Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS[1]

José L. Sierra, Antonio Navarro, Baltasar Fernández-Manjón,Alfredo Fernández-Valmayor

Dpto. Sistemas Informáticos y Programación. Fac. Informática. Universidad Complutense. 28040. Madrid. Spain
{jlsierra, anavarro, balta, alfredo}@sip.ucm.es

Abstract. ADDS is our proposal for the development of content-intensive applications. Applications in ADDS are produced by processing documents that describe their main aspects and that have been marked up using application-dependent, Domain-Specific Markup Languages (DSMLs). To make ADDS of practical value, we promote the incremental definition of DSMLs, as well as their incremental operationalization (i.e. the incremental construction of their processors). In this paper we describe how this incremental definition is carried out by using a technique called PADDS (DSML Provision in ADDS), and how incremental operationalization is accomplished using a model called OADDS (Operationalization in ADDS).

Keywords: Domain-Specific Markup Languages, Modular Document Grammars, Modular Language Processors, Software Development Approach, XML

1 Introduction

With the advent of languages like HTML [26] and other proposals like HyTime [9] (a SGML [8] extension for the description of hypermedia applications) at the end of the 80's and beginning of the 90's it became evident that descriptive markup languages could be used in software application development, regardless of their being initially intended for the publishing domain [5]. This use was later popularized in the XML world [26], where descriptive markup languages were conceived as a standard way for information interchange between applications, despite being initially geared to simplifying the definition, maintenance and processing of electronic documents. We have applied these ideas in order to create a *document-oriented* paradigm for the development of *content-intensive* applications that we have already used in several domains: educational [7] and hypermedia applications [17], as well as knowledge-based systems [19]. In all these domains, we have successfully defined and used descriptive, application-dependent, *domain-specific markup languages* (DSMLs) to make the structure of documents that describe the main aspects of the applications (e.g. contents and user interface appearance) explicit. Moreover, by processing applications associated documents (i.e. with suitable processors for these DSMLs), the final applications are automatically generated.

During our development experiences, we realized that the systematic applicability of the document-oriented paradigm is heavily dependent on the incremental formulation of DSMLs, as well as on their incremental operationalization (i.e. how their processors are incrementally built) [22]. The adoption of an incremental strategy helps to palliate the initial high costs associated with the provision of the DSMLs and their processors. This also provides the flexibility required by the development of complex applications, because the DSMLs can be extended when new markup needs are discovered. Finally, this strategy also avoids having to include very general or complex markup structures in the DSMLs consequently easing their use. Our *Approach to Document-based Development of Software* (ADDS) is a specific implementation of the document-oriented paradigm that promotes these strategies. This approach is detailed in [22][25].

The present paper is focused on how DSMLs are incrementally defined in ADDS, and how they are incrementally operationalized. The first of these aspects is covered by the PADDS technique (*Provision of DSMLs in ADDS*). The second one is addressed by the OADDS model (*Operationalization in ADDS*). The structure of this paper is as follows: Section 2 introduces a simple example that will be used throughout to illustrate its different aspects. Section 3 analyzes

[1] The Spanish Committee of Science and Technology (TIC2001-1462, TIC2002-04067-C03-02 and TIN2004-08367-C02-02) has partially supported this work.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

the PADDS technique. OADDS is described in section 4. In section 5 we compare our approach with some related work. Finally, section 6 presents some conclusions and future work.

2 A Simple Application Domain

This section introduces a simple and well-known application domain (*evaluation of arithmetic expressions*) that will be used to exemplify the different aspects discussed in the following sections. Although we are aware that this illustrative example, classic in the field of language processors [1], might be considered a strange choice because it does not fit into the claimed problem domain of content-intensive applications, its simplicity avoids the collateral details of more realistic applications, such as those used in [22][25] to illustrate the higher – level aspects of ADDS. In addition, the richness in the markup of its documents lets us focus on the technical issues of the incremental provision and operationalization of DSMLs, which is the main goal of the present paper.

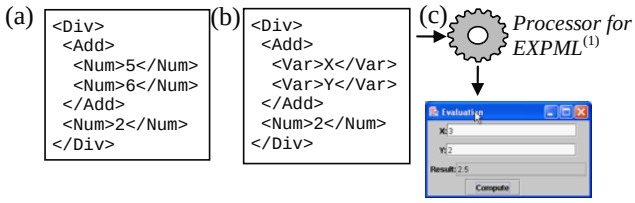


Fig. 1 (a) EXPML(0) document for `(5+6)/2`, (b) EXPML(1) document for `(X+Y)/2`, (c) the document in (b) can be processed with a processor for EXPML(1) to yield an application that takes the values of the expression's variables and computes the value of this expression.

The DSML in this example, which is called EXPML, enables the description of arithmetic expressions. We will consider two different evolutions of this DSML: EXPML(0), for the markup of basic expressions, and EXPML(1), for the incorporation of variables into the expressions. Thus, arithmetic expressions will be represented as marked documents conforming to one of these EXPML evolutions, such as shown in Fig. 1a and Fig. 1b. In turn these documents will be processed with a suitable processor to generate evaluators for these expressions (Fig. 1c).

