In Hesham El-Rewini and Bruce D. Shriver, editors, Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, volume II, pages 226 -- 235, January 1994. IEEE Computer Society Press.

ALCHEMIST - an object-oriented tool to build transformations between Heterogeneous Data Representations*

Henry Tirri

Department of Computer Science
University of Helsinki
Helsinki, Finland

Greger Lindén

Department of Computer Science
University of Helsinki
Helsinki, Finland

Abstract

ALCHEMIST is a general purpose object-oriented transformation generator. ALCHEMIST provides a new approach to developing transformations between any well-defined representations. It allows users to define the syntactic structure of the data representations and the related structure associations with a grammar notation. These grammars can then be augmented with semantic operations. From this description a persistent object-oriented representation is formed and a transformation module is generated automatically. This transformation module relies on an object representation of the parse tree. In this paper the principles underlying ALCHEMIST are discussed and its object-oriented design decisions are described.

1 Introduction

The modern paradigm to provide open software environments with the resulting heterogeneous software platforms has dramatically increased the importance of effective methods of translating between different data representations. Even the most optimistic predictions acknowledge that several different representation standards will coexist in the future, regardless of the various standardization efforts. Examples of such coexistence can be found everywhere: object structure representations (CORBA, PCTE [5]), data representations (CODASYL [1], relational representations [4]), knowledge representations (INTERLINGUA [11], KL-ONE [3]), etc. It should be observed that transforming between data and knowledge representations is inherently more difficult than pure syntactic translations needed for image translations, since the semantics of the objects involved typically has a great effect on the translation process itself.

For data (or knowledge) representation transformations we adopt a view that a representation formalism X is a machine language of a corresponding abstract machine P_X. In the extreme case the transformation mapping has to incorporate most of the information about this abstract machine executing the "data language", i.e., provide a simulation mapping between the two abstract machines. Consequently in many cases the transformation has to be semi-automatic, and is even in the best case user-assisted. Developing such transformations with proper user interfaces is a tedious task, especially as the set of representations to be included tends to increase with time. Consequently the process of developing transformation software would benefit from a toolkit supporting the design of the various components involved. Such a transformation toolkit is akin to compiler-compilers [15], editor generators and report generators [13], however with many interesting special features and capabilities of its own. It differs considerably from compiler-compilers by assuming that in the general transformation case, the target code also has a complex structure that requires a grammar description. Similarly the semantic actions involved are more complex than in the regular translation process. The most closely related work to ours is the Intergrated Chameleon Architecture data translator [9]

ALCHEMIST is an object-oriented transformation tool under development in the VITAL project [10]. The VITAL project aims at developing an open industrial-strength software engineering workbench for building heterogeneous knowledge based (KBS) applications. However, ALCHEMIST is a gen-

*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 addresses of the authors are tirri@cs.Helsinki.FI and linden@cs.Helsinki.FI

