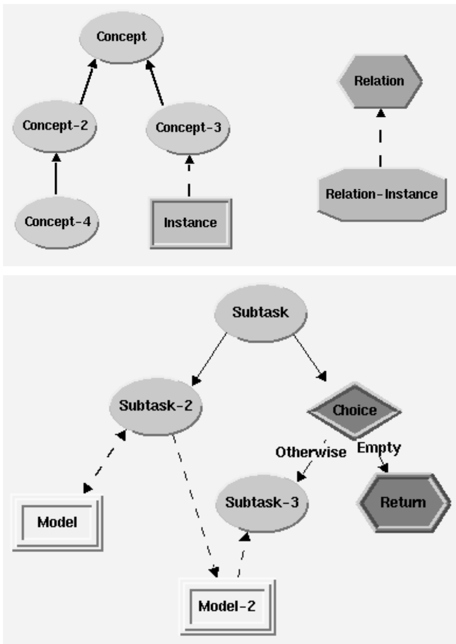

Figure 1: Examples of OCML domain and task diagrams.

There is no global repository for the objects processed in the VITAL workbench. The persistent representation of OCML is Lisp. Part of the information included in the persistent representation (e.g., the *properties* of concepts, see Figure 2) is not presented in the diagram, but it can be manipulated through the menus of the OCML editor. This information should also be included in the transfer.

```
;;; -*- Mode: Lisp, Design Domain Layer; ...
(dale:defgraphics dale:coordinates
  dale::concept (private (505 132 585 182)
                 client (275 35 355 85) ...))
(def-concept private ((\:\:\: client
   dale::kldesign-isa-link dale::concept))
   ((:slots age independent civil-status)
   (:inter-diag-alias))) ...
```

Figure 2: An example of OCML's persistent representation.

3.2 The CASE environment

The CASE environment is Andersen Consulting's FOUNDATION[2] [2], which is a typical example of advanced commercial CASE environments. FOUNDATION runs in PCs and uses the OS/2 operating system. The FOUNDATION toolset for the analysis and design phases, FOUNDATION Design, uses a repository that is built on the commercial database system DB2/2[3]. Since FOUNDATION Design supports object import and export [4] building an interface does not require changing the tool itself. FOUNDATION Design contains several tools for the analysis and design phase [3], including diagramming tools for *entity-relationship diagrams* (ERD), *data flow diagrams* (DFD), and *procedure diagrams* (PD) (Figure 3).

Importing and exporting repository information is based on using separate import/export files (*exp*-files, see Figure 4). In addition to the *exp*-files, graphical information of certain diagrams is included in separate binary coded graphical files (*lod*-files).

The repository uses an entity-relationship model. Diagrams and their elements are presented in the repository as entities. Information such as "Entity E belongs to diagram X" is presented in the repository as relationships between the corresponding entities.

A natural basis for tool cooperation is to transform OCML domain diagrams to FOUNDATION Design ERDs since they contain elements that correspond fairly well. For the OCML task diagrams there is no single representation in FOUNDATION Design that could contain all the information involved. The OCML task diagrams could be transformed either to FOUNDATION Design DFDs or to PDs, where the DFDs can describe the data flow part and the PDs the control part of the task diagram.

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

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

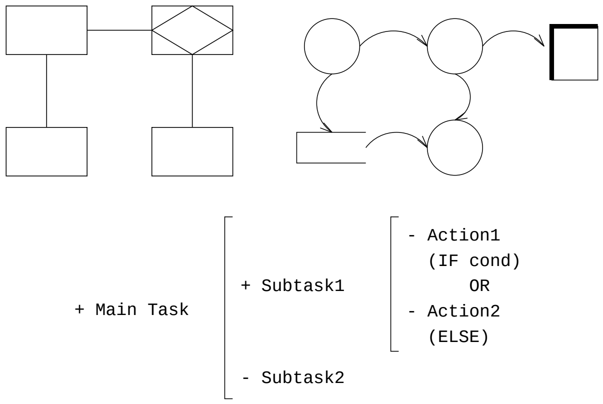

Figure 3: Examples of FOUNDATION Design ERD, DFD, and PD.

```
HEADER                                      ...
0                                           ...
2DEDMODEL        1                          ...
2DEDMODEL        2ENTIRE DIAGRAM            ...
2DEENTITY        1                          ...
2DEENTITY        2BOX                       ...
 ...
3DEDMODELDEENTITY3                          ...
3DEDMODELDEENTITY4ENTIRE DIAGRAM            ...
3DEDMODELDERELSHP3                          ...
3DEDMODELDERELSHP4ENTIRE DIAGRAM            ...
 ...
```

Figure 4: Part of a FOUNDATION Design export file.

