Using Object-Oriented Attribute Grammars as ODB System Generator

T. Hagiwara[1] & K. Gondow[2] & T. Imaizumi[3] & T. Katayama[4]

1: Dept. of Information Engineering, Niigata University,
8050 Ikarashi 2-nocho, Niigata 950-2181, Japan
hagiwara@ie.niigata-u.ac.jp
2: Japan Advanced Institute of Science and Technology, Ishikawa, Japan
gondow@jaist.ac.jp
3: Dept. of Information and Image Sciences, Chiba University, Chiba, Japan
imaizumi@tj.chiba-u.ac.jp
4: Japan Advanced Institute of Science and Technology, Ishikawa, Japan
katayama@jaist.ac.jp

Abstract

This paper presents MAGE2 system. It implements a computational model OOAG (Object-Oriented Attribute Grammars) and creates its attributed object trees in object-oriented database (ODB) using persistent object allocation mechanism of object-oriented database management systems (ODBMS). The MAGE2 is a programming support and execution environment for OOAG. The focus of this paper is on an execution system. We indicate core techniques to implement MAGE2, that is, how to execute specifications of OOAG and how to generate an ODB system.

We are planning to use MAGE2 to design databases for storing data that have logical structures such as program source files, XML documents and so on. That is to say MAGE2 is a tool for generating ODB system.

1. Introduction

An object-oriented database management system (ODBMS) has many interesting features such a persistent object allocation, seamless interface to object-oriented programming language and so on. But it seems only few applications exist. We suppose this reason is because it is difficult to use ODBMS in full functional programming language interface such as C++ and JAVA. Designing an object-oriented database (ODB) using the programming language interface of ODBMS includes a complicated work similar to usual programming of data structures. This also makes it difficult to debug, because their data structures often have many pointer links.

Our idea is to make attributed trees of attribute grammars [1] (AGs) persistent using an ODBMS. In standard AGs model, operation for replacing or exchanging subtrees has done only by cut&pasting subtrees from external environment. For example, the Synthesizer Generator [2], that is structured editor generator and has an incremental attribute evaluator, modifies target subtree by editing text. So in the case of using AGs in an database maintenance, it may be impractical to make attributed trees persistent as it is, because there are hard problems to deal with updating objects (i.e. subtrees) in the specification.

In a computational model OOAG [3, 4] which is an extension of standard AGs, we can describe dynamic tree operations by putting message passing rules between attributed tree nodes in the specification. We developed the MAGE2 system as an implementation of OOAG. It has a specification language OSL[5], a code translator from OSL to C++ classes and some programming support environment tools.

Basic functions of MAGE2 are storing attributed trees in the database as persistent objects by following mechanisms:

- Creating persistent object trees using ODBMS.
- Transferring the state of an object tree by message passing in OOAG model. (replacing subtrees, evaluating temporary attributes, etc.)
- Evaluating attributes incrementally if it is needed.

It will execute an incremental attribute evaluator and will restore the state of persistent object trees consistently when subtree will be replaced by message passing. (Messages will be sent if the conditions put in the specification hold). In this way, the MAGE2 (i.e. OOAG) is special attribute grammars system in the sense that are treated as data itself like higher order AGs.

Tools constructed with MAGE2 can act as an ODB generator. MAGE2 generates an ODB generator from OSL specifications like AGs. It is useful for designing complex ODB like AGs are useful for compiler construction. One of major application of MAGE2 is a software repository development. In such use, we describe relations between definitions and references of symbol names in program files in AGs for example. Parsed trees and attribute values are stored in the database persistently and can be used in future accesses.

In this paper, we explain a computational model OOAG and its specification language OSL in section 2. Section 3 describes the MAGE2 system including how to implement OOAG persistently. Section 4 indicates an example of application of MAGE2. Section 5 evaluates MAGE2 as an ODB generator generator in comparison with conventional database developments. Section 6 and 7 are conclusion and future works.

2. The OOAG and the OSL language

2.1. Features of OOAG

OOAG has been derived from attribute grammars. Declarative structures, separation of semantics and syntax definition, and local description resulting in high readability and high maintainability, and clear description due to functional computation of attributes are all desirable characteristics of AGs. We summarize the OOAG features as a generator of database systems as follows:

- OOAG is based on attribute grammars.
- We can program how to manipulate software objects by message passing.
- We can describe data structure and manipulation method of software objects at the same place.
- We can generate software repository system automatically from formal repository specification written in OSL language. OSL language is explained in 2.2.

2.2. The OSL language

An OOAG description is separated into two parts; one is *static specification* and the other is *dynamic specification*. They are described in a specification language OSL (Object Specification Language). We describe briefly each part below, and then give the correspondence of OSL language constructs to conventional attribute grammars constructs.

2.2.1. Static specification

The static specification describes static relation of the object that is described.

Class

A class declaration has following form:

$$\textbf{class } X_0 \to R[X_1, \ldots, X_m] \quad \{\ldots\}$$

where X_0 is the object to be defined, and $X_1, \ldots, X_m$ are its internal objects. R is a label for the rule and is used as a constructor of the object being described. X_0 is referred to as *LHS class*, and R is referred to as *RHS class*. $\{\ldots\}$ defines *static semantic rules.*

Static attributes

Declarations of static attributes are attached to X_i $(0 \le i \le n)$ in the class declaration, whose form is

$$X_i(i_1, \ldots, i_p | s_1, \ldots, s_q)$$

where $i_1, \ldots, i_p$ are *static inherited attributes*, and $s_1, \ldots, s_q$ are *static synthesized attributes.*

Whereas static inherited and synthesized attributes are associated with object X_i, *static local attributes* are associated with classes. A declaration of static local attribute l has the form

$$\textbf{local} \quad l$$

and is written with static semantic rules.

Static inherited attributes and static synthesized attributes are the external interfaces of an object. Native attributes hold the object state. We also use the term *native attributes* for internal objects X_i, because internal objects also hold the object state. Native attributes are similar to instance variables in the object-oriented paradigm.

