Formal Design Constraints*

Nils Klarlund[†]
AT&T Labs–Research, klarlund@research.att.com

Jari Koistinen[‡]
Hewlett-Packard Laboratories, jari@hpl.hp.com

Michael I. Schwartzbach
BRICS, University of Aarhus, mis@brics.dk

Keywords: software architecture, architectural style, software evolution, parse tree, design constraints, tree logic

Large software systems are often built on system platforms that support or enforce specific characteristics of the source code or actual design. These characteristics are either captured informally in design guideline documents or in specialized design and implementation languages.
In our view, both approaches are unsatisfactory. Informal descriptions do not allow automated analysis and lead to vague constraint descriptions. The language-based approach leads to different languages for different platforms or even for different versions of the same platform.
Our approach is to describe and name the constraints separately in a design constraint language called *CDL*, which is based on an extraordinarily concise logic of parse trees. Designs are then annotated with the names of the constraints they are supposed to satisfy. We discuss how the design constraint language is integrated into a design language environment. We exhibit industrial and experimental evidence that our choice of design constraint language allows us to formalize naturally and succinctly common design characteristics.

1. Introduction

Companies building large systems are commonly using their own system platforms. These platforms support the building of applications with specific characteristics. As an example, telecommunication companies build platforms supporting signaling, error recovery, flexible service execution, and other functionality needed in most telecommunications applications. The characteristics of a platform and an application architecture are expressed as a set of general *design constraints*. In order to use these platforms in efficient or even correct ways, the programmer must ensure that design constraints are satisfied.

Examples of design constraints are:

> Classes with persistent instances should inherit only from other classes with persistent instances and should not provide asynchronous operations.

> Classes abstracting hardware resources should provide asynchronous operations and respond through an event channel.

Such constraints can be discovered in many specifications and architectural documents. As an example, consider the *typed push model* of OMG Event Services [19]. It requires that the event consumer and the event supplier must have an interface of operations that do not return values and that do not have *out* or *inout* parameters. This is an example of a design constraint that can not be defined in the OMG interface definition language itself [18].

Design constraints are often described informally in design guideline documents. For larger systems, specialized languages may be developed that by their definition enforce the design rules. Both solutions have significant disadvantages. Informal descriptions are often incomplete and ambiguous. They are also difficult to check, since this can only be done through manual reviews.

Received March 05, 1997

*This article appeared in a preliminary version in Proceedings of Object-Oriented Programming Systems, Languages, and Applications, October 1996.

†This work was done while the author was with BRICS, University of Aarhus, Denmark.

‡This work was done while the author was with the Ericsson Telecommunication Systems Laboratories.

In contrast, specialized design languages are formal and allow automatic checking. Their disadvantage is that the language must evolve with the platform and the different applications. Unfortunately, proprietary platforms usually evolve rapidly—leading to situations where several versions of the platform are used simultaneously for different applications.

1.1 Contributions of this paper

In this paper, we propose a language, called *Category Description Language* (CDL), for the explicit description of architectural aspects of platforms and software applications in a clear and unambiguous way. By the *architecture* of a system, we mean the structure as expressed in terms of formal components such as object and object type declarations; method invocations; name, variable, and interface use; etc. Following Perry and Wolf [20], we consider an architecture to be specific to a particular system. In contrast, an *architectural style* captures commonalities among several architectures. In a sense, architectural styles define equivalence classes for architectures.

CDL allows us to formalize an architectural style as a set of design constraints. Each design constraint limits the way the concrete syntax of the design or implementation language can be used. A constraint enforces or disallows certain language constructs depending on the context.

CDL is based on newly developed decision procedures for logics on parse trees [11]. These logics, which are variations on first-order logic (predicate logic), can express quantification over nodes in a parse tree (and are thus of very high computational complexity). Consequently, informal constraints on parse tree can often be transliterated directly into CDL. In fact, we provide experimental evidence that design constraints found in practice can be expressed very concisely in CDL—while still being computable by the decision procedure. Thus CDL offers substantial advantages of ease-of-use and readability compared to e.g. attributed grammars or recursive functions on parse trees.

We show how CDL design constraints can be translated to attributed grammars from the output of the decision procedure. Any proposed design can then be held up against the attribute grammars to check that all constraints are satisfied.

CDL [13] originates from work on languages and platforms for large software systems at the telecommunications company Ericsson. CDL was evaluated at Ericsson for the development of distributed object-oriented systems, expressed in the DELOS [3] design language on a proprietary platform with a CORBA-like [18] architecture.

The architectural style for the proprietary platform was initially formulated informally by Ericsson designers working in telecommunications applications. (Such styles and the Delos language have been used for real applications.) There were on the order of 50 constraints, and approximately 85-95% of these were readily expressible in CDL (and a few, when formalized, were found to be unnecessary). Each constraint is specified in a couple of lines like the examples we present in this paper. Some of these constraints are described in detail in the evaluation study [15].

By incorporating CDL as a part of the Delos language we allow systems architects to tailor Delos according to their own general design considerations.

In this paper, we provide examples of constraints for designs expressed in both DELOS and OMG-IDL [18], although we have primarily been using CDL with DELOS.