3 The PADDS Technique

PADDS (*Provision of DSMLs in ADDS*) gives a set of guidelines for the incremental definition of DSMLs in ADDS. The following subsections explain the technique. Subsection 3.1 gives an implementation-independent description of the technique. In its turn, subsection 3.2 shows one of the possible implementations based on XML *Document Type Definitions* (DTDs) [8][16].

3.1 Implementation-independent description of PADDS

DSMLs in ADDS are provided incrementally by formulating *linguistic modules* through the combination and extension of simpler ones. Intuitively, a *linguistic module* is a grammatical characterization of a part of the final DSML. In this module each syntactic category is represented as an *element type* and has a *content model* associated with it. In our simple application domain we can have two different linguistic modules: *lmexp0*, which defines markup for basic expressions, and *lmexp1*, which extends *lmexp0* with markup for variables. Therefore the incremental definition of DSMLs largely depends on being able to combine and extend linguistic modules. To facilitate this, PADDS introduces three mechanisms. First, the content models for the element types in a module can refer to definitions in other modules. In grammatical terms this is equivalent to the combination of pre-existing grammars by adding new productions. Second, a module can include a set of *parameters*. These parameters play a role similar to the parameter entities in the SGML or XML DTDs [8][16], and they can be specialized with element types in the same module or in others. Hence,

they can be grammatically interpreted as *undefined* non-terminals leading to open grammars that can be subsequently extended. Third, a module can add new attribute lists to the element types defined in other modules.

Once all the linguistic modules for a DSML are available, the DSML itself must be described with a *document grammar*. This grammar is obtained by following a *grammar production specification*. This specification has a double aim. On the one hand, this specification is used for resolving conflicts between linguistic modules (e.g. name conflicts). On the other hand, this can be used to adapt the concrete markup vocabulary to different contexts (e.g. different languages such as English or Spanish). Also, by providing alternative production specifications it is possible to get different alternative document grammars for the same DSML. These grammars generate isomorphic markup languages, each one of which with a different concrete markup vocabulary. This feature enhances the usability of a DSML because this can be adapted to the particular social and cultural circumstances of the domain experts. For example, a markup vocabulary is much better understood and accepted by a domain expert if it is presented in his/her mother tongue.

3.2 An implementation of PADDS based on XML DTDs

PADDS is a conceptualization that can be implemented using any of the multiple schema languages proposed in the XML arena [13]. Based on usability and comprehensibility criteria we have developed an implementation based on XML DTDs [26] regardless of their expressive limitations [15]. Although XML DTDs are less powerful than other schema languages like XML Schema [26] or Relax NG [4], they are more understandable and simpler-to-use mechanisms for domain experts [16]. This subsection describes our implementation.

We have defined the notation used in Fig. 2a to define linguistic modules. The parameters of the module can be listed in the **Parameters** section. The **Specializations** section is used to add element names to parameters defined in the same or in other modules (i.e. to indicate *parameter specializations*). Finally, the grammar of the sublanguage described in the module can be specified as a set of productions of an XML DTD in the **Grammar** section. The dot in the names of element types and parameters has a special meaning: to prefix a name with the name of the module where it is defined. Notice that in modules whose only purpose is to integrate simpler modules the grammar section will be empty.

(a)

Module: *module-name*
[**Parameters:** {*parameter-name*}+]
[**Specializations:** {*parameter* ⊇ {*element-name*}+}
[**Grammar:** *dtd*]

(b)

{ **Module:** *used-module-name*
{ **Map:** *element-type-name-in-DTD* [→ *resulting-element-type-name*]
{**AMap:** *attribute-name-in-module* [→ *resulting-attribute-name*]
{ *symbol-name-in-module* [→ *resulting-symbol-name*] }*
}*
}*}+

Fig. 2 (a) A notation for the definition of linguistic modules; (b) a notation for the description of grammar production specifications.

Grammar production specifications are defined using the notation of Fig. 2b. Basically, they name the linguistic modules to be used, and, for each module, a set of renaming rules for the markup vocabulary in this module. Thus, each module used is specified in a **Module** section. In this section it is also possible to specify a set of renaming rules for the different element types of the modules. The renaming of each element type is specified in a **Map** section. In this section a new name for the element tag can be specified, and also a set of renaming rules affecting the attributes for the element. The rules for each attribute are grouped in an **AMap** section, where a new name for the attribute can be chosen, and a set of renaming rules for symbols in this attribute domain can be specified. Notice that these renaming facilities enable the resolution of the name conflicts produced by collisions among the DTDs of the different linguistic modules giving an analogous functionality to the use of XML namespaces [26][2].

[2] Nevertheless, it should be noted that the only objetive of grammar production specifications is to simplify the authoring of the documents does *not intend to be nor is conceived* as an alternative to namespaces. Another different PADDS implementation based on a namespaces-aware schema language could take advantage of the namespaces mechanism in the implementation of grammar production specifications.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

(a)
```
Module: lmexp0
Parameters: EXP
Specializations:
   EXP ⊇ {Num,Add,Sub,Mul,Div}
Grammar:
 <!ELEMENT Num   (#PCDATA)>
 <!ELEMENT Add   (EXP,EXP)>
 <!ELEMENT Sub   (EXP,EXP)>
 <!ELEMENT Mul   (EXP,EXP)>
 <!ELEMENT Div   (EXP,EXP)>
```

(b)
```
Module: lmexp1
Specializations:
   lmexp0.EXP ⊇ {Var}
Grammar:
<!ELEMENT Var  (#PCDATA)>
```

(c)
```
Module: lmexp0
   Map:Add → Suma
       Sub → Resta
Module: lmexp1
```

Fig. 3. (a) Linguistic module *lmexp0*; (b) linguistic module *lmexp1*; (c) grammar production specification for the Spanish version of EXPML(1).