Static semantic rules

Static semantic rules define the values of the static synthesized attributes of an object, the values of static inherited attributes of the object's internal objects and the values of static local attributes associated with a class if necessary. The form of each static semantic rule is

$$a = f(a_1, \ldots, a_r)$$

where a is an attribute to be defined by $a_1, \ldots, a_r$, which are others' attributes in the description; Finally, f is a function of $a_1, \ldots, a_r$. In the sequel, a notation $X.i$ represents an occurrence static attribute i of an object X and a notation n represents a native attribute n or a static local attribute n. Of course, the dependency graph of static attributes over any tree must be acyclic to be computable. The static attributes can be bound with tree structures as their values. This increases the power of OOAG compared to standard attribute grammar and is found very useful in database description.

2.2.2. Dynamic specification

A set of dynamic specifications that describe message passing defines the dynamic behavior of objects. The dynamic specification consists of the following:

Messages

A message passing description has the form like following:

$$in_1, \ldots, in_s \Rightarrow out_1, \ldots, out_t \quad \{\ldots\} \qquad (s, t \geq 0)$$

where in_i $(1 \leq i \leq s)$ is an *input message* and out_i $(1 \leq j \leq t)$ is an *output message*. $\{\ldots\}$ is the definition of *dynamic semantic rules*.

Dynamic attributes

Each message may have some dynamic attributes. All messages take the following form

$$Obj : mesg\_name(i_1, \ldots, i_p | s_1, \ldots, s_q)$$

where Obj is an object, $mesg\_name$ is a message name, $i_1, \ldots, i_p$ are *dynamic inherited attributes*, and $s_1, \ldots, s_q$ are *dynamic synthesized attributes*.

Dynamic semantic rules

Each dynamic semantic rule is of the form $a = f(a_1, \ldots, a_r)$, which has the same form as that of the static semantic rule, with the exception that values of native attributes can be defined and dynamic attributes can appear in $a_1, \ldots, a_r$. The form to define values of native attributes is

$$(\mathbf{new}\ a) = f(a_1, \ldots, a_r)$$

where **new** is a reserved keyword, which distinguishes new next values of native attributes from old ones. Values of native attributes are replaced by new values after evaluation of dynamic attributes is completed. This form permits the tree to be expanded as a result of dynamic attribute computation.

Condition of message passing

A message passing description is a pair of input messages and output messages. Output messages will be sent to target object in the following conditions.

- All input messages arrived at this object, moreover all values of output attributes of an output message have been evaluated. or
- There is no input message.

However, a set of output messages may have a guard expression that decides condition of message passing. Guard expression controls whether output messages will be sent. In other words, output messages will not be sent in a case guard expression is false.

2.3. Evaluation Loop

Static specifications and dynamic specifications are evaluated by turns, because descriptions in dynamic specification may replace subtrees and it may cause inconsistency of static attributes value. If static attribute value will be change, additional message passings may be arised. So computation in OOAG loops until object tree becomes stable state. Figure 1 shows this image.

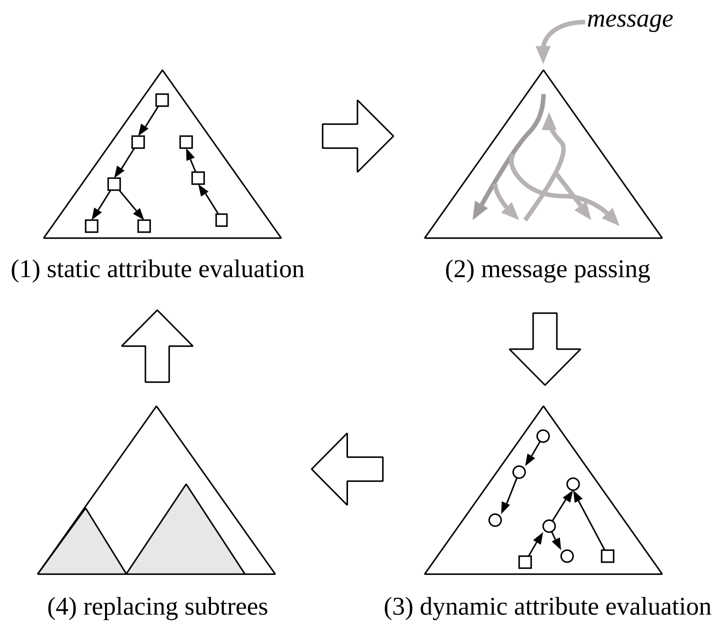


Figure 1: evaluation loop in OOAG

2.4. The correspondence of OSL components with standard AGs.

In this section, we consider the correspondence of OSL components to standard attribute grammars and show that an OOAG is more suited for database systems than standard AGs.

The class definition of OOAG corresponds to a production rule of standard AGs, and static specifications is equivalent to semantic rules of standard AGs. Table 1 shows the correspondence of the terms used in OOAG static specifications and standard AGs.

in Standard AG		**in OOAG**
Production instance	$\Rightarrow$	Object (NODE object)
Terminal symbol	$\Rightarrow$	Leaf Native attribute
Non-terminal symbol	$\Rightarrow$	LHS class name
Label of production rule	$\Rightarrow$	RHS class name

Table 1: The correspondence of the term in the static specifications

Dynamic specification is an extension from standard attribute grammars. This extension makes it possible to change the structure of already built attributed tree and re-evaluate attribute values that is essential in describing repository system. A message passing mechanism may be considered as a function call on the structure of attributed trees.

