ALCHEMIST: A General Purpose Transformation Generator

Greger Lindén
Henry Tirri
A. Inkeri Verkamo

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF HELSINKI
FINLAND

REPORT C-1995-43

University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1995-43

ALCHEMIST: A General Purpose Transformation Generator

Greger Lindén
Henry Tirri
A. Inkeri Verkamo

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

Helsinki, September 1995

Computing Reviews (1991) Classification: D.3.4, D.2.2
Helsinki 1995
Yliopistopaino

ALCHEMIST: A General Purpose Transformation Generator

Greger Lindén
Henry Tirri
A. Inkeri Verkamo

Department of Computer Science, University of Helsinki
P.O. Box 26, Teollisuuskatu 23, FIN-00014 University of Helsinki, Finland
Report C–1995–43 , Helsinki, September 1995
37 + 7 pages

Abstract

ALCHEMIST is a general purpose transformation generating environment which supports specification, generation, and execution of data transformations. ALCHEMIST allows an abstract specification of the transformation through a window-based interface and supports the generation and compilation of transformation program code from these specifications. Unlike compiler-compilers, ALCHEMIST is intended to automate building transformations between two complex representation formats and is thus especially suitable for constructing transformations between database tools, CASE tools, graphical editors or text formatters. In this paper we will describe the design principles and the structure of ALCHEMIST, and demonstrate its use. We will also discuss our experiences with several example transformations and present a real-life case study of using ALCHEMIST for interfacing between two software development environments.

CR Categories and Subject Descriptors:

D.3.4 [Programming Languages]: Processors — Translator writing systems and compiler generators.
D.2.2 [Software Engineering]: Tools and Techniques — Computer-Aided Software Engineering (CASE).

General Terms: Design.

Additional Key Words and Phrases: transformation generator, data transformation, structured transformation, tree transformation

Contents

1 GENERAL **1**

2 THE TRANSFORMATION GENERATOR ALCHEMIST **3**

2.1 Introduction to ALCHEMIST . . . 3

2.2 ALCHEMIST structure . . . 4

2.3 Spell construction . . . 7

2.4 Specification phase: specifying spells with SPELLBOUND . . . 8

2.4.1 Specifying the grammars . . . 8

2.4.2 Specifying the mapping . . . 9

2.5 Generation phase: generating the spell . . . 15

2.6 Compilation phase: compiling the spell code . . . 15

2.7 The ALCHEMIST platform . . . 17

3 EXECUTING A SPELL — RUNNING A TRANSFORMATION **17**

3.1 Creating macro spells with preprocessing and postprocessing . . . 21

4 CASE STUDIES IN THE USE OF ALCHEMIST **21**

4.1 Examples . . . 21

4.2 Interfacing software development environments with ALCHEMIST . . . 22

4.3 Some observations . . . 29

5 RELATED WORK **31**

6 CONCLUSION AND FUTURE WORK **34**

A AN EXAMPLE SPELL SPECIFICATION **A-1**

A.1 ODL grammar (source grammar) A-1

A.2 SQL grammar (target grammar) A-1

A.3 ODL to SQL mapping specification A-2

A.4 Notes on the example spell A-4

B TT-GRAMMARS **B-1**

1 GENERAL

For any computer user data transformations are everyday issues. Even the integrated software packages such as Microsoft Office perform data transformations to allow data interchange between various applications. In fact one of the main issues in integration is the support provided by the software designers as the transformations are embedded in the individual tools themselves so that the users do not have to be concerned about the transformations. As the hope to achieve a single standard format for all data (even in one application area) has proven to be unreasonable, commercial tools typically support several different representation formats incorporating the necessary transformations already in their code. For such tightly integrated environments it is also possible to provide, at least to some extent, online propagation of changes. This means that some of the transformations are "online" as opposed to the "offline" type of transformations one uses for example when importing or exporting data files from an application.

However, in many cases one is not fortunate enough to be able to use only integrated software. The present trend to provide open software environments and heterogeneous software platforms has dramatically increased the importance of developing offline transformations between different applications. Even in the more mature subareas (e.g., image processing) several standard representation formats exist. With the creation of each new representation format comes the need to translate between this new format and the old related representation formats. A good example are the structured text and hypertext representations (e.g., HTML) which have suddenly become more important due to the fast development of Internet, causing a dramatic increase in the need of translating various document formats into "WEB-format". In addition to this emergence of new canonical formats there also exist fields which almost completely lack strict formats. For example complex data or knowledge base domains, where the data is more structured, do not have commonly agreed persistent representation formats — a fact that is probably noticed by anybody who has tried to convert a database from one database management system to another. Even translation between various data definition languages (DDL) in relational databases can be a complex task. Consequently the need for transformations is ever increasing in the current trend towards more and more distributed platforms and new ways to store data.

It is interesting to observe that in the past decades considerable amount of research has focused on automating the construction of compilers with compiler-compilers. On the other hand very little has been done to build supporting tools for general transformations, where the target language, i.e., the representation which is the result of the transformation, can have an equally or even much more complex syntax (and semantics) than the source language. In this general transformation setting it is not enough to provide parser and lexical analyzer generators, one needs to address questions such as how to guarantee that the generated representations are syntactically correct, or how to provide user interfaces for resolving ambiguous transformation instances. Such interfaces are needed as it is evident that in some cases fully automatic transformations cannot be developed, due to the semantic mismatch between the representations. A typical example for such a case would be a transformation from a nontyped representation to an object-oriented representation where there is no way of automatically deciding whether a translated object is a class or an instance.

Naturally one always has the possibility to build a tailored transformation for the task at hand. However, in most cases this is a tedious task resulting in code, which is neither easily modifiable to accommodate changing representations, nor does support a reverse transformation between the representations. In addition, no software reuse is possible as the existing "hand-made" transformation usually does not help in creating other transformations with the same target representations, a very common need when de facto standard formats such as HTML exist.

In this paper we advocate that transformation generators offer an appealing alternative to writing individual transformation code each time a new transformation is needed. When using such a generator, much of the code writing can be automated and, more importantly, many parts of the abstract transformation definition can be reused in later transformations. Together with the fact that a generic transformation structure can be easily defined, this reuse makes the transformation generator approach very promising from the software engineering point of view. In this paper we will discuss our experiences of developing and using the transformation generating environment ALCHEMIST [TL94]. ALCHEMIST is a set of software modules, i.e., a toolkit, to assist the user in developing offline transformations addressing the needs identified in the previous discussion. It was originally developed for providing data interchange facilities between a large set of tools within an experimental software engineering workbench, but quickly developed to be a general purpose transformation generator with applications outside the original scope of the software.

We have organized our presentation as follows. We will first proceed by giving an introduction to the ALCHEMIST environment, where we describe how the transformations can be defined as well as executed. This is partly done by a "walk-through" in a simple example case to illustrate the concepts used in the ALCHEMIST specification. In the latter part of the paper we also discuss the extensive experience we have acquired in building and using ALCHEMIST for several transformations ranging from small examples we used to get feedback for the design, to a fully operational complex transformation between two software environments. Finally we will give a conclusion and discuss some of the topics for future work.

2 THE TRANSFORMATION GENERATOR ALCHEMIST

2.1 Introduction to ALCHEMIST

Any common computer program that manipulates data relies on a persistent data representation. Depending on the functionality of the program in question, the format of this representation varies considerably. Word-processing software stores textual data together with coded formatting information, graphical editors store a linear representation of the graphical representation defined by the user, and databases store structured data using formats supporting space efficient storage and fast access to large volumes of data. These, often conflicting goals result in a vast number of different representation formats. On the other hand interoperability and the modern trend towards open software environments require interchange of data between different software modules. Heterogeneity of the software and hardware platforms hinders any large scale attempts for tight integration and thus results in an increasing need for transformation modules that allow data exchange between various user tools.

Traditionally this data interchange has relied on tailored, manually built transformation modules. Designing such transformations has proven to be a tedious and error-prone task. However, it is clear that at least the offline transformations, i.e., persistent-to-persistent representation transformations, share a common generic structure (Figure 1). In addition, in many cases there exists a need for several closely related transformations, for example to/from a common representation format such as HTML. And finally, in many instances also reverse transformation support is needed, e.g., between design tools that support an iterative design process. All these reasons indicate a need to reuse existing transformation code. Such reuse requires a more abstract way of defining transformations. Transformation generators offer a solution for this purpose.

The transformation generator environment ALCHEMIST provides a sophisticated way of developing transformations between any well-defined persistent representations. ALCHEMIST is based on the idea that the user defines the syntactic structure of the data representations and the related structure associations using a grammar notation. Such specification defines the way to transform source parse tree structures into target parse tree structures. These representation grammars can then be augmented with semantic operations to perform more complex transformation operations such as counting or symbol generation. From this specification ALCHEMIST forms a persistent object-oriented representation which is used to generate transformation code automatically. Analogously the generated transformation program uses an object representation of the parse tree (Figure 1). Thus the approach adopted relies heavily on persistent object repositories, as in a real world transformation the intermediate object representations are too large to fit in the main memory. In our design we have only considered offline transformations, that is, from persistent to persistent representation translation. The questions that have to be addressed in online transformations have additional complexities as one needs to access internal run-time data structures, which typically violates the assumption of providing transformations without modifying the code of the tools themselves. For commercially available tools such modifications are not even possible. An example of the ALCHEMIST general interface is shown in Figure 2.

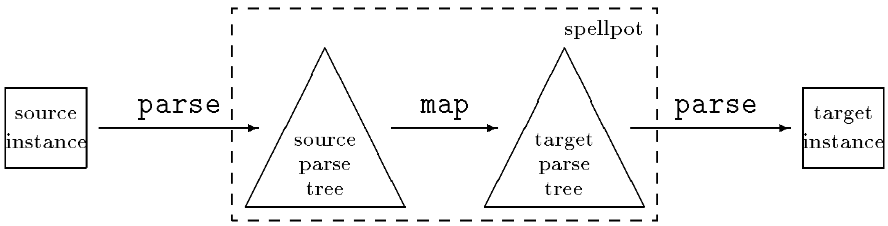

Figure 1: Data flow of a spell.

The ALCHEMIST transformation generator environment consists of several interacting software modules. The environment tools are intended to be usable also independently, thus all the specification and generation modules use well defined text file interfaces. ALCHEMIST follows the general guideline that all objects designed by the user have a linear textual representation. These two underlying design principles allow the user to add or use his own tools in conjunction with ALCHEMIST modules, and offer the possibility to use regular editors for editing ALCHEMIST generated representations. ALCHEMIST reuses UNIX tools such as the `yacc` parser generator [Joh75] and thus it is possible to directly import `yacc` definitions if such specifications exist.

In the following we will first give a short overall description of the software environment and then illustrate the use of ALCHEMIST with a "design walk-through" in a simple example case. The discussion on our experiences in the case of real life transformation examples is postponed to the later sections of the paper.

2.2 ALCHEMIST structure

The general structure of the ALCHEMIST environment is presented in Figure 3. (The dashed line indicates components not yet implemented in the current version of ALCHEMIST.) It consists of modules supporting the three main functionalities of ALCHEMIST: specifying transformations, generating transformation code and linking system related components. In addition to these main modules the ALCHEMIST environment also provides a transformation agent module called APPRENTICE, which allows the user to call different transformations through a single graphical interface. Such an agent is useful for example in software workbench related transformations, where several pre-existing tools need tool-to-tool transformations in a predefined sequence. Since such transformations were one of the original design motivations for ALCHEMIST, a special module was included for this purpose.