Fig. 3a and Fig. 3b show the definition of the linguistic modules associated with EXPML. *Lmexp0* is defined in Fig. 3a. Notice that this module is made parametric in the possible arithmetic expressions (EXP parameter), which is its main point of extension. *Lmexp1* is defined in Fig. 3b. This module specializes the EXP parameter of *lmexp0* with the element type Var for the markup of variables. Hence *lmexp0* is extended. In turn, the grammar production specification for the Spanish version of EXPML(1), is shown in Fig. 3c. This specification renames the Add and Sub tags with the corresponding Spanish tags Suma and Resta.

```
<!ENTITY % lmexp0.EXP "(Div|Mul|Num|Resta|Suma|Var)">
<!ELEMENT Num   (#PCDATA)>
<!ELEMENT Suma  (%lmexp0.EXP;,%lmexp0.EXP;)>
<!ELEMENT Resta (%lmexp0.EXP;, %lmexp0.EXP;)>
<!ELEMENT Mul   (%lmexp0.EXP;, %lmexp0.EXP;)>
<!ELEMENT Div   (%lmexp0.EXP;, %lmexp0.EXP;)>
<!ELEMENT Var  (#PCDATA)>
<!ATTLIST Num adds.cls NMTOKEN #FIXED "lmexp0.Num lmexp0.EXP">
<!ATTLIST Suma adds.cls NMTOKEN #FIXED "lmexp0.Add lmexp0.EXP">
<!ATTLIST Resta adds.cls NMTOKEN #FIXED "lmexp0.Sub lmexp0.EXP">
<!ATTLIST Mul adds.cls NMTOKEN #FIXED "lmexp0.Mul lmexp0.EXP">
<!ATTLIST Div adds.cls NMTOKEN #FIXED "lmexp0.Div lmexp0.EXP">
<!ATTLIST Var adds.cls NMTOKEN #FIXED "lmexp1.Var lmexp0.EXP">
```

Fig. 4 DTD for the Spanish version of EXPML(1).

In PADDS, document grammars are represented by using DTDs. The final DTD for a DSML can be systematically produced from the linguistic modules and the grammar production specification. Each parameter *p* in a module *m* is represented in the final DTD by a parameter entity named *m.p*. The value of this XML entity will be the disjunction of the element types that specialize the *p* parameter, once they have been renamed according to the renaming rules of the production specification. The DTDs of the modules are then processed according to the renaming rules, their parameters are substituted by references to the corresponding parameter entities, and the result is appended to the final DTD. Finally, this DTD is enriched with fixed-valued attributes that relate its markup vocabulary to that defined in the linguistic modules. The values of these attributes will be used during operationalization to decouple the processing from the names chosen in the different production specifications. In this way, an `adds.cls` attribute is associated with each element type in the DTD. This attribute indicates the module and the element type where such a DTD's element type is originally defined. In addition, if a parameter is specialized by an element type, this is also indicated in the value of this attribute. The value itself is a list of tokens with the format *module.elementTypeOrParameter*. Similarly, the renaming rules for the attributes and the symbols in their domains are encoded by using pre-established fixed-values attributes. For each element type whose attributes have been renamed, the map is represented as the value of an `adds.amap` attribute. In the same way, if *a* is an attribute of an element type with renaming rules for the symbols in its domain, these rules are encoded as the value of an `adds.smap.`*a* attribute. The DTD for the Spanish version of EXPML(1) is shown in Fig. 4.

To conclude it should be noted that the manual production of document grammars according to this implementation of PADDS is a tedious task. To facilitate this task we have developed a tool for the automatic production of DTDs from the linguistic modules and the grammar production specification for a DSML.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

4 The OADDS Model

The production of applications from marked documents requires appropriate processors for the DSMLs used. Because DSMLs are incrementally provided, their processors must also be incrementally built. Addressing the operationalization problem in ADDS has led us to devise OADDS (*Operationalization in ADDS*), a conceptual model for the incremental development of DSMLs' processors. OADDS adapts the classic ideas of syntax-directed translation [1] with concepts taken from the well-known approaches to the development of modular language processors [6][11][14].

This section details OADDS. Subsection 4.1 gives an overview of the model, which is heavily based on the incremental operationalization of document trees. Subsection 4.2 details the structure of such operationalized document trees, and subsection 4.3 describes the components involved in such an incremental operationalization. Finally, subsection 4.4 deals with the OADDS processors themselves.

4.1 Overview

OADDS regulates the incremental construction of DSMLs processors in ADDS. The main components used in this construction are the *operationalization modules*. Operationalization modules are used to attach suitable operational meanings to the linguistic modules used to define the DSMLs. In addition to the operationalization modules, OADDS processors also include other predefined facilities, and a small domain – dependent program that brings together all the other components of the processor.