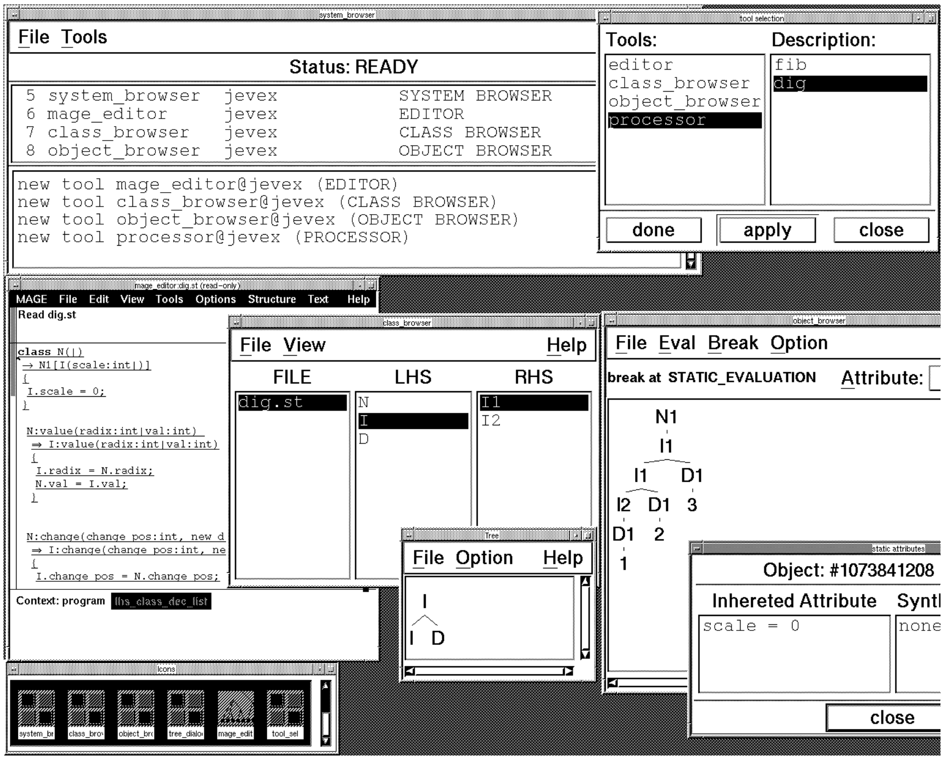


Figure 2: Screen snapshot of MAGE2 system execution.

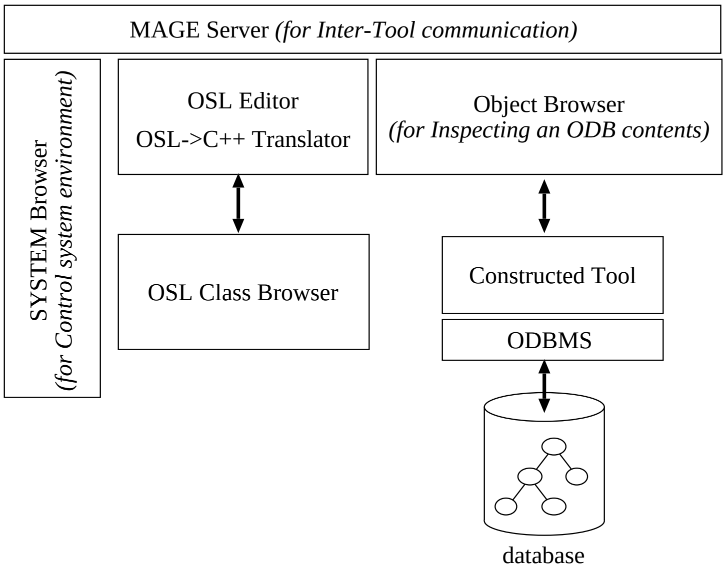


Figure 3: MAGE2 System Architecture

3. Overview of MAGE2

3.1. MAGE2 Environment

MAGE2 code translator translates OSL specifications to a set of C++ classes. We can obtain an execution system image by compiling them with OOAG attribute evaluator library and with ODBMS library. The generated system creates object trees in an ODB. In other words, MAGE2 is an ODB generator construction tool.

To make object trees persistent, we have used an ODBMS. We are now using ObjectStore[6, 7] whose virtual memory mapping architecture (VMMA) achieves the high performance by using sophisticated memory mapping, caching, and clustering techniques to optimize data access. It does not have bad influence on our implementation technique for efficient execution of OOAG, for VMMA can handle persistent data as fast as transient data, and migrating existing system with ObjectStore is easy.

MAGE2 system is an OSL specification development environment which consists of some tools. Figure 2 is a screen snapshot of MAGE2 and figure 3 shows an architecture of MAGE2 tools. Main tools of MAGE2 environment are as follows:

- Several browser tools for controlling and monitoring system states
- A syntax-directed editor for OSL and its graphical class browser
- Code translation system from OSL specification to C++ classes
- An attribute evaluator library of the OOAG
- An object-oriented database management system (ObjectStore)

We mention the amount of MAGE2 system code. MAGE2 attribute evaluator library is about 6,200 lines C++ program. The result of translating short OSL descriptions for evaluator test are table 2. In this table, dig.ooag is a program of digit sequence representation and interpretation, and fib.ooag is a program which compute Fibonacci number. OSL editor and translator is about 11,000 lines SSL program (SSL is the editor specification language of the Synthesizer Generator). Browsers and tool communication library codes total 21,000 lines C program.

	OSL spec.	translated C++ code
dig.ooag	3 LHS, 4 RHS class	
	76 line	→ 800 line
fib.ooag	1 LHS, 3 RHS class	
	30 line	→ 500 line

Table 2: translating result

3.2. Creating OSL specifications and Databases

Figure 4 shows how to use MAGE2. To construct an ODB system, we begin with describing database structure and data management rules using OSL editor and OSL class browser in figure 3. This OSL specification is translated to C++ classes by OSL code translation system, and compiled with an attribute evaluation library and an ODBMS library. A constructed tool needs an initial object tree description. It will be passed in tool execution time by a tool user.

