LORD: Lay-Out Relationship and Domain Definition Language*

András Benczúr, Zsolt Hernáth, and Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics Pázmány Péter sétány 1/C H-1117 Budapest, Hungary abenczur@ludens.elte.hu, hernath@ullman.inf.elte.hu, gsd@elte.hu

Abstract. Autonomous applications have a central role in modern database architectures. Already existing standalone autonomous applications could be woven into complex systems in a highly automated way. However, integration of independent applications originated from different domains raises the question of syntactical considerations when a common host language is used to express intentions from arbitrary domains. In this article we introduce LORD - a domain-independent high-level language. LORD has been designed for natural embedding into arbitrary host languages. Using generative programming techniques, like metamacro definitions and context-dependent macro expansions, LORD is able to encapsulate domain specialities and present a heterogenous syntax for application programmers of integration platforms.

1 Introduction

The notion of applications' autonomy was first introduced in [5], and were motivated by a real industrial need of integrating two legacy CAD application systems and their application domains. The idea behind applications' autonomy concerned there was set around two issues:

- just as relational data model and relational database management systems makes their applications or application systems independent of the physical representations and organizations of their data, application systems or stand-alone applications have to be made independent of a higher level abstraction and representation of their data environment;
- instead of defining unified data views for application areas, using a low-level proper data model, data themselves can carry all non-application specific knowledge about themselves.

In our earlier paper [6] we presented a formal model, of applications' autonomy by introducing abstract domains schema, medium and abstract access environment, and gave formal semantics to application autonomy via defining abstract access method as an instance of abstract access environment that controlled data access between particular instances of schema and medium. To achieve real

 $^{^{\}star}$ Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002

domain and data access semantics, a GDDM principled bind mechanism, and a language support to generate GDDM representation of data was proposed also in [5]. Recall, that GDDM is a DDM, where all data are generalized, and in addition, there is a G schema interpreter which governs data, always with adequate structuring, through all processing phase including retrieval and storage.

Integration of independent applications originated from different domains raises, however, the question of syntactical considerations. It is obvious, that the integration requires a common domain definition language, which is used to express arbitrary intentions from all the domains to be referred. On the other hand, application programmers, who are strong professionals in their favorite language, have frequently difficulties when they are forced to use some domain-specific language. For instance well-trained, experienced C++ programmers are not necessarily experts e.g. in all the details of the syntax and the semantics of embedded SQL - a typical domain-specific language. It is more convenient for them to present domain-specific information embedded into the host language also not strange against the host language. Suppose, we have an autonomous application with a certain domain (called Domain), and entity (called Entity). In the integrating application we should refer this entity, e.g. we should write and retrieve it. In the Java host language, it is common to use getter and setter functions to access and modify data entities. Such methods have usually trivial semantics and most cases they can be automatically generated.

```
/* Retrieve Entity in Java host environment */
Entity x = Domain.getEntity(xoid);
```

In the COBOL language, the application programmer should declare the corresponding ENVIRONMENT DIVISION and DATA DIVISION entries. Embedded LORD invocations can detect and use this information to generate code.

```
* Retrieve Entity in COBOL host environment
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
ASSIGN IN-FILE TO 'DOMAIN'
ORGANIZATION IS LINE SEQUENTIAL.

* ...
DATA DIVISION.
FILE SECTION.
FD IN-FILE.
O1 ENTITY-DETAILS.
O3 ENT-NAME PIC X(20).
O3 ENT-NUM PIC 9(6).

* ...
READ IN-FILE
```

In C++ one can apply the iostream-like syntax to retrieve data from a certain Domain. Here Entity, as a selector from the domain was defined as a *manipulator* [7], and overloaded extractor operators are used to read into x.

```
// Retrieve Entity in C++ host environment
Domain >> Entity(xoid) >> x
```

Hence application programmers can express domain-related intentions in the syntax of the host language. As domain-specific details are encapsulated, the host code is more maintainable, and in the case of autonomous applications the integration level become independent from the components data structure and mapping details.

This paper introduces LORD – a domain-independent high-level language, as domain definition and bind mechanism. LORD has been designed for natural embedding into arbitrary host languages. Using LORD and its context-sensitive macro replacement facilities, host language application programmers are able to use their familiar syntax to express references to arbitrary domain.