As it is easy to get confused between the meta-level, i.e., specification of individual trans-

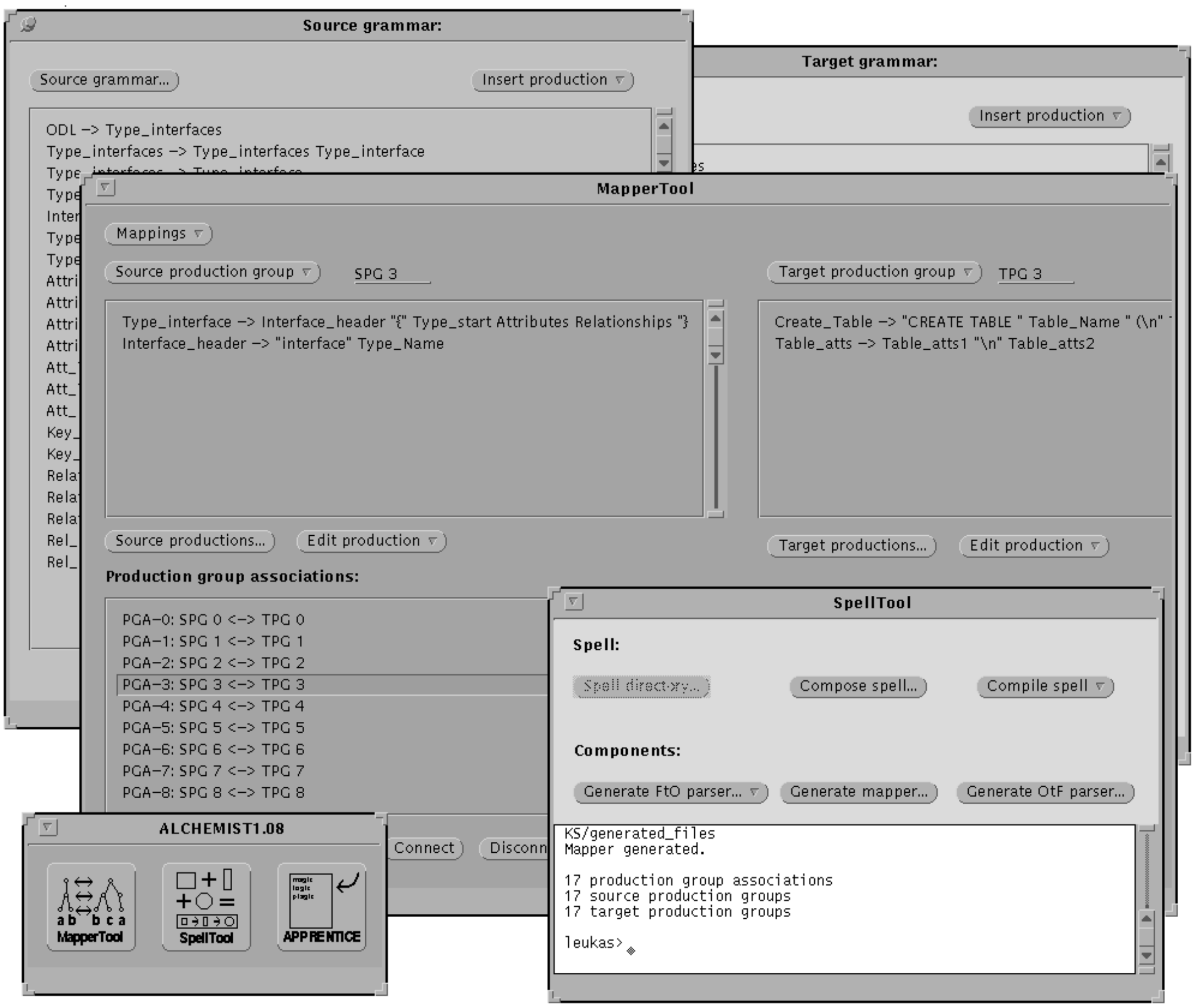

Figure 2: Snapshot of ALCHEMIST.

formation components, and the transformation level, i.e., execution of a generated transformation with an input file, we have adopted the convention to name the meta-level components from the alchemy domain with capital letters (see Figure 3), and call individual transformations *spells*. Consequently the ALCHEMIST environment consists of the following components:

- SPELLBOUND. The user interface to specify spells. It is based on an OpenWindows OPEN LOOK interface and consists of one specification tool, MAPPERTOOL. MAPPERTOOL is the interface to specify the syntax and semantics of the spell.
- SPELLTOOL. The user interface to generate and compile all the components of the spell.
- SUBSTANCE. The user interface to specify object structures used to store spell

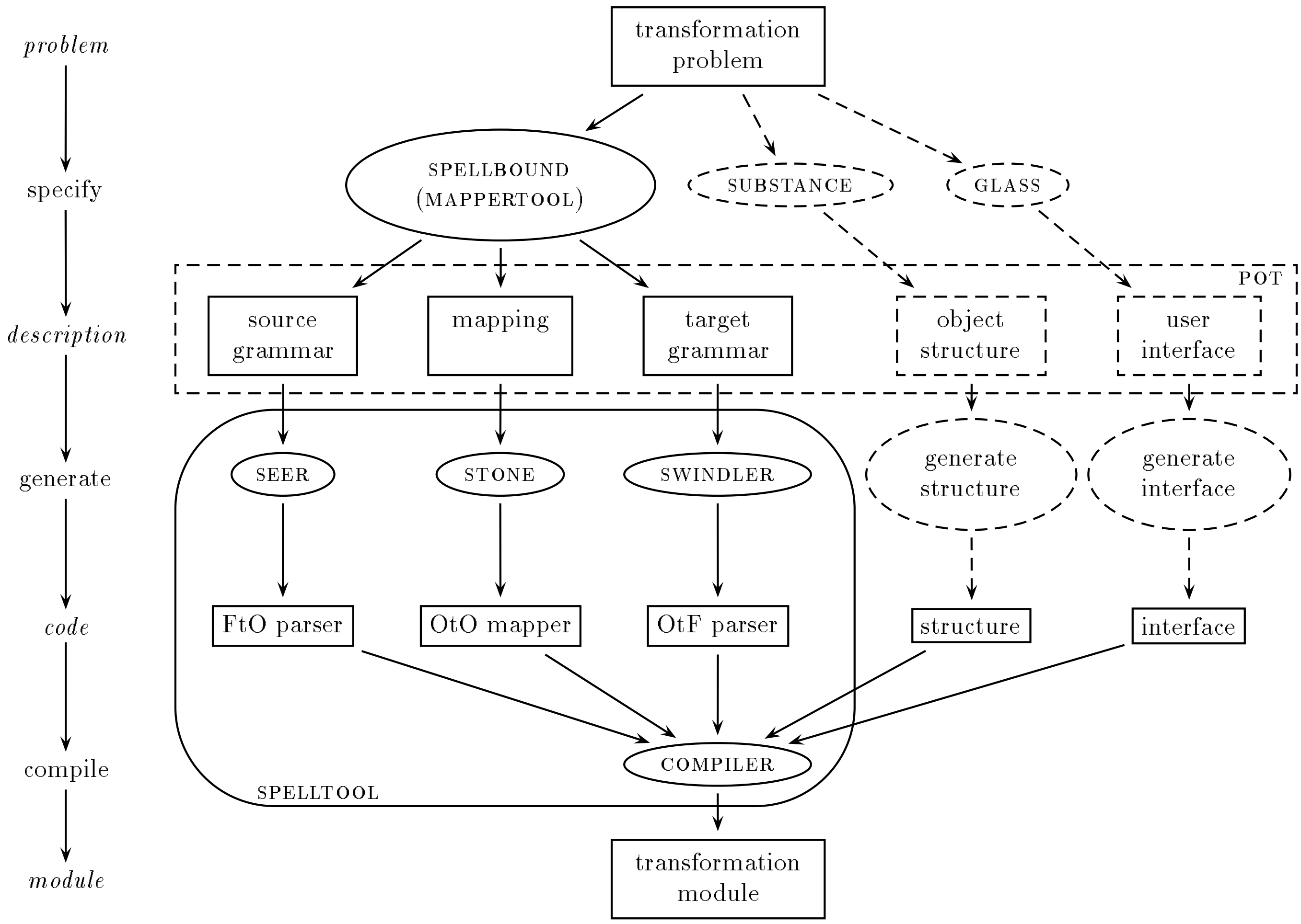

Figure 3: The ALCHEMIST environment structure.

objects (e.g., parse tree nodes). If the user does not specify these object structures, ALCHEMIST uses default values.

- GLASS. The user interface to specify the spell user interface. If the transformation cannot be performed fully automatically due to incomplete semantic information, the so called *semi-automatic spells* need to have an interactive user interface to resolve the ambiguities. GLASS allows the user to specify the look-and-feel of such an interface. If such an interface has not been defined, and the need arises, the spell uses the default GUI.

- SEER. A file-to-object (FtO) parser generator. SEER generates the source input parser for the spell. This parser reads the source file and creates a parse tree on which the mapper component of the spell operates.

- STONE. An object-to-object (OtO) mapper generator. STONE generates the mapper component of the spell. The spell mapper transforms the source parse tree created by the file-to-object parser and constructs the target parse tree according to the transformation rules specified by the user. This target parse tree is the input for the object-to-file parser.

- SWINDLER. An object-to-file (OtF) parser generator. SWINDLER generates the parser that traverses through the target parse tree and forms a linear representation for the target file.
- POT. An interface to an underlying object management system. POT interfaces ALCHEMIST with an object management system (OMS).

2.3 Spell construction

The transformation specification is based on the user description of the source and target representations as context-free LR grammars [Knu65]. The actual transformation mapping is defined as rules between the productions of these grammars. This approach has been adopted to avoid one of the most difficult problems in transformations: how to guarantee that the generated transformation representation has a syntactically correct structure. In traditional compiler-compiler environments syntactic correctness has not been such an issue as usually the syntactic structure of the target language is relatively simple. For example, the instructions in machine code can be freely permuted and the only syntactic restrictions are related to the well-formedness of the instruction itself. For the general transformations addressed by ALCHEMIST such freedom is not possible.

Technically the ALCHEMIST mapping specification is based on the formal notion of TT-grammars [KPPM84]. Intuitively a TT-grammar can be viewed as a mechanism for generating subtrees over the target grammar from subtrees over the source grammar. However, in most cases purely structural transformations are not sufficient. In many cases one needs to evaluate complex attributes related to the syntactical structures, and for such operations no fixed predefined set of operations would be powerful enough to handle all the possible needs. Thus in the specification of the transformations the full expressive power of C++ is allowed in so-called "semantic actions". ALCHEMIST design has an object-oriented flavor, thus the whole specification of a spell is stored into an object-management-system using the POT interface. Using a special interface for persistent storage makes ALCHEMIST "OMS-independent", in fact we have experimented with both a flat file system and the ODE object-oriented database system [AG89]. Since all ALCHEMIST retrieval and store calls are through this interface, changing an underlying OMS only requires writing the interface code for POT.

The transformation specification discussed above is used by ALCHEMIST to generate the transformation code. The generated result includes C++ code for a file-to-object parser, an object-to-object mapper, and an object-to-file parser all linked together with a user interface. ALCHEMIST also allows the user to generate spells with a textual user interface in case the spell code needs to be embedded in a larger software module. These textual spells only use standard input and output.

To produce a spell with ALCHEMIST we have three major design phases, typically allowing iteration between them (Figure 3):

1. *Specification phase.* Specify source and target grammars, and a mapping between these grammars with MAPPERTOOL and editors.
2. *Generation phase.* Generate a file-to-object parser, an object-to-object mapper, and an object-to-file parser (in C++) from the specifications given in the specification phase.
3. *Compilation phase.* Compile the parsers and the mapper, and link them together with the user interface, and object representation code into a spell.

In the following sections we describe each of these three phases in more detail. For a more thorough explanation of the use of ALCHEMIST, see the user guide [LT95].

2.4 Specification phase: specifying spells with SPELLBOUND

The first phase of building a spell includes giving its abstract specification. As discussed above, the structure of the source and target representations is defined through context-free LR grammars [Knu65] and a mapping between these grammars based on the TT-grammar technique [KPPM84]. In the following we assume that the reader is familiar with the basics of the compiler technology and understands the elementary notions of context-free grammars such as productions, terminals, etc. An uninitiated reader is referred to any of the widely available compiler design text books, such as [FL88]. We will first briefly discuss some issues related to the techniques for grammar and mapping specification and then illustrate the approach by a walk-through of a simple example.

2.4.1 Specifying the grammars

The source representation and the target representation are specified with context-free LR grammars, a *source grammar* and a *target grammar*, respectively. The grammars follow the conventional simple syntax. A grammar consists of a set of productions. Every production has a left hand side which is a nonterminal, and a right hand side consisting of a sequence of nonterminals and terminals. A nonterminal is an identifier. A terminal is either a constant string surrounded by double quotes or a special identifier. Special identifiers used in ALCHEMIST are `IDENTIFIER`, `NUMBER`, `BINARY_NUMBER`, `IDENTIFIER[n]`,

NUMBER[n], BINARY_NUMBER[n], STRING, CHAR[n], NULL[m], CURRENT_DATE, CURRENT_TIME, and INCLUDE_FILE which all have natural semantics.