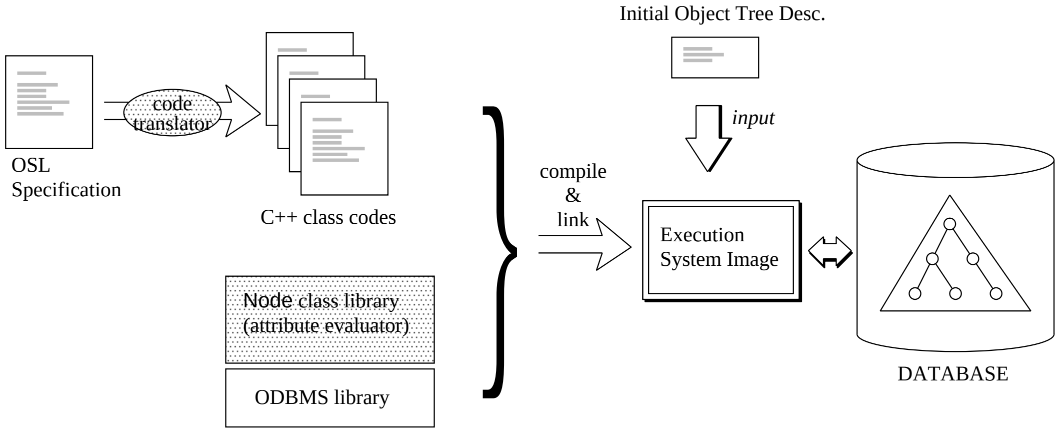


Figure 4: Creating OSL specification and Databases

3.3. Design of Persistent Object Trees

Object trees in MAGE2 must be created in a persistent storage, that is, in a database.

Our approach of object trees design are following rules:

- The relations between LHS and RHS classes is kept in the OSL class specification.
- All tree node objects are instantiated from RHS classes translated from OSL specifications. We call these tree node objects "*Node objects*".
- All translated classes inherit an abstract class Node. The class Node implement an attribute evaluator and object tree maintenance routines.
- No global information should be used. All resources about an tree node are stored together in the Node object, that is, in the Node class and RHS classes. LCDIA algorithm have been introduced in [3] for evaluating attribute values from only local informations.
- Object Allocation is done using STL container. STL is the ISO standard C++ library.

3.3.1. Translation Rules

We translate OSL specifications to C++ classes keeping its LHS-RHS class relations. Following codes will be generated in each RHS class, however most of implementation codes stay in Node class. Node class codes are pre-compiled as a system library.

- Local data structures an array of Attribute and their initialization code.
- Functions for attribute equations in semantic rules.
- Functions for message passing (passing condition and passing rules).

3.3.2. Allocating and Deallocating of Node objects

It is very important how to allocate objects in MAGE2, because we intend MAGE2 to generate object-oriented databases. Basically, we use only two classes to design attributed object trees; class Node and class Attribute. Node object encapsulates all informations about an attributed tree node and Attribute object encapsulates informations about an attribute instance. These two classes are closely related with each other; A Node object has Attributes and an Attribute object may points Node object because of its higher orderness. Figure 6 shows a relation of Node and Attribute object.

Because Node encapsulates all informations includes Attribute objects, all works about object deleting complete only deallocating Node object. It will deallocate all related objects to the node together.

class Node Class Node and its derived classes, that is, LHS and RHS classes have mainly following information:

- Array of Attribute objects. Class Attribute has informations about attribute instances.
- A dependency graph of local attribute instances.
- Methods which implement an attribute evaluation algorithm.
- Fields about intermediate information of attribute evaluation.

The array size of Attribute objects depends on the OSL specification. We dislike to this object size dynamically changes, so we create derived classes from Node by OSL code translator. Node is just an abstract class.

class Attribute Class Attribute has informations about attribute instances, only its actual attribute value is stored in different space. Attribute has its pointer value in `value` field.

Figure 5 shows important fields of Attribute class; `name` is a name of this attribute; `type` is a type information of this value; `my_node` is a pointer to an owner of this instance. There is explanation about `value2` field in 3.3.3.

```
class Attribute
{
  string        name;
  ATTR_TYPE     type;
  Node*         my_node;
  BaseType*     value;
  Attribute*    value2;
  int           (*eval)(Node *);
  ...
};
```

Figure 5: class Attribute

3.3.3. Managing Local Attribute Graph

The current attribute evaluator can compute non-circular AGs. The LCDIA algorithm for OOAG computation is published in [3]. It needs to construct some attribute dependency graphs dynamically in execution time. We show here how to construct a dependency graph for Attribute objects actually in the database.

Attribute informations of each tree nodes are held by a static array in the Node object. Figure 6 shows this image. The dependency graph is constructed only by local informations in the Node

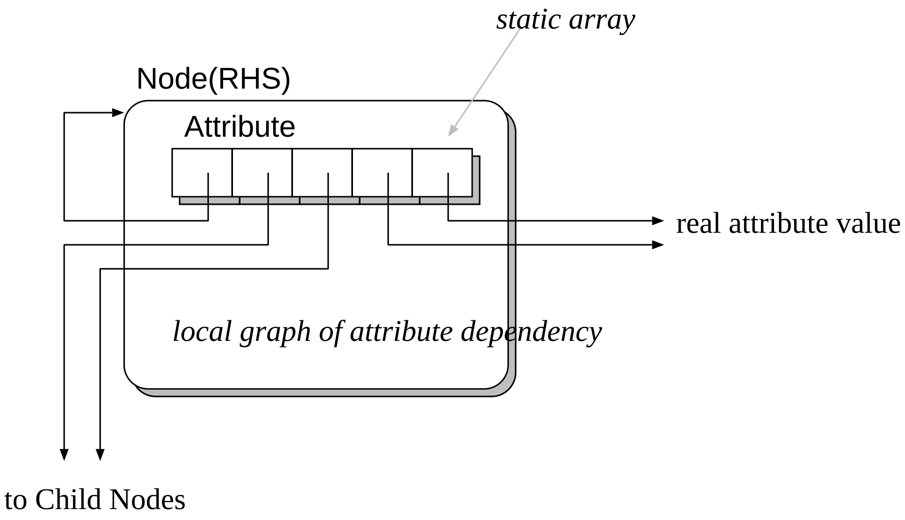


Figure 6: Relation between Node and Attribute object

object, that is, by index values of an array of Attribute object. So there are two Attribute objects that represent the same attribute instance; one is in the parent Node object and the other is in its child one.