1.2 An Introductory Example

Next, we will illustrate our approach with a small example. Assume we want to use DELOS to design applications for an OMG/CORBA compliant object request broker. DELOS provides functionality not available in OMG-IDL. Therefore appropriate constraints on DELOS descriptions must be imposed so that a design will satisfy the requirements of the CORBA platform. One such constraint is that CORBA interfaces cannot pass objects as operation arguments—only references to objects can be passed.

Design constraints are imposed in two phases:

- The constraints are named and described in CDL. A named set of constraints is called a *category*. A set of categories is called a *CDL style*, which constitutes our attempt at formulating the notion of an architectural constraint.
- The source code of the actual design is annotated with the appropriate category names.

The first phase is the responsibility of a systems architect. Let us for a moment assume that the architect has already defined a category corba, and let us take the role of a programmer, who in the second phase writes an example in DELOS source code. Specifically, we declare an object type called subscriber:

```
OBJECT TYPE corba : subscriber IS
  ATTRIBUTES
    id : INTEGER
  OPERATIONS
    addService(s : REFERENCE TO service);
    setStatus(st : status);
END
```

Here, the object type has one public attribute and two public operations. The first operation addService takes an argument that is a reference to an instance of the ob-

ject type service. The second operation setStatus takes an object of type status as argument and passes it as a value. Both status and service are object types that are declared elsewhere. In the first line of the example, we have annotated subscriber so as to enforce the corba category.

Let us now take the role of the system architect, who already in the first phase formalizes the constraints of the corba category. For simplicity, let us look only at the constraint that operation arguments of corba interfaces cannot denote object values. We formulate constraints in terms of parse trees for the concrete syntax. (Our constraints could also be formulated in terms of abstract syntax trees.) Each node in a parse tree corresponds to a syntax production. The non-terminal on the left-hand side of the production is called the *type* of the node. For example by "operation argument node," we refer to a node denoting an instance of the syntax production for operation arguments; technically, it has type Argument, if we assume that the grammar has a single non-terminal Argument defining operation arguments.

In parse trees, nodes corresponding to non-terminals have subnodes (children) corresponding to the right hand side of the production. Informally, our object value constraint is:

> *If* x *is an operation argument node in the syntax tree of a* corba *object type, then the node* y *below denoting its type cannot represent an object type.*

To use Delos appropriately for design on CORBA, the system architect has formalized this constraint in CDL as

```
CATEGORY corba FOR
   ObjectTypeSpecification IS
   ∀ x: Argument. root ◁ x ⇒ ∃ y: Type.
            x ◁ y ∧ ¬ OT(y);
END
```

Here, ObjectTypeSpecification, Argument, and Type are DELOS syntax [3] production names and OT(y) is a DELOS specific predicate—defined in CDL—that evaluates to true if y is a node that denotes an object type. The expression **root** denotes the node of type ObjectTypeSpecification to which the category applies, and x ◁ y holds when node x is the parent of node y. Thus, the category CORBA declaration allows the programmer to annotate object type specifications in the source code with "corba:". For each such annotated specification, the constraint states that if x and y are the subnodes of the object type specification node such that x is an argument node and that y is the node declaring the type of the argument x, then y must not represent a object type.

Does the object type subscriber satisfy the category corba? No, since the name status used in the operation setStatus above is an object type name, the constraint of the category is violated. Thus, the definition of subscriber would be rejected as a CORBA interface.

1.3 The Design Cycle

From the example above, we see that the use of CDL involves the following steps:

- *Architectural style design.*
  - A systems architect defines the appropriate set of categories and general constraints.
  - He (or she) uses the CDL decision procedure to verify that the style is internally consistent, e.g. that the constraints are consistent with the syntax and not mutually contradictory.
- *Application design.*
  - The application developer selects the appropriate architectural style for the applications and development platform that is used.
  - He (or she) annotates the design with the categories of the style.
  - He (or she) requests automatic checks of the design against the style to determine whether the design satisfies all categories.

Tool support The systems architect may use a design editor tool to define models in graphical and textual representations. In addition to defining classes and their interrelationship, the system architect may also use the tool to define interfaces in more detail. In particular, the architect may adorn each design entity with the categories that it should satisfy.

Figure 1 shows a screen from a DELOS tool. The tool is used to describe high-level decomposition of systems into multi-class modules (rectangles) and the interfaces used and provided by modules. Modules are annotated with names of CDL categories. DELOS category names are placed before the entity name, separated by a colon. The categories SE, SA, DefG, and SWILIB used in this example stem from an architectural style for telecommunications.

This is a high-level architectural style that defines how modules can be composed into larger systems and what categories of interfaces they provide and use. Such constraints forces designers and implementors to maintain high-level architectural invariants for systems that evolve for long-periods of time—sometimes decades. It also simplifies maintenance by helping developers identify where in an architecture particular kinds of functionality can be located.

In addition to the commands usually applicable in design editors, the proposed extension of this tool will provide two additional menu commands:

- Load architectural style.