The use of the special identifiers is not symmetric: only some special identifiers can be used in source grammars, e.g., IDENTIFIER, NUMBER, and STRING, for parsing identifiers, numbers or constant strings. On the other hand all special identifiers can be used in target grammars. In particular, to be able to produce certain special strings or identifiers we have: CURRENT_DATE, and CURRENT_TIME which produce the current date and time, respectively; IDENTIFIER[n], NUMBER[n], BINARY_NUMBER[n] which produce identifiers and numbers of a certain length; CHAR[n] which produces the character corresponding to the ASCII code n; and NULL[m] which produces m NULL characters. The INCLUDE_FILE identifier is convenient for including text directly from a file.

ALCHEMIST grammars also provide the common construct operators. Iterations are specified with recursive productions, whereas disjunctions are specified by giving several productions with the same left hand side symbol. If a double quote is part of a string it is repeated once. An example of an ALCHEMIST grammar is given in Figure 4. It should be observed that as opposed to the initial version of ALCHEMIST [TL94] the current version does not support the specification of grammars. We found it more natural to use regular editors (such as emacs) for writing the grammars as ordinary text files, instead of the somewhat cumbersome graphical editor used in the first version of SPELLBOUND.

2.4.2 Specifying the mapping

As expected, the mapping specification is the most difficult and time consuming part of the spell specification process. A mapping specification contains rules which specify how source grammar structures are related to target grammar structures. This specification is done through the MAPPERTOOL which is part of the ALCHEMIST SPELLBOUND interface (Figure 2). The MAPPERTOOL implements a subset of the TT-grammar technique [KPPM84] mentioned before.

A *TT-grammar* [KPPM84] is a formal description technique for describing transformations from one language to another. A TT-grammar describes a relationship between a syntax tree over a grammar G_1 and a syntax tree over a grammar G_2. 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 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.

Let us assume that we have two different representations, i.e., a source and a target representation. Our purpose is to build a transformation module changing each instance of the source representation into an instance of the target representation. As discussed in the previous section, we first define the structures of these representations with two grammars, a source grammar and a target grammar.

A mapping is specified by defining a set of *production group associations*, each association consisting of a *source production group* and a *target production group.* A source production group consists of a subset (subgrammar) of source grammar productions (source productions), and *a target production group* consists of a subset of target grammar productions (target productions). We require that the source production groups have a unique start symbol from which all the other symbols in the group must be derivable. Intuitively, this can be seen as describing source tree patterns to be matched against and target tree patterns to be generated. A spell will search for these source tree patterns in the source parse tree and produce a forest of target subtrees according to the corresponding target production groups.

Every production group association can contain *symbol associations* between source and target symbols. A symbol association links a source symbol to a target symbol within the same production group association. The reason for these associations is twofold. Firstly, in the case of identifiers, we need a way of transferring the identifier strings to the target, i.e., we map source identifiers to target identifiers. Secondly, the ALCHEMIST mapping rules will produce a set of disconnected target parse trees. These trees must be connected to form the final target tree. The linking of these subtrees is done through the symbol associations: if two target nodes have the same label and if they are associated to the same source symbol, they actually denote the same node; in this case the nodes are united to form a larger target subtree. After the production of target subtrees, this linking process is repeated until the target tree is complete. For a more formal description of TT-grammars, see Appendix B.

In addition, *semantic actions* can be included with each target symbol. Typically, a semantic action performs book-keeping operations such as symbol table checking, or some other type of additional computing.

Example 1 To give an example of designing an ALCHEMIST mapping let us look at a small transformation between the Object Definition Language (ODL) [Cat94] and SQL [CAE+76]. ODL, intended for object databases defines objects with attributes and methods. SQL gives the appropriate definitions for schema description in a relational database system. This example should not be taken as realistic, it is presented only for illustrative purposes. Any realistic example would be unnecessarily complex in its details; this can be seen later on in our case studies.

Let us consider the following example instance of an ODL definition which describes a `Course` object with a `name` and a `number`, and a reference to a `Professor` object associated with the `Course`.

```
interface Course {
        extent courses ;
        key (name) ;

        attribute string name ;
        attribute string number ;

        relationship Professor is_taught_by ;
}
```

We would like this instance to be transformed into the following SQL DDL definition.

```
CREATE TABLE Course (

        Course# NUMBER(8,0)

        name CHAR(20)
        number CHAR(20)

        is_taught_by# NUMBER(8,0)
)
```

Observe that in this example we have decided to create "object identifiers" for all "objects" included in the definition due to the fact that relational databases do not support object identifiers. Our new object identifiers consist of special numbers, not actual reference links. In addition to simplify the specification we have also chosen a default length for identifiers and numbers.

```
ODL -> Type_interfaces ;
Type_interfaces -> Type_interfaces Type_interface ;
Type_interfaces -> Type_interface ;
Type_interface -> Interface_header "{" Type_start Atts Relshps "}" ;
Interface_header -> "interface" Type_Name ;
Type_start -> "extent" IDENTIFIER ";" "key" "(" Key_List ")" ";"   ;
Type_Name -> IDENTIFIER ;
Atts -> ... ;
Rlshps -> ... ;
```

Figure 4: Part of a context-free grammar of the object definition language (ODL).

For illustrative purposes we restrict ourselves to a part of the real ODL grammar (see Figure 4). The example is chosen in such a way that all main features of a grammar description can be used. For example, iteration is described through recursive productions (see the second production). Disjunction is described by giving several productions with the same left hand side (e.g., the second and the third production). The grammar contains constant strings (e.g., `"interface"`) as well as special terminals or identifiers (`IDENTIFIER`), and nonterminals (`ODL`, `Type_interfaces`, etc.). In ALCHEMIST grammars, the first production is always the start production, and its left hand symbol (in this case, `ODL`) is always the start symbol. Intuitively, this grammar tells us that an ODL definition consists of a set of type interfaces. Each type interface has a header, a set of attributes and a set of relationships.

```
SQL -> Create_Tables ;
Create_Tables -> Create_Tables Create_Table ;
Create_Tables -> Create_Table ;
Create_Table -> "CREATE TABLE " Table_Name " (\n" Table_id Atts ")\n\n" ;
Table_Name -> IDENTIFIER ;
Table_id -> "\n\t" IDENTIFIER "# NUMBER(8,0)\n\n" ;
Atts -> ... ;
```

Figure 5: Part of a context-free grammar of the SQL.

The target grammar (see the partial grammar in Figure 5) is similar in form to the source grammar. Intuitively the grammar says that an SQL definition consists of a set of create table definitions. Each table definition contains the table name, its identifier, and a set of attributes. Analogously to the source grammar, the start symbol is the left hand symbol of the first production (i.e., `SQL`).

We define the transformation mapping by choosing corresponding subsets of productions and symbols from these two grammars. As a first rule in constructing mappings, we always associate the start symbols of the grammars. In our example each `ODL` definition corresponds to a `SQL` specification.

Let us examine these principles by giving the first three rules of the mapping and examining their semantics. We denote production group associations by numbering them and symbol associations in the target production groups by `input_symbol.output_symbol`. These rules are the following:

```
1   ODL ->                          ODL.SQL ->
       Type_interfaces                 Type_interfaces.Create_Tables


2   Type_interfaces ->              Type_interfaces.Create_Tables ->
       Type_interfaces                 Type_interfaces.Create_Tables
       Type_interface                  Type_interface.Create_Table
```

```
3   Type_interfaces ->                Type_interfaces.Create_Tables ->
       Type_interface                    Type_interface.Create_Table
```

We see that the source production

```
ODL -> Type_interfaces
```

corresponds to the target production

```
SQL -> Create_Tables
```

This has the following semantics. Every time the source production has been applied in the source tree, the mapping produces a target subtree based on the target production (Figure 6). In addition to this transformation rule we have also associated the symbols `ODL` and `SQL` with each other as well as the symbols `Type_interfaces` and `Create_Tables` with each other. In the same way, applying the production

```
Type_interfaces -> Type_interfaces Type_interface
```

produces the target subtree corresponding to the production

```
Create_Tables -> Create_Tables Create_Table
```

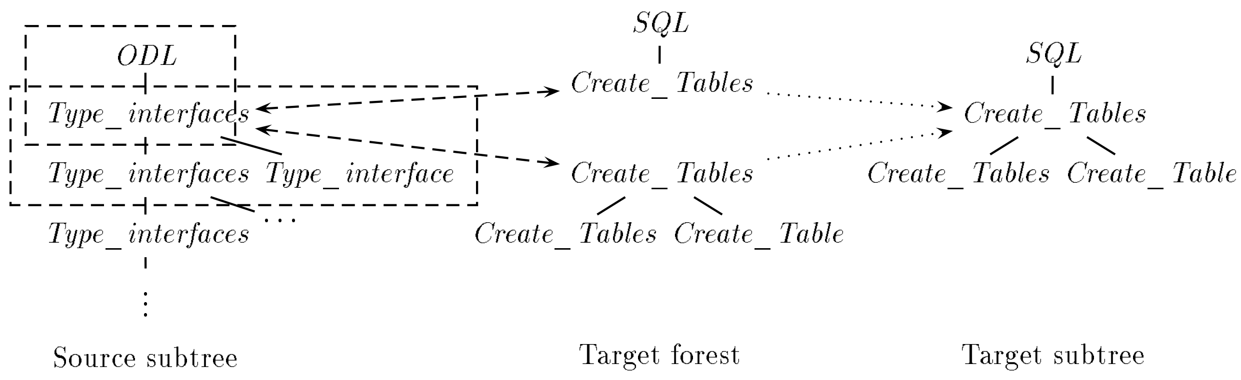

Figure 6: Part of transforming the source parse tree.

As a result we now have two disconnected target subtrees in a target forest (Figure 6). However, both nodes labeled `Create_Tables` are connected through symbol associations to the same instance of a node labeled `Type_interfaces`. In the final phase of the transformation these two nodes will be substituted by one single node with the same label and thereby the two subtrees will be connected to form a single target subtree (Figure 6).

When we associate a source identifier (e.g., in source production 7) with a target identifier (in target productions 5 and 6), we know that every instance of this source identifier (e.g., `Course` or `Professor`) has to be copied to the target side. This is the normal way of handling instance dependent symbols.

The next two rules of the mapping are:

```
4   Type_interface ->                  Type_interface.Create_Table ->
       Interface_header                   "CREATE TABLE "
       "{"                                Type_Name.Table_Name
       Type_start                         " (\n"
       Atts                               Type_Name.Table_id
       Rlshps                             Type_interface.Table_atts
       "}"                                ")\n\n"

    Interface_header ->                Type_interface.Table_atts ->
       "interface"                        Atts.Table_atts1
       Type_Name                          "\n"
                                          Rlshps.Table_atts2


5   Type_Name ->                       Type_Name.Table_Name ->
       IDENTIFIER.IDENTIFIER              IDENTIFIER

                                       Type_Name.Table_id ->
                                          "\n\t"
                                          IDENTIFIER.IDENTIFIER
                                          "# NUMBER(8,0)\n\n"
```

Here we describe how the type interface corresponds to a create table definition and especially how the type name is copied to target symbols. In particular one should note these production association groups, which contain production groups with several productions, and also how we can choose to copy an identifier several times to the target side.

All the specifications above are made in the MAPPERTOOL and its symbol associations window (Figure 2). For more details, the complete example is given in Appendix A. A detailed description of how the mapping is specified with the SPELLBOUND user interface is given in the ALCHEMIST user guide [LT95].

2.5 Generation phase: generating the spell

The second phase of the spell construction includes generation of the code for the transformation module. In principle the choice for the generated language platform is independent of the abstract spell specification. The only exceptions from this principle are the semantic actions which have to be given in the generated language. However, the current version of ALCHEMIST uses several standard UNIX environment tools and thus generates C++ code for a file-to-object parser, an object-to-object mapper, and an object-to-file parser as components of the spell.

As Figure 3 illustrated, each of the spell main components, the file-to-object parser, the object-to-object mapper and the object-to-file parser are generated by a corresponding ALCHEMIST component. The file-to-object parser is generated by SEER from the source grammar. Since ALCHEMIST uses the parser generator `yacc` to produce parsers, it generates code that can be directly read by `yacc`. The basic `yacc` code is also augmented by actions for constructing parse trees, etc. Hence the actual parser generation is performed by `yacc` itself. As mentioned above, the function of the file-to-object parser is to parse the source file and build an internal source parse tree.

The object-to-object mapper is generated by STONE from the source and target grammar files and the mapping specification the user has produced with the MAPPERTOOL. The function of the object-to-object mapper is to construct the target parse tree from the source parse tree by applying the mapping rules given by the user. This code is generated from the production group association definitions with the optional augmenting code from the semantic actions.

