INFORMATICA, 1999, Vol. 10, No. 1, 109–126

© 1999 *Institute of Mathematics and Informatics, Vilnius*

Data Conversion Development: A Tool-Supported Approach

Janis PLUME
Riga Information Technology Institute
Kuldigas iela 45, LV-1083, Riga, Latvia
e-mail: janis.plume@dati.lv

Received: December 1998

Abstract. This article provides a brief introduction to an approach toward data conversion development. The article discusses activities in the area of conversion software development, as well as a model for the life cycle of this development process. Also analyzed is a possible method of tool support for the development process.

Key words: software tools, data conversion, software engineering processes.

1. Introduction

It is a well-accepted fact that the quality of a software product is largely determined by the quality of the processes that are used to develop and maintain it (Kellner and Hansen, 1988). Software tools are often used to improve these processes (development, maintenance, etc.). When using software tools, it must be remembered that the main aspect is the process, and tool support must be adjusted to the software process. This means that tools cannot bring any significant contribution to software development productivity and quality if the overall project is undisciplined or chaotic (Paulk *et al.*, 1993). There should be a clear understanding in advance about the way in which the tool will be used, as well as about the role which the tool will play in the entire process.

Software development can be performed in many different ways, and tools usually support one particular method of software development, one particular kind of a process. This is frequently the reason why software tools with excellent functionality are used successfully on some sites, but do not bring the same benefits to others. Others use different software processes, and thus they cannot take full advantage of the tool's features. The real benefit from the use of a tool highly depends on whether the tool has adequately been integrated into the entire software process.

The extent to which a process can be supported by a tool differs from case to case. One approach is to understand that each tool is designed to perform a single activity, and an entire process, then, is supported by a set of small tools. Integration of such a set of interrelated tools is another approach. In this case, you obtain a "bigger" tool, one which automates a larger part of the software process.

The aim of this paper is to provide a short introduction to one approach toward data conversion software development, showing how the approach is transformed into a process that is supported by tools.

The conversion software development approach that is described here is supported by a set of small tools which are not highly integrated among themselves. The advantage of this is that each tool can be improved, or replaced by something different, without affecting the other tools that are involved in the process. In this case it is also easier to change the software process that is being supported. It is possible, moreover, to avoid the use of one of the tools altogether if it is felt that the respective activity can better be performed by hand.

Our approach toward data conversion software development is described in the first section of the paper. After that we will describe the process of developing data conversion software (i.e., a data converter) by using this approach. The final section contains a description of the tools that used in the conversion development process.

2. The Metamodeling Approach in Conversion Software Development

2.1. *Overview*

This approach is fundamentally based on what are known as metamodels (EIA/PN-2387, 1993). Metamodels are used to describe the information structure. Although this approach originally comes from data bases in which metamodels are used to describe the structure of the data base, the fact is that the method can be extended. Metamodels can be taken as a description of the abstract syntax of data. Thus very different kinds of data can be described through the use of metamodels.

In this paper we will devote specific attention to the "non-traditional" applications of data conversion, i.e., CASE tool data conversion, reverse engineering and code generation, as opposed to the application data conversion. The conversion process is usually file-based in these non-traditional applications of data conversion. This means that the conversion involves taking a file that is in one format as input, and then generating the file in another format (e.g., the import-export format of a CASE tool).

The basics of the approach that is described here line in the fact that any data format with a fixed syntax can be described by a metamodel (not unique). A metamodel can be more abstract, reflecting the logical contents of the information, or it can be more "technical". This means that the metamodel very much reflects the syntax of the file format.

There is an essential advantage in using an abstract description of information in data conversion software development. It allows one to refrain from thinking about the representation of data while the conversion software is being developed. Thus the main effort can be concentrated on the logical implementation of the conversion. This is of particular importance if the source and target file formats (and the logical contents of the files) are completely different.

In this approach, the following basic functions of file-based data conversion software are separated out:

1) *Data import*. This function represents the first step in the data conversion process, where the information from the source data file is imported into the internal data store of the conversion software.
2) *Data conversion*. This function represents the logical data conversion which is performed in the internal data store.
3) *Data export*. This function represents a generation of the information in a form that can be read in the target environment.

