Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

An extensible language for the generation of parallel data manipulation and control packages

H.R. Walters, J.F.Th. Kamperman and T.B.Dinesh

Computer Science/Department of Software Technology

CS-R9575 1995

Report CS-R9575
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

An Extensible Language for the Generation of Parallel Data Manipulation and Control Packages

H.R.Walters (pum@cwi.nl)
J.F.Th.Kamperman (jasper@cwi.nl)
T.B.Dinesh (dinesh@cwi.nl)

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

The design and implementation of the language fSDL (full Structure Definition Language) is discussed. In fSDL, complex user-defined data types such as lists, tables, trees, and graphs can be constructed from a tiny set of primitives. Beyond mere structure definitions (also offered by previously existing tools) high-level functionality on these data types can be specified. In the COMPARE (ESPRIT) project, the C code generated from an fSDL specification will be used by compiler-components running in parallel on a common data pool.

fSDL is first translated into a sublanguage, *flat* fSDL, from which the actual C code is produced. *Flat* fSDL is a convenient interface for cooperation with other compiler generation tools. There is a formal relation between the input fSDL and the resulting flat form.

CR Subject Classification (1991): D.2.2 `[Tools and Techniques]`: Modules and interfaces; D.2.7 `[Distribution and Maintenance]`: Extensibility; D.3.3 `[Language Constructs and Features]`: Data types and structures; D.3.4 `[Processors]`: Compilers.

AMS Subject Classification (1991): 68N20: Compilers and generators.

Keywords & Phrases: Compiler Construction, Program Generation, Interface Definition Language, Abstract Data Types, Data Description Language, C.

Note: This report appeared as a paper in *Proceedings of the Poster Session of Compiler Construction '94*, ed. P. Fritzson, technical report LiTH-IDA-R-94-11, University of Linköping. Partial support received from the European Communities under ESPRIT projects 5399 (Compiler Generation for Parallel Machines – COMPARE) and 2177 (Generation of Interactive Programming Environments II – GIPE II). This paper describes joint work of the COMPARE consortium. The members of the COMPARE Consortium are ACE Associated Computer Experts bv, GMD Forschungsstelle an der Universität Karlsruhe, Harlequin Limited, INRIA, STERIA, Stichting Mathematisch Centrum (CWI), and Universität des Saarlandes.

1. Introduction

Data structuring is an important technique which aids the development and maintenance of large software. Most programming languages offer some facility to define data structures,

albeit with limited capabilities for manipulation and organization. Most *object oriented* languages and systems are widely used since they allow organization and manipulation of data, albeit restricted to large grain data structures. In practice, general purpose languages enforce large grain data structuring while allowing minimal support for finer grain relationships between these data structures. A structure definition language, on the other hand, can provide maximum flexibility with respect to data structure definition, manipulation and organization. For such a structure definition language to be used in conjunction with a specific general purpose language (target language), the defined structures and their organization need to be suitably interpreted in the general purpose language.

fSDL is a language which allows flexible definitions and manipulations of data structures in terms of a calculus of domains. The calculus provides constructive and restrictive operators over domains for their construction and refinement. The process of construction and refinement of domains results in an implicit set of data structures, and domains serve as a definition of a fine grain view of these data structures. fSDL can thus be used to flexibly define a group of data structures and various views of them. When compared to object oriented languages, an fSDL domain could represent a group of *classes*, where arbitrary attributes of them are hidden.

The bare domain calculus is extended with *opaque* and *functor* domains, for the purpose of making the calculus freely extensible to a particular target language. Opaques free fSDL from language specific pre-defined types, whereas functors free fSDL from all pre-defined abstract data types (e.g., lists, graphs, sets, locks) as well as facilitate polymorphic library functionality for a given target language.

A higher level language like fSDL for defining data structures helps generate utilities, which might be application specific, at a target language level [Sno89, CIL89, JT89]. These may vary from type-checking functionality to generic readers, writers, memory management and debugging routines. In a multi user context, users can extend and/or restrict views of common data structures. Incremental modification of views or data structures by one user need not affect other users' views.

fSDL allows users to associate properties, which are simple labels, with the domains. These properties can be used by application specific code-generation to allow or disallow certain functionality (read, write, $\cdots$) in views and domains. Thus domains can be used to guarantee denial of access to components of a structure which can help assure data consistency.