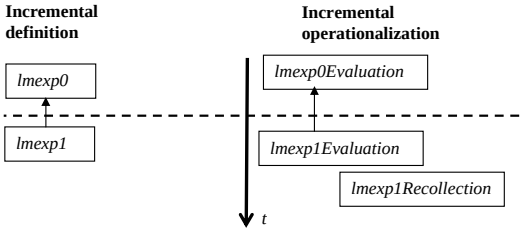


Fig. 5 Incremental definition of EXPML versus its incremental operationalization.

Processors use the operationalization modules to operationalize element nodes in the document trees by *decorating* them with a set of *semantic attributes*, together with the code for computing the values of these attributes. Trees operationalized in this manner can be *evaluated* subsequently by invoking the code that decorates their roots. This decoration can be incrementally performed. Hence semantic attributes can be incrementally added to the element nodes, and the code for computing their values can be incrementally extended. This is one of OADDS' key features for enabling the incremental operationalization of DSMLs.

Therefore, the incremental definition of linguistic modules in PADDS is mirrored in OADDS with the incremental construction of operationalization modules. This point is illustrated in Fig. 5, where the provision of linguistic modules for EXPML is parallel to the incremental construction of operationalization modules for evaluating the EXPML expressions. Thus, the provision of markup for the basic expressions (regulated by *lmexp0*) leads to the construction of the *lmexp0Evaluation* operationalization module for computing the values of such expressions. The extension of *lmexp0* by *lmexp1* to contemplate variables supposes the extension of *lmexp0Evaluation* by *lmexp1Evaluation*. This new module will deal with the evaluation of the variables, and also with the propagation of a binding for the variables along the inner nodes of the document trees. This propagation will be added without changing *lmexp0Evaluation*. In turn, the module *lmexp1Recollection* lets the final processor recollect the variables used in the expressions. These variables will be used to configure the user interfaces of the final running applications.

4.2 Operationalized Document Trees

The most characteristic task carried out by an OADDS processor is the *evaluation* of document trees (i.e. the calculation of values for the semantic attributes in their element nodes). To enable this evaluation, document trees must be

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

previously *operationalized* (i.e. appropriate semantic attributes and the code to compute their values must be assigned with each element node).

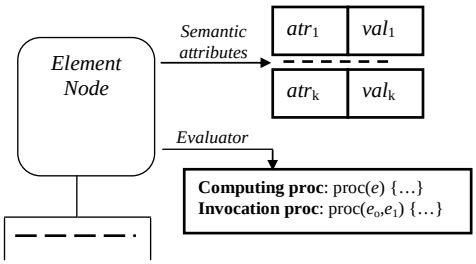


Fig. 6 Operationalization of element nodes in OADDS.

The operationalization of element nodes in document trees is depicted in Fig. 6. During this process, the code to compute the values of the attributes is provided by an *evaluator*. In turn, evaluators are composed of two different procedures: (i) a *computing procedure* that actually carries out the computation, and (ii) an *invocation procedure*, which is used by the computing procedure to invoke the computing procedures of other evaluators.

The computing procedure takes an element node *e* as input, and computes the values of its semantic attributes. This computation will normally be based on the values of the attributes for the element nodes in the vicinity of *e*. Before consulting these values, the procedure has to call the computing procedures in these element nodes. These calls must be carried out using the invocation procedure for the evaluator in *e*. This procedure takes two inputs (the element node associated with the caller computing procedure, and the element node where the one called resides) and encodes how this invocation is performed. Thus, computing procedures are parametric in their calls to other computing procedures because they never directly perform these calls but use the invocation procedures of their associated element nodes. This enables the new operations surrounding the calls to be included (e.g. to propagate new attribute values between the associated element node and its vicinity) by only changing the corresponding invocation procedure. This is indeed the key issue to facilitate incremental operationalization in OADDS.

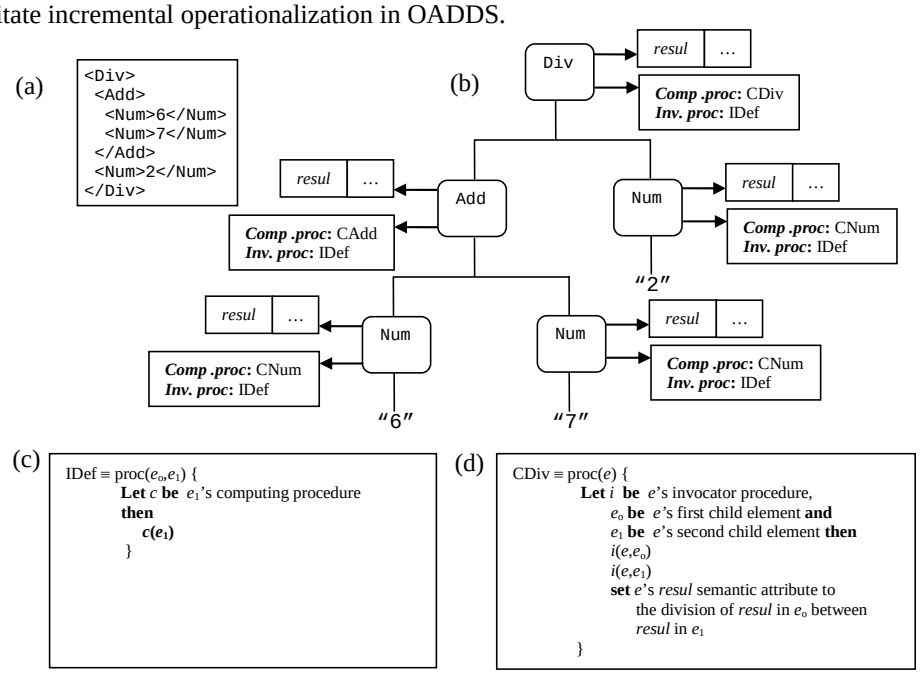


Fig. 7 (a) EXPML document for `(6+7)/2`, (b) an operationalization of the tree for the document in (a), (c) the `IDef` *default* invocation procedure, (d) the `CDiv` computing procedure.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