It is reasonable to implement all of these functions in different modules or components of the conversion software. Thus, three major components in the data converter can be identified: the importer, the converter and the exporter.

If all of these components are to operate together, there must be an interface, and this interface is provided by a common data store for all components (internal data store). As the conversion data must be accessible from the internal store during the conversion runtime only, it is reasonable to implement the data store in the main memory.

This approach was invented and implemented for CASE data conversion, reverse engineering and code generation areas (Plume *et al.*, 1998). Actually, there is no essential problem in using this approach for application data conversion, but in that case there are some differences that must be considered:

- application data conversion is characterized by a larger amount of information, and the internal data store cannot be implemented in the main memory;
- it can be more convenient not to implement application data conversion in a file-based way, but rather to use direct access to the data base.

2.2. *Converter Components*

On the basis of all of this, the following scheme can be drawn to illustrate (Fig. 1) the data conversion process:

The development of this kind of conversion software can be supported by two major methods:

- standard components that may be included unchanged (or parameterized) in data conversion software;
- software tools that support the development of some (conversion-specific) conversion software components.

Let us turn to a brief description of the components in data conversion software (both standard and conversion-specific). The latter sections of the paper will focus on the development of conversion-specific components.

2.2.1. *The Data Importer*

Data import is the first step in a data conversion process, and it is responsible for the following functions:

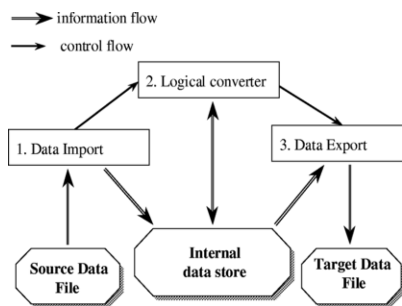

Fig. 1. The components of a simple converter.

- parsing of the source data file;
- storing information in the internal data store, according to the metamodel of the source data.

Data import functions are encapsulated in one component – the data importer. This is a pure, conversion-specific component of the conversion software, because it depends on the syntax of the source file and the metamodel of the source data.

2.2.2. *The Converter*

This is a component of data conversion which performs the essential step of the data conversion process – it converts the source metamodel instances into the target metamodel instances. This is a conversion-specific component because it depends on the metamodel of the source and target environment. From here on, we shall refer to this component of the conversion software as a "logical converter" or a "converter".

2.2.3. *The Data Exporter*

The data exporter is responsible for the export of target model data that have been produced during the logical conversion. The data exporter is aware of the syntax of the target data format and generates an appropriate export file.

2.2.4. *A Metamodel-Based Internal Data Store*

A common data store (repository) is necessary in order to implement the converter process that was described in the previous section. This repository is used to store the data that are produced during the conversion runtime.

The internal data store is a component which is common to all conversions that are developed on the basis of this approach. Thus the repository must support multiple metamodels for data storage of different conversions.

In general, an internal data store in a metamodel based data converter must support two kinds of information store:

- metadata
- data

A metamodel can be used to maintain the transparency of the internal data store. The metamodel consists of two basic parts – one to store the metadata, the other to store the data. Both metamodels are loaded into the internal data store before the data import process is begun. Both models are created in an internal data store during the conversion process and are consistent with the corresponding metamodels. An example of a repository metamodel is given in source (Plume *et al*., 1998).

2.2.5. *The Conversion Engine*

All of the conversion process is conducted by a central component known as the conversion engine. The conversion engine is a standard component that is adapted for the particular conversion through the process of parameterization.

There is also an important concept that can be called a "conversion package", or a "conversion software package". Sometimes it makes sense to collect more than one type of data conversion in a single tool, and such a facility is the responsibility of the conversion engine. In this aspect, parameterization includes specifying the conversions that will be present in a conversion package.

A conversion engine contains all of the user interfaces of the conversion software. The conversion engine also contains all of the application programming interfaces (API) that are used by the data importer and the data exporter.