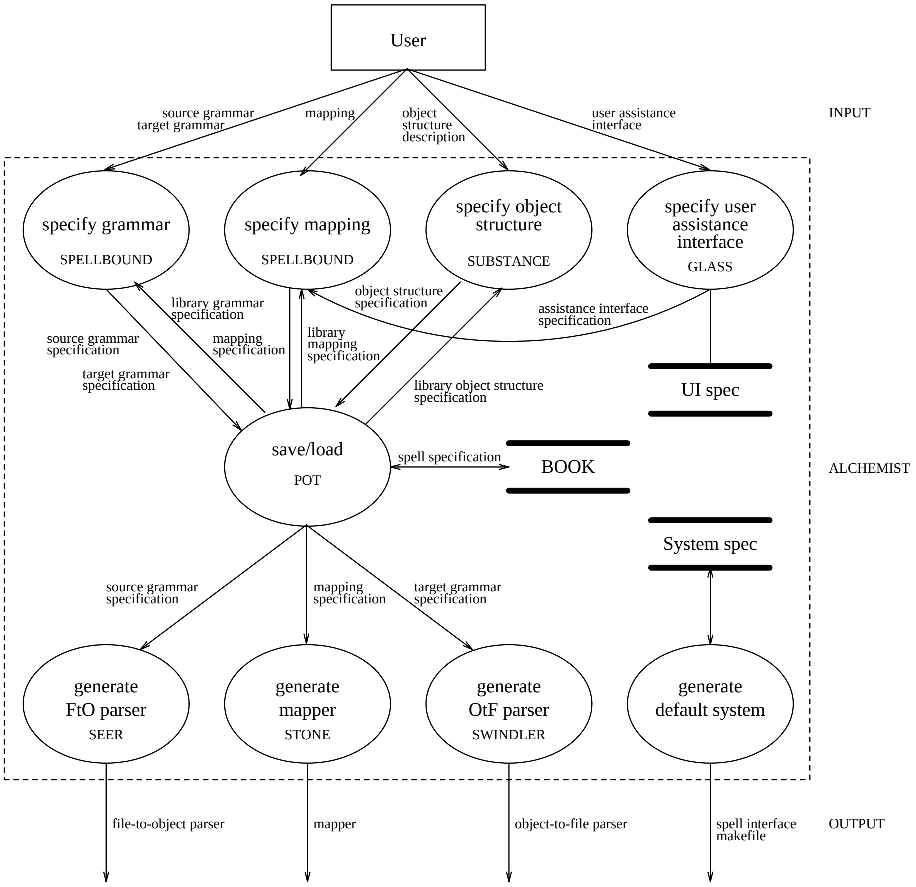


Figure 1: Data flow diagram of ALCHEMIST.

eral purpose transformation generator and its use is not limited to its development context of KBS applications. ALCHEMIST provides a sophisticated way of developing transformations between any well-defined representations, and is especially suitable for defining mappings between data representations in a heterogeneous database environment. ALCHEMIST allows users to define the syntactic structure of the data representations and the related structure associations with a grammar notation. These grammars can then be augmented with semantic operations. From this description a persistent object-oriented representation is formed, and this representation is used to generate automatically a transformation program, coined as a "spell". Similarly the generated program uses an object representation of the parse tree. Thus the approach adopted relies heavily on persistent object repositories, as in a "real world" translation the intermediate object representations are too large to fit in the main memory. Currently the system is being built on top of ODE object-oriented database system from AT&T [2]. The transformations are off-line, i.e., from file to file, but eventually the toolkit will be extended to handle also online transformations.

The ALCHEMIST transformation generator environment comprises of the following modules (for the structure of ALCHEMIST and a generated spell see Figures 1 and 2):

- a spell specification interface (SPELLBOUND),
- a file-to-object parser generator (SEER),
- an object-to-file parser generator (SWINDLER),
- a mapper generator (STONE),
- a graphical user interface generator (GLASS),
- an object structure generator (SUBSTANCE),
- a mapping library manager (BOOK),
- an object manager (POT).

ALCHEMIST produces transformation programs, spells. Although in principle spells can be executed as stand-alone processes, in many environments it is use-

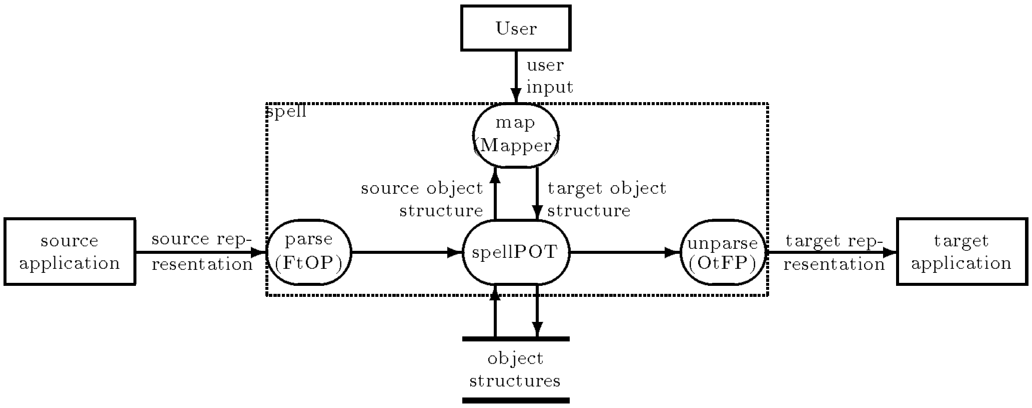


Figure 2: Data flow diagram of a transformation.

ful to collect several spells together to form a "macro spell". The need for macro spells is especially common for heterogeneous systems, where the development process requires a multi-step transformation between various applications. APPRENTICE is a tool for executing the actual transformations, i.e., for using a set of spells. APPRENTICE allows users to select graphically what transformations take place for the given files, and also to create macro spells.

