Implementing Modules Specified in the Trace Assertion Method

Michal Iglewski[a], Janina Mincer-Daszkiewicz[b]

a. Département d'informatique, Université du Québec à Hull, Hull, Québec, Canada J8X 3X7
Email: iglewski@uqah.uquebec.ca Tel: (819) 773 1602 Fax: (819) 773 1638
b. Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland
Email: jmd@mimuw.edu.pl Tel: (48-2) 658 3165 Fax: (48-2) 658 3164

ABSTRACT

A software module may be described precisely and completely by a set of related documents: module interface specification providing a "black-box" description of its behavior, module internal design containing its "clear-box" description, and the code itself. A special formalism is needed in each of this documents. We use the trace assertion method for specification of module interfaces, and LD-relations, known as program functions, to specify behavior of separate programs within a module. We started the project aiming at the implementation of the integrated set of syntax-driven editors supporting system documentation process. Many practical problems of the involved documentation methodology were recognized, some solutions were proposed and verified. In this paper we summarize our experiences. Appendices contain a complete set of documents for sample modules.

1 Introduction

Use of proper documentation techniques at every stage of software development [7, 8] is crucial for the success of any software project. Documents describing different software components should be precise, complete, and readable. It is difficult or hardly possible to achieve this goal without a support of appropriate software tools. In this paper we present a set of documents used to describe components of a software system and introduce an integrated set of tools supporting this task.

According to a well established model of software design, software should be hierarchically structured and consist of a set of information-hiding modules [6]. A module implement objects which can be manipulated from outside the module by means of its access-programs.

In the model considered in this paper, the description of each of these modules consists of three documents. A *module interface specification* provides a "black-box" description of the behavior of the module's objects. We use the *trace assertion method* (in short: TAM) [1, 9, 10, 4] as the specification formalism.

For every implementation of a module interface specification there should be a document describing the *module internal design*. That document presents the module's internal data structures and the effect of its access-programs on the state of that structure, i.e., it provides a "clear-box" description. We use an abstraction function to describe the interpretation of the concrete states of data structures in terms of the abstract states of objects (recognized at the interface specification level). LD-relations are used to define the mapping between the starting and final states of access-programs.

The third document is the code of the module.

In this paper we focus our attention on writing internal design documents which conform to interface specifications given in TAM. At the internal design level a decision should be made either to make objects part of the module's internal data structure or to make them available to other modules as building blocks of these modules' own objects — hierarchically structured objects may be built in this way. Different forms of sharing the data structure between a

module and its clients give rise to a classification of module internal designs into centralized, decentralized, and mixed. The exact form of the internal design document is proposed and explained on examples. The simplified form of the program functions (as compared to [3]) is presented.

We also discuss and propose solutions to certain problems concerning the proper use of objects during the coding phase.

In Section 2 of the paper we explain the main concepts of TAM and characterize the structure of module interface specifications. Certain aspects of specification of hierarchically dependent modules are also discussed there. Section 3 describes the structure of a module internal design document. Special attention is given to the design of hierarchical modules. Problems concerning the transformation of the module internal design into the code are considered in Section 4. Conclusions are formulated in Section 5. Appendices contain the complete set of documents for three sample modules.

2 The module interface specification using TAM

2.1 Main concepts of the trace assertion method

According to the information hiding principle, software systems are decomposed into *modules*. The interface to a module is an abstraction that is defined without any reference to the hidden decisions such as choice of data structure or algorithms. TAM is a formal method for "black-box" specification of module interfaces. A module implements one or more *objects*. Objects within a module are homogeneous and independent. An object may change its state only as a result of an *event* and may produce outputs in response to it. The only events regarded in this study are invocations of the module's *access-programs*. A finite sequence of such invocations is called a *trace* and corresponds to the current state of the object; the empty trace is considered to be the initial state of the object. Two traces are *observationally equivalent* if the object's behaviors after these traces are the same. It is reasonable to reduce the state space by regarding only representatives of the equivalence classes of this relation; they are called *canonical traces*. The state changes of the object are described by an *extension function* mapping from single event extensions of canonical traces to the equivalent canonical traces. Outputs produced by the object are described by an *output relation*.

2.2 Structure of module interface specifications

The notation of trace specifications used in this paper is that from FUN-SPEC — the syntax-driven editor for module interface specifications ([2]). Examples are given in Appendices.

A module interface specification consists of five sections. The Characteristics Section states the name of the module (type being specified — called the *domestic* type) and its overall features: whether it is single or multi-object, deterministic or non-deterministic. This section also lists *foreign* types used in the module, and optionally, the type used to name its objects. Names of types are written by convention in angle brackets. Certain types like <int>, <real>, <bool>, <char>, and <string> are supposed to be known, and we call them *built-in types*. We use the orthodox notation for their literals and operators.

The Syntax Section contains a syntax table which lists the module's access-programs and for each program — its arguments and the returned value (if any). Each argument is described by its position in the argument list, its type and a descriptor which denotes its input-output characteristics. The descriptor is a combination of letters "V", "O" (or "R"), and "N". Their meaning is as follows:

- "V" — the value of the argument may be used by the program,
- "O" — the value of the argument may be changed by the program (if "R" is used in place of "O", then the new value is non-deterministic),

• "N" — the name of the argument may be used by the program.

These descriptors are general enough to cover most of the argument passing rules found in the imperative programming languages. The actual way of passing arguments to the program is determined at the implementation level.

In the case of a single-object module the only object of this module could be omitted in access-programs invocations[1]. In order to keep a uniform notation for both kinds of modules, we will write it explicitly as zeroth argument.

The Canonical Trace Section defines a predicate "canonical", whose domain is the set of syntactically correct traces. This section may also include definitions of auxiliary functions, which are introduced to simplify expressions used in specifications. If the empty trace is not canonical, then a canonical trace equivalent to it should also be specified in this section.

