Combining VDM-SL Specifications with C++ Code

Brigitte Fröhlich[1] and Peter Gorm Larsen[2]

[1] University of Technology Graz, Münzgabenstr. 11/II, 8010 Graz, Austria
[2] IFAD (The Institute of Applied Computer Science), Forskerparken 10, 5230 Odense M, Denmark

Abstract. Experience shows that it is not economically feasible to formally specify all parts of a system in an industrial application. Either one already has a number of existing components which are trusted and therefore desirable for reuse, or components are so simple that there is no gain in formally specifying their behavior. In both cases it may be felt that it is not worth spending time on developing a detailed formal specification of the entire system. This raises the question what tools should be provided for the analysis of the entire system in which actual code is combined with specifications. In this paper we propose an approach which enables integration of code into a formal specification for prototyping facilities. The integration of code is supported by an extension to the IFAD VDM-SL Toolbox such that heterogeneous models can be interpreted.

1 Introduction

The successful introduction of formal methods into an industrial setting depends very much upon the choice of the parts of a given system that are subject to formal specification [14]. In the ConForm experiment it was concluded that formal specification is beneficial when either the functionality is simple but critical or the data structure or the functionality is complex [18]. The skill for selecting the most appropriate subsystems to be formally specified comes with experience in both formal specification and the given application domain. There is hardly any new input one can supply to meet this challenge. However, for tool developers supporting formal methods, it is a new challenge to enable users to gain insight in the interaction between the parts of the system which are being formally specified and the parts which simply are developed conventionally. With such a heterogeneous model of the system new problems arise.

There are a number of facilities, traditionally provided by operating systems or the programming languages, which for various reasons have been left out of conventional model-oriented specification languages such as Z and VDM-SL. These facilities include e.g. human-computer interfaces, input/output routines or the generation of random numbers. In this paper we will use as example the trigonometric functions which cannot appropriately be described in these specification languages because these kinds of primitives have not been included in Z and VDM-SL. Of course, as a tool developer one could extend the specification

language to include all of these useful features. The main drawback of doing this directly is that this approach is static and requires the tool developer to re-compile the entire tool whenever a new feature is desired by another user. Therefore in this paper we advocate a dynamic approach which enables the user to supply the extensions (s)he is interested in for a particular application.

The motivation for the approach we are presenting here is to enable the user to interpret an executable part of a heterogeneous model. The combination we are dealing with includes the formal specification language VDM-SL and the programming language C++, but the approach presented here could just as well be used with other notations. The intention is to provide a prototyping facility by which one can determine how the parts that have been specified interact with the coded parts.

The Vienna Development Method (VDM) [16, 4] is one of the most mature formal methods, primarily intended for the formal specification and the subsequent development of functional aspects of software systems. Its specification language VDM-SL [7] is used during the specification and design phases of a software development project, and it supports the production of correct high quality software. VDM-SL is being standardized under the auspices of the International Standard Institution (ISO) [2]. It is currently a Committee Draft standard under ISO and it is expected that this draft will be accepted as a final standard soon.

A modular extension to ISO VDM-SL is supported by the IFAD VDM-SL Toolbox [10, 21, 12]. The Toolbox supports extensive type checking, LaTeX pretty-printing facilities, test coverage, code generation, interpretation and many debugging facilities. A large subset of IFAD VDM-SL is executable by the existing interpreter of the Toolbox. The work presented in this paper is an extension of the Toolbox. The main impact of the extension is on the interpreter as will be illustrated below.

The Toolbox has been implemented in C++ using a VDM library which provides a generic implementation of all the data types of VDM-SL. The code generator from the Toolbox also produces C++ code using this library. C++ is an object-oriented extension of C and this library uses inheritance to build a class hierarchy of general VDM values in C++ [13]. For historical reasons the objects from this VDM library have been called "Meta-IV values".[3]

After this introduction we will present a small example illustrating the basic idea underlying our approach. This is followed by a short introduction to the parts of the existing interpreter which are relevant for this work. Then the extensions to the interpreter to incorporate this approach are presented. Different alternative communication primitives are then discussed. Finally, some related work is mentioned and a few concluding remarks about future work and the applicability of this approach are given.

[3] "Meta-IV" was the name of the Danish dialect of VDM-SL before the standardization started.