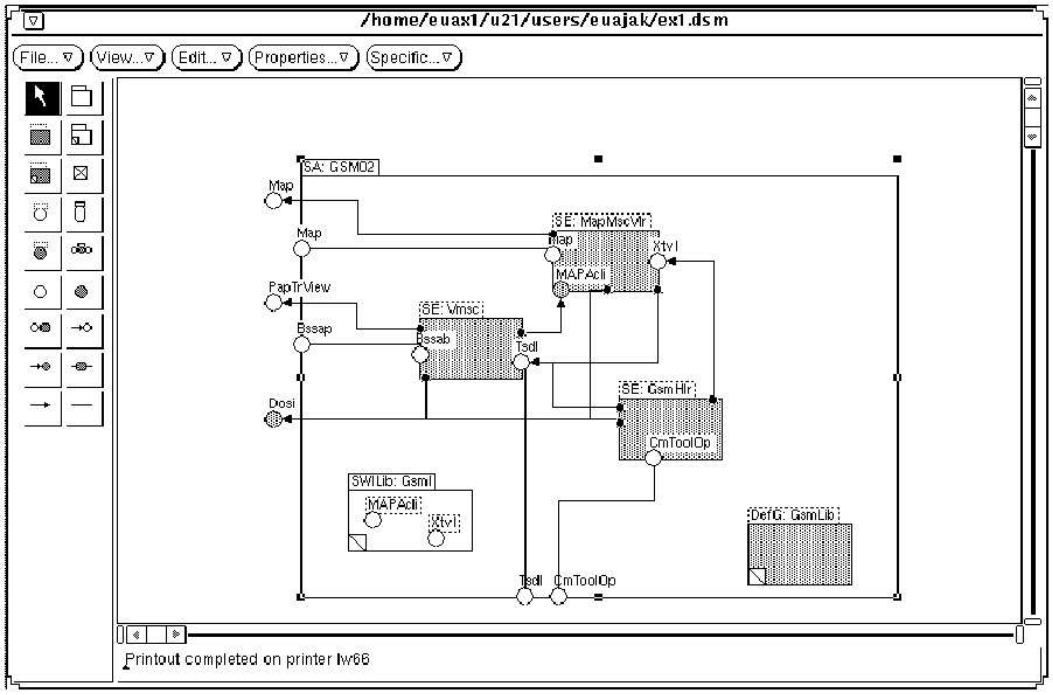


FIG. 1. An example of a tool supporting CDL

- Check for satisfaction.

The first menu alternative shown is used to load a style, i.e. a set of categories. The style loader is also a style compiler so that any syntactic errors in the style are detected and appropriate error messages are displayed. The style consistency checker might be part of this command or it could be a separate tool or menu alternative.

The check command is used when the architect or application designer wants to control that one particular CDL satisfies the current style. In a more sophisticated tool, the satisfaction check could be performed incrementally as the design is constructed.

We have already prototyped the most difficult aspects of integrating CDL support into a design tool, including:

- dynamic loading and checking of style with respect to a particular design; and
- consistency checking of a style.

We report in Section 4 on the technically most challenging aspect, namely to translate styles into a format usable by a constraint checker.

1.4 Related Work

A CDL description defines formally a set of architectural concepts that we call categories. The description does not represent any particular architecture, rather it defines what can be called a design style or an architectural style [20].

The book on software architectures by Shaw and Garlan [23] is an important advance in understanding software architectures. Shaw and Garlan formulate a very general notion of style:

> an architectural style defines a *vocabulary* of components and connector types, and a set of *constraints* on how they can be combined.

According to Shaw and Garlan, UNIX-like *pipe-and-filter* mechanisms constitute a style; similarly, object-oriented organization of software is a style. Tools like AESOP [6] can be used to describe such high-level styles and to support building systems according to them.

In contrast, CDL is used to define styles for systems designed or implemented using a particular source language. Thus, if we design with an object-oriented language, then CDL is used to define specialized styles within the boundaries of the object-oriented paradigm.

In [17], a comprehensive approach to specifying regularities in large software systems is outlined. These regularities are called *laws of the system*. They encompass both dynamic and static properties. Dynamic properties are checked or enforced during runtime.

Static properties are enforced during compilation time by viewing the system under development as a collection of objects. When an object is changed, rules formulated in a logic programming language are invoked. Thus as with our approach, the environment provides the enforcement of design constraints. The part of the

method in [17] that deals with source code constraints does not make explicit how syntactic constraints like the ones we consider should be modeled.

Meyers et al. [16] describe a language called CCEL for defining constraints on C++ programs. C++ programs can be statically checked to satisfy associated CCEL constraints. CCEL has C++ specific predicates—such as the is_friend predicate—and is therefore in principle only applicable to C++ programs. However, the ideas of CCEL and parts of its implementation can be used for other languages than C++. It is not clear if CCEL allows any consistency checks or how the checking of constraints is implemented. In CCEL, constraints are imposed within a certain scope such as a file, class, or member function. This is quite different from CDL, where syntactic elements are adorned with category names.

In recent years, design patterns [5] have become a popular way of describing solutions to common design and implementation problems. Usually, a design pattern describes a problem, outlines a solution, and has certain consequences. Gamma et al. [5] define design patterns as describing

> ...communicating objects and classes that are customized to solve a general design problem in a particular context.

Unfortunately, the general term of *design patterns* is commonly used for these specific kinds of general design solutions described by Gamma et al. and others. While a design pattern is intended to propose a solution in a limited context, a CDL style is intended to enforce certain design invariants on a complete system or a significant portion of a system. Moreover, design patterns are described informally by means of examples in some selected implementation language; a CDL style is a formal description of categories that can be assigned to design elements such as interface definitions, classes, and coarse-grained modules. In addition, a CDL style is defined relative to a particular formal design or programming language. Thus, according to the way the term *design pattern* is generally used, a CDL style is not a design pattern.

