

Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS



Michal Tomčányi

CTL-to-Java Compiler for CloverETL

Department of Software Engineering
Supervisor: Mgr. Václav Matouš
Study program: Computer Science, Database Systems

I would like to thank my advisor Václav Matouš for guidance, Martin Zátpek, Jaro Urban and David Pavlis from CloverETL development team for advice and Ling for her unconditional support.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 16th April 2009

Michal Tomčányi

Contents

1	Introduction	8
1.1	Motivation	8
1.2	Overview	9
2	CloverETL	10
2.1	Tools	10
2.2	Purpose	10
2.3	Records and metadata	11
2.4	Components	11
2.5	Ports	12
2.6	Edges	12
2.7	Lookup tables	13
2.8	Sequences	13
2.9	Transformations as graphs	14
2.10	Graph assembling	14
2.11	Graph execution	15
2.12	Graph termination	15
3	CTL Language	16
3.1	Evolution	16
3.2	Specification	17
3.3	Lexical structure	17
3.3.1	Comments	17
3.3.2	White spaces	18
3.3.3	Identifiers	18
3.3.4	Keywords	18
3.3.5	Literals	19
3.3.6	Separators	19
3.3.7	Operators	20
3.4	Types and values	21
3.4.1	Record type	21
3.4.2	Primitive types	22
3.4.3	Container types	22
3.4.4	Function types	23
3.4.5	Expressions and type conversions	23
3.5	Declarations, scopes and name resolution	23
3.5.1	Function scope	24
3.5.2	Global scope	24
3.5.3	Eval scope	24
3.5.4	Name resolution	25
3.5.5	Records and fields resolution	25

3.6	Code execution	26
3.6.1	Interface RecordTransform	27
3.6.2	Interface RecordNormalize	28
3.6.3	Interface RecordDenormalize	28
3.6.4	Interface Partition	29
3.7	Lookups and sequences	30
3.8	Features summary	31
4	CTL Revised	32
4.1	Strong typing	32
4.1.1	Justification	32
4.1.2	Amendment	33
4.2	Unified use of records	33
4.2.1	Justification	33
4.2.2	Amendment	34
4.3	Implicit type conversion	34
4.3.1	Justification	34
4.3.2	Amendment	35
4.4	Function overloading	35
4.4.1	Justification	35
4.4.2	Amendment	36
4.5	Name-based access to graph elements	36
4.5.1	Justification	36
4.5.2	Amendment	36
4.6	Single assignment operator	37
4.6.1	Justification	37
4.6.2	Amendment	38
4.7	Iteration through fields	38
4.7.1	Justification	38
4.7.2	Amendment	39
4.8	Others	39
5	Compiler Design	41
5.1	Implementation language	41
5.2	Supported Java version	41
5.3	On Source to Source Translation	41
5.4	Interpreted and compiled execution	42
5.5	Architectural fit	42
5.6	Compiler modules	43
5.6.1	Syntactic analysis	43
5.6.2	Semantic analysis	43
5.6.3	Type checking	43
5.6.4	Flow control	44

5.6.5	Runtime initializer	44
5.6.6	Code translation	44
5.7	Representation of values	44
5.8	Standard functions	45
6	Compiler Implementation	46
6.1	Parser	46
6.1.1	Tokenizer	46
6.1.2	Error recovery	47
6.2	Resolver	48
6.2.1	Semantic checking	48
6.2.2	Element references binding	48
6.2.3	Type extraction	49
6.3	Type checker	49
6.3.1	Types representation	49
6.3.2	Function resolution	51
6.4	Flow control	51
6.5	Translator	52
6.5.1	Code unit translation	53
6.5.2	Types	53
6.5.3	Assignments	54
6.5.4	Expressions	54
6.5.5	Looping statements	55
6.5.6	Branching statements	56
6.5.7	Global scope	56
6.5.8	Functions	56
6.5.9	Lookups tables	57
6.5.10	Sequences	57
7	Standard library	58
7.1	Plugins	58
7.2	Function registration	58
8	Intepreter	60
8.1	Function calls	60
8.2	Lookup tables	60
9	Conclusion	61
9.1	Known limitations	61
9.2	Supported validation functions	62
9.3	Supported components	62
9.4	Future work	62
	References	64

A CD Contents	66
B Installation and usage	66

Abstract

Název práce: CTL-to-Java Compiler for CloverETL

Autor: Michal Tomčányi

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Václav Matouš

e-mail vedoucího: vaclav.matous@mff.cuni.cz

Abstrakt:

CloverETL je skupina nástrojů pro integraci dat, umožňující snadný vývoj datových transformací. CTL je jazyk prostředí CloverETL pro popis manipulací s daty. Tato práce se zabývá překladem programů v CTL do zdrojového kódu jazyka Java tak, aby výsledný kód mohl být přeložen a spuštěn jako součást datové transformace CloverETL. Práce diskutuje podstatné vlastnosti CTL (i CloverETL) a předkládá seznam změn které umožní jeho translaci do jazyka Java. Pro upravený jazyk CTL je navržen překladač i výkonná část umožňující spuštění kódu v rámci datové transformace. Výsledkem práce je funkční implementace překladače CTL která je plně začleněna do prostředí CloverETL a může být dále rozšiřována.

Klíčová slova: *CloverETL, implementace CTL, Java, překladač*

Title: CTL-to-Java Compiler for CloverETL

Author: Michal Tomčányi

Department: Department of Software Engineering

Supervisor: Mgr. Václav Matouš

Supervisor's email address: vaclav.matous@mff.cuni.cz

Abstract:

CloverETL is a set of data integration tools aimed for rapid development of data transformations. CTL is a proprietary language of CloverETL for definition of data manipulations. Our goal is to design and implement translator of CTL programs into Java source code which can be compiled and executed as part of CloverETL data transformation. The work discusses key features of CTL (and CloverETL) and proposes necessary changes to translate CTL into Java. Design and implementation of compiler and runtime environment for amended CTL is presented afterwards. The result is a working implementation of compiler for the CTL language allowing future extensions, fully integrated with CloverETL environment.

Keywords: *CloverETL, CTL implementation, Java, compiler*

1 Introduction

CloverETL is a set of data integration tools aimed for rapid development of data transformations. CTL (Clover Transformation Language) is a proprietary language of CloverETL for definition of data manipulations.

The main goal of this work is to design and implement a compiler for CloverETL [1] environment translating CTL programs into Java source code while focusing on improving language syntax and runtime performance.

The result is a working compiler for amended language capable of generating Java source code for a broad set of CTL programs. The compiler is integrated with CloverETL transformation components thus CTL program can be translated into Java source code, compiled into bytecode and executed as part of CloverETL data transformation.

To allow translation of CTL to Java the language was redesigned, mainly to enable sufficient code validation, remove known limitations and to improve its future extensibility. As a result, language runtime infrastructure was significantly modified to improve runtime performance and allow easier development of new functions.

As the CTL language is inseparably closely to CloverETL environment an introductory section about its purpose and workings is provided.

1.1 Motivation

Main motivation for the thesis is the experience from using CloverETL tools in commercial data processing projects. Data transformations must be usually developed in a short time while runtime stability and high processing throughput is required. Additionally a readability of the overall transformation is a must, especially for the purpose of auditing and maintainability. The appearance of CTL in CloverETL simplified development and improved readability of transforming code as its notation is much simpler than of Java that had been used earlier.

Unfortunately the current CTL implementation suffers from a lack of code validation, unclear behavior and sometimes confusing syntax. Missing proper code validation naturally leads to many errors being discovered only in transformation runtime, in some cases even after large set of data has already been processed. In some cases the errors only appear when code is tested on a specific data set as the program enters a previously unvisited branch. Obviously this complicates transformation development and prolongs necessary testing.

The idea of a compiler translating CTL into Java arose to address the previous issues. Compiler thoroughly validating CTL code at the development time allows fixing the errors before the transformation enters testing phase.

Additionally translation of CTL into Java can help improving transformation runtime performance at almost no development cost as the CloverETL allows using Java transformation code from early days of its existence.

1.2 Overview

The work explains key concepts of CloverETL in Section 2 and CTL in Section 3. Section 4 discusses amendments to CTL and their motivation. Key design decisions are summarized in Section 5, followed by implementation specifics for compiler in Section 6, standard library in Section 7 and interpreter in Section 8.

2 CloverETL

This section sets up context for our work to readers unfamiliar with CloverETL. It explains in brief the purpose of data transformations and the basic concepts behind CloverETL.

It explains common terms used with CloverETL transformations, then gives insight into how transformations are assembled and executed.

2.1 Tools

At the time of writing there are three tools in the CloverETL family; all based on the Java [2] platform

- *CloverETL Engine* is an open source Java library allowing to assemble and execute data transformations via Java API calls or based on definition in XML [3] notation.
- *CloverETL GUI* is a visual designer of data transformations, developed as a plugin for the Eclipse platform [4]. It produces data transformation definitions in XML format.
- *CloverETL Server* is a production platform for continuous parallel execution of CloverETL data transformations in enterprise deployments.

The Engine library is common for both the GUI as well as the Server. Used mostly for on-the-fly code validation during development in GUI, the Server uses Engine to assemble and execute the transformations.

The library offers basic building blocks that form the transformations and allow further extensions via plugging functionality. It contains predefined extension points allowing dynamic registration of new modules implemented in Java.

CTL language (parser, interpreter) is implemented in the core of the Engine, while libraries with validation functions are distributed separately as Engine plugins.

2.2 Purpose

CloverETL is a typical member of ETL¹ software family, oriented on retrieval of data from heterogeneous sources, their modification and restructuring to

¹Extract-Transform-Load

various output formats.

Common data sources (or targets) are text or binary files, XML documents, database tables or messaging middleware. The volume of data processed in single execution ranges from hundreds of megabytes to hundreds of gigabytes, while expected processing time varies from minutes to hours.

Concept of ETL is used especially in data warehousing to extract data from operational systems and prepare it for load into dimensional model in data presentation area [5]. However ETL tools find much broader use also in other areas than data warehousing. They are often used as initial analysis and profiling tools to determine data quality, as high-performance processing platform in mass data migrations, but also operate in transaction mode as easy to maintain translators for messaging middleware.

2.3 Records and metadata

CloverETL represents data independently from their source or destination to allow processing of arbitrary formats. It uses a concept of abstract *records* to represent elements of processed data.

Record structure consist of named, strictly typed fields with fixed position within the record, usually matching the order of data elements in input or output. The structure is predefined for each record by *metadata* descriptor.