Both ALCHEMIST and APPRENTICE interfaces run on OpenWindows/Unix environment. In the definition of the transformations full expressive power of C++ is allowed. This paper describes the general structure of the ALCHEMIST, as well as the object-oriented technical solutions used in building the toolkit. The theoretical framework of TT-grammars is also briefly discussed. We proceed by first describing how the transformations are specified using ALCHEMIST (Section 2). In Section 3 we address the problem of generating a transformation program, and Section 4 discusses the additional components in ALCHEMIST for transformation design support.

2 Specifying spells

The spells are specified using the SPELLBOUND window-based interface (see Figure 3). The specification process produces an object-oriented data structure which has all the necessary information to produce the C/C++ code for the transformation module. Although the SPELLBOUND interface is graphical, all the information has also a textual representation that can be edited by a standard character-based editor.

A spell specification object structure consists of object-oriented specifications of *grammars* of the source and target representations and *mappings* between these grammars. Correspondingly it includes objects representing individual productions, symbols, production groups, production group mappings, etc.

2.1 Specifying grammars

A source or target representation is specified by giving its context-free grammar. This grammar can be defined either interactively using the SPELLBOUND interface, or by giving a `yacc`-description [6] of the grammar in a file. For the spell specification a *context-free grammar* is represented as a four-tuple $G = (N, T, P, S)$, where N is the set of nonterminals, T the set of terminals, P the set of productions, and S a particular nonterminal called the start symbol. The set $V = N \cup T$ is the alphabet of the grammar. The elements in the alphabet are called symbols or tokens.

The specification of a representation grammar is highly syntax-directed. SPELLBOUND lets the user insert, delete, or modify existing productions. Since each production corresponds to an object, many productions can be edited at the same time. On the other hand, when processing a production, only one right hand symbol can be modified at any given time.

A grammar object has a name and a set of productions. Similarly each production consists of a name, a left hand side and a right hand side. A left hand side is one nonterminal, a right hand side consists of an ordered set of symbols, each either a nonterminal or a terminal. Productions with iterations are given with recursive nonterminals, and productions with disjunctions are represented as separate productions. Thus the grammar is defined in a traditional manner, but implemented as a collection of objects rather than table representations.

A grammar object is specified by giving links to its production objects. The nonterminal and terminal objects are specified by the production objects. The start symbol is the left hand side nonterminal of the first given production. For the production objects each right hand side can be associated with a semantic action related to the mapping in question.

Grammar and production objects can also be saved or loaded in textual form. This allows character based editing of the grammar productions with a regular editor. However, normally a grammar is displayed (and specified) through the SPELLBOUND *grammar interface*. With this interface the user can insert new productions, and delete or modify old ones. Adding new productions or modifying old ones is performed through the SPELLBOUND *production interface*.

A representation grammar with its productions is stored as persistent objects to an Object Management System. The source representation grammar is then used to generate a parser for the source files given to the spell in question. However, the primary purpose of having grammar objects in the spell specification is to create a mapping between a source grammar and a target grammar.

2.2 Specifying mappings

In a regular compilation process the target representation (e.g., machine code) is less structured than the source representation. For data transformations this is not the case, also the target representation can be highly structured. This brings us to an interesting problem. It is not enough that the transformation scheme attaches target code to each node in the parse tree of the source code, it also has to guarantee that the produced code is structurally correct, i.e., conforms to the constraints imposed by the target grammar. Thus the transformation mapping is between two parse tree structures. Adopting such a viewpoint is also useful when reverse transformations ("counterspells") are needed, since in many cases this symmetry allows reuse of the already existing mappings by simple reverse operations. Following this idea, the transformation mappings are defined between sets of grammar productions.

2.3 TT-grammars

The SPELLBOUND specification of mappings is based on the notion of TT-grammars developed at SDC [7]. TT-grammars are a formal description technique for describing transformations from one language to another. A TT-grammar contains context-free grammars for both languages. As discussed above the transformation between the representation languages is described by associating productions in both grammars. SSAGS — a syntax and semantics analysis and generation system [12] was extended to support certain TT-grammars.

A *TT-grammar* describes a relationship between a syntax tree over a grammar G_1 and a syntax tree over a grammar G_2. A TT-grammar may be interpreted as a description of a tree transformation technique. Transformations can be specified both ways, from trees over grammar G_1 to trees over grammar G_2 or vice versa, thus being especially suitable for our purposes where two-way transformations are common. Here we concentrate on one-way transformations from trees over a *source grammar* G_s to trees over a *target grammar* G_t. The relationship is described by associating groups of productions in G_s with groups of productions in G_t. In addition one needs to associate symbol occurrences in G_s with symbol occurrences in G_t.