2 Autonomy and Binding Environments

LORD is used to define and describe abstract application domains (including mappings between domains, e.g. abstract access environment in [6]) and to give their real representations via macro definitions in particular applications' host environments. LORD is designed to provide a macro definition, context-dependent macro invocation and expansion mechanism. Utilizing LORD to define and describe application domains and their data carrier means, offer basically two kinds of making applications autonomous.

2.1 Static or Early-bind Autonomy Environment

Using static or early-bind autonomy presumes that LORD's bind definitions are macro definitions that expand pure host language text. Such bind environment the source program containing LORD embeddings looks autonomous, but the generated host language programs contain real embedded SQL or call interface e.g. in case of using relational databases, i.e. the generated target executive is statically bound to a particular data environment. Porting the application to another environment or reorganizing application data environment may need reimplement LORD domain definition of the program. An early-bind environment components and architecture is shown by figure 1.

2.2 Dynamic or Late-bind Autonomy Environment

Using a late-bind environment the host language program has to link to an LDLL (LORD DLL) passing over access demands with specifying the abstract domains

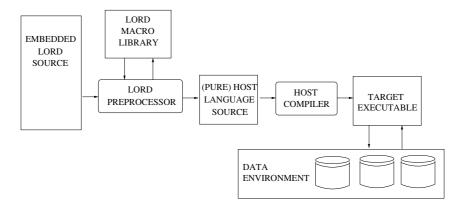


Fig. 1. Static or early-bind environment.

of the application, using document (DDM) or rather generalized document data model (GDDM) [5,6]. Using GDDM, knowledge about data processing is carried by data themselves, and the knowledge is interpreted by the G schema interpreter. To get and interpret such knowledge, data instances are represented by their schema components rather than as ordered pairs, and in a GDDM principled data environment autonomous applications control data access via knowledge carrier, labelled as Domains and Mappinstags Repository on figure 2.

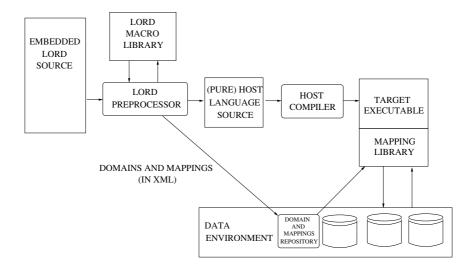


Fig. 2. Dynamic or late-bind environment.

3 A LORD overview

Without detailing the full syntax of the language we focus on, and try to give a picture about LORD's approach of describing abstract domains, and LORD's macro facilities. To describe or to define domains at the present stage LORD provides for only a few constrainted type construction machanism, just as composing sets, sequences, relations, and references to to defined abstract types. LORD presumes a set of not predefined abstract atomic types or sub-domains the composite types can be constructed from. To describe particular host representations of defined abstract types and domains LORD provides an embedded macro language that tries to consolidate the features of traditional macro languages' macro statement and macro function. In traditional macro languages a macro statement typically generates one or more host language statements, while macro functions expand usually a piece of host language text inside a host language statement. In traditional macro languages an invocation of a macro statement typically stands by itself in the embedding text, an invocation of a macro function stands as a part of a host language statement. LORD's macro engine combines this two kinds of invocation forms into one. In LORD embeddings, each statement may obtain an arbitrary number of macro invocations. In LORD, one can easily define a macro that partly expands as statements, and also as functions at the same time. In LORD, just as in traditional macro languages, any host language statement that embeds macro invocations may generate single or more pure or embedding host language statements, depending only on the corresponding macro definitions. The recursive process of such text generations is called the expansion of the embedding statement, and controlled by LORD's expansion method – called CEM^1 .

LORD's macro language is pattern controlled, and LORD's macro engine offers context-dependent expansion of macro invocations with generating embedded or pure host language statements, and even LORD's macro definitions as well. LORD's approach, instead of newly defined abstract operations on data, rather types and instances of types i.e. data themselves appear as macro invocations. LORD macro definitions may obtain complete, and also incomplete (embedding or pure host language) statements; the latter can be composed by applying completion indicators. Expanding a macro invocation, complete statements are expanded in the usual way: after parameters' value setting, the invocation is substituted by the corresponding macro definition. Expanding a complete embedding statement, all of its invocations, in the order of invocations from left to right, are recursively expanded. The left to right rule is necessary, since LORD's macro invocations have free syntax, more exactly, the syntax of a particular macro call within a particular embedding context is defined by the macro definition itself. Incomplete statements shall, however, not be expanded in the usual way. They have to be completed first and the completion is then to be expanded in the usual way.