Except storing structure information the metadata also contain formatting information that allows records to be either parsed from input source or formatted into end target.

Formatting on record level provides information such as field delimitation and quoting, types of newline character etc. For fields it holds information such as date and numeric formatting patterns.

2.4 Components

The actual transformation of the data happens in series of modification steps subsequently applied on the data records. The steps are represented by transformation *components*. Components serve different purposes and can be roughly split into the following categories:

- *Readers* parse input from data sources into the form of abstract records.
- *Writers* serialize data records into format of the output.
- *Transformers* modify content or structure of records.

- *Sorters* change overall order in which the records are processed.
- *Joiners* combine two or more input records into single output record.
- *Executors* triggered by incoming records, execute external programs, transformations or manipulate data sources.

Each category contains several types of components specialized for some manipulation with the data. To improve reusability of components, these can be further configured by a set of text-based properties affecting their behavior. Thus a single component may appear more than once in a single data transformation performing slightly different operation based on its actual configuration.

With some components the properties serve as simple parameters while the component behavior is more or less predefined. Opposingly, many components allow embedding Java or CTL code in their properties thus allowing much greater level of customization.

2.5 Ports

Ports are input and output points of transformation components. Number of ports for a component varies with its type, designation and configuration. Generally speaking, there is no upper limit on the number of ports for a component² but also having a component with no input ports or no output ports is not an exception. For the purpose of identification both groups of ports are uniquely numbered.

Records enter the component through input ports and after processing leave the component through output ports. Component inputs and outputs form disjunct sets, i.e. every port serves either as an input or an output.

2.6 Edges

Edges serve as channels transporting records between components. This allows sequencing of components in order to build a complex processing scenario - the very basic idea behind constructing CloverETL data transformations.

Every edge connects a pair of ports from different components. Edges can be thought of as directed as they always lead from one component's output port to other component's input port.

²For example Merge component can collect records from any number of input ports.

Each edge in transformation must be assigned a metadata descriptor. In runtime all records flowing through the edge obey structure prescribed by the descriptor. This explains another important role of edges - they serve as format carriers.

Records within edge remain unchanged and leave the edge in the same order as they entered it. Under normal circumstances edges pass records instantly, in memory. In special cases however, when there is a significant difference between processing speed of connected components the edge serves as data buffer, spooling data into a temporary file.

2.7 Lookup tables

Lookup tables are data structures allowing fast access to their values based on a key. Values of lookup tables are data records; all stored records must obey the format defined by metadata descriptor that is part of lookup table definition. Lookup key is formed by one or more fields of stored data records. Lookup table may optionally allow storing multiple records under the same key.

Lookup tables may be used by components from within the transformation code. Generally the transformation code is only allowed read access to the lookup table, while ad-hoc insert operations are not supported. This is to prevent a necessity of having a locking mechanism that would be required with parallel access into table from multiple components.

Lookup tables are either populated automatically from a predefined data source, or loaded dynamically from within transformation by a dedicated component. Transformation code in component may trigger disposing or repopulation of lookup table at any time.

There is a variety of uses for lookup tables in data transformations. Mostly they serve as a source of predefined translation mappings for field values.

CTL provides the following operations for lookup tables: retrieving value stored under specific key, finding the number of values stored under a key and their iterative reading, disposal and repopulation of particular table.

2.8 Sequences

Sequences are thread safe autoincrementing counters capable of generating series of unique numeric values. Optionally they offer persistence, meaning they retain their value over consecutive executions of transformation where they are used.

Similarly as lookup tables the sequences are used by transformation code in components. They support three elementary operations: *next*, *current* and *reset*. Operation *next* returns actual value of the internal counter then increments it with predefined step. Operation *current* returns actual value of the counter without incrementing and *reset* sets counter back to the starting value.

2.9 Transformations as graphs

Assembled CloverETL data transformation can be viewed as a directed graph having transformation components as nodes connected by edges. The directed graph representation is actually used by the CloverETL GUI to visualize data transformation in more comprehensible format than its XML notation.

A data transformation implemented in CloverETL is therefore often referred to as *transformation graph*, defined by its configuration and layout. Configuration contains definitions of graph elements - components, metadata, lookup tables, sequences and others. The layout defines linking of ports with edges and assignment of metadata to individual edges.

Every element definition contains a string identifier that can be used as a unique reference from other graph elements.

2.10 Graph assembling

CloverETL Engine supports assembling data transformations programmatically using Java API or from graph XML definition. Graph XML definition contains configuration of all graph elements stored in XML nodes. In the process of graph assembling the information is extracted from XML and its nodes are converted into Java objects.

Part of the graph assembling process is a compilation of embedded Java code. Components that support code embedding define Java transformation interface that the embedded code must implement in order to be compiled and executed by the component.

Embedded CTL code is handled differently. It is parsed and passed to component's interpreter instance. The interpreter is wrapped in an adapter class providing a bridge between component's transformation interface and universal API of the interpreter.

Providing that no issues are found during graph assembly the transformation is executed.

2.11 Graph execution

Transformation execution is data driven and employs pipeline parallel processing. Once a record is processed by a component it is immediately passed through the edge to the following component for further processing. All components run in parallel as separate threads.

Components transform records in processing cycles. In single cycle the component reads records from all input ports and calls the transformation code (if any) to process records. Transformation code may produce multiple records and pass them to arbitrary connected output ports.

Component invokes transformation code by calling methods in Java transformation interface. In case of CTL the call is delegated to the interpreter which in turn evaluates expression or invokes a CTL function.

The transformation code cannot communicate with other components in any other way than writing records to output ports. However as mentioned before it can access external graph elements such as lookups or sequences.

2.12 Graph termination

Life of transformation component ends when all data has been processed. This can result either from no more data coming to input ports or from component having exhausted its internal data source³. When component detects such state it exits the main processing loop and closes all of its output ports. Immediate neighbouring components replicate this behavior and termination event propagates throughout the whole data transformation.

The same approach is taken in case a component encounters a runtime error or is deliberately stopped. Additionally to the previous steps the component sets an error indicator that is recognized and reported by watchdog thread monitoring the execution.

Graph transformation is successful providing that all components finished running and none of them exited with erroneous status.

³Relates to Reader components.

3 CTL Language

CTL is a proprietary scripting language oriented on data processing in transformation components of CloverETL. It is designed to allow simple and clear notation of how data is processed and yet provide sufficient means for their manipulation.

Language syntax resembles Java with some constructs common in scripting languages. Although scripting language itself, CTL code organization into function resembles structure of Java classes with clearly defined methods designating code entry points.

CTL is a high level language in both abstraction of processed data as well as its execution environment. The language shields programmer from the complexity of overall data transformation, while refocusing him to develop a single transformation step as a set of operations applicable onto all processed data records.

Closely integrated with CloverETL environment the language also benefits the programmer with uniform access to elements of data transformation located outside the executing component, operations with values of types permissible for record fields and a rich set of validation and manipulation functions.

During transformation execution each component running CTL code uses separate interpreter instance thus preventing possible collisions in heavily parallel multi-threaded execution environment of CloverETL.

This section does not seek to provide formal specification of CTL language. Instead it introduces the language from lexical, semantic and runtime perspective highlighting practical features as well as limitations that had to be considered when designing the compiler.

3.1 Evolution

CTL developed from record-access macros originally used in embedded Java transformation code. The current syntax for accessing processed record still resembles the original macros. It further evolved into expression language for definition of conditions supported by a limited set of filtering components.

The language was further expanded with control statements and support for user-defined functions. Finally the standard library with validation functions was added to comfort the most common tasks needed in data manipulation. At the time of writing, CTL is widely supported by CloverETL components and is the preferred alternative of writing transformation code.

3.2 Specification

There is no formal specification of CTL language as its existence and field of usage is only limited to the environment of CloverETL. The most complete documentation of the language can be found in [6]. Next to providing information about elementary language concepts it also includes detailed overview of existing validation functions.

The documentation was almost complete after the initial release of CTL, however in two years of language existence some parts of the documentation became outdated and newer features were not documented. Therefore the most reliable source of information on language structure and behavior was Java source code of the original implementation as well as test suite encompassing the interpreter. Knowledge of CTL limitations comes mostly from author's own experience using the language.

3.3 Lexical structure

CTL programs are written using UTF-8 character encoding. Unicode code points are allowed to occur only within string literals⁴, but are not to be part of any other input elements (such as identifier or function names). CTL supports Java Unicode escape sequences which are automatically translated to corresponding Unicode code points before input stream is split into lexical elements.

Language lexical elements form sets typical for other programming languages: comments, white spaces and tokens. Tokens are further split to identifiers, keywords, literals, separators, and operators. CTL is generally case sensitive with exception of some operators.

3.3.1 Comments

CTL supports both types of comments typical for all C-style programming languages. Text enclosed by `/*` and `*/` is treated as possibly multi-line comment and ignored. The sequence of two forward slashes `//` introduces the comment spanning until end of line.

The first comment in CTL code may have a special meaning if it contains the string `#TL`. This string is scanned for by transformation components to determine that embedded code is CTL and not Java.

⁴Commonly used for matching string fields with international characters.

3.3.2 White spaces

The character treated as white spaces are the ASCII space character, horizontal tab as well as any combinations of carriage return and line feed characters⁵.

3.3.3 Identifiers

To avoid lengthy definition of allowed identifier values we say that CTL identifiers follow the same regulations as Java identifiers with the exception of using only ASCII characters. Identifiers are case sensitive thus `customerName` generally refers to a different entity than `CustomerName`.

3.3.4 Keywords

CTL contains a set of keywords common for other programming languages and further expands this set with keywords specific for CloverETL and data processing. The keywords can be split into the following groups based on their typical use:

1. **Type names**

`boolean, decimal, int, long, numeric, date, string, record, list, map`

2. **Control statements**

`break, case, continue, default, do, else, for, if, return, switch, while;`

3. **Date fields**

`millisec, second, minute, hour, day, week, month, year;`

4. **Error levels**

`error, warn, info, trace, debug;`

5. **Literals**

`true, false, null; and`

6. **Special code units**

`function, import`

⁵This is important as CloverETL runs on variety of software platforms

7. Lookups and sequences

`lookup`, `lookup_admin`, `lookup_found`, `lookup_next`, `sequence`

Date fields group can be used to identify parts of date-time values. Keywords from log level group can be passed to `print_log` function in order to assign message an error level recognized by component's logging system.