The Demeter Method [14] offers techniques for adapting and reusing object-oriented program code for new requirements and problems. It does so by providing higher level abstractions —*class dictionaries* and *propagation patterns*—for building programs.

A *class dictionary* is an abstract representation of classes and their relationships. Demeter presents different kinds of relationships on which different kinds of operations can be performed. Relationships are declared among parts or method calls.

A *propagation pattern* is a description of a program that traverses a class structure and adds computational descriptions. A Demeter tool takes a class dictionary and a propagation pattern as input and generates programming language code. Thus, propagation patterns and class dictionaries are together used to describe complete programs.

CDL does not by itself allow the specification of complete programs. Rather, CDL is used to specify and apply constraints of languages that are used to describe programs. Both Demeter and CDL use tree structures as a basis. CDL descriptions are currently tightly knit to the syntactical appearance of the language to which CDL is applied. Demeter uses class dictionaries as a more abstract representation of trees that allows propagation patterns to be independent of programming language syntax.

Possibly, CDL constraints could be applied to Demeter class dictionaries. Such as combination would make CDL constraints less syntax dependent, and it would allow the use of CDL for Demeter programs.

The Unified Modeling Language [2] (UML) introduces *stereotypes*. Stereotypes are similar to CDL categories in that they can be used to add semantic annotations to designs. However, UML does not provide a precise semantics for stereotypes. Neither does it provide analysis or checking mechanisms.

Superficially, our aims are similar to those of meta-object protocols (MOP), which also specialize object types [8]. However, a MOP specializes through programmed extensions of the behavior of object creations and message sends. In contrast, we never change the run-time semantics of objects. Our specializations are only imposed through more or less intricate syntactic restrictions.

CDL is an application of the FIDO programming language for expressing *regular* (finite-state recognizable) sets of labeled trees. FIDO is introduced in [11] as a high-level notation for the *Monadic Second-order Logic* (M2L) on finite trees, see [24]. While the M2L has been known to be decidable since the 1960s, it is only recently that practical implementations have been available, largely due to the adaptation of BDD techniques [4]. In [12], the translation techniques for M2L on strings of [7] are extended to trees along with combinatorial techniques and data structures for avoiding state space explosions. FIDO is also used in [9] for the behavioral description of distributed programs and their verification. M2L has also been applied to hardware verification [1].

In a technical sense, the most closely related work is in formal linguistics, where recent work has focused on constraint-based formalisms. Here the classical rewriting mechanisms of context-free grammars are augmented with formalized constraints on parse trees. Such constraints make it possible to avoid combinatorial explosions in grammars, for example those that occur when modeling agreement. James Rogers in his thesis [22] develops a theory of the use of formalisms based on M2L for expressing parse tree constraints. Our use

of constraints is similar in that it avoids multiplication of syntactic categories for parse trees of programs.

2. Applying Constraints to OMG-IDL

In this section, we introduce the category definition language in some more detail, and we provide some realistic examples. To illustrate that CDL is generally applicable, we have here chosen to use OMG-IDL as the specification language instead of DELOS. Thus, we will use CDL to define design constraints for IDL specifications. To allow category annotations in IDL definitions we have extended IDL slightly. We believe making an explicit extension of IDL is preferred over so called structured comments.

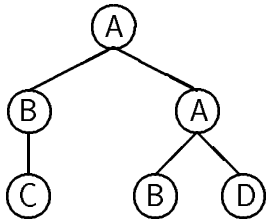
2.1 The Constraint Language

CDL is based on predicate logic where first-order terms denote nodes in a parse tree over which a formula is interpreted. The logical connectives $\wedge$, $\vee$, $\Rightarrow$, etc. have the same meaning as in conventional predicate logic. In addition, CDL provides operators that express relations among tree nodes as follows.

If x and y are variables denoting nodes, then the formula $x \leq y$ states that y is a descendant of x, i.e. it is in the subtree rooted by x; whereas the formula $x \lhd y$ states that y is a direct descendant of x. Furthermore, if x is a node, then $x.i$ is its i'th child in the syntax tree, counted from left to right. The quantifiers $\forall$ and $\exists$ range over nodes in the syntax tree, possibly restricted to a subset of the non-terminals. Finally, the predicate x=t holds when the node x is labeled with the terminal symbol t.

The example below shows a simple grammar and a possible syntax tree where each node is labeled with its type, a non-terminal symbol.

A ::= BA | BC | BD
B ::= id | C
C ::= id
D ::= id



Consider now these three formulas:

$\forall$ x: A. $\exists$ y: B. $x \lhd y$
$\neg \exists$ x: A. $\exists$ y: B. $\exists$ z: C. $(x \leq y) \wedge (x \leq z)$
$\forall$ x: C. $\exists$ y: D. $x \leq y$

The first formula states that for all nodes x of type A there exists a direct child of type B; for our grammar this formula is trivially valid, since it holds for *all* syntax trees. The second formula states that no subtree with a root of type A may contain both a node of type C and one of type B; this formula holds for only *some* syntax trees. The third formula states that any node of type C must have a child of type D; this is an absurdity that holds for *no* syntax tree.