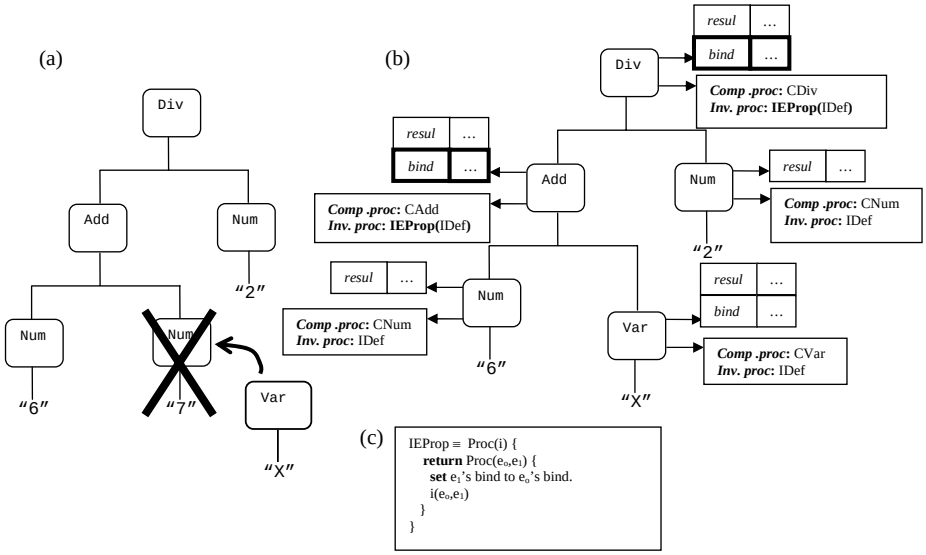


Fig. 8 (a) Modification of the document tree depicted in Fig. 7b, (b) operationalization of (a), (c) extension for invocation procedures `IEProp`.

Fig. 7 and Fig. 8 illustrate how the incremental operationalization of document trees is facilitated by the structure of the evaluators described above. In Fig. 7b an operationalized document tree for the `EXPML` document depicted in Fig. 7a is described. Note that a semantic attribute *resul* has been attached to each element node. This attribute will contain the value of the sub-expression corresponding to that node. In addition, appropriate evaluators have also been attached to these nodes. For instance, the evaluator for `Div` includes an invocation procedure called `IDef` and a computing procedure called `CDiv`. The pseudo-code[3] of `IDef` is included in Fig. 7c. This is a *default* invocation procedure that calls the computing procedure invoked without further calculations. In Fig. 7d the pseudo-code for `CDiv` is given. This procedure calls the computing procedures in the child element nodes and sets the *resul* attribute to the division of the values for *resul* in these children. Calls are always performed using the invocation procedure, and this is always accessed through the element nodes. Therefore, if this procedure is changed, the new one will be used without modification.

If the document tree in Fig. 7b is changed as described in Fig. 8a (i.e. the second argument of `Add` is replaced by the `X` variable to yield the `(6+X)/2` expression), its operationalization must be changed accordingly. The resulting operationalized tree is shown in Fig. 8b. Operationalizations for the element nodes preserved can be incrementally obtained from the existing ones. With each inner node a new semantic attribute (*bind*) is assigned, which will refer to the binding of variables with values. In addition, the invocation procedures of these nodes will be extended to deal with the propagation of this new attribute. The extension itself (`IEProp`), which is outlined in Fig. 8c, is represented as an invocation procedure parametric in another invocation procedure. This carries out the propagation, and then calls the invocation procedure given by its parameter. As shown in Fig. 8b, the final invocation procedure is obtained by applying `IEProp` to the existing one (in this case, `IDef`).

4.3 Operationalization Modules

Operationalization modules in OADDS group together a set of *operationalizers* and an *operational assignment*. The operationalizers enclose the code to operationalize a kind of element nodes. Thus, operationalizers are modeled as procedures that, by taking an element node as input, add the appropriate semantic attributes to this node, and return the evaluator to be used with the node. The reason for returning the evaluator is to allow other operationalizers to extend it.

[3] Note that we use pseudo-code because OADDS is a conceptual framework independent of specific programming languages or specification formalisms.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

In turn, the operational assignments determine the appropriate operationalizer to be used on each element node. In this way, they are modeled as procedures that by taking an element node as input, select an operationalizer and apply it to such a node.