3.3.5 Literals

CTL has literals for all primitive types, `null` type as well as literal for the container `list` type. Numeric literals can be represented by their decimal, hexadecimal or octal value. Integer and decimal literals can additionally be suffixed by a distinguisher character. Character `l` or `L` suffixing an integer literal tells the parser to treat the literal as of type `long` instead of type `int`. Character `d` or `D` suffixing a decimal literal tells parser to treat the literal as of type `decimal` instead of type `double`.

Literals of `string` type are either single (`'literal'`) or double quoted (`"literal"`). Single quoted string literal can span over multiple lines. For such case the line terminator must be escaped by preceding it with backslash character.

Literals of the `date` type are character sequences of format *yyyy-mm-dd HH:MM:SS*, where the first part describes the date value while the second part describes the time value. Both parts are separated by ASCII space character. Date part consists of four digits for year value, two digits for month in the year, and two digits for value of day in month. Time part contains two digits for hour value, two digits for minute value and finally two digits for seconds.

Literals of the `list` type have syntax similar to Java array initializers. They are specified by comma-separated list of of primitive type literals enclosed by square brackets. Numeric literals prefixed by minus character representing negative numeric values that would be normally treated as expressions may also be part of list literals. However any other expressions are not allowed. As `list` type supports arbitrary types of its elements the literals do not necessarily have to be of the same type.

3.3.6 Separators

CTL uses separators common for most of the programming languages: curly brackets (`{ }`), brackets (`()`), square brackets (`[]`), semicolon (`;`), comma (`,`) and period (`.`). As opposed to many scripting languages it is not possible

to separate statements using line terminators; instead the semicolon serves this purpose.

3.3.7 Operators

Except the standard set of arithmetic, boolean and relational operators the language offer several uncommon ones, namely:

- **Record access operator** ($\$$). Indicates access to input or output record. Its parameter may be an identifier (access by name) or number (access by position i.e. port number). It must be immediately followed by a field selection operator (\cdot).
- **Mapping assignment operator** ($:=$). Used only for assignments into fields of output data records.
- **String concatenation operator** ($+$). Used for string concatenation. Triggers conversion of value to its string representation.
- **Operator 'matches'** ($\sim=$). Validates if a string value on left-hand side matches the regular expression on the right-hand side.
- **Operator 'contains'** ($?=$). Validates if a string value on left-hand side contains at least one match of the regular expression on the right-hand side.
- **Operator 'in'** ($\cdot\text{in}\cdot$). Tests if the value of expression on left-hand side is included in a `list` or a `map` on the right-hand side.
- **Conditional assignment operator** ($:$). Instructs the interpreter to try assigning values on right-hand side (until first of them succeeds) providing that the assignment to an output field on left-hand side had failed.

Many of the operators in CTL do also have equivalent textual notation as shown in Listing 1. This textual notation of operators is case insensitive.

```
// traditional syntax of operators
boolean r1 = a <= b;
boolean r2 = ( a == b ) || r1;

// equivalent notation of the above
boolean r1 = a .le. b;
boolean r2 = ( a .EQ. b ) OR r1;
```

Listing 1. Textual notation of CTL operators.

3.4 Types and values

CTL is a weakly typed language in that the type checking happens at runtime, together with implicit type conversions and not all language elements declare their type at compile-time. Despite the previous the local variables are always declared with a type.

CTL type system contains three general groups of types:

- **record type** is the type of CloverETL data records;
- **primitive types** correspond to types permissible for record fields
- **container types** are the `list` and the `map`.

There is also a special **null** type having only single value represented by the null literal. It is impossible to declare a variable of this type however the null literal can be cast to any other type; practically it means that null value may be assigned to variable of any type.

3.4.1 Record type

Record type is the type of value representing CloverETL data records. As such, it is always bound with a metadata descriptor that must be defined in transformation graph. Defining a record type referencing nonexistent metadata is reported as a compile-time error.

Values of record types are data records obeying the structure prescribed by the metadata definition. The record structure therefore contains from named, ordered, strictly typed fields. The fields can be accessed by name or position within the record.

The following code listings show the relation between the metadata definition (Listing 2) and the record type declaration (Listing 3).

```
<Record id="Metadata0" name="Customer">
  <Field name="Name" type="string" delimiter=";" />
  <Field name="Age" type="numeric" delimiter=";" />
  <Field name="City" type="string" delimiter="\r\n" />
</Record>
```

Listing 2. XML definition of CloverETL metadata

```
record (Metadata0) customer;  
customer.Name = "Michal";  
customer.Age = 20.5;  
customer.City = "Prague";
```

Listing 3. Declaration and usage of variable of record type

3.4.2 Primitive types

The primitive types are derived from the record type in that they represent permissible types of record fields. From all CTL primitive types the `int`, `long`, `double`, `boolean` and `string` types are identical (and implemented) with their Java equivalents. There are additionally two primitive types that can be used for record fields: `decimal` and `date`.

The `decimal` is a numeric type and its values are numbers with predefined precision. It takes two integer parameters - *precision* and *scale*; precision being the maximum possible number of digits of unscaled stored value and scale being the number of digits to the right of the decimal point.

Values of `date` are instants of time with millisecond precision.

Primitive types are used in variable declarations, but can also be passed as arguments to library functions implementing explicit type cast operations.

3.4.3 Container types

There are two container types in CTL⁶ language: `list` and `map`. Variables of these types are containers capable of storing elements of arbitrary types. `List` represents an ordered collection supporting positional access to its elements, while `map` represent an associative array mapping from keys to values.

Values of these types are preallocated and grow automatically with new elements added. A container may contain values of mixed types, `null` values as well as other lists or maps.

⁶CloverETL implementation of type `bytearray` was stable enough to include it into our work.

3.4.4 Function types

User-defined functions declare neither types of their parameters nor their return type, while library functions do declare both. Standard library functions may additionally accept variable number of arguments⁷.

CTL therefore features some kind of *function overloading*, although correct behavior is neither checked or guaranteed.

3.4.5 Expressions and type conversions

The CTL defines a simple rule for calculating types of expressions: The resulting type of CTL expression is always given by the type of its first argument regardless of any other participating types. A direct implication of the previous is that widening conversions of numeric types common for other programming languages do not take place in CTL.

Implicit conversion to string takes place in CTL but differently from Java. If the first argument of binary + operator is a string type, the other value is converted to its string representation and expression value is concatenation of the two strings.

There is no language construct for explicit type conversion in CTL, however there are library functions implementing conversions between types.

3.5 Declarations, scopes and name resolution

CTL program has three scopes of variable visibility:

- **Function scope** is a scope of function body.
- **Global scope** is a scope of code outside of function body.
- **Eval scope** is a scope of code passed as an argument to the `eval` function.

Declaration of two variables sharing the same name within the same scope is treated as a duplicate and reported as compile-time error.

⁷Known in other languages as vararg functions

3.5.1 Function scope

Function arguments as well as any local variable declarations that are part of function body are only visible in function scope. Declaration of a local variable in function body may shadow a global variable providing that their share the same name. Local variables must not share name with any of function parameters. It is not possible to nest function declarations; i.e. a function declaration cannot be part of other function body.

3.5.2 Global scope

The global scope contains declarations of all functions, statements outside function bodies as well as any declarations included with `import` statement. Declarations from the global scope are visible within all function scopes as well as the eval scope.

Record passed to code by component act as if they were declared in the global scope so they are visible from within all scopes. Additionally they cannot be shadowed by any local variable declaration. This is achieved syntactically by using the `$`-operator to refer to a global record. Global records sharing the same metadata name however can shadow each other. In that case positional access using numeric index (corresponding to a port number) can be used.

Similarly to variables the function declarations are checked for duplicities. Any two function declarations in the global scope having the same name (regardless of their parameters) are treated as duplicates. Local function declaration may shadow visibility of a library function with the same name.

3.5.3 Eval scope

The language provides two built-in functions `eval` and `eval_exp` functions (we use `eval*` notation to refer to both of them) that evaluate their string argument as if it was a CTL code. While `eval_exp` function allows only evaluation of expressions, `eval` accepts even statements and function declarations.

The code executed by `eval*` function shares the global scope with the code from where the call to `eval*` originated. If a duplicate variable or function is redeclared within the code created by `eval`, it is reported as runtime error. Any function or variable declared in the global scope of the `eval*`-code is permanently added to the scope of originating code and can be used from successive calls to `eval*` function.

3.5.4 Name resolution

The name resolution takes place immediately when the code is parsed into AST representation. This means that function declaration must precede its first use and disallows declaration of recursive functions.

To determine entity represented by an identifier the context in which the identifier appears is examined and the following possibilities are evaluated in order until one of them succeeds:

1. If identifier is in the context of a `lookup` keyword the identifier is treated as a reference to graph lookup table.
2. If identifier is in the context of a `sequence` keyword the identifier is treated as a reference to graph sequence.
3. If identifier is in the context of a the `record` keyword the identifier is treated as a reference to graph metadata.
4. If identifier is preceded by "\$" the identifier is treated as a metadata name to a global record.
5. If identifier is preceded by "." the identifier is treated as a field name of a record.
6. If identifier is followed by "(" the identifier is treated as a function call.
7. If identifier matches a local variable declaration it is a reference to that local variable.
8. If identifier matches a global variable declaration it is a reference to that global variable.
9. The identifier referst to an undeclared variable.

As mentioned in the Section 3.5.1 when resolving function calls to respective function declarations the local functions are considered before standard library functions; i.e. local function declaration shadows standard library function declaration.

3.5.5 Records and fields resolution

Access to input or output record is indicated by using \$-operator before identifier name. To determine if an input or an output records is being accessed, its position relative to assignment operator is considered.

Record access expression on the left-hand side of assignment operator refers to an output record. At any other position the expression refers to an input record.

The data records are passed to CTL interpreter in separate arrays containing input and output records respectively. The arrays are indexed by port numbers i.e. data record in output array at position i represents a record that is to be sent through i -th output port, record at i -th position in input array arrived from i -th input port.

Based on the above the $\$$ -operator may be followed by an identifier or an integer literal. Identifier is used to match the corresponding record based on its metadata name. In the latter case the record is accessed by position (or port number) and the literal is used as an index to corresponding array. The positional access is used especially because metadata names are not guaranteed to be unique.

The record-selection expression must be immediately followed by a field selection expression introduced by the $.$ operator⁸. Similarly as records, the fields can be addressed by name of position in metadata definition.