The Equivalence Section describes legality of invocations and the extension function. For each access-program the function *Legality* defines legal invocations of this program and characterizes possible illegal invocations denoting them with separate error tokens (e.g. %not enough memory%, %no init%). The extension function, *ef*, defines new states of domestic arguments. The new state of each domestic argument of each access-program is given in a separate equation (the argument being defined is tagged by an arrow). If the state does not change for certain invocations, then these invocations can be omitted in the definition of *ef*.

The Return Value Section defines new values of output arguments of foreign types. The argument being defined is marked by an arrow. If an invocation is illegal then values of its output arguments can be specified as *undefined*. If for a particular access-program invocation values of some output arguments depend on each other, they should be described in one definition.

In all sections, tables may be used to improve the readability of definitions of functions and relations.

The wild card symbol '*' is used throughout the specification to replace an argument of an access-program invocation if the name or value of this argument is obvious or not used in the given context.

The syntax of trace specifications has evolved during the process of implementing FUN-SPEC and using it in the course on Software Engineering at the University of Quebec at Hull. The feedback from the students helped us to improve readability of the notation.

2.3 Specifications of hierarchical modules

Modules within a software system are usually inter-dependent. This dependence is expressed by listing names of the referenced modules in the foreign type list of the defined module. This gives access to all definitions from the interface specification of a given foreign module, in particular to its access-programs, canonical traces, and possible auxiliary functions.

A module may construct its objects from another module's objects.

Example:

The predicate "canonical" in the module <point> might be defined as follows:

canonical(T) $\Leftrightarrow \exists$ x, y:<real> [T = SETPOINT(*, x, y)]

A vector is an ordered pair of points, so it might be defined as follows:

canonical(T) $\Leftrightarrow \exists$ p, q:<point> [T = SETVECTOR(*, p, q)]

A broken line is an ordered sequence of points, so the predicate "canonical" in the module <bline> might be:

canonical(T) $\Leftrightarrow \exists\, p_1 \ldots p_n$:<point> $[T = [\mathrm{ADDPOINT}(*, p_i)]_{i=1}^{n}]$

SETPOINT, SETVECTOR, and ADDPOINT are access-programs in the modules <point>, <vector>, and <bline>, respectively. The module <point> constructs points from the objects of the built-in type <real>, the modules

1. This convention is used in [9, 3].

<vector> and <bline> construct their objects from the objects of the user-defined type <point>.

A module may use objects from another module by referencing their values and/or assigning them new ones.

Example:

Let us assume that the module <vector> defines the access-program MOVE(<point>:VO, <vector>:V) which moves a given point by a given vector:

MOVE(p , v) = q where q:<point>, qx,qy:<real>

(q = SETPOINT(*, qx, qy) ∧ qx = *point::getx*(p) + *getdx*(v) ∧ qy = *point::gety*(p) + *getdy*(v))

and *getdx*, *getdy* are the auxiliary functions in the module <vector> defined as:

getdx, getdy: <vector> → <real>

getdx(v) $\stackrel{df}{=}$ (*point::getx*(q) - *point::getx*(p)) where p,q:<point> (v = SETVECTOR(*, p, q))

getdy(v) $\stackrel{df}{=}$ (*point::gety*(q) - *point::gety*(p)) where p,q:<point> (v = SETVECTOR(*, p, q))

and *getx*, *gety* are the auxiliary functions in the module <point> defined as:

getx, gety: <point> → <real>

getx(p) $\stackrel{df}{=}$ x where x,y:<real> (p = SETPOINT(*, x, y))

gety(p) $\stackrel{df}{=}$ y where x,y:<real> (p = SETPOINT(*, x, y))

The name of a foreign type followed by two colons is used to prefix entities defined in the foreign module (*point::* in the example above).

These module dependences may be specified without the knowledge of module implementations.

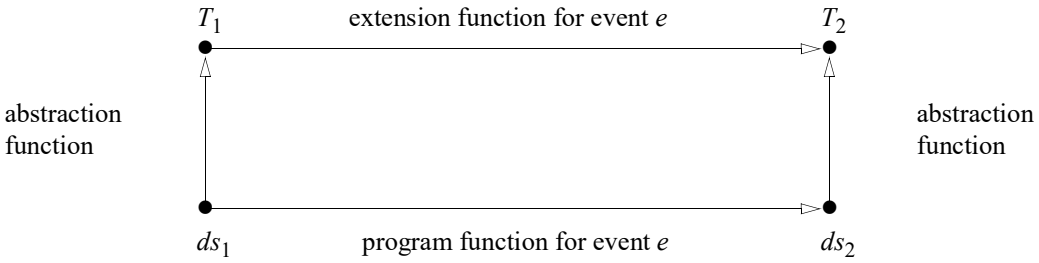
3 From module interface specification to module internal design

3.1 Introduction

A module internal design is a bridge between the module interface specification and its implementation. It gives a “clear-box” description of the module by describing its implementation in terms of a data structure that consists of other data types. This data structure is considered to be the module’s secret and is not known outside it (in particular, it can not be referenced in the interface specification of any other module). The internal design provides a mapping from *concrete states* of the module’s internal data structure to *abstract states* of objects (recognized at the interface specification level as canonical traces). This mapping is described by the *abstraction function*.

The semantics of the module’s access-programs is defined by describing their effect on the state of the module data structure. This effect is expressed by an *LD-relation* which consists of a relation and a set. The relation describes a mapping between the starting and final states of the program. The set is a collection of starting states for which termination is guaranteed (it may be omitted if it is equal to the set of all starting states).