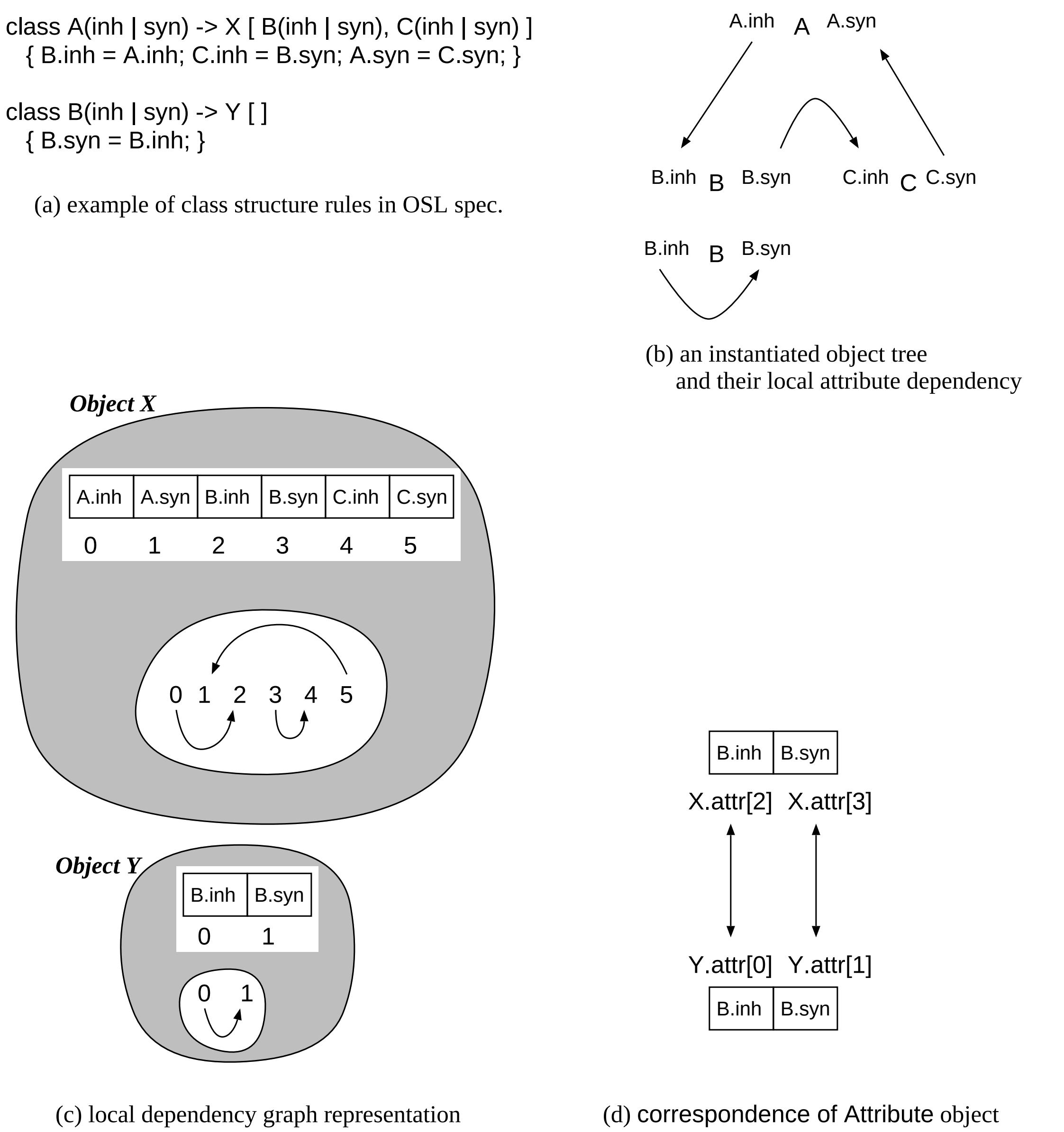
When an object tree is constructed, it is needed to manage two Attribute objects in the parent and child tree node represent the same attribute instance. We must maintain correspondence of these two Attribute objects. Figure 7-(d) shows the correspondence of Attribute object between two nodes. For this correspondence, the evaluator have to know an object-ID of referring child (or parent) and an index value of Attribute array. However, the current implementation stores the corresponding Attribute object-ID in the Attribute object (`value2` field in figure 5). This value will be computed in grafting of two nodes will be done.

3.3.4. Initializing Node Objects

Although initialization codes of Node object are generated by the OSL translator described in 3.3.1, we explain here about linking of Attribute objects between tree node described in 3.3.3, because it is slightly complicated.

The location of the corresponded Attribute object changes depending on which RHS object is grafted in actual object tree. Therefore it is needed to correct the correspondence of Attribute object, when Node object is connected by graft operation. To put it more concretely, although corresponded Attribute object can be gotten by the form `node_ptr->attr[i]` where `node_ptr` represents pointer to Node object, a variable `node_ptr` and `i` will change which RHS object have been grafted in that place.

We have to store corresponded `i` or object pointer itself in Attribute object. In the current implementation, we store a computed value in the Attribute object in grafting of two objects, and use its cached value in the referring time.



(a) example of class structure rules in OSL spec.

(b) an instantiated object tree and their local attribute dependency

(c) local dependency graph representation in each Node object

(d) correspondence of Attribute object between object X and object Y

Figure 7: Representation of Attribute Graph

3.4. Persistent Object Allocation in the Database

MAGE2 allocates all objects finally in the database. It uses the persistent object allocation mechanism of ODBMS for this purpose.

Many ODBMSs have the C++ library interface and can allocate STL container classes also persistently such vector, list and set persistently. (STL is an ISO C++ standard library.) So the attribute evaluator is constructed using STL classes in MAGE2 implementation.

When applying an attribute evaluator to persistent objects, we have to consider about execution speed of it. ODBMSs that have a client caching architecture can access to objects in the database as fast as in the heap memory, because they map the database in the memory. So the MAGE2's attribute evaluator will work in reasonable speed. For example, ObjectStore[7] and Objectivity/DB[8] have client caching architecture.

We are now implementing persistent object trees using the ObjectStore. Although there are some difference about a method for persistent allocation and use of an object-ID depending on ODBMSs, we show a procedure in the case of the ObjectStore.

- Replace object allocation code to using persistent `new` of ODBMS.
  For example, `Node` object allocation are written following in ObjectStore:

  ```
  Node *node = new(db, os_ts<Node>::get()) Some_RHS_Node_Class();
  ```

  where `db` is a database and `os_ts` is a type information.