In this paper, we describe the language fSDL through an example, and highlight its characteristic features. An fSDL specification is type-checked, functors expanded (analogous to template instantiation in C/C++) and flattened to a form with only constructive operators (the flat form or ffSDL). ffSDL can be used by other components such as view-analyzers and target language code generators (the fSDL compiler or fSDC). The target language considered in this paper is C [KR78]. In Section 3, the domain calculus is introduced and in Section 4 the flexible functor mechanism is explained. Support for parallelism, the fSDC and code generation are discussed in later sections.

fSDL is designed in the context of the CoSy[Con92] compiler model which provides a framework for flexibly combining and embedding compiler phases to facilitate the construction of parallelizing and optimizing compilers. The flat form, ffSDL, can be used by analysis

tools in conjunction with engine descriptions and configuration descriptions. The ffSDL specification can in turn be modified, e.g., appropriate locks are added to assure consistent use of data in a parallel programming context, and fed back to the flattener. A data manipulation and control package (fDMCP) specific to a target language is generated which can be used by the engines writers for accessing the pool of data.

1.1 Related work

In the Gandalf programming support environment [HN86] tools share a representation based on the abstract syntax of the language being supported, and all the tools are written in a single language. A similar approach is taken in [Gro90, CIL89].

IDL [Sno89] and NewGen [JT89] generalize these ideas in the Gandalf project, so that tools could potentially be built in different languages. This generalization has resulted in identifying a set of pre-defined types as "internal" types, while others are "external" types. The result of such a distinction, and the lack of capabilities similar to functors in fSDL hinder building libraries that are polymorphic over the views of data structures. IDL communicates by channels which require interfaces to be defined for converting external types to internal types. IDL channels, in practice, promote large grain communication. fSDL aims at an open system configurations centered around a black-board, where data access requires permission but not cooperation of the submitting process. The refinement mechanisms provided by IDL, although conceptually similar to the restrictive operations of fSDL, are not written as a calculus which limits use of refinement and makes it hard to define a domain via a complex mix of restrictions.

2. An illustrative example

In this section, we will give a small example illustrating the use of fSDL in a C context. We will show how datastructures for representing the abstract syntax of a tiny programming language (Femto) are defined in fSDL. Then we will, by way of some extracts of a C program, show how these datastructures can be used to implement an interpreter for Femto.

2.1 An fSDL definition of Femto

Femto is a tiny language with a tiny context-free syntax (in BNF):

```
Exp  ::= Id | Int
     |   Exp "+" Exp
     |   Exp "-" Exp
Stat ::= begin Stat* end ";"
     |   Id ":=" Exp ";"
     |   write Exp ";"
     |   while Exp do Stat
```

where every statement can be interpreted as a program. Datastructures for implementing the abstract syntax of Femto are defined by an fSDL specification, which we will introduce here, divided into small parts.

Basic notions in fSDL are *domains*, *operators* and *fields*. Domains determine the visibility of operators and fields, analogous to the way classes determine accessibility of instance variables

of objects in object oriented languages. Operators are analogous to structs or records, because they hold values of a cartesian product of several domains (one domain for every field). Every alternative in the context-free syntax above corresponds to an operator. For example, the domain `Un` defines the unary operators `id` and `const` that hold a single identifier `Id` and a constant value `Int`, respectively.

```
domain Un : { id < name: Id > }
          + { const < val: Int > };
```

The domains `Id` and `Int` are defined as basic values in the underlying programming language by the following, so-called *opaque*, definitions:

```
opaque Id: decl[| typedef char* Id;
                  Id Id_create(char *s);
                  void Id_delete(Id);
                  Id Id_copy(Id); |];
opaque Int: decl[| typedef int Int;
                   Int Int_create(int i);
                   void Int_delete(Int);
                   Int Int_copy(Int); |];
```

Where `decl[| .. |]` contains the declaration of functionality in the underlying programming language. For brevity, we have omitted the implementation (to be specified with `impl[| .. |]`) of the declared functionality.

When several operators have the same field, it is advantageous to declare these fields only once. Thus the fSDL part

```
domain BinOp : {plus,minus};
domain BinFld : <right:Exp, left:Exp>;
domain Bin : BinOp + BinFld;
```

declares a domain `Bin` with two binary operators, both having a `left` and a `right` field. Note that the final domain, `Bin`, is defined as the sum of the domains `BinOp` and `BinFld`. In fSDL, we have a calculus, called *domain calculus*, describing the construction of domains from other domains. Apart from the `+` operator for adding domains, we also have operators for restricting domains. In the domain Exp,

```
domain Exp : Un + (Bin-<*>);
```

the operators in `Un` and `Bin` are known, but the fields of `Bin` are not accessible as shared fields, because '`left`' and '`right`' fields are meaningless for unary expressions. The last domain definition in the fSDL specification of the abstract syntax of Femto concerns Femto's statements:

```
domain Stat : {seq<stats:LIST(Stat)>,
               asgn<id:Id,exp:Exp>,
               while<cond:Exp,body:Stat>,
               ifst<cond:Exp,thenp:Stat,elsep:Stat>,
               write<exp:Exp>};
```