The operationalization pattern described in the previous subsection makes the incremental provision of operationalization modules feasible. This incremental provision relies on the incremental provision of operationalizers and operational assignments. Operationalizers can statically call other operationalizers, or they can be left parametric in an evaluator (i.e. they can take an evaluator to be extended as additional input). In this case, they will be dynamically applied by operational assignments to extend evaluators produced by other operationalizers. Likewise, the incremental provision of operational assignments is achieved by calling other pre-existing operational assignments.

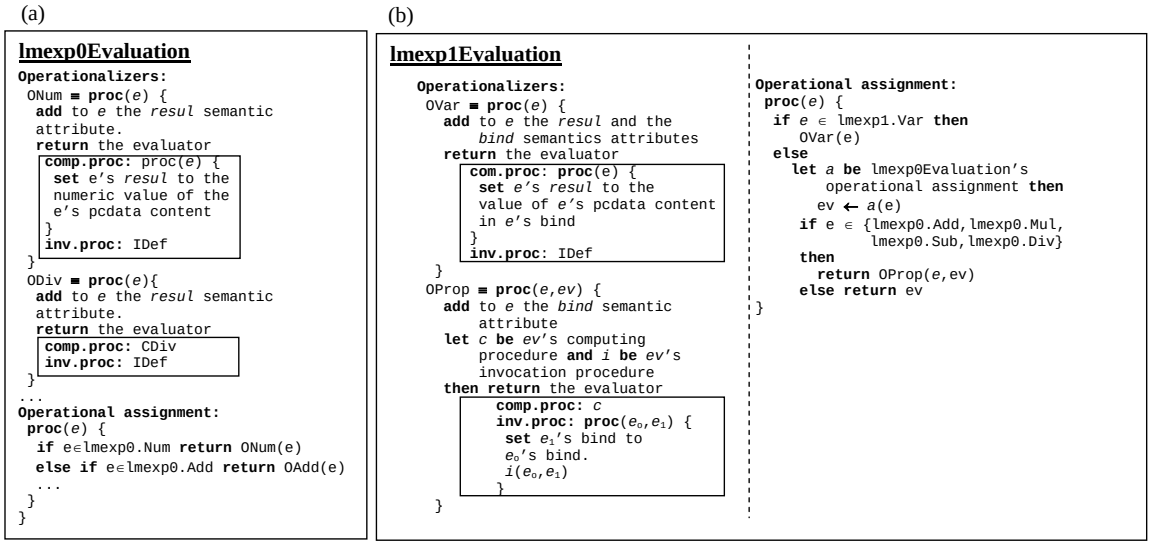


Fig. 9 Pseudo-code for the operationalization modules (a) `lmexp0Evaluation`, (b) `lmexp1Evaluation`.

The incremental provision of operationalization modules is exemplified in Fig. 9, where the pseudo-codes for *lmexp0Evaluation*, and *lmexp1Evaluation* (see subsection 4.1) are outlined. Thus, the pseudo-code for *lmexp0Evaluation* is partially outlined in Fig. 9a (operationalizers for dealing with additions, subtractions and products are analogous to `ODiv`). The pseudo-code for *lmexp1Evaluation* is shown in Fig. 9b. Besides `OVar`, the operationalizer for variables, this module introduces the parametric operationalizer `OProp`, which enables the propagation of the semantic attribute *bind* along the inner nodes of the document tree. Notice that this operationalizer is left parametric in an evaluator (i.e. in addition to the element node, it takes an evaluator as input). Then, it adds the semantic attribute *bind* to the element node, and produces an evaluator extending the invocation procedure of the evaluator accepted as input. The extension deals with the propagation of the attribute *bind* to the corresponding child element node.

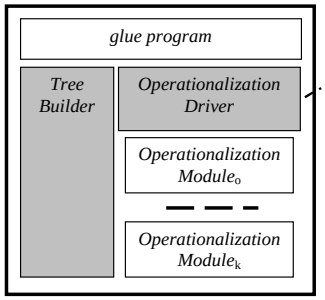
The operational assignments depicted in Fig. 9 always refer to the markup vocabulary introduced in the linguistic modules. Indeed, with $e \in$ *module.tag* we denote an expression that is true when *module.tag* appears in the value of *e*'s `adds.cls` attribute. This ensures the independence of the operationalization modules from the particular markup vocabularies chosen in the different grammar production specifications. Thus, by being aware of the values for the `adds.cls`, `adds.amap` and `adds.smap.`*a* attributes, the same processor will be able to work with the different versions of a DSML, regardless of their vocabulary.

4.4 Processors

The structure of a processor in OADDS is outlined in Fig. 10a. The processor uses a *tree builder*, an *operationalization driver* and an appropriate set of operationalization modules. While the two first components are predefined, operationalization modules vary with each particular processor. All these components are appropriately combined by a *glue program*. The tree builder parses marked documents and produces the corresponding document trees. In turn, the

operationalization driver enables the operationalization of document trees with operationalization modules. This component, which is modeled as a procedure in Fig. 10b, takes a tree and an operationalization module as input, and visits the element nodes in the tree, applying the module's operational assignment to each node, and setting their evaluators to the returned ones. Finally, the glue program dictates how to use all these components to process the marked documents[4]. Because almost all the work will be performed by the other components, this program will usually be small. Notice that, while glue programs are components specific to each processor and therefore hardly reusable, processors can extend others that are simpler by extending their operationalization modules as illustrated in section 4.3.