2 Illustration of the Basic Idea

The combination of formal specifications and code written in C++ only becomes feasible by establishing a common framework combining them. The most basic problem is that the values in these two different worlds have different representations. For the two worlds to communicate, it is necessary to convert values from the specification world to the code world and vice versa. By means of small examples this section illustrates the idea of integrating code with a specification in order to obtain the required functionality.

The examples present the integration of trigonometric functions into a specification using the IFAD VDM-SL Toolbox. A specification of the trigonometric functions is most naturally formulated algebraically stating the relationship between the different trigonometric functions. An algebraic specification style is not supported by VDM-SL and implicit functions belong to the not executable subset of VDM-SL. Therefore the trigonometric functions cannot appropriately be described in the specification language with the intention to execute it. However, an implementation of these functions is part of almost every programming language.

2.1 Basic Idea

The aim of this approach is to be able to analyze the combination of specification and implementation. By combining code and specification our main intention is to provide a prototyping facility by integrating the execution of code into the interpretation of a specification. We enable definitions, made at the code level, to be integrated with a formal specification, such that their execution during the interpretation of a specification is enabled. Therefore we distinguish between the specification level, which refers to the Toolbox, and the code level which relates to the integration of code for the execution.

Figure 1 shows our approach of combining code and specification. The dashed boxes mark the different levels; the IFAD VDM-SL Toolbox represents the specification level and the code level corresponds to the code units. These code units are represented as "dynamic linked libraries" to the interpreter process. The bold boxes indicate the parts which the user has to develop additionally in order to combine code and VDM-SL specification.

For every unit of code, which may consist of one or more C++ files, an interface at the specification level as well as an interface at the code level must be developed by the user of the Toolbox. At the specification level a new kind of module, called *implementation module*, has been introduced. The module concept of the IFAD VDM-SL specification language provides a facility for the combination of multiple modules, which has been used for our approach.

An implementation module contains the corresponding type information for every definition of the C++ code to be integrated with the VDM-SL specification. For functions, this type information consists of a signature in VDM-SL and a VDM-SL type declaration for values. The implementation modules have to be imported by every module that accesses a function or value defined in code.

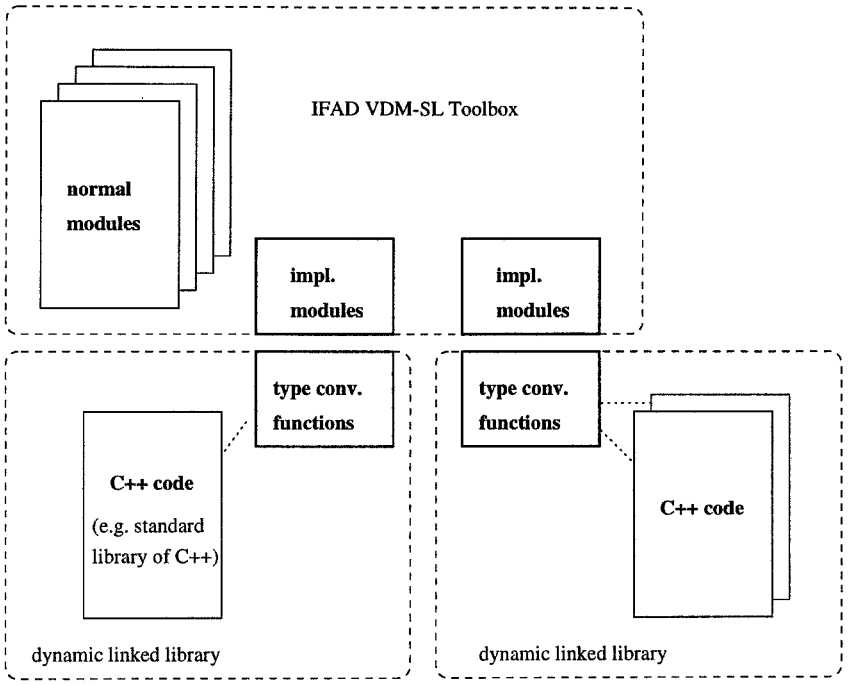

Fig. 1. Basic idea of combining code and specification in the Toolbox

The aim is to provide a prototyping facility by combining the interpretation of a specification with the execution of a module implemented in C++. As the C++ code and the IFAD VDM-SL interpreter have different representations for their types, a type conversion of values between those two worlds by the definition of type conversion functions is established. A type conversion function takes as argument values of the interpreter process, which are values of the VDM C++ class types (Meta-IV values), and converts these to values required by integrated code. The results of the C++ module are converted to Meta-IV values before being returned.