3.6 Code execution

CTL code is always executed within context of transformation component in a separate interpreter instance. The component interacts with the embedded code by one of the following possibilities:

1. Evaluating a single CTL expression residing in global scope; or
2. calling a specific function of embedded CTL code.

The first is a legacy approach used only by a single component type, the `ExtFilter`, which based on a boolean expression possibly rejects records from further processing.

Instead, most of the CTL-enabled components use the second possibility as it is more flexible, but this approach requires presence of an adapter class. While component publishes methods it calls regularly in its processing cycles via Java interface, the embedded CTL code defines functions that perform data manipulation and may be called by interpreter. The interpreter is therefore wrapped into an adapter class specific for each Java transformation interface and in this form passed to the component. In runtime the adapter

⁸Enforced syntactically. There is however undocumented $\@$ operator that seems to access only records.

class translates invocations of transformation methods to invocations of CTL functions through interpreter.

Java transformation methods from the interface therefore uniquely map to specific CTL functions. That is why we refer to a set of CTL functions a component invokes as a *CTL interface* of a component.

When a components starts it first initializes CTL global scope, then it calls the user initializer function `init`. After the record processing has been finished and no more input records are coming the component calls the `finished` function. Both these function are optional and need not to be defined in embedded code.

Based on its processing algorithm the component calls additional methods from its CTL interface. Input records are repeatedly read by the component from its input ports and passed to the embedded CTL code through interpreter. After processing the records the component collects output records from interpreter and send them through corresponding output ports.

The following sections give an overview of CTL interfaces used by various component types. Each interface carries the name of the corresponding Java transformation interface.

3.6.1 Interface RecordTransform

Components performing a simple transformation of multiple input records into multiple output records use the `RecordTransform` interface. The interface requires presence of function `transform`. An example use of the interface in a component with two ports is shown in Listing 4.

```
function transform() {
    // copy customer unchanged from input port 0 to
    // output port 0
    $0.customerFirst := $0.first;
    $0.customerLast := $0.last;
    // build customer full name on output port 1
    $1.fullName := $0.first + " " + $0.last;
}
```

Listing 4. Simple transform function for a multi-port transformer.

The `RecordTransform` interface is used by `Reformat` component as well as all components from joiners category (Section 2.4).

3.6.2 Interface RecordNormalize

RecordNormalize interface is used only by a single component, the `Normalizer`. The component decomposes a single record into multiple records usually populating the new records with values from the original. An example usage of Normalizer component is given in the Listing 5.

The interface defines three functions:

- Required `count` function returns an integer value indicating how many records to produce.
- Required `transform` function with single integer argument to populate new data records.
- Optional `clean` function is called after all output records have been populated.

```
list [] phones

function count() {
    // extract phones from string separated by |
    phones = split($0.phoneList, '|');
    // return number of phones
    return length(phones);
}

function transform(i) {
    // send out the i-th phone number
    $0.phone := phones[iteration];
}

function clean() {
    // clean extracted phones
    truncate(phones);
}
```

Listing 5. Normalization of records with arbitrary number of phones.

3.6.3 Interface RecordDenormalize

The RecordDenormalize interface is a counterpart to the RecordNormalize interface. It builds up a single output record from an arbitrary number of input records. Listing 6 shows an example use of RecordDenormalize interface.

The interface defines three functions:

- Required `addInputRecord` function is called for every input record in the group.
- Required `getOutputRecord` function is called once for each group to build the output record.
- Optional `clean` function is called after the group is processed.

```
list [] phones

function addInputRecord() {
    // save customer phone
    phones [] = $0.phone;
}

function getOutputRecord() {
    // build and send out the phone list
    $0.phoneList := join(phones, '|');
}

function clean() {
    // clean extracted phones for next input
    truncate(phones);
}
```

Listing 6. Building list of phones with variable size.

3.6.4 Interface Partition

Partition interface defines a multi-way filter. It allows routing an input data record into any of connected output ports. It is used by a single transformation component `Partition`. The interface contains a single required function `getOutputPort` returning target port number. Function `init` in `Partition` interface accepts integer argument specifying number of connected output ports.

```

function getOutputPort() {
    // debtors: must notify
    if ($0.balance < 0) { return 0;}
    // everything ok
    if ($0.balance == 0) { return 1; }
    // overpaid: must refund
    if ($0.balance > 0) { return 2; }
}

```

Listing 7. Account balance check with Partition interface.

3.7 Lookups and sequences

It is possible to use lookup tables and sequences within CTL transformation code. Both can be referred to using the unique identifier in their definition in transformation graph. CTL provides the following operations for lookup tables:

- `lookup` performs lookup operation in specified table using any number of values as lookup keys. Lookup table reference is specified by identifier.
- `lookup_next` retrieves next value (if any) from lookup table stored under key accessed by the last `lookup` call.
- `lookup_found` returns number of values stored under key last accessed by `lookup` call.
- `lookup_admin` initializing (keyword `init`) or disposing (any other parameter) a lookup table.

The sequences operations have syntactically also form of function calls introduced by keyword `sequence` followed by identifier and expected return type of sequence value. A selector of operation follows:

- `next` retrieves sequence value and increments internal counter.
- `current` retrieves sequence value without incrementing.
- `reset` set the counter to the value specified in sequence definition.

3.8 Features summary

CTL is a high-level loosely-typed scripting language that can be embedded into CloverETL components. Most of the type conversions are performed implicitly and explicit type casting is only possible via standard library functions.

It has elementary string processing features for regular expression matching and concatenation. The language is dynamic in that it supports functions with variable arity and dynamic code execution through eval* functions.

Close integration of CTL with CloverETL allows access to other elements of data transformation such as metadata, lookups and sequences.

4 CTL Revised

Originally we anticipated to build the Java translator on the top of the existing language implementation. However already in an early stage of analysis it was realized that current CTL is too loose to allow straightforward generation of compilable Java code. Generating valid Java code though was crucial in order to prevent user from tedious debugging and necessity to track errors from Java code back to CTL.

The CTL was therefore redesigned, mostly to allow sufficient code validation that would enable generating compilable Java code. Part of the redesign also focused on removing known drawbacks and limitations

This section gives overview of most important changes to the language and motivation behind them.

4.1 Strong typing

4.1.1 Justification

CTL loose typing is not a good fit into strongly-typed environment of CloverETL. The elementary type of CloverETL - the data record - consists of fields that themselves are strictly type. The original transformation code in Java generally followed this principle because it either operated with data records, or Java primitive types - again, strongly type

Introduction of CTL loose typing discontinued this approach but brought no advantage. It can be viewed more as an absence of proper implementation than an advantage for programmer. The loose typing delegates discovery of type-related error to transformation runtime and in our case would cause many Java compilation errors. Unfortunately the data driven execution model of CloverETL may cause some parts of the transforming code only to be evaluated for a specific record⁹, which may lead to following:

1. Error is never discovered during testing as the erroneous code is never executed.
2. Error is discovered only after lengthy testing because the activating record comes only at end of data set.

⁹For example an if-statement with condition examining a data field.

4.1.2 Amendment

CTL was redesigned to be strongly typed. With variable declarations already containing type information the introduction of type checking mostly affected container types and functions. Container types must declare the element type, while user functions must declare return type as well as types of their arguments.

Naturally, strict typing for functions required introduction of `void` type for functions not returning any value and resulted in implementation of function overloading both in local code as well as the standard library. (Section 5.8

A strict type checking was further extended to validation of lookup tables and sequences access. For lookup tables the actual arguments of lookup operation are validated against lookup table keys, while using the `record` returned by table in further type checking. The sequences support three possible return types explicitly set by user: `int`, `long` and `string`. Therefore the sequences can be integrated into type checking algorithm. Rectified types and functions are shown in the Listing 8.

```
int [] intList; // list declaration
map[string,int] mapVar; // map declaration
// function declaration
function int sum(int a, int b) {
    return a+b;
}
```

Listing 8. List, map and function declaration with strong typing.

4.2 Unified use of records

4.2.1 Justification

Despite semantically representing the same entity - data record - the global records are treated differently in CTL than local variables of `record` type both on syntactic as well as operational level.

The following is an overview of differences between global records and local variables of `record` type.

Global records

1. Reference to a global record requires use of `$`-operator.
2. Data record cannot be retrieved as an R-value.
3. Record fields are accessed by the field-selection `.` operator.
4. Assignment to record fields uses the mapping-assignment `:=` operator.
5. Wildcard mapping allows copying multiple field values between two records.

Local variables

1. Variables of `record` type do not use the `$` operator.
2. Data record can be retrieved as variable R-value.
3. Record fields are accessed using the array access `[]` operator.
4. Assignment to variable uses plain assignment `=` operator.
5. It is impossible to assign multiple fields in a single expression.

4.2.2 Amendment

It was decided to retain the use of `$`-operator as it highlights access to component global inputs and outputs. In reference to (2), the R-value of both local variables as well as global records the R-value is the underlying data record. Access to record fields (3) was unified and uses the field-access operator. The possibility of mass-copying record fields (4) was extended also onto local variables as well. Usage of modified records is shown in Listing 9.

```
CustomerRecord customer;  
customer.* = $0.*;  
customer.validated = true;  
$validatedCustomer.* = customer.*;
```

Listing 9. Accessing and operations with global and local records.

4.3 Implicit type conversion

4.3.1 Justification

Implicit type conversions cause CTL mixed-type operations to behave in a confusing manner and are inconsistent with what can be seen in other programming languages.

Missing widening numeric types conversions may lead to unexpected results especially with mixed-type numeric operations. Similar situation arises when the operator `+` is applied on a numeric¹⁰ and a string argument (in this order) with programmer expecting to perform string concatenation but instead the string argument is (unsuccessfully) cast to the numeric type resulting in runtime error. Both scenarios are shown in the Listing 10.

¹⁰Or any non-string type.

```
decimal d = 10.12;
int i = 10;
decimal r1 = d + i; // r1 is 20.12
decimal r2 = i + d; // r2 is 20
string concat = i + " value"; // causes runtime error
```

Listing 10. Implications of missing widening type conversions.

4.3.2 Amendment

Widening primitive type conversions are applied when applicable

- `int` to `long`, `double`, or `decimal`
- `long` to `double`, or `decimal`
- `double` to `decimal`

Additionally an implicit conversion of any type to `string` takes place providing that the `+` operator has either of its argument of type `string`.