Note that the domain of the field `stats` in the operator `seq` is not an ordinary domain, but has a parameter. The domain `LIST(Stat)` is an instance of a parameterized domain expression,

called a *functor* in fSDL. The call of a functor leads to a domain with extended functionality, in the case of the `LIST` functor the functions `LIST_create`, `LIST_delete`, `LIST_hd`, `LIST_tl`, `LIST_cons` and `LIST_mapc`, for creating, deleting, getting the head of a list, getting the tail of a list, adding an element to the list, and mapping a function over a list, respectively.

2.2 Excerpts of parsing and evaluating FEMTO

We will now look at some typical examples of the use of code generated from the fSDL specification above. During parsing, the body of a typical creation function for a node corresponding to some binary expression (with operator '+' or '-') would be:

```
{ Exp exp;
  exp = Bin_create(opc);
  Bin__set_left(exp,left);
  Bin__set_right(exp,right);
  return(exp);
}
```

where the functions `Bin_create, Bin__set_left` and `Bin__set_right`, and unique integral values (opcodes) for all the operators have been generated from the fSDL specification. Note that the `left` and `right` fields can be set without knowing their actual operator code.

A bit more of the power of fSDL is shown in the function that evaluates an expression:

```
Int eval_Exp(Exp e)
{ switch(Exp_op(e))
    { case op_Exp_const:
        return Exp_const_get_val(e); break;
      case op_Exp_id:
        return(lookup(Exp_id_get_name(e))); break;
      Bin_case: return(eval_Bin(e)); break;
      default: printf("error in eval_Exp\n");
               return(0);
    };
}
```

where for the operators `const` and `id`, the value is computed directly, and for the binary operators, another function is called. The identifiers `op_Exp_const` and `op_Exp_id` are the generated operator codes, and the macro `Bin_case` contains cases for all operators in the domain `Bin`.

The expressivity of functors is illustrated by the following excerpt from the evaluation function for a sequence of statements, where the '`mapc`' function defined by the `LIST` functor is used to map the evaluation function over the sequence of statements.

```
void eval_Stat(Stat s)
{ switch(Stat_op(s)) {
    /* other cases left out */
    case op_Stat_seq:
      LIST_Stat_mapc(Stat_seq_get_stats(s),
                     eval_Stat); break;
    }; }
```

3. fSDL, domains and the domain calculus

3.1 fSDL and types

In many languages there is a very strict notion of 'type'. Variables and values have types and only assignments of the same type are legal. Often some implicit or explicit casts are available for conversion of the most obvious similar values.

In many object-oriented languages, this scheme is less strict: objects of a sub-class can (implicitly or explicitly) be cast to a super-class, and vice versa.

In fSDL there is even greater freedom: objects can generally be cast (implicitly) to any 'type', but fSDL focusses on *visibility* rather than type. That is, although some meaningless cast is technically allowed, the resulting object can not in any way be inspected, since all aspects are invisible. Visibility is checked either statically or, if this is impossible, dynamically, and provides security comparable to type-checking.

Technically, the type of a value (we are considering tree or graph structures, so a value is a single node) is an *operator*. In practice, however, the *domain* is more important. In a domain, some field in some operator can be defined, and thus become visible. Within that domain, the operator appears to have only certain fields, and access functionality through that domain is only defined for those fields.

3.2 Operators, Fields, Properties and Domains

An *operator* can be thought of as a C structure type. It has a name, and a number of fields, each with a name and a domain. Operator names are unique: each occurrence of the same name refers to the same operator. An operator does not 'belong' to a single domain; it may be defined in a number of domains, each of which defines a (restricted) view of the operator.

A *field* is one of the components of an operator. It has a name and a domain, and in an operator *instance* each field can hold (the reference to) a value of this domain (which is an instance of an operator defined in that domain). Fields are used to represent (directed) edges of trees and graphs.

Fields and operators can have certain properties. A *property* is an arbitrary identifier or quoted string. Properties control the fDMCP generation process and the functionality made available in the fDMCP, or the behavior of the fDMCP or other tools. Only a few properties (such as READONLY) are meaningful to the fSDC itself; others are accepted in fSDL specifications, but are passed on to the flat form, to be used by other tools than the fSDC.