4 Typical problems in interfacing the tools

This section describes what kind of problems must be solved to be able to transfer information between the source and target environments. Particularly we study incompatibility of the source and target languages, multiple views used within the environments, different naming conventions, and different ways of representing graphical information.

4.1 Source and target language incompatibility

A concept of the source language may not have any corresponding concept in the target language, or vice versa. If the source model contains concepts that cannot be presented (or are not meaningful) in the target environment a straightforward solution is to omit such concepts. Information about the omitted concepts could be included as comments in the target environment. If a reverse transformation is performed these comments must be processed to regain the information that was omitted in the original transformation. This requires more complexity from the interface.

On the other hand, if the target environment requires some structures that have no corresponding elements in the source model some elements must be added by the interface. This usually requires user involvement, but the interface can assist by notifying when such additions are needed and by suggesting appropriate elements [6].

As an example, let us look at an OCML domain diagram that we want to transform into an ERD (Figure 5). The domain diagram includes a concept `Client` and an instance `Mrs A. Client`. In an ERD, instances are usually not presented. Technically, we could of course produce an *instance* entity (`Mrs A. Client`) that has an `is instance of` relationship to the *concept* entity (`Client`, see Figure 5). However, for the purposes of an ERD this is usually not reasonable. Also, if we use both *instance* entities and *concept* entities in the ERD, we must be able to differentiate between them, even though the target environment treats them conceptually similarly.

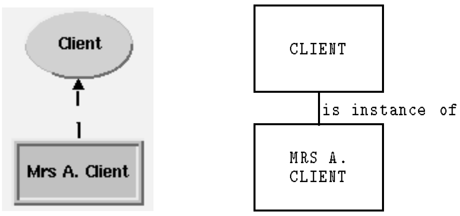

Figure 5: An OCML instance and a corresponding FOUNDATION Design entity.

4.2 Multiple views

One element in the source language may sometimes correspond to several elements in the target language. An element might also produce elements in two separate diagrams or target files. Similarly, several elements of the source language may be needed to produce one element of the target language, or information within two source files might have to be combined to produce one target file.

This problem occurs when transforming an OCML task diagram into a DFD. Figure 6 contains the persistent representation of a simple OCML task diagram consisting of a single subtask (`classify_client`). In FOUNDATION Design, graphical information of a DFD is presented in a *lod*-file. The *exp*-file contains the process and diagram definitions and the *lod*-file contains the coordinates of the process within the diagram. The *exp*-file references the *lod*-file, and the elements in the *lod*-file contain references to the corresponding elements within the *exp*-file.

```
;;; -*- Mode: Lisp, Design Task Layer; ...
(dale:defgraphics dale:coordinates
  dale::subtask
     (classify_client (528 12 608 62)))
(def-task classify_client NIL NIL)
```

Figure 6: Persistent representation of a small OCML task diagram.

An example of producing two targets from one source is the OCML task diagram which must be transformed into two diagrams, a DFD and a PD, in order to present both the data flow content and the control structure. We could do this by performing two separate transformations, producing first a DFD and a then PD. However, if we wanted to bind the two views together (e.g., place links from the DFD to the corresponding elements of the PD) both transformations would have to be produced during the same run.

An example of combining several source views into one target is a situation where the source environment permits using subdiagrams, but the target environment only allows one level of diagramming. The information presented in all the subdiagrams must be combined into a single diagram. Even though the connections between the combined elements of several graphs can be produced on the conceptual level there is no automatic way to produce a graphically pleasing representation of the combined diagram. A similar problem arises if the source environment permits using aliases or duplicates of the graphical elements, but the target environment doesn't. In this case all duplicates must be collapsed into a single element by the interface.