4.4 Function overloading

4.4.1 Justification

The original CTL design does not allow user functions to specify type of their arguments. Unfortunately it neither provides any means of runtime code introspection determine types of its arguments by use of language constructs. The only possible solution is to use a library function, however such approach is clumsy and lowers language performance.

Declaration of local functions is inconsistent with library functions, which declare formal parameter types as well as their return type. Additionally library functions may accept arbitrary number of arguments as opposed to user defined functions. Since library functions are implemented in Java, they can determine types of their actual parameters - and many do so to handle values correctly. Such implementation however naturally lowers performance and complicates code due to unnecessary runtime type checking.

Both problems above could be addressed by function overloading, however it is not possible in CTL.

4.4.2 Amendment

User-defined functions as well as standard library functions may be overloaded by sharing the same name but declaring different number or types of their parameters. A compile-time algorithm proposed by Doering and Pericas [7] is used to determine which variant of the overloaded function will be invoked in runtime. The algorithm is identical to Java behavior, disregarding function return type, considering only number and types of actual arguments to find the most specific variant. Example of overloaded user function is shown in Listing 11

```
function int sum(int a, int b) {  
    // add integers  
    return a+b;  
}  
  
function string sum(string a, string b) {  
    // concatenate strings  
    return a+b;  
}  
print_err(sum(1,2)); // prints 3  
print_err(sum("Hello","world")); // prints: Helloworld
```

Listing 11. Example of function overloading.

4.5 Name-based access to graph elements

4.5.1 Justification

Accessing metadata, lookup table or a sequence from CTL requires using their internal graph identifier. The identifier is however constructed in a way that guarantees its uniqueness, but lacks any logical meaning of its object.

4.5.2 Amendment

Instead of graph references, the element object names are used to refer to sequences, lookup tables and metadata. Using element names instead of references comes with a price of having to deal with possible duplicates. Although duplicate names are not very common, it generally cannot be prevented, thus if duplicate elements exist a warning is issued for the programmer who can possibly rename conflicting elements or avoid using them.

```

// previously: record (Metadata0) customer;
CustomerRecord customer;

// previously: lookup(LookupTable3,"street").id;
lookup(AddressTable).get("street").id;

// previously: sequence(Sequence2,int).next
sequence(IDGenerator ,int).next()

```

Listing 12. Graph elements may be accessed by names.

4.6 Single assignment operator

4.6.1 Justification

There are two assignment operators in the CTL which only differ in allowed target of their operation. When the left-hand side of an assignment is a global record field the mapping operator `:=` must be used, while assignment to variables uses the `=` operator.

The use of mapping operator is further limited by the following (syntactic) constraints:

1. Mapping statement¹¹ can only appear at the end of CTL code unit or function.
2. After the first mapping statement no other type of statements except `return` statement are permitted.

The first constraint causes all mapping statements to be always located at the end of code unit while their values must have already been computed because no other statements are allowed. It means that programmer (or automatic code analyzer) can quickly determine what values the code produces just by reading from the first occurrence of a mapping statement.

Unfortunately the constraint is too restrictive and may lead to complicated code just to bypass it. Firstly it directly requires using unnecessary local variable just to store the value to assign. Furthermore it prevents use of branching statements such as `switch` or `if`.

In case the branching condition also affects choice of output port as showed in Listing 13 a local function must be declared separately for each individual port to assign values properly.

¹¹Statement with assignment using mapping operator.

```

function map_port0(condition) {
    $0.result := condition;
}

function map_port1(condition) {
    $1.result := condition;
}

function transform() {
    int port_used;
    if (condition) {
        // if condition is true send to port 0
        map_port0(condition);
        port_used = 0;
    } else {
        // otherwise to port 1
        map_port1(condition);
        port_used = 1;
    }

    // indicate to component which port was written
    return port_used;
}

```

Listing 13. Mapping with condition affecting output port choice. Requires use of functions.

4.6.2 Amendment

Use of mapping operator is treated as obsolete and semantically both operator are treated identical. Providing that the mapping operators is used in CTL code a warning is issued for the programmer. The syntactic limitations on mapping assignment operator do not apply anymore.

4.7 Iteration through fields

4.7.1 Justification

There is no possibility in CTL to iteratively access all fields in a record. Thus if the transformation requires applying the same processing algorithm on multiple fields, the programmer must write the code separately for each processed field.

4.7.2 Amendment

A type-safe looping statement `foreach` was introduced to the language. The statement is similar to Java extended `for`-statement and requires variable declaration and expression that is iterated through. The variable is only visible within `foreach` statement and in runtime is consecutively populated with values taken from the expression. Type of the expression must be one of `record` or `list`.

The statement guarantees a type-safe iteration. Providing that the expression is of `list` type, the list element type must be assignable into the declared variable. For `record` only fields with the type identical with the variable are iterated through. The behavior was deliberately chosen because the programmer may need to process only fields of single specific type. In case the assignability was the determining factor, the programmer would not be able to iterate through `double` fields as also `int` fields would be assignable to a `double` variable.

```
int [] integerList = {1,2,3};  
foreach (double d : integerList) {  
    print_err(d); // prints 1..2..3  
}  
  
$0.intField = 10;  
$0.doubleField = 0.12;  
foreach (double d : $0.*) {  
    print_err(d); // prints only 0.12  
}
```

Listing 14. Differences between iterating through list and record.

4.8 Others

Generally the language syntax was further extended to allow constructs typical for other programming languages. An overview of changes follows:

- Variable declarations may contain an initializer expression.
- Usual ternary conditional expression `?:` (originally implemented by function `iif`) was introduced to the language.
- A declaration of variable may occur as `for` statement initializer.
- All parts of a `for` statement - initializer, condition, update - are optional.

- Broken syntax of `switch` and `case` statement with unnecessary blocks.

Many of these were addressed in the parsing phase of our compiler.

5 Compiler Design

This section gives an overview of important decisions made in general compiler design and affected its implementation.

5.1 Implementation language

Java programming language was chosen to implement the compiler. The choice is obvious as CloverETL Engine builds on the Java platform and the compiler is expected to become an integral part of it. Nevertheless this takes no credit from Java as its automated memory management, comfortable string API and plenty of existing libraries make it a perfect candidate for writing any language compiler.

5.2 Supported Java version

The translator generates Java source code compliant with Java 1.5 and Java 1.6 which are at the time of writing the most current releases of Java platform. Generated code will not compile with Java 1.4 due to use of generic types, autoboxing of primitive types and extended for-loop that were only introduced to Java release 1.5. Obviously in order to dynamically compile the generated code a JDK version of Java platform with *javac* compiler must be present in the system.

5.3 On Source to Source Translation

The translation of CTL code to Java was decided to be built on source-to-source basis. Such approach was chosen deliberately for several reasons.

In the first place using *javac* to compile source code into bytecode instead of generating it directly by own facilities has the advantage of retaining complete set of optimizations and validations them without the necessity of reimplementing them in CTL compiler. Further development of the compiler can therefore focus on the front end while the backend will constantly improve with next releases of Java platform.

It is also important to point out that Java has a good support for backwards compatibility of source code, hence the backend should not be a subject of often change often due to new Java releases.

Secondly, generated Java classes are serialized to `.java` source files which

are useful for tracking transformation executions. In case the transformation does not produce expected results or fails, the source file can be used by programmer to investigate cause of wrong behavior. Had the CTL code been translated to bytecode, the programmer would be forced to decompile the bytecode first for post mortem analysis. This is also a motivation behind generating Java code as close as possible to original CTL structure.

Finally the presence of Java source files helps with the development and debugging of the CTL compiler itself.

5.4 Interpreted and compiled execution

As explained in Section 3.6, transformation component defines a Java interface with methods that the component calls regularly in order to perform record processing and uses a wrapper to execute corresponding functions in embedded CTL code.

We decided to retain the original interpreted execution and add compile execution mode as optional. While the interpreter builds on top of original adapter architecture, the compiled mode takes advantage of component ability to accept dynamically compiled Java code.

Making compiled execution as optional feature allowed splitting the compiler into frontend modules common for both execution modes and an optional Java backed phase, which was moved into separate CloverETL plugin. In case the backend plugin is not present in CloverETL installation the interpreted execution is chosen automatically. Default execution mode for whole CloverETL installation can be also set globally via compiler priority attribute in plugin descriptor file `plugin.xml` of Java translator.

Execution mode can be also forced by CTL programmer. We extended the indication comment concept to be recognized by the compiler (Section 3.3.1), and if it contains string `#TL:COMPILE` the compiled mode of execution is forced (providing that the translator is present in installation).

5.5 Architectural fit

The CTL implementation was designed to fit into architecture used by the existing implementation without necessity of rewriting major parts of it. Compiler as well as runtime support (interpreter, standard library, wrappers) were designed to be modular, oriented on a specific functionality with a good separation of concerns.

Interpreter and frontend modules remain in core of CloverETL Engine library,

so CTL code interpreting is always possible

5.6 Compiler modules

Compiler was built to consist of multiple phases, each implementing a specific compilation step as traditionally known from compilers [15] theory: parsing, semantic analysis, type checking, flow control and optional code translation. The phases are sequentially executed. Next phase does not start if errors were found by the previous one. Such approach allows discovery of all possible errors related to a specific compilation phase but also prevents having necessity of error recovery code scattered throughout phases.

To satisfy our goal of precise error reporting and avoid replicating AST the compiler avoids changes to the structure of AST tree as much as possible. If the modification is inevitable it is performed as late as possible in compilation process.

5.6.1 Syntactic analysis

The parsing phase was designed to be lightweight with focus on syntactic analysis and construction of AST tree, but also covering part of semantic analysis by constructing symbol tables. Some of the syntactic errors that could not be reported properly by the parser were rather delayed until later phases to improve the quality of error messages.

5.6.2 Semantic analysis

The semantic analysis phase partially overlaps with parsing, however most activities are done once parsing is finished. It generally includes validating language constructs where syntactic rules are not sufficient (and semantic rules must be used), resolving of references to graph elements and establishing initial type information in leaf nodes of AST tree.

5.6.3 Type checking

Type checking phase spreads the type information collected by previous phase and spreads it throughout the AST tree. Resolution of overloaded functions is delayed until the type checking phase as well as it requires knowledge of actual argument types in function calls.

5.6.4 Flow control

Flow control phase validates correct use of `break`, `continue` and `return` statements. Depending on determined mode of execution one of the following phases kicks in.

5.6.5 Runtime initializer

In case of code interpreting, the runtime initialization phase is started. The phase is actually part of the interpreter as it may need access to access runtime structures of component executing the code.