Finally, the object-to-file parser is generated by the SWINDLER from the target grammar. The function of this parser is to traverse the target parse tree and write its contents into a file.

Following the ALCHEMIST design principles, any part of the above mentioned generated code can be inspected and edited, should the need arise, by any common text editor. This feature allows ALCHEMIST to be used also as a "transformation template generator" for semiautomatic transformation design.

2.6 Compilation phase: compiling the spell code

Once all the code for the spell is generated, it can be compiled and linked with standard system code modules, which provide for example the default window interface for the spell. In this phase also the object repository interface (spellpot) code is added to the spell main modules. Since both the source and target parse tree nodes are implemented as objects, the memory usage of a spell with a large input file can easily exceed tens of megabytes.

Therefore in the general case the parse trees need to be stored into a repository which relaxes the virtual memory limitations posed by the platform. If it is known in advance that the input file sizes are moderate, ALCHEMIST allows also compilation of main memory based spells which do not use any repository. In main memory spells the reduced size of accepted input files is traded for increased transformation speed as no object accesses to the repository are required.

All ALCHEMIST generated code is C++ code which can be compiled with a standard C++ compiler. For our experiments we have used both the AT&T C++ compiler and the Gnu `g++` compiler – ALCHEMIST code is compatible with both of them. In the compilation phase the user can choose whether to produce a window-based or a textual version of the spell. The functional body for both of the spells is the same, the difference lies in the interface definitions. The window-based spell (Figure 7) has a point-and-click windowing interface for invoking the spell. In addition it allows the user to add the optional preprocessing and postprocessing modules via the graphical interface. The textual spell can be called from command level, which makes it easy to be embedded in a larger piece of software. The textual version of the spell only uses the standard input and output interface of C++.

An additional functionality offered in the spell compilation phase is the possibility to produce only stand-alone parsers for either the source grammar or the target grammar. This facility is motivated by the fact that when producing new spells, it is usually convenient to test the parsers for both grammars before constructing the mapping. It is good practice to convince oneself that the grammars really correspond to the representations in question, a fact which is usually nontrivial as in most cases the grammars have to be constructed from instance examples in the absence of formal syntax definitions. Therefore the waste of design effort caused by building a mapping between incorrect grammars can be greatly reduced. The compilation phase is implemented by several generic UNIX `makefiles`.

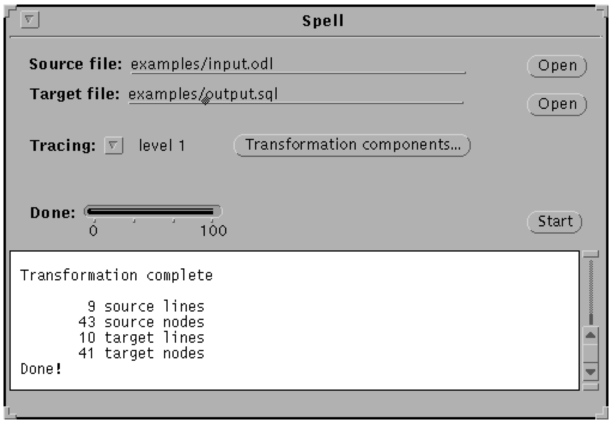

Figure 7: Snapshot of a spell interface.

2.7 The ALCHEMIST platform

The ALCHEMIST environment including the transformation agent APPRENTICE runs on Solaris 2.x (SunOS 5.x) platforms in the OpenWindows/OPEN LOOK environment on Sun machines.

The spells produced by ALCHEMIST use the XView toolkit for their window interfaces and `yacc` for parsing the source. They are compiled with the AT&T C++ compiler or the public domain Gnu `g++` compiler, and they also run on Solaris 2.x (SunOS 5.x) in the OpenWindows/UNIX environment on Sun machines.

3 EXECUTING A SPELL — RUNNING A TRANSFORMATION

In the previous section we have described how ALCHEMIST can be used to specify and generate a transformation, i.e., a spell. In this section we will discuss how the spells are used and take a look at the generic structure of a spell. Some of the spell structure is already evident from the discussion above, but for completeness sake we will repeat also those issues.

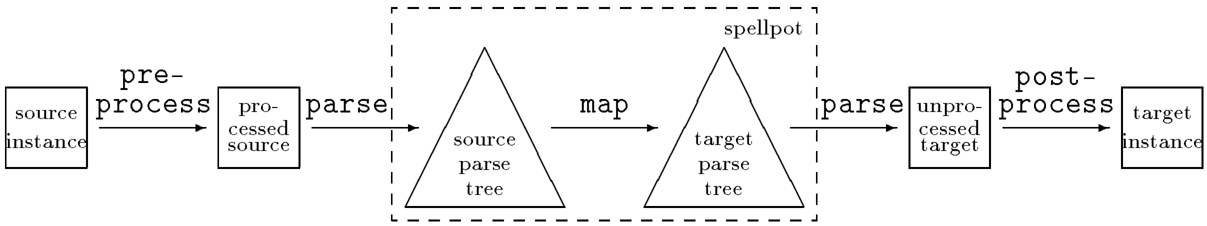

Figure 8: Data flow of a spell with preprocessing and postprocessing.

A spell consists of three main components. The file-to-object parser reads the source file and builds an intermediate representation, a source parse tree over the source grammar. The mapper uses a tree matching technique to find instances of applied source productions in the source tree and produces a target forest, a set of disconnected target subtrees. It also links the target subtrees together to form the target tree. The object-to-file parser traverses the target tree and writes its contents to a file. In addition to these main modules, instructions for preprocessing or postprocessing the source or target file can be included. Consequently a spell consists of the following modules (see Figure 8):

- *Preprocessing commands.* These commands are used to modify the source file into a form which is more susceptible to parsing.
- *A file-to-object parser.* This module parses the source file to be transformed and produces the source parse tree structure.

- *An object-to-object mapper.* This module generates the target parse tree structure from the source parse tree structure by suing the rules given by the user in the form of product group and symbol associations.

- *An object-to-file parser.* This module traverses the target parse tree structure and produces a linear representation of its contents. The result is not necessarily in the proper target representation format.

- *Postprocessing commands.* These commands are used to modify the target file into a final form readable by the programs "understanding" the target representation format.

- *spellpot.* An interface to an object management system motivated by the very large parse trees produced in the transformation.

In the spell execution preprocessing and postprocessing can consist of any set of (UNIX) commands. Though they are not necessary in principle, preprocessing and postprocessing have proven to be extremely useful in cleaning up the source and target files before and after the transformation. This cleaning, for example removing unnecessary control characters, simplifies the specification grammars considerably which in its turn produces more understandable mappings. The actual transformation consists of parsing the source representation (from a file) and forming a source parse tree, mapping the source parse tree into a target parse tree, and parsing (writing) the target parse tree into the target representation (to a file). For portability reasons as well as to support embedded transformations, ALCHEMIST can produce spells with either a window-based or text-based interface.

A typical execution of a window-based spell follows the sequence below:

1. Produce a persistent representation (in a file) of the source instance. Typically this is performed by saving a file from the "source tool" using the format for which the spell is constructed.

2. Initiate the spell execution.

3. Set proper source and target file names in the spell interface.

4. Choose the spell components to be included in the transformation (Figure 9). If preprocessing and/or postprocessing is used, the corresponding commands have to be included either from a library definition or directly to the spell interface.

5. Run the spell. In case of ambiguities answer the questions posed by the transformation module (valid only for semi-automatic spells).

6. Use the persistent target representation with the "target tool".

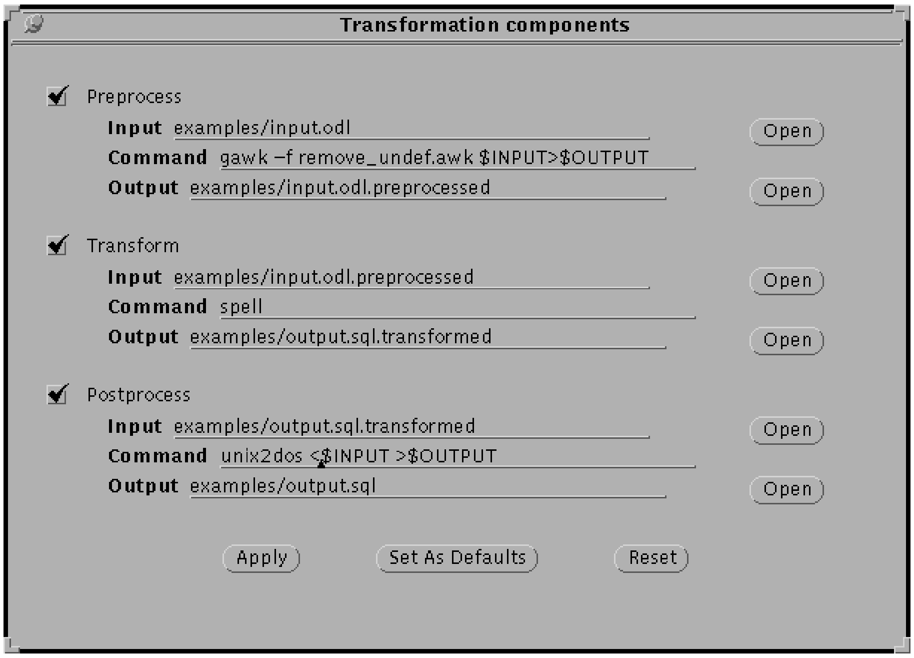

Figure 9: Spell components to be included in the spell.

The user inserts the source and target file names in the spell interface (Figure 7) and clicks the "Start" button to start the spell. The spell components to be used (preprocessing, transformation, and/or postprocessing) can be chosen in a separate window that is opened by clicking the "Transformation components" button. During the spell execution various levels of tracing can be performed. The log produced by the spell execution is especially valuable when a transformation error is debugged during the design of a spell, but it is useful also for book-keeping purposes. The log file allows the user to inspect the correspondence between source and target constructs, which is useful for example when the spell generates identifiers for the target representation (Figure 10).

APPRENTICE is a user interface agent used for grouping a set of transformations, i.e., for using a set of spells. APPRENTICE allows users to select from a list which transformations are to be started (Figure 11). This type of grouping is useful in workbench environments where several tools need interfaces to each other. Grouping the necessary transformations together in a special agent is usually a much more viable alternative than modifying the tools individually with embedded transformation modules (which is even impossible in the case of commercial tools as only binaries are available). As ALCHEMIST was originally developed with such a software workbench in mind, including an APPRENTICE type of an agent in the ALCHEMIST environment is natural.

```
Transforming
Source file: examples/input.odl
Target file: examples/output.sql

     Source id     Type         Target id     Type
  1: Course        Type_Name    Course        Table_Name
  2: courses       Type_start   --
  3: name          Key_list     name          Table_Att_Name
  4: number        Att_Name     number        Table_Att_Name
  5: Professor     Att_Name     --
  6: is_taught_by  Rel_Name     is_taught_by  Table_Att_Name

Transformation complete

       9 source lines
      43 source nodes
       9 target lines
      41 target nodes
      19 matches in source
```

Figure 10: Example of a spell log file.

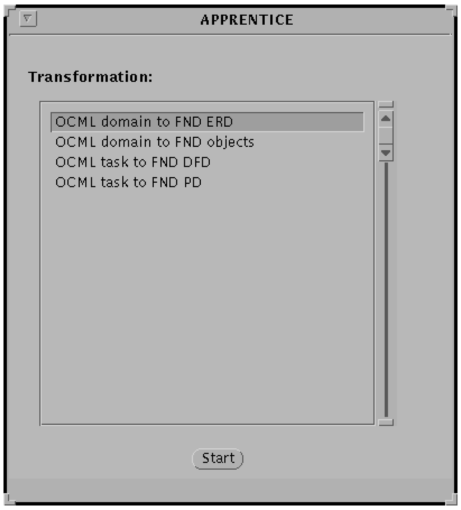

Figure 11: Snapshot of APPRENTICE.

3.1 Creating macro spells with preprocessing and postprocessing

As already mentioned above, preprocessing and postprocessing commands can be included in any transformation. The preprocessing commands will be executed before the actual transformation takes place; the postprocessing commands will be executed after the transformation.

The initial reason to include preprocessing and postprocessing commands was to simplify the source and target representations. Some representations contain excessive or repeated information, which might not even be necessary for the source or target application (for example control characters or comments). This redundant information can be removed before the source is parsed, or added before the target is completed. The grammars involved are simplified, and decreasing redundancy makes it also easier for the user to construct unambiguous grammars.