Formally, a *TT-grammar* is a sextuplet (G_s, G_t, S_s, S_t, PA, SA), where G_s and G_t are the source and target grammars, respectively, S_s and S_t are sets of source and target subgrammars, respectively, PA is the set of *production group associations*, and SA the set of *symbol associations*. The *source* and *target grammars* are context-free grammars. The *source* and *target subgrammars* consist of sets of productions from the source and target grammars, respectively. A *production group association* is a pair consisting of a source subgrammar in S_s and a target subgrammar in S_t. A *symbol association* relates a symbol in a source subgrammar to a symbol in a target subgrammar (within a certain production group association).

The source subgrammars must satisfy the following restrictions. First, there must be a single start symbol in every subgrammar. Second, every other symbol in the subgrammar must be derivable from this start symbol. Source subgrammars specify subtree patterns to be matched against in the source tree; the target subgrammars specify the subtrees that are the result from a match in the source tree. A target subgrammar is *not* required to have a single start symbol; it can have several, resulting in a forest of target subtrees.

Informally, a TT-grammar may be viewed as generating subtrees in G_t from subtrees in G_s as follows. Let (pg_s, pg_t) be a production group association, where pg_s is a source subgrammar and pg_t a target subgrammar. The productions in pg_t are applied every time the productions in pg_s apply (all of them) to the source tree. Let two of the productions produced in the resulting forest of target subtrees be $t \rightarrow \alpha s \beta$ and $s \rightarrow \gamma$. Assume also that both instances of the symbol s are associated with the same source symbol. Then the two target subtrees are linked through s to form a single target subtree.

Example 1 Consider the following case, where a transformation between a simple entity relationship model and a relational model is needed. Assume that the source grammar G_s contains (among others) the following productions.

Part of source grammar G_s

ER	$\rightarrow$	*Ss*
Ss	$\rightarrow$	*S Ss*
Ss	$\rightarrow$	*S*
S	$\rightarrow$	*Entity*
Entity	$\rightarrow$	"ENTITY" "(" *Name* ")"
Name	$\rightarrow$	ID

Assume also that the target grammar G_t has the following (corresponding) productions.

Part of Target grammar G_t

Relational_db	$\rightarrow$	*Schemes*
Schemes	$\rightarrow$	ϵ
Schemes	$\rightarrow$	*Scheme Schemes*
Scheme	$\rightarrow$	*Name* "(" *Attributes* ")"
Name	$\rightarrow$	IDENTIFIER

The mapping must describe how we translate a persistent representation of an entity into a relation scheme. The following production group associations and symbol associations belong to the TT-grammar describing the required transformation. We follow the notation of Keller et al. and denote symbol associations by *source_symbol.target_symbol*.

Source subgrammars S_s

s_{i1} :	*ER*	$\rightarrow$	*Ss*
s_{i2} :	Ss_1	$\rightarrow$	*S* Ss_2
	S	$\rightarrow$	*Entity*
s_{i3} :	*Ss*	$\rightarrow$	*S*
	S	$\rightarrow$	*Entity*
s_{i4} :	*Entity*	$\rightarrow$	"ENTITY" "(" *Name* ")"
	Name	$\rightarrow$	IDENTIFIER

The corresponding target subgrammars are the following (s_{ij} corresponds to s_{oj}).

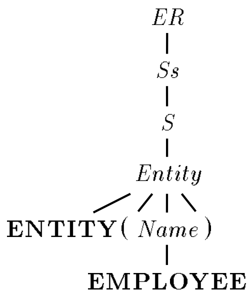
Target subgrammars S_t

s_{o1} :	*ER.Relational_db*	$\rightarrow$	*Ss.Schemes*
s_{o2} :	Ss_1*.Schemes*	$\rightarrow$	*Entity.Scheme* Ss_2*.Schemes*
s_{o3} :	*Ss.Schemes*	$\rightarrow$	*Entity.Scheme* *Entity.Schemes*
	Entity.Schemes	$\rightarrow$	ϵ
s_{o4} :	*Entity.Scheme*	$\rightarrow$	*Name.Name* "(" *Entity.Attributes* ")"
	Name.Name	$\rightarrow$	IDENTIFIER.ID

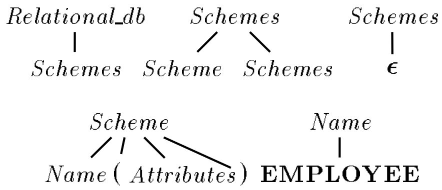
We translate an entity *EMPLOYEE* to a relation scheme. The graphical representation is, e.g.,