Our approach allows the integration of more than one dynamic library with a formal specification. The only requirement of the Toolbox is that an implementation module defined at the specification level exists for any dynamic linked library.

An important design decision has been to enforce the reuse of code and to support the reuse of specifications. The reuse of code is forced by establishing this prototyping facility. The type conversion functions contain all modifications to the code which are required in order to communicate with the interpreter process such that the C++ code used should not need to be modified for its integration into a specification. The reuse of formal specifications is accomplished by the implementation module which can be imported in every ordinary VDM-SL module.

2.2 An Example

In order to illustrate the idea we present an example which integrates trigonometric functions into a specification. The functionality of trigonometric functions can only be described by approximations given the primitives of VDM-SL. However, almost every programming language contains an implementation in their standard libraries. This is one of the examples where the approach advocated here may be beneficial with respect to prototyping.

The C++ code, which is to be integrated into a VDM-SL specification, is given by the mathematical standard library of C++ called *math*. The examples make use of implementations of the trigonometric functions *sin*, *cos* and of the constant *pi*. First we present a specification which applies these definitions and then we show the extensions at specification level as well as at code level.

The VDM-SL Specification

The structuring concept of the module extension of VDM-SL which is used in the Toolbox enables a combination of modules based on import and export of constructs [8]. Modules can also be parameterized but we will not consider that possibility here. To combine modules, a module has to provide an interface to other modules including export of definitions which may be used by other modules.

The following VDM-SL specification gives an example of how the definitions of the implementation module `MY_MATH` are imported into the module `USE_MATH`.

```
module USE_MATH

  imports
    from MY_MATH
      functions
        MyCos : real -> real;
        MySin : real -> real;

      values
        MyPI : real

   definitions
     functions
       CircCyl_Vol : real * real * real -> real
       CircCyl_Vol (r, h, a) = MY_MATH`MyPI * r * r * h *
                                              MY_MATH`MySin(a)

end USE_MATH
```

The module `USE_MATH` imports the functions `MyCos` and `MySin` and the value `MyPI` from the implementation module `MY_MATH`. The function `CircCyl_Vol` evaluates the volume of a circular cylinder and makes use of the constant `MyPI` as well as the function `MySin`.

The Interface

As mentioned above, the interface between code and specification has to be provided at two different levels. The interface at the VDM-SL level specifies VDM-SL types of the implemented functions. The interface at the code level is based on the definition of type conversion functions, i.e. functions which establish type conversions between code and interpreter based on the VDM C++ library. Both interfaces must be developed by the user of the Toolbox in order to apply this prototyping facility. As the definitions of both parts are strongly related, we present both interfaces in two boxes below.

```
implmodule MY_MATH

  exports
    functions
      MyCos : real -> real;
      MySin : real -> real

    values
      MyPI : real

    uselib
      mymath.lib
end MY_MATH
```

```
#include "metaiv.h"
#include <math.h>

Generic MyCos(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( cos(rad) ));
}

Generic MySin(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( sin(rad) ));
}

Generic MyPI (Sequence sq)
{
  return(Real(M_PI));
}
```

The part between **implmodule MY_MATH** and **end MY_MATH** is the definition of an implementation module and constitutes the interface to VDM-SL. An implementation module always contains an export section which declares all constructs that are exported to the outside world. The constructs from the export section can be imported by other modules. The export section consists of the signatures for function definitions and the type information for value definitions related to the code. A value declaration in an implementation module relates either to a constant or variable definition in the code.

The **uselib** field contains a reference to the dynamic linked library where the code can be found. The relation between implementation module name **MY_MATH** and dynamic linked library **mymath.lib** is important for interpreting the specification later on.

The interface at the code level is developed in C++ and consists of some

declarations and the definitions of the type conversion functions. The declaration part includes the standard mathematical library. The type conversion function converts the generic Meta-IV sequence value used by the interpreter to values accepted by the implemented code and vice versa. A generic VDM C++ type can have an underlying value of any VDM type. (e.g. the generic VDM C++ type `Sequence` represents a VDM-SL sequence which can contain arbitrary VDM elements.)