A *category* consists of a named set of constraints and a designation of the language construct to which it applies. The following category is named O, and it is applicable to nodes of type A:

CATEGORY O **FOR** A **IS**
  $\exists$ y: B. **root** $\lhd$ y;
  $\exists$ z: C. **root** $\leq$ z;
END

This particular category states that every node of type A annotated with the category name O must have a direct child of type B and some subnode of type C. The name **root** denotes the node of type A to which we apply the category.

The category concept is a grouping and naming mechanism that maps into tree logic expressions. The first constraint of the category above can be mapped to the formula

$\forall$ root: A. O(root) $\Rightarrow \exists$ y: B.root $\lhd$ y

In addition to grouping of contraints, categories bind names to these groups. These names are essential, since they are used in the decoration of parse trees.

2.2 A Subset of OMG-IDL

We need a syntax for IDL, and we need to extend the syntax in order to allow category annotations in IDL specifications, specifically for interface definitions. Below, we have included parts of the OMG-IDL syntax [18] with our extensions underlined.

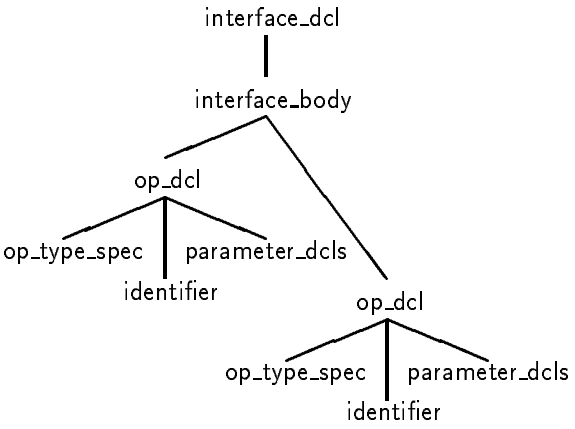
```
<specification>      ::  <definition>*
<definition>         ::= <type_dcl> ";"
                       | <const_dcl> ";"
                       | <except_dcl> ";"
                       | <interface> ";"
                       | <module> ";"

<interface>          ::= <interface_dcl>

<interface_dcl>      ::= <interface_header>
                         "{" <interface_body> "}"
<interface_header>   ::= "interface" <identifier>
                         [<category_dcls>*]
                         [<inheritance_spec>]

<interface_body>     ::= <export>*

<export>             ::= <type_dcl> ";"
                       | <const_dcl> ";"
                       | <except_dcl> ";"
                       | <attr_dcl> ";"
                       | <op_dcl> ";"

<const_dcl>          ::= "const" <const_type>
                         <identifier> "=" <const_exp>

<attr_dcl>           ::= [readonly_dcl] "attribute"
                         <param_type_spec>
```

```
                        <simple_declarator>
                        "," <simple_declarator>*

<readonly_dcl>      ::= "readonly"

<op_dcl>            ::= [<op_attribute>]
                        <op_type_spec> <identifier>
                        <parameter_dcls>
                        [<raises_expr>]
                        [<context_expr>]

<op_attribute>      ::= <oneway_dcl>

<oneway_dcl>        ::= "oneway"

<category_dcls>     ::= "!" <identifier>
                        {"," <identifier>}* "!"

<inheritance_spec> ::= ":" <identifier>
                        {"," <identifier> }*
```

The following is a simple OMG-IDL interface:

```
interface node !catName! {
   void addChild(in node c);
   void setParent(in node p);
};
```

Note that the category annotation (!catName!) is one of our slight extensions to IDL. Below, we show a partial syntax tree, where nodes in the tree are labeled by the corresponding node types:



The following subsections provides some examples of possible design constraints for IDL interfaces. Our examples are limited to a few categories although, in our experience, a realistic style consists of between 5 and 20 categories with each category containing up to 10 separate constraints.

2.3 Peer-to-Peer Interfaces

Our first example will demonstrate how we formally can introduce for a peer-to-peer communication model between objects specified in IDL. By annotating object interface definitions with the *peer* category, we can ensure that the communication adheres to certain fundamental principles. In addition, the annotations allows us to more easily understand how individual objects communicate on a higher level of abstraction.

The communication model we want to introduce is derived from the standardized Remote Operation Specification (ROS) [21] model. In the ROS model, we may have two peer interfaces exchanging asynchronous messages, thus we have no pure client server model. We use OMG-IDL one-way operations to denote asynchronous message passing. Interfaces that represent a peer are expected to be annotated with the peer category, which informally expresses:

> *All nodes* x *that are subnodes of an interface declaration node and represent operation declarations should have a subnode that is a one-way declaration.*

The underlying constraint is that peer interfaces should only provide operations with asynchronous semantics. IDL one-ways declarations imply that operations have asynchronous semantics. Therefore, we require that all operations of peer interfaces are one-way operations. Formally we define this constrains as:

CATEGORY peer **FOR** interface_dcl **IS**
   $\forall$ x: op_dcl.**root** $\leq$ x $\Rightarrow$
         $\exists$ y: oneway_dcl. x $\triangleleft$ y;
END

We also want to add that an interface clustering a set of one-way operations should not reveal any concrete state. Exposing concrete state is avoided by not having publicly available attributes. Some (abstract) state information can still be revealed through operation calls. This constraint leads us to refine the peer category as follows:

CATEGORY peer **FOR** interface_dcl **IS**
   $\forall$ x: op_dcl.**root** $\leq$ x $\Rightarrow$
         $\exists$ y: one-way_dcl. x $\triangleleft$ y;
   $\forall$ z: attr_dcl. $\neg$(**root** $\leq$ z);
END

This constraint disallows attributes in interfaces of category peer by prohibiting any subnodes of type attr_dcl.