The following is the data conversion process from the point of view of the conversion engine:

- the engine loads both the source and the target metamodels into the internal data store;
- the engine starts the data importer.
- the engine starts the data converter.
- the engine starts the data exporter.

3. The Converter Development Process

3.1. *Conversion Development Activities*

The main conversion development activities are the described in this section, and the description is performed on the basis of two major concepts in software development activities (IEEE, 1991):

1) *The entry condition*. This is a description of the conditions which must be satisfied in order to perform the activity. Usually the entry condition is characterized by the readiness of some intermediate results or the results of the previous phases of the development life cycle.
2) *Output*. This describes the products that must be developed as a result of the activity. Usually products are software programs, data necessary for a program to operate, or documents describing the data or program.

In the following sections, an informal description of activities is provided. Activities are divided up into smaller tasks where this is possible and reasonable.

3.1.1. *Metamodel Development*

The initial activity that is necessary in developing metamodel-based data conversion software, of course, is the development of the metamodels of the source and the target data. This can be done in two steps:

1) logical understanding of the data;
2) development of the metamodel using the description in the form of an ER-diagram.

It is very often necessary to become familiar with the environment (the CASE tool) before proceeding with the first step in this activity. If the developer of the metamodel is an experienced person in this area, the activity may require very little time. This may also prove that the tool has already explicitly defined the metamodel in the documentation of the tool (environment) that is to be processed. For instance, more and more CASE tools are using relational data bases as repositories. In such cases the data base structure can serve as a metamodel for the CASE tool's data.

The second step very much depends on the level of transparency in the tool. Very often a metamodel can be obtained easily if the tool's data store (repository) is built through an explicit use of the metamodel. It is creative work to develop a metamodel for a data format that is not explicitly based on metamodels.

Entry condition: source and target tools (or data formats) available;

Output: metamodel descriptions.

3.1.2. *The Converter Description*

After both metamodels are ready, the development of a conversion description can be begun. The result of this activity is the converter component of the conversion software. This activity, too, can be performed in two steps:

1) logical design of the converter, using the source and the target metamodel.
2) implementation of the converter.

The logical design of the converter must be performed in line with the requirements of the customer. It may turn out that the requirements cannot be fulfilled with adequate efficiency. In that case the metamodels must be redesigned.

In the next section we will present a tool which allows one to obtain a ready converter from the high-level logical description of the conversion.

Entry condition: source and target metamodels available;

Output: converter description.

3.1.3. *Importer Development*

After the source metamodel is developed, the next step is to develop the importer component. The importer is responsible for shipping the data from the source environment data

format to the internal data store. The conversion development framework offers an application programming interface (API) for such purposes. Thus the importer development splits up into two steps:

1) parser development;
2) extension of the parser with the API calls, which performs data shipping in the internal data store.

Parser development has been discussed extensively in the literature (Aho *et al.*, 1986). There are several tools that support the development process, among the most popular being the lexical analyzer generator LEX and the parser generator YACC (Levine *et al.*, 1992). LEX uses regular expressions to generate the lexical analyzer. YACC uses the description of the grammar to generate the parser. The restriction is that YACC needs the description of the grammar in the form of LALR (1). LEX and YACC work together very well, however.

Importers vary very greatly in terms of the effort that is necessary to develop them. The level of effort, in turn, depends on the complexity of the language that is to be processed. Most CASE tools have an import/export format with a very simple syntax, and in such cases there is no need to use YACC. The situation is very complex, however, when it comes to reverse engineering. In such cases a full syntax analysis of the processed language is necessary. Experience shows that it is very difficult to tune up large grammar to be in the form of LALR (1). In order to reduce the complexity of the parsing, it is necessary to restrict the syntax description in terms of the level of detail. Only those constructs of the language are processed which present relevant information for the further steps (converter, exporter).

Entry condition: source metamodel available and source data file syntax available;

Output: importer component of the conversion.

3.1.4. *Exporter Development*

After the metamodel of the target data is ready, the exporter component can be developed. Exporter development means producing the component that reads information from the internal data store and generates an output file.

An exporter is produced by using the standard exporter API. The exporter API functions are allowed to access only target model data, i.e., only that information which is produced by the conversion component.