The interpreter process puts the arguments of the called function into a value of the generic VDM C++ type sequence and passes it to the type conversion function. It converts the elements of the sequence into values required by the integrated C++ code. The type conversion function `MySin` extracts the first element of the sequence and casts the generic type to the VDM C++ type `Real`, which is casted automatically to the C++ type `double`. After evaluating the `sin` at C++ level, the result is converted into a Meta-IV value and returned to the interpreter process.

In this case we used a C++ standard library as code, but it could also be a user-defined C++ package. Our approach is open to every kind of module developed in C++. The user is only required to develop an implementation module and to define type conversion functions. Type conversion functions and executable modules together form the dynamic linked library.

3 The Existing VDM-SL Interpreter

The IFAD VDM-SL Toolbox includes an interpreter for the execution of specifications for testing purposes. This interpreter supports a large subset of the VDM-SL specification language. In this work we extend the interpreter to include executions of functions which are implemented in C++ and integrated into a specification.

In order to understand the modifications of the interpreter, basic knowledge about certain parts of the existing interpreter is required. Before the interpreter is able to interpret an expression which makes use of definitions from a module it is necessary to translate each of the modules. The module translation takes as input an abstract syntax tree of the modules and creates a semantic representation for every module, which establishes the context for the interpretation of a construct in a given module. By initializing the specification, the global values of the semantic domain are initialized and parameterized modules are instantiated. The actual evaluation of an expression makes use of the context which has been incorporated in the previous steps.

3.1 Relation between Abstract Syntax and Semantic Values

The module translation establishes a related semantic value to an abstract syntax representation of a module. A module consists of an interface section and a definition section. Again, the module interface is separated in an import section, which lists all definitions used from other modules, and an export section which

creates an interface to other modules. All the definitions in the export section will be visible outside the module. The translation of the interface section creates global values which form a semantic module value called `SigmaMO` together with the semantic values created by the definitions.

In most cases the semantic values of the definitions are strongly related to the abstract syntax representation. An explicit function definition is used to show this correlation.

$$\begin{array}{lll} ExplFnDef :: nm & : Name \\ \quad tp & : FnType \\ \quad parms & : ParametersList \\ \quad body & : Expr \end{array}$$

The abstract syntax for an explicit function *ExplFnDef* consists of the function name, *nm*, the function type, *tp*, a list of the formal parameters, *parms*, and the body expression, *body*.

$$\begin{array}{lll} ExplFN :: tp & : FnType \\ \quad parms & : ParametersList \\ \quad body & : Expr \\ \quad env & : BlkEnv \end{array}$$

Its semantic value *ExplFN* used in the interpreter is very similar to the abstract syntax. The semantic value of an explicit function definition consists of the function type, *tp*, the formal parameter list, *parms*, the function body expression, *body*, and a closure environment, *env*.

A closure model is used to establish the defining environment of locally defined functions. VDM-SL uses, like most programming languages, static scoping for functions evaluation, which means that a function is evaluated in its defining environment.

$$\begin{array}{lll} SigmaMO :: explfns & : Name \xrightarrow{m} ExplFN \\ \quad \ldots & : \ldots \end{array}$$

For every created semantic value a representation in the module value is introduced. Usually this is a mapping from name to its semantic value as shown above for explicit function definitions. A semantic module value is integrated in the interpreter state as a mapping from module name to module value. The context for the evaluation of a specification, which consists of one or more modules, is established by the combination of all related module values.

3.2 The Evaluation Model

The IFAD VDM-SL interpreter uses an environment based evaluation model [19]. The main structure of the semantic domain is the environment (ENV_L) which is organized as a stack of function application environments (ENV).

$$ENV_L = ENV^*;$$

$ENV = BlkEnv^*$

When a function is called, it pushes a function application environment on top of the stack. A function application environment contains a number of *block environments*. These block environments are created as a result of the introduction of a local identifier, for example in a let-expression. A block environment introduces a new scope within the current function application environment.

$BlkEnv = IdVal^*$;

$IdVal = AS`Name \times VAL$

The function application environment is organized as a stack of block environments. A block environment is a sequence of *IdVal* which relates a unique name to its semantic value.