After their inclusion, preprocessing and postprocessing have proven to be useful for another purpose as well. Since they can include any commands available on the current platform they can also contain other transformation calls. It is therefore possible to construct "macrospells", spells that consist of several component spells.

4 CASE STUDIES IN THE USE OF ALCHEMIST

4.1 Examples

ALCHEMIST has been developed during several years as part of a larger software workbench development [SMR93]. Therefore we have been able to use ALCHEMIST in several case studies to gain experience and to further develop the initial ideas. The case studies we have performed vary in size from small illustrative examples to a full-blown transformation between an in-house developed graphical software design toolkit and a commercial CASE environment. Many of the features in the current version of ALCHEMIST have resulted from the feedback we have received from other users of ALCHEMIST[1].

Our first case studies concentrated on testing ALCHEMIST for building purely syntactic transformations. One of the first of such transformations was a textual transformation editing SGML representations. Although not foreseen at its design time, such simple textual transformations have become important parts of the "Information Superhighway" since documents presented in various text formats are required to be available in HTML

[1] Our special thanks go to the VITAL team at Andersen Consulting Spain: Luis Montero, Angel Luis Muñoz and José Manuel Quesada whose feedback on the prototype software is gratefully acknowledged.

format for the World Wide Web interface. It seems that ALCHEMIST is especially suitable for creating transformations between various text formatting representations, partly because of the conceptual match of the elements to be transformed (paragraphs, sections, itemized lists etc.), partly due to the already existing well-defined syntax. The former reason makes the object-to-object mapping construction more straightforward, the latter reason speeds up the often time-consuming design of the grammars. It is very clear that for this type of transformations the generator approach offers substantials benefits with respect to the manual approach, in particular for the reuse of existing code.

For the first version of ALCHEMIST we chose a test example from the database domain, and we experimented with specifying and generating a transformation from a subset of the ER model to the relational model [TL94]. This transformation required several passes through the source instance to produce the target, and showed that purely syntactic transformations are clearly not sufficient for more complex transformation needs. Part of the experience gained could be transferred to the large software environment transformation design.

We have also built a complex prototype transformation for generating C++ code from a graphical knowledge acquisition tool supporting laddering hierarchies (for knowledge acquisition construction by the ALTO tool, see [MR90]). This work was motivated by the overall aims of the larger project, but proved to be a very good demonstration of the flexibility of the approach we had adopted in ALCHEMIST. Since the ALTO tool was based on defining object hierarchies, the transformation implemented handling of object definitions and (single) inheritance. The transformation produced directly compilable C++ code with proper attribute definitions etc., thus operationalizing the definitions of the somewhat abstract knowledge acquisition tool.

The most thorough case study where ALCHEMIST has been used involved building a transformation between toolsets in two software development environments, namely a workbench for KBS development [SMR93] and a commercial CASE environment containing several tools for the analysis, design, and implementation phases of software development [And93a]. Since this case study was quite extensive in its effort and involved building a transformation for two actual production environments, we will now focus on our experiences in this example giving a more detailed account in the following section.

4.2 Interfacing software development environments with ALCHEMIST

Before describing the transformation itself, it is useful to motivate our initial task of building an interface between two software development environments.

When building a KBS application, skills and experience both in KBS development and more traditional software engineering are needed. Of course one possibility is to embed all the tools required for building the knowledge base and the inference mechanism in a KBS development environment, such as the VITAL workbench [SMR93, DMW93]. However, large parts of the application to be produced are traditional software that is relatively independent of the AI capabilities, and developing these is best assisted by traditional CASE tools. Consequently, to build the entire application, we need to have an interface between the KBS development environment and the CASE environment. Typically, we could start developing a KBS application in the KBS development environment, using the KBS specific tools for knowledge acquisition and conceptual modelling, and then transfer the conceptual model into the CASE environment with more advanced design and implementation tools for designing and implementing the system [Ver94].

In this general framework we have built an interface to transfer information from the VITAL workbench [DMW93] to the FOUNDATION (FND) Design CASE environment of Andersen Consulting [And93b]. More specifically, the source environment is the graphical conceptual modeling tool of the VITAL workbench, OCML (Operationalizable Conceptual Modelling Language), and the target environment consists of several analysis and design tools of FOUNDATION Design, such as tools for producing entity-relationship diagrams, data flow diagrams, and procedure diagrams.

We shall briefly describe the description power of the OCML tool. OCML is used for describing both the information domain and the task (or control flow) structure of the system. The OCML diagrams for domain data and knowledge representation, the *domain diagrams*, consist of graphical elements for concepts, relations, concept instances and relation instances. The left hand diagram of Figure 12 presents a small example containing concepts (ovals) and concept instances (rectangles), where the concepts form a hierarchical structure (e.g., `manager` is a `lab_member`) and there is an instance for each of the lower level concepts (e.g., `eva_i` is an instance of `manager`). The OCML domain diagram corresponds fairly well to the entity-relationship diagrams of FOUNDATION Design (the right hand diagram of Figure 12) [VL95].

The OCML diagrams for problem solving behavior, the *task diagrams*, contain graphical elements for describing hierarchies of tasks and roles to produce or obtain information needed or produced by the tasks. The tasks are connected by control structures, such as sequencing, choice elements, or return elements. The left hand diagram of Figure 13 presents a small example of a task diagram where ovals represent tasks (e.g., `choose role`), diamonds represent choice elements (e.g., `roles_left?`) and rectangles represent roles (e.g., `order`). There is no single diagram in FOUNDATION Design that could represent all the information of the OCML task structure. The flow of information can be described by data flow diagrams and the control structure of the tasks can be described by procedure diagrams of FOUNDATION Design (see the right hand diagrams of Figure 13) [VL95]. The discussion above is illustrative to the complexities of real life transformation problems: what was initially thought as a single transformation turned out to be a set of related transformations.

Therefore, our interface consists of the following three independent but related transformations:

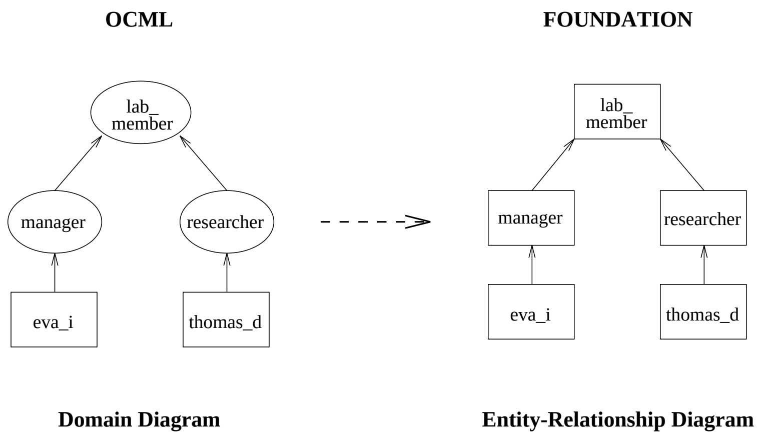

Figure 12: Transformations from OCML domain diagrams to FOUNDATION.

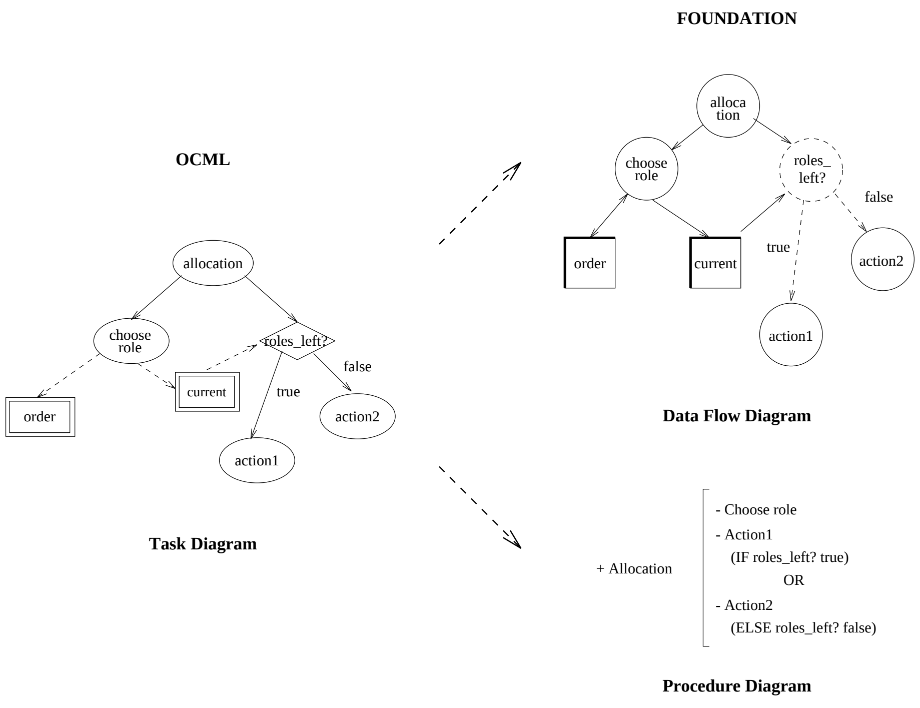

Figure 13: Transformations from OCML task diagrams to FOUNDATION.

- OCML domain diagrams to FOUNDATION Design entity-relationship diagrams
- OCML task diagrams to FOUNDATION Design data flow diagrams
- OCML task diagrams to FOUNDATION Design procedure diagrams

The transformations are closely related as the source representations are very similar, two of the sources being actually the same, and likewise the target representations have important similarities, e.g., as the environment uses a common repository for storing all diagram elements.

The representation format of OCML is a linear textual representation based on Lisp (Figure 14). 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 OCML persistent representation consists of a *graphical part* and a *declaration part*, where the graphical part (starting with `(dale:defgraphics` in Figure 14) contains graphical information of all elements, and the declaration part (starting with `(def-task choose-role` in Figure 14) contains a definition for each element, and describes its connections to other elements (`kldesign-data-link` in Figure 14), and its properties (`:represented-as` in Figure 14).

The target environment, FOUNDATION stores its elements in a repository based on the entity-relationship model. Diagrams and their elements are presented in the repository tables as entities. Information such as "Element E belongs to diagram X" is presented in the repository as relationships between the corresponding entities. For representing graphical information, FOUNDATION uses two strategies. The graphical information of entity-relationship diagrams is included in the relationships between the elements of the diagram and the diagram itself. The graphical information of data flow diagrams and procedure diagrams is presented in separate binary coded graphical files (Figure 14) with the repository entity for the diagram containing a reference to the name of the corresponding graphical file.

All the above CASE transformations have been built with the ALCHEMIST transformation generator. The sizes of the source and target grammars as well as the mappings are represented in Table 1. In this case we also had to implement the transformations in such a way that the heterogeneity of the execution platforms (OCML in the VITAL Tower environment on Sun workstations under UNIX and FOUNDATION on PCs using IBM DB/2 under OS/2) were taken into account.

In order to transform an OCML diagram into a FOUNDATION diagram, the logical and graphical information must be imported into the FOUNDATION repository. Importing (and exporting) repository information is based on using FOUNDATION specific import/export files (see Figure 14). In addition to the import files, the graphical files referenced by the import files must be available for producing the diagrams. The import routine reads both files and stores the corresponding elements in the repository. The source file of each

OCML task diagram:

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

(dale:defgraphics dale:coordinates
 dale::subtask (choose-role (130 130 210 180)
               action1 (305 225 385 275)
               allocation (248 16 328 66)...)
 dale::choice (roles_left? (502 117 582 167)...)