Entry condition: target metamodel and target data file syntax available;

Output: exporter component of the conversion.

3.1.5. *Integration*

Once all of the conversion components are ready, they can be integrated together amongst themselves, and with the conversion engine. As the interface among the components is standardized, the integration means just arranging the right components in the right sequence. This is done by parameterization of the conversion engine.

The following are the main features that can be obtained through parameterization of the conversion engine:

- Registration of the converter components, which will assist in the conversion process (importers, exporters, converters). A conversion package usually contains more than one type of conversion. For instance, customers usually need a data converter that can convert data in two opposite directions. Sometimes customers need "gateways" or "bridges" from their tool to more than one other tool. In such cases all of those converters can be incorporated into a single package. Registration means registering all of the conversion components that are present in one conversion package and all of the metamodels that are going to be used.
- Simple (3-component) conversion definition.
- Compound conversion description. If the source and target metamodels are very different, it is sometimes necessary to introduce an intermediate metamodel. In such cases it is permissible to establish so-called "compound conversions", where more than one converter component goes to work before the data are exported.

This parameterization information must be stored in a file that can be read by the conversion engine each time that it starts up. For instance, an ".ini" file can be used as the place where the conversion parameters are stored.

This integration activity can be performed early on in the project in order to obtain a conversion prototype. In this case some stubs for the conversion components must be developed. Integration can also be performed late in the project, when all of the required components have been tested.

Entry condition: stubs for the conversion software components or ready components available;

Output: a conversion package prototype where all components are involved, or a ready conversion package.

3.1.6. *Debugging and Testing*

3.1.6.1. *Integration Testing*

Integration testing involves checking that the conversion components have been arranged in the right sequence and that the correct metamodels are attached to the corresponding components. This is usually a very small task, and it can be performed early in the project. At early stages stubs must be used in place of the real data conversion components.

Checking the interface among the conversion components implies checking that the conversion components are using the correct metamodels to store data in the internal data store.

Entry condition: conversion software prototype available;

Output: tested conversion prototype.

3.1.6.2. *Component Testing*

Although conversion software is strictly divided up into components, it is not all that easy to test the components individually. What are the difficulties? First of all, importer testing implies the following checks:

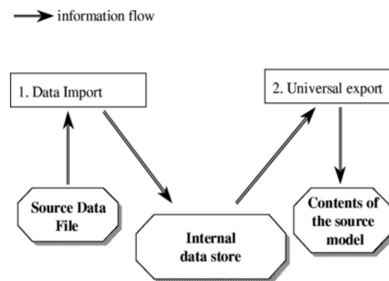

Fig. 2. Testing of an importer.

1) Does the parser of the input data file work correctly?
2) Are the correct data imported into the internal data store?

Testing of the parser can be performed easily and independently. It is reasonable to assume that the import files will be in the correct syntax, and that is why no complex error processing and recovery are necessary with respect to the parser. The main task in parser testing is to check that the parser will accept the files of the correct syntax. It is, however, also reasonable to include a few error handling features in the parser in order to make it easier to tune up the grammar.

To check the correctness of imported data, an additional component is necessary which makes it possible to look into the internal data store. The most common way to see the contents of the memory is to develop a so-called "memory dump" or universal exporter. This is a component which is equivalent to the exporter and can print out a source or target model. A universal exporter prints out all object instances that are present in the internal data store.

This is the way in which the importer can be tested. It requires a re-parameterization of the conversion engine in order to ensure that the universal exporter works immediately after the importer and prints out the model that the importer has just imported (see Fig. 2). If the imported data are correct, the printout of the imported data may serve as a standard for further checks of the importer, using them to check the importer again in case there are changes.

The same testing applies to the converter component. In this case it is important the importer, which works before the converter, be tested first. The universal exporter can be used for converter testing, too. It can be done by using the universal exporter instead of the real data exporter. The printout of the universal exporter can be used as a standard for the conversion testing in the event of changes in the converter component.