$\boxed{\text{EMPLOYEE}}$

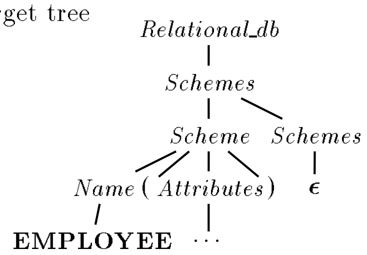
and according to the source grammar, the persistent representation is `ENTITY(EMPLOYEE)`. We have the source tree over the source grammar



A transformation produces target subtrees



which are connected through the symbol associations to the target tree



The result of the transformation is `EMPLOYEE()`. If the entity has attributes, appropriate subtrees will be produced and inserted into the target tree. □

2.4 Mappings

In ALCHEMIST's object-oriented view, each of the TT-grammar associations discussed above is also represented as an object. Thus a mapping between two grammars is implemented as a set of *production group association (PGA) objects*, each consisting of two production group objects, one over the source grammar and the other over the target grammar. Connected with each production group association object we have a set of *symbol association objects*. A production group object consists of a set of production objects, correspondingly a symbol association object consists of two symbol objects, a source symbol object and a target symbol object. For more complex transformation needs, semantic actions (C/C++ code) can be attached to production group association objects. Production group association objects are also linked to conflicting production group association objects, i.e., PGAs with the same source production group.

The SPELLBOUND specification of mappings is based on specifying the corresponding PGA associations. The user forms and connects subgrammars through the *production group association manager*. This interface allows the user to insert source productions in the production group at the left, and target productions in the production group on the right. Productions can be deleted from any group. However, the user cannot modify the productions; modifications must be done through the production interface. A symbol association connects a source symbol with a target symbol, where the symbols must belong to the same production group association. The user connects symbols in source and target productions in a production group association by using the *symbol association interface*.

An ALCHEMIST snapshot from a specification phase is shown in Figure 3.

3 Generating the spell code

As discussed above, in ALCHEMIST the user can specify the transformation interactively with the SPELLBOUND interface. This incremental process produces a persistent object structure which is stored into an OMS. The persistent object structure consisting of grammar and mapping objects with the semantic actions can then be used to generate the actual spell code, i.e., the C/C++ code that performs the transformation. The spell code itself is modular, and consists of the following modules:

- a File-to-Object Parser (generated by SEER),
- a Mapper (generated by STONE),
- an Object-to-File Parser (generated by SWINDLER),
- a User Interface (generated by GLASS) and
- an Object Management System.

Correspondingly, ALCHEMIST has components that generate each of these spell modules from the object structure specifying the spell in question. Thus in the following we proceed by describing briefly the spell components together with the corresponding generating ALCHEMIST component.

3.1 Generating File-to-Object Parsers with SEER

In the spell implementation, the File-to-Object (FtO) Parser reads a source file and constructs an object structure (OS) according to a context-free grammar. The object structure represents the parse tree of the text file to be transformed. To generate parsers ALCHEMIST uses the facilities provided by `yacc` and `lex` [6, 8].

To implement a File-to-Object Parser the following parts are needed:

- a *lexical specification* — specifies which lexemes the file representations contains. For the `yacc`-parser, this specification consists of a default `lex` specification (possibly edited).
- a *parsing specification* — specifies the syntax of the file representation. For the `yacc`-parser, this is a `yacc` specification (without semantic actions). The `yacc` specification can be automatically derived from a grammar specification provided by SPELLBOUND.
- an object structure (OS) construction specification — specifies how to build the parse tree from the source file. In the `yacc`-parser, this specification is given as semantic actions associated with every symbol of the grammar.

The File-to-Object Parser Generator (SEER) reads the grammar specification of the source representation language and produces a parser for it. SEER defines the generation of the parser locally, giving every grammar and production object its own method for producing a part of the parser. In the case of producing a `yacc` specification, the grammar specification can be considered as a user interface for `yacc` and `lex`. Automatic semantic actions, which are inserted into the `yacc` file for every production, build a subtree with the left hand side of the production as root and the right hand side symbols as children.