¹ Complementary Expansion Method

3.1 Domain definition and representation

Domain definition and representation in LORD serves the purpose of host language-independent description of arbitrary data and their mappings. LORD macros are used for mapping domain definitions into certain host languages, creating type definitions and auxiliary methods.

Concerning formal autonomy, a LORD description of applications' data environments typically obtain three abstract domains: schema definition, medium definition and schema-medium mapping. To describe either schemata or media layouts a finite set of atomic domains as elementary types is given. The mapped representation of atomic domains are very much system dependent.

A set is a non-ordered collection of values from a certain type called base type. A set can hold zero or more values. No duplicated value occurs in a set.

```
set work_days
{
     days d;
};
```

A sequence is an ordered collection of a certain component type. A sequence may be empty. The main difference between a set and a sequence is that a sequence allows repetition of the same value, and the order of the values is important.

```
sequence missing
{
    employee missing;
};
```

A tuple type is a model of Cartesian product of component types. The order of component types is important. Two entities are equivalent if the types of their components (in the same order) are the same.

```
tuple employee
{
    string name;
    int id;
};
```

Inheritance is a special kind of tuple construction. There is no semantic meaning of inheritance defined, especially not the one like in object-oriented languages. The only valid use of inheritance is code factoring: i.e. this is the means to avoid code duplication at tuple definitions.

```
tuple manager : employee
{
    string role;
};
```

There are *modifiers* in type constructions.

A ref is a reference to an instance of some type above. The semantic of the reference is not defined. Representation of a reference could be e.g. a pointer. Referred instances are typically not stored inside the objects on the medium. Any recursion including self-reference is allowed. Opt defines optional attributes of a certain component type in a type composition. The meaning of opt is that the mapped image of this component type may not be stored in medium. Storing or loading optional components may have any mapping-defined semantics. Optional types sometimes refer to computed values.

Mapping is a non-intrusive way to describe relationships between two or more domains, i.e. the relations are defined outside of the domain definitions. In fact, a mapping description is also a domain definition.

```
tuple mapping12
{
    source domain1;
    target domain2;
};
```

LORD is being abstract domain definition language and not a language extender, i.e. developers and programmers apply host language constructions of their own referencing to instances of LORD-defined abstract types and data-exchange, as if those types would have defined only by host language construction. Utilizing LORD bind-mechanism, may extend the host language in a comfortable way.

3.2 Contexts and Patterns

To achieve context-dependent text-generation, LORD provides mechanisms for matching particular contexts of macro invocations – called local invocation context. An embedding statement always defines a particular context for macro invocations embedded by that statement. Considering a macro invocation within a host language program, what may and shall be considered as the left- and the right-hand sided contexts of that invocation? More clearly, how far back and ahead it is possible to match a particular piece of text. For the simplicity, one physical line (i.e. piece of text between two newline characters) is considered by default as one statement, and there are tokens to identify the beginning and the end of an embedding statement. Local invocation contexts are statically defined, and they can therefore be referred and identified by context patterns –

an extended mixture of UNIX- and XML-like regular expressions – inside macro definitions. A context pattern refers to a particular sub-context of a local invocation context. If more then one such sub-contexts exist, the closest one to the macro invocation is considered. A context pattern is given in one of the three forms

```
\(\(pattern\)...\)
\(...\(pattern\)\)
\(\(pattern1\)...\(pattern2\)\)
```

referencing a left-hand sided, a right-hand sided and full invocation context, respectively.

LORD does not define a unified syntax for macro invocations. Instead, it provides invocation patterns – that describes the syntax according to which a macro is intended to and shall be invoked. An invocation pattern declares formal parameters (variables) for a macro invocation, and describes what their actual values look like, and where they can be found within the invocation context for a particular macro invocation. Expanding macro invocations, formal parameters' actual values are taken from the invocation context, described by the matched invocation pattern. The only rule of assigning a piece of text to a formal parameter is that formal parameters cannot share the same piece of text as values of their own.