A *domain* coincides with a set of operators, each with a list of named and typed fields. Possibly operators and fields are annotated with properties.

A domain is defined by a number of domain expressions. Each expression adds to the global definition of that domain. This aspect allows the local extension of a domain which was primarily defined elsewhere.

A domain expression can be an atom (e.g., a single operator or field), or two expressions combined with an operation such as + and !. In addition, short-hands can be used for common situations.

3.3 The Domain Calculus

In fSDL a choice is made to define specifications in terms of a calculus (rather than, perhaps, a solely constructive definition style such as found in, for example, [Gro90]).

There are several considerations.

- The calculus allows for constructive specification without the need for local completeness. That is, conservative extensions can be made without altering the global specification (or unrelated code).
- The concept of visibility is enhanced by restricted views on domains defined elsewhere. For example:

  ```
  domain Exp : Un + (Bin-<*>);
  ```

  This definition defines expressions with binary operations, of which the arguments need not be visible (in the underlying context). This can be expressed in an intuitively clear manner without untoward redundancy.

  Note, though, that restrictive operations are notational convenience only; every specification could be presented purely constructively.
- The fine granularity of the calculus allows for great expressibility. For example, a view can be defined which consists of a single field inside a single operator.

We will now present the domain calculus. Then we will briefly discuss a few aspects.

- Operators, fields and properties may occur as atomic domains. Examples: {`op`} is an operator; `<fld:dom>` is a field and `[prop]` is a property.
- The + operation indicates union. The defined domain contains the union of the left and right arguments of the + operation. As mentioned, domain definitions may be distributed. The over-all definition of a domain is the union (+) of all definition constituents.
- Juxtaposition indicates extension of the left argument. For example, {`op`}`<f:dom>` extends the operator `op` (which is declared implicitly) with the field `f` of domain `dom`. In this way operators are given fields and properties, and fields are given properties.
- A single domain name can be used as an atom. It indicates importation. The resulting domain contains all operators with properties and fields defined in the imported domain. Implementation code domains are not imported. This type of importation is what is commonly needed.
- The operation . indicates full importation. The left argument is a domain name; the right argument is an arbitrary domain expression or a *quantifier expression*. It acts as a filter, indicating precisely what should be inherited. The quantifiers are used to indicate classes of things. For example, `D.{*}` imports precisely all operators from domain `D`.
- The operation ! and − indicate restriction and subtraction, respectively. Both arguments are domain expressions. In addition, the right argument may contain quantifiers.

  Again the right argument acts as a filter; In the case of ! describing what is imported, discarding the rest, and in the case of − describing what should be discarded, importing the rest.

- Several short-hands are defined in terms of the calculus. For example, {a,b<f:D,g:D>} is short for {a} + {b}<f:D> + {b}<g:D>.
- Finally, there are a few special atoms, which will be discussed in following sections.

3.4 Private and Shared Fields

Ordinarily, a field is defined for a single operator, as in {an_op <a_fld: a_dom>}. Such a field is called a *private* field.

Often it is useful to specify a field which is to be shared by all operators in some domain. This is done by defining a *shared* field: it appears as an atomic field, not associated with any operator. Such a field becomes private to each operator in that domain.

Consider the following example:

```
domain BinOp : {plus,minus};
domain BinFld : <right:Exp,left:Exp>;
domain Bin : BinOp + BinFld;
```

Each operator in domain Bin has the two fields right and left of type EXP.

Remember that a domain may be defined in several distributed expressions. A shared field distributes over all operators in the entire definition, not just the clause in which it appears.

Ordinarily, shared fields are imported. That is, if a domain imports a domain with shared fields, the importing domain also has these shared fields, and any operators introduced in that domain will have those fields. To avoid importing shared fields, the import must be restricted.

Apart from importation, the distinction between private and shared fields is significant in the access functionality generated for such fields.

- Private fields are specific to an operator, and hence their access functions is specific for the operator and the field.
- Shared fields are implemented as private fields and can be accessed as such. In addition, in the domain in which the field occurs as a shared field, it can be accessed in an 'anonymous' manner, without indicating the operator.

3.5 Quantifiers

The full import operation ., and the restrictive operations ! and − allow the indication of a filter, which may contain *quantifiers*. The *scope* of a quantifier is the left argument of the enclosing operation. The following quantifiers are defined in fSDL:

`*` (Only to be used with .). This indicates everything in the scope.

`{*}` This refers to all operators in the scope.