The evaluation of a function application expression modifies the environment stack ENV_L in the following ways. The actual arguments are evaluated and a new and empty function application environment is pushed on top of the evaluation stack. Then their block environments are pushed onto the function application environment. These block environments contain bindings introduced by pattern matching and the bindings of a closure environment. One block environment is created by matching the function parameter against the evaluated arguments and pushing it onto the top of the function application environment.

During evaluation of the body of a function, all the identifiers must be defined either locally, which means in one of the block environments of the top function application environment, or defined globally. After function body expression evaluation the top function application environment is removed from the evaluation stack and the result value is returned.

The global outline of the specification of *EvalApplyExpr* is:

```
EvalApplyExpr : ApplyExpr ->o VAL
EvalApplyExpr (mk-ApplyExpr (fct-e, arg-le)) ≜
  let fct-v = EvalExpr (fct-e),
      arg-lv = [EvalExpr (arg-le (i)) | i ∈ elems arg-le] in
  (PushEmptyEnv() ;
   PushBlkEnv(fct-v.closenv) ;
   let benv ∈ PatternMatch (fct-v.parms, arg-lv) in
   (PushBlkEnv(benv) ;
    let res-v = EvalExpr (fct-v.body-e) in
    (PopEnvL() ;
     return res-v )))
```

where *fct-e* is the function expression to be evaluated and *arg-le* the argument expressions to the functions. The result of evaluating *fct-e* is a semantic value, *fct-v*. This function value contains the parameters of the function, *parms*, and the body of the function, *body*.

4 Modifications to the VDM-SL Interpreter

The introduction of the concept of implementation modules to IFAD VDM-SL requires some modification of the IFAD VDM-SL specification language and the IFAD VDM-SL Toolbox. This section illustrates the extensions related to the specification language and the dynamic semantic specification.

4.1 Abstract Syntax

The implementation module *ImplModule* has been invented for the specification of an interface to VDM-SL .

$$
\begin{array}{l}
\mathit{ImplModule} :: \mathit{nm} \quad : \mathit{Name} \\
\qquad\qquad\quad \mathit{intf} \quad : \mathit{ImplInterface} \\
\qquad\qquad\quad \mathit{uselib} : \mathit{Name};
\end{array}
$$

$$
\begin{array}{l}
\mathit{ImplInterface} :: \mathit{imp}: \mathit{Name} \xrightarrow{m} \mathit{ImplImportSig} \\
\qquad\qquad\qquad\quad \mathit{exp} : \mathit{ImplExportSig};
\end{array}
$$

$$
\mathit{ImplImportSig} :: \mathit{tps} : \mathit{Name}\text{-set};
$$

$$
\begin{array}{l}
\mathit{ImplExportSig} :: \mathit{val} \ : \mathit{Name} \xrightarrow{m} \mathit{Type} \\
\qquad\qquad\qquad\quad \mathit{fns} \ : \mathit{Name} \xrightarrow{m} \mathit{FnType} \\
\qquad\qquad\qquad\quad \mathit{ops} : \mathit{Name} \xrightarrow{m} \mathit{OpType}
\end{array}
$$

The abstract syntax of an implementation module consists of a module name, *nm*, an interface section, *intf*, and information related to the integrated code, *uselib*. A module interface consists of an import and an export section. The import section of an implementation module is restricted to importing type names. The type names are required, by the type checker, in case a user wants to refer to types defined in other modules. The export section mentions all constructs which can be imported by other modules.

The distinction between functions and operations is pragmatic in an implementation module. In VDM-SL a "procedure" which defines or modifies a state is called an operation. In the case of implementation modules the distinction is without meaning as the related definitions are C++ code.

4.2 Extensions to the Semantic Domain

The translation of an implementation module creates the semantic values from the definition in the export section. In addition, a reference to the C++ code library must be used such that it will be possible to refer to it when needed. The module translation of an implementation module establishes a relation between the implementation module and its related code. Related to the VDM-SL specification level the module name (`USE_MATH`) contains the necessary information

for establishing the context for the interpreter as it refers to the module value (see Section 3.1). The interpreter process requires access to the dynamic linked library for executing the code. The `uselib` section of an implementation module contains the name of the dynamic linked library. This name is stored in the semantic module value, such that the interpreter process can access its operation during execution of a specification.