5.6.6 Code translation

Opposingly, if a compiled execution mode is expected the translation phase is started. During the translation phase the AST tree of CTL code may be partially modified if it contains `for` looping statements. Finally the actual translation is executed by rewriting the AST tree of CTL code into AST tree of corresponding Java code. Afterwards the constructed AST is serialized into Java source file. Finally the source file is compiled into Java class using bytecode compiler and dynamically loaded. The loaded class instance is the final output of the compiler.

5.7 Representation of values

An important decision was determining how to represent values in both modes of execution. The original implementation was using value containers accepting values various type. Except storing the value, each container had type information attached. Therefore when value was being assigned, the container validated could inspect the type and convert the value as needed. Such approach was necessary for implementation of loose typing of the original CTL.

The value containers were used not only by the interpreter, but also passed to functions in standard library. Therefore the decision of value representation affected both the interpreter as well as the standard library.

Considering the unneeded (and unwanted) implicit type conversions, unclear implementation and possible performance improvement, we decided to relinquish the use of containers and start using classes of Java primitive types. More specifically, in order to be able to represent `null` value and

simplify value handling with `list` and `map` types, the object variants of Java primitive types were chosen.

Representing the values by identical Java classes both in emitted class as well as interpreter proved to be a key decision as much better consistency between two modes was achieved. Actually, the translating backed is basically the interpreter rewritten to Java code generator.

5.8 Standard functions

After having consistent representation of values, another important step was to achieve consistent use and implementation of standard library function.

Desired result of standard library implementation was to have interpreter and emitted class invoking identical implementation of standard library function. If implementing functions as simple methods, the emitted transformation code could directly call method from library class. The interpreter on the other hand requires the opposite: an object representation of function and a uniform access method to perform the invocation.

To keep function calls at high performance we decided that standard functions will be implemented as `static` methods in library classes and compiled code will be calling them directly. We considered use of Java reflective API to create an object representation of library function by means of Java, but refrain from using it because of poor runtime performance. Instead, each function has a corresponding adapter class representation with single entry method, making them uniform from the interpreter point of view.

When interpreter wants to invoke a library function it passes types to the adapter a reference to current stack and an array with types of function actual parameters. Based on the number and types of actual parameters the adapter knows which variant of possibly overloaded function to call. It pops needed number of parameters from the stack, passes them to method implementing the function, then pushes possible return value back onto stack. The Java transformation class calls the method implementing the function. Thus both interpreter as well as compiled class end up calling identical code.

Representing functions as objects brings disadvantage in necessity of implementing the adapters separately for each function. This is however balanced by increased performance especially when compared to use of Java reflective API.

6 Compiler Implementation

As explained in Section 5.5, the compiler is highly modular and uses phased approach to perform the compilation process. From implementation point of view most of the phases take form of tree walkers operating solely on AST tree; no intermediate code representation is used. Storing the code in AST tree may look memory intensive, however CTL code is usually not too extensive thus the memory taken by AST is insignificant when compared to memory consumed by the data transformation.

This section discusses the most important implementation steps for each phase of compilation process.

6.1 Parser

In parser implementation we tried to keep compatibility with the previous version to allow execution or at least parsing of old CTL programs. Incompatibilities between the two version are reported depending on their severity either as syntactic errors or as warnings together with hints how the code should be changed by the programmer to satisfy new requirements.

Rather than writing own parser from scratch the JavaCC [8] parser generator was employed. As the old implementation used JavaCC as well, the original grammar file was taken as a base and gradually modified for updated syntax. Implementation of some tokens and grammar rules was inspired by the grammar file for Java 1.5 distributed with JavaCC.

The parser is implemented by the following classes or modules:

1. `TransformLangParser` is the parser class with adjusted `jj_consume_token` method. Generated by JavaCC from `TransformLangParser.jjt`.
2. `ParserHelper` manages symbols and symbol tables.
3. `ProblemReporter`, a class used throughout all compiler process. Collects errors and warnings.

6.1.1 Tokenizer

The lexical analysis is more or less straightforward however requires handling of two specific cases, where default tokenizes behavior must be redefined. In JavaCC this is achieved by defining additional lexical states. Their overview follows.

- *Default* lexical state is active on parser startup and during normal operation.
- Lexical state *WithinFieldIdentifier* prevents sequence `$2.1` referring to first field of the second port to be recognized as `$`-symbol followed by decimal literal `2.1` and creates a token for field-access instead. The tokenizer switches to this state on occurrence of `$` symbol.
- *WithinComment* prevents tokenizing contents of multi-line comments, passing them as a special token instead. It is activated by `/*` sequence and exited on `*/` sequence occurrence.

6.1.2 Error recovery

Parser error recovery takes advantage of *shallow* as well as *deep* error recovery methods proposed by JavaCC documentation [9]. The skeletal symbols on which the parser recover are generally the separator symbols (typically `;` and `}`). In case of a syntactic error the parser skips input until their next occurrence.

After the initial testing the error reporting was found insufficient for cases where an expected separator symbol was missing. Possible implementation of automatic error correction by inserting tokens with use of Burke-Fisher algorithm [10] or its variant presented in [11] was considered was found unsuitable for JavaCC-based parser.

The unsuitability is generally caused by the fact that JavaCC represents transitions between internal automaton states as Java method calls. When an error occurs in particular state (method invocation), it is not possible for the parser to return to the previous state (i.e. deliberately jump up the stack to the calling method), repair the error and re-enter (i.e. re-invoke the current method) the erroneous state. Detailed explanation can be found in [12].

Despite we were unable to implement satisfactory automatic error repair some improvement in error reporting was possible. The main problem with JavaCC-based parser behavior is that the error messages contain listing of a complete follow-set of tokens for parser state where an error occurred. The result is a cryptic error message with tenths of expected tokens although for many cases (usually missing `;` or `}` symbol) only one symbol could be easily reported. Unfortunately parser generated by JavaCC by default overwrites the information about expected token before invoking user error-handling code. To override this behavior we had to modify the generated parser's `jj_consume_token` method to pass the expected token to error-handling `tokenError` method. The method utilizes the token information to construct

a user-friendly error message, then the parser continues in the default error-recovery activity.

6.2 Resolver

The parsing phase is followed by AST resolving phase performing most of the activities of semantic analysis. The main role of resolver is to check derivations in AST that require semantic knowledge, discover undeclared symbols, bind references of graph elements and establish elementary type information. The resolver is implemented as a single-pass AST visitor in Java class `ASTBuilder`.

Before the phase starts AST processing it examines elements of graph in which the code exists, retrieves lists of metadata, lookup tables and sequences populates its translation tables so that element names can be transformed into graph references. If duplicities are found, the phase issues a general warning for the user.

6.2.1 Semantic checking

In order to keep the CTL grammar clean and simple enough some of grammar rules used by the parser may result in unwanted derivation and require additional semantic checking. This mostly relates to derivation of statement expressions; plus assuring the presence of `foreach` loop variable declarator. Parsing of literals into corresponding values is also deferred until this phase to keep grammar rules unpolluted with error-handling code.

6.2.2 Element references binding

In the resolver pass all graph elements names are firstly converted to unique references, then bound with respective objects. This process includes

- Determining if expression using `$`-operator refers to input or output field based on its relative position in an assignment expression.
- Binding lookup AST nodes with graph lookup tables.
- Binding sequence AST nodes with graph sequences.
- Binding `record`-type AST nodes with referenced metadata descriptors.

The resolver also translates name-based access to record field into corresponding positional access. The positional access is faster and is used by both interpreter as well as in generated Java class.

6.2.3 Type extraction

When external references have been bound to graph elements it is possible to propagate the type information from elements to AST nodes. Metadata descriptors are therefore converted into `record` types and assigned to variable declaration nodes. Lookup nodes are assigned type formal parameters extracted from lookup key types and lookup table return type is set to corresponding `record` as per metadata of records stored in the table.

6.3 Type checker

Type checker builds upon information collected by the resolving phase and propagates it throughout AST tree using type checking rules. Function calls binding is deferred until here as the types of actual parameters of function call must be known prior to resolving overloaded functions.

The type checked is implemented by the class `TypeChecker` which once again is a single-pass visitor over AST tree.

6.3.1 Types representation

Various kinds of CTL types are represented by classes extending from `TlType`. Primitive types are represented by singletons extending class `TlTypePrimitive`. Container types `TlTypeList`, `TlTypeMap` and record type `TlTypeRecord` are then constructed from primitive type classes representing their element types. Type checker additionally uses the following synthetic types:

- `TlTypeObject` is used for legacy built-in functions accepting argument of any type (such as `iif`, `isnull`).
- `TlTypeError` used to indicate type mismatch.
- `TlTypeDateField` represents type of symbolic constants for date fields.
- `TlTypeLogLevel` represents type of symbolic constants for log levels.
- `TlTypeVariable` represents Java type variables extracted from generic methods implementing validation functions in standard library.

Class `TType` defines three methods for elementary operations with the types:

- Method `canAssign` tests assignment-compatibility of two types.
- Method `promoteWith` returns greater of the types under partial type ordering, returns `TTypeError` if no ordering exists for the types in question.
- Static method `distance` calculates distance of two types based on the type distance matrix.

	int	long	double	decimal	string	boolean	date	null	void	date symbol	log symbol
int	0	1	2	3	∞	∞	∞	∞	∞	∞	∞
long	∞	0	1	2	∞	∞	∞	∞	∞	∞	∞
double	∞	∞	0	1	∞	∞	∞	∞	∞	∞	∞
decimal	∞	∞	∞	0	∞	∞	∞	∞	∞	∞	∞
string	∞	∞	∞	∞	0	∞	∞	∞	∞	∞	∞
boolean	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
date	∞	∞	∞	∞	∞	∞	0	∞	∞	∞	∞
null	0	0	0	0	0	0	0	0	0	∞	0
void	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
date symbol	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0
log symbol	∞	∞	∞	∞	∞	∞	0	∞	∞	∞	∞

Figure 1. Type distance matrix.

There are additional rules for calculating distance of types:

- Distance of two `list` types is a distance of their element types.
- Distance of two `map` types is a sum of distances of their key and value types respectively.
- Distance of two `record` types is 0 if their metadata are equal, ∞ otherwise.
- Distance of primitive or container types to a type variable type is 10.

Using the sum to calculate distance for container types follows the idea of incremental construction of container types from primitive types. Comparing `record` types by metadata is necessary to keep consistency with CloverETL generally using nominative typing. The distance value for type variables guarantees that any function not using a type variable will be considered as more specific.