`{...}<*>` This refers to all private fields in the scope. The scope is limited to the operators shown; this may be a list of operator identifiers (such as in {a,b}<*>), in which case it refers to the private fields of those operators, or again a quantifier: {*}<*> refers to all operators with their private fields. There is no notation for private fields without operator, for these are not expressible entities in the calculus.

`<*>` This refers to all shared fields in the scope. Note that the fields are still visible as private fields; they just lose their 'shared-ness'.

For example, consider the domain `Expr` in Section 2:

```
domain Exp : Un + (Bin-<*>);
```

`[*]` All properties in the scope. As with fields this quantifier may be appended to expressions to limit its scope, such as in {*}`<*>[*]`, which refers to all operators with private fields and their properties.

3.6 Code domains

fSDL specifications are translated (by the fSDC) into packages with executable functionality. This functionality is then used to create, manipulate and destroy trees and graphs, which are structured according to the specification.

In addition to the default functionality generated by the fSDC, fSDL provides a mechanism with which code related to a specification can be inserted in that specification.

For this purpose fSDL provides *code-domains*. A code domain is an atomic domain expression which contains pieces of code (including function declarations and definitions). This code is passed along by the fSDC and is inserted into the generated fDMCP.

There are two types of code-domain: declaration code and implementation code. They appear as:

```
impl[| ... text ... |]
decl[| ... text ... |]
```

The 'decl' code relates to interface descriptions; the 'impl' code relates to implementation oriented code. In the context of C, declaration code ends up in in header files (`.h`), and implementation code in code files (`.c`).

Code domains are atoms which are left unchanged by juxtaposition (for example, one can not juxtapose a field to a code domain).

There are special quantifiers associated with code domains: `impl[*]` and `decl[*]` refer to all implementation and interface code domains, respectively.

Ordinary import does not pass on implementation code (which can usually not appear twice). It does pass on interface code, in order to propagate visibility of such code. Note that importing code from a domain does not extend that code to the importing domain. In Section 4 extendibility in general will be discussed.

Any code domain is imported by full import. In fact, ordinary import of some domain `D` is defined as `D.* - impl[*]`.

The full power of code domains can not be understood without looking at fSDL's extension mechanism (Section 4).

3.7 Support for parallelism

fSDL's domain calculus supports the fine-grain static detection of possible conflicts between parallel processes accessing the fDMCP. From the domains known to the processes, it is straightforward to infer which operators and fields could be involved in such a conflict.

In addition to the static analysis, dynamic assertions as known from Jade [LR91] or FX [LG88] can be used to determine which of these conflicts can really occur at runtime. Some

of these conflicts may be solved by protecting a field with a semaphore, or by replacing the logical field by several physical versions ('shadows').

Subsequently, (nested) virtual functors can be inserted in the specification in order to implement these solutions in a transparent way. Here, transparency means that the original processes still call the same functions as before. We give an example of such a transformation on page 13.

4. Functors

Functors are fSDL's mechanism for adding generic or non-standard functionality.

A *functor* is a parameterized fSDL specification, the parameters of which are again domains. A functor can be instantiated from most positions where a domain-name could be used. Each instantiation of a functor defines an addition to the enclosing, global specification; the body of the functor is expanded by replacing the formal parameters with the actual parameters, and by unambiguously renaming local names.

A functor definition looks like this:

```
functor F [Props] (D1, ..., Dn)
begin
  domain Aux1 : ... ; ...
end;
```

The body of a functor is an fSDL specification (a list of domain definitions), with the exception that it can not contain nested functor definitions. The parameters are domain names. In addition, a functor may use the implicit parameter `&sdom`, which refers to the domain in the definition of which the functor was called. The properties can be used to define *variants* of functors. We will not go into this here.

Each different functor instantiation is instantiated exactly once. A functor instantiation is regarded different if it has different actual arguments or properties, or if it uses context information and is instantiated in a different context.

Upon instantiation all local names are uniquely renamed. The instantiated result is a partial fSDL specification which is added to the global fSDL specification.

Some special attention must be given to parameter replacement in code domains. Remember that a code-domain is an expression like `impl[|.....|]` (or `decl|.....|]`). The body of this expression is arbitrary (C) code. This text may contain bar-quoted identifiers such as `|Id|`. These identifiers will be replaced by the fSDC for the actual (partial) identifier. For example, some code-domain may contain the following text:

```
|&sdom| parent = |Dom|_get_exit(x);
```

Suppose this occurs in a functor instantiation in the type of a field in a domain `Stat`, and suppose that the actual parameter corresponding to `Dom` happens to be `Proc`. Then, this piece of code is expanded to:

```
Stat parent = Proc_get_exit(x);
```

There are two kinds of functors: plain and virtual.