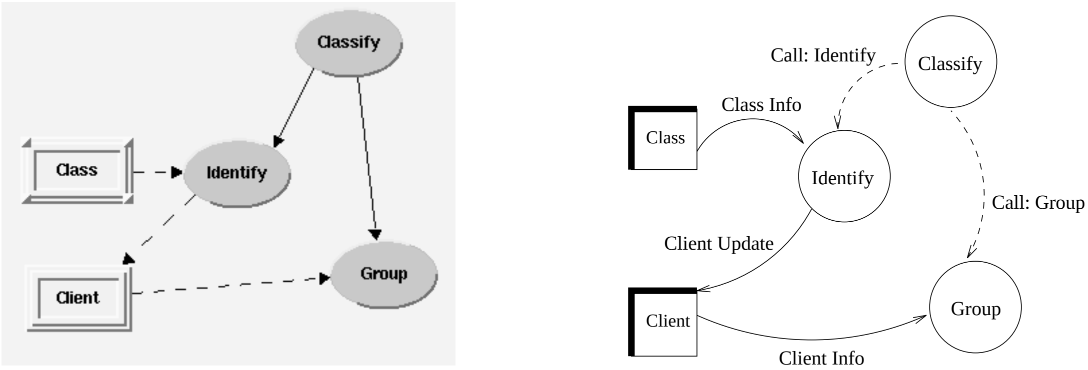

Figure 7: Examples of arrow naming in OCML and FOUNDATION Design.

4.3 Name incompatibility

Name incompatibility problems typically result from the non-standardization of methods. The source and target languages may use different naming strategies. These include decisions about which elements of the diagram are named, what is the structure of a name, is there some sort of automatic naming, and which names are required to be unique.

Some objects in the source model may not have a name at all while the target environment requires them to be named. A trivial example in our environment is the diagram itself: OCML does not name the diagrams, but FOUNDATION Design requires all repository elements to have a name, including the diagram. Another example of the naming problem are the arrows in OCML task diagrams vs. FOUNDATION Design DFDs (Figure 7). In an OCML task diagram, the arrows representing information flow, resp. control flow are presented as dashed, resp. solid lines. In general, the flows are not named (Figure 7). In the corresponding DFD, all flows must have unique names (Figure 7). Therefore, unique flow names must be generated by the interface.

The source and target language may favor or require different types of identifiers. The set of acceptable characters or the maximum size of a name may be different. The transformation may require changing or truncating some names. In our example environment, `client.name` is a valid OCML identifier, but not a valid FOUNDATION Design identifier.

Each environment requires some unique identifiers for the elements. These may or may not be the same as the names in the diagrams. If system-generated unique identifiers are used, these typically have a very tightly defined structure to avoid identifier clashes. In our example environments, OCML uses names as identifiers and checks for their uniqueness as they are defined. Since no common repository is used, the names only need to be unique within a diagram. FOUNDATION Design uses two types of unique identifiers: it generates internal unique identifiers for internal repository use, but it also requires that the user defined object names are unique within a project environment. In the case of import, the user is responsible for the uniqueness of object names to avoid unexpected effects such as replacing a previous object by a new one.

4.4 Transforming graphical information

Some problems may arise because the source and target environments use different ways of representing graphical information. If the coordinate systems are different this may require scaling and other arithmetic operations on the coordinates. Some additional coordinates may need to be computed by the interface. In our example, OCML presents the coordinates of a box by giving the coordinate pairs of the upper left and the lower right corner, e.g., `(420 40 513 98)`. For the same box, FOUNDATION Design uses the coordinate pair of the upper left corner and the dimensions (width and height) of the box, e.g.,

```
DMD_ENT_POS_X          +000420
DMD_ENT_POS_Y          +000040
DMD_ENT_SIZE_X         +000093
DMD_ENT_SIZE_Y         +000058
```

Another part of the graphics that typically requires additional computation is the placement of arcs or arrows. This may be complex if the source and target environments use a different placement strategy. In our example, OCML gives no explicit coordinates for the arrows since the tool draws them automatically as straight lines between the two connected objects. The ERD diagrammer of FOUNDATION Design uses polylines consisting of straight horizontal and vertical parts and stores the coordinates of both endpoints and each corner of the line (Figure 8). The DFD diagrammer of FOUNDATION Design uses arcs and stores the coordinates of the end points and the middle, or high point of the arc (Figure 7).

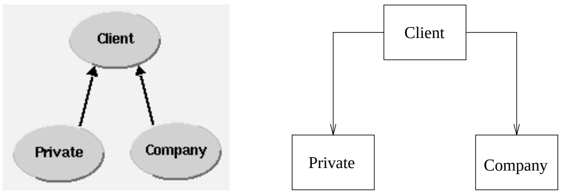

Figure 8: Examples of arrow types in OCML and FOUNDATION Design.