An internal design of the module is consistent with its interface specification if the abstract states used in both documents are the same and the abstraction function and the LD-relations are consistent with the output relation and the extension function in the sense that the following diagram commutes [7]:



In the diagram above ds_1 and ds_2 denote data structures, and T_1 and T_2 denote canonical traces.

3.2 Classification of internal designs

TAM distinguishes between single-object and multi-object modules. The rationale for this distinction is that single-object modules are used quite often, and since the identity of the only object in such a module is known, a user should not be obliged to mention explicitly the object name. This distinction is even more justified in the module internal design document. The data structure of a single-object module belongs physically to this module while the data structure of a multi-object module is usually distributed over client programs. Different forms of sharing the data structure between a module and its clients, and the moment of creation of an object give rise to a classification of module internal designs:

- *centralized internal design*; the data structure of all objects belongs to the module; all objects whose number is limited, are created by the module at the very beginning; there may exist another module providing access to object names,
- *decentralized internal design*; there exists a separate copy of the data structure for each object and this copy belongs to the client program which created a given object; the objects are created by the client when required,
- *mixed internal design*; the data structure of objects is shared between the module and its clients; the objects are created by the client when required.

An internal design of a single-object module is always centralized.

Example: (centralized design) Module implements a limited number of stack objects. In order to reduce memory requirements, stacks are kept inside a single global array.

Example: (decentralized design) Module implements stacks. A client creates a stack by declaring it inside the client program. The whole object's data structure belongs to the client program.

Example: (mixed design) Module implements objects being "technical reports". Certain reports are consulted more often, and the designer of the module decides to keep ten most accessed reports inside the module in uncompressed form. Other reports are kept in client programs in compressed form.

Note that objects with shared data structures may still be treated as independent since modifications carried on these structures are not externally observable.

There are situations when a module is allowed to use external variables. For example, a software project could use a global error reporting mechanism, or gather statistics concerning the usage of access-programs. These external variables are not used to represent abstract values and their types are not included into the domain of the abstraction function. However, they can be modified by access-programs, and their values have to be specified in descriptions of program functions, cf. Section 3.4.3.

The examples of multi-object modules designed according to a centralized approach can be found in [3]. The centralized design of a single-object module and the decentralized design of a multi-object module are presented in Appendices.

3.3 The language of implementation

At the internal design level we choose a programming language for the module implementation. The dependence of an internal design document on the implementation language is limited to the syntax and semantics of declarations and value constructors. Pascal is used as an implementation language in this document.

3.4 Structure of internal design documents

The notation used in this paper is that from FUN-INT — the syntax-driven editor for module internal design (currently under development).

In the remainder of this text an *abstract type* means a set of canonical traces, and a *concrete type* means a type defined in the implementation language. Names of concrete types are written without brackets.

An internal design document contains four sections (examples are given in Appendices). The Characteristics Section states the name of an abstract type and the category of the internal design. This section also lists foreign types used in the module including the foreign types from the module interface specification. Other sections are Data Structure Section, Abstraction Function Section, and Program Function Section.

3.4.1 Data Structure Section

The Data Structure Section describes the module's local data structures and specifies their initial values. The information is structured into subsections, most of them are optional:

- *built-in type section*; it specifies which built-in abstract types are implemented as built-in concrete types in a given implementation language,
- *constant definition section*,
- *type definition section*,
- *data declaration*; if the module design is centralized or mixed then this section describes the variables forming the centralized data structure,
- *data initial value section*; the section defines a predicate describing initial values of centralized data structure;
- *data constraint section*; the section defines the predicate "well formed data structure", *wfds*, constraining the set of values of the centralized data structure,
- *exported data type section*; if the module internal design is decentralized or mixed then this section indicates the type of decentralized data structure,
- *exported data initial values section*,
- *exported data constraint section*,
- *external data section*; the section describes external variables accessible to this module,
- *auxiliary function section*.

3.4.2 Abstraction Function Section

The Abstraction Function Section defines an abstraction function. To describe the signature of this function we denote the part of an object's data structure in a client program by *cds*, the module's data structure by *mds*, and the type of objects' names by *name*. The signature of the function, *af*, depends on the kind of internal design as follows:

- centralized design

  af: *name* × *mds* → <abstract type>

- decentralized design

  af: *cds* → <abstract type>
- mixed design

  af: *name* × *mds* × *cds* → <abstract type>

3.4.3 Program Function Section

The Program Function Section contains a set of relations, one relation for each access-program. These relations, called *program functions* or *parameterized program functions*, describe modifications of module data structure resulting from access-program invocations and the possible return values of these invocations.

Program functions (in short: *pf*) were introduced by Mills and others ([5]) and are used in many formal methods for the description of program behavior. The more general concepts of parameterized program functions (in short: *ppf*) and schemata of parameterized program functions (in short: *sppf*) are introduced in ([3]). A detailed algorithm for obtaining the domains and ranges of these three classes of functions is given there and illustrated on examples. These signatures, though mathematically sound, are difficult to handle in practice and hard to deal with for users with less mathematical background. We propose to use a simplified version. We start with the description of program functions as given in [3] and then explain our approach.

The actual domain of a *pf* is a set of tuples containing one element for every variable in the machine. In practice, however, each program uses and affects only a very small part of that data structure. Consequently, for practical purposes, we write the *pf* as if the domain contained only a relatively small number of variables, which we consider relevant, or potentially relevant, to the behavior of the specific program being described. For an invocation of a procedure without arguments, this consists of a *local data structure* used by that text. For an invocation of a procedure with arguments, the data state used in the *pf* description includes the local data structure plus additional data elements indicated by the actual arguments. We describe the program's effects by an LD-relation on this *extended data structure*. The resulting *ppf* is defined by an expression in which the formal argument names may be used as if those arguments were part of the data structure used by the program. The actual *pf* for an invocation is obtained by substituting the actual argument for the formal argument in the *ppf* description.