4.1 Plain Functors

In a plain functor there is one special domain called the primary domain, which, in the functor body, has the same name as the functor itself. After expansion (and renaming of local names), the functor instantiation is replaced by the generated name of the primary domain.

An instantiation to a plain functor may occur anywhere: as the type of a field, in an import clause, nested, as the argument to another functor instantiation or inside a code domain.

As an example we will now show how the `LIST` functor defines an alternative `create` function, and the function `mapc` that maps a function over all the elements in the list. This function was used in Section 2.

```
functor LIST(S)
begin
domain LIST: {nil,cons<hd:S,tl:LIST>}
  + [my_create]
  + decl[|
   |LIST| |LIST|_create();
   |S| |LIST|_hd(|LIST| This) ;
   |LIST| |LIST|_tl(|LIST| This) ;
   |LIST| |LIST|_cons(|LIST| This, |S|) ;
   void |LIST|_mapc(|LIST| This,
                    void (*mappedfun)(|S|));
  |]
  + impl[|
   |LIST| |LIST|_create()
     { return d_|LIST|_create(op_|LIST|_|nil|);}
   |S| |LIST|_hd(|LIST| This)
     { if (|LIST|_op(This) != op_|LIST|_|cons|)
         lerror(Head_of_empty_list);
       return |LIST|_|cons|_get_hd(This);}
   void |LIST|_mapc(|LIST| This, void (*mappedfun)(|S|))
     { |LIST| tmp;
       tmp = This;
       while (|LIST|_op(tmp) == op_|LIST|_|cons|)
         { mappedfun(|LIST|_|cons|_get_hd(tmp));
           tmp = |LIST|_|cons|_get_tl(tmp);
         }
     }
  |];
end;
```

The `LIST` functor defines a domain with two operators, `nil` and `cons`, where the latter has fields for the head and the tail of a non-empty list.

Next to the operators and fields, a number of functions `|LIST|_create`, `|LIST|_hd`, `|LIST|_tl`, `|LIST|_cons` and `|LIST|_mapc` are defined, where it should be noted that the actual name of any formal argument `x` will be substituted for the expression `|x|`.

Finally, it should be noted that the function `|LIST|_create` overrides the function that would be generated by default for the domain `|LIST|`. As a result of the property `[my_create]`, this function is renamed as `d_|LIST|_create`.

4.2 Virtual Functors

A virtual functor is 'virtual' in that a field with such a functor instantiation as type is not a proper field; it is removed from its context upon instantiation of the functor. It is up to the functor body to decide how the virtual field should be implemented, by adding one or several (hidden) fields instead of the virtual field, or in any other manner. A virtual functor does not define a primary domain.

A virtual functor may use additional implicit parameters defining its context: the domain (`&sdom`), operator (`&sop`) and field (`&sfld`) in whose definition the functor was instantiated. In addition the notation `&context` may be used to define something in exactly the same context: as a private field in an operator, or as a shared field. For example: `&context: <&sfld: something>` replaces the field containing the virtual functor instantiation by a field with the same name but type `something`, in a context.

A virtual functor may not be imported. Hence, it may only be used in the type of a field, or as a nested occurrence in some other functor instantiation, if, eventually, that argument only occurs as the type of a field, and never as a direct import.

Due to the limited space available, we will not present an example of virtual functors.

5. The compiler

The fSDL compiler, fSDC (short for 'full Structure Definition Compiler'), consists of two major phases, Flatten and Codegen. Flatten transforms the input specification into so-called *flat* form, a sublanguage of fSDL that contains only constructive definitions. From the flat form, Codegen produces the actual code for the fDMCP.

The flat form provides an interface for the cooperation with other compiler-generation tools. In a number of iterations, these tools may transform the *flat* fSDL specification by adding or deleting domains, operators or fields, and inserting functor applications. Flatten is used to produce a new flat form as result of every iteration.

In Section 5.1, we will describe the transformation into flat form, and we will give an example of a transformation of this flat form by an external tool. Then, in Section 5.2, we will discuss the generation of code from a flat fSDL specification.

5.1 The transformation into flat form

From a code-generation point of view, fSDL specifications have a number of awkward properties:

- An fSDL specification has a distributed character. Information pertaining to a certain domain may be specified in several places.
- Many domains are constructed from (by 'importing') other domains.
- Plain functors define new domains that depend on their arguments. These domains are only implicit in the original fSDL specification.

- There are many syntactic ways of defining an fDMCP that are semantically equivalent according to the domain calculus. For example, an identical fDMCP can be defined by using either constructive or restrictive fSDL operations.