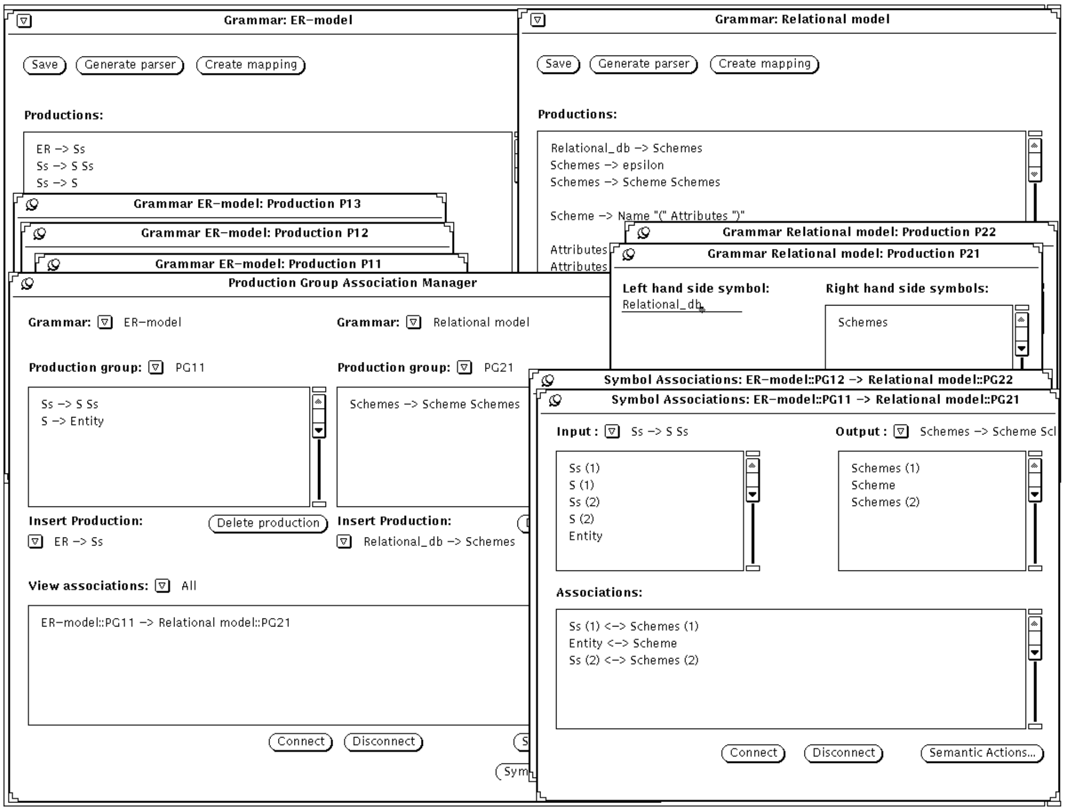


Figure 3: The SPELLBOUND interface.

User defined actions can be given while the grammars are specified, or attached later (with GLASS). The `yacc` and `lex` files can also be modified directly by the user. The SPELLBOUND interface provides also a possibility to import a `yacc` specification and present the grammar and the semantic actions in the grammar and production interfaces.

3.2 Generating Object-to-File Parsers with SWINDLER

In the spell implementation, the Object-to-File (OtF) Parser reads an object structure and writes the contents to a file. Extracting the text out of an object structure is slightly more complicated than just writing the contents of the leaves of the parse tree to a file. The specific OtF Parser is defined by the user. It traverses the object structure and writes the textual representation, with the structure pruned, to a file. During this process some objects may be ignored (e.g., information added by user), some consulted and the textual representation of some objects may be written to a file. In general the OtF Parser writes the contents of all (leaf) objects to a file except for some predefined nodes. It should be observed that theoretically, the OtF Parser can perform transformations, pretty printing, etc., even if primary transformations are done during the mapping phase.

The OtF Parser Generator (SWINDLER) produces an object-to-file parser for the target language. There are two main strategies for generating an OtF parser. First, we can produce separate OtF-generating methods for each object type in an object structure. Such a method knows how to extract the textual representation. An alternative would be to use a global traversal of the object structure that knows how to extract the textual representation from different objects. SWINDLER follows the former approach. In simple cases, an OtF parser is independent of the grammar specification, extracting text from certain objects and ignoring other objects. Generating an independent, general OtF parser is easy, as the same code can be used for every generated spell. In more complicated cases, the OtF parser has to be tailored for a specific grammar specification. Currently ALCHEMIST only supports grammar independent OtF parsers.

In the generation process SWINDLER produces a generating method for each type of object (each nonterminal). This method generates the textual representation of its owner object and call methods for generating the textual representation of its subobjects (children). Since the resulting representation can be very large, writing to the file is performed incrementally.

3.3 Generating Object-to-Object Mappers with STONE

The actual transformation in the spell is performed by the Object-to-Object Mapper, which transforms one object structure to another. These transformations can be either automatic or user-assisted.