(def-task choose-role
 ((|:::| allocation dale::kldesign-data-link dale::role-alias))
 ((:rule-iteration-type . :try-once)
  (:represented-as :rules choose-role1 choose-role2 choose-role3)
  (:inter-diag-alias)))

(def-choice-task roles_left?
 ((false action2 dale::kldesign-f-control-link dale::subtask)
  (true action1 dale::kldesign-t-control-link dale::subtask)) NIL)

(def-role order
 ((|:::| choose-role dale::kldesign-data-link dale::subtask))
 ((:represented-as :relation role-order)
  (:inter-diag-alias)))
...
```

FND data flow diagram, import file:

```
HEADER                ...
0                     ...
2DEDFDIAG        1  ...
2DEDFDIAG        2sisyphus
2DEPROCSS        1  ...
2DEPROCSS        2CHOOSE-ROLE
...
2DEEXTENT        1  ...
2DEEXTENT        2ORDER
...
3DEDFDIAGDEPROCSS3  ...
3DEDFDIAGDEPROCSS4sisyphus
 ... 0000100000 ACR  CHOOSE-ROLE
...
3DEDFDIAGDEEXTENT3  ...
3DEDFDIAGDEEXTENT4sisyphus
 ... 0000100000 ACR  ORDER
...
```

FND data flow diagram, graphical file:

```
  F   F   O   R   M   X   X   X   U 050
049 052 032 032 032 032   J   B   B 032
032 032 032 032 032 032 032 032 032 032
...
  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   A   L   L   O   C   A
  T   I   O   N 013
...
```

Figure 14: Examples of OCML and FOUNDATION file representations.

Transformation	Source grammar	Target grammar import	Target grammar graphical	Mapping rules
Domain to ERD	85	71	–	35
Task to DFD	80	49	61	37
Task to PD	80	17	37	13

Table 1: Grammar sizes (number of productions) and mapping sizes (number of rules).

transformation uses the OCML representation and the target files are the FOUNDATION import files and the necessary graphical files.

As an example of the kind of transformations that were required for the CASE interface, let us consider the mapping described in Figure 15, required to map a task in the OCML task diagram on a process in the data flow diagram (see Figure 13). Compared to the example transformation in the previous section, this transformation involves also semantic actions: in addition to mapping the identifier (`Task_Name`) to the name of the process (the identifier in the field `Task_Name.ENTITY_ID`) we must insert the name in the symbol table. The symbol table contains both OCML identifiers and the corresponding FOUNDATION identifiers which are not always completely identical (e.g., FOUNDATION limits the length of an identifier while OCML doesn't).

Generating the CASE transformations involved solving problems related to the different views that the source and target languages used for representing diagrams. Typically, the OCML representation described only the logical structure of the diagram (e.g., choice element `roles_left?` is connected to task `action1` by a link with label `true`), while FOUNDATION required also physical information (e.g., control flow `true` starts from point `(405,126)` and ends at point `(385,214)`). The semantic actions producing the necessary coordinates had to combine information from several parts of the source presentations.

Another problem resulted from the differences in the naming strategies of the two environments. Some elements that did not have a name in the OCML diagram (such as links between concepts) had to have unique names in FOUNDATION. This is a point where some user involvement would be useful: the user could produce unique names that would correspond to the meaning of the elements involved. Without user interaction, a simple but noninformative naming strategy was used where the links were given short names and indexed for uniqueness (e.g., all `is a` links of a diagram were named `<diagram name>: is a <index>`).

A more detailed description of this interface provided by transformations can be found in [LV95].

OCML task diagram productions:

```
Task -> "(" "def-subtask"  Task_Name  Links  Attributes ")" ;
Task_Name ->
        OCML_IDENTIFIER ;
```

FOUNDATION data flow diagram productions:

```
Entity_procss_data_record ->
        "2"  "DEPROCSS"  "        "  "2"  Task_Name.Part_entity_id
        "0000100000"  " "  "ACR  "  "        "  "        "  "        "
        "        "  "          "  "  "  "00001"  "        "  "00225"
        "000100000000"  "        "  "ACRA"  "+0001"  "+0000"
        "DEPROCSS"  Task_Name.ENTITY_ID  "  "  Task.PRC_TYPE
        "0000+000"  "0"  "          N"  "VITAL   +"
        CURRENT_DATE  "VITAL   +"  CURRENT_DATE  "+"  CURRENT_TIME
        "        "  "+000000"  "+000000"  "+0+0+0+0+0"
        Task_Name.ENTITY_SHORT_DESC  "E"  Task_Name.Short_description
        "\n" ;

Task.PRC_TYPE ->
        "2" ;