Exporter testing can be performed only after the other components in the conversion process are tested. The sequence for component testing that has been described here can differ when producing the conversion package with conversion from one tool to another, and vice versa. For testing purposes, the following conversion can be produced: the importer and exporter for the same environment are integrated together. The criterion for the testing process is that the input and the output of such a "null-conversion" are equivalent. This approach allows one to obtain tested components more quickly.

Entry condition: component ready for testing, and previous component tested;
Output: a tested component.

3.1.6.3. *System Testing*

Once the components are tested, system testing of the conversion can be started. This activity does not differ very much from testing other kinds of software products. The essential thing is the ability to run the conversion in a batch mode in which the test can be run without assistance from a human being. It is also necessary to automate the checking of the test results.

First of all, it is necessary to ensure that the conversion engine can work in the batch mode. The solution lies in the fact that there are some script files which are interpreted by the conversion engine. In those script files the user provides the necessary information to run the tests automatically. The information includes:

1) the location of the test example;
2) the location of the results;
3) the type of conversion that must be performed, as the testing is performed with a conversion package.

The result of the conversion is a text file, and an obvious way to check the results of the conversion is to compare the file with the standard. Any simple file comparison tool can be used here (e.g., the DOS command "fc"). There are, however, some problems with automatic comparison, because the results may contain information that is dependent upon, for example, the time when the conversion was performed. That is why some human assistance may be necessary at this stage.

Entry criteria: all components tested;
Output: tested conversion software.

3.1.7. *Documenting the Metamodel-Based Conversion*

Two kinds of user documentation are necessary:

- the conversion description;
- the user manual.

In those instances when the conversion is complicated and meant for advanced users of the software, documentation describing the conversion may be necessary. The document should clearly define the issue of which concepts from the source data are to be converted to which concepts in the target data. It is useful to develop this documentation early on in the process and to validate it (Ince, 1995). If such a document is developed, it may be treated as a kind of requirements specification. The document can also be used as a basis for the development of conversion components. The main reason for developing the document early is that it can be presented to the customer before the real implementation of the converter is started. One prerequisite for writing a document description is that both metamodels must be developed. It can be agreed with the customer that he approves the description first, and only then the work actually begins. This approach re-

duces the need for additional work that occurs because the customer has not been clear in his requirements.

This approach means that the customer or the potential user of the conversion software must become accustomed to the metamodeling approach in general and with the particular metamodels of the source and target environments. This may prove very difficult, especially if the user is not familiar with the basic concepts of data bases and ER modeling.

Entry condition (conversion description): metamodels ready;

Output: conversion description.

Very little effort is needed to develop a user manual for the conversion software. As the application of the data conversion software is clear and evident, the main mission of a user manual is to describe the user interface of the conversion software, or user access to the product (EIA/IEEE, 1995).

The user interface is very small, and it is standardized for all data conversions. In general, the user of the data converter must deal with at least the following issues:

1) What type of conversion is it (i.e., what set of conversion components shall be used)?
2) Where are the source data to be taken?
3) Where are the target data to be stored?

A user manual should contain illustrations of the screens via which these parameters can be selected.

Entry condition (user manual): conversion prototype ready;

Output: user manual.

3.2. *The Life Cycle Model of Converter Development*

Development of a converter means producing the metamodels for the source and the target data, and then developing the three conversion modules – the importer, the converter and the exporter. The components and the entire converter must be tested, and the software must be documented. Let us take a closer look at the usual sequence in the production of these components.

Although the product that is obtained in this process is a very specific kind of software, the development process as such generally contains the same phases as does traditional software development.

The most desirable development process model may well be the one that is illustrated in Fig 3. The model was obtained after considering the outputs and entry conditions of the activities that were described in the previous sections.

The left side of the picture contains a description of the conversion development activities. The rounded rectangles are the activities, while the arrows represent the sequence of activities. The keyword “AND” inside an activity symbol means that the activity cannot be started before all previous activities are completed. This notation is adopted from (Barzdins *et al.*, 1998; Infologistik, 1997).