If the source and target environment use different types of arcs or arrows the placement of the objects may need modification to look pleasing. Such changes must be done manually after the transformation since automatic placement strategies are too complex to be embedded within the interface.

4.5 Other problems

Because of the different structure of the import/export interface the source and target environments may require different counters and numbering mechanisms for the objects. In our example, the persistent files of OCML contain no number information about the diagram except for the coordinates of the graphical elements. The *lod*-files of FOUNDATION Design require counts for the different types of elements, numbering of the graphical elements used for connection information, etc. These numbers have to be computed by the interface.

The physical information transfer from source to target environment may be done through `ftp` or manually using a diskette. If the environments run on different platforms additional filtering may be needed before the actual transfer. The environments may, e.g., use different character sets. In our example environments, information is transferred from Sun/Unix to PC/OS2 which requires transformation of certain control characters.

5 Some suggestions for the interfacing strategy

We have built a prototype transformation between OCML and FOUNDATION Design by using the ALCHEMIST transformation generator [18]. ALCHEMIST is used in the following way: The user describes the source and target representations as context-free grammars and defines a mapping between the rules of the grammars (Figure 9). The specification of the mapping is based on the notion of TT-grammars [9]. Informally, a TT-grammar may be viewed as generating subtrees over the target grammar from subtrees over the source grammar. ALCHEMIST generates the transformation code from the transformation specification (Figure 9). The result includes `C++` code for a parser, a mapper, and an unparser. A similar approach to generating transformations has been used in the Integrated Chameleon Architecture [10].

During transformation execution, the transformation module builds a parse tree from the source file, maps the source parse tree to a target parse tree, and writes the result to a target file (Figure 10). The target file is then imported into the FOUNDATION repository.

As examples of our transformation specifications, the OCML domain and task grammars contain 85 and 80 productions, respectively. The *exp*-file (repository import file) grammars for ERDs, DFDs, and PDs contain 202, 170, and 111 productions, respectively. The *lod*-file (graphical file) grammar contains 127 productions. The mappings for ERDs, DFDs, and PDs contain 35, 37, and 13 mapping rules, respectively. The OCML to FOUNDATION Design transformations have been found fairly efficient for our needs, typical transformation speeds are about 1 kB/s. Our OCML files are comparably small, often less than ten kilobytes while the FOUNDATION Design files can be as much as thirty times larger.

We shall briefly characterize some solutions for the problems described in section 4, offered by using the ALCHEMIST approach. Source and target language incompatibility do not pose a problem. Objects that are omitted in the transformation are simply not described in the mapping. Several target objects can easily be produced from one source concept. By simple pre- and postprocessing filters, we can remove unnecessary concepts from the source file or divide one target file into two. Name incompatibility checking is added to the transformation module by including generation and lookup facilities for identifiers. Some computing of information, such as counting the diagram elements, is likewise added to the generated `C++` code. All changes due to semantic action, e.g., name modifi-

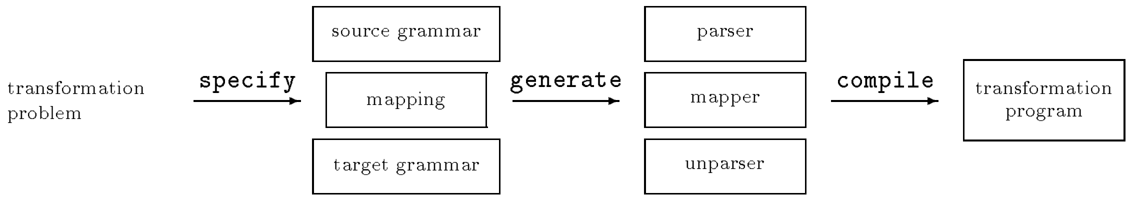

Figure 9: The transformation generation in ALCHEMIST.

cations, are saved in a log file, in order to be usable in case of reverse transformation. – The transformation work is still in progress, but a more detailed evaluation of the experiences in the implementation will be reported later.

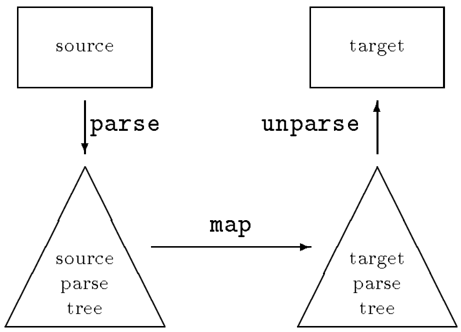

Figure 10: The transformation process.