(a)



(b)

```
proc OperationalizationDriver(e,o) {
   clear the e's semantics attributes and
            the evaluator.
   Let a the operational assignment of o
   then
     ev ← a(e)
     set the evaluator of e to ev
     foreach child element e_h of e
          OperationalizationDriver(e_h,o)
}
```

(c)

```
prog {
  doc ← query the document URI.
  t ← build the document tree for doc.
  operationalizationDriver(t,lmexp1Recollection)
  c(t) with c the computing procedure of
          t's evaluator.
  cvars ← value of t's cvars semantic attribute.
  operationalizationDriver(t,lmexp1Evaluation).
  create the application GUI with cvars and
         the following callback for the compute
         button:
          proc() {
            if is possible to create the variable
               binding
               set t's bind to this binding.
               c(t) with c the computing procedure
                    of t's evaluator.
               display t's resul.
          }
}
```

Fig. 10 (a) Structure of an OADDS processor. Predefined components are shadowed, (b) pseudo-code for the operationalization driver, (c) Pseudo-code of the glue program for the EXPML(1)'s processor.

In Fig. 10c the glue program of the processor for the calculation of the EXPML(1)'s expressions is outlined.

5 Related Work

PADDS has recently been proposed in [25]. The origin of OADDS is dated to [23]. In [20] a first attempt to introduce semantic modularity mechanisms in the model is given. Preliminary versions of the model, closer to that described in this paper, are outlined in [22][24]. In [21] an updated survey on ADDS is presented, which include short descriptions of PADDS and OADDS.

Modern schema languages [13] are equipped with modularization facilities for the incremental definition of XML-based descriptive markup languages. With PADDS we don't attempt to provide an alternative to these proposals, but we abstract some useful patterns for the incremental definition of these languages. In addition, these patterns can be subsequently implemented by using a specific grammatical formalism. We have chosen a DTD-based implementation because DTDs are easy-to-use by developers and easy-to-understand by domain-experts. This choice doesn't discard alternative implementations that use more sophisticated schema languages. In fact, our implementation is based on proposals for modularization of DTDs like that used in the modularization of XHTML [26]. Moreover, our grammar production specifications resemble the mechanisms underlying *architectural forms* in HyTime [9].

There are many proposals for the processing of descriptive markup languages. In their more basic form, they are focused on the intermediate representation of the marked documents produced by the parsers. From these, the *event-based* and the *tree-based* ones are the most widely used [2]. The particular implementations of our OADDS model can take advantage of these representations. Hence OADDS can be considered as regulating and structuring the domain-specific code that operates in these representations. This is facilitated by the wide availability of object-oriented frameworks that implement these representations [2] and by the ease of giving object-oriented implementations of OADDS (like that described in [22][25]).

[4] This program plays a role analogous to the `main` function provided with YACC specifications [10].

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

OADDS is inspired by the techniques for the construction of modular language processors. These techniques have been popularized inside the functional programming community, where the main approach is based on *monads* and *monads transformers* [14]. Similar ideas have also been applied in the object-oriented arena [6], where *mixins* [3] are used as an alternative to monad transformers. In turn, attribute grammars [12][18] are also amenable to including semantic modularity mechanisms, which are usually achieved by using some sort of attribute propagation patterns [11]. These proposals promote modularity by abstracting the context (e.g. the input and outputs of a processor) and the control (e.g. how the basic processing steps are chained). OADDS focuses on the evaluation of document trees. In this way it abstracts context in terms of tables of semantic attributes that can be incrementally extended. In addition, OADDS has a limited abstraction of control using pluggable invocation procedures. Finally, one of the main pragmatic limitations of OADDS is to obviate the static typing of the OADDS processors. While static typing of modular processors remains an open problem, approaches to static typing followed in [6][14] could be adapted to the OADDS context, as well as those behind the type-safe solutions of the so-called *expression problem* (i.e. definition of functions on trees that can be extended with new types of nodes) [27]. In addition, other types of static analysis that are not currently contemplated in OADDS could be incorporated into the model to facilitate the construction of processors (e.g. circularity tests in the context of attribute grammar-based specifications [18]).

6 Conclusions and Future Work

The practical use of DSMLs in the development of applications depends critically on being able to incrementally define and operationalize these DSMLs. This is a lesson learned during our experiences in the document-oriented construction of content-intensive (e.g. educational, hypermedia and knowledge-based) applications. Therefore, we have included mechanisms to deal with these matters in ADDS, our approach to the document-oriented development of this kind of applications.

DSMLs in ADDS are incrementally defined according to the PADDS technique, and their processors are incrementally built following the OADDS model. PADDS abstracts several basic mechanisms that enable the incremental definition of DSMLs by combining and extending linguistic modules. The technique also contemplates the resolution of conflicts between linguistic modules, and the production of different versions of the same DSML varying the names in the markup vocabulary. We have carried out a DTD-based implementation of the technique, because DTD simplicity was well suited to our usability requirements. Nevertheless, the technique can be applied to other schema languages in a straightforward manner.

The incremental definition of DSMLs is parallel at the operational level with the incremental construction of their processors. This construction is regulated with the OADDS model. Incremental operationalization in OADDS is enabled by extensible tables of semantic attributes, and by the explicit representation of the invocation inside the evaluators. We think that our solution achieves a good trade-off between the configurability of the final processors and the usability of the approach.