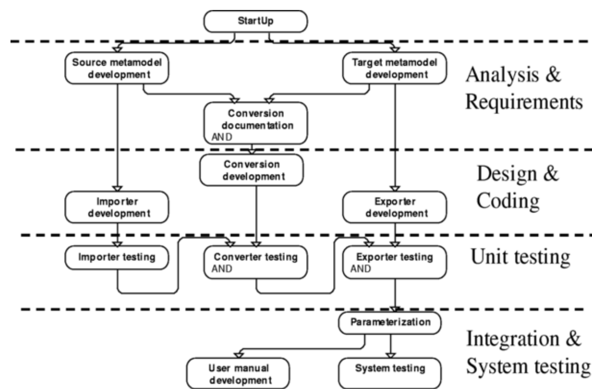

Fig. 3. The life cycle of converter development.

The right side of the picture contains the names of the traditional waterfall or sequential software development life cycle model, and the way in which the conversion development activities are grouped to correspond to the traditional software development life cycle is demonstrated (Pressman, 1994).

It is worth pointing out that the process described in Fig. 3 has some features from other software development life cycles, not just waterfall.

For example, the converter component is developed through a typical fourth-generation technique approach, because the appropriate tool for this approach is available in the framework. In Fig. 3 this is represented by the union of the design and the implementation in one single phase. Actually, real projects can very seldom be carried out via a pure sequential development model. Although this is not seen in Fig. 3, the reality is that in almost any activity may discover problems that can be solved only by making changes in the results of the previous activities. For example, when we try to develop the importer component, we may find that the importer cannot be implemented effectively enough with the existing metamodel. In that case the source data metamodel must be redesigned. This, in turn, involves changes in the conversion description, too. Generally speaking, problems that are found in any phase or activity may involve a return to the previous phase or activity.

3.3. *Interaction with the Customer*

There can be small modifications of the conversion development life cycle, depending on the model of cooperation between the developer of the conversion and the customer. There are two basic situations:

1) the customer needs a conversion tool to perform the data conversion himself;
2) the customer has data which he wants to have converted.

The main difference here lies in the fact that in the second case, the correctness of the tool is measured on the basis of how correctly it works on the particular set of data. This usually means a reduced testing phase. If a tool is to be delivered to the customer,

however, it will always require more rigorous testing than is the case if the customer merely wants to obtain converted data.

There is also a potential difference in the development process model, depending on the clarity of the conversion rules or requirements. If the rules are strictly defined by the customer from the very beginning, or if they are worked out by the developer early in the process and approved by the customer, the life cycle model is more like a waterfall. If the requirements are vague, some kind of evolutionary development model is more realistic.

The crucial aspect in keeping the data converter development under a precise time frame is defining the conversion rules or requirements early on in the project. Actually, this is true with any other kind of software, too; requirements are the key document in any software project (Ince, 1995).

It is also possible that the customer may need a prototype of the data converter to be developed early in the project. In that case, stubs of some sort must be developed instead of real components, and integrated into a conversion package. This can easily be done via this approach, because the integration activity (the parameterization of the conversion engine) requires very little effort. What's more, the effort hardly depends on the moment at which the integration (parameterization) is performed, because the interfaces between components are clear and standardized.

4. Conversion Development Framework for Converter Component Development

As was mentioned previously, there are a few software development tools that can support the conversion development process. In this section we shall describe some tools that were used in the approach that is described in Plume *et al*. (1998).

4.1. *A Metamodel Development Tool*

The converter uses a metamodel for the source and target data to perform a conversion. Thus the converter needs the metamodels in a form that can be loaded into the internal data store. On the other hand, it is useful to develop metamodels via some CASE tool, which allows the graphical editing of the ER diagrams. There are many tools which support these kinds of diagrams, and we may ask which of these tools might be the best for our purposes. There is no definite answer; it varies from case to case. The following are merely the major criteria in selecting a CASE tool for metamodel description:

1) The similarity of the CASE tool's metamodel to the metamodel of the internal data store.
2) Graphical representation of metamodels. It is important to have a graphical representation of both the source and the target metamodels of conversion. The main reason for this is the readability of the metamodel. It also makes it easier to make changes to the metamodel if necessary. The diagram is also very useful if conversion documentation is to be developed. The diagram is a must in the documentation if the conversion is described on the basis of metamodel concepts.