The first phase of fSDC, Flatten, transforms an fSDL specification into its *flat* form, which serves a dual purpose. Firstly, it does not have the awkward problems mentioned above, thus rendering the generation of code a fairly straightforward process. Secondly, the flat form serves as an interface for other tools (external to fSDC) to insert their modifications and annotations.

The flat form consists of a list of functor definitions (for referencing by other tools operating on the flat form) and domain definitions. The domain definitions have the following characteristics:

- Every domain is defined exactly once.
- There are no imports of other domains. Other domains may only appear as the type of a field.
- There are no restrictive fSDL operations.
- There are no calls to functors.
- A definition contains a single list of operators, where every operator is specified with all of its private fields, visible in that domain.
- There is a single list of fields that have shared access through the domain.
- There are no multiple copies of `decl[||]` or `impl[||]` parts in a flat domain.

A flat fSDL specification is a fixpoint under application of Flatten; subsequent runs of Flatten will produce the same result.

Of course it is only useful to apply Flatten to a flat form which has been *modified* by tools external to fSDC . For example, in order to solve a conflict where two processes running in parallel that have access to the same field, it is possible to replace the field by a locked version:

```
domain D: {op<f:D>..} + .. ;
```

could be replaced by

```
domain D: {op<f:LOCK(D)>..} + ..;
```

which, after another application of fSDC , would be transformed into

```
opaque SEMA: ..;
domain D: {op<F_LOCK_f_D_sema:SEMA,
              F_LOCK_f_D_val:D>..}
          + decl[| D D_op_f_get(D); |]
          + impl[| D D_op_f_get(D d) {..}; |]
          ..;
```

where `F_LOCK_f_D_sema` is a field containing a semaphore (defined in the opaque domain SEMA), and `F_LOCK_f_D_val` is a field containing the actual value of the locked field. In code

domains, the original access functions for the field have been redefined in order to obtain the semaphore, before actually processing the field.

5.2 Code generation
From the flat form, code is generated to allow the actual use of the fDMCP. In section 5.2.1, we will discuss the choice of the implementation language of the fDMCP and in sections 5.2.2 and further, we describe what code is actually generated.

5.2.1 Choice of implementation language It is important to note that fSDL itself is implementation language indepedent, providing the implementation language provides sufficiently powerfull operations to implement the required functionality. This does not necessarily mean that a generated fDMCP can be addressed from engines written in different programming languages, but rather that the implementation language is not dictated by fSDL .

The current implementation of fSDC only generates C code. Earlier in the Compare project, C++ was chosen as the implementation language. From the point of view of fSDL this language has advantages as well as disadvantages, when compared to C.

The most notable advantage is the stronger type system of C++. In the C++ variant, every domain corresponds to a class. This domain-class only has conversion functions to domain-classes which have operators in common. The conversion functions do dynamic type-checking which can easily be compiled away if a conversion to a domain with a superset of operators is to be performed. Apart from stronger type-*checking*, the type system of C++ permits *overloading* of functions, which leads to significantly shorter names, especially for functions generated by functor applications.

Another advantage of C++ is the use of inheritance to specify functionality common to all or a large subset of all domains. In the C variant, this functionality has to be repeated many times with only minor differences.

An important disadvantage of C++ is a lack of power in it's concepts of inheritance. Especially the use of multiple inheritance, where a subclass inherits from several superclasses, can lead to many problems. A natural implementation of multiple inheriting fSDL domain hierarchy requires the use of 'virtual base classes'. However, this leads to implementations of classes which are, unexpectedly, many times larger than what was intended.

Another weak point is the lack of possibilities to selectively hide functions from inheritance. These problems led to an implementation that was effectively much more complicated than an implementation in C.

The most important point, however, is the current state of C++ compiler technology with respect to interoperability, availability and reliability. The support for overloading seriously hampers interoperability; at the moment, it is very well possible to incorporate C functionality into a C++ program, but the other way around is near impossible. Availability and reliability differ from company to company, but are too low to warrant a dependency of the Compare project. Therefore, Compare has abandoned C++ in favour of ANSI C.

5.2.2 A summary of the generated code A detailed description of every aspect of the generated code would far exceed the scope of this paper. Therefore, we will give a summary here.

Basically, the code generated from an fSDL specification consists of creation, deletion, copy and equality functions, functions yielding a unique identifier, functions to access fields,

and of course all additional functions generated by functor applications. For the purposes of debugging, and reading or writing an fDMCP from or to file, generic access functionality is provided. This means that for every domain and every operator, there is a list of functions providing access to all fields of the operator in the domain.