A spell transformation consists of the following modules which are iterated until the entire source structure has been traversed:

1. Matcher — finds a substructure in the object structure (OS) that matches a source substructure.
2. Producer — creates target substructures and links the target objects to associated source objects.
3. Linker — Merges target objects that are of the same type and associated to the same source object.

The matching part is implemented as a tree matching process, a traversal of the OS and simultaneous checking of a source substructure library. Producing a substructure of the other representation is similar to the constructing of the FtO Parser. However, the same order in which the objects are created cannot be used. The mapping construction may produce target substructures in a very different order, thus an object structure indexing scheme is used. The linker part must merge loose ends among the target substructures. It initially looks for objects that are associated to the same source object and are of the same type. Then it merges these target objects.

The mapper generator (STONE) produces an Object-to-Object Mapper that translates objects of the source language to objects of the target language. STONE generated mappings can be (semi-) automatically reversed. Correspondingly STONE produces the following parts:

- a matcher process – finds the matching source subgrammar at a specified node in the object structure. The matching process can start at the root and traverse the object structure in some order, or randomly pick objects out of the structure until all objects have been mapped.
- a producer process – creates target substructures and links symbols in the target substructures to symbols in the source object structure. This process demands at least class definitions for the objects to be created, and a library of pairs of source and target subgrammars
- link instructions connecting source and target symbols (two-way linking)

Although a local mapping strategy could be used, STONE adapts a global control of the mapping process. The generated matching process is an object structure traversal that finds the right objects in the structure. Constructions is performed according to a library of source/target subgrammars and the resulting target trees are maintained in a list. Connections are performed by traversing the list of target subtrees.

3.4 Object Structure (OS)

An object structure can be considered as the parse tree of a file over a certain grammar. In addition, such a structure can contain user added information. An object structure consists of objects (nodes) linked with each other through structural links. Every node corresponds to an instance of a nonterminal or terminal appearing in the file.

With an OS we associate a (physical) file, a representation (grammar), time stamp, etc. Representing objects in an OS is based on a general production/symbol class of which all objects (nodes) are specified versions. Defaults (e.g., name, number of children, children) can be overridden. In spite of this traversals and matching can take advantage of the general class definitions, i.e., every object is similar enough to enable matching. On the negative side, object types must be determined dynamically.

4 ALCHEMIST transformation design support tools

In addition to the facilities to specify and generate the actual spell modules, the ALCHEMIST environment offers tools for specifying the objects structures used by spells and user interfaces needed for semi-automatic translations, a mapping library for reuse of designed mappings, etc. ALCHEMIST is independent of the OMS used in the sense that it provides an Object Manager which uses an object interface that can be redefined (both for the generated spells or the

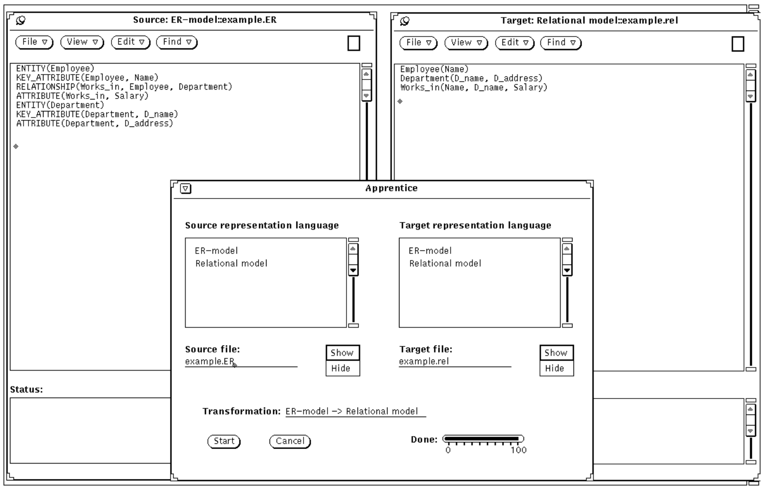


Figure 4: APPRENTICE --- interface to casting spells.

ALCHEMIST itself). Most of these components are used if the default data structures are not satisfactory, e.g., a more complex user interface is needed for semi-automatic translations. Here these facilities are only briefly discussed.

Object manager (POT): The object manager handles the object-oriented spell specification, i.e., loading, saving, and updating of persistent objects. The object dataface (data interface) manager lets the user specify the interface to an underlying object management system (OMS). The dataface allows the user to specify how objects actually are loaded, saved, and updated in the OMS. This solution simplifies the building of interfaces to different object management systems.