There is another context notion in LORD embeddings, called conjunctive invocation context. Conjunctive invocation contexts are not necessarily static, but always computable. They can be computed from some part of a local invocation context, invocation pattern and neighbouring macro invocation, if any. Conjunctive invocation contexts play an important role in the completion process i.e. when an incomplete statement within a macro definition is to be made complete. The completion process – with the exception of the leftmost macro invocation – needs always computing the right-hand sided conjunctive invocation contexts. The computation of a right-hand sided invocation context is a recursive process that ends if in a computation step the resulted character string does not obtain a trailing completion indicator. The computation of the left-hand sided conjunctive invocation context takes not recursively place only for the leftmost macro invocation, and once computed, the result remains static.

3.3 Variables, Indicators and Directives

LORD is a declarative language, it does not support variable declarations, their value settings, or other operations on them in general, but supports declarations of patterns and macro definitions' formal parameters.

A declared variable is either a predefined pattern, called *term*, or a formal parameter of a macro definition. Formal parameter variables can hold single structured

or unstructured values – such variables are called atoms –, and lists of values, where any value is of the same structure or unstructured – such variables are called aggregates. Both structuring and aggregating can be declared by applying patterns. The following example shows an invocation pattern with predefined patterns:

```
#term s\(\w\)
#term 1\([a-zA-Z].[a-zA-Z0-9]*\)
#term 2\([a-zA-Z]+\)
#term 3\(01*,0s*01\)
[[:
entity @s+ name\(@1\) @s*
{@s*
body[]\(members[]\(\(@1@s*,\)*@s*@1\)@s*:@s*
type\(@2\)@s*;@s*\)+ @s*
};
:]]
```

The pattern above describes an invocation syntax of the macro named *entity* and defines an unstructured variable atom named *name*, a structured aggregate named *body* with its members – an unstructured aggregate named *members* and an unstructured atom named *type*. Elements of *body* are separated by semicolon and the very last element is terminated by semicolon. The aggregate *member* stands for a comma-separated unstructured value list, where the last element of that list is terminated by a colon. A macro invocation that matches the above pattern is

```
{{:
entity intbool;
a, b, c, d : integer;
e, f : boolean;
end_entity;
:}}
```

References to variables cause that variable references are substituted with their values. Any variable may be referred as an unstructured atom. Such references independently of the declaration pattern are to be substituted with an unstructured value including the structuring separator or terminator symbols and white spaces between (pieces of texts). Considering the above example, variable body can be referenced as %body and the substituted text is

```
a, b, c, d : integer;
e, f : boolean;
```

Aggregate variables may be referenced as vectors; in the case of a structured variable, members of the structure can also be referenced. The following examples show some variable references and also the substituted values in the case of

the variable body declared above:

With invocation patterns optional parameters or optional members of structuring can also be described. Parameters declared as optional are to hold optionally matched values. If no value is specified, optional parameters are valueless; a reference to an optional parameter is, however, always valid. If the parameter is valueless, a reference to it as atom represents the empty string. A reference to a valueless aggregate expands to the empty string.

LORD has a number of indicators. Indicators are similar to programming languages' operators and control structures, such as while loop or switch in e.g. C. Indicators are interpreted during macro expansion. There are indicators that are interpreted differently within different context: e.g. variable references body[*] and body[@], refer to the value list of aggregate variable body as row and coloumn vector, respectively, while body refers the pattern labelled by body. The statement indicator $\{::\}$ that indicates the beginning and the end of invocation contexts of particular macro calls, the scope indicator $\{::\}$, or the link indicator $\{::\}$ are context-independent indicators.