6.3.2 Function resolution

Function resolution is a two step activity. The first step determines if the called function is local - part of embedded CTL code, possibly imported - or external - part of the standard library. This is achieved by searching declared function symbols by name. Providing that a match is found with both the local code as well as the the standard library the local function declaration takes precedence. Such behavior allows programmer to shadow functions from standard library if necessary.

The second step applies only if multiple candidates were found in the first step, meaning an overloaded function is being resolved. Additionally to function overloading the type checker must also take into account functions with variable number of arguments. To determine the most specific function declaration a distance function is used. Distance between function call and a candidate function is given as a sum of distances between actual types of the function call and formal types of matched candidate. The sum is minimized over the set of candidates to obtain the most specific function variant that will be dispatched in runtime.

6.4 Flow control

Flow control verifier is a single-pass visitor checking correct usage of `break`, `return` and `continue` statements. The purpose of this phase is to detect:

1. Misplacing the `break` statement outside loop or `switch` statements.
2. Misplacing the `continue` statement outside loop statements.
3. Misplacing the `return` statement outside function declarations.
4. Unreachable code situations caused by any of the three statements.
5. Assure that functions with return type other than `void` return value.

The flow control phase does not perform checking of definite variable assignment. However this checking is performed by *javac* so to prevent

Java compilation errors we have to guarantee that every variable is definitely assigned. This is achieved in the translation phase by generating a default variable initializer immediately in variable declaration.

Implementation of the flow control phase can be found in the `FlowControl` class.

6.5 Translator

The translation is the last phase of compilation process. It finalizes the compilation by transforming AST tree of CTL code into AST tree of corresponding Java code. Although it may seem as a straightforward process there are several obstacles which the translator must take into account. These are explained in the following sections.

When designing the Java translator an emphasis was put onto generating the Java code in human-readable form. Therefore in implementation the following steps were taken:

1. Translator preserves names of all used variables.
2. The temporary variables do not use cryptic names but are formed just by `tmpVar` prefix followed by counter value.
3. Loop control variables of `for` statements preserve the original variable name.
4. The generated Java code structure is as close as possible to the original CTL code.

Originally it was anticipated to build Java AST directly with classes used by *javac*. Such approach would allow passing the AST directly to the `javac` and eliminate the need for time-consuming lexical and syntactic analysis. Unfortunately the necessary API for constructing Java AST is only available since Java release 1.6. To comply with CloverETL support for Java 1.5 we have to pass the Java code in the form of source file instead.

The translator is implemented by Java class `ASTRewriter`. Rewriting of `for` statements (6.5.5) is performed by its inner class `ASTPostProcessor`. Translator implementation uses the class `Java` to construct Java AST, thereafter `UnparseVisitor` to serialize it into Java source code. `CTLDefaultImport` is automatically included into any class class generated by translator. The class holds constants and simplifies some common tasks needed in the generated Java code. Classes `Java` and `UnparseVisitor` come from Janino project [13], but were significantly modified for our use.

6.5.1 Code unit translation

A CTL code embedded into component is always translated into a single Java class implementing the interface expected by nesting component. In order to determine how the CTL functions are bound to methods of transformation interface Java annotations were employed. The annotation used for marking methods that need to be generated is represented by the class `CTLEntryPoint`. The class provides three pieces of information:

1. **Name** of CTL function that should be translated into annotated method.
2. **Flag**¹² if the annotated method must be present in CTL code or if it is optional.
3. **Names of formal parameters** that should be applied when method is generated.

Obviously the information about function name allows translator to find particular CTL function and generate its body into annotated method. Names of formal parameters are used for binding of formal parameters of generated method with formal parameters of the function used in CTL code.

Translator receives `Class` of expected transformation interface and using Java reflection API, examines method marked by annotations to generate the required methods accordingly. Annotation processing is performed by class `ClassScanner` just prior to code translation.

6.5.2 Types

CTL numeric primitive types as well as the boolean type are translated into object variant of Java primitive types. For example CTL `long` type is translated into Java `Long` type. The remaining types are translated as follows:

- CTL `string` type is translated into Java `String` type.
- CTL `date` type is translated into Java `java.util.Date` type.
- CTL `decimal` type is translated into Java `BigDecimal` type.
- Lists are translated into instances of Java `ArrayList` type with type parameter bound to corresponding element primitive type.

¹²Currently not used. Missing method are automatically generated.

- Maps are translated into instances of Java `HashMap` type with key and value type parameters bound to corresponding primitive types.
- CTL type `record` is translated into instance of `DataRecord` type which is used also by CloverETL to represent data records.

6.5.3 Assignments

Depending on underlying type the assignments are either translated into Java assignment or to a method call setting the value into LHS. Since all Java primitive type wrappers are immutable¹³ classes their values are passed via Java assignments. Assignment of value to a map variable is translated into `put` method call.

Assignment of value to record field results in invocation of `setValue` method on `DataField` class. In case a wildcard copying of fields is used between two records, the assignment is translated into `copyFieldsByPosition` invocation on `DataRecord`. If additionally to wildcard copying the RHS value is a CTL null literal, the assignment is translated into calling `reset` method instead.

For `list`-type variables the assignment cannot use the `set` method, hence this method may cause `ArrayIndexOutOfBoundsException` exception being thrown due to assignment of value to index being out of current list size. To support assignment to any index within the list the call is wrapped into `setElement` method of `CTLDefaultImport` which prior to calling the `set` method possibly resizes the list to prevent the exception.

6.5.4 Expressions

Not all CTL expressions can be rewritten into an equivalent Java expression. A special care must be taken with expressions over `decimal` type as these often translate into additional sequence of statements computing actual expression value. The logical expressions on the other must be implemented with the lazy evaluation in mind. This again results in translation of a logical expression into sequence of statements.

The issues with expression translation can be visualized on a post-increment expression of a `decimal` variable. In Java code the values of CTL `decimal` type are represented by instances of immutable class `BigDecimal`. Incrementing the variable value therefore results in allocation of a new `BigDecimal` instance that must be handled in additional steps. A Java code computing expression value thus consists of the following steps:

¹³Immutable class does not allow modifying its state after it is created.

1. Save the original value of incremented variable to a temporary variable.
2. Add one to the value of target variable and assign the new instance to incremented variable.
3. Return the reference to temporary variable holding the original value as computation result.

To solve the problem with expression translation we used modified algorithm proposed by Majda [14]. The algorithm first examines the AST of an expression to determine if it is to be rewritten into single expression of the target language. Based on the result either a single expression of target language is emitted, or statements computing expression value are emitted while the computed value is stored into temporary variable. The temporary variable is then used in further computation.

Our implementation of the algorithm does not perform the first step of the algorithm thus avoiding one descent through expression subtree. Instead, it directly generates the code for computing expression value and stores it in instance of class `RValueBlock`. The class allows storing both the value of the expression as well as possible statements computing its value. Emitted expression can be retrieved by the `access` method - it either contains the equivalent Java expression or and identifier of a temporary value holding the computation result. Have the expression rewriting required emitting the statements as well, these can be retrieved by the `getStatements` method. Providing that the CTL expression was rewritten into single Java expression the `getStatements` method returns an empty sequence.

After a CTL expression has been translated to Java, the statements are appended to closest nesting block containing the expression. Variable reference or the actual expression are used in further computation.

6.5.5 Looping statements

The `while` and `do` statements are translated directly to their Java equivalents. The CTL `for` statements are transformed into equivalent CTL `while` statements, then rewritten into Java `while` statements.

Additional problem with loops outstands from abovementioned issues with translation of expressions. Loop condition in Java must have a form of `boolean` expression that is repeatedly evaluated with each loop iteration. In case CTL the code emitted from the condition contains statements the computation cannot happen inside Java loop condition. The scenario is show in Listing 15.

```
decimal d = 0;
while (d++ < 10) {
    print_err(d);
}
```

Listing 15. Loop with `decimal` expression in condition.

Hence if a loop expression translates into sequence of statements it is firstly evaluated inside the loop body, then checked by synthetic `if` statement with `break` terminating the loop if the condition is not satisfied.

6.5.6 Branching statements

The `if` statements are translated straightforwardly into Java `if` statements. Depending on the type of its expression the translation of CTL `switch` statement either produces a Java `switch` statement or a sequence of `if` statements. The `switch` variant is used only if the expression is of `integer` type.

6.5.7 Global scope

CTL global scope may contain variable declarations as well as arbitrary statements. Variables declared in global scope must be visible in all nested scopes including local function declarations, therefore they are translated into fields of generated Java class.

Statements from global scope are included into `global` method that is invoked by a component before the transformation is started. The same approach is used for initializers of variables declared in global scopes since initializer expression may emit additional statements and thus cannot be translated into Java variable initializer.

6.5.8 Functions

Local function declarations are translated into `private` method declarations in the generated class. Functions that are part of component interface are generated with `public` modifier instead.

Function calls are translated into corresponding method calls. In case a standard library function is reference, the call is translated into invocations of static method on library class. For example a `concat` function is translated into `StringLib.concat` invocation.

6.5.9 Lookups tables

References to access lookup tables are stored as fields in the target class. Initialization code for each lookup table is executed as part of `global` method. Lookup `get` operations accept value to search for as a data records and specification of fields that server as lookup key. Therefore when performing `get` call a new `DataRecord` is created, populated with values and passed to lookup table.

The operations are translated as follows:

- Operation `get` is translated into `seek` method call on interface `Lookup`.
- Operation `next` is translated into a sequence of calls to methods `hasNext` and `next` both on interface `Lookup`.
- Operation `count` is translated into a sequence of calls to methods `seek` and `getNumFound`, both on interface `Lookup`.
- Operation `init` is translated into `init` method call on interface `LookupTable`, followed by refreshing reference stored in the field; and finally
- Operation `free` is translated into calling `free` method call on interface `LookupTable`.

6.5.10 Sequences

Storing of references and handling of initialization code is done in the same way for sequences as is done for lookup tables.

The individual operations are translated to method calls on interface `Sequence`:

- Operation `next` is based on specified return type translated to one of `nextValueInt`, `nextValueLong` or `nextValueString`.
- Operation `current` is based on specified return type translated to one of `currentValueInt`, `currentValueLong` or `currentValueString`.
- Operation `reset` translates to invocation of `resetValue` method.

7 Standard library

Necessary redesign of standard library is explained in Section 5.8. The changes resulted in rewriting almost the whole standard library and its internal mechanisms from scratch.