3) Convenient means for the export and import of data. A developed metamodel cannot be used if it is stored only in the repository of the CASE tool. The metamodel must be available during the conversion software runtime. This means that the metamodel must be stored in a file format that can easily be loaded into the internal data store at the beginning of the conversion process.

The first criterion looks very simple, at least at first glance. Although CASE tools which support ER diagrams are similar, there are often very significant differences in terms of the features of the ER modeling that they support. The main differences lie in support for the following features:

- n-ary relationships. Very few CASE tools support this feature, because it can easily be modeled using simple binary relationships. Support for n-ary relationships means that the role concept must be introduced in the metamodel;
- attributed relationships. This is another feature that can easily be modeled by using an additional ("associative") entity and simple binary relationships;
- multiple inheritance. This construct comes from the object-oriented analysis and can also be applied in the ER modeling area.

How complex do metamodels usually tend to be? Is there any need for the advanced features that are discussed above? That very much depends on the situation. If the tool's data is stored in a repository that uses such advanced features, then it is reasonable to use them in the metamodel, too. This allows you to preserve the real appearance of the metamodel in the data conversion data store, too. On the other hand, these advanced features are a rarity. Most CASE tools use relational data bases for repository and, due to implementation restrictions, no advanced features are usually present. If there is no data store of source or target data (i.e., a plain text file is used to store the data), the development of the metamodel is up to the developer. Experience shows that in such cases it is easy to avoid advanced modeling features.

When it comes to graphical representation of metamodels, two kinds of diagramming can be used. The metamodels can be developed by using either simple ER-diagrams (Martin and McClure, 1985) or object structure diagrams (Rumbaugh *et al.*, 1991). It is important that the entities (objects), relationships and attributes be present in the diagram.

After the development of the metamodel by the case tool, a model describing the metamodel is stored in the repository of the CASE tool. Most CASE tools offer data export ability, and the metamodel can be exported and stored in a data file. These data are of great interest in terms of the conversion runtime. The conversion development framework must support data transformation from the CASE tool data export format to some other format that can easily be imported by the conversion engine into the internal data store. Hence the following illustration (Fig. 4) can be drawn, representing the workbench of metamodel processing.

4.2. *A Conversion Description Language Compiler*

The conversion description compiler is one of the most powerful tools for data conversion development in this approach. This tool allows us to use the fourth-generation technique

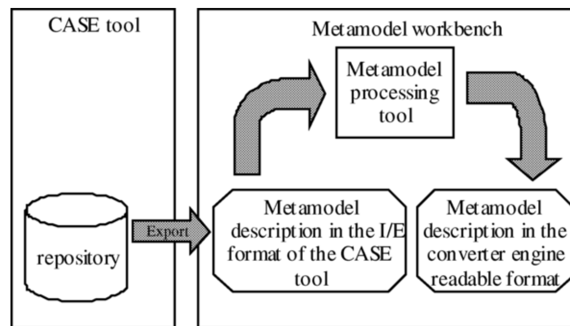

Fig. 4. The metamodel development workbench.

(4GT) approach (Pressman, 1994) in the converter component development. For purposes of data conversion description, the data conversion development framework offers a special-purposes, non-procedural language (Conversion Description Language – CDL) (Plume *et al.*, 1998). The description of the conversion is written in the CDL language, using metamodel concepts of both the source and the target data.

The language is compiled to the C language and further, using a standard C compiler to the executable code. If the CDL language constructs are not adequate for conversion description, C programming language fragments can extend the conversion description. The illustration of the CDL compilation is shown in Fig. 5.

The language compiler takes as input the conversion description and the metamodels for both source and target data, producing the C language text. The standard API calls are used in this generated C language text to access the internal data store. Further, with the help of a standard C compiler, executable code is produced. The C language is used here as an example. Actually, any other procedural programming language can also be used for the compilation.

The main concept in the conversion description language (CDL) is the rule. This concept is very much similar to the concept of the Prolog language. The practice of using the language compiler indicates that very seldom are more than 100 rules necessary in order to develop a conversion description.