The following is an example of two collaborating peer interfaces, one representing a player resource for a telephony application and the other representing a player resource client which we call the controller.

```
interface controller !peer! {
   one-way void done();
   one-way void error();
};

interface player !peer! {
   one-way void play();
   one-way void rewind();
   one-way void stop();
   one-way void pause();
```

};

A controller may start, stop, rewind, or pause the player without waiting for the previous operation to terminate. When an operation such as play terminates, it calls the done operation on the controller. For simplicity, we have omitted any arguments the operations may convey in this example.

The example illustrates the type of problems where peer interfaces and their characteristics are applicable. In particular, the asynchronous semantics and encapsulation of state is of vital importance. If attributes or non-one-way operations are introduced, then the interface would no longer satisfy the constraints of the peer category.

By enforcing certain constraints on peer interfaces, we are able to ensure the characteristics of a peer-to-peer communication model. For instance, since we are guaranteed that the communication between peers is asynchronous, we need not worry about peers mutually blocking while calling each other. Furthermore, a particular implementation mechanism—that supports ROS effectively—can be used for this particular communication link.

By introducing explicit categories, we have also extended our design language with the peer concept and consequently included explicit design level support for the ROS communication model.

2.4 Services and Resources

In this example, we introduce several categories. The resulting CDL style captures a systems architecture that allows telephony services and resources to be more easily introduced and changed. This is achieved by clearly separating the user services from the resources. Services and resources communicate in a way that allows a loose coupling and an asynchronous communication model. In addition, the constraints allow us to use implementation mechanisms well-suited for these categories of objects. These mechanisms support dynamic run-time installation and upgrade.

Observe that IDL does not fully support the needs of software design. Therefore, our example is limited to what can be expressed by IDL specifications. Other languages, such as DELOS, allows the expression of a more complete design model.

For the purpose of this style we have identified the following architectural concepts: service, resource, plain, and factory.

A *service* object would represent a user level service, such as call, voice mail, etc. Service objects can be mapped to specific mechanisms in the platform on which it is implemented. These mechanisms simplify upgrade and change of service in run-time. Service objects use a specific model for communication with resources and other services.

The *resource* objects model resources that are shared among multiple instances of the same or different services. Examples of such are speech recognition, tone sender, secondary storage, etc.

A *plain* object is neither considered a service nor a resource from an architectural point of view. Rather it is an object that is used to implement a service or resource abstractions without itself being one from the high-level architecture perspective. In addition, plain objects can be implemented using standard language mechanisms without any special platform support.

Finally, we want to adopt the object *factory* concept in order to decouple the usage and creation of objects.

CDL allows us to specify formally what characterizes the interfaces that correspond to each of these concepts. Each concept will be represented by one CDL category.

Objects of category service should have a single one-way operation called execute. This operation represents the main flow of the service. Having a convention for the name also simplifies service management since all services have the same static interface signature.

We define the service category, where we use ∃! x: to denote that "there exists exactly one value of x such that . . .," as:

CATEGORY service **FOR** interface_dcl **IS**
  ∃! x: op_dcl.∃ y: identifier.∃ z: one-way_dcl.
    root ≤ x ◁ y ∧ y="execute" ∧ x ◁ z;
  ∀ z: attr_dcl. ¬**root** ≤ z;
END

Because of the uniform and simple interface of service objects a code generator can map it on an execution mechanism that allows dynamic upgrade of services. In addition, we architecturally enforce each service to be defined as a separate object which will will enable fine-grained reuse and upgrade. It is these kinds of considerations that lead architects to identify *service* as an design concept.

In our model, resources typically provide many one-way operations commonly called by services. Services are notified about the results through a separate event service. It could also provide a concrete state, but only for reads. Resource interfaces should only inherit from other resource interfaces. We define the resource category below.

CATEGORY resource **FOR** interface_dcl
IS
  ∀ x: op_dcl. **root** ≤ x ⇒
      ∃ y: one-way_dcl. x ◁ y;
  ∀ x: inheritance_spec.∀ y: identifier.
      root ≤ x ◁ y ⇒ resource(y);
  ∀ x: attr_dcl. **root** ≤ x ⇒
      ∃ y: readonly_dcl. x ◁ y;
END

The communication model for services and resources allows them to be loosely coupled. This means resources do not know about the services that call them and services are not blocked calling resources. Since the model is essential for overall system qualities it is captured as concepts in the architectural style. Figure 2 illustrates the relationships between these concepts. Solid arrows represents operation calls and dotted arrows event sends.