7.1 Plugins

The standard library consists of five Java classes, each being CloverETL Engine plugin. Separation of standard library into plugins is identical with the original implementation:

- `ContainerLib` holds functions for manipulation with composite types.
- `ConvertLib` provides functions for type checking and format validation.
- `DateLib` deals with date arithmetics.
- `MathLib` contains common mathematical functions and constants.
- `StringLib` contains string-manipulating functions and additional support for regular expressions.

All classes extend `TFunctionLibrary` which is a common ancestor for all expected standard library plugins. The class also implements scanning and registration functionality described in further sections.

7.2 Function registration

In order to make functions accessible in CTL the library classes in original implementation must have published their functions via configuration file `plugin.xml`. Keeping the file up to date with class implementation was clumsy, therefore the function registration was reimplemented to allow automatic function discovery.

A validation function is always implemented as a `static` method of library class and to make it available to CTL the programmer must mark it with annotation class `TFunctionAnnotation`. Although the annotation carries a description about function purpose it serves mostly as a marker.

When a plugin is activated the ancestor class `TFunctionLibrary` scans all methods declared by library class using Java reflective API and extracts only those marked with annotation. For each such method a descriptor object is

created (`TLFunctionDescriptor`) is created holding method name, number and types of formal parameters as well as return type. All types are converted to CTL types. The CTL compiler uses the descriptor the same way as if it was created for a local function declaration.

The descriptor object also contains information about possible variable number of arguments the function accepts or that the implementation is parameterized. Parameterized implementation of some functions was necessary in order to allow determining function return type from actual parameters. It is used for implementation of almost all functions handling CTL container types.

8 Interpreter

The interpreter is implemented as a simple tree visitor using a stack to store results of computation. After AST is created and validated by frontend it may be passed to interpreter for execution. However before the execution is started, the interpreter needs to build data structures for functions calls and library functions. This is the main purpose of initializer pass.

8.1 Function calls

Section 5.8 explains that interpreter requires presence of an adapter class to be able to call library function. The adapters are descendants of interface `TFunctionPrototype` and to save memory, are only allocated before execution in an initializer pass.

In runtime the interpreter invokes adapter's method `execute`, passing it stack reference and an array of type of actual function parameters. In turn the adapter class pops the values of parameters and based on actual types it determines which variant of function is called.

8.2 Lookup tables

In order to implement lookup table operations correctly it is necessary for all operations accessing the same lookup table to share a single instance of `Lookup` class representing an active lookup 'transaction'. This is achieved by storing all `Lookup` instances within a global table indexed by lookup table identifier. Actually to improve access to `Lookup` instances, they are stored in an array a globalarray and lookup identifier are translated into integer indexes into the array. All of it happens during the initializer pass.

Since `Lookup` instance behaves as an iterator, sharing the `Lookup` instance guarantees that the `next` function call returns value of the last `get` operation, while after `init` call it is sufficient to renew only single `Lookup` instance for all AST nodes.

9 Conclusion

The primary objective of this work was to design and implement a compiler translating a CTL programs into Java. Additionally the work focused on improving CTL syntactic and runtime features.

The goal of implementing the compiler was successfully reached. Implemented compiler is fully integrated with transformation components which allows automatic code compilation before transformation is started. From the point of programmer whole compilation process is transparent and does not require any manual assistance. Also, the new compiler detects much wider variety of errors that would cause runtime errors or require lengthy testing with the previous implementation.

Unfortunately the original idea of building the compiler on top of existing implementation was found unfeasible and the CTL language must have been amended in order to make the compilation possible. Most of the changes were necessary due to need of type checking or inconsistent language behavior, other - mostly of syntactic character - were result of flawed implementation. Despite the changes, our compiler is capable compiling some of the original CTL programs unchanged, or at least issue error report that instructs the programmer how to make the code compilable.

Next to the compiled execution, code interpreting is still possible. Due to changes in the language the interpreter was reimplemented in a way very close to compiled execution. Similarly, the standard library was redesigned so that identical code is used for both types of execution. Mechanism for extending the CTL library was also improved to automate and simplify development of new functions.

During the work on the compiler much time was spent studying the original source code to understand interpreter specific behavior and in many cases also to correcting it. We do not decline that our implementation could be closer to the previous, however once it was clear that major part of the language must be rewritten we focused on making the language clearer and more consistent.

9.1 Known limitations

The most important feature of the original language not supported by our implementation are the `eval` and `eval_exp` functions. Although `eval` functionality is common in other scripting languages author humbly believes that there is no space for them in the world of data transformations, where it must be clear what is processing the data. Additionally the existence of

`eval` function as seen in CTL is unique even among scripting languages as its invocation adds new nodes to AST of the running code!

Implementation of `eval_exp` function is possible though, even in the compiled class, but would require storing all parameters necessary to dynamically call the compiler and initialize interpreter.

We also deliberately skipped two undocumented language constructs - the `try...catch` statement and the `@` operator - as they are never used and their correct behavior could not be determined.

It is also important to notice, that during the work on the compiler, the original CTL language as well as CloverETL were under continuous development, so some of the new features may not be supported.

9.2 Supported validation functions

The CTL implementation provided with this work contains only a subset of more than 70 validation functions distributed with standard release of CloverETL. The listing of available functions together with the name of their plugin module follows:

9.3 Supported components

Currently the components that support compilation of CTL into Java and its correct execution are the following: `Reformat`, `Partition`, `Normalizer` and `Denormalizer`. The code of remaining CTL-enabled components must be updated to correctly call functions of generated Java class.

9.4 Future work

The compiler is expected to become a part of standard CloverETL distribution in a very near future. To achieve production-level quality the first steps in future development will lead to implementation of missing validation functions and features added to CloverETL while the compiler was still under development.

The CTL language integration with CloverETL should improve in upcoming releases, mostly by enabling CTL for more transformation components and allow definition of custom records that could be converted into (or from) graph metadata. Also the language should be enhanced to support type-safe mass manipulation with record fields.

An important improvement planned for CTL is a development of debugger integrated with CloverGUI that would allow runtime inspection of code as well as data.

Future development of the CTL compiler will focus on improvement of error reporting and code optimization. Obviously all usual code optimization techniques can be implemented both in the CTL front and as well as in the Java backend. Even more interesting area of optimization work would be optimizing code performance based on the data knowledge, for example a fact that they are sorted or grouped.

References

- [1] OpenSys, a.s.: CloverETL data integration tools
<http://www.cloveretl.com/>, retrieved on 2009-04-03
- [2] Sun Microsystems, Inc.: Java Platform.
<http://java.sun.com/>, retrieved on 2009-04-03
- [3] World Wide Web Consortium: XML: Extensible Markup Language
<http://www.w3.org/XML/>, retrieved on 2009-04-03
- [4] The Eclipse Foundation: Eclipse Platform
<http://www.eclipse.org/>, retrieved on 2009-04-03
- [5] Kimball, Ralph; Margy Ross (2002): *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (2nd edition)*, Wiley, 2002, 358–362
- [6] OpenSys, a.s.: CloverETL Wiki
<http://wiki.cloveretl.org/>, retrieved on 2009-04-03
- [7] Karl Doerig, Santiago M. Pericas (1998): *Espresso, A Java compiler written in Java*, Work Report, November 1998,
<http://types.bu.edu/Espresso/report/espresso.pdf>, retrieved on 2009-04-03
- [8] Sun Microsystems, Inc.: JavaCC: The Java Parser Generator
<https://javacc.dev.java.net/>, retrieved on 2009-04-03
- [9] JavaCC: Error Reporting and Recovery
<https://javacc.dev.java.net/doc/errorrecovery.html>, retrieved on 2009-04-03
- [10] Michael Burke, Gerald A. Fisher (1987): *A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery*, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, 164-197
- [11] Philippe Charles (1991): *An LR(k) Error Diagnosis and Recovery Method*, Proc. of the 2nd Int. Workshop on Parsing Technologies, Cancun, Mexico, February 1991
- [12] P. van der Spek, N. Plat, C. Pronk (2005): *Syntax Error Repair for a Java-based Parser Generator*, ACM SIGPLAN Notices, Vol. 40, No. 4, April 2005, 47-50
- [13] Janino: An Embedded Java Compiler <http://www.janino.net/>, retrieved on 2009-04-03

- [14] David Majda (2009): *Translating Ruby to PHP*, Master Thesis, Charles University, Prague, 2009, 43-44
- [15] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (1986): *Compilers, principles, techniques, and tools*, Addison Wesley, Reading, Massachusetts, 1986

A CD Contents

The CD attached to this work contains the following directories:

- *src* contains ZIP archive with sources of CloverETL Engine including the compiler. All directories in the ZIP file are Eclipse projects. The compiler implementation consists of the following projects:
 - *cloveretl.engine*: implementation of core compiler modules and interpreter
 - *cloveretl.compiler.commercial*: implementation of Java translator
 - *cloveretl.tlfunction*: classes of standard library
- *dist* contains ZIP archive with distribution package of CloverETL Engine
- *simpleExamples* directory with configured data transformation graphs that can be executed.
- *doc* contains this works in PDF format as well as LaTeX source.

B Installation and usage

In order to run the compiler an installation of JDK 1.5 or higher is required. We also suggest setting environment variable `JAVA_HOME` to directory with JDK installation.

To install CloverETL and start using it please follow these instructions:

1. Unzip contents of distribution package into any disk directory (`$INSTALL_DIR`).
2. Set environment variable `CLOVER_HOME` to point to the `$INSTALL_DIR`.
3. Add `$CLOVER_HOME\bin` to your `PATH` variable.
4. Copy directory *simpleExamples* onto disk (`$EXAMPLES_DIR`).
5. `cd $EXAMPLES_DIR`
6. Start graphs by: `clover.bat graph/graphFile.grf`

Graphs available for execution are:

- `graphDenormalizeTL.grf`
- `graphNormalizeTL.grf`
- `graphOrdersTLReformat.grf`
- `graphPartitionTL.grf`
- `graphSequence.grf`

The *simpleExamples* directory has standard structure for CloverETL data transformations. The following directories are important:

- *data-in*: contains subdirectories with example data files used by graphs.
- *data-out*: after graph execution the output files appear here.
- *graph*: contains XML definitions of transformation graphs.
- *meta*: contains definitions of metadata
- *seq*: contains files created by persistent sequences.
- *trans*: when Java compilation is used will contain generated source code as well as compiled classes.