5.2.3 Interface and implementation The generated code consists of interface parts (.h files) and an implementation part (.c files). An fSDL specification causes many aliases for the same function. For example, the following extract from the FEMTO example:

```
domain Un: { id<n:Id>, const<v:Int> };
domain Exp: Un + (Bin - <*>);
```

causes, among others, the functions

```
Id Un_id_get_n(Un);   Id Exp_id_get_n(Exp);
```

In order to keep the executables, produced from code generated by fSDC, small, we have chosen to generate only one real function `x_Exp_id_get_n` in the implementation part ('invisible' to the user), and to define the aliases by way of macros in the interface part.

5.2.4 Typing and sort analysis There is some freedom in the mapping of domains to types (or classes) in the implementation language. In a very strict and type-safe approach, every domain would map to a distinct type or class. This approach has two disadvantages. Firstly, a lot of explicit (C) or implicit (C++) conversions have to be performed when the same operator is accessed through different domains. Secondly, it is hard to express those aspects that several operators have in common. In C++, this is problematic because of the limited power of its inheritance concepts, and in C it is impossible to find a layout of structs, such that a common field is always located at a common offset.

In a very loose and dynamic approach, all domains would map to the same type. In this type, any field of any operator should be represented. This leads to huge structures, which is not really compensated by the advantage that common aspects of several operators are expressed satisfyingly.

We have chosen for a compromise, in which the set of all operators is partitioned into *sorts*. The *sorts* are the smallest sets of operators such that for any domain, its operators are in a single sort. Every sort is assigned a distinct C type, and for every domain, a typedef declaration states C type equivalence between the domain and the sort. Note that a sort may contain strictly more operators than occur in any single domain, the extreme being a single sort that contains all operators.

In most cases, shared fields are implemented at the operator level. Therefore, shared fields only consume space if they must really be present, but their access functions do dynamic checking because the implementation of the access function depends on the actual operator. If all operators in a sort share a field, it can be implemented at the sort level, thus alleviating the need for a dynamic type check.

6. Conclusions

We have discussed an extensible language for the generation of parallel Data Manipulation and Control Packages. In several aspects, fSDL is complementary to IDL [Sno89].

Due to the intensive feedback of the COMPARE user community, the design and implementation of this language meets actual needs of compiler constructors.

fSDL differs from other interface definition languages in several aspects:

- Opaque domains provide access to arbitrary external datatypes without the need to specify a translation into internal primitives.
- The domain calculus provides very fine-grain access control, which is more flexible than the often rigid class hierarchies found in object oriented languages.
- The domain calculus allows for a conservative extension of data structures, for new processes added to a system. This is of importance when considering maintenance and extension of software systems.
- Functors provide extension with abstract data types. A powerful functor library, supporting the needs of the COMPARE user community is presented in [Tho93].

As an implementation language, C was preferred over C++. The main difficulties in using C++ were a lack of power in its inheritance concepts, and the unreliability (in our experience) of most current implementations. It must be mentioned that C++ does allow for a greater degree of static type checking. The main disadvantage of C, from the viewpoint of fSDL, is the lack of overloading. Thus, the names of the functions generated by fSDC are long.

We would like to express our thanks to all other members in the COMPARE consortium that have contributed to the design of fSDL.

REFERENCES

[CIL89] D. Clément, J. Incerpi, and B. Lang. *The virtual tree processor.* INRIA, Sophia-Antipolis, 1989.

[Con92] The COMPARE Consortium. *Investigation into CoSy*, 1992. ESPRIT project COMPARE delivarable D1.3.2/1.

[Gro90] J. Grosch. Ast – a generator for abstract syntax trees. Report 15, GMD, Karlsruhe, Karlsruhe, September 1990.

[HN86] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[JT89] Pierre Jouvelot and Rémi Triolet. *NewGen: A language-independent program generator.* Ecole des Mines, 1989.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic Effect Systems. *ACM Symposium on Principles of Programming Languages*, January 1988.

[LR91] M. S. Lam and M. C. Rinard. Coarse-Grain Parallel Programming in Jade. In *ACM Conference on Principles and Practice of Parallel Programming (PPOPP)*, pages 94–105. Computer Systems Laboratory, Stanford University, 1991.

[Sno89] Richard Snodgrass. *The Interface description language, Definition and use.* Principles of Computer science series. Computer Science Press, 1989.

[Tho93] François Thomasset. *A library of functors in fSDL: Reference Manual.* INRIA, 1993.