Currently we are exploring the use of *meta* DSMLs (i.e. DSMLs used to mark up documents describing OADDS processors). Thus we hope to overcome some of the limitations of the model regarding static analysis, because additional information required by typing and other static proofs (e.g. circularity tests) can be explicitly incorporated in the markup added to the documents describing the processors. As future work, we are considering experimenting with alternatives to PADDS and OADDS based on object-oriented attributed grammars [18].

References

[1] Aho, A., Sethi, R. and Ullman, J. D. (1986). “Compilers: Principles, Techniques and Tools”. Adisson-Wesley.
[2] Birbeck, M et al. (2001). “Professional XML 2nd Edition”. WROX Press.
[3] Bracha, G. (1990). “Mixin-based Inheritance”. ACM SIGPLAN Notices, 25(10),303-311.
[4] Clark,J. and Makoto,M (eds) (2001) “Relax NG Tutorial”. Oasis Specification.
[5] Coombs, J. H., Renear, A. H. and DeRose, S. J.(1987). “Markup Systems and the Future of Scholarly Text Processing”. Communications of the ACM 30(11),933-947.

Draft version. Citation: Sierra,JL; Navarro, A; Fernández-Manjón, B; Fernández-Valmayor, A. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices 40(12),28-37. 2005

[6] Duggan, D. (2000). "A Mixin-Based Semantic-Based Approach to Reusing Domain-Specific Programming Languages". ECOOP'2000. Cannes. France.

[7] Fernández-Valmayor, A., López Alonso, C., Sèrè A. and Fernández-Manjón,B. (1999). "Integrating an Interactive Learning Paradigm for Foreign Language Text Comprehension into a Flexible Hypermedia system". IFIP WG3.2-WG3.6 Conf. Univ. California Irvine, California, USA.

[8] Goldfarb, C. F. (1990).The SGML Handbook. Oxford University Press.

[9] International Standards Organization ISO. (1997). "Hypermedia/Time-based Structuring Language (HyTime) – 2d Edition". ISO/IEC 10744.

[10] Johnson,S.C. (1975)."YACC-yet Another Compiler-Compiler". CS Tech. Report 32. AT&T Bell Laboratories.

[11] Kastens,U. and Waite,W.M. (1994). "Modularity and Reusability in Attribute Grammars". Acta Informatica, 31(7), 601-627.

[12] Knuth,D.E. (1968). "Semantics of Context-free Languages". Mathematical Systems Theory, 2(2), 127-145.

[13] Lee,D. and Chu,W.W. (2000). "Comparative Analysis of Six XML Schema Languages". ACM SIGMOD Record. 29(3).

[14] Liang,S., Hudak,P. and Jones,M.P. (1995). "Monad Transformers and Modular Interpreters". 22nd ACM Symposium on Principles of Programming Languages. San Francisco. CA.

[15] Makoto,M., Lee,D. and Mani,M. (2001). "Taxonomy of XML Schema Languages Using Formal Language Theory". Extreme Markup Languages 2001. Montreal, Canada.

[16] Megginson,D., Leventhal, M. and Ducharme,R. (1998). "Structuring XML Documents". Prentice-Hall.

[17] Navarro, A., Fernández-Valmayor, A., Fernández-Manjón, B. and Sierra, J.L. (2004). "Conceptualization prototyping and process of hypermedia applications". International Journal of Software Engineering and Knowledge Engineering, 14(6), 565-602.

[18] Pakki, J. (1995). "Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation". ACM Computing Surveys 27(2),196-255.

[19] Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A. and Navarro, A. (2004a). "A Document-Oriented Approach to the Development of Knowledge-Based Systems". LNAI 2040. Springer-Verlag.

[20] Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A. and Navarro, A. (2002). "An Extensible and Modular Processing Model for Document Trees". Extreme Markup Languages 2002. Montreal. Canada.

[21] Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A. and Navarro, A. (2005). "Document-oriented Software Construction based on Domain-Specific Markup Languages". ITCC'05. Las Vegas. USA.

[22] Sierra, J. L., Fernández-Valmayor, A., Fernández-Manjón, B. and Navarro, A. (2004b). "ADDS: A Document-Oriented Approach for Application Development". Journal of Universal Computer Science, 10(9), 1302-1324.

[23] Sierra, J. L., Fernández-Valmayor, A., Fernández-Manjón, B. and Navarro, A. (2001). "Operationalizing Application Descriptions with DTC: Building Applications with Generalized Markup Technologies". SEKE'01. Buenos Aires. Argentina.

[24] Sierra, J. L., Fernández-Valmayor, A., Fernández-Manjón, B. and Navarro, A. (2003). "Building Applications with Domain-Specific Markup Languages: A Systematic Approach to the Development of XML-based Software". ICWE 2003. Oviedo.

[25] Sierra, J.L. (2004c). "Hacia un Paradigma Documental de Desarrollo de Aplicaciones (*Torwards a Document-Oriented Paradigm for Application Development*)". Ph.D. Thesis (in Spanish). Univ. Complutense de Madrid. Madrid. Spain.

[26] W3C World Wide Web Consortium. Web site "www.w3.org/TR".

[27] Wadler,P et al. (1998). "The Expression Problem". Discussion on the Java Genericity Mailing List.