A *ppf* is used when the effect of the program is independent of the names of the variables. If the code is such that the identities of the actual arguments can change the behavior (e.g. if aliasing could cause changes in the effect of the program), this simple substitution scheme does not suffice and a *sppf* is used. The arguments to this *sppf* are the names of those variables whose names matter. The actual *ppf* is obtained by substituting the actual names for the formal arguments in the *sppf* description.

We propose a slightly modified approach. The main idea is to flatten this composition of functions by merging a *sppf* with a *ppf*. The resulting function will still be called a *ppf*.

The signatures of *ppf*s are not given explicitly in the Program Function Section but they may be easily derived from the syntax table which is copied to the internal design document from the interface specification (the internal design document should contain all information needed by a module implementor to write the code). For each access-program this table determines the domain and the range of its *ppf*. The formal argument list corresponding to the *ppf* domain is used by the implementor to write the access-program header in the code.

The signature of a *ppf* is constructed according to the following rules:

(1) The domain of a *ppf* consists of tuples containing for every argument, with the exception of the zeroth argument in single-object modules, a name, a value, or a pair (name, value). The exact form depends on the used descriptors, the type of the argument being domestic or foreign, and the category of the internal design (in the case of domestic arguments). This is described by the table below. If *t* is a type of the argument, then *v*

denotes a value of type *t*, and *n* denotes a value of type of names for *t*.

Descriptors	Domestic type arguments			Foreign type arguments
	Centralized (multi-object)	Mixed	Decentralized	
V	*n*	(*n, v*)	*v*	*v*
O, VO	*n*	(*n, v*)	*v*	*v*
N	*n*	*n*	*n*	*n*
NO, NV, NVO	*n*	(*n, v*)	(*n, v*)	(*n, v*)

(2) The range of a *ppf* is a set of *pf*s. Their signatures are as follows:

- the domain of a *pf* consists of tuples containing:
  - one value of *cds* for every domestic argument if the design is decentralized or mixed,
  - one value of *mds* if the design is centralized or mixed,
- the range of a *pf* consists of tuples containing:
  - one value of *cds* for every domestic argument if the design is decentralized or mixed,
  - one value of *mds* if the design is centralized or mixed,
  - one value of type *t* for every output argument of foreign type *t*,
  - one value of type *t* if the access-program is a pascal like function of type *t*.

To establish a matching between the data structure modified by an access-program invocation and the mathematical variables used to define the *pf* we introduce the concept of environment. An *environment* maps variables to types and is defined as follows:

- if *n* is an argument of the *ppf*, *n* is of type *t*, and *n* corresponds to a name of an object then $(n, t) \in env$,
- if *x* is an argument of the *ppf, x* is of foreign type *t*, and *x* corresponds to a value of an object then $(`x, t) \in env$ if the argument's descriptor contains "V", and $(x',t) \in env$ if the argument's descriptor contains "O*",*
- if *x* is an argument of the *ppf,* and *x* belongs to the client data structure of type *t*, then $(`x, t) \in env$ and $(x',t) \in env,$
- if the module data structure contains a variable *x* of type *t*, then $(`x, t) \in env$ and $(x', t) \in env,$
- if the *ppf* is defined for a pascal like function of type *t*, then $(\ , t) \in env.$

If *x* is a variable, then x primed on the left (‘*x*) denotes the value of *x* before the invocation and x primed on the right (*x*’) denotes the value of *x* after the invocation. We use the macro *NC* (not changed) to specify that *x* is not changed by the invocation: $NC(x) \stackrel{df}{=} `x = x'.$

The domain of the environment for a given *pf* contains variables which can be referenced inside this *pf*. Note that if a *ppf* parameter corresponds to a value from *cds* (*v* in columns 3 and 4 in the table above) or to a foreign type output value, then it does not belong to the domain of the environment, and hence, it cannot be referenced inside the *pf*. Its name is used only to construct the names of variables denoting the old state and the new state of the corresponding object.

It should also be noted that in the signature of a *pf* foreign type parameters are treated in a different way than domestic type parameters. This can be justified by the following arguments. The module has the knowledge of the data structure used to implement domestic objects. If a domestic type argument has the descriptor "V", than its abstract value may be used but not modified. However, many concrete values could be used to implement the same abstract value. The invocation may change the concrete value of such an argument under the condition that its abstract value is not changed. In the case of foreign type arguments the module does not know their data structures. If the descriptor is "V", then we do not need to specify the new state since it is known.

Below we illustrate the described rules by a few examples. We define a *ppf* for an access-program SET-POINT(<point>:O,<real>:V,<real>:V) from the module <point>. The symbol ↔ represents a relation.

Example: Centralized internal design. Single-object module implementing a point.

type point = record x,y: <real> end;

var origin: point;

cds ≡ ∅, *mds* ≡ point

ppf_SETPOINT: <real> × <real> → (point ↔ point)

ppf_SETPOINT(x, y) $\stackrel{df}{=}$ (origin'.x = 'x) ∧ (origin'.y = 'y) ∧ *NC*(x, y)

Example: Centralized internal design. Multi-object module implementing MAX points.

type for names: <int>

type point = record x,y: <real> end;

type all_points = array [1..MAX] of point;

var points: all_points;

cds ≡ ∅, *mds* ≡ all_points

ppf_SETPOINT: <int> × <real> × <real> → (all_points ↔ all_points)

ppf_SETPOINT(name, x, y) $\stackrel{df}{=}$ *NC*(x, y) ∧ (points'[name].x = 'x) ∧ (points'[name].y = 'y) ∧
∀i:<int> (1 ≤ i ≤ MAX) [i ≠ name ⇒ NC(points[i])]

Example: Decentralized internal design. Multi-object module implementing a point.

type point = record x,y: <real> end;

cds ≡ point, *mds* ≡ ∅

ppf_SETPOINT: point × <real> × <real> → (point ↔ point)

ppf_SETPOINT(p, x, y) $\stackrel{df}{=}$ (p'.x = 'x) ∧ (p'.y = 'y) ∧ *NC*(x, y)

3.5 Internal design of hierarchical modules

At the internal design level, hierarchical dependences between modules are introduced by use of foreign types in the internal design document. Only the interface specification of a foreign type module may be referenced. However, the complete project documentation should provide an internal design of this module.