The semantic values of functions and operations, which consist either of an implicit or an explicit definition, are extended by *external* definitions. An external function is declared in an implementation module and is defined in the related code.

$$\begin{aligned} &ExtFN :: tp \quad : FnType \\ &\qquad\quad body : [LOC] \end{aligned}$$

The semantic value $ExtFN$ of an external function consists of the function type, tp, and the function body, $body$. Because of the missing function definitions at the VDM-SL level neither a closure environment, nor a parameter list nor a function body expression exist. The interpreter implementation uses the memory location, where the function is stored, as representation of the function body. This concept is applied by the semantic value LOC of the function body.

Besides the introduction of new semantic values, the global values are extended by a new field for implementation modules. The translation of an implementation module creates a module value $SigmaIMO$ and extends the state definition by a mapping from the module name to this value.

4.3 The Dynamic Semantic

The dynamic semantic specification is extended with a specification for applying external functions. If the interpreter needs to apply an external function, it evaluates the function expression and arguments first. The result of evaluating the function expression is a semantic value of type $ExtFN$.

The global outline of the specification $EvalApplyExt$ is:

$$\begin{aligned} &EvalExtApply : ApplyExtExpr \xrightarrow{o} VAL \\ &EvalExtApply\,(\mathsf{mk\text{-}}ApplyExtExpr\,(fct\text{-}e, arg\text{-}le)) \triangleq \\ &\quad \mathsf{let}\ fct\text{-}v = EvalExpr\,(fct\text{-}e), \\ &\qquad arg\text{-}lv = [EvalExpr\,(arg\text{-}le\,(i)) \mid i \in \mathsf{elems}\ arg\text{-}le]\ \mathsf{in} \\ &\quad (\mathsf{let}\ res\text{-}v = EvalExtBody\,(fct\text{-}v.body, arg\text{-}lv)\ \mathsf{in} \\ &\qquad \mathsf{return}\ res\text{-}v\ ) \end{aligned}$$

where $fct\text{-}e$ is the function expression to be evaluated and $arg\text{-}le$ the argument expressions to the functions. The result of evaluating $fct\text{-}e$ is a semantic value, $fct\text{-}v$.

Comparing with Section 3.2 the application of an external function does not require the creation of a function application environment in the interpreter. An external function is evaluated in the environment established by the execution of the related code.

The application of an externally defined function cannot be specified in an appropriate way because the C++ code is not explicitly modeled. The interpreter cannot make any assumptions about the functionality of the function because of the missing function body. Instead of interpreting a function body a function defined in a dynamic linked library is executed.

The dynamic semantic of the function *EvalExtBody* can be described in terms of the implementation but not in the specification. The meaning of the function *EvalExtBody* related to the semantic value of an external function is expressed in the implementation of the interpreter as

```
res_v = (*fct_v.body)(arg_lv);
```

where $fct_v.body$ is a reference to a function in a dynamic linked library. A reference to a function can be executed in C++ after *dereferencing* it [22].

5 Different Communication Primitives

The implementation of the interpreter has to provide a facility to execute functions which are defined in code. In case a function from an implementation module is applied, the Toolbox process knows the name of the function and can calculate the actual arguments. Based on this information, communication between the Toolbox process and the executable code has to be assembled. The use of dynamic linking offers the required functionality and meets our implementation requirements.

5.1 Dynamic Linking

Static linking of the integrated code and the interpreter implementation would make the integrated code part of the interpreter process. Every time the code changes, a new creation of an interpreter process is required. This approach does not support our intentions of flexibility and efficient usage of this feature. The interpreter would combine internal specification and code. A separation of both seems to be desirable in order to enable a dynamic system implementation which supports the replacement of a specified module by its implementation. If the other parts of the specification are also executable, a kind of integration test for the implementation is provided by executing the specification [6].

Dynamic linking postpones the linking of components until execution time [1]. Functions defined in shared libraries are linked and loaded if their objects are accessed during run-time. Shared libraries are "transparent" to the process using them. This means that an object in a shared library is added to the address space at the time of execution. Further it does not restrict the types passed between the Toolbox process and the interface to code to a subset of VDM-SL types . However, the conversion functions have to transform the Meta-IV values into values which are used in the C++ code.