- Replace default allocator of STL classes to persistent version of an ODBMS's allocator class.
  For example, set of `Node*` are often used in the attribute evaluator;

  ```
  typedef set<Node*> nodep_set;
  ```

  are changed like following.

  ```
  typedef os_allocator<Node*, os_value_node<Node*>> nodep_set_alloc;
  typedef std::set<Node*, nodep_set_alloc>          nodep_set;
  ```

3.5. Benchmark Result

In order to show that an object tree in the database is evaluated in reasonable speed, we have performed an simple benchmark test. In this test, we use Fibonacci program that create many number of objects dynamically (see appendix B). In the i-th turn of OOAG evaluation loop, it creates 2^{i-1} objects in the highest case and replaces leaf nodes to them. Finally, it stops evaluation after creating $fib(n)$ leaf objects.

We record the execution time of the non-database version and database version of evaluator from `fib.in = 10` to `12`. Table 3 shows result of this test.

The difference of execution time between non-DB and DB version widen as the number of objects, but it is proper result taking into consideration of character of the program — it creates and deletes many objects frequently.

4. Application Example

We have started the MAGE2 system development for generating a repository system for software development environments[9]. But it is useful for managing other structural data objects. For example,

n	Execution time [s]		
	non-DB version	DB version	(created DB size)
10	11	24	(500 KB)
11	19	49	(930 KB)
12	34	98	(1,500 KB)

These results have been get
under following environment:
Intel MMX Pentium 266MHz PC
64MB Memory
Windows 2000 / ObjectStore PSE Pro

Table 3: Execution time of Fibonacci program

XML documents are fully structured text data and favorably it has tree structures. However we consider about constructing OSL class code repository here that we are developing.

Although we have constructed MAGE2 translator by the Synthesizer Generator, we are now constructing it by MAGE2 itself. Furthermore we intend to construct an OSL specification repository from this translator code using MAGE2's ODB system generator features.

If we construct the translator by MAGE2 as usual, it will create production trees in an ODB as persistent data. Suppose output code of C++ classes is generated in synthesized attribute at root node of production trees. C++ class code is always computed consistently correspond to the production tree stored in the database. However, we need to construct the OSL syntax parser by another tool separately and pass the production tree to the translator, because MAGE2 does not have parser generator functions.

In the OSL translator development, a production tree of OSL specification file contents may be defined in a list of LHS class definitions. But it does not have special meaning to manage by "file", when creating OSL code repository. So it can be defined in separate trees for each LHS classes. This makes it easy to manage OSL specifications in class units. This means we do not declare

```
class osl_spec -> OSL_Spec[lhs_class_decl_list(|)]
```

but

```
class lhs_root -> Lhs_Root[lhs_class_decl(|)]
```

Now, we must construct the OSL translation system interface in "main()" function. Although [9] indicates advanced applications of OOAG to software development environment construction, we construct OSL code repository as only simple application here, that is, we implement only following functions: storing and retrieving of OSL code and retrieving corresponding C++ code. So we only create dictionary from LHS name to root node of production tree. Such dictionary can be implemented by basic function of ODBMS. Figure 8 shows our OSL translation system and code repository system.