Object structure generator (SUBSTANCE): The user can modify the structure of objects produced in parsing or mapping using the SUBSTANCE tool. Since arbitrary modifications are allowed, they should be used with extreme care as they will make the update of the parsing and the mapping processes very difficult.

Specifying the interactive user assistance code (GLASS): The user can assist the system through a windowing interface or a text-based interface. The specification consists of three parts.

- Question specification. The question can be parameterized, depending on input tokens.
- How the input is entered by the user.
- Which actions should be taken due to the input.

The GLASS windowing interface is specified through using DevGuide [14].

Mapping library (BOOK): The BOOK library is based on existing (or predefined) spell specifications. The user can reuse any part of a specification. In addition, also partial spell specifications can be provided, showing how a particular concept is usually mapped for a particular representation. The user can reuse this specification or further specialize it.

Agenda (GUARD): The Agenda is a "metafacility" for a naive user. Specifying a spell typically consists of several subtasks that must be performed in a particular order, such as:

1. Specify first grammar.
2. Generate parser of first grammar.
3. Specify second grammar.
4. Generate parser of second grammar.
5. Specify mapper.
6. Generate mapper.

Every subtask can be further divided into subsubtasks. GUARD keeps track of performed subtasks and inserts new ones in a to-do list. GUARD also notifies the user of what to do, and what to update due to a modification in a specification.

Using spells (APPRENTICE): APPRENTICE provides a user interface for using the spells (transformations) generated by ALCHEMIST. A typical spell performs the following actions. The File-to-Object Parser (FtOP) reads and parses a file (File) constructing an object structure (OS) of the file contents. The Object-to-Object Mapper (OtOM) transforms the object structure into another object structure (over another grammar). Finally, the Object-to-File Parser extracts a file representation from the object structure and writes the representation to a file (see Figure 2).

The user executes a spell through the APPRENTICE window based interface, which allows selection of the source and target representations graphically (see Figure 4).

The user must also specify the names of the source and target files. For convenience, the user can open the files and follow the transformation process (in case of syntax errors, etc.). In the present form APPRENTICE does not yet support creation of macro spells, i.e., the "pipe" operation is not yet implemented.

5 Conclusion

ALCHEMIST is still under development, but the core functionalities have been implemented. The experience so far has clearly shown that the "Unix-style" toolkit approach has proved to be successful and allows an evolutionary development of the complex software involved. The full ALCHEMIST version offers a powerful tool for providing a degree of interoperability between existing software components, regardless of their nature (database software, text formatters, name servers, etc.). Editing ALCHEMIST specifications offers a convenient way to maintain data interchange capabilities regardless of changing versions of the primary application software version changes. However, ALCHEMIST does not aim at being a toolkit for a regular end-user of primary applications—its use requires basic understanding of context-free grammars as well as moderate programming skills for semantic actions.

References

[1] ACM. *Report of the CODASYL Data Base Task Group*, April 1971.

[2] R. Agrawal and N. H. Gehani. ODE (object database and environment): The language and the data model. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 36 – 45, June 1989.

[3] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 2(9):171–216, August 1985.

[4] E. Codd. A relational mode for large shared data banks. *Communications of the ACM*, 13(6):377 – 387, June 1970.

[5] ECMA (European Computer Manufacturers Association). *Standard ECMA-149. Portable Common Tool Environment (PCTE). Abstract Specification*, 2nd edition, June 1993.

[6] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report 32, AT & T Bell Laboratories, Murray Hill, N. J., 1975.

[7] 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.

[8] M. E. Lesk. Lex — a lexical analyzer generator. Technical Report 39, AT & T Bell Laboratories, Murray Hill, N. J., 1975.

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

[10] E. Motta, N. Shadbolt, and A. Rouge. VITAL software technology for embedded knowledge based systems technology. To appear in IEEE Software, November 1993.

[11] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36 – 56, Fall 1991.

[12] T. Payton, S. Keller, J. Perkins, S. Rowan, and S. Mardinly. SSAGS: A syntax and semantics analysis and generation system. In *Proceedings of the IEEE Computer Society's Sixth International Computer Software and Applications Conference, Chicago*, pages 424 – 432, November 1982.

[13] T. Reps and T. Teitelbaum. *The Synthesizer Generator. A System for Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[14] SunSoft. *OpenWindows Developer's guide 3.0. User's Manual, revision A*, November 1991.

[15] M. Tofte. *Compiler generators: what they can do, what they might do and what they will probably never do.* Springer-Verlag, 1990.