It is worth noting that the interface to code and the implemented code together form a shared library for each implementation module. The application

of a shared library does not depend on process communication but requires operating system support [23]. The IFAD VDM-SL Toolbox is available for different operating systems which support dynamic linking in a similar way. The operating systems contain functions which provide a simple programmatic interface to handle dynamic linked libraries.

These programmatic interfaces always contain an **open** function which provides access to the library. It returns a reference to the shared library which can be used for later accesses to their objects. A `symbol` function retrieves the address binding for a function defined in the objects of the library. A reference to a shared object is deleted by a `close` function.

If the interpreter process has to evaluate an external function, it evaluates the arguments first. Afterwards the process retrieves the address binding of the function in the shared library. It should be mentioned that the interpreter searches for the type conversion function defined in the interface to code which, in turn, calls a function of the implemented code.

By accessing an object defined in a shared library a pointer is returned, which must be casted to a function type. The type checking of C++ prohibits a cast to a function type which has a variable number of arguments [9]. A solution for this problem is provided by the usage of Meta-IV values. The interpreter process passes a Meta-IV sequence value to the interface, where the length of the sequence is equal to the arity of the function. The effort related to the interpreter is to put the evaluated argument as Meta-IV values in a sequence. The type conversion functions must extract the values from the sequence and convert those into types required as arguments by the implemented code. The return value exchanged between the interface and Toolbox process is a generic Meta-IV value.

5.2 Alternative Communication Approaches

Other possible approaches establishing the communication primitives are Unix pipes. They provide a stream oriented communication medium between two different processes. A process can establish a pipe to exchange data with another process. As a pipe enables communication in one direction, two pipes would be required for bidirectional communication.

A disadvantage is that the stream oriented communication between interpreter and interface process will restrict types to a subset of the common VDM-SL types. There is also a considerable overhead for doing type conversions between the Toolbox process, the interface to C and the implemented code. An example of using pipes for interprocess communication in connection with the VDM-SL Toolbox is used in the IPTES Project [26]. In this case the data types are restricted to the VDM basic types and to sequences and tuples of it.

The interprocess communication facilities like message passing, shared memory or remote procedure calls would be required if the feature should support the integration of code whereas code and specification are stored on different machines. This case was not considered in this work but is supported in the IPTES Project.

6 Related Work

A number of researchers have worked on a common framework which can be used to combine different formalisms. This includes both theoretical work on the foundations [3] and more practical applications [27, 28]. Most of the existing work, however is focusing on how to combine different specification notations.

In our work we wish to enable a heterogeneous combination of a specification language and an implementation language. In this area related work can be found in the Larch community [17]. An approach of mixed evaluation of an algebraic specification and implemented parts in code is given in [5]. The execution is obtained by applying rewriting techniques to the algebraic specifications and direct evaluation of implemented parts.

The production of an executable model from a given VDM specification is presented in [25]. This work shows the integration of a VDM specification into the KIDS Toolbox which is used for executing the specification [20]. Another work in the VDM community which is somewhat related is the specification of library components in a programming language [11]. However, there is a large number of industrial applications of formal methods, where only a part of the system has been formally specified (e.g. [15]) which could benefit from the approach presented in this paper.

The idea of dynamically extending the functionality of a tool has already been used in the programming language community [24]. This concept is naturally a powerful feature which is desirable from a tool point of view.

7 Concluding Remarks

In this paper we have presented an approach enabling users to dynamically extend tools supporting the interpretation of formal specification. We believe that this approach will turn out to be valuable for practical applications where one wishes to gain insight into the interaction between parts of a system which have been formally specified whereas other parts simply have been developed conventionally. As specification languages get increasingly standardized and the supporting tools get more powerful we believe that a dynamic approach is essential.

Incorporating the ability to supply pre and post conditions for the different functions that are coded externally in C++ is considered as future work. It should then be possible to switch such conditions on or off during the interpretation. We also feel that more investigation is needed in order to determine what to do in the situation where the code for some reason fails to deliver an appropriate answer or result in a run-time error. We have not yet had time to properly work on the issue of such exception.