A foreign type name may occur in two contexts:

(a) as a type of an argument of a module's access-program,

(b) in the definition of the local data structure.

The interpretation of these occurrences depends on the foreign type used and the following cases may be distinguished:

(1) Foreign type is a built-in type.

In that case, the Data Structure Section may provide a definition of this foreign type in the implementation language. In Pascal, for example, the abstract data type <int> may be defined as Integer or the subrange, 1..10. All other occurrences of the foreign type may be treated as if the type name has been replaced by this definition.

(2) Foreign type is specified as a single-object module.

In a single-object module the only object of the module is hidden inside it and may not be used as part of the local data structure of any other module. Use of such a foreign type to define the local data structure of

the domestic module is thus illegal.

(3) Foreign type is specified as a multi-object module.

Such a foreign type may be legally used in the definition of the data structure. The interpretation of this usage depends on the category of the internal design of the foreign type and will be known at the implementation level. If this internal design is decentralized, i.e., the foreign type is exported, then the domestic objects are composed hierarchically from the objects of the foreign type. If the internal design is centralized, the foreign type is not exported, and the actual type used in the definition is the type of names for the foreign type objects.

The implementation language usually imposes restrictions on the use of types in certain contexts. For example, an index type in arrays has to be an ordinal one. This eliminates all not built-in types.

The following example demonstrates the usage of foreign types in definitions of local data structures. We assume that <point>, <vector>, and <halfline> are multi-object modules.

Example:

type vector = record p, q: <point> end;

type halfline = record p: <point>; v:<vector> end;

wfds: halfline → <bool>

wfds(hl) $\stackrel{df}{=}$ (hl.p = *vector::getstart*(hl.v)) ∧ *(vector::getlength*(hl.v) = 1)

type angle = record l, k: <halfline> end;

wfds: angle → <bool>

wfds(a) $\stackrel{df}{=}$ *halfline::getpoint*(a.l) = *halfline::getpoint*(a.k)

4 From internal design to implementation

During the next step of software development the module internal design is transformed into the code. A well-structured code should be composed of two parts:

(1) a public part - this part is visible to the module's clients.

(2) a private part - this part contains secrets of the implementation and is not visible outside the module.

Unfortunately, not all programming languages support this approach and it is a responsibility of the designer/programmer to structure the code in a way as much as possible reflecting the paradigm of the "black-box" design applied at the specification level. It might be a good idea to accompany the code with a document (e.g. "User Guide") describing the proper usage of the module.

Generally, rules of transforming the module internal design into the code are straightforward:

(1) Each access-program is implemented as a separate routine.

(2) Each access-program argument has its counterpart at the implementation level (with the additional argument for the returned value). An argument with the descriptor N may be replaced by one argument (object's name) or two arguments (object's name and value) depending on the implementation language and the remaining descriptors (see [3] for examples).

(3) Local definitions and declarations should appear in the private part of the code.

(4) Exported data type should be included in the public part (with its definition given in the private part). However, not all programming languages support transparent export of the data type.

If a module design is decentralized or mixed then the module is responsible for delivering an operation for "creating" objects of the implemented data type in a proper initial state. The module interface specification determines the

initial state of every object but it cannot specify the "creation". In the simplest case this operation is equivalent to the declaration of the object accompanied by its initialization. However, in the programming languages like Pascal these two actions are separate and the programmer should not only deliver the appropriate initialization routine but also make a user aware of its existence and responsible for its invocation — the User Guide should contain necessary information. In languages like C++ object constructors solve the initialization problem.

If a module design is centralized or mixed then the module should also provide an operation for initializing its centralized data structure.

In the case of a centralized multi-object module the usage of names in this module can be controlled by another module. A user of the multi-object module should first acquire such a name and then use it to access the object. The implementation should support this scenario (examples may be found in [3]).

5 Conclusions

We are convinced that at every stage of the software development software engineers should be equipped in proper documentation techniques and supported by appropriate tools. The techniques should be easy to apply, precise, and intuitively appealing. They should be acceptable for engineers without a profound mathematical background. They should also be able to express hierarchical dependences between software components. In our opinion the specification formalisms presented in this paper fulfill these requirements.

The integrated set of tools for software documentation is currently under development. The FUN-SPEC editor supports syntax-driven editing of interface specifications. This editor also produces a preliminary version of the internal design document. This version serves as a starting point for the internal design process, supported by the separate syntax-driven editor, FUN-INT. We plan to integrate FUN-INT with the software supporting the Display Method ([8]). The Display Method helps in documenting a single program. A prototype version of such a tool has been developed.

Our plans for the near future aim at further development of the described tools and their integration, including a mechanism that will help to debug specifications.

Acknowledgements

Krzysztof Stencel read an early version of the paper and gave us many valuable comments.

This work was partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the State Committee for Scientific Research in Poland (KBN, grant 8 S503 040 04), and by the NATO Linkage grant (HTECH. LG. 941314).