LORD defines join on vectors called linking. Linking two vectors of the same size is the concatenation of the corresponding elements of the vectors. Linking two vectors of different size can be traced back to the above by repeating the last element of the shorter vector. Linking a vector to a single piece of text can be considered as linking a vector to a vector of size of one element. The result of linking is a column-vector if one of the operands is a column-vector. Considering the formal parameter declaration in our first example, the following example presents a use of the link indicator:

```
[:< member name='\%body[@].members[@]' type='\%body[@].type' />:].
```

Notice that two column-vectors %body[@].members[@] and %body[@].type with single pieces of text between and around are to be linked, but the member variable members is also referenced as a column-vector. LORD processor interprets the variable reference %body[@].members[@] as nested loops: the external loop runs over the element of body, and the inner loop over the element of members, and since %body[@].type is controlled by the external loop, the expanded text is:

```
< member name='a' type='integer' />
< member name='b' type='integer' />
< member name='c' type='integer' />
< member name='e' type='boolean' />
< member name='f' type='boolean' />
```

LORD provides completion indicator to compose incomplete statements. The completion indicator is three periods following each other. Completion indicator are interpreted only inside macro definitions. Its main role is to indicate that a statement is not complete. One of its simplest use is to implement traditional macro functions. The following example shows the LORD definition of the C macro