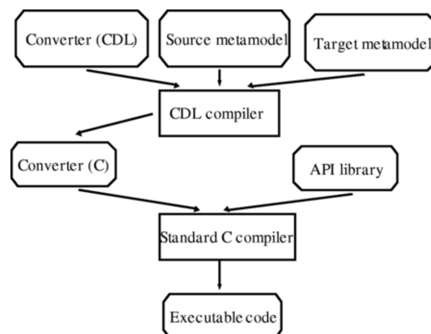

Fig. 5. Conversion description compilation.

The use of a language such as CDL is crucial to preserve the maintainability and modifiability of the data conversion software. Through the language, the conversion description can be modified easily, and this is a very powerful way to deal with the changing requirements of the customer.

5. Conclusion

This article has illustrated a systematic approach to the data conversion development process. This approach is supported by software tools that are designed to perform specific activities in the development process. Namely, conversion description development is supported by a special-purpose language, while metamodel development is supported by the metamodel development and processing tools.

As the tools have proven to be very effective, the next obvious step would be tool integration. The possible consequences of such integration, however, must still be investigated in order to ensure that it is worth doing.

References

Aho, A.V., R. Sethi, J.D. Ullman (1986).*Compilers, Principles, Techniques and Tools*. Addison-Wesley.

Barzdins, J., J. Tenteris, E. Vilums (1998). Biznesmodelesanas valoda GRAPES-BM 4.0 un tas pielietosana, Riga (In Latvian).

EIA/IEEE (1995). *Software Life Cycle Processes. Software Development. Acquirer – Supplier Agreement. Standard for Information Technology, J-STD-016*. The Institute of Electrical and Electronics Engineers, Inc.

EIA/PN-2387 (1993). *The CDIF Framework for Modeling and Extensibility*, Electronic Industries Association, Washington.

IEEE (1991). *Standard for Developing Software Life Cycle Processes, Std 1074*. The Institute of Electrical and Electronics Engineers, Inc.

Ince, D. (1995). *Software Quality Assurance – A Student Introduction*, McGraw-Hill.

GRADE version 4.0 (1997). *Business Modeling Language Reference Manual*. Infologistik GmbH.

Kellner, M.I., G.A. Hansen (1988). Software process modeling. *Technical Report CMU/SEI-88-TR-9*, Software Engineering Institute, Carnegie Mellon University.

Levine, J.R, T. Mason, D. Brown (1992). *LEX and YACC*, O'Reilly & Associates.

Martin, J., C. McClure (1985). *Diagramming Techniques for Analysts and Programmers*. Prentice Hall, Englewood Cliffs, N.J.

Paulk, M.C., B. Curtis, M.B. Chrissis, C.V. Weber (1993). Capability Maturity Model for Software, Version 1.1, *Technical Report CMU/SEI93-TR-024*, Software Engineering Institute, Pittsburgh.

Plume, J., J. Strods, I. Karlsons, U. Smilts, J. Smotrovs, G. Gavars, I. Stasko, M. Dancis (1998). Metamodel Based Data Conversion. In J. Barzdins (Ed.), *3rd International Baltic Workshop on Databases and Information Systems*, Vol. 1, Latvian Academic Library, Riga. 175–186.

Pressman, R.S. (1994). *Software Engineering. A Practicioner's Approach*, 4th ed., McGraw-Hill.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N.J.

J. Plume was born in Sigulda, Latvia, on 1972. He received a BS degree (1994) and a master's degree (1996) in computer science from the Faculty of Physics and Mathematics of the University of Latvia. He is currently a doctorate student at the university. His research interests focus on software process improvement. The research is being done under the auspices of the Riga Information Technology Institute, a private non-profit software research and engineering company.

Duomenų konvertavimas: instrumentinis požiūris

Janis PLUME

Straipsnyje aptariamas instrumentinis požiūris į duomenų konvertavimą. Išnagrinėtas duomenų konvertavimo proceso gyvavimo ciklo modelis ir kiekvienai jo stadijai pasiūlytas instrumentų rinkinys, padedantis greičiau ir efektyviau atlikti toje stadijoje atliekamus darbus.