References

1. Bartussek, W., Parnas, D.L., "Using Traces to Write Abstract Specifications for Software Modules", in: *Proceedings of 2nd Conference of European Cooperation in Informatics*, Lecture Notes on Computer Science, 65. Springer-Verlag, Venice, 1978.
2. Iglewski, M., Kubica, M., Madey, J., "An Editor for the Trace Assertion Method", in: *Proceedings of the 10th ISPE/IFAC International Conference on CAD/CAM, Robotics and Factories of the Future*, August 1994, Ottawa, Canada, pp. 876–881.
3. Iglewski, M., Madey, J., Parnas, D.L., Kelly, P.C., "Documentation Paradigms", *CRL Report* No. 270, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.
4. Iglewski, M., Madey, J., Stencel, K., "On Fundamentals of the Trace Assertion Method", *Technical Report* RR 94/09-6, University of Quebec at Hull, Hull, Quebec, Canada, 1994.

5. Mills, H.D., “The New Math of Computer Programming”, *CACM*, Vol. 18, No. 1, January 1975, pp. 43–48.
6. Parnas, D.L, “On the Criteria To Be Used in Decomposing Systems Into Modules”, *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053–1058.
7. Parnas, D.L., Madey, J., “Functional Documentation for Computer Systems Engineering. (Version 2)”, *CRL Report* No. 237, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1991; to appear in: *Science of Computer Programming.*
8. Parnas, D.L., Madey, J., Iglewski, M., “Precise Documentation of Well-Structured Programs”, *IEEE Trans. on Software Eng.*, Vol. 20, No. 12, 1994, pp. 948–976.
9. Parnas, D.L., Wang, Y., “The Trace Assertion Method of Module Interface Specification”, *Technical Report* 89-261, Queen’s University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1989.
10. Wang, Y., “Specifying and Simulating the Externally Observable Behavior of Modules”, (Ph.D. Thesis)*, CRL Report* No. 292, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

Appendix A

Point Module Interface Specification

(0) CHARACTERISTICS

• type specified: <point>

• features: multi-object, deterministic

• foreign types: <bool>, <real>

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Arg#3	Value Type
SETPOINT	<point>:O	<real>:V	<real>:V	
EQUAL	<point>:V	<point>:V		<bool>
GETX	<point>:V			<real>
GETY	<point>:V			<real>

(2) CANONICAL TRACES

canonical(T) $\Leftrightarrow \exists$ x, y:<real> [T = SETPOINT(*, x, y)]

AUXILIARY FUNCTIONS

getx: <point> $\rightarrow$ <real>

getx(p) $\stackrel{df}{\equiv}$ x where x,y: <real> (p = SETPOINT(*, x, y))

gety: <point> $\rightarrow$ <real>

gety(p) $\stackrel{df}{\equiv}$ y where x,y: <real> (p = SETPOINT(*, x, y))

(3) EQUIVALENCES

_ ≡> SETPOINT(*, 0, 0)

Legality(SETPOINT(*, x, y)) = %legal%

SETPOINT(* , x, y) ≡> SETPOINT(*, x, y)

Legality(EQUAL(T, U)) = %legal%

Legality(GETX(T)) = %legal%

Legality(GETY(T)) = %legal%

(4) RETURN VALUES

Invocation	Value
EQUAL(T,U)	T = U
GETX(T)	*getx*(T)
GETY(T)	*gety*(T)

point Internal Design Document

(0) CHARACTERISTICS

- abstract type: <point>
- features: multi-object, decentralized
- foreign types: <bool>, <real>

(1) DATA STRUCTURE

BUILT-IN TYPES

<bool> $\stackrel{df}{=}$ boolean

<real> $\stackrel{df}{=}$ real

TYPE DEFINITIONS

```
type point = record
        x, y: <real>
    end;
```

EXPORTED DATA STRUCTURE

type point;

EXPORTED DATA STRUCTURE INITIAL VALUES

initial_value: point $\rightarrow$ boolean

initial_value(p) $\stackrel{df}{=}$ p.x = 0 $\wedge$ p.y = 0

EXPORTED DATA STRUCTURE CONSTRAINTS

wfds: point $\rightarrow$ boolean

wfds(p) $\stackrel{df}{=}$ true

(2) ABSTRACTION FUNCTION

af : point → <point>

af(p) $\stackrel{df}{=}$ SETPOINT(*, p.x, p.y)

(3) PROGRAM FUNCTIONS

Program Name	Arg#1	Arg#2	Arg#3	Value Type
SETPOINT	<point>:O	<real>:V	<real>:V	
EQUAL	<point>:V	<point>:V		<bool>
GETX	<point>:V			<real>
GETY	<point>:V			<real>

ppf_SETPOINT(p, x, y) $\stackrel{df}{=}$ (p'.x = 'x) ∧ (p'.y = 'y)

ppf_EQUAL(p, q) $\stackrel{df}{=}$ (= (('p.x = 'q.x) ∧ ('p.y = 'q.y))) ∧ *NC*(p, q)

ppf_GETX(p) $\stackrel{df}{=}$ (= 'p.x) ∧ *NC*(p)

ppf_GETY(p) $\stackrel{df}{=}$ (= 'p.y) ∧ *NC*(p)

Pascal code of the point module

```
unit points;

interface
type point = record
          x, y: real
       end;

procedure point_init (var p: point);
procedure point_SETPOINT (var p: point; x, y: real);
function point_EQUAL (p: point; q: point) : boolean;
function point_GETX (p: point) : real;
function point_GETY (p: point) : real;

implementation
procedure point_init (var p: point);
begin
  p.x := 0; p.y := 0;
end;

procedure point_SETPOINT (var p: point; x, y: real);
begin
  p.x := x; p.y := y;
end;
```

```
function point_EQUAL (p: point; q: point) : boolean;
begin
  point_EQUAL := (p.x = q.x) and (p.y = q.y);
end;

function point_GETX (p: point) : real;
begin
  point_GETX := p.x;
end;

function point_GETY (p: point) : real;
begin
  point_GETY := p.y;
end;
end.
```