6 Conclusions

During recent years we have seen a substantial amount of software development environments being created. In some environments, the full integration between development tools has been stressed, e.g., by requiring tools to use a common look-and-feel or a common repository [20]. Other environments are more open, emphasizing the possibility to include but not integrate external tools. Information transfer between tools is needed in order to make the software development proceed from analysis to executable code in a convenient way. In integrated development environments external tools may need to be slightly modified to be fully integrated into the environment. In an IPSE, information transfer typically means transformation of the output of one tool into the input of another tool used for the next phase of development. Few, if any, have concentrated on integrating different environments and providing information transfer between tools at the same level of abstraction.

Integrating two ready-made development environments brings out a number of information transfer problems. Transformations may be required between repositories of repository-based environments, between single tools of different environments, or between a single tool of one environment and the repository of another environment. Transformations can be uni- or bidirectional, automatic or user-assisted. More technical problems occur as well, such as differences in tool scopes and granularity of information.

Information transfer needs have usually been solved ad hoc, by generating a specific transformation module for each interface. We believe that by abstraction of the problems involved we can find more general solutions for information transfer needs. Typical problems to be solved when building interfaces between independent tools include language incompatibility, different or multiple views within the tools, naming differences, different presentation of graphical information, and different software platforms using different character representations. We have demonstrated some ways of treating these problems in a prototype transformation between a KBS conceptual modeling tool and a commercial CASE environment. Our experience shows that even if tedious or complicated, the information transfer problems can be solved.

References

[1] Bruno Alabiso. Transformation of data flow analysis models to object oriented design. In *Proceedings of OOPSLA'88*, pages 335 – 353, September 1988.

[2] Andersen Consulting. *FOUNDATION Application Development, Version 2.0*, 1993.

[3] Andersen Consulting. *FOUNDATION Design, Analyze Application Requirements, Version 2.0*, 1993.

[4] Andersen Consulting. *FOUNDATION Design, Data Administration, Version 2.0*, 1993.

[5] Hany M. Atchan. An experiment in eliminating the separation between data, functional, and behavioral models during requirements specification. In *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE '94)*, pages 2 – 9, June 1994.

[6] Germinal Boloix, Paul G. Sorenson, and J. Paul Tremblay. Transformations using a meta-system approach to software development. *Software Engineering Journal*, 7(6):425 – 437, November 1992.

[7] John Domingue, Marc Eisenstadt, and Blaine Price. The VITAL bug location methodology. Internal Release OU/231/P/2, ESPRIT Project P5365 VITAL, April 1993.

[8] Gertjan van Heijst, Peter Terpstra, Bob Wielinga, and Nigel Shadbolt. Using generalised directive models in knowledge acquisition. In *Current Developments in Knowledge Acquisition, 6th European Knowledge Acquisition Workshop EKAW '92*, pages 112 – 132, May 1992.

[9] 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*, pages 190 – 201, June 1984.

[10] Sandra A. Mamrak, Julie Barnes, and Conleth S. O'Connell. Benefits of automating data translation. *IEEE Software*, 10(4):82 – 88, July 1993.

[11] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49 – 57, January 1991.

[12] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. Expressing the relationships between multiple views in requirements specification. In *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pages 187 – 196, May 1993.

[13] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199 – 236, 1983.

[14] James Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151 – 174, January 1994.

[15] 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.

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

[17] Michael H. Sokolsky and Gail E. Kaiser. A framework for immigrating existing software to new software development environments. *Software Engineering Journal*, 6(6):435 – 453, November 1991.

[18] Henry Tirri and Greger Lindén. ALCHEMIST — an object-oriented tool to build transformations between heterogeneous data representations. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume II, pages 226 – 235. IEEE Computer Society Press, January 1994.

[19] A. Inkeri Verkamo. Cooperation of KBS development environments and CASE environments. In *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE '94)*, pages 358 – 365, June 1994.

[20] Anthony I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments, Proceedings of the International Workshop on Environments, Chinon, France*, number 467 in Lecture Notes in Computer Science, pages 137 – 149. Springer-Verlag, 1990.

[21] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5 – 53, 1992.