We feel that the most important work that needs to be done is to get feedback from industrial users of the IFAD VDM-SL Toolbox about this approach. We have already agreed with some of the users that they will get access to a beta version of this extension before it is released to everyone. Their feedback about

the applicability of this approach will ultimately determine whether these new ideas will be adopted by other tool vendors to provide support for heterogeneous models of a system.

Acknowledgments

We would like to thank John Fitzgerald, Nico Plat and Paul Mukherjee for giving their remarks at an oral presentation of this work. We would also like to thank Peter Lucas and Werner Kerschenbauer for valuable remarks on an earlier version of this article. In addition we have had constructive remarks from our colleagues at IFAD. We are grateful to the support of the European Commission (Comett Grant) for Brigitte Fröhlich's stay at IFAD.

References

1. Silberschatz A., Peterson J., and Galvin P. *Operating System Concepts.* Addison Wesley, 1991.
2. D.J. Andrews, H. Bruun, B.S. Hansen, P.G. Larsen, N. Plat, et al. *Information Technology — Programming Languages, their Environments and System Software Interfaces — Vienna Development Method-Specification Language Part 1: Base language.* ISO, 1995.
3. E. Astesiano and M. Cerioli. Multiparadigm Specification Languages: A First Attempt at Foundations. In J.F.G̃root̃e D.J.Ãndrews and C.A. Middelburg, editors, *Semantics of Specification Languages*, pages 168–185, 25–27 October 1993, Utrecht Springer-Verlag 1994.
4. D. Bjøner and C.B. Jones, editors. *Formal Specification and Software Development.* Prentice-Hall International, 1982.
5. S. Kapplan C. Choopy. Mixing abstract and concret Modules: Specification, Development and Prototyping. In *12th IEEE-ACM International Conference on Software Engineering, Nice 1990*, pages 173–184, 1990.
6. Christine Choppy. Formal Specifications, Prototyping and Integrations Tests. In D. Simpson H.K. Nichols, editor, *ESEC'87 1.st European Software Engineering Conference, Strasbourg, 1987*, Lecture Notes in Computer Science 289, pages 172–179. Springer Verlag, September 1987.
7. John Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.
8. Kees de Bruin. Towards an Interpreter for Full VDM-SL. Master's thesis, Delft University, June 1993.
9. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.
10. René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.
11. B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-based Software Component Retrieval Tool. Technical Report 94-08, Technische Universität Braunsweig, November 1994.
12. The VDM-SL Tool Group. The IFAD VDM-SL Language. Technical report, IFAD, December 1994.

13. The VDM-SL Tool Group. The VDM C++ Library. Technical report, IFAD, October 1995.
14. Michael G. Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.
15. Iain Houston and Steve King. CICS Project Report: Experiences and Results from the Use of Z in IBM. In S. Prehn and W.J. Toetenel, editors, *VDM'91 – Formal Software Development Methods*, pages 588–696. Springer-Verlag, October 1991.
16. Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
17. J.J. Horning J.V. Guttag and J.M. Wing. Larch in Five Easy Pieces. Technical report, Digital Systems Research Center, July 1985.
18. Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Lessons Learned from Applying Formal Specification in Industry. *Submitted to IEEE Software*, August 1995.
19. Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods.* VDM Europe, Springer-Verlag, March 1991.
20. Yves Ledru. Proof-Based Development of Specifications with KIDS/VDM. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 214–232. Springer-Verlag, October 1994.
21. Paul Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.
22. B. Stroustrup. *The C++ Programming Language, 2nd edition.* Addison Wesley Publishing Company, 1991.
23. SUN-OS. Programmer's Overview Utilities & Libraries. Technical report, Sun Microsystems, Inc., 27 March 1990.
24. Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc, 1992.
25. M.-H. Liégeois Y. Ledru. Integrating REFINE prototypes in a VDM development framework. In *Constructing Programs from Specifications*, pages 236–258, 1991.
26. Nabil Zakhama. Run-Time Adaptation System – Target Code Procedures Activato (RTAS – TCPACT) – User's Guide. Technical report, Télésystèmes, March 1993. IPTES Doc.id.: IPTES-TS-53-V2.1.
27. Pamela Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software*, pages 15–25, September 1989.
28. Pamela Zave and Michael Jackson. Conjunction as Composition. *ACM Transactions on Software Engineering and Methology*, 2(4):379–411, October 1993.