Appendix B

vector Module Interface Specification

(0) CHARACTERISTICS

• type specified: <vector>

• features: multi-object, deterministic

• foreign types: <point>, <bool>, <real>

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Arg#3	Value Type
SETVECTOR	<vector>:O	<point>:V	<point>:V	
EQUAL	<vector>:V	<vector>:V		<bool>
GETDX	<vector>:V			<real>
GETDY	<vector>:V			<real>
GETSTART	<vector>:V			<point>
GETEND	<vector>:V			<point>
MOVEP	<point>:VO	<vector>:V		
MOVEV	<vector>:VO	<vector>:V		

(2) CANONICAL TRACES

canonical(T) $\Leftrightarrow$ $\exists$ p, q:<point> [T = SETVECTOR(*, p, q)]

AUXILIARY FUNCTIONS

equal: <vector> × <vector> → <bool>

equal(v, w) $\stackrel{df}{\equiv}$ *getstart*(v) = *getstart*(w) ∧ *getend*(v) = *getend*(w)

getdx: <vector> → <real>

getdx(v) $\stackrel{df}{\equiv}$ (*point::getx*(q) - *point::getx*(p)) where p,q: <point> (v = SETVECTOR(*, p, q))

getdy: <vector> → <real>

getdy(v) $\stackrel{df}{\equiv}$ (*point::gety*(q) - *point::gety*(p)) where p,q: <point> (v = SETVECTOR(*, p, q))

getstart: <vector> → <point>

getstart(v) $\stackrel{df}{\equiv}$ p where p,q: <point> (v = SETVECTOR(*, p, q))

getend: <vector> → <point>

getend(v) $\stackrel{df}{\equiv}$ q where p,q: <point> (v = SETVECTOR(*, p, q))

movep: <point> × <vector> → <point>

movep(p, v) $\stackrel{df}{\equiv}$ *point::*SETPOINT(*, newx, newy)
where x,y,newx,newy:<real> (p = *point::*SETPOINT(*, x, y)) ∧ (newx = x+*getdx*(v)) ∧ (newy = y+*getdy*(v))

movev: <vector> × <vector> → <vector>

movev(v, w) $\stackrel{df}{\equiv}$ SETVECTOR(*, newp, newq)
where p,q,newp,newq:<point> (v = SETVECTOR(*, p, q)) ∧ (newp = *movep*(p, w)) ∧ (newq = *movep*(q, w))

(3) EQUIVALENCES

_ ≡> SETVECTOR(*, *point::_*, *point::_*)

Legality(SETVECTOR(*, p, q)) = %legal%

SETVECTOR(* , p, q) ≡> SETVECTOR(*, p, q)

Legality(EQUAL(*, v)) = %legal%

Legality(GETDX(*)) = %legal%

Legality(GETDY(*)) = %legal%

Legality(GETSTART(*)) = %legal%

Legality(GETEND(*)) = %legal%

Legality(MOVEP(*, v)) = %legal%

MOVEP(T , v) ≡> *movep* (T, v)

Legality(MOVEV(*, v)) = %legal%

MOVEV(T , v) ≡> *movev* (T, v)

(4) RETURN VALUES

Invocation	Value
EQUAL(T,U)	T = U
GETDX(T)	*getdx*(T)
GETDY(T)	*getdy*(T)
GETSTART(T)	*getstart*(T)
GETEND(T)	*getend*(T)

vector Internal Design Document

(0) CHARACTERISTICS

- abstract type: <vector>
- features: multi-object, decentralized
- foreign types: <point>, <real>, <bool>

(1) DATA STRUCTURE

BUILT-IN TYPES

<bool> $\stackrel{df}{=}$ boolean

<real> $\stackrel{df}{=}$ real

TYPE DEFINITIONS

```
type vector = record
        p, q: <point>
    end;
```

EXPORTED DATA STRUCTURE

```
type vector;
```

EXPORTED DATA STRUCTURE INITIAL VALUES

initial_value: vector→ boolean

initial_value(v) $\stackrel{df}{=}$ (v.p = *point::_*)∧ (v.q = *point::_*)

EXPORTED DATA STRUCTURE CONSTRAINTS

wfds: vector → boolean

wfds(v) $\stackrel{df}{=}$ true

AUXILIARY FUNCTIONS

movep: <point> × vector → <point>

movep(p, v) $\stackrel{df}{=}$ *point::*SETPOINT(*, newx, newy) where x,y,newx,newy:<real> (p = *point::*SETPOINT(*, x, y)) ∧
(newx = x+*point::getx*(v.q) - *point::getx*(v.p)) ∧ (newy = y+*point::gety*(v.q) - *point::gety*(v.p))

(2) ABSTRACTION FUNCTION

af : vector → <vector>

af(v) $\stackrel{df}{=}$ SETVECTOR(*, v.p, v.q)

(3) PROGRAM FUNCTIONS

Program Name	Arg#1	Arg#2	Arg#3	Value Type
SETVECTOR	<vector>:O	<point>:V	<point>:V	
EQUAL	<vector>:V	<vector>:V		<bool>
GETDX	<vector>:V			<real>
GETDY	<vector>:V			<real>
GETSTART	<vector>:V			<point>
GETEND	<vector>:V			<point>
MOVEP	<point>:VO	<vector>:V		
MOVEV	<vector>:VO	<vector>:V		

ppf_SETVECTOR(v, p, q) $\stackrel{df}{=}$ (v'.p = 'p) ∧ (v'.q = 'q)

ppf_EQUAL(v, w) $\stackrel{df}{=}$ (= ('v.p = 'w.p) ∧ ('v.q = 'w.q)) ∧ *NC*(v, w)

ppf_GETDX(v) $\stackrel{df}{=}$ (= *point::getx*('v.q) - *point::getx*('v.p)) ∧ *NC*(v)

ppf_GETDY(v) $\stackrel{df}{=}$ (= *point::gety*('v.q) - *point::gety*('v.p)) ∧ *NC*(v)

ppf_GETSTART(v) $\stackrel{df}{=}$ (= 'v.p) ∧ *NC*(v)

ppf_GETEND(v) $\stackrel{df}{=}$ (= 'v.q) ∧ *NC*(v)

ppf_MOVEP(p, v) $\stackrel{df}{=}$ (p' = *movep*('p, 'v)) ∧ *NC*(v)

ppf_MOVEV(v, w) $\stackrel{df}{=}$ (v'.p = *movep*('v.p, 'w)) ∧ (v'.q = *movep*('v.q, 'w)) ∧ *NC*(w)