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
without detailing the proper invocation pattern.
#macro MAX
{{:
[[:
## here stands the invocation pattern
## defining formal parameters x and y
:]]
\dots(((x) > (y)) ? (x) : (y))\dots
:}
## further invocation patterns, if any
:}}
int a, b, c;
--- some computation ---
a = MAX(b,c) * c;
After LORD preprocessing the following C code is generated:
int a, b, c;
```

--- some computation ---

a = (((b) > (c)) ? (b) : (c)) * c;

exactly the same as if it were generated by the C preprocessor, i.e. the LORD preprocessor interprets a completion indicator that the statement from the corresponding side has to be completed with its conjunctive invocation context of

the side in question. In our case above, using UNIX regular expressions, the conjunctive context of macro invocation MAX(b,c) from the left- and the right-hand side are

A context indicator is similar to the control structure *switch* in C, and is used to match the all-time invocation context of a macro call specifying a set of context patterns. To each specified context pattern a piece of text to be generated for the matched context may be enclosed. A host language indicator is (:' ':), and is used to indicate that the piece of text inside that would otherwise be interpreted as LORD indicator or directive, is a term of the host language. Host language indicators may be used nested.

Directives are instructions to the LORD parser. Directives helps the LORD parser to create work areas, to temporarily extend or build macro libraries, symbol tables and index tables. LORD has few directives, such as #macro, #import, #term, and #onto, to declare or define the beginning of a macro definition, to load a library, to define locally or commonly used patterns, and to specify the file the generated text is to be written to.

3.4 Complementary Expansion Method – CEM

CEM is a natural extension of the usual macro expansion for the case where incomplete statements (statements with leading or trailing completion indicators) occur in macro bodies. Expanding complete embedded statements using CEM means recursively substituting macro invocations with their definitions. Having only complete embedding statements and more than one macro invocation within the embedding statement, the invocations shall be expanded strictly from left to right. To expand an incomplete embedding statement, the statement has to be completed first. Completion of right-hand sided incomplete statements is the substitution of the trailing completion indicator with the right-hand sided conjunctive invocation context of the incomplete statement. CEM recursively maintains a cursor expanding an embedding statement. The cursor administers for each already detected macro invocation the first not-yet-processed embedding statement of the associated macro definition. The cursor is a dynamic vector with non-descendant size: whenever during expansion of an embedding statement a new macro invocation is detected, the cursor is appended with an additional element. At the beginning the cursor is empty. If during the expansion a macro invocation completely expanded, the corresponding element within the cursor is denoted by $M^{\$}$, and is called *inactive*. Cursor elements different from of form $M^{\$}$, are called active. CEM is carried out by the recursive algorithm below:

(i) Starting the expansion of a complete embedding statement s with its leftmost embedded macro invocation M_0 , the statement – after a successful match and interpretation of an invocation pattern – may be considered as of form

$$s: c_0 m_0 c^1,$$

where c_0 is the left-hand sided conjunctive invocation context of all left-hand sided incomplete statements within macro definition M_0 , and c^1 is a not yet parsed pure host language or embedding text. Whenever an invocation pattern of a macro definition is successfully matched and interpreted, all visible variable references, if any, within the matched macro body are substituted with their values defined by the matched invocation pattern. Any value substitution above may result in more than one line.

(ii) Later, during the expansion, due to some completion process of an incomplete statement, statement s may be of form

$$m_0c_1m_1...c_{n-1}m_{n-1}c^n$$
,

where c_i and m_i $(0 \le i \le n-1)$ are pure host language pieces of text and macro invocations, respectively, and c^n is a not yet processed pure host language or embedding text. Suppose, the cursor actual state is $[S_0^{i_0}, S_1^{i_1}, ..., S_{n-1}^{i_{n-1}}]$ i.e. the first not yet processed statements in macro definition M_j is $s_j^{i_j}$ for each $(0 \le j \le n-1)$. In such an intermediate state, the next statement scanned by CEM is $s_0^{i_0}$, and one of the following may happen:

- $s_0^{i_0}$ is complete, or only from the left-hand side incomplete;
- $s_0^{i_0}$ is incomplete from both or, at least, from the right-hand side.

For the first case, the expansion of $s_0^{i_0}$ or $c_0s_0^{i_0}$ (the left-hand sided completion of $s_0^{i_0}$) starts according to (i). For the second case, the right-hand sided conjunctive context of $s_0^{i_0}$ or $c_0s_0^{i_0}$ is to be computed according to (iii).

(iii) If $s_0^{i_0}$ or its left-hand sided completion is incomplete from the right-hand side, suppose, the first not yet processed left-hand sided incomplete statement in M_1 is $s_{l_1}^{M_1}$ – if no such statement is present in M_1 , the completion process fails –, not necessarily different from $s_1^{i_1}$. All statements between $s_1^{i_1}$ (inclusive) and $s_{l_1}^{M_1}$ (exclusive) are expanded according to (i)-(vi). The right-hand sided conjunctive invocation context for $s_0^{i_0}$ or $c_0s_0^{i_0}$ is $c_1s_{l_1}^{M_1}$, if $s_{l_1}^{M_1}$ is complete from the right-hand side. If so, the expansion of $s_0^{i_0}c_1s_{l_1}^{M_1}$ (or $c_0s_0^{i_0}c_1s_{l_1}^{M_1}$) starts according to (i). If not, the completion process goes on with looking for and positioning the first not yet processed left-hand sided incomplete statement in macro definition M_2 , and so on. In general, having a right-hand sided incomplete string as a leading substring of the right-hand sided conjunctive invocation context to be computed, the completion of that

will continue with concatenating the pure host language piece of text that terminates the currently processed macro invocation and finding and concatenating the first not yet processed left-hand sided incomplete statement of the right-hand sided neighboring macro definition, if any.

(iv) If during the expansion of the statement

$$s: c_0 m_0 ... c_m m_m c^{m+1},$$