Plain interfaces represent auxiliary objects rather than abstractions that are important from an high-level application architecture point of view. Plain interfaces should be restricted so as not to provide one-way operations. This restriction limits the degree of asynchronous message passing, thereby simplifying debugging. Also, plain interfaces should be restricted so as to only inherit from other plain interfaces

CATEGORY plain **FOR** interface_dcl **IS**
  $\neg\exists$ x: one-way_dcl. **root** $\leq$ x;
  $\forall$ x: inheritance_spec.$\forall$ y: identifier.
    root $\leq$ x $\triangleleft$ y $\Rightarrow$ plain(y);
END

Finally, for interfaces representing object factories, we wish to enforce a naming convention and make the factory concept more explicit than an informal convention.

CATEGORY factory **FOR** interface_dcl
IS
  $\exists!$ x: op_dcl.$\exists$ y: identifier.
    x $\triangleleft$ y $\wedge$ y="getObj";
END

Below, we outline a small example where categories have been used to annotate an OMG-IDL interfaces. The annotations enable us to more readily identify the architectural role of each interface. Since categories are described formally, we can automatically ensure that the interfaces satisfy the associated constraints.

```
interface wakeUpCall !service! {
  void setTime(time t);
  one-way void execute() raises (not_avail);
};

interface toneSender !resource! {
  ...
}

interface lineInterfaceCtl !resource! {
  ...
};

interface licFactory !factory! {
  lineInterfaceCtl getObj(licnr ln);
  ...
};
```

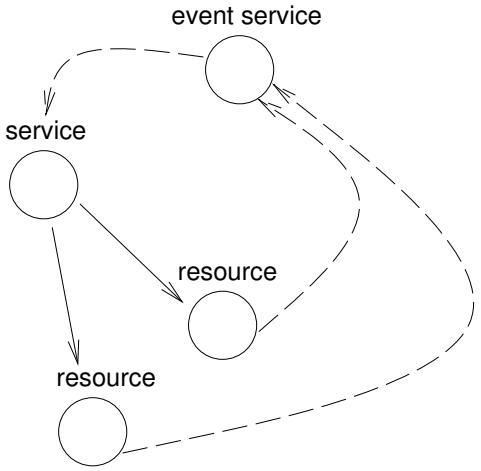


FIG. 2. Services call resources that sends events

Consistency The style given above is obviously consistent, since we have just sketched an example of a legal design. However, suppose that we stupidly added the following constraint:

$\forall$ x: interface_dcl. resource(x) $\wedge$ plain(x)

which states that all interfaces must be both of category plain and resource. Unfortunately, the added constraint has the implication that *no* design can possibly be legal—a fact a consistency check would discover. Without such a check, we could perhaps enter a costly and frustrating cycle of trying to complete a design. For a large and complicated design style developed simultaneously by several architects, the risk of inconsistency is very real.

The following section will discuss the theoretical basis for our constraint language and the possibilities it provides.

3. Theoretical Basis

The CDL formalism is based on the Monadic Second-order Logic on finite binary trees. In M2L, the universe of discourse is the nodes of a binary tree. There are basic predicates for relating the positions of nodes. Second-order terms are *monadic* relations, i.e. *sets* of nodes. There are the usual logical connectives and both first- and second-order quantifiers.

Each formula in M2L denotes a set of trees: those for which the formula holds. These sets coincide with regular tree sets. The fundamental difference between the two representations is that a formula may be non-elementary more succinct than the corresponding automaton. Thus an extremely complicated automaton may be described by a brief and elegant formula.

CDL is essentially the first-order fragment of M2L; however, the full logic is needed to correctly model the underlying syntax trees of a given grammar. Since reg-

ular tree sets need not be sufficient, we also extend the logic with externally computed *unary* predicates. In the translation these are represented simply as free second-order variables.

For the satisfaction problem, a CDL formula is translated into the underlying tree automaton, which is essentially a simplistic attribute grammar, as discussed in the following section. It is now a straightforward task to see if a given syntax tree is accepted.

For the consistency problem, we must decide if some constraints C_1, C_2, ..., C_n are mutually contradictory. Thus we construct the automaton for the combined formula $C_1 \wedge C_2 \wedge \ldots \wedge C_n$. The constraints are now consistent if this automaton accepts any trees at all, which can be determined through a simple depth-first search.

The M2L formalism is known to be a very ambitious compromise between expressibility and decidability. Almost any extension leads to an undecidable logic. Thus, we can rest assured that CDL is as strong as it can possibly be for expressing regularity. For example, regular expressions with negation and conjunction operators can be translated into M2L with only a linear increase in size.

CDL cannot express all properties that one might want to use as design constraints. For example, we cannot express that two positions in the parse tree contains the same, but unknown, string. This deficiency could be remedied by embedding CDL into a more general computing notation.

4. Implementation

The translation of CDL into tree automata by means of the FIDO compiler and MONA decision procedure [11] is reasonably quick. For example, each formula in Section 2.4 is translated in about 15 seconds (on a Sparc 1000). The consistency check is potentially costly, since it considers several formulas at the same time; for the style in Section 2.4, it lasts several minutes. New versions of these tools are under development and they will run at least an order of magnitude faster due to an improved BDD package [10].