Task_Name.ENTITY_ID -> IDENTIFIER[32] ;
        /* semantic action */
        symbol_table.insert( OCML_IDENITIFIER ) ;
        task_name = symbol_table.get_FND_id( OCML_IDENTIFIER ):
        set_id(IDENTIFIER[32], task_name) ;
        /* end semantic action */
Task_Name.Part_entity_id -> IDENTIFIER[32] ;
        /* semantic action */
        set_id(IDENTIFIER[32], task_name) ;
        /* end semantic action */
Task_Name.ENTITY_SHORT_DESC -> IDENTIFIER[32] ;
        /* semantic action */
        set_id(IDENTIFIER[32], task_name) ;
        /* end semantic action */
Task_Name.Short_description -> IDENTIFIER[25] ;
        /* semantic action */
        set_id(IDENTIFIER[25], task_name) ;
        /* end semantic action */
```

Figure 15: Productions and actions needed for mapping an OCML task to a DFD process.

4.3 Some observations

All the case studies discussed above allowed us to make some observations about the usefulness of the ALCHEMIST approach and the capabilities of the tool itself. As mentioned already earlier, some of these experiences have also directed the development of ALCHEMIST in suggesting new functionalities and also given ideas for future work.

In our experience, the primary strength of ALCHEMIST is in the high level of abstraction and generality it provides. Separating transformation generation and transformation execution (ALCHEMIST itself and the generated spells) was found particularly useful in cases where either the source representation or the target representation is not stable. When building the CASE transformations where the OCML representation was still under development we had several times to adjust to changes in the source representation. This required changing the source grammar and some parts of the mapping, while the target grammar remained fixed. Limiting the effects of even small changes in the OCML representation would have been much more difficult in an end-to-end hand-made transformation. This is also a very common situation in general when new versions of software are released. Therefore at least one of the original motivations, ease of changing the transformation code was achieved.

In addition, in the CASE interface, we built three transformations for which the source representations were fairly similar. Likewise, the target representations had similar components, for example the general structure of the import file. In this case the task of defining the grammars was much reduced because of this similarity since we were able to reuse parts of the source grammars in the specification phase. Even though part of the target representations were very different, such as representation of the graphical information, some general routines could be reused even in those cases (e.g., generation of constants and binary information).

When developing ALCHEMIST, we used a strategy of inductive development: to gain experience of what the spells should be like, we initially implemented a couple of fairly simple spells, and then produced the first version of the ALCHEMIST environment. It was interesting to observe that some of the original ideas of nodes being "active objects" which would be able to transform themselves, were quickly abandoned due to the implementation experience with simple spells. It also became evident that the approach of reusing some of the UNIX compiler software like `yacc` had its own problems, especially obvious in the internal limits we faced in the case of real transformations.

Naturally during the evaluation process we have also been able to point out some shortcomings of the ALCHEMIST environment, some of which will be addressed in the future versions of the software. Although using a generated spell is straightforward and simple, building a new spell requires a certain amount of expertise in defining the grammars and the mapping. The ALCHEMIST user must have some knowledge about context-free grammars, LR parsing, and basic understanding of the underlying concept of TT-grammars.

One should observe that even for a well defined representation there may not exist a grammar, and building a grammar by examining examples alone (like we had to do for some parts of our CASE interface transformations) is tedious and error-prone. For all but completely syntactic transformations, some part of the transformation must typically be hand-coded by adding semantic actions in C++. ALCHEMIST is not well suited for transformations that require much global computing, e.g., where a local part of the target representation is dependent on the entire target. Examples of typical global computations are cases where one has to compute how many objects the target contains, or number objects according to their order in the target, referencing objects according to their order, etc. Such computations cannot be properly handled by any set of local transformations alone and require the use of an attribute evaluation mechanism implemented by semantic actions.

ALCHEMIST uses existing standard or public domain UNIX tools, and the spells it produces also use some of these tools, e.g., `yacc` and `lex`. The advantage of this is simplicity and faster implementation, and the obvious disadvantage is reliance on the performance of these tools. It is interesting to observe that with larger grammars, we sometimes found the performance of `lex` and `yacc` inadequate: in some cases we had to modify the grammars in order to keep the size and number of productions small enough for the tools to handle.

However, our overall impression on the performance of ALCHEMIST was quite satisfactory. As could be expected, the most time consuming task in implementing the transformations is to deduce the grammars for the source and target representations. This is mainly due to the fact that exact representation information is not usually provided by the tool developers. In building the CASE interface, both OCML and FOUNDATION grammars were constructed partly based on documentation, partly by using the tools with test examples and then examining their persistent representations. Once the correct grammars have been constructed, building the mapping itself is a straightforward task. In our experience preliminary results can be reached within a day, and a complete transformation of this size within a week.

Naturally one of the interesting factors is the performance of the transformations themselves. On our Sun platform a typical spell execution time for the CASE transformations ranged from 5 to 120 seconds elapsed time depending on the size of the diagrams (see Figure 16) and the time increases relatively slowly with the size of the diagram. In all cases, the dominant time was not the spell execution but the time required for importing the files into the FOUNDATION environment. The resulting import files were as much as 10 times the size of the source file, the graphical files were considerably smaller (see Figure 17). The largest source files we used were close to 10 kilobytes, containing about 50 graphical elements: for the environment in question larger diagrams were not considered reasonable. However, in test cases we used source files that were up to 40 kB containing some 200 graphical elements. From our experience it also seems that the performance difference between the generated transformations and tailored transformations is insignificant and in most cases nonexistent.

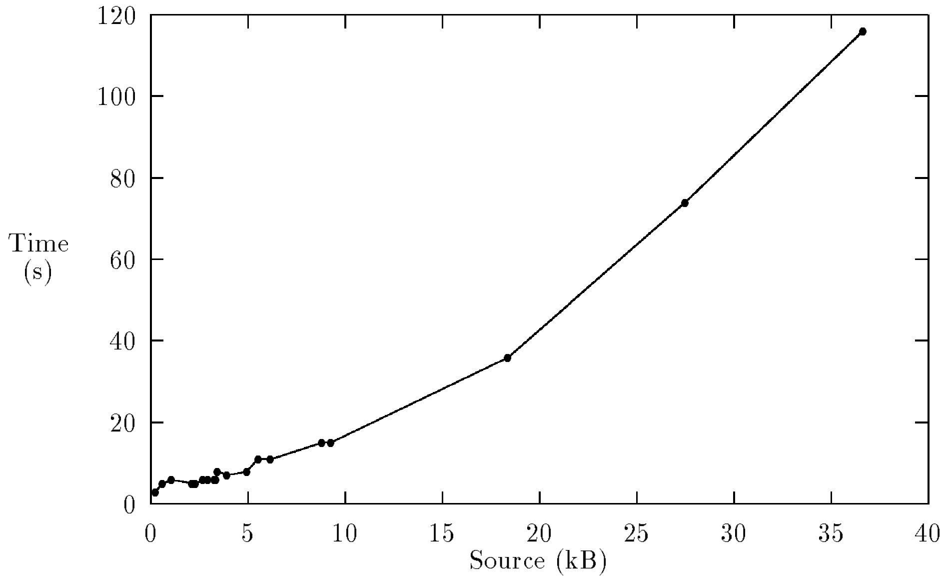

Figure 16: Effect of source file size to spell execution time, transformation from task diagram to data flow diagram.

5 RELATED WORK

There is a vast amount of compiler compilers and parser generators available both commercially and in the public domain. The probably most widely known example of a parser generator is `yacc` [Joh75]. Together with the scanner generator `lex` [Les75], it constitutes the best known compiler compiler tool set for the UNIX world. Another similar tool is the Purdue Compiler Construction Tool Set, PCCTS, [PDC92] which also consists of a parser generator ANTLR and a scanner generator DLG. Both take highly structured source grammars as input but do not "support" a structured syntax for the target; the user may specify any code or action to be performed when parsing the source. Unlike most parser generators, some transformation generators (including ALCHEMIST) also support the syntactical construction of the target. The user has to explicitly give a target grammar according to which the target is constructed. This helps to guarantee that the produced target is syntactically correct.

The transformation generator SSAGS [PKP$^+$82] implements some subclasses of the general TT-grammars described earlier on which ALCHEMIST is based on. The Dual Grammar Translation Scheme (DGTS) [KPPM84] restricts source subgrammars to single productions, where left hand side symbols may be associated only with left hand side symbols in the target subgrammar within the same production group association. A DGTS corresponds to a syntax-directed translation scheme [AU72]. A single input production – explicitly qualified (SIPEQ) TT-grammar [KPPM84] is a subclass of TT-grammars which is also restricted to single production source subgrammars, but symbol associations may be established between any source and target symbols. The implementation of the SIPEQ TT-grammar also includes a simple case statement for choosing between target subgram-

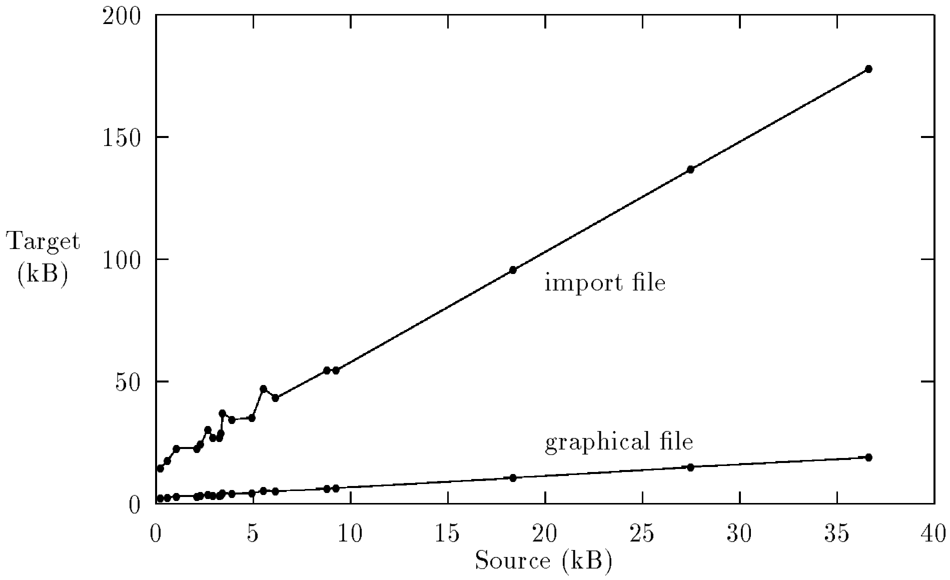

Figure 17: Effect of source file size to target file sizes, transformation from task diagram to data flow diagram.

mars, a copy instruction for multiplying target subtrees, and pseudoproductions for simplifying symbol associations. Both these subsets of TT-grammars are less general than the ALCHEMIST TT-grammar which allows multiple input production subgrammars.

The Integrated Chameleon Architecture (ICA) [MKNS89, MBO93] is a transformation generator consisting of several tools that require a general intermediate representation used as a platform to or from which specific representations can be transformed. The user only specifies one grammar, the *general intermediate representation grammar*. The grammars for the source and target representations are derived from this grammar by permutations of the right hand side symbols of the productions. Due to the intermediate representation, less transformations usually need to be defined in order to produce a complete set of transformations between several representations. ALCHEMIST does not use an intermediate representation, but lets the user define one, if it is appropriate (and if such a representation can be found). ICA is very dependent on the similarity of the source and target representations in order to find an intermediate representation. ALCHEMIST, on the other hand, lets the user define mappings between arbitrary different representations.

An ICA transformation consists of several phases. The first phase adds 'tags' to simplify the transformation. The retagged representation is the source document represented with SGML where all tags have been made explicit, e.g., all start-section tags also have corresponding end-section tags. The motivation for the retagged representation is in the underlying translation model which makes it easier to translate from a fully braced language. The next phase transforms the retagged version into an instance of the general intermediate representation. This instance is further translated down to an SGML-tagged instance of the target document and finally from there to a linear version of the target document.

ICA consists of the several tools. The *Develop Grammar Tool* is used for specifying the general intermediate representation grammar and thereby also for the instances of the specific SGML-tagged source and target documents. The *Replace Tags Tool* lets the user replace existing tags with SGML tags and *Insert Tags Tool* lets the user insert new tags. The *Map Specific to General Structure Tool* specifies the mapping from a specific representation to the general intermediate representation — the user can, e.g., specify alternative orderings if needed — and the *Map General to Specific Structure Tool* specifies the mapping from the general intermediate representation to a specific representation.

The following differences to ALCHEMIST can be noted. ICA saves all intermediate representations as SGML documents (files), while ALCHEMIST keeps them in main memory or uses an object management system. Many of the mapping problems are solved during retagging in ICA, while ALCHEMIST solves them in the actual mapping phase. Both systems can be user assisted. Due to the intermediate representation, less transformations usually need to be defined in order to produce a complete set of transformations between several representations. ALCHEMIST does not use an intermediate representation, but lets the user define one, if it is appropriate (and if such a representation can be found). Finally, ICA is very dependent on the similarity of the source and target representations; in order to find an intermediate representation, the source and target languages must not differ too much. ALCHEMIST, on the other hand, lets the user define mappings between arbitrary different representations; this can also lead to very complex and difficult mapping specifications.

A spell of ALCHEMIST can be seen as first producing a retagged version of a source document and then, without using an intermediate version, mapping this retagged source document directly to a tagged target document. An ICA translator can be seen as producing a specific parse tree of the source document, then "modifying" this parse tree to a general parse tree of the document, and finally modifying this general parse tree into a specific parse tree of the target document. (N.B. The specific parse trees do not contain representation specific tags.)

The Helsinki Structured Text Database System (HST) [KLMN90] is an environment for writing and querying structured documents. The system uses context-free grammars for modelling the structure of documents. HST is a database system with a powerful query language. The HST system allows the user to keep multiple views of the same document simultaneously. Whenever the text in one view is modified, the system automatically updates the other views correspondingly. Transformations in HST are defined through syntax-directed translation schemes [AU72] thereby restricting the transformations to reordering of elements, and insertion and deletion of constant strings. However, HST also defines reverse mappings thereby allowing the user to make updates in any of his view; the modifications will always be propagated to other views. ALCHEMIST can easily implement one-way HST translations because ALCHEMIST is a more general tool and lets the user build transformations between arbitrary grammars.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to automate the design of transformations which support data interchange between applications. This approach is implemented in the ALCHEMIST transformation generating environment which supports specification, generation, and execution of data transformations. We have discussed the benefits of having an abstract specification of the transformation from which the transformation code itself is generated, both from the reusability and correctness point of view. Unlike compiler-compilers, the resulting software environment is intended to automate building transformations between two complex representation formats and is thus especially suitable for constructing transformations between applications for structured data, i.e., database tools, CASE tools, graphical editors or text formatters. Several case studies have been performed to study the applicability of the approach ranging from simple syntactic transformations to a real-life case study of using ALCHEMIST for interfacing two software development environments. Through these case studies we have demonstrated that automatic generation of transformations can improve the design process considerably with respect to the traditional manual approach. The advantages have been especially evident in cases where the representation formats have not been stable but they have changed due to the design decisions of the application involved as well as for cases where parts of an earlier abstract transformation specification could be reused. We believe that the approach adopted is widely applicable in several integration domains.

There are several new directions where the current version of ALCHEMIST could be extended. In addition to the obvious ones, i.e., extending the grammar classes that can be used for structure specification by allowing also LL(k)-grammars, building a better interface for the interaction between the user and the spell in execution etc., we want to point out a few special topics we identified as important during our case studies.

First of these functionalities is related to supporting two way transformations, an area where our approach could clearly be further automated. Such a feature is motivated by the fact that in many integration cases there is a common need to make also a reverse transformation. We encountered such requirements working with software workbenches, where the information flow between tools typically is not one-way. Currently in the ALCHEMIST environment the transformation-reverse transformation pair has to be implemented by building two independent transformations where only the specification grammars can be reused. However, there exists no support for reversing the object-to-object mappings. It is obvious that in many cases at least part of the reverse mapping could be constructed by reversing the association roles of the productions in the production groups.

Furthermore, from the experience in our case studies it became evident that in large transformation instances a built-in trace mechanism for spells should be included in the module set that is linked in the compilation phase to the main spell modules. Such tracing would allow the user to examine also the corresponding source representation when an error is observed while working with the target application. Elementary trace facilities already exist in the form of the trace log file discussed, but augmenting the produced target representation with comments would increase the functionality of the trace mechanism.

Finally, as discussed in the experience section, the real bottleneck in building a transformation specification is the construction of proper grammars for the representations. In most cases no syntax description exists for the representation formats, and this syntax has to be deduced from designer descriptions or more commonly from example files. Constructing grammars from tagged examples by a learning mechanism that would be embedded in the ALCHEMIST environment is an appealing approach, which would help in the initial construction of the grammars. We are currently investigating the possibilities to augment the ALCHEMIST tool set with such a learning module.

Acknowledgements

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

References

[AG89] 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.

[And93a] Andersen Consulting. *FOUNDATION Application Development, Version 2.0*, 1993.

[And93b] Andersen Consulting. *FOUNDATION Design, Analyze Application Requirements, Version 2.0*, 1993.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation and compiling*, volume I. Prentice-Hall, 1972.

[CAE+76] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. SEQUEL 2: a unified approach to data definition, manipulation and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.

[Cat94] R. G. G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[DMW93] 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*, pages 320 – 339, September 1993.

[FL88] C. N. Fischer and R. J. Leblanc. *Crafting a Compiler*, volume I. Benjamin-Cummings, 1988.

[Joh75] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report 32, AT & T Bell Laboratories, Murray Hill, N. J., 1975.

[Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229 – 256, 1980.

[KLMN90] Pekka Kilpeläinen, Greger Lindén, Heikki Mannila, and Erja Nikunen. A structured document database system. In Richard Furuta, editor, *EP90 — Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Gaithersburg, Maryland*, The Cambridge Series on Electronic Publishing, pages 139 – 151. Cambridge University Press, September 1990.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[KPPM84] 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.

[Les75] M. E. Lesk. Lex — a lexical analyzer generator. Technical Report 39, AT & T Bell Laboratories, Murray Hill, N. J., 1975.

[LT95] Greger Lindén and Henry Tirri. ALCHEMIST — the handbook. Version 1.08. Technical Report T416/DS/2, ESPRIT-II Project 5365 VITAL, April 1995.

[LV95] Greger Lindén and A. Inkeri Verkamo. An interface between different software development environments. To appear in the *Proceedings of the 10th Annual Knowledge Based Software Engineering (KBSE) Conference*, Boston, November 1995.

[MBO93] Sandra A. Mamrak, Julie Barnes, and Conleth S. O'Connell. Benefits of automating data translation. *IEEE Software*, 10(4):82 – 88, July 1993.

[MKNS89] Sandra A. Mamrak, Michael J. Kaelbling, Charles K. Nicholas, and Michael Share. Chameleon: A system for solving the data-translation problem. *IEEE Transactions on Software Engineering*, 15(9):1090 – 1108, sep 1989.

[MR90] Nigel Major and Han Reichgelt. ALTO — an automated laddering tool. In B. Wielinga et al., editor, *Current Trends in Knowedge Acquisition.* IOS Press, 1990.

[PDC92] T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual, version 1.0. *ACM SIGPLAN Notices*, 27(2), February 1992.

[PKP+82] 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.

[SMR93] Nigel Shadbolt, Enrico Motta, and Alain Rouge. Constructing knowledge-based systems. *IEEE Software*, 10(6):34 – 39, November 1993.

[TL94] 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.

[Ver94] 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.

[VL95] A. Inkeri Verkamo and Greger Lindén. Problems in interfacing tools of different development environments. In *Proceedings of the Seventh International Conference on Software Engineering and Knowledge Engineering (SEKE '95)*, pages 429 – 437, June 1995.

A AN EXAMPLE SPELL SPECIFICATION

A.1 ODL grammar (source grammar)

```
1   ODL -> Type_interfaces
2   Type_interfaces -> Type_interfaces Type_interface
3   Type_interfaces -> Type_interface
4   Type_interface -> Interface_header "{" Type_start Attributes Relationships "}"
5   Interface_header -> "interface" Type_Name
6   Type_start -> "extent" IDENTIFIER ";" "key" "(" Key_List ")" ";"
7   Type_Name -> IDENTIFIER
8   Attributes -> Attributes Attribute
9   Attributes -> Attribute
10  Attribute -> "attribute" Att_Type Att_Name ";"
11  Attribute -> "attribute" Att_Name Att_Type ";"
12  Att_Type -> "integer"
13  Att_Type -> "string"
14  Att_Name -> IDENTIFIER
15  Key_List -> IDENTIFIER
16  Key_List -> Key_List "," IDENTIFIER
17  Relationships -> Relationships Relationship
18  Relationships -> Relationship
19  Relationship -> "relationship" Rel_Type Rel_Name  ";"
20  Rel_Type -> IDENTIFIER
21  Rel_Name -> IDENTIFIER
```

A.2 SQL grammar (target grammar)