Pascal code of the vector module

```
unit vectors;
uses points;

interface
type vector = record
        p, q: point
     end;
```

```
procedure vector_init (var v: vector);
procedure vector_SETVECTOR (var v: vector; p, q: point);
function vector_EQUAL (v: vector; w: vector) : boolean;
function vector_GETDX (v: vector) : real;
function vector_GETDY (v: vector) : real;
procedure  vector_GETSTART (v: vector; var p: point);
procedure  vector_GETEND (v: vector; var p: point);
procedure vector_MOVEP (var p: point; v: vector);
procedure vector_MOVEV (var v: vector; w: vector);

implementation
procedure vector_init (var v: vector);
begin
  point_init(v.p); point_init(v.q);
end;

procedure vector_SETVECTOR (var v: vector; p, q: point);
  var x, y: real;
begin
  x := point_GETX(p); y := point_GETY(p);
  point_SETPOINT(v.p, x, y);
  x := point_GETX(q); y := point_GETY(q);
  point_SETPOINT(v.q, x, y);
end;

function vector_EQUAL (v: vector; w: vector) : boolean;
begin
  vector_EQUAL := point_EQUAL(v.p, w.p) and point_EQUAL(v.q, w.q));
end;

function vector_GETDX (v: vector) : real;
begin
  vector_GETDX := point_GETX(v.q) - point_GETX(v.p);
end;

function vector_GETDY (v: vector) : real;
begin
  vector_GETDY := point_GETY(v.q) - point_GETY(v.p);
end;

procedure  vector_GETSTART (v: vector; var p: point);
begin
  p := v.p;
end;

procedure vector_GETEND (v: vector; var q: point);
begin
```

```
  q := v.q;
end;

procedure vector_MOVEP (var p: point; v: vector);
  var x, y: real;
begin
  x := point_GETX(p) + vector_GETDX(v);
  y := point_GETY(p) + vector_GETDY(v);
  point_SETPOINT(p, x, y);
end;

procedure vector_MOVEV (var v: vector; w: vector);
begin
  v.p := vector_MOVEP(v.p, w);
  v.q := vector_MOVEP(v.q, w);
end;
end.
```

Appendix C

Point-2 Module Interface Specification

(0) CHARACTERISTICS

• type specified: <point>

• features: single-object, deterministic

• foreign types: <real>

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#0	Arg#1	Arg#2
SETPOS	<point>:O	<real>:V	<real>:V
GETPOS	<point>:V	<real>:O	<real>:O

(2) CANONICAL TRACES

canonical(T) ⇔ ∃ x, y:<real> [T = SETPOS(*, x, y)]

(3) EQUIVALENCES

_ ≡> SETPOS(*, 0, 0)

Legality(SETPOS(*, x, y)) = %legal%

SETPOS(* , x, y) ≡> SETPOS(*, x, y)

Legality(GETPOS(*, x, y)) = %legal%

(4) RETURN VALUES

Invocation	Value
GETPOS(T, * , *)	x where T=SETPOS(*, x, y)
GETPOS(T, *, *)	y where T=SETPOS(*, x, y)

point-2 Internal Design Document

(0) CHARACTERISTICS

- abstract type: <point>
- features: single-object, centralized
- foreign types: <real>

(1) DATA STRUCTURE

BUILT-IN TYPES

<real> $\stackrel{df}{=}$ real

TYPE DEFINITIONS

```
type point= record
        x, y: <real>
    end;
```

VARIABLE DECLARATIONS

```
var origin: point;
```

VARIABLE INITIAL VALUES

initial_value: point $\rightarrow$ boolean

initial_value(p) $\stackrel{df}{=}$ p.x = 0 $\wedge$ p.y = 0

VARIABLE CONSTRAINTS

wfds: point $\rightarrow$ boolean

wfds(p) $\stackrel{df}{=}$ true

(2) ABSTRACTION FUNCTION

af : point $\rightarrow$ <point>

af(p) $\stackrel{df}{=}$ SETPOS(*, p.x, p.y)

(3) PROGRAM FUNCTIONS

Program Name	Arg#0	Arg#1	Arg#2
SETPOS	<point>:O	<real>:V	<real>:V
GETPOS	<point>:V	<real>:O	<real>:O

ppf_SETPOS(x, y) $\stackrel{df}{=}$ (origin'.x = 'x) $\wedge$ (origin'.y = 'y)

ppf_GETPOS(x, y) $\stackrel{df}{=}$ (x' = 'origin.x) $\wedge$ (y' = 'origin.y) $\wedge$ *NC*(origin)

Pascal code of the point-2 module

```
unit points;

interface
procedure point_init ();
procedure point_SETPOS (x, y: real);
procedure point_GETPOS (var x, y: real);

implementation
type point = record
                x, y: real
             end;
var origin: point;

procedure point_init ();
begin
  origin.x := 0; origin.y := 0;
end;

procedure point_SETPOS (x, y: real);
begin
  origin.x := x; origin.y := y;
end;

procedure point_GETPOS (var x, y: real);
begin
  x := origin.x; y := origin.y;
end;
end.
```