We illustrate with an example how tree automata are represented as attribute grammars. Recall this simple grammar:

A ::= BA | BC | BD
B ::= id | C
C ::= id
D ::= id

Consider the constraint that for every node of type A there must below be a node of type D for which the external predicate P holds:

$\forall$ x: A. $\exists$ y: D. x $\leq$ y $\wedge$ P(y)

We now describe in detail the attribute grammar that is generated. All attributes are synthesized, and the attribute values are simply integers. A rule looks like:

$T : [m_1,\ldots,m_k] \mapsto n$ **if** $P_1,\neg P_2,\ldots$

The meaning is: if we are at a given node of type T and the i'th subnode has synthesized the value m_i and the external predicate P_1 is true, P_2 is false, etc., then we synthesize the value n. A given syntax tree is accepted if a bottom-up run yields an accepting attribute value. The full attribute grammar is as follows:

attributes 0,1,2
accepting 0,1

A : [0,0] $\mapsto$ 2
A : [0,1] $\mapsto$ 1
A : [0,2] $\mapsto$ 2
B : [] $\mapsto$ 0
B : [0] $\mapsto$ 0
C : [] $\mapsto$ 0
D : [] $\mapsto$ 0 **if** $\neg$ P
D : [] $\mapsto$ 1 **if** P

The claimed succinctness of formulas is not apparent from this trivial example. However, the full resource category generates 9 attribute values and several dozen intricate rules; in comparison, the resource constraints are rather intuitive, completely modular, and easy to maintain.

The generated attribute grammars are guaranteed to be *minimal*, since they are generated from uniquely minimized tree automata. Thus, the design architect need not be concerned with efficiency of the particular phrasing of a given constraint.

In the new version of CDL under development, the compiled automata are actually more complicated than just explained. They are factorized according to the principles explained in [12] so that the automata correspond to grammars that have both inherited and synthesized attributes.

We have not yet completed the integration with a design tool for DELOS or any other language. Note, however, that attribute grammars corresponding to design constraints can be expected to be as simple as above. Thus existing programming environments can easily be extended to deal very efficiently with design constraint checking.

5. Concluding remarks

Our proposed design constraint language CDL arose from our experiences in developing design languages and tools for object-oriented systems. Our approach to architectural styles allows a formal treatment of design constraints without forcing them to be built into the design languages.

The main practical advantages of CDL are:

- we identify and formalize important architectural concepts in named categories; and
- a design can be annotated with category names, and we can automatically verify that a design satisfies the stated constraints.

Our industrial experience and the examples presented here make us confident that the expressive power of CDL is well-balanced, since it allows interesting constraints to be expressed concisely and precisely, while allowing automated support by a decision procedure.

In this paper we have applied CDL to textual languages. There are, however, no hindrance to the application of CDL to graphical languages with an underlying tree structure. The only—general—prerequisite is that the language has a well-defined syntax and that syntactic entities can be annotated with categories.

Acknowledgments

We wish to thank anonymous reviewers for helpful comments on earlier versions of this paper.

References

1. D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer aided verification : 7th International Conference, CAV '95, LNCS 939*, 1995.
2. Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Unified Modeling Language Semantics. Rational Software Corporation, Version 1.0, January 1997. Also available through http://www.rational.com.
3. Martin Boström, Eui-Suk Chung, Jari Koistinen, and Mats Svensson. Delos 2.2 language description. Ellemtel Telecommunication Systems Laboratories. December 1995.
4. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, August 1986.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
6. David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT*, (12), December 1994.
7. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996. Also available through http://www.brics.aau.dk/~klarlund.
8. Gregor Kiczales, Jim des Rivires, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
9. N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. Technical Report RS-95-53, BRICS, 1995. To appear in Proceedings of PODC '96.
10. Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. Technical Report RS-96-05, BRICS, 1996. Submitted.
11. Nils Klarlund and Michael I. Schwartzbach. A Domain-Specific Language for Regular Sets of Strings and Trees. *USENIX Conference on Domain Specific Languages*. October 1997.
12. Nils Klarlund and Michael I. Schwartzbach. Efficient compilation of a high-level symbolic language into tree automata. In preparation, 1997.
13. Jari Koistinen. The Delos category definition language: Definition and rational. Ellemtel Telecommunication Systems Laboratories. July 1995.
14. Karl J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, 1996.
15. Johan Liseborn. The Delos Category Definition Language: A user's first impression. Ellemtel Telecommunication Systems Laboratories.
16. Scott Meyer, Carolyn K Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. In *First Workshop on Principles and Practice of Constraint Programming. Brown University Computer Science Technical Report CS-93-12*, April 1993.
17. N. L. Minsky. Law-governed regularities in object systems; part 1: Principles. To be published in Theory and Practice of Object Systems (TOPAS)). Also, available through http://www.cs.rutgers.edu/~minsky/pubs.html., 199?
18. Object Management Group. *The Common Object Request Broker: architecture and specification*, July 1995. revision 2.0.
19. Object Management Group. *Common Object Services Specification, Volume 1*, March 1994. OMG doc nr. 94-1-1, Revision 1.0.
20. Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes*, 17(4), October 1992.
21. ITU-T recommendation X.880. Remote operations: Concept, model, and notation. July 1994.
22. James Rogers. *Studies in the logic of trees with applications to grammar formalisms*. PhD thesis, University of Delaware, 1994.
23. Mary Shaw and David Garlan. Software Architectures: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
24. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.