1. Creating abstract syntax tree for OSL.

2. Defining attribution rules for translating from OSL to C++ classes.

3. Creating OSL parser. This parser generates OSL production trees in a text form.

4. Designing OOAG message interface for retrieving codes from database.

5. ODBMS interaction such database initialization, transaction, look up interface and so on.

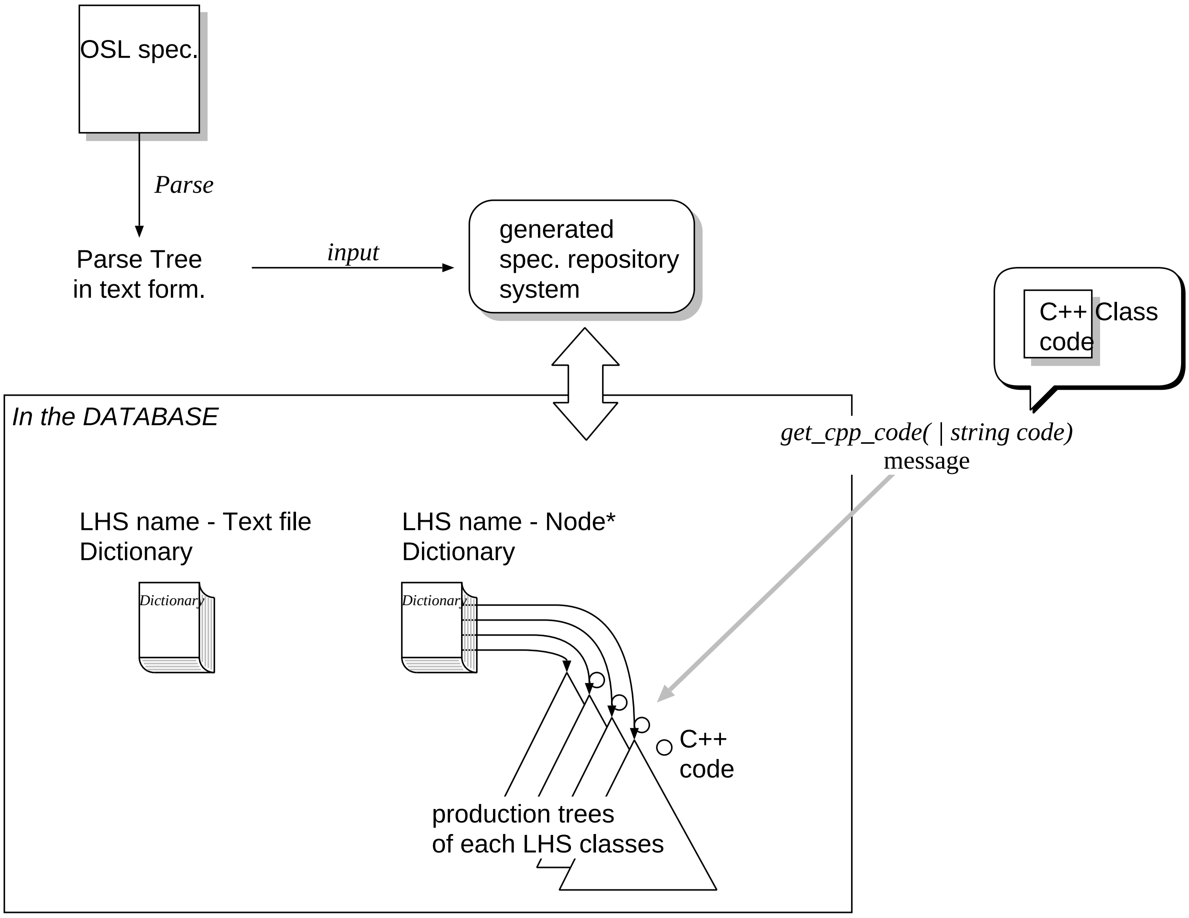


Figure 8: OSL Specification Repository System

5. The comparison with conventional database development

In this section, we make a comparative study between the our way of developing databases and the conventional way especially for software development environments.

5.1. Handling small grain software objects

In the tool development of conventional software development environments, each development tool have different representations of the objects and wrote them in individual files. But these individual object representations should be integrated into common representations in order to make it easier to combine and reuse various tools. For example, if compiler and static program analyzer are highly integrated where ASTs are used as common data representation, static program analyzers do not have to parse the program again, and also reuse useful information like symbol tables.

In order to find smart common data representation, many studies have been done; software databases or repositories to handle small grained objects with complicated types (e.g. PCTE OMS[10], CDIF, etc.).

PCTE OMS aims to share CASE objects by typing in the E-R model, and CDIF tries to provide the uniform format to exchange objects between tools.

The Mjølner[11], one of these studies, utilizes abstract syntax trees (ASTs) as the common representation of the program information. It could construct software development environments for manipulating programs. Our study belongs under this category.

For the sharing information, we must handle software objects in small grained. If we do not handle small grained objects, we have to begin the work with breaking down large objects into convenient small parts when we will construct some tool. This makes no sense in sharing objects between tools.

ASTs are a good choice of sharing objects for an environment to handle programs, as their parts are small enough and highly abstracted.

Our plan is to widen a application area to not only handling programs but other software development lifecycle such as design document handling. Key points are “representation method” and “database construction method” of software objects. About a grammar based representation method of software objects are discussed in [9]. So we discuss about the following in the rest of this section.

(5.2) Programming with the RDBMS

(5.3) Programming with the ODBMS

(5.4) Special systems for software database development

5.2. Making the database with hierarchical classification

In this section, we show why it is difficult to manage software objects using traditional RDB. We also show the solution in our approach.

RDB is not suitable for managing software objects, mainly because:

- The granurarity of software objects are widely diverse, so it is difficult to model them by using fixed-length record in RDB.
- The relatoins among software objects are also widely diverse, so it is difficult to model them by indexing in RDB.

In our methodology,

1. Various granurarity of software objects is modeled naturally by using CFG, since CFG gives an uniform way to handle software objects with simple descriptions.

2. Various relations among software objects are encoded as attribute values.

   For example, let us take the case of compiler construction using AGs. As a typical relation, there is a relation between a declared variable and the use of the variable, called def-ref relation here. We collect information about all defined variables in a synthesized attribute on the root as a symbol table and distribute the symbol table through inherited attributes. This symbol table makes it possible to detect inconsistent def-ref relation such as refered but not defined variable. To summarize, semantic definition using attributes can provide smart ways to handle complex relations among software objects.

5.3. Programming directly with the ODBMS

This section explains why ODBMS is not good enough for our purpose.

ODBMS is a good tool to program software databases, since ODBMS can describe any structured database schema, and user-defined complex typed objects. Furthermore, in ODBMS, we can use various convenient library functions for persistency, mutual exclusion, versioning and so on.

Then, what are we dissatisfied with the programming in the ODBMS? The answer is that ODBMS requires too low level programming. This property is not acceptable, because what we want here is a high-level description tool for software databases in order to make the best use of ASTs as common data representation.

5.4. OOAG: Specialized System for Software Database Development

This section briefly presents our idea using OOAG for software database development.

In the previous two sections, we pointed out the following two requirements for software database construction tool.

- The tool should handle various granurarity and relations of software objects.
- The tool should have a high-level description language for them.

Moreover, we point out the third requirement:

- The tool should efficiently manage stored object, especially derived, and small-grained objects.

For example, PCTE OMS does not satisfy the last requirement. Efficient algorithms to maintain derived small-grained objects have been studied, e.g. [12, 13].

OOAG aim a software database construction tool that satisfies the above all requirements. Basic ideas and distinctive features of OOAG compared with conventional AGs are follows:

- OOAG can modify attributed tree dynamically in its computation mechanism. This ability enable us to describe object management in software databases while preserving the advantages of AGs.
- Parsing trees, derived values, and relations as attribute values are stored into database directly, which implies that various tools can share and reuse not only parsing trees, but also attribute values.

With the above properties, OOAG is useful in the following situations.

1. Constructing pattern matching tool based on syntactic structure.

   For example, the tool searches all parts that match "**if** $(\langle exp \rangle = \langle exp \rangle)$ **then** $\langle exp \rangle$ {**else** $\langle exp \rangle$}", and replace them by "**if** $(\langle exp \rangle == \langle exp \rangle)$". It is easy to construct the tool because OOAG handle ASTs as common data representation.

2. Sharing context-sensitive information between tools.

   For example, symbol tables that are created by compiler can be shared and reused by other tools, since the symbol tables exist in database and are accessible.

3. Constructing tools incrementally.

   The advantages inherited from AGs guarantee high readability and high maintainability of the tools.

6. Conclusion

In this paper, we introduced the MAGE2 system as an ODB generator construction tool. A constructed tool by MAGE2 system creates attributed object trees in an ODB persistently. Operations to persistent object trees can be described in OSL specifications by the message passing mechanism of OOAG model. Programmers who use MAGE2 will only write interface codes between generated tool.

Created ODBs will be maintained by the OOAG evaluator. This includes updating, adding or deleting objects. They will be described in dynamic subtrees operation by message passing. If the state of object trees in the database will be inconsistent, OOAG evaluation loop will keep them consistent.