the cursor's actual state is $[S_0^{k_0},...,M_j^{\$},...,S_m^{k_m}]$, and unless the expansion of s fails, no incomplete statement from the right-hand side, and no incomplete statement from the left-hand side shall occur in macro definitions M_{j-1} and M_{j+1} , respectively.

- (v) If during expansion of s, the sequence of active cursor elements in the order of their ascendant indexes are $S_{i_0}^{j_{i_0}}, S_{i_1}^{j_{i_1}}, ..., S_{i_k}^{j_{i_k}}$, and only complete statements are scanned reading from $S_{i_k}^{j_{i_k}}$ till its end, the next statement to be read is pointed to by $S_{i_1}^{j_{i_1}}$.
- (vi) The expansion of the statement

$$s: c_0 m_0 ... c_r m_r c^{r+1}$$

ends, if the cursor's actual state is $[M_0^{\$},...,M_r^{\$}]$, and no further macro invocation has been detected in c^{r+1} .

4 Limitations

Some of the modern 4GL environments such as *Uniface* [12], *Magic* [13] capture domain description in a higher abstraction level. Their internal domain description is typically not open for third party tools. Unless the 4GL engine gives opportunity to call externally defined functions LORD domain definition can not be utilized.

Another issue is the relationship with other preprocessor languages, like the standard C/C++ preprocessor. LORD parser invocations preced the C/C++ preprocessor phase, which makes LORD control over standard preprocessor directives, like #include and #define. However, this narrows the LORD invocation context to a single compilation unit, since #includes are effective after LORD parsing.

LORD's macro language is pattern controlled, and LORD's macro engine offers context-dependent expansion of macro invocations with generating embedded or pure host language statements, and even LORD's macro definitions as well. What is a useful behaviour in valid execution context, might cause a serious problem when the system needs to be debugged.

5 Related works

There have been many other approaches to make programming languages either syntactically extendible or integrating domain specific concepts into the host language.

Extendible Syntax [9] was introduced for incremental syntax definition by extending the core language. Syntax extensions were placed into the host language between special syntactical delimiters. The implementation is based on LL parsing. Since the class of LL languages is not closed under union and concatenation, the syntax definition sometime uncomfortable.

Syntax macros [10] define syntactical abstractions over code fragments. Syntax macros usually operate on abstract syntax tree representations as a rewriting transformation. The macro accepting abstract syntax tree(s) as argument(s) and producing another abstract syntax tree replacing the macro invocation.

The Java Syntactic Extender (JSE) [11] is a macro system. The expressiveness of the syntax that can be introduced is limited. As in most macro systems, a macro identifier is required in invocations and the JSE parser is not extensible. Opposite to the above, LORD does not restricts the syntax of macro names.

Bravenboer and Visser gives a detailed discussion in [8] on syntactical extension of host languages respect to domain specific extensions, advertising the METABORG method. The METABORG is a method providing concrete syntax mainly for domain abstractions to application programmers. In contrast, our goal, concerning LORD, is not to extent the host language rather than integrate and assimilate domain-specific concerns that otherwise could be formulated in the language. This way our approach is more make map a certain domain into many host languages.

6 Conclusions

Application integration needs to design and implement a common data model and the corresponding data access layer. Information and software industry hoped finding real solutions by establishing data standards for particular groups of application fields, like IFC, CIS/2 [1–3]. Such standards mostly concern data exchange format and are highly domain specific. Another approach is e.g. a data warehouse, that implements a common data access layer over heterogeneously structured data sources for applications.

Integration of applications of possibly different domains and/or following different standards is not generally solved and still costs a large amount of human and computer resources.

Autonomous applications using *Document Data Model* can easily follow structural differences of data, and in addition, structural information of data is carried by themselves. This integration model is also capable of express collaboration protocols between applications processing data of possibly different domains or using different standards.

References

- 1. Industrial automation systems and integration Product data representation and exchange Part 11: Description methods: The EXPRESS language reference manual Reference Number ISO 10303-11:1994, ISO Switzerland 1994.
- 2. Industrial automation systems and integration Product data representation and exchange Part 22: Implementation methods: Standard Data Access Interface specification, Reference Number ISO/DIS 10303-22. ISO, Switzerland 1993.
- 3. CIMsteel Integration Standards Release 2 (Second Edition) http://www.cis2.org/
- 4. Extensible Markup Language (XML) 1.0 (Third Edition) http://www.w3.org/TR/2003/PER-xml-20031030
- 5. Zsolt Hernath, Zoltan Vinceller: Generalized Document Data Model for Integrating Autonomous Applications In Proceedings of International Conference of Applied Informatics (ICAI'6), Eger, Hungary, January 2003.
- András Benczúr, Zsolt Hernáth, Zoltán Porkoláb: Autonomous Applications Towards a Better Data Integration Model ADBIS 2005 Tallinn, 2005.
- 7. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
- 8. Martin Bravenboer, Elco Visser: Concrete Syntax for Abstract Objects: Domain-Specific Language Embedding In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), pp.365-383. Vancouver, Canada, October 2004.
- 9. Luca Cardelli, Florian Matthes, and Martín Abadi: Extensible Syntax with Lexical Scoping SRS Research Report 121, DEC Systems Research Center, February 1994.
- Daniel Weise, Roger Crew: Programmable Syntax Macros In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93), pp.156-165. Albuquerque, New Mexico, June 1993.
- 11. Jonathan Bachrach, Keith Playford: The Java Syntactic Extender. In Proceedings of Object-Oriented Programming, Languages, Systems, and Applications (OOP-SLA'01), pp.31-42. Tampa, Florida. October 2001.
- 12. The Uniface homepage: http://www.compuware.com/products/uniface/
- 13. The Magic homepage: http://www.magicsoftware.com/