```
1   SQL -> Create_Tables
2   Create_Tables -> Create_Tables Create_Table
3   Create_Tables -> Create_Table
4   Create_Table -> "CREATE TABLE " Table_Name " (\n" Table_id Table_atts ")\n\n"
5   Table_Name -> IDENTIFIER
6   Table_id -> "\n\t" IDENTIFIER "# NUMBER(8,0)\n\n"
7   Table_atts -> Table_atts1 "\n" Table_atts2
8   Table_atts1 -> Table_atts1 Table_att1
9   Table_atts1 -> Table_att1
10  Table_atts2 -> Table_atts2 Table_att2
11  Table_atts2 -> Table_att2
12  Table_att1 -> "\t" Table_Att_Name " " Table_Att_type "\n"
13  Table_att2 -> "\t" Table_Att_Name "# " Table_Att_type "\n"
14  Table_Att_Name -> IDENTIFIER
15  Table_Att_type -> "CHAR(20)"
```

```
16   Table_Att_type -> "NUMBER(8,0)"
17   Table_Att_type -> IDENTIFIER
```

A.3 ODL to SQL mapping specification

```
1    ODL ->                          ODL.SQL ->
        Type_interfaces                 Type_interfaces.Create_Tables


2    Type_interfaces ->              Type_interfaces.Create_Tables ->
        Type_interfaces                 Type_interfaces.Create_Tables
        Type_interface                  Type_interface.Create_Table


3    Type_interfaces ->              Type_interfaces.Create_Tables ->
        Type_interface                  Type_interface.Create_Table


4    Type_interface ->               Type_interface.Create_Table ->
        Interface_header                "CREATE TABLE "
        "{"                             Type_Name.Table_Name
        Type_start                      " (\n"
        Attributes                      Type_Name.Table_id
        Relationships                   Type_interface.Table_atts
                                        ")\n\n"
     Interface_header ->
        "interface"                  Type_interface.Table_atts ->
        Type_Name                       Attributes.Table_atts1
                                        "\n"
                                        Relationships.Table_atts2


5    Type_Name ->                    Type_Name.Table_Name ->
        IDENTIFIER                      IDENTIFIER.IDENTIFIER

                                     Type_Name.Table_id ->
                                        "\n\t"
                                        IDENTIFIER.IDENTIFIER
                                        "# NUMBER(8,0)\n\n"
```

```
6   Attributes ->                  Attributes.Table_atts1 ->
       Attributes                     Attributes.Table_atts1
       Attribute                      Attribute.Table_att1

7   Attributes -> Attribute        Attributes.Table_atts1 ->
                                      Attribute.Table_att1

8   Attribute ->                   Attribute.Table_att1 ->
       "attribute"                    "\t"
       Att_Type                       Att_Name.Table_Att_Name
       Att_Name                       " "
       ";"                            Att_Type.Table_Att_type
                                      "\n"

9   Attribute ->                   Attribute.Table_att1 ->
       "attribute"                    "\t"
       Att_Name                       Att_Name.Table_Att_Name
       Att_Type                       " "
       ";"                            Att_Type.Table_Att_type
                                      "\n"

10  Att_Type ->                    Att_Type.Table_Att_type ->
       "integer"                      "NUMBER(8,0)"

11  Att_Type ->                    Att_Type.Table_Att_type ->
       "string"                       "CHAR(20)"

12  Att_Name ->                    Att_Name.Table_Att_Name ->
       IDENTIFIER                     IDENTIFIER.IDENTIFIER

13  Relationships ->               Relationships.Table_atts2 ->
       Relationships                  Relationships.Table_atts2
       Relationship                   Relationship.Table_att2

14  Relationships ->               Relationships.Table_atts2 ->
       Relationship                   Relationship.Table_att2
```

```
15  Relationship ->                 Relationship.Table_att2 ->
      "relationship"                   "\t"
      Rel_Type                         Rel_Name.Table_Att_Name
      Rel_Name                         "# "
      ";"                              Rel_Type.Table_Att_type
                                       "\n"


16  Rel_Type ->                     Rel_Type.Table_Att_type ->
       "integer"                       "NUMBER(8,0)"


17  Rel_Name ->                     Rel_Name.Table_Att_Name ->
       IDENTIFIER                      IDENTIFIER.IDENTIFIER
```

A.4 Notes on the example spell

The grammars contain some typical features of context-free LR-grammars. Both the source and the target grammar contain several productions. Iteration is expressed though recursiveness (e.g., source productions: 2, 8, or 17; or target productions 2, 8, or 10). A choice or disjunction is expressed by giving several productions with the same left hand side (e.g., source productions 2 & 3, 8 & 9, or 20 & 21; or target productions 2 & 3). Both grammars contain terminals in the form of constant strings (e.g., source productions 4, 5, 6; or target productions 4, 6), or as identifiers (e.g., source productions 6, 7; or target productions 5, 6).

The mapping contains several production group associations (pgas). Some pgas even contain production groups (pgs) with several productions (pga 4, 5). The mapping changes the order of the structure (from source to target, pga 8, 15), or deletes structure (= some of the source structure is left out): e.g., `Key_List` productions 15 & 16 in source grammar; all constant strings), or inserts new structure (all constant strings, repetition, pga 4).

Symbol associations (sa) are used for copying identifiers (pga 5, 12, or 17) and for linking the target tree together (pga 4 especially, but also all other pgas) No other semantic actions in the form of programming code has been included here.

Generation and compilation from this specification is straight forward.

B TT-GRAMMARS

A *TT-grammar* [KPPM84] is a formal description technique for describing transformations from one language to another. A TT-grammar describes a relationship between a syntax tree over a grammar G_1 and a syntax tree over a grammar G_2. 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 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 sextuple (G_s, G_t, S_s, S_t, PA, SA), where G_s and G_t are the *source grammar* and the *target grammar*, 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, or in separate production group associations).

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.

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 $t \to \alpha s \beta$ and $s \to \gamma$. 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 2 The following example is taken from [KPPM84]. Assume that we want to translate binary numbers to the decimal value as the sum of powers of two, e.g., the binary number `101` will be translated to `2**(0+1+1) + 2**(0)`. The source grammar (Figure 18) describes the binary numbers as strings of '0's and '1's starting with a '1'. The target grammar (Figure 18) describes values as sums of powers of two according to a kind of programming syntax, i.e., '**' for 'power of', etc.

source grammar	*target grammar*
$\texttt{root}_{\texttt{i}}$ `-> seq ;`	$\texttt{root}_{\texttt{o}}$ `-> sums ;`
`seq -> seq d ;`	`sums -> sums '+' exp ;`
`seq -> '1' ;`	`sums -> sums ;`
`d -> '0' ;`	`sums -> exp ;`
`d -> '1' ;`	`exp -> '2' '**' '(' sums1 ')' ;`
	`sums1 -> sums1 '+' '1' ;`
	`sums1 -> '0' ;`

Figure 18: Source and target grammars of a simple example.

```
1   root$_{\tt i}$ -> seq          root$_{\tt i}$.root$_{\tt o}$ -> seq.sums
                                   seq.sums -> '0'

2   seq -> seq$_{\tt 2}$ d         seq$_{\tt 2}$.sums1 -> seq.sums1 '+' '1'
    d -> '0'                       seq.sums -> seq$_{\tt 2}$.sums

3   seq -> seq$_{\tt 2}$ d         seq$_{\tt 2}$.sums1 -> seq.sums1 '+' '1'
    d -> '1'                       seq.sums -> seq$_{\tt 2}$.sums '+' 'exp
                                   exp -> '2' '**' '(' seq.sums1 ')'

4   seq -> '1'                     seq.sums -> 'exp
                                   exp -> '2' '**' '(' seq.sums1 ')'
```

Figure 19: Rules of an example TT-grammar.

The mapping in Figure 19 associates production groups with each other. A symbol association is denoted by *source_symbol.target_symbol* in the target production groups of the production group associations. A symbol can be indexed to distinguish it from a same label symbol within the same production group association.

We see a sample transformation in Figure 20. The sample source consists of the binary number `10`. The source tree is built over the source grammar. The TT-grammar matches three substructures according to rules 1, 2, and 4. It produces five target subtrees with appropriate symbol associations (dashed lines in Figure 20). The target subtrees are linked together by merging nodes with the same associations (dotted lines in Figure 20) and the target tree is formed.

Some subclasses of the general TT-grammars described above have been implemented in the SSAGS transformation generator [PKP+82]. The Dual Grammar Translation Scheme (DGTS) [KPPM84] restricts source subgrammars to single productions, where left hand

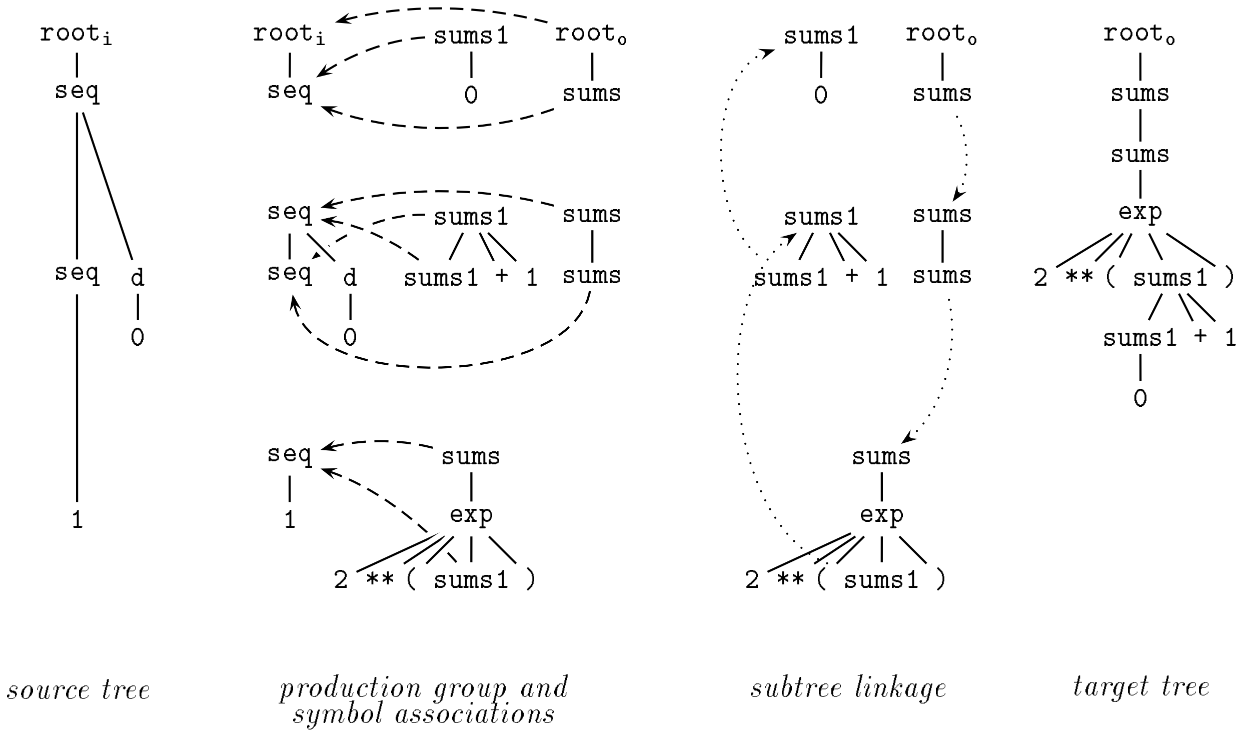

Figure 20: Transformation of a source tree to a target tree using a TT-grammar.

side symbols may be associated only with left hand side symbols in the target subgrammar within the same production group association. A DGTS corresponds to a syntax-directed translation scheme [AU72]. The TT-grammar in the example above violates both these restrictions.

A single input production – explicitly qualified (SIPEQ) TT-grammar [KPPM84] is a subclass of TT-grammars which is also restricted to single production source subgrammars, but symbol associations may be established between any source and target symbols. A SIPEQ TT-grammar corresponds to ordered attribute grammars [Kas80]. The implementation of the SIPEQ TT-grammar also includes a simple case statement for choosing between target subgrammars, a copy instruction for multiplying target subtrees, and pseudoproductions for simplifying symbol associations.

The ALCHEMIST TT-grammar allows multiple production source subgrammars, but restricts symbol associations to be within the same production group associations. The implementation of an ALCHEMIST TT-grammar allows copying of identifiers and numeric terminals to the target, as well as combining the power of a programming language with the construction of the target tree.

Helsinki 1995
Yliopistopaino