From above features of MAGE2 system, we can develop easily complicated object-oriented database systems which manage structured data efficiently.

7. Future Works

From the viewpoint of an execution speed of attribute evaluation, it is better to implement more efficient evaluator such ordered AGs [14] one. We plan to implement another evaluator using improved ordered AGs class OAG* introduced in [15].

Bibliography

[1] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[2] GrammaTech, Inc., One Hopkins Place, Ithaca, NY 14850, U.S.A. *The Synthesizer Generator Reference Manual*, 4.1 edition, 1993.

[3] Yoichi Shinoda and Takuya Katayama. Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm. In *Proceedings of the International Workshop on Attribute Grammars and their Applications,* Lecture Note in Computer Science *Vol. 461*, pages 177–191. Springer-Verlag, 1990.

[4] Yoichi Shinoda. *On Application of Attribute Grammars to Software Development*. PhD thesis, Tokyo Institute of Technology, 3 1989.

[5] Yasuhiro Iida. OSL language specification ver.2.0 (in Japanese). Technical Report 96TR-0012, Department of Computer Science, Tokyo Institute of Technology, June 1996.

[6] Object Design Inc. 25 Burlington Mall Road Burlington, MA 01803-4194. *ObjectStore User Guide: Release 3.0 for UNIX Systems*, 1993.

[7] Object Disign, Inc., Burlington, MA 01803-4194. *ObjectStore PSE/PSE Pro for C++ Tutorial and API User Guide*, September 1998.

[8] Objectivity, Inc. *Objectivity/C++*, June 1996.

[9] Katsuhiko Gondow, Takashi Imaizumi, Youichi Shinoda, and Takuya Katayama. Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars. In *Object Technologies for Advanced Software*, pages 77–94. Springer-Verlag, 1993. LNCS 742.

[10] Gerard Boudier, Ferdinando Gallo, Régis Minot, and Ian Thomas. An Overview of PCTE and PCTE+. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 248–257, November 1988.

[11] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *OBJECT-ORIENTED ENVIRONMENTS: THE MJØLNER APPROACH*. PRENTICE HALL, 1993.

[12] Scott E. Hudson and Roger King. CACTIS: A Database System for Specifying Functionally-Defined Data. In *Proceedings International Workshop on Object-Oriented Database Systems*, 1986.

[13] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, A Graph-Oriented Database System for Engineering Applications. In Jarzabek Lee, Reid, editor, *CASE '93 6th Int. Conf. on Computer-Aided Software Engineering*, pages 272–286. IEEE Computer Society Press, 1993.

[14] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[15] Shin Natori, Katsuhiko Gondow, Takashi Imaizumi, Takeshi Hagiwara, and Takuya Katayama. On Eliminating Type 3 Circularities of Ordered Attribute Grammars. In *2nd Int. Workshop on Attribute Grammars and their Applications (WAGA'99)*, pages 99–112. INRIA rocquencourt, 3 1999.

A. Example of OSL specifications

We show small example of OSL specifications "dig.ooag" below. Their specifications are for representing and interpreting a digit sequence.

Suppose we pass following text as an initial object tree:

```
N1[I1[I1[I2[D1[1]], D1[1]], D1[1]]]
```

The generated system creates object tree and waits for messages `value` or `change`. If message `value` is received by an object `N1`, its output attribute value is evaluated according to its input attribute `radix`. In this program, digit sequence will be interpreted by `radix`; for example if `value(10|val)` will be send to `N1`, `val` will be 111; and if `value(8|val)` will be send, `val` will be 73. Message `change_pos` will change specified subtree to `new_digit`.

```
class N(|) -> N1[I(int scale|)]
  { I.scale = 0; }

    N:value(int radix|int val) => I:value(int radix|int val)
      { I.radix = N.radix; N.val = I.val; }

    N:change(int change_pos, D new_digit|)
      => I:change(int change_pos, D new_digit|)
      { I.change_pos = N.change_pos; I.new_digit = N.new_digit; }

class I(int scale|)
  -> I1[I(int scale|), D(int scale|)]
  { I$2.scale = I$1.scale + 1; D.scale = I$1.scale; }

    I$1:value(int radix|int val)
      => I$2:value(int radix|int val), D:value(int radix|int val)
      { I$2.radix = I$1.radix; D.radix = I$1.radix; I$1.val = I$2.val + D.val; }

    I$1:change(int change_pos, D new_digit|)
      case(I$1.change_pos == I$1.scale)
        =>
        { (new D) = I$1.new_digit; }

      otherwise
        => I$2:change(int change_pos, D new_digit|)
        { I$2.change_pos = I$1.change_pos; I$2.new_digit = I$1.new_digit; }

  -> I2[D(int scale|)]
  { D.scale = I.scale; }

    I:value(int radix|int val) => D:value(int radix|int val)
      { D.radix = I.radix; I.val = D.val; }

    I:change(int change_pos, D new_digit|)
      case(I.change_pos == I.scale)
        =>
        { (new D) = I.new_digit; }

class D(int scale|)
  -> D1[int digit]
   {}
    D:value(int radix|int val) =>
      { D.val = digit * D.radix exp D.scale; }
```

B. Fibonacci program used in a benchmark

When we pass " `Root[FibNull[]]`" as an initial object tree to this program, it grows replacing `FibNull[]` to `FibOne[]` or `FibPair[FibNull[], FibNull[]]` as its `fib.in` value.

```
class top(|int out) -> Root[fib(int in|int out)]
  { fib.in = 5; top.out = fib.out; }

class fib(int in|int out)
  -> FibNull[]
  { fib.out = 0; }

      case(fib.in < 2)
        =>
        { (new fib) = FibOne[]; }
      otherwise
        =>
        { (new fib) = FibPair[FibNull[], FibNull[]]; }

  -> FibOne[]
  { fib.out = 1; }

  -> FibPair[fib(int in|int out), fib(int in|int out)]
  { fib$1.out = fib$2.out + fib$3.out;
    fib$2.in = fib$1.in - 1; fib$3.in = fib